Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelor's Thesis

Performance study on GPU offloading techniques using the Gauß matrix inverse algorithm

vorgelegt von

Yannik Könneker

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	7159149
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Georgiana Mania
Betreuer:	Georgiana Mania

Hamburg, 28.01.2022

Abstract

Inverting matrices is a crucial part in many algorithms in linear algebra, computer graphics and data analysis. There are many libraries providing algorithms to achieve this but none that allow for calling from the GPU context. GPUs and accelerators become more and more prevalent in high performance computers. Having no ready-to-use implementation scientists need to write their own algorithms. In this thesis the Gauß-Elimination algorithm is implemented using OpenMP, OpenACC, CUDA, HIP and OpenCL. These implementations are then compared to the already existing libraries Eigen and cuBLAS in terms of speed, precision and implementation effort.

Contents

1 Introduction

Inverting matrices is a crucial part in many algorithms spanning from linear algebra [Dem97] over 3D graphics rendering [SP08], image processing [LSFL07] and data analysis [Web21]. Fast solving algorithms are needed for ever growing data sizes. There are already many efficient implementations of algorithms designed to solve large matrices quickly e.g. Eigen-Library [Eig].
But CPUs are not getting much faster as they were just one or two decades ago. As Moore's law predicts, the amount of transistors on a microprocessor doubled every two years. The single thread performance however started to lose its momentum in the mid 2000's as seen in Figure 1.1. At the same time the amount of logical cores in a

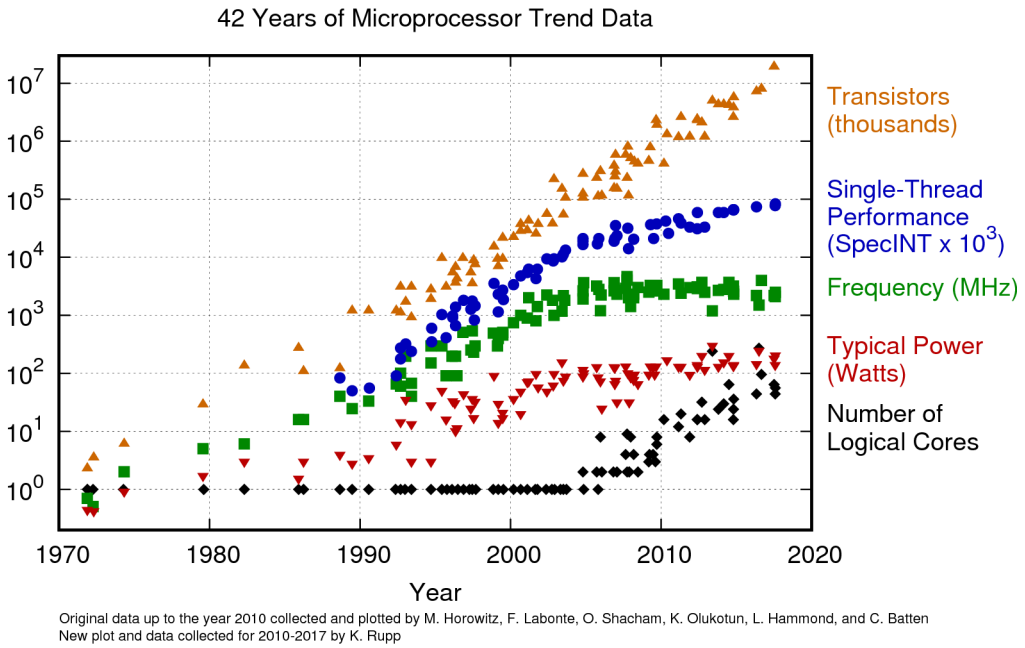


Figure 1.1: Visualization of microprocessor trend data commonly referred to as "Moores Law"
[Rup]

CPU has drastically increased. Modern CPUs offer up to 64 cores (128 threads with hyperthreading) [WLHL+21] allowing for a good deal of parallelism. Due to their unique architecture, modern GPUs can offer up to 8192 cores [Corc], granting access to massive parallelization. This shows that it is a mistake to rely on advancements in the CPU production to accelerate more complex and precise algorithms. To access the growing supply of GPU power, less demanding per core demanding and more parallelized code

should be written for heterogeneous systems.
The website Top500.org collects statistics about supercomputers around the world, to give an overview of the manufacturers, locations, performance and many more characteristics. Figure 1.2 shows that an increasing number of supercomputers use accelerators, such as GPUs. Because of this development more and more code will be written for

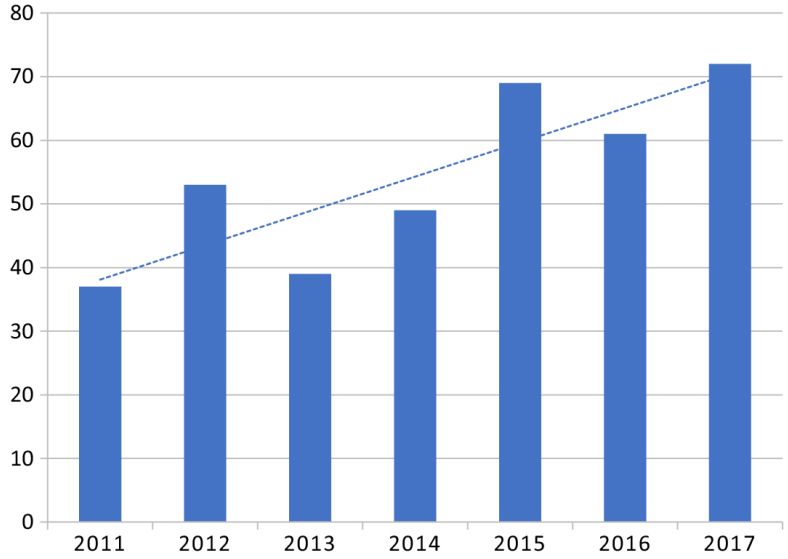


Figure 1.2: Trend of heterogeneous systems in the Top500 [Kha19]

accelerators, often by scientists in their respective domains and not by computer experts. Typically, this results in poor performance utilization of the used hardware. Up until 2013 writing code for heterogeneous systems (CPU with additional accelerators) was only possible through API's such as CUDA [Corb] and OpenCL [Gro]. Both however are not the easiest to learn and use, because they require a lot of hardware specific concepts and constructs, as well as the C programming language to understand. Since 2013 multiple groups released more high level API's like OpenMP [Boa] or OpenACC [Urb21a].

Another motivation for this thesis was given by the DESY in Hamburg. Matrix inversion is used in their Kalman Filter algorithm [Mur12] to identify accurate particle tracks in high-energy physics. This is being done by combining the actual measurements from a detector, like ATLAS at CERN, with the trajectory estimates, which are obtained by integrating the equation of motion twice [AAC+]. This works fine when using Eigen library on the CPU, but as they started porting the code over to CUDA neither Eigen nor cuBLAS support calling the matrix inversion algorithms from the accelerator [AMG+21].

This thesis will look at the Gauß-Elimination-Method as a basis for a parallel matrix inversion implementation and compare different offloading techniques.

In this thesis I will begin with introducing some related papers that build the ba-

sis of this work. Following up the fundamentals of inverse matrices, GPUs and the used offloading techniques are discussed. Chapter 4 is about the specific compilers that were used, as well as the implementations of different offloading techniques. In chapter 5, the setup alongside the results of the benchmark is discussed. In chapter 6 everything regarding the reproducibility of this work is collected together for easy access. In Chapter 7 possible future works are discussed that could follow this thesis.

2 Related Work

In this chapter, I will highlight related research.

Girish Sharma et al. [SAB13] propose a CUDA implementation for the Gauß elimination algorithm. The paper focuses on the runtime bounds. Sequentially this algorithm has a runtime of $\mathcal{O}(n^3)$ but it's shown that when using GPUs, the runtime bound is only $\mathcal{O}(n)$, or linear, as long as n^2 is less than the amount of cores the GPU has. The algorithm is the basis for this thesis.

Suejb Memeti et al. [MLP+17] compare a variety of benchmarks ranging from fluid dynamics over medical imaging to data mining with different offloading techniques like CUDA, OpenCL, OpenMP and OpenACC. They use two different systems, one with an NVIDIA GPU and one with a Xeon Phi accelerator. The metrics used are "programming productivity", "performance" and "energy consumption". "Programming productivity" is measured in lines of code written. The results are mixed but I will present the most important observations:

1. OpenCL takes on average two times as many lines as CUDA and a lot more than OpenMP and OpenACC.
2. OpenMP and OpenACC are similar in code lines written.
3. The human factor can significantly impact the lines required and the performance of the code. For example the OpenCL implementation of the same benchmark on different systems was written by two different engineers and varied n the amount of code by 2%.
4. OpenMP implementations, in some cases, performed significantly worse than the OpenCL and CUDA implementations.
5. Overall CUDA and OpenCL were relatively similar in their performance and the same holds for OpenMP and OpenACC.

Tiago Gimenes et al. [LGPB18] evaluate the performance and cost of acceleration in seismic processing. They compare CUDA, OpenCL, OpenMP and OpenACC on six different CPU models and four different GPU models. This paper shows that OpenACC is on average 9% slower than both CUDA and OpenCL as well as 5% slower than OpenMP. Another interesting finding is, that the OpenCL implementation outperforms OpenMP and OpenACC on the CPU by a factor of 2 to 4. The reason given for this is that no vectorization is used for OpenMP and OpenACC. Performance is similar when the OpenCL code is not vectorized.

Xuechao Li et al. [LSO+16] study the programmer productivity empirically, by giving 28 undergraduate and graduate students two algorithms to parallelize with OpenACC and CUDA. They conclude that offloading with CUDA takes about 50% longer than with OpenACC while yielding a 9x performance uplift. The authors state that their sample size of 28 was very small and a bigger emphasis needs to be put on participants previous programming experience.

3 Fundamentals

In this chapter, I will discuss the unique architecture of the GPU, the algorithm implemented and the offloading techniques used.

3.1 Inverse Matrix

The inverse matrix is a square matrix that multiplied with the initial matrix, equals to the identity matrix. A square matrix has two dimensions and equally many rows or columns per dimension. The identity matrix is also a square matrix with ones on the main diagonal, and zeroes everywhere else. The identity matrix is denoted by I. Not all square matrices are invertible. Such matrices are called singular matrices and have a determinant of zero which can be calculated using e.g. LU decomposition. These matrices are of no relevance to this thesis.

3.2 GPU

"The [...] GPU (Graphical Processing Unit) provides much higher instruction throughput and memory bandwidth than the CPU [...]. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU [...]. This difference in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput). The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control." [Corb] At first, most applications that needed highly parallel computations were ones that calculated graphics of some kind, because when working with graphics there are many computations needed for every pixel that can be fully parallelized.
Figure 3.1 shows an example distribution of chip resources for a CPU versus a GPU.
Table 3.1 shows the vocabulary used by the different GPU vendors. I will use the terminology given by NVIDIA to explain how GPUs work. The GPU consists of multiple streaming multiprocessors (SM) and global memory. Each streaming multiprocessor is independent from the others and has its own shared memory, instruction unit and $n \cdot 32$ streaming processors (SP) ($n \geq 1$). Each SP has its own local memory often implemented with registers. Figure 3.1 illustrates the differences between CPUs and GPUs. On the

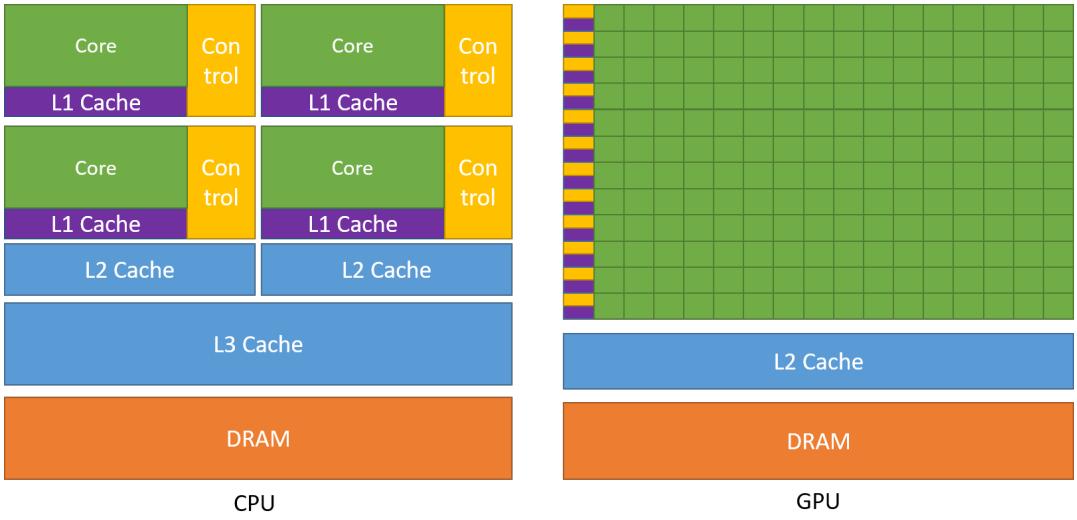


Figure 3.1: Abstraction of CPU and GPU layout.
[Corb]

right side of the graphic, each row represents a streaming multiprocessor.
Table 3.2 shows the different directives, keywords and methods to target the GPU architecture.

3.3 Libraries and APIs

3.3.1 Eigen

Eigen is an open-source header-only C++ library for linear algebra, matrix/vector operations et cetera. It is very fast because it is built with vectorization wherever possible and C++ expression template meta programming to evaluate expressions only when needed (lazy evaluation). There are also bindings for a lot of other programming languages like Haskell and Python. Using Eigen is very simple as all algorithms are implemented for arbitrary matrix-sizes and all numerical types in the standard library. It's even possible to use Eigen-matrices as a virtual overlay over an existing C-array. Eigen was initially released in 2006, with the newest stable release being 3.4 from August 2021. [Eig]

3.3.2 OpenMP

OpenMP (Open Multi-Processing) is an API that supports shared-memory multiprocessing programming in C, C++ and Fortran. There also are multiple spin-offs for other programming languages like Java [UoA]. OpenMP runs on many platforms, instruction-set architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that enable parallel execution [vdPST17]. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface

NVIDIA/CUDA	AMD	Intel	Description
Streaming Multiprocessor (SM)	Compute Unit (CU)	SubSlice (SS)	One of the independent parallel vector processors in a GPU that contain multiple SIMD ALUs.
Kernel	Kernel	Kernel	Function launched to the GPU that will be executed by multiple parallel workers.
Warp	Wavefront	Vector Thread	Operations that execute in lockstep, run the same instructions and follow the same control-flow.
Thread block	Workgroup	Workgroup	Group of warps/wavefronts/vector threads.
Thread	Work item/Thread	Work item/Vector lane	Individual lane in a warp/wavefront/vector thread.
Global Memory	Global Memory	GPU Memory	DRAM memory of the GPU.
Shared Memory	Local Memory	Shared local memory	Scratchpad memory for communication between warps/wavefronts/vector threads within a threadblock/workgroup
Local Memory	Private Memory	GPRF	Per-thread private memory, often mapped to registers.

Table 3.1: Terminology table for NVIDIA, AMD and Intel [Urb21a]

GPU	NVIDIA	OpenCL	OpenMP	OpenACC
GPU	Kernel	NDRange	target	kernels/parallel
Thread block-/SM	implicitly through kernel	implicitly through NDRange	teams distribute	gang
Warp	implicitly through kernel	implicitly through NDRange	/	worker
Thread	implicitly through kernel	implicitly through NDRange	parallel for	vector
Global Memory	CUDAMemcpy	clEnqueue-WriteBuffer / -ReadBuffer	target data	data
Shared Memory	__ shared__	__ local	shared(x)	shared(x)
Local Memory	variable declared in kernel	variable declared in kernel	private(x)	private(x)

Table 3.2: Terminology table for targeting the GPU's architecture. [Urb21b]

for developing parallel applications for platforms ranging from desktop computers at home to high performance computers.

In 2013 OpenMP released their specification for version 4.0 and with such the ability to offload code to hardware accelerators. OpenMP now enables compiling specific sections of the code to be executed on hardware accelerators like GPUs with the known ease of use that OpenMP provides. They added an abstraction layer for the hardware to make code compile and run on theoretically every accelerator.

OpenMP implements the fork-join model. In this model the master thread executes code in sequence from start to finish. When OpenMP reaches directives that define parallel regions the main thread creates new threads to split the work as seen in Figure 3.2. This is called the fork. Once the work of a given thread is done it reaches a barrier, forcing it to synchronize with all other threads after which the threads are terminated. This is called the join.

OpenMP uses pragma directives to parallelize code. The most simple directive to parallelize a for loop would be `#pragma omp parallel for`. To target an accelerator the directive would be `#pragma omp target parallel for`. The for-loop following the pragma will automatically be offloaded to the GPU. Because OpenMP can't probe the specific accelerator architecture like how many streaming processors are in an streaming multiprocessor, simply writing `#pragma omp target parallel for` wouldn't utilize the accelerators hardware, as OpenMP defaults to create as many threads as there are

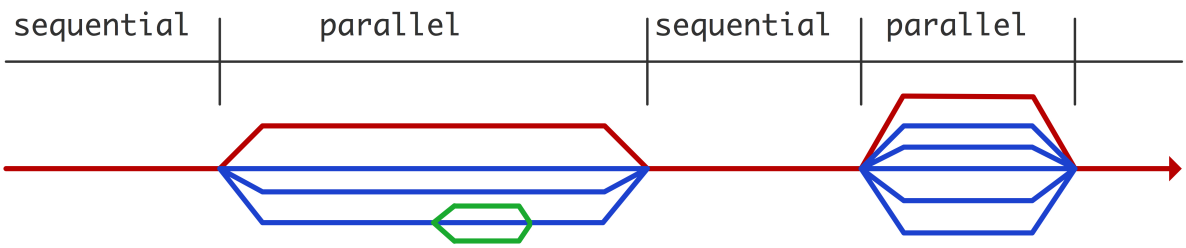


Figure 3.2: OpenMP: fork-join model
[EvdGC16]

CPU cores and assign these threads to a single SM. To enable full GPU utilization, OpenMP introduces two keywords to target accelerators which almost always come in a pair: `teams distribute`. The directive `teams` creates a league of threads which are independent of each other. Each team is assigned to a different SM and has to calculate a part of the for loop. `distribute` distributes the iterations of each independent thread across the streaming processors of the streaming multiprocesor.
OpenMP can move data implicitly and explicitly; implicit data movement happens, when the user has not given instructions for specific data structures and only if the data is allocated on the stack. Single value variables and more complex C++ data structures can be copied over implicitly when calling a kernel. C arrays however need to be explicitly copied, because OpenMP can't read the array's size and would just copy the pointer and the single pointed to value. The same is true for data allocated on the heap. Being able to optimize data transfers, by e.g. reducing them, is another advantage of explicit data movement. There are two ways to move data explicitly between host and device: first, using `#pragma omp target data map(tofrom: arr[0:size])` moves the array `arr` to the device at the beginning of the following scope. At the end of the scope, the data will be copied back to the host. `tofrom` can be replaced by `to`, `from` or `alloc` where `to` only copies to the device, `from` only creates the array on the device and copies the data back at the end of scope and `alloc` only allocates the data structure on the device without copying anything. The other way of managing data on the device is independent of scopes. `#pragma omp target enter data map(to: arr[0:size])` copies data to the device, `alloc` could be inserted as well. `#pragma omp target update data from(arr[0:size])` copies given data back from the device to the host, where all keywords apply. Lastly `#pragma omp target exit data map(from: arr[0:size])` copies back the data and deletes the array on the device, where only `from` is usable. [Boa]

3.3.3 OpenACC

OpenACC (Open Accelerators) is very similar to OpenMP in regards to the use of compiler directives to parallelize or offload code. Although OpenACC is more focused on accelerators, it supports CPU level parallelization too. OpenACC was founded in 2011 with a focus on NVIDIA GPUs and supports many accelerators now. OpenACC is

highly interoperable with CUDA in contrast to OpenMP.

The directive `#pragma acc kernels` followed by braces tells the compiler to try and find out what can be parallelized in the following code. This can work very well but sometimes the compiler sees data dependencies that are not relevant for the algorithm and because of this, doesn't parallelize the code. In these cases the directive `#pragma acc parallel loop` is needed to tell the compiler explicitly that there are no data dependencies between iterations. With OpenACC it's possible to specifically target the architecture of a GPU. E.g. `#pragma acc parallel loop gang worker vector` tells the compiler that the following loop should be parallelized,. `gang` explicitly shares this parallelization across multiple streaming multiprocessors, `worker` across multiple warps and `vector` across multiple threads and, if vectorization is possible, also vectorizes the code.[Urb21b]

3.3.4 CUDA

CUDA (Compute Unified Device Architecture) is a low-level parallel computing platform and API that allows for general purpose programming of GPUs from NVIDIA. This approach is called GPGPU (general purpose computing on GPUs). In 2006 NVIDIA released the first version of CUDA, which is also the first API to enable GPGPU ever. CUDA's newest release is 11.5 in October 2021. CUDA can be used in many programming languages like C++, Haskell, Java, Python and Fortran through either direct language implementation or libraries. There are also many libraries written in the CUDA package like cuBLAS, MAGMA and TensorRT.

In CUDA, work is split into threads, threads are grouped into three-dimensional blocks and blocks are grouped into three dimensional grids. Grids are created when invoking a kernel. Kernels are GPU functions that can be called from the CPU and with newer CUDA versions also from the GPU. Kernels cannot return rvalues.
Programming for CUDA requires algorithms to be rewritten almost entirely, because all independent loops from the sequential code will be executed in parallel. To achieve this, the code executed in a loop iteration will be the kernel program code. Each thread will be assigned an ID and a block-ID for three dimensions. Given these IDs, it's possible to calculate the iteration step that this thread should work on by for instance using `threadIdx.x` for the first loop, `threadIdx.y` for the second and so on. `threadIdx` is populated by the CUDA runtime API. If there are more than three nested but independent loops, the loops should be flattened to three or fewer nested loops. This is because Blocks and Grids support a maximum of three dimensions. Memory on the GPU needs to be allocated and copied with `cudaMalloc` and `cudaMemcpy`. [Corb]

3.3.5 cuBLAS

cuBLAS is a library using CUDA for basic linear algebra subroutines. This library allows for many complex operations to be easily offloaded to CUDA capable GPUs. The library

consists of three APIs: the general cuBLAS API, the cuBLASXt API, which focuses on multi GPU support, and the cuBLASLt API, which adds better general matrix to matrix multiply operations. For this thesis only the first is relevant. To use cuBLAS functions the programmer has to manage the memory himself. [Cora]

3.3.6 HIP

HIP (Heterogeneous-compute Interface for Portability) is a collection of libraries and compilers, with the goal of unifying programming for GPUs. Included are CUDA, its libraries and compilers as well as ROCm (Radeon Open Compute Platform), the AMD equivalent to CUDA, and its libraries and compilers. HIP works the same way as CUDA, in fact every CUDA call can be replaced by an equivalent HIP call in place. Code written in HIP can then be compiled for both NVIDIA and AMD GPUs. The tool 'hipify' automatically translates all CUDA calls to HIP calls. There are few differences that need manual translation e.g. the separation of initialization and declaration of shared variables. HIP claims to have no performance loss over CUDA code. [AMD]

3.3.7 OpenCL

"OpenCL [Open Computing Language] is a programming framework and run time that enables a programmer to create [kernels], that can be compiled and executed, in parallel, across any processors in a system. The processors can be any mix of different types, including CPUs, GPUs, DSPs, FPGAs or Tensor Processors - which is why OpenCL is often called a solution for heterogeneous parallel programming.

The OpenCL framework contains two APIs. The Platform Layer API is run on the host CPU and is used first to enable a program to discover what parallel processors or compute devices are available in a system. By querying for what compute devices are available, an application can run portably across diverse systems - adapting to different combinations of accelerator hardware. Once the compute devices are discovered, the Platform API enables the application to select and initialize the devices it wants to use. The second API is the Runtime API, which enables the application's kernel programs to be compiled for the compute devices on which they are going to run, loaded in parallel onto those processors and executed. Once the kernel programs finish execution the Runtime API is used to gather the results." [Gro]

OpenCL kernels are written in C99 and have to be imported as a char-array to be compiled or can be compiled into an intermediate form like SPIR-V. Kernels and data transfers are queued up in an OpenCL command-queue. These queues can be in-order or out-of-order. A single device can have multiple command queues.

Rewriting the code for OpenCL is similar to CUDA, as both use kernels to offload code. With OpenCL, however, a OpenCL-context has to be created first for the device that the code is offloaded to later. Next a command-queue has to be created and the

kernel must be compiled during runtime for the specific device in two steps. There are options to precompile but these aren't considered in this work. The next step is to create the memory buffers on the device and copy data over. Before adding kernel calls to the command queue, the function's parameters need to be set one by one.

4 Implementation

In this chapter, I will discuss the algorithms themselves and their implementations including optimizations. The compilers used and some of their quirks are also discussed.

4.1 Compilers

Comparing different compilers is important because they differ greatly when parallelizing and/or offloading code. I will discuss the different compilers with their benefits and drawbacks, along other interesting findings along the way.

4.1.1 GCC + nvptx

GCC or rather G++ has offloading capabilities built in, but they aren't configures towards any vendor. Spack, a package manager often used in high performance computing, offers a simple configuration script, which installs the required dependencies and configures the offloading support. For the newest version of GCC (11.2) this script enables offloading to NVIDIA GPUs:

```
spack install gcc+nvptx~bootstrap
```

Specific versions can be installed like this:

```
spack install gcc@10.2.0+nvptx~bootstrap
```

The Spack repository only offers offloading to NVIDIA GPUs for now as there are no prebuilt assemblers or GNU binutil ports for AMD's GCN [Com]. This compiler was used for the Eigen, OpenMP CPU, OpenMP offloading and OpenACC implementations. OpenMP and OpenACC (`-fopenmp` and `-fopenacc`) are mutually exclusive compiler flags.

4.1.2 Clang/LLVM + CUDA

Clang, similarly to GCC, offers offloading out of the box but isn't configured. Again, Spack offers a package that is easy to install:

```
spack install
  ↪ llvm+Clang+CUDA+omp_debug+omp_tsan+all_targets
  ↪ CUDA_arch=61
```

for the current version 12.0.1. Clang currently compiles Eigen, the OpenMP implementations and OpenCL, but does not have support for OpenACC. There is an experimental OpenACC Clang compiler called CLACC that is still in early development and performs significantly worse than GCC or NVC++ [JED20]. Getting Clang to work isn't the easiest either, as the compiler, per default, has either bad or no links to the C++ STD libraries or CUDA libraries. To get OpenMP to work, Clang has to know which offloading target file to use; this flag solves the issue for CUDA 11.1 and a GPU with a CUDA compatibility of 6.1:

```
--libomptarget-nvptx-bc-path=<path_to_llvm_spack_
  ↪ installation>/lib/libomptarget-nvptx-CUDA_111-sm_61.bc
```

The CUDA compability is written as `sm_61` and is a metric to determine the architectural capability of the GPU. A higher version means that the GPU has more features that can be used. Unfortunately there is not a file for every combination of modern CUDA versions and CUDA compatibilities, a compromise has to be made between CUDA compatibility and CUDA version.

4.1.3 NVC++

NVC++ is the C++ compiler from the NVIDIA-HPC package. The compiler was originally called PGI++, before NVIDIA completely merged PGI into their system. NVC++ is based on LLVM and hides all linking between CPU host code and GPU device code. NVC++ supports OpenACC out of the box and OpenMP only if the target GPU has a CUDA compatibility of 7.0 or higher. This could be tested on the clusters of the DESY. NVC++ was used for the CUDA, OpenCL, OpenACC and cuBLAS implementations.

4.1.4 HIPCC

HIPCC is the main compiler tool of the HIP package. With HIPCC it's possible to compile C++ for CPUs as well as CUDA, ROCM and HIP for GPUs. It's important to note that this is not a compiler by itself, but that HIPCC automatically selects the correct compiler (Clang for ROCM and NVC++ for CUDA) and links the correct libraries. This compiler has been used for the HIP implementation.

4.1.5 AOMP

This compiler is an LLVM based compiler used specifically for OpenMP offloading on AMD GPUs. Manual installation is required, because the spack script has an error at some point. To offload code the following parameters have to be added: `-fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx1032` where `-fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa` tells the compiler that the offloading target is an AMD GPU and `march` tells the compiler the exact GPU

model to target. The library doesn't have a usable offloading binary for the used Radeon RX 6600 XT.

4.2 Gauß-Elimination-Algorithm

The Gauß-Elimination-Algorithm is a general matrix inversion algorithm. This means that it is possible to invert all invertible matrices. There are other algorithms like the eigen decomposition that can only invert matrices which can be decomposed using eigen values or the Cholesky decomposition to name a second.

The Gauß-Elimination-Algorithm requires the identity matrix I to be, at least visually, attached to the matrix denoted as A like this:

$$A|I = \left[\begin{array}{ccc|ccc} a_{11} & \cdots & a_{1n} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} & 0 & \cdots & 1 \end{array}\right]$$

An index of the matrix is denoted as $A[x][y]$, this corresponds to the value found in the x'th row and y'th column of the matrix A. $A[x]$ corresponds to the whole row x. dim denotes the size of one dimension. In a 2 by 2 matrix dim would be 2. row is $(0 <= row < dim | row \neq iter)$. Each iteration, one row of the matrix A gets transformed to the corresponding row of the identity matrix. The current iteration will be denoted as $iter$. This happens in three steps, as Algorithm 1 shows.
Optimizations are already possible: In the first step, it is faster to just add or subtract

Algorithm 1 Gauß elimination algorithm pseudocode

while $iter = 0$ in dim **do**
  if $A[iter][iter] == 0$ **then**
    $newline \leftarrow$ first row where $A[iter][iter]! = 0$
    current row is swapped with the row $newline$
  end if
  divide the row $A|I[iter]$ by $A[iter][iter]$.
  $A[row][iter] \cdot A|I[iter]$ is deducted from the rows where $row \neq iter$. ▷ This results in $A|I[row][iter] == 0$.
end while

another line than to swap two lines. In the second step, since all values with $x < iter$ should be zero or $x == 0$, given the third step from previous iterations, only the indices $A|I[iter][iter]$ to $A|I[iter][dim]$ change. This can be further optimized at the end so that only indices up to $A|I[iter][dim + iter]$ have to be changed, because beyond the index $dim + iter$ there will only be zeros which, again, won't change. These optimization steps of the second step can also be applied to the third step of the algorithm.

Listing 7.1 shows a simple implementation of the Gauß-Elimination-Algorithm with the optimizations already in place. In this implementation A and I are not glued together. This issue has been solved by manipulating the I instead of A, if x is bigger than dim. Line 2 corresponds to the iterations, the lines 3 to 17 implement the first step. Line 19 to 26 implement the normalization in step 2. Because $A|I[iter][iter]$ is the first value that gets manipulated while being the divisor for all other calculations in this step, the value gets saved in an extra variable named *divisor*. Step three of the algorithm is written from line 28 to line 39. $A|I[row][iter]$ would be the first value to be changed but as step three dictates: every value must be multiplied by the value at $A|I[row][iter]$. So, again, this value is saved into a variable $factor$ for each row. These issues could be worked around by calculating these indices last, but setting up the algorithm like this will help with parallelization later on.

4.3 Eigen and cuBLAS

Eigen and cuBLAS are both libraries for C++ or CUDA respectively, that offer matrix inversion algorithms and are often used for this purpose. I compare these libraries to my offloading implementations. Eigen has it's own matrix types, but my code reads matrices as C style arrays. To use Eigen, C style arrays have to be converted to an Eigen matrix. First I define a new type `MatrixXs` as an array of type `float` and dynamic size in both dimensions. Next I map the C array to `MatrixXs` using `Eigen:Map`. Now it is possible to invoke the `inverse()` method on the resulting Eigen matrix. All that needs to be done is to map the resulting Eigen matrix back to a C array. This implementation is shown in Listing 7.2.

cuBLAS is a bit more difficult to use, as this library requires a few CUDA calls in order to work. First, two functions that would print out precise error messages are defined. As CUDA/cuBLAS work on the device, an error doesn't always lead to the program crashing. The data will still be corrupted. cuBLAS also offers a matrix inverse method `cublas<t>cublasSmatinvBatched`. This method automatically calls `cublas<t>getrfBatched` and `cublas<t>getriBatched`, however only if the matrix dimensions are less than 32. If the dimensions are larger, the user has to call both functions. The intention of these methods is to process a bunch of smaller matrices at once, but a single larger matrix can be processed too. Because of this, the methods require arrays of arrays. Algorithm 2 shows the steps in pseudocode, the complete code is in Listing 7.4. To achieve this on the device, I have to allocate both the matrix `A` and `d_A` as well as the array of matrices `As` as `d_As`, on the device. Now I need to set `As[0]` equal to `d_A` and then `memcpy As` to `d_As`. `A` of course needs to be copied over to `d_A` as well. Because cuBLAS doesn't need `I`, its values were not copied over. `I` is used later to store the inverted matrix. Then the cuBLAS handle is created along with some other necessary output parameters for the cuBLAS methods. `d_info` contains status messages about each array and `pivot_element` stores the pivot sequence from the LU factorization. Next, the pivot sequences of the matrix `A` are calculated and stored. `cublasSgetriB-`

Algorithm 2 cuBLAS pseudocode

allocate d_A, d_I, d_As and d_Is on stack
allocate As and Is on heap
allocate d_A, d_I, d_As and d_Is on the GPU
$As[0] \leftarrow d\_A$
$Is[0] \leftarrow d\_I$
copy As and Is to d_As and d_Is
prepare cuBLAS variables on the GPU
cublasSgetrfBatched(handle, dim, d_As, dim, pivot_elements, d_info, 1)
cublasSgetriBatched(handle, dim, (const float **) d_As, dim, pivot_elements, d_Is, dim, d_info, 1)
copy d_Is back
free all allocated memory

`atched` calculates the inverse matrix using forward and backward triangular solvers. The calculation is out of place and is saved into `d_I`. The inversion is now complete, all that is left to do is to copy the results back and to free the allocated memory. [Cora]

4.4 OpenMP

4.4.1 Approach

To approach offloading with OpenMP I first implement a parallelized CPU version as shown in Listing 4.1, to see how data dependencies are affected by the parallelization. The outermost loop cannot be parallelized, because values in the second iteration depend on what was calculated in the first iteration. Let's look at the first step: Finding an appropriate row to add to the current row could be parallelized, but it is expected to find a fitting row quickly as sparse, not prepared, matrices are not considered. Parallelization would likely only add synchronization overhead. It is however possible and sensible to parallelize the loop that adds the suitable row to the current row in lines 10/11. `simd` accelerates this parallelization further by telling the compiler that the rows can be vectorized, thus allowing for multiple calculations at once. The loop of the second step in line 21/22 has a data dependency, because every iteration depends on the initial value of `A[iter][iter]`, but this would be one of the first indices to be calculated. To prevent this, the required values is saved in line 20 and used in shared memory by all cores. The third step of the algorithm has two nested loops that can't be collapsed into a single, better parallelizable, loop. This is because the factor needs to be held temporarily for every row. The outer loop in line 32/33 can be parallelized, because rows can be manipulated independently from each other. Iterations of the inner loop are also independent of each other, but OpenMP doesn't support nested parallelism. However, the inner loop can be vectorized using the `simd` directive.

Offloading the code from here is no longer a big step. All that had to be done is to copy the arrays to the GPU, in line 2 before starting the algorithm, and adding `target teams distribute` to every directive between `omp` and `parallel`. In the second step of the algorithm, copying the divisor to shared memory will not be sufficient, as each `team` has its own shared memory. Because of my approach to launch many small kernels instead of one large kernel, I have to create a kernel that creates the global variable `divisor` and then executes the loop iterations. The code can be seen in Listing 7.5. Unfortunately LLVM based compilers don't support nested directives when offloading. The data race is fixed the next best way: Ignoring the first iteration that would overwrite the divisor and write 1.0 to the index after the normalization. The implementation can be seen in Listing 7.6.

```
1  void openmp_cpu(float *A, float *I, int dim) {
2    for (int iter = 0; iter < dim; iter++) {
3      // swap lines if 0
4      if (A[iter * dim + iter] == 0) {
5        // find new line
6        for (int j = iter + 1; j < dim; j++) {
7          if (A[j * dim + iter] == 0) {
8            continue;
9          }
10 #pragma omp parallel for simd
11         for (int column = iter; column < dim; column++) {
12           A[iter * dim + column] += A[j * dim + column];
13           I[iter * dim + column] += I[j * dim + column];
14         }
15         break;
16       }
17     }
18
19     //normalize
20     float divisor = A[iter * dim + iter];
21 #pragma omp parallel for simd
22     for (int column = iter; column < dim + iter + 1; column++) {
23       if (x < dim) {
24         A[iter * dim + column] /= divisor;
25       } else {
26         I[iter * dim + column - dim] /= divisor;
27       }
28     }
29
30     //gauss
31 #pragma omp parallel for
32 for (int row = 0; row < dim; y++) {
33       float factor = A[row * dim + iter];
34       if (row != iter && factor != 0.0f) {
35 #pragma omp simd
36         for (int column = iter; column < dim + iter + 1; column++) {
37           if (x < dim) {
38             A[row * dim + column] -= A[iter * dim + column] * factor;
39           } else {
```

```
40                I[row * dim + column - dim] -= I[iter * dim + column -
                      ↪ dim] * factor;
41              }
42            }
43          }
44        }
45    }
46 }
```

Listing 4.1: First parallelization using OpenMP on the CPU

4.4.2 Optimizations

GCC 10.2.0 was the first version used for compiling the algorithm and had some issues with the offloading. Every kernel launched issued an implicit memory copy that could not be avoided, even though the required data was already on the device. This had a terrible performance impact, that could only be solved by upgrading the compiler to version 11.2. Another version of this code had the whole algorithm offloaded, instead of launching kernels each iteration. This, however, increased the timings at least with GCC 10.2.0. I assume this is because running the algorithm on the GPU and launch kernels from there is slower than to have the main algorithm run on the CPU and launch kernels from there. It may also be linked to the compiler problem I had. This has not been tested. Another idea for optimization is to copy `matrix[iter][iter]` back to the host, copy it's value into another variable and then let OpenMP implicitly use it in the kernel. Both are almost equally efficient, but using nested directives proves to be faster.

4.5 OpenACC

4.5.1 Approach

As OpenACC is not that different to OpenMP, the approach is very much the same. Instead of writing `#pragma omp target teams distribute parallel for simd`, the code to offload a loop across the GPU is `#pragma acc parallel loop gang worker vector`. As OpenACC's `gang worker` replaces OpenMP's `teams distribute`, so does `vector` replace `simd`. With OpenACC it is not possible to create a single threaded kernel that itself starts the parallelized loop calculation. Instead I had to skip the first iteration in the calculation to avoid the data race. Afterwards `matrix[iter][iter]` is set to 1.0 on the device as seen in line 28/29. The complete code is shown in Listing 7.7.

4.5.2 Optimization

Optimizing OpenACC is done by primarily optimizing the underlying algorithm for parallelism and data structures. These topics have been covered in Section 4.2 and Section 4.4 and are applied here as well. OpenACC also offers to set how many thread

blocks, warps and threads should be created, by adding e.g. `vector(10)` only 10 threads would be created. If no number is specified OpenACC will try to find parameters that fit the used accelerator. I'm unable to select a configuration that could beat the defaults OpenACC provides. This may not be true on every GPU and it's recommended to optimize this, if there are multiple different GPUs on a node or the code is compiled on an entirely different node.

4.6 CUDA

4.6.1 Approach

CUDA is very different to the previous offloading techniques, as it is a language subset of C++ specifically created for GPU offloading. First, for debug purposes, I define a function in lines 1-9 that would print the line and error to the console. This is helpful because there can be a lot of soft errors, which don't cause the program to crash, but render it's output useless. Algorithm 3 shows pseudocode for easier readability. First, space for the matrices

Algorithm 3 cuda_offload(float* A, float* I, dim)

```
allocate d_A and d_I on the GPU
copy A to d_A and I to d_I
setup kernel dimensions to be max(1024, dim)
while iter ← 0 < dim do
    if A|I[iter][iter] == 0 then
        finddiagonal(d_A, d_I, iter, dim)
        synchronize
    end if
    normalize(d_A, d_I, iter, dim)
    synchronize
    gauss(d_A, d_I, iter, dim)
    synchronize
    gauss_fix(d_A, d_I, iter, dim)
    synchronize
    iter ← iter + 1
end while
copy back d_A and d_I.
free memory
```

`A` and `I` gets allocated after which they are copied to the GPU. Next the device properties are pulled. `maxThreadsPerBlock` is one of these properties and needed to create an upper limit for the amount of threads per block. Other configurations are discussed in Section 4.6.2. Because the Gauß-Elimination algorithm has three steps that can each be offloaded, three kernels need to be created. The first kernel is called if `A[iter][iter] = 0.0` and shown as pseudocode in Algorithm 4. The kernel adds another row to `A[iter]`.

Algorithm 4 finddiagonal(float* A, float* I, iter, dim)

```
column ← threadIdx.x
if column == 0 then
    ___shared___newline ← first row where A[row][column] ≠ 0
end if
synchronize threads

while i = column < 2 · dim do
    A|I[row][column ← A|I[row][column] + A|I[newline][column]
    i ← i+ block dimension
end while
```

The kernel itself is tagged as `__global__`, to tag it as callable from both host and device, and it's run in only one block, because all threads need to know which row to add to the current one. This is done most easily with shared memory in a single block. It's not expected for this kernel to be called often either. First, each thread gets its ID which corresponds to a column of the current row. Next, the thread with ID = 0 searches for the first suitable row and saves it into `newline`. When a new row has been found, all threads synchronize to share the `newline`. Blocks can only hold a maximum of 1024 threads, but matrices can be much larger than that. A loop has been added in line 23 so that for example the thread with ID = 0 calculates the columns 0 but also 1024, 2048 et cetera.

Algorithm 5 normalize(float* A, float* I, iter, dim)

```
column ← threadIdx.x
___shared___diag_elem ← A|I[iter][iter]
synchronize threads
while i = column < 2 · dim do
    A|I[row][i] ← A|I[row][i]/diag_elem
    i ← i+ block dimension
end while
```

Normalization works very similarly in a single block. The pseudocode in Algorithm 5 illustrates this. At the beginning the diagonal element `A[iter][iter]` is saved as a shared variable between all threads. `__syncthreads()` creates a barrier to ensure that every thread has the same value stored before starting the calculation. The calculation itself stayed the same as in previous code snippets.

The Gauß kernel as illustrated in Algorithm 6 is launched on a 2D grid, as the inner loop is mapped to the x-axis, which in this case is only one block, and the outer loop is mapped to the y-axis. This creates one block per row. The third step is split into two kernels because of the data race in the inner loop of the calculation, where `A[row][iter]` would be overwritten if not saved into shared memory. Testing different

Algorithm 6 gauss(float* A, float* I, iter, dim)

$column \leftarrow threadIdx.x$
$row \leftarrow threadIdx.y$
if $row == iter$ **then**
    return
end if
while $i = column < 2 \cdot dim$ **do**
    $A|I[row][i] \leftarrow A|I[row][i] - A|I[iter][i] * A|I[row][iter]$
    $i \leftarrow i+$ block dimension
end while

Algorithm 7 gauss_fix(float* A, float* I, iter, dim)

$row \leftarrow threadIdx.x$
if $row == iter$ **then**
    return
end if
while $i = row < \cdot dim$ **do**
    $A|I[i][iter] \leftarrow 0$
    $i \leftarrow i+$ block dimension
end while

setups showed that both block shared memory and kernel wide shared memory were slower than to ignore `A[row][iter]` in each row and set the index to 0 when the kernel is finished. This is done by a separate kernel called `gauss_fix` and shown in Algorithm 7.

Once the algorithm is done, the matrices are copied back to the host and the memory is freed on the device. Listing 7.8 shows the used code.

4.6.2 Optimization

The CUDA implementation could be optimized in two ways: better usage of shared data and better configurations. The first implementations had multiple blocks per line. There are two 'golden' rules for setting the block sizes and grid dimensions. First, because a warp always launches 32 threads at the same time, a block should feature a multiple of 32, if possible. Secondly, it's recommended to create more threads than there are cores on the GPU to increase the GPUs usage, because some threads or warps might finish faster than others. Current NVIDIA GPUs can manage up to 1024 threads per block [Cord]. The GPU I am working with has 128 CUDA cores per SM. The first idea is to launch kernels where each block had 128 threads on the horizontal axis. The amount of blocks per row would be determined by the matrix size and the last block would have a few threads that were launched in vain. With these configurations, the relevant indices for the data race were copied back to the host and then given to the kernel as an argument. Many different widths were tried out, but launching a single block with a loop

proved to be about 20% faster than the other configurations. After this optimization, it is easy to implement shared variables per block. Further optimization might resolve the need for the gauss_fix kernel.

4.7 HIP

4.7.1 Approach

The HIP implementation could not be finished in time. The issue is, that the first `hipMalloc` causes the program to throw a segmentation fault. This is unexpected behavior because this call only allocates memory on the GPU but instead tries to access memory in the RAM that the program doesn't have access to. There aren't enough resources online that could help finding and solving the problem.

4.7.2 Optimization

There were no further optimizations done, after the original CUDA implementation.

4.8 OpenCL

OpenCL too couldn't be finished in time, as I'm not able to get this code to run. It does compile, but throws errors when executing kernels. Debugging OpenCL is very arduous because it doesn't provide meaningful status messages. The best idea to find the bug is to read the matrices back from the device after every kernel and print it. This proves to be ineffective as copying back the code in line 217, directly after the normalization kernel, already throws an `CL_OUT_OF_RESOURCES` error which means that the target buffer size couldn't get allocated. This is very confusing, because the data was written back into the buffer where it was read from. It should have more than enough space. Because other avenues seemed more fruitful, finishing OpenCL was dropped.

4.9 Benchmark infrastructure

To manage the amount of used compilers, possible input matrices and implementations, it's important to create a code structure that supports these demands. First, there is a main function for each compiler, as each compiler supports only part of the implementations, and each implementation is in its own file. This creates e.g. a binary from the Clang compiler with only the Eigen and OpenMP implementations and a GCC binary with only the OpenACC implementation. Next the matrix representation was decided to be written as a space separated string in a file. Section 5.1 shows one such matrix of dimensionality 2.

Each `main` takes four arguments: the relative path to the matrix that is to be read from a file, the the leading dimension's size of the given matrix, the algorithm/implementation

that should be used and lastly the run. Run will be discussed later in this section. After reading the arguments, space for the matrices is allocated and the given file is read into `matrix`. To read matrices of any size a simple function as seen in Listing 7.9 line 1-12 has been used.

```
0.6726750046641483 0.716604808416375
0.08822066004144324 -0.5053353327699652
```

Listing 4.2: Randomly generated matrix of size 2 with values between -1 and 1.

The file is put into an input file stream and then read line by line. The » operator allows for reading the space separated numbers one by one. After reading in the matrix, the identity matrix is filled with ones and zeros. To speed this process up, OpenMP has been used to parallelize the collapsed loops. Now the read `matrix` is copied into `calc_matrix`, to be able to compare the twice inversed results to the original matrix. This is done again with OpenMP, to accelerate the process over e.g. `std::copy`. In line 47/48 the time measurement objects are created, these are needed to calculate how much time each implementation takes. The lines 51 to 63 show what one implementation benchmark looks like. First, the given argument is compared against a fixed string to determine whether this is the implementation to use. This enables human readable arguments on the one hand and multiple, simple differentiations between implementations on the other hand. The code inside consists of two blocks, doing exactly the same thing. First, the starting time is measured, then the inversion algorithm is called and then the end time is measured. Next, the difference of the two is printed to the console. This creates two timings per run.
The run is a number that tells the program whether it should print the error margin or not. The benchmarks are designed to let each algorithm run 10 times. Because every algorithm runs twice per call, the error margin has to be printed on the 5th run, or 4th in this case, because counting starts at 0. The error is calculated as the difference of the original matrix and the twice inverted matrix and then printed to the console. The printed errors have a precision of 10 decimal places to include all errors, but keep the output files somewhat small. If something goes wrong and the error is NaN, the program quits. Here is an example on how to call e.g. the OpenMP offloading routine compiled by GCC with the size 1024 and no error output:

```
./gcc-offload ./matrices/randomMatrix_normal_1024.txt 1024
  ↪ openmp-offload 1
```

5 Results

In this chapter, I will discuss the benchmarks and their results.

5.1 Setup

The benchmarks were run on the HPC of the DESY and my local machine. The DESY cluster features multiple GPU nodes with NVIDIA Tesla V100 GPUs. These GPUs feature 5120 CUDA cores and 32GB of VRAM. Each node has two Intel Xeon Gold 5218 (16c/32t) clocked at 2.3GHz with a maximum boost clock of 3.9GHz. The DESY system will be called 'System 1'. My local machine features a GTX 1080 with 2560 CUDA cores and an Intel i7 9700k (8c/8t) clocked at 4.9GHz. This system will be referenced to as 'System 2'

The DESY cluster uses condor as a batch queue system to distribute jobs across the different nodes. To start jobs the code first has to be compiled. Because the code changes during the tests to fix smaller issues found, a python script was written to automatically compile all needed binaries with the correct compiler options. The compilers are all installed using spack and built with GCC 9.3.0. Eigen 3.4.0 is also installed using spack. For offloading GCC 11.1.0 is used, Clang is installed with version 12.0.0 and the NVIDIA HPC package with version 21.9. The next command is used for GCC in combination with OpenACC.

```
g++ -o gcc-offload-acc mains/gcc-offload.cpp
    ↪ -I<spack_location>/eigen-3.4.0-57ptt6xndy7fmu3f6w6uzuvg
    ↪ l74z56tx/include/eigen3 -O2 -fopenacc
```

The compile command used for GCC with Eigen, OpenMP-CPU and OpenMP-Offload is:

```
g++ -o gcc-offload mains/gcc-offload.cpp
    ↪ -I<spack_location>/eigen-3.4.0-57ptt6xndy7fmu3f6w6uzuvg
    ↪ l74z56tx/include/eigen3 -O2 -fopenmp
```

For Clang compilation with Eigen, OpenMP-CPU and OpenMP-Offload this command is used:

```
clang++ -o clang-offload mains/clang-offload.cpp
    ↪ -fopenmp-fopenmp-targets=nvptx64-nvidia-cuda
    ↪ -I<spack_location>/eigen-3.4.0-57ptt6xndy7fmu3f6w6uzuvg
    ↪ l74z56tx/include/eigen3 -O2 -I <spack_location>/llvm-
    ↪ 12.0.0-fmx2razucqcja44det235qdbqelf2soi/include/ -lcudart
    ↪ -L<spack_location>/nvhpc-21.9-yiw3cngl4t4
```

```
    ↪ nwambluvyxdq2ctmgcmch/Linu_x86_64/21.9/cuda/lib64 -ldl -lrt
    ↪ -pthreads --libomptarget-nvptx-bc-path=<spack_location>/
    ↪ llvm-12.0.0-fmx2razucqcja44det23qdbqelf2soi/lib/
    ↪ libomptarget-nvptx-cuda_102-sm_70.bc -Wl,-rpath,
    ↪ <spack_location>/llvm-12.0.0-fmx2razucqcja44det235qdbqe
    ↪ lf2soi/lib -B/usr/bin
```

And lastly, this command is used for NVC++ with CUDA, cuBLAS, OpenMP-Offload and OpenACC.

```
nvc++ -o nvcc-offload mains/nvcc-offload.cu -fast -acc -fopenmp
    ↪ -I<spack_location>/nvhpc-21.9-yiw3cngl4t4
    ↪ nwambluvyxdq2ctmgcmch/Linux_x86_64/21.9/math_libs/10.2/
    ↪ targets/x86_64-linux/include -lcublas -gpu=cc70
```

The compiler optimizations are set to `-O2`, because `-fast` is the recommended optimization when using NVC++ and `-fast` only uses `-O2` itself. OpenMP is enabled by adding `-fopenmp` to the compiler options. Offloading with Clang requires to specify the offloading target, in this case `nvptx64-nvidia-cuda`. The LLVM installation isn't able to find the offloading binaries for OpenMP offloading by itself so the option `-libomptarget-nvptx-bc-path=...` with the given path has to be added. In the NVIDIA terminology the CUDA compability is an important concept. It describes the architectural capability of the GPU with a major and a minor number. The Tesla V100 GPU that is installed on the cluster has a CUDA compatibility of 7.0, so an offloading binary ending with `sm_70` needs to be taken. The binary that supports `sm_70` with the newest CUDA version only uses CUDA 10.2 as shown by `cuda_102`. For OpenACC `-fopenacc` is required by GCC, while NVC++ uses the option `-acc` to enable offloading. With GCC only Eigen needs to be included. Clang has more trouble finding even the standard C++ libraries of its own installation. To be safe, all libraries of the LLVM installation are linked and all includes added to the compiler invocation. The CUDA library is also linked, as per recommendation of the LLVM/OpenMP group, adding `-ldl -lrt -pthreads` to the options. Because LLVM can't find some of the needed binaries used for linking either, the whole /usr/bin is added with `-B/usr/bin`.
It's also benchmarked how much of an impact double precision has over single precision, so each main file is compiled for single and for double precision with the added `-D dbl` as compiler option. This defines `dbl`. Each files checks if `dbl` has been defined and if so, sets the scalar to `double`, if not it's set to `float`. This results in eight binaries.
To be flexible in the choice of benchmark, and with the minutia of changes required of the cluster, an extra python script was made. The following would've been very tedious in C++, because of all of the recompiling. It takes up to 4 arguments (<binary> <algorithm> <type> <min. size>) and requires only <binary> as first parameter to start benchmarks. The <algorithm> parameter concerns the algorithm and can either name a specific algorithm or be 'all', meaning that all algorithms that can be benchmarked with the given binary will be executed. If there is no second parameter, 'all' is defaulted. The <type> parameter sets the types that are benchmarked. Again, specific types can be selected or 'all' for all types, without a third parameter, 'all' is defaulted as well. The last parameter <min. size> sets the minimum size of the matrix

that is supposed to be benchmarked. E.g. setting this to 512 would start the benchmark for all matrices with size >= 512. Now the script Listing 7.10 iterates over the with the given parameters created lists and executes the program with the given configurations. The output (i.e. timings and errors) is then saved into a file. Matrices with sizes above 1024 are only run once because the DESY cluster has timing restrictions on the jobs. Here is an example on how to call the script:

```
python benchmark-desy.py ./clang-offload all normal 512
```

Condor can create jobs from a list of parameters. These jobs can not be python scripts so the Condor job have to call a shell script that itself calls the above mentioned python script on the node. The parameters are piped through the shell script.

To work with the same matrices on my local system and HPC clusters without having to copy them every time, another python script was written. The script uses a seed for the random function to guarantee the same values every time. There are four matrix types generated. The normal matrices contain floating values between -1.0 and 1.0, the natural matrices integers between -100 and 100. The sparse matrix is a special matrix that has floating values between -1.0 and 1.0 only on the main diagonal, the rest is filled with zeroes. This is the minimum amount of non-zero values required for a matrix to be invertible, as every row and column has exactly one value. This setup of the matrix is also not going to require step 1 of the algorithm. This helps making e.g. run times of different matrix types more comparable. Triangular matrices are filled with floating values between -1.0 and 1.0 in every index where `column` >= `row`. The rest, again, is filled with zeroes. Section 5.1 shows an example of the normal matrix type.
Benchmarking a wide variety of matrix sizes is also important, to see if the GPU implementations behave differently with e.g. changing configurations. The sizes used, are powers of two to have many smaller matrices that are more likely to be used, but also larger matrices to really stress the GPUs and have runs that take longer time.
Users on the DESY cluster can only submit a limited amount jobs in a given time frame, so not all scenarios could be run. The limit was overstepped by rerunning the benchmarks too often, because of small changes. This limit wasn't known to me until the limit was reached. This results in some graphs being incomplete.
In order to reduce the time it takes to create graphs and because of memory limitations, a random sample of one million values is taken from the saved errors. This random sample is seeded as well, to make the graphics reproducible.

5.2 General Benchmarks

5.2.1 Speedup and Timings

The first benchmarks were done on normal matrices. Figure 5.1 shows a speedup graph of the GCC compiled implementations. Eigen was used as the baseline and thus has a speedup of 1 against itself. OpenMP on the CPU seems to have a relatively stable speedup of about two while both of the offloading techniques OpenMP and OpenACC

perform very badly on smaller matrices. Added to that OpenMP offloading seems to be about 10 times slower than OpenACC. Only at a size of 1024 and upwards, offloading seems to outpace the Eigen library. OpenMP unfortunately wasn't benchmarked beyond matrices of size 1024 with Clang, but OpenACC scales really well with larger matrices. At 8192 the OpenACC implementation is well above 150 times faster than Eigen library. Figure 5.2 shows a very interesting picture in comparison. Here OpenMP CPU loses its

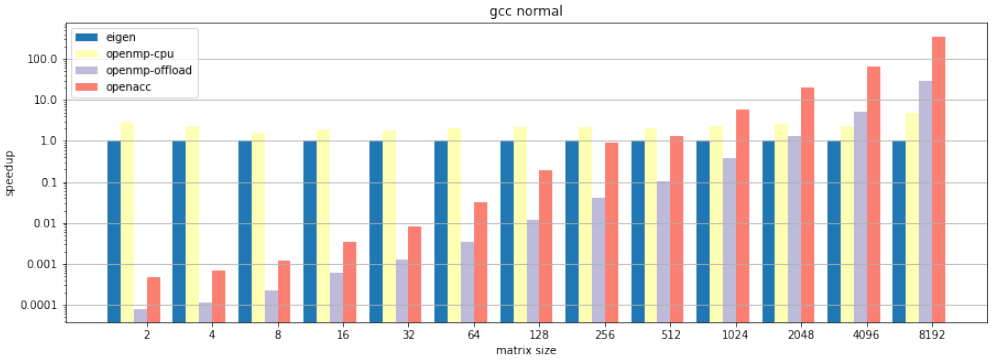


Figure 5.1: Speedup graph comparing GCC compiled implementations. (System 1)

initial edge with increasing matrix sizes. This could be because of greater latency needed for the parallelization. OpenMP offloading behaves very much the same. Comparing

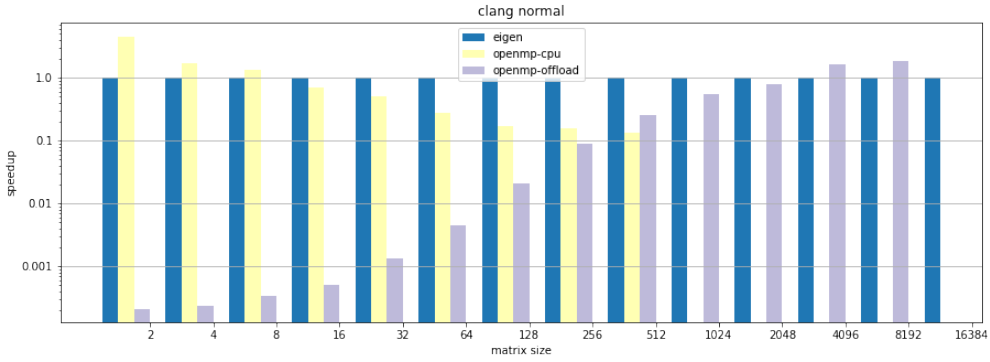


Figure 5.2: Speedup graph comparing Clang compiled implementations. (System 1)

GCC and Clang against each other, as shown in Figure 5.3, taking GCC's Eigen as base again, is very interesting as well. This is largely to the fact that Eigen runs faster when compiled with Clang. In fact, the larger the matrix, the bigger the speedup is of using Clang over GCC. This tops out at a speedup of almost 50 at size 4096. OpenMP both, on the CPU and offloaded, are reliably more than 10 times faster when compiled with Clang. The NVC++ compiled implementations in Figure 5.4 show that cuBLAS is unable to invert matrices larger than 256 by 256. Another interesting finding is that both, the CUDA and the OpenACC, implementation are faster than cuBLAS which is

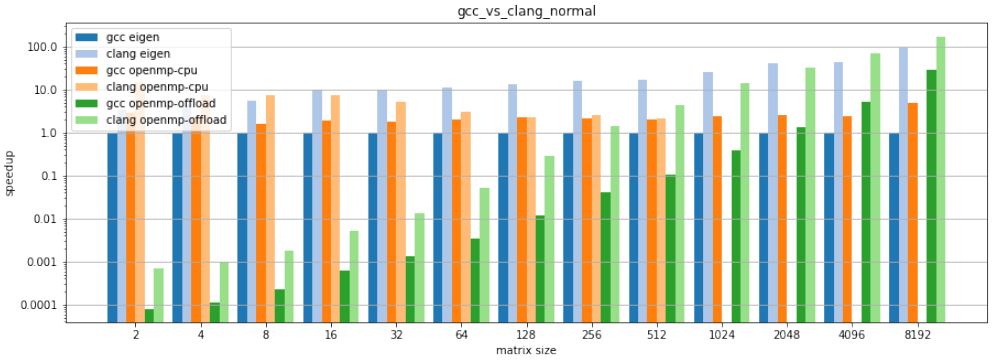


Figure 5.3: Speedup graph comparing GCC versus Clang compiled implementations. (System 1)

rather unexpected, because the cuBLAS implementation is the industry standard. The slight speed advantage of CUDA over OpenACC could be explained with the splitting of the Gauß kernel or the better block configurations. It's also noteworthy that CUDA and OpenACC have the same temporal growth rate on matrices of size 4096 and larger as Eigen. This can be seen by in the graphic as the NVC++ compiled implementations speedups are not increasing anymore. Figure 5.5 accentuates these observations, as

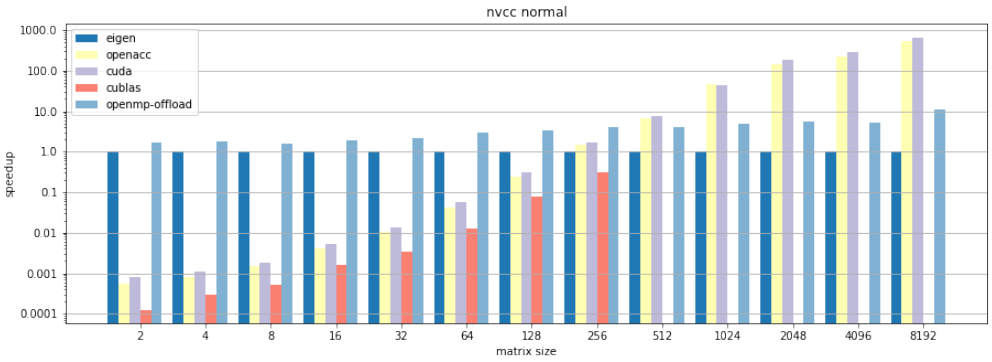


Figure 5.4: Speedup graph comparing NVC++ compiled implementations using Clang's Eigen as base. (System 1)

CUDA and OpenACC do not scale with increasing matrices at all, up to the size of 512. Then there is a small curve to be seen until the lines are parallel to the one of Eigen. Whats also very interesting to see is, that Clang's OpenMP offload behaves very similar to CUDA and OpenACC, but starting to scale at 128. NVC++'s OpenMP offloading on the other hand has a much lower initial setup time than the other offloading techniques, but it starts scaling right from the start in a parallel manner to Eigen. The only explanation that could fit this observation is, that NVC++ failed to offload the code. cuBLAS too seems to not scale with increasing matrix sizes. On my local machine, the benchmarks look a little different because the CPU is much stronger in single core

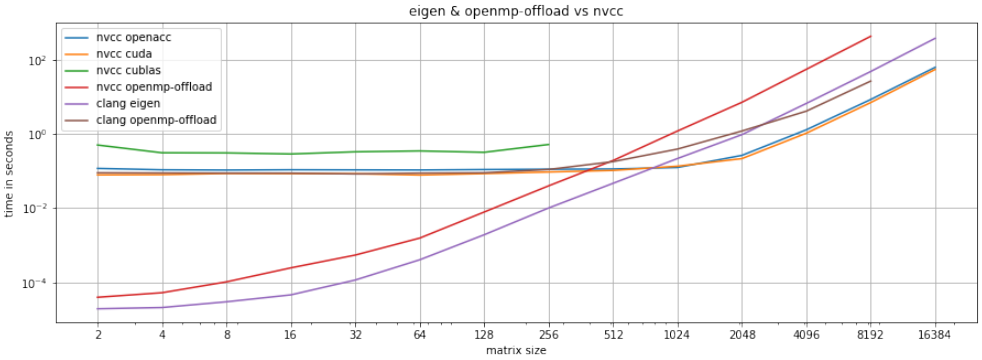


Figure 5.5: Timing graph comparing Clang's Eigen and OpenMP offload against NVC++. (System 1)

applications and my GPU is (in theory) only half as capable as the V100. This difference in CPU capability really shows in Figure 5.6. It has to be noted that neither the GPU nor the CPU could run without any interference of other applications that had to be run in the background.

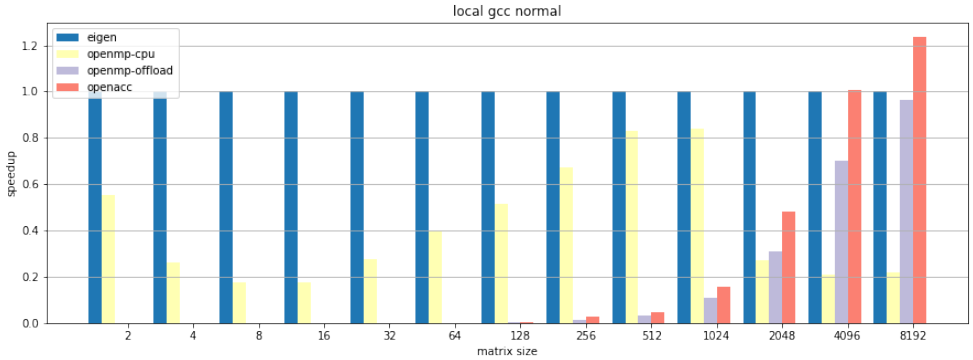


Figure 5.6: Speedup graph of the GCC implementations described in Figure 5.1. (System 2)

5.2.2 Errors

Different algorithms have different precisions and that is no different when comparing the Gauß-Elimination algorithm with Eigen or cuBLAS. A few selected graphs will be used to show the issue at hand. Figure 5.7 for example shows that with very small matrices, the errors of the Gauß elimination algorithm are very similar to Eigen and cuBLAS. The errors are only 2 to 3 times as large but still in the realm of 10^{-7}. With increasing matrix sizes as shown in Figure 5.8 as well as Figure 7.1 and Figure 7.2 the errors increase dramatically over the Eigen and cuBLAS implementations. Especially the amount of outliers that have extreme differences increase. This is because of the

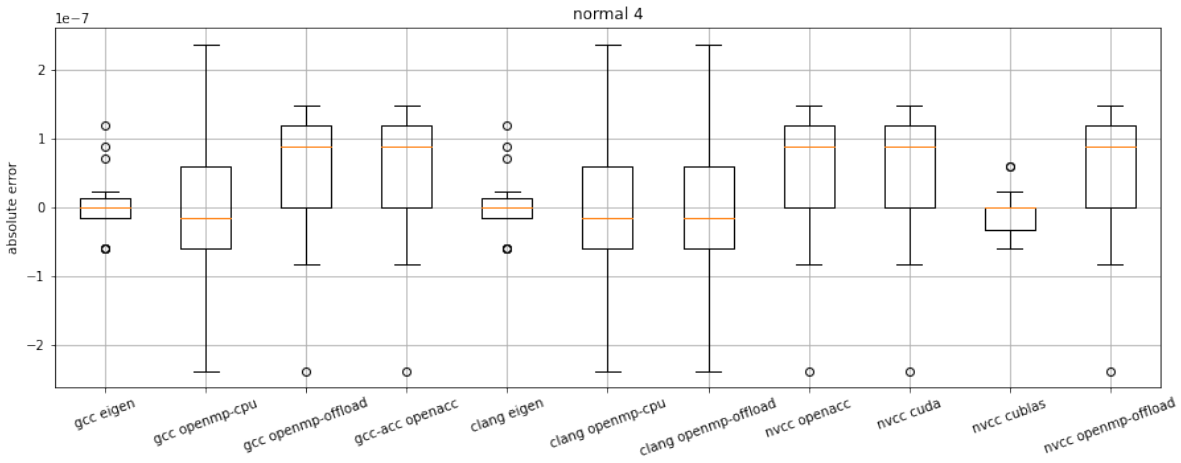


Figure 5.7: Errors on a normal 4 by 4 matrix. (System 1)

numerical instability caused by having too little precision on the numbers bit wise. This causes numbers, that should be zero, but are just very close to it, to cause great effect on the whole matrix. A partial remedy may be to use double precision instead of single

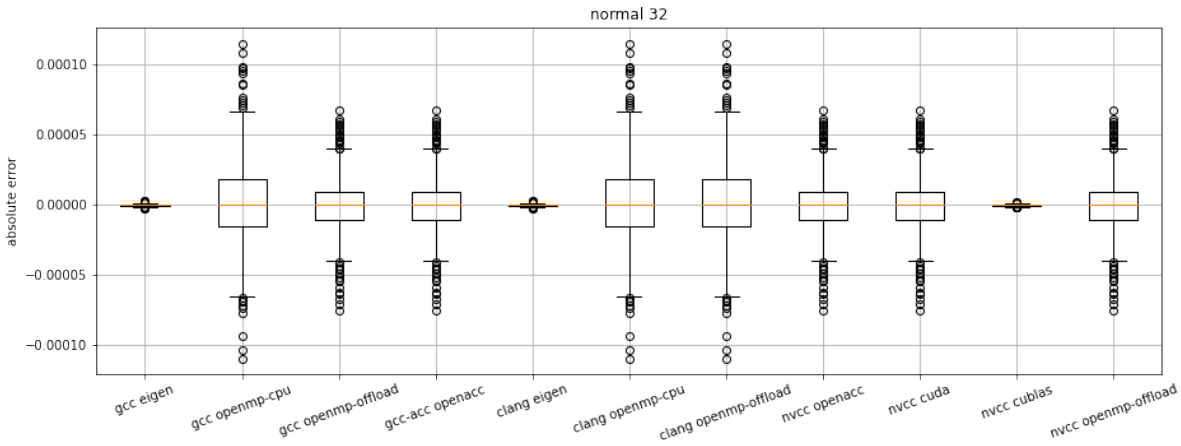
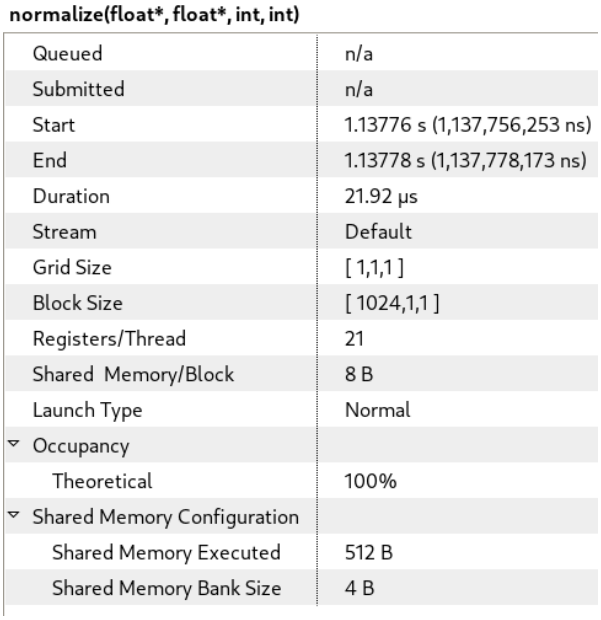


Figure 5.8: Errors on a normal 32 by 32 matrix. (System 1)

precision.

5.3 NVIDIA Profiler (NVVP)

The NVIDIA Visual Profiler (NVVP) is a graphical tool to analyze programs run on the GPU. The tool can show the specifications of the used GPUs but also detailed information about the kernels. Figure 5.9 shows such information. Occupancy is a new metric which shows how much of the GPU is being used by a given kernel. In this example the normalize kernel theoretically uses 100% of the GPUs cores. Because a direct connection to the GPU is needed, this tool has only been used on System 2. NVVP's



normalize(float*, float*, int, int)	
Queued	n/a
Submitted	n/a
Start	1.13776 s (1,137,756,253 ns)
End	1.13778 s (1,137,778,173 ns)
Duration	21.92 µs
Stream	Default
Grid Size	[1,1,1]
Block Size	[1024,1,1]
Registers/Thread	21
Shared Memory/Block	8 B
Launch Type	Normal
▽ Occupancy	
Theoretical	100%
▽ Shared Memory Configuration	
Shared Memory Executed	512 B
Shared Memory Bank Size	4 B

Figure 5.9: Properties of the normalize kernel, analyzed by NVVP. (System 2)

results on my machine should be comparable to the possible results on the DESY cluster. Table 5.1 contains the gained insights. The first insight is that with all techniques, besides my CUDA and NVC++ OpenACC implementations, the kernel configurations don't change with increasing size. This shows that the compilers have a huge influence on how well the offloading technique perform. Let's see, how these heuristics compare against each other on my local system. Figure 5.10 shows the configurations against each other in terms of speedup where the GCC OpenMP-offload is the baseline. What is interesting is, that the NVC++ configurations (ignoring cuBLAS) scale very good with larger configurations up until the size of 512 where the speedup starts dropping very fast. It's interesting, that the small Clang OpenMP configuration has the same development when comparing against the GCC configuration. The GCC OpenACC configuration has a very large grid with very large blocks and outperforms the OpenMP implementation very consistently with larger matrices. This only makes sense, if the compilers create very different kernels.

5.4 Single vs. Double Precision

Figure 5.11 shows that when using double precision the runtime of the algorithm roughly doubles, which is to be expected. This holds true for all compilers and scenarios. What is not expected is, that the errors stay at zero, or below the ten digit precision, when using doubles, as Figure 5.12 shows. With larger sizes the CUDA implementations seems to have a bug which couldn't be identified causing it to create huge errors as seen in Figure 7.3. Without CUDA however, as shown in Figure 7.4 the errors are still very close together. This suggests, that increasing the precision does actually fix the numerical

Scenario	Normalize config	Gauß config	Fix kernel config
GCC OpenMP	60,1,1 / 32,8,1	60,1,1 / 32,8,1	
GCC OpenACC	2560,1,1 / 32,32,1	1920,1,1 / 32,24,1	1,1,1 / 32,1,1
Clang OpenMP	1,1,1 / 33,1,1	1,1,1 / 128,1,1	1,1,1 / 33,1,1
NVC++ CUDA	1,1,1 / max(2*dim,1024),1,1	1,dim,1 / max(2*dim,1024),1,1	1,1,1 / max(2*dim,1024),1,1
NVC++ OpenACC	(dim / 8),1,1 / 32,4,1	(dim / 4),1,1 / 32,4,1	1,1,1 / 1,1,1
NVC++ cuBLAS	1,1,1 / 128,1,1		

Table 5.1: Comparison of different scenarios compiled with GCC using NVVP. The kernel configurations contain the grid configuration first and the block configuration second. Fix kernel refers to the kernels that were created to fix the indices that were skipped to avoid data races. cuBLAS doesn't use the Gauß Elimination algorithm but the called kernels configurations are interesting nonetheless. (System 2)

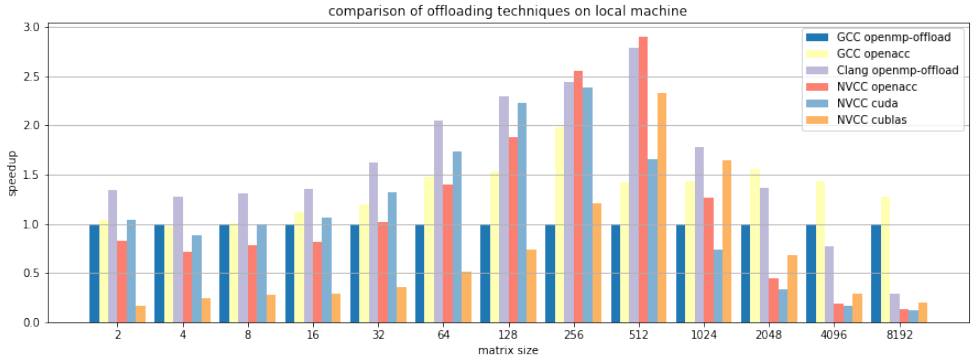


Figure 5.10: Speedup graph of offloading techniques. (System 2)

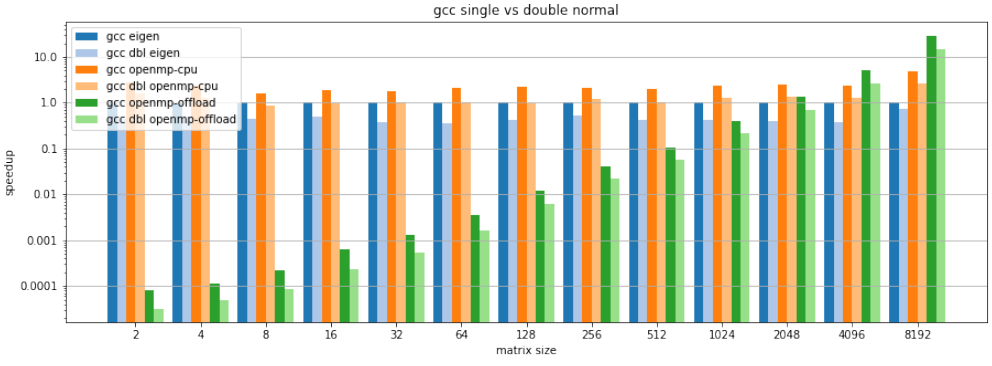


Figure 5.11: Speedup graph of single and double precision. (System 1)

instability at the cost of runtime.

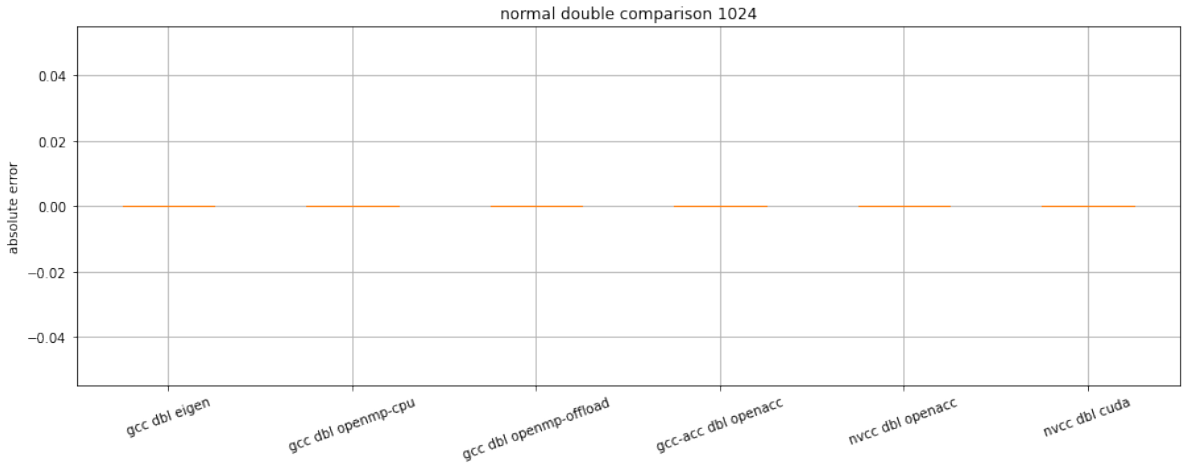


Figure 5.12: Absolute errors of GCC and NVC++ implementations of a normal matrix with size 1024. (System 1)

5.5 Natural numbers

Matrices that contain only natural numbers have no impact on the execution time itself as can be seen in Figure 7.5. The interesting part is the errors, because the values used are much further away from zero now. Figure 5.13 shows a very similar picture to the normal matrices, but with increasing matrix sizes (as seen in Figure 5.14) the error margins close in on each other. This, however, falls off, as the matrices get larger again, shown by Figure 7.6. The cause of this might be, that the errors grow in a more linear

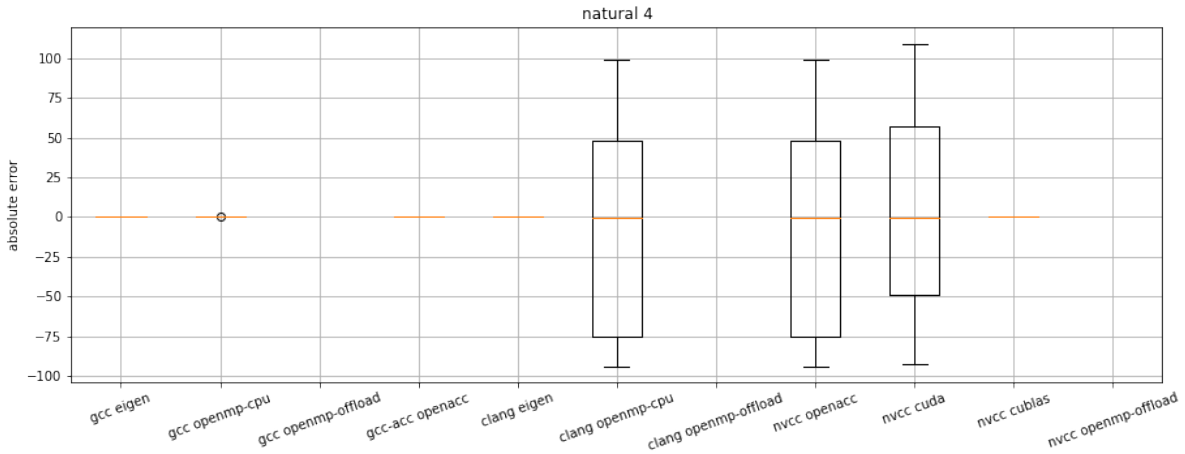


Figure 5.13: Errors on a natural 4 by 4 matrix. (System 1)

fashion with Eigen and cuBLAS but rather exponentially with the Gauß Elimination algorithm.

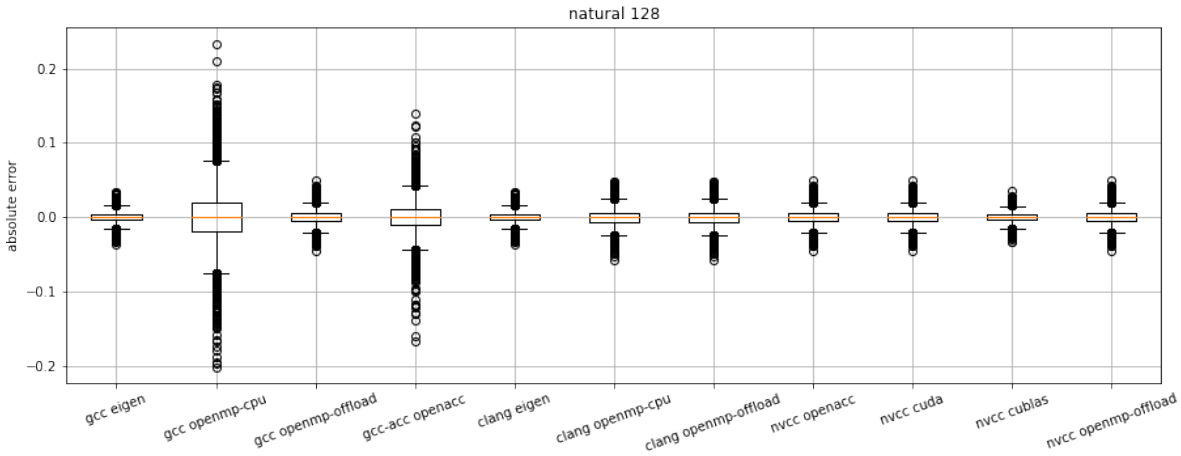


Figure 5.14: Errors on a natural 128 by 128 matrix. (System 1)

5.6 Sparse matrices

Sparse diagonal matrices should have the same runtime as normal or natural matrices. Figure 7.7 is representative for all compilers and shows this to be true. The errors on the other hand show a different story to the other implementations. In Figure 5.15 the

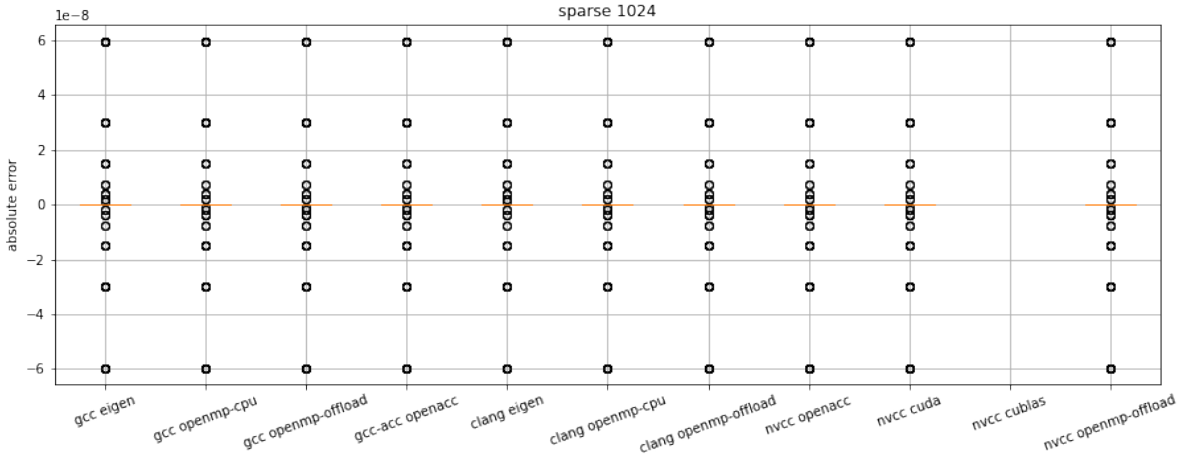


Figure 5.15: Errors on a sparse 1024 by 1024 matrix. (System 1)

errors look like they exactly match the ones of Eigen and cuBLAS this also holds true for smaller matrices. With larger matrices, CUDA has an issue creating errors well above 1.0 as can be seen in Figure 7.8. The GCC compiled OpenMP offloading implementation fails this type of matrix, with size 4096 and larger, as some numbers are NaN. This is not visible in the graphs. This makes the Gauß Elimination algorithm very suitable in terms of errors and runtime. Except, of course, larger matrices and the CUDA algorithm.

5.7 Triangular matrices

Triangular matrices, similarly to sparse matrices, don't have big timing differences. For the error part I am unsure how to interpret the data, as shown in Figure 5.16, because the errors, even at smaller sizes, are many times larger than the values themselves. This indicates to me that the created matrices may not be invertible for some reason, because Eigen and cuBLAS too failed to invert these matrices with good precision.

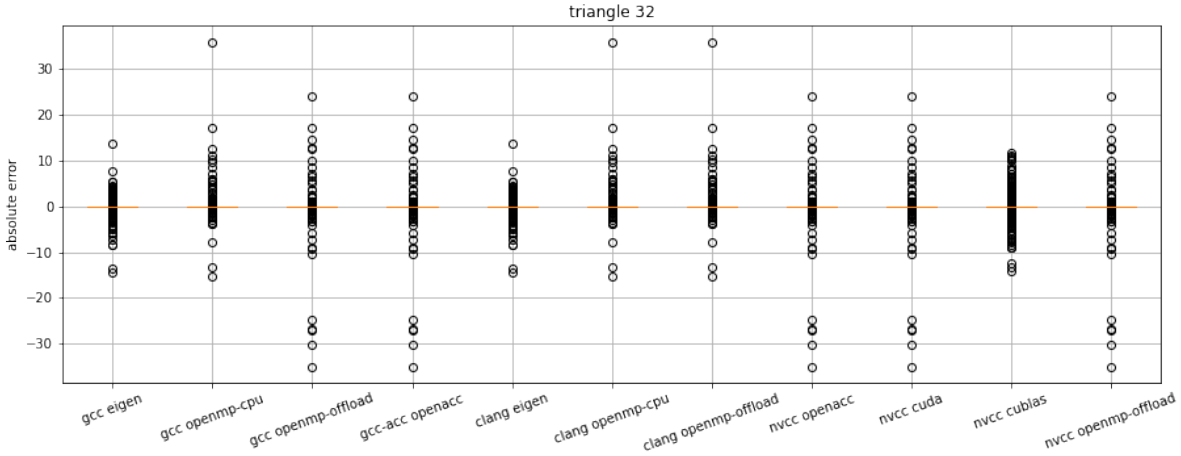


Figure 5.16: Errors on a triangular 32 by 32 matrix. (System 1)

5.8 Implementation effort

Implementation effort has multiple meanings like how much did the programmer have to work to achieve the implementation, in regards to the complexity of the work done. Another possible meaning is the lines of code written, as it's assumed that more code lines mean that more work was done. This may be somewhat true, but in this chapter I will try to touch on both meanings, to give a full picture on the effort it took to implement the offloading techniques.

5.8.1 Eigen

Implementing the Eigen inverse is very easy, even for beginners. The only part that is a bit hard at first, is Eigen's own matrix implementations and mapping matrices to and from their representation.

5.8.2 cuBLAS

cuBLAS is a library built on top of CUDA, that can't be used with at least some basic CUDA knowledge. Memory allocation and transfer are needed at a more intermediate level because of the rather complex way to allocate the lists of matrices on the GPU.

The inversion calls themselves are easy to understand, although the official cuBLAS documentation doesn't explain the parameters too well.

5.8.3 OpenMP and OpenACC

The effort needed to offload an algorithm with OpenMP and OpenACC relies heavily on the algorithm itself. In this case, the base algorithm has been written with offloading in mind so many extra steps are not needed. One issue with OpenMP is, that it's not clear what the compiler is actually doing when compiling OpenMP code for offloading. This makes it hard to find ways of optimizing the code or searching for issues. Another point is the lack of fine grained control over blocks and grids. OpenACC together with NVC++ had much more to offer in this regard. Not much code has to be written extra as there are only 3-6 lines written extra in comparison to the base algorithm.

5.8.4 CUDA and HIP

Implementing with CUDA and HIP is very much the same effort wise. Beginning with no knowledge at all, it's not too hard to get the first kernels to run, but to really utilize the GPU's capabilities, a lot of detailed knowledge about architectures, shared memory models and finding good configurations for the blocks/grids are required. To gain this knowledge a lot of presentations, papers and forums have to be read. Looking at the amount of code lines, it's about three times as many compared to the algorithm's base implementation for a single CPU. It's not easy to debug kernels because classical debuggers like GDB can't be used to step into the code. Copying back the data and looking at what is happening, combined with very descriptive error messages, help to identify the errors quickly.

5.8.5 OpenCL

OpenCL is still similar to CUDA and HIP but has a lot of extra methods one has to go through to offload code, because OpenCL isn't designed to simply call a kernel, but to launch a bunch of kernels on different accelerators and CPUs at the same time and fully utilize the whole system. This creates more than double the code lines that the CUDA/HIP implementations took. It is very hard to debug OpenCL because OpenCL uses the kernel model too, making it impossible to use e.g. GDB. But to make matters worse the OpenCL errors didn't help me to find or fix any problem I had on the way.

6 Reproducibility

The topic of reproducibility has gained a lot of traction since 2015 [Pen15] and is in essence the problem, that many papers do not provide enough insight to reproduce its results. I will make an effort here to make reproducing all results and analysis of this thesis as easy as possible. The most important thing to mention is the GitHub repository (https://github.com/DoodleSchrank/matrix_inversion) on which the code used is hosted as long as possible. The repository contains all needed scripts to create the matrices as described in Section 5.1 and the main files and launch scripts described in Section 4.9. There also is a Jupyter notebook file, that contains calls to all graphics created for this thesis. The algorithms should always create the same results when offloading and the matrices used to benchmark should be created the same on every system. The graphics use pseudo randomly selected data points making them reproducible too.

7 Future Work

In this chapter, I will discuss possible future works.

To further develop the findings of this thesis multiple aspects can be worked upon:

1. Getting benchmarks on a comparable AMD GPU would be interesting to see, if and how big the difference between GPU vendors is. This comparison could be especially interesting when looking at cost and power usage.
2. Fixing the OpenCL would also add another offloading technique to the race that could compete quite well in other benchmarks, as shown by [MLP+17] and [LSO+16]. For the specific application with the Kalman filter at the DESY, OpenCL would not make a valuable addition.
3. Another future work would be to rework the CUDA algorithm to start dim^2 threads at the beginning of the algorithm and have them calculate everything in parallel all the time. Implementing that was discussed to be out of scope of this thesis.

Bibliography

[AAC+] Xiaocong Ai, Corentin Allaire, Noemi Calace, Angéla Czirkos, Markus Elsing, Irina Ene, Ralf Farkas, Louis-Guillaume Gagnon, Rocky Garg, Paul Gessinger, Hadrien Grasland, Heather M. Gray, Christian Gumpert, Julia Hrdinka, Benjamin Huth, Moritz Kiehn, Fabian Klimpel, Bernadette Kolbinger, Attila Krasznahorkay, Robert Langenberg, Charles Leggett, Georgiana Mania, Edward Moyse, Joana Niermann, Joseph D. Osborn, David Rousseau, Andreas Salzburger, Bastian Schlag, Lauren Tompkins, Tomohiro Yamazaki, Beomki Yeo, and Jin Zhang. Preprint: A common tracking software project.

[AMD] Inc. Advanced Micro Devices. Hip. https://rocmdocs.amd.com/en/latest/Installation_Guide/HIP.html (accessed: 23.12.2021).

[AMG+21] Xiaocong Ai, Georgiana Mania, Heather M. Gray, Michael Kuhn, and Nicholas Styles. A gpu-based kalman filter for track fitting. *Computing and Software for Big Science*, 5(1):20, Oct 2021.

[Boa] OpenMP Architecture Review Board. Openmp application programming interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (accessed: 12.01.2021).

[Com] GNU GCC Community. Offloading support in gcc. https://gcc.gnu.org/wiki/Offloading (accessed: 23.12.2021).

[Cora] NVIDIA Corporation. cublas. https://docs.nvidia.com/cuda/cublas/index.html (accessed: 21.01.2021).

[Corb] NVIDIA Corporation. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed: 23.11.2021).

[Corc] NVIDIA Corporation. Nvidia a100 tensor core gpu architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf (accessed: 12.03.2021).

[Cord] NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Kepler gk110/210. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf (accessed: 23.12.2021).

[Dem97] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, jan 1997.

[Eig] Eigen. Eigen main page. https://eigen.tuxfamily.org/index.php (accessed: 23.11.2021).

[EvdGC16] Victor Eijkhout, Robert van de Geijn, and Edmond Chow. *Introduction to High Performance Scientific Computing*. Zenodo, April 2016. http://pages.tacc.utexas.edu/ eijkhout/istc/istc.html.

[Gro] Khronos Group. Opencl guide. https://github.com/KhronosGroup/OpenCL-Guide (accessed: 23.11.2021).

[JED20] Jeffrey S. Vetter Joel E. Denny, Seyong Lee. Clacc: Openacc support for clang and llvm, 2020. https://www.openacc.org/sites/default/files/inline-images/events/F2F20%20presentations/BoF-clacc.pdf (accessed: 23.11.2021).

[Kha19] Hamidreza Khaleghzadeh. *Novel Data-Partitioning Algorithms for Performance and Energy Optimization of Data-Parallel Applications on Modern Heterogeneous HPC Platforms*. PhD thesis, 03 2019.

[LGPB18] Tiago Lobato Gimenes, Flávia Pisani, and Edson Borin. Evaluating the performance and cost of accelerating seismic processing with cuda, opencl, openacc, and openmp. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408, 2018.

[LSFL07] Jean-Daniel Leroux, Vitali Selivanov, Rejean Fontaine, and Roger Lecomte. Fast 3d image reconstruction method based on svd decomposition of a block-circulant system matrix. In *2007 IEEE Nuclear Science Symposium Conference Record*, volume 4, pages 3038–3045, 2007.

[LSO+16] Xuechao Li, Po-Chou Shih, Jeffrey Overbey, Cheryl Seals, and Alvin Lim. Comparing programmer productivity in openacc and cuda: An empirical investigation. *International Journal of Computer Science, Engineering and Applications (IJCSEA)*, 6(5):1–15, Oct 2016.

[MLP+17] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ARMS-CC '17, page 1–6, New York, NY, USA, 2017. Association for Computing Machinery.

[Mur12] Kevin P. Murphy. *Machine learning - a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press, 2012.

[Pen15] Roger Peng. The reproducibility crisis in science: A statistical counterattack. *Significance*, 12(3):30–32, June 2015.

[Rup] Karl Rupp. Microprocessor trend data. https://github.com/karlrupp/microprocessor-trend-data (accessed: 12.03.2021).

[SAB13] Girish Sharma, Abhishek Agarwala, and Baidurya Bhattacharya. A fast parallel gauss jordan algorithm for matrix inversion using cuda. *Computers & Structures*, 128:31–37, 2013.

[SP08] Hans Fuhan Shi and Shahram Payandeh. Gpu in haptic rendering of deformable objects. In *Proceedings of the 6th International Conference on Haptics: Perception, Devices and Scenarios*, EuroHaptics '08, page 163–168, 2008.

[UoA] New Zealand University of Auckland. Pyjama. https://parallel.auckland.ac.nz/ParallelIT/PJ_About.html (accessed: 15.03.2021).

[Urb21a] John Urbanic. Advanced openacc, 2021. https://www.psc.edu/wp-content/uploads/2021/03/OpenACC_Advanced_OpenACC.pdf (accessed: 23.11.2021).

[Urb21b] John Urbanic. Openmp and gpus, 2021. https://www.psc.edu/wp-content/uploads/2021/06/OpenMP-and-GPUs.pdf (accessed: 23.11.2021).

[vdPST17] Ruud van der Pas, Eric Stotzer, and Christian Terboven. *Using OpenMP-The Next Step: Affinity, Accelerators, Tasking, and SIMD*. The MIT Press, 2017.

[Web21] Dr. Cornelius Weber. Lecture notes in data driven intelligent systems for university of hamburg, April 2021.

[WLHL+21] Tsai-Wei Wu, Stephen Lien Harrell, Geoffrey Lentner, Alex Younts, Sam Weekly, Zoey Mertes, Amiya Maji, Preston Smith, and Xiao Zhu. Defining performance of scientific application workloads on the amd milan platform. In *Practice and Experience in Advanced Research Computing*, PEARC '21, New York, NY, USA, 2021. Association for Computing Machinery.

Appendices

Appendix

Code appendices

```
1  void cpu(float *matrix, float *iden, int dim) {
2    for (int iter = 0; iter < dim; i++) {
3      // swap lines if 0
4      if (matrix[iter * dim + iter] == 0) {
5        // find new line
6        for (int j = iter + 1; j < dim; j++) {
7          if (matrix[j * dim + iter] == 0) {
8            continue;
9          }
10         // add lines together
11         for (int column = i; column < dim; column++) {
12           matrix[iter * dim + column] += matrix[j * dim + column];
13           iden[iter * dim + column] += iden[j * dim + column];
14         }
15         break;
16
17       }
18     }
19
20     //normalize
21     float divisor = matrix[iter * dim + iter];
22     for (int column = iter; column < dim + iter + 1; column++) {
23       if (column < dim) {
24         matrix[iter * dim + column] /= divisor;
25       } else {
26         iden[iter * dim + column - dim] /= divisor;
27       }
28     }
29
30     //gauss
31     for (int row = 0; row < dim; row++) {
32       float factor = matrix[row * dim + iter];
33       if (row != iter && factor != 0.0f) {
34         for (int column = iter; column < dim + iter + 1; column++) {
35           if (column < dim) {
36             matrix[row * dim + column] -= matrix[iter * dim +
                   ↪ column] * factor;
37           } else {
38             iden[row * dim + column - dim] -= iden[iter * dim +
                   ↪ column - dim] * factor;
39           }
```

```
40            }
41          }
42        }
43      }
44  }
```

Listing 7.1: A simple implementation of the Gauß-Elimination-Algorithm

```
typedef Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic>
    ↪ MatrixXs;

void eigen(float *matrix, int dim) {
  Eigen::Map<MatrixXs>eigenMatrix(matrix, dim, dim) ;
  Eigen::Map<MatrixXs>(matrix, dim, dim) = eigenMatrix.inverse();
}
```

Listing 7.2: Eigen implementation using C arrays as base.

```
#define cudacall(call) \
  do { \
    cudaError_t err = (call); \
    if (cudaSuccess != err) { \
      fprintf(stderr, "CUDA Error:\nFile = %s\nLine = %d\nReason =
          ↪ %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
      cudaDeviceReset(); \
      exit(EXIT_FAILURE); \
    } \
  } while (0)

#define cublascall(call) \
  do { \
    cublasStatus_t status = (call); \
    if (CUBLAS_STATUS_SUCCESS != status) { \
      fprintf(stderr, "CUBLAS Error:\nFile = %s\nLine = %d\nCode =
          ↪ %d\n", __FILE__, __LINE__, status); \
      cudaDeviceReset(); \
      exit(EXIT_FAILURE); \
    } \
  } while (0)

void cublas_offload(float *A, float *I, int dim) {
  auto **As = (float **) new float *;
  auto **Is = (float **) new float *;
  float **d_As;
  float **d_Is;
  float *d_A;
  float *d_I;

  cudacall(cudaMalloc(&d_As, sizeof(float *)));
  cudacall(cudaMalloc(&d_Is, sizeof(float *)));
  cudacall(cudaMalloc(&d_A, dim * dim * sizeof(float)));
  cudacall(cudaMalloc(&d_I, dim * dim * sizeof(float)));
  As[0] = d_A;
  Is[0] = d_I;
  cudacall(cudaMemcpy(d_As, As, sizeof(float *),
      ↪ cudaMemcpyHostToDevice));
  cudacall(cudaMemcpy(d_Is, Is, sizeof(float *),
      ↪ cudaMemcpyHostToDevice));
  cudacall(cudaMemcpy(d_A, A, dim * dim * sizeof(float),
      ↪ cudaMemcpyHostToDevice));
```

```
38
39
40      cublasHandle_t cu_handle;
41      cublascall(cublasCreate_v2(&cu_handle));
42      int *pivot_element;
43      int *d_info;
44      cudacall(cudaMalloc(&pivot_element, sizeof(int)));
45      cudacall(cudaMalloc(&d_info, sizeof(int)));
46
47      cublascall(cublasSgetrfBatched(cu_handle, dim, d_As, dim,
            ↪ pivot_element, d_info, 1));
48
49      cublascall(cublasSgetriBatched(cu_handle, dim, (const float **)
            ↪ d_As, dim, pivot_element, d_Is, dim, d_info, 1));
50      cudacall(cudaMemcpy(I, d_I, dim * dim * sizeof(float),
            ↪ cudaMemcpyDeviceToHost));
51
52      cudaFree(d_As);
53      cudaFree(d_A);
54      cudaFree(d_I);
55      cudaFree(d_Is);
56      free(As);
57      free(C);
58      cudaFree(pivot_element);
59      cudaFree(d_info);
60      cublasDestroy_v2(cu_handle);
61  }
```

Listing 7.3: cuBLAS matrix inversion using LU factorization.

```
1  void openmp_cpu(float *A, float *I, int dim) {
2    for (int iter = 0; iter < dim; iter++) {
3      // swap lines if 0
4      if (A[iter * dim + iter] == 0) {
5        // find new line
6        for (int j = iter + 1; j < dim; j++) {
7          if (A[j * dim + iter] == 0) {
8            continue;
9          }
10 #pragma omp parallel for simd
11         for (int column = iter; column < dim; column++) {
12           A[iter * dim + column] += A[j * dim + column];
13           I[iter * dim + column] += I[j * dim + column];
14         }
15         break;
16       }
17     }
18
19     //normalize
20     float divisor = A[iter * dim + iter];
21 #pragma omp parallel for simd
22     for (int column = iter; column < dim + iter + 1; column++) {
23       if (x < dim) {
24         A[iter * dim + column] /= divisor;
25       } else {
26         I[iter * dim + column - dim] /= divisor;
27       }
28     }
29
30     //gauss
31 #pragma omp parallel for
32 for (int row = 0; row < dim; y++) {
33       float factor = A[row * dim + iter];
34       if (row != iter && factor != 0.0f) {
35 #pragma omp simd
36         for (int column = iter; column < dim + iter + 1; column++) {
37           if (x < dim) {
38             A[row * dim + column] -= A[iter * dim + column] * factor;
39           } else {
40             I[row * dim + column - dim] -= I[iter * dim + column -
                 ↪ dim] * factor;
41           }
42         }
43       }
44     }
45   }
46 }
```

Listing 7.4: First parallelization using OpenMP on the CPU

```
1  void openmp_offload(float *A, float *I, int dim) {
2  #pragma omp target data map(tofrom: A [0:dim * dim], I [0:dim * dim])
3    for (int iter = 0; iter < dim; iter++) {
4      // swap lines if 0
5      if (A[iter * dim + iter] == 0) {
6        // find new line
7        for (int j = iter + 1; j < dim; j++) {
8          if (A[j * dim + iter] == 0) {
9            continue;
10         }
11 #pragma omp target teams distribute parallel for simd
12         for (int column = iter; column < dim; column++) {
13           A[iter * dim + column] += A[j * dim + column];
14           I[iter * dim + column] = I[j * dim + column];
15         }
16         break;
17       }
18     }
19
20     //normalize
21 #pragma omp target
22     {
23       float divisor = A[iter * dim + iter];
24 #pragma omp target teams distribute parallel for simd
25       for (int column = iter; column < dim + iter + 1; column++) {
26         if (x < dim) {
27           A[iter * dim + column] /= divisor;
28         } else {
29           I[iter * dim + column - dim] /= divisor;
30         }
31       }
32     };
33
34     //gauss
35 #pragma omp target teams distribute parallel for
36     for (int row = 0; row < dim; y++) {
37       float factor = A[row * dim + iter];
38       if (row != iter && factor != 0.0f) {
39 #pragma omp simd
40         for (int column = iter; column < dim + iter + 1; column++) {
41           if (x < dim) {
42             A[row * dim + column] -= A[iter * dim + column] * factor;
43           } else {
44             I[row * dim + column - dim] -= I[iter * dim + column -
                   ↪ dim] * factor;
45           }
46         }
47       }
48     }
49   }
50 }
```

Listing 7.5: OpenMP Offloading code

```
1  void openmp_offload(float *A, float *I, int dim) {
2  #pragma omp target data map(tofrom: A [0:dim * dim], I [0:dim * dim])
3    for (int iter = 0; iter < dim; iter++) {
4      // swap lines if 0
5      if (A[iter * dim + iter] == 0) {
6        // find new line
7        for (int j = iter + 1; j < dim; j++) {
8          if (A[j * dim + iter] == 0) {
9            continue;
10         }
11 #pragma omp target teams distribute parallel for simd
12         for (int column = iter; column < dim; column++) {
13           A[iter * dim + column] += A[j * dim + column];
14           I[iter * dim + column] = I[j * dim + column];
15         }
16         break;
17       }
18     }
19
20     //normalize
21 #pragma omp target teams distribute parallel for
22     for (int column = iter + 1; column < dim + iter + 1; column++) {
23       float divisor = A[iter * dim + iter];
24       if (column < dim) {
25         A[iter * dim + column] /= divisor;
26       } else {
27         I[iter * dim + column - dim] /= divisor;
28       }
29     }
30 #pragma omp target
31     A[iter * dim + iter] = 1.;
32
33     //gauss
34 #pragma omp target teams distribute parallel for
35     for (int row = 0; row < dim; row++) {
36       float factor = A[row * dim + iter];
37       if (row != iter && factor != 0.0f) {
38 #pragma omp simd
39         for (int column = iter; column < dim + iter + 1; column++) {
40           if (column < dim) {
41             A[row * dim + column] -= A[iter * dim + column] * factor;
42           } else {
43             I[row * dim + column - dim] -= I[iter * dim + column -
                 ↪ dim] * factor;
44           }
45         }
46       }
47     }
48   }
49 }
```

Listing 7.6: OpenMP Offloading code for LLVM

```
1  void openacc_offload(float *A, float *I, int dim) {
2  #pragma acc data coprow(A[0:dim * dim], I[0:dim * dim])
3    for (int iter = 0; iter < dim; iter++) {
4      // swap lines iter 0
5      if (A[iter * dim + iter] == 0) {
6        // find new line
7        for (int j = iter + 1; j < dim; j++) {
8          if (A[j * dim + iter] != 0) {
9  #pragma acc parallel loop worker vector
10           for (int column = iter; column < dim; column++) {
11             A[iter * dim + column] += A[j * dim + column];
12             I[iter * dim + column] += I[j * dim + column];
13           }
14           break;
15         }
16       }
17     }
18
19     //normalize
20 #pragma acc parallel loop gang worker vector
21     for (int column = iter; + 1 column < dim + iter + 1; column++) {
22       if (column < dim)
23         A[iter * dim + column] /= A[iter * dim + iter];
24       else {
25         I[iter * dim + column - dim] /= A[iter * dim + iter];
26       }
27     }
28 #pragma acc serial
29     A[iter * dim + iter] = 1.;
30
31     //gauss
32 #pragma acc parallel loop gang worker
33 for (int row = 0; row < dim; row++) {
34       float factor = A[row * dim + iter];
35       if (row != iter && factor != 0.0f) {
36 #pragma acc loop vector
37         for (int column = iter; column < dim + iter + 1; column++) {
38           if (column < dim) {
39             A[row * dim + column] -= A[iter * dim + column] * factor;
40           } else {
41             I[row * dim + column - dim] -= I[iter * dim + column -
                 ↪ dim] * factor;
42           }
43         }
44       }
45     }
46   }
47 }
```

Listing 7.7: OpenACC offloading code

```
#define cudacall(call) \
  do { \
    cudaError_t err = (call); \
    if (cudaSuccess != err) { \
      fprintf(stderr, "CUDA Error:\nFile = %s\nLine = %d\nReason =
        ↪ %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
      cudaDeviceReset(); \
      exit(EXIT_FAILURE); \
    } \
  } while (0)

__global__ void finddiagonal(scalar *A, scalar *I, int iter, int
  ↪ dim) {
  int column = threadIdx.x;
  __shared__ int newline = 0;
  if (column == 0) {
    for (int row = iter + 1; row < dim; row++) {// find new line
      if (A[row * dim + iter] != 0) {
        newline = row;
      }
    }
  }
  __syncthreads();

  for (int i = column; i < 2 * dim; i += blockDim.x) {
    if (i < dim) {
      A[iter * dim + i] += A[newline * dim + i];
    } else {
      I[iter * dim + i - dim] += I[newline * dim + i - dim];
    }
  }
}

__global__ void normalize(scalar *A, scalar *I, int iter, int dim) {
  __shared__ scalar diag_elem = A[iter * dim + iter];
  __syncthreads();

  int column = threadIdx.x;

  for (int i = column; i < dim + iter + 1; i += blockDim.x) {
    if (i < dim) {
      A[iter * dim + i] /= diag_elem;
    } else if (i < 2 * dim) {
      I[iter * dim + i - dim] /= diag_elem;
    }
  }
}

__global__ void gauss(scalar *A, scalar *I, int iter, int dim) {
  int column = 1 + iter + blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

```

```
51    if (column >= 2 * dim || row == iter)
52      return;
53
54    scalar factor = A[row * dim + iter];
55
56    for (int i = column; i < dim + iter + 1; i += blockDim.x) {
57      if (i < dim) {
58        A[row * dim + i] -= A[iter * dim + i] * factor;
59      } else if (i < 2 * dim) {
60        I[row * dim + i - dim] -= I[iter * dim + i - dim] * factor;
61      }
62    }
63  }
64
65  __global__ void gauss_fix(scalar *A, int iter, int dim) {
66    int row = blockIdx.x * blockDim.x + threadIdx.x;
67
68    if (row >= dim || row == iter)
69      return;
70    for (int i = row; i < dim; i += blockDim.x) {
71      A[i * dim + iter] = 0;
72    }
73  }
74
75  void cuda_offload(scalar *A, scalar *I, int dim) {
76    scalar *d_A, *d_I;
77
78    // setup and copy matrices to gpu
79    cudacall(cudaMalloc(&d_A, dim * dim * sizeof(scalar)));
80    cudacall(cudaMalloc(&d_I, dim * dim * sizeof(scalar)));
81    cudacall(cudaMemcpy(d_A, A, dim * dim * sizeof(scalar),
        ↪ cudaMemcpyHostToDevice));
82    cudacall(cudaMemcpy(d_I, I, dim * dim * sizeof(scalar),
        ↪ cudaMemcpyHostToDevice));
83
84    // setup kernel sizes
85    struct cudaDeviceProp properties;
86    cudacall(cudaGetDeviceProperties(&properties, 0));
87
88    int threads = min(2 * dim, properties.maxThreadsPerBlock);
89    dim3 norm_block(threads);
90    dim3 norm_grid(1);
91
92    threads = min(2 * dim, properties.maxThreadsPerBlock);
93    dim3 gauss_block(threads);
94    dim3 gauss_grid(1, dim);
95
96    for (int iter = 0; iter < dim; iter++) {
97      // swap lines if 0 -> divide by 0 is not allowed
98      if (A[iter * dim + iter] == 0) {
99        cudacall(finddiagonal<<<norm_grid, norm_block>>>(d_A, d_I,
            ↪ iter, dim));
```

```
100       }
101
102       //normalize
103       cudacall(normalize<<<norm_grid, norm_block>>>(d_A, d_I, iter,
              ↪ dim));
104       cudacall(cudaDeviceSynchronize());
105
106       //gauss
107       cudacall(gauss<<<gauss_grid, gauss_block>>>(d_A, d_I, iter,
              ↪ dim));
108       cudacall(cudaDeviceSynchronize());
109       cudacall(gauss_fix<<<norm_grid, norm_block>>>(d_A, iter, dim));
110       cudacall(cudaDeviceSynchronize());
111     }
112
113     // Copy results back to host
114     cudacall(cudaDeviceSynchronize());
115     cudacall(cudaMemcpy(I, d_I, dim * dim * sizeof(scalar),
            ↪ cudaMemcpyDeviceToHost));
116     cudacall(cudaMemcpy(A, d_A, dim * dim * sizeof(scalar),
            ↪ cudaMemcpyDeviceToHost));
117     cudacall(cudaFree(d_A));
118     cudacall(cudaFree(d_I));
119 }
```

Listing 7.8: CUDA offloading code

```
1  void matrix_read(char *file, int dim, float *matrix) {
2    int row = 0;
3
4    std::ifstream infile(file);
5    for (std::string line; std::getline(infile, line);) {
6      std::istringstream inputline(line);
7      for (int i = 0; i < dim; i++) {
8        inputline >> matrix[row * dim + i];
9      }
10     row++;
11   }
12 }
13
14 int main(int argc, char *argv[]) {
15   std::cout << std::fixed << std::setprecision(10);
16
17   char *file = argv[1];
18   int dimension = std::stoi(argv[2]);
19   char *algorithm = argv[3];
20   int run = std::stoi(argv[4]);
21
22   auto matrix = new float[dimension * dimension];
23   matrix_read(file, dimension, static_cast<float *>(matrix));
24
25   auto calc_identity = new float[dimension * dimension];
```

```
26    //fill identity matrix
27  #pragma omp parallel for collapse(2)
28    for (int i = 0; i < dimension; i++) {
29      for (int j = 0; j < dimension; j++) {
30        if (i == j) {
31          calc_identity[i * dimension + i] = 1;
32        } else {
33          calc_identity[i * dimension + j] = 0;
34        }
35      }
36    }
37
38    float *calc_matrix = new float[dimension * dimension];
39  #pragma omp parallel for collapse(1)
40    for (int y = 0; y < dimension; y++) {
41      for (int x = 0; x < dimension; x++) {
42        calc_matrix[y * dimension + x] = matrix[y * dimension + x];
43      }
44    }
45
46    std::chrono::time_point<std::chrono::system_clock> start, end;
47    std::chrono::duration<float> measurement;
48    double error;
49
50    if (!strcmp(algorithm, "openmp-offload")) {
51      start = std::chrono::high_resolution_clock::now();
52      openmp_offload(calc_matrix, calc_identity, dimension);
53      end = std::chrono::high_resolution_clock::now();
54      measurement = end - start;
55      printf("%f\n", measurement.count());
56
57      start = std::chrono::high_resolution_clock::now();
58      openmp_offload(calc_identity, calc_matrix, dimension);
59      end = std::chrono::high_resolution_clock::now();
60      measurement = end - start;
61      printf("%f\n", measurement.count());
62    }
63
64
65    if (run == 4) {
66      printf("------------------------------\n");
67      for (int y = 0; y < dimension; y++) {
68        for (int x = 0; x < dimension; x++) {
69          error = matrix[y * dimension + x] - calc_matrix[y *
                ↪ dimension + x];
70
71          if (std::isnan(error)) {
72            printf("NaN\n");
73            return 0;
74          }
75          std::cout << error << std::endl;
76        }
```

```
77      }
78    }
79    return 0;
80  }
```

Listing 7.9: Example main file for OpenMP offloading.

```
1  Executable = ~/ba/matrix_inversion/script.sh
2  Log        = ~/ba/logs/log_$(Cluster)_$(Process).txt
3  Output     = ~/ba/logs/out_$(Cluster)_$(Process).txt
4  Error      = ~/ba/logs/err_$(Cluster)_$(Process).txt
5  +RequestRuntime = 14400
6  Request_Memory = 12GB
7
8  Requirements = (CUDACapability == 7.0)
9  Request_GPUs = 1
10
11 queue arguments from (
12 './gcc-offload openmp-offload sparse 2048'
13 './gcc-offload openmp-offload natural 2048'
14 './gcc-offload openmp-offload triangle 2048'
15 )
```

Listing 7.10: Condor job script used on the DESY cluster.

Graphic appendices

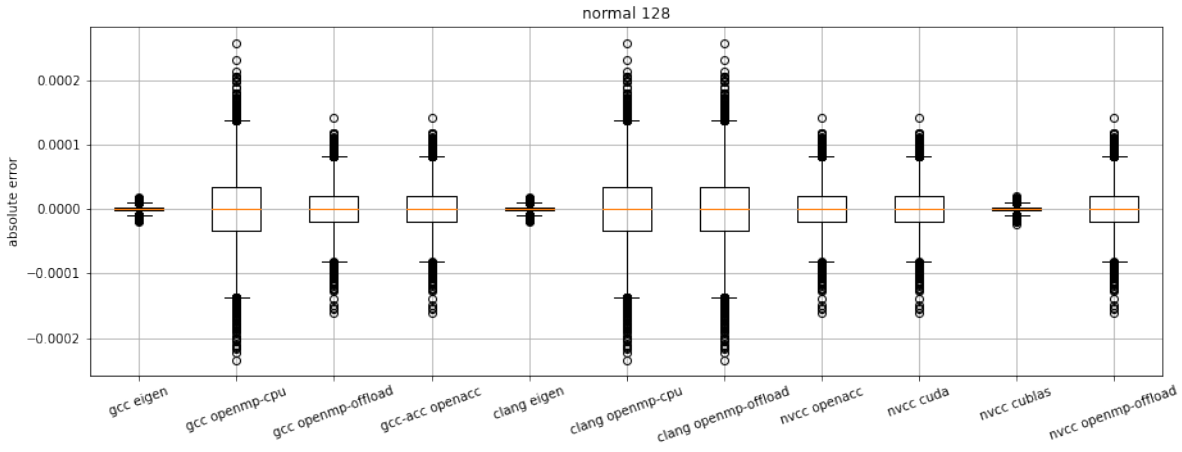


Figure 7.1: Errors on a normal 128 by 128 matrix.

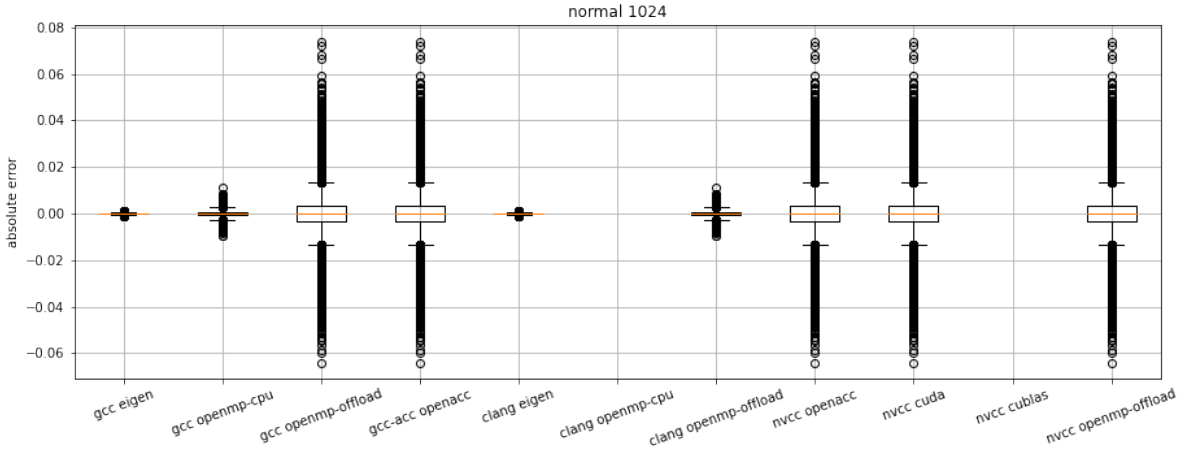


Figure 7.2: Errors on a normal 1024 by 1024 matrix.

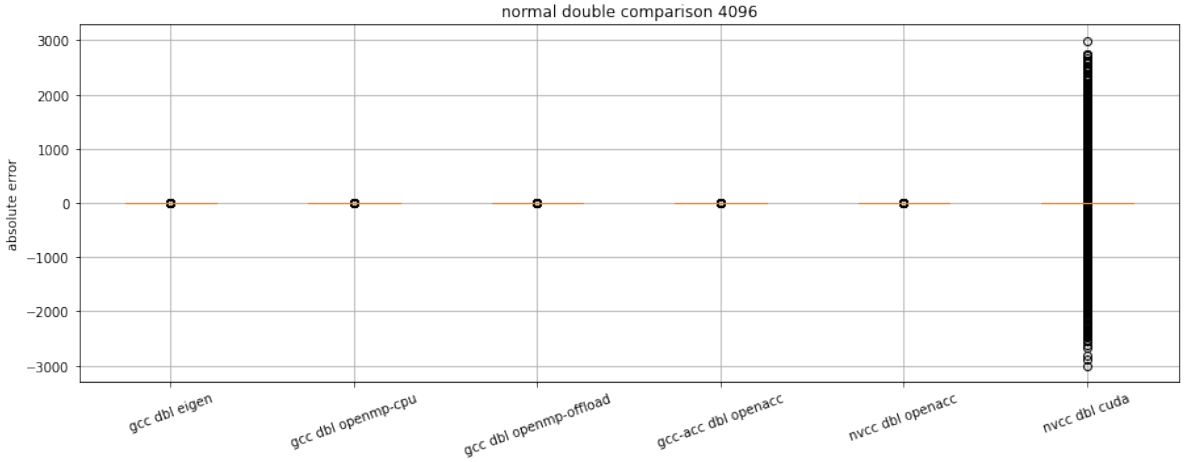


Figure 7.3: Absolute errors of GCC and NVC++ implementations of a normal matrix with size 4096.

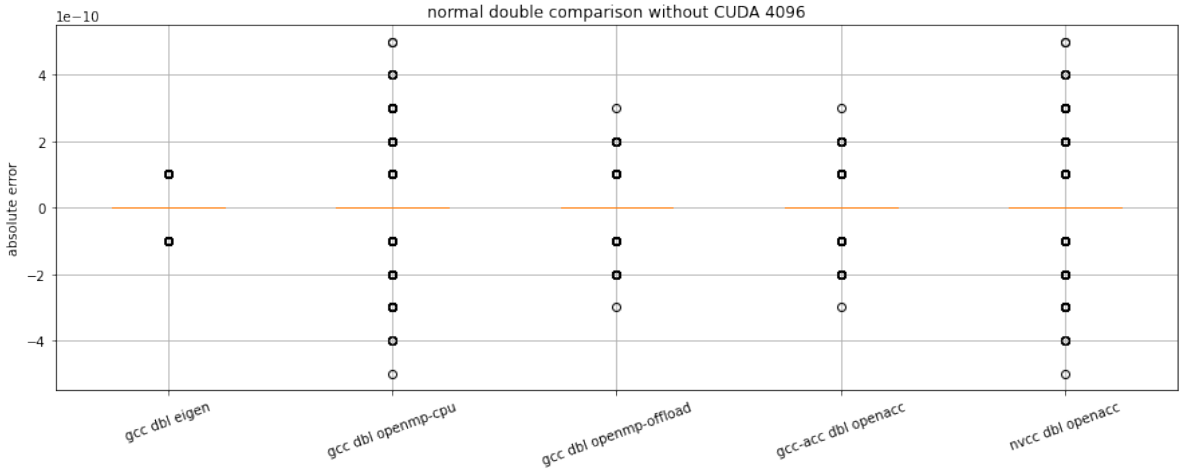


Figure 7.4: Absolute errors of GCC and NVC++ implementations of a normal matrix with size 4096 without the CUDA errors.

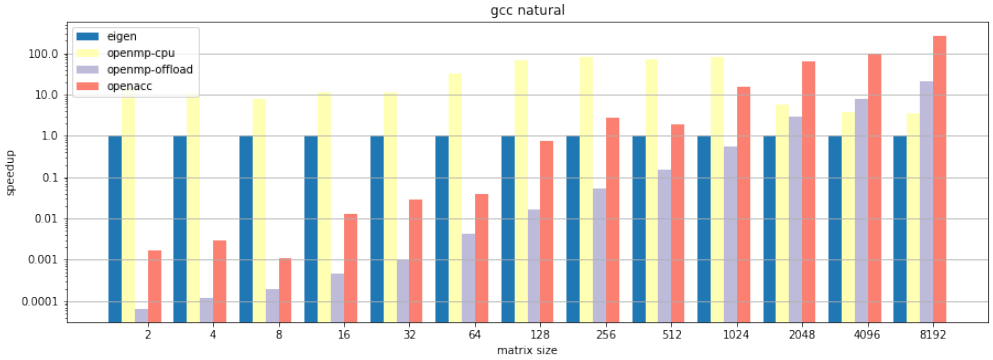


Figure 7.5: Speedup graph of the GCC implementation as representative on matrices with natural numbers.

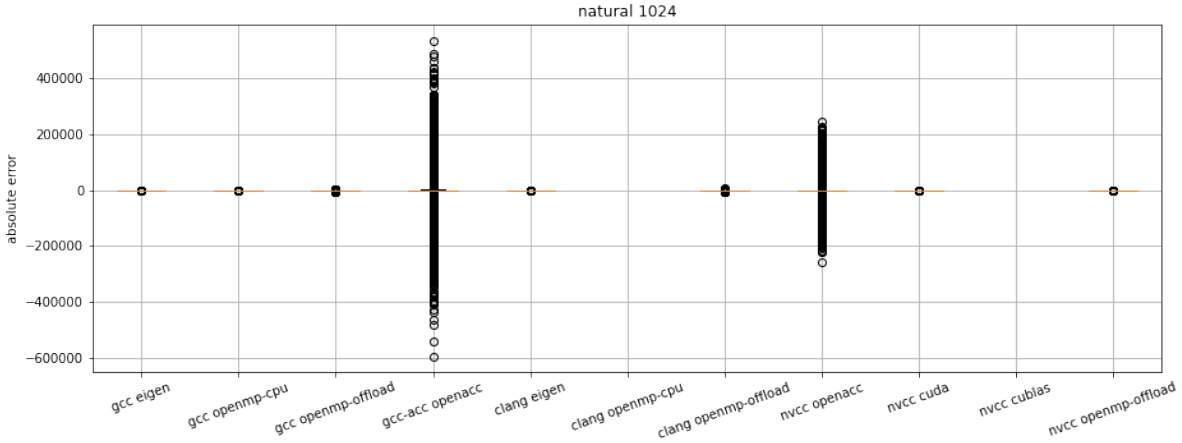


Figure 7.6: Errors on a natural 1024 by 1024 matrix.

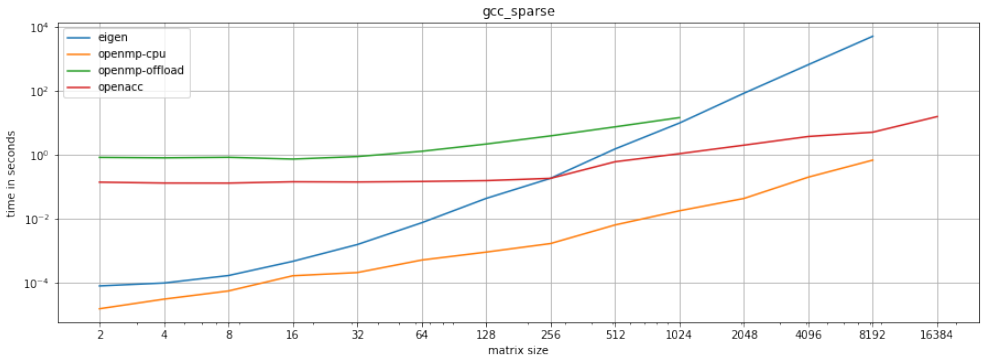


Figure 7.7: Speedup graph of the GCC implementation as representative on sparse matrices.

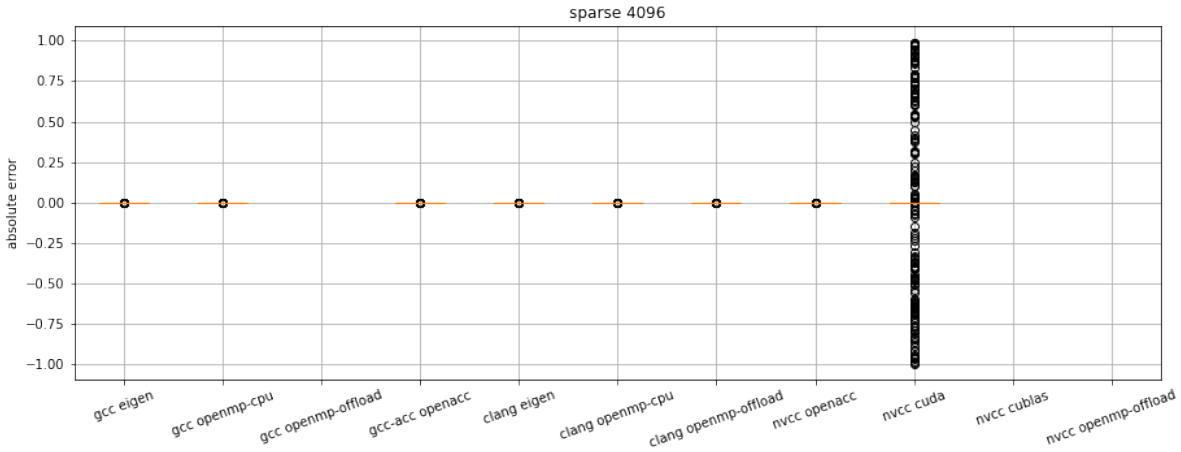


Figure 7.8: Speedup graph of the GCC implementation as representative on sparse matrices.

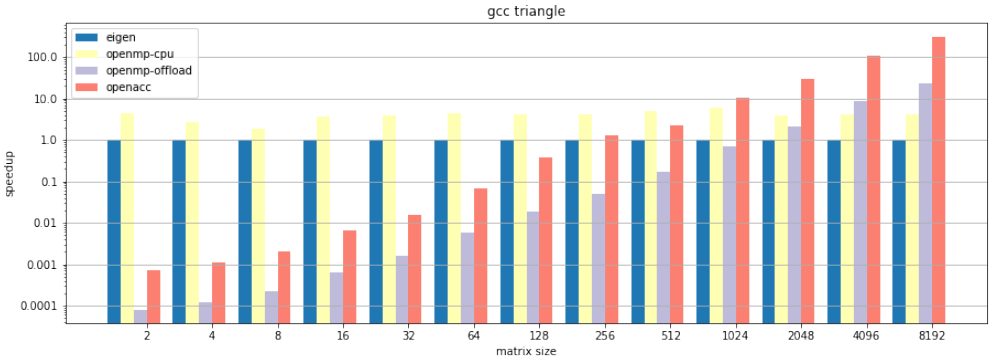


Figure 7.9: Speedup graph of the GCC implementation as representative on matrices.

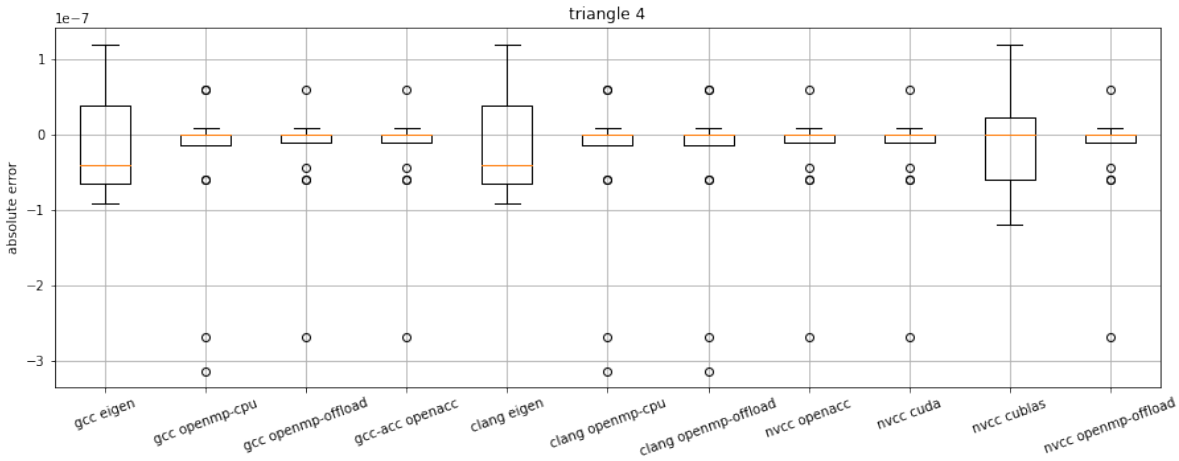


Figure 7.10: Errors on a triangular 4 by 4 matrix.

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Bachelor of Science, Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg den 28.01.2022 Jannik Lönneker

Ort, Datum Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Hamburg den 28.01.2022 Jannik Lönneker

Ort, Datum Unterschrift