# HPC CUDA compiler performance comparison

**Seminar "High-Performance Computing with GPUs" at University of Cologne, 2025**

*Date of submission: 04. August 2025*

TOBIAS CTISTIS, Universität zu Köln, Germany (Author, Student)
DR. ANDREAS HERTEN, Forschungszentrum Jülich, Germany (Supervisor)

This seminar thesis compares the performance of selected CUDA samples and an real-world application between the compilers Clang and NVCC combined with GCC runned on the HPC system JURECA with five optimization flags `-O0` up to `-O3`, `-Ofast` and `-ffast-math`. An empirical experiment was runned to benchmark compilation and execution time as well as binary size. The benchmarks were automated with scripts written for compilation and execution with Clang and NVCC for all five optimization stages. The analysis of the results reveals that no compiler can be clearly recommended and the performance is highly dependent of the compiler, its optimization flag and the problem.

## 1 Introduction

HPC systems (High performance computing) provide dedicated hardware optimized for executing applications with high computational performance. They often also provide GPUs (Graphic processing units) which can be utilized in order to increase computational power. For applications running on those systems, performance is an important design factor. Beside the code itself, the used compiler could also impact the application performance during its compilation process.

*1.0.1 Setting.* This seminar thesis aims to study this impact on the HPC system JURECA at Jülich supercomputing centre (Germany) which has 4 NVIDIA A100 GPUs [1]. For GPU utilization, applications can be written using the CUDA API (Compute Unified Device Architecture) [2]. They contain code segments to be executed on the CPU (host code) and the GPU (device code) [2]. Because the device code syntax is not C/C++ standard compliant, another compiler needs to interpret it [2]. NVIDIA CUDA Compiler (NVCC) for device code is widely used with GCC for host code compilation [2]. Alternatively, Clang can both compile host and CUDA device code [3].

*1.0.2 Research questions.* This raises the question, which compiler should be used for which purpose. The **main research questions** of this seminar thesis are:

(1) Does Clang and GCC+NVCC differ in performance of compilation time, execution time and/or binary size of generated binary files?
(2) What can be said about this impact?

*1.0.3 Approach.* To answer these questions, an empirical experiment is conducted on JURECA. It runs benchmarks for compilation time of selected samples with both compilers runned in five optimization stages. It further benchmarks the execution time and binary size of the generated binaries. Finally a quantitative analysis is done for the results.

Authors' Contact Information: Tobias Ctistis, tctistis@netcologne.de, Universität zu Köln, Cologne, North Rhine-Westphalia, Germany (Author, Student); Dr. Andreas Herten, a.herten@fz-juelich.de, Forschungszentrum Jülich, Jülich, North Rhine-Westphalia, Germany (Supervisor).

## 2 Fundamentals

After introducing the topic, this chapter provides necessary fundamentals. Programming and compilation of CUDA applications are covered, as well as compiler optimizing techniques.

### 2.1 CUDA API

This subsection introduces the usage of the CUDA API and outlines the compilation process with further detail on differences between Clang and NVCC.

*2.1.1 Main concepts.* SIMD (Single instruction multiple data) is a programming model in which a single operation is applied to different operands at the same time. GPUs are capable to operate at high throughput rates. They use dedicated hardware which is designed to implement SIMD. In the CUDA API, data for code execution is mapped to hardware using a grid of threads. This grid is further partitioned into blocks. All threads within a block can be processed in parallel following the SIMD model. [2]

*2.1.2 Host-Device interaction.* CUDA applications code is separated into two parts, the host and device code. The host code is executed on a CPU and consists of C/C++ code. The device code is executed on a GPU and contains language extensions. The function declaration specifier `__global__` marks this function as device code, callable from both host and device code. This is presented for an example in listing 1. [2]

```
__global__ function kernelFunction(PARAMETERS ...) {...}
```

Listing 1. Example for CUDA device code function

From host code, this functions can be called as shown in listing 2 with an additional specifier `<<<BLOCKS, THREADS>>>` which specifies the number of blocks in the grid and the number of threads within each block. [2]

```
kernelFunction<<<BLOCKS, THREADS>>>(PARAMETERS ...);
```

Listing 2. Example for CUDA device code called from host code

Also the CUDA API provides functions used for GPU interaction. This includes CUDA runtime which provides memory organization like shared memory. [2]

*2.1.3 Compilation.* NVCC uses split compilation. First it separates host from device code. Host code is compiled by an specified C/C++ compiler, here this is GCC. Then device code is compiled with NVCC into an assembler object (PTX) or binary object (cubin object). Finally, all device calls are replaced by code from CUDA runtime library necessary to call the precompiled PTX or CUBIN code. Clang on the other hand uses merged parsing. Here first the host code is compiled. After this the device code is compiled and both are merged into a single binary file. [3]

### 2.2 Compiler optimization

In this subsection, techniques used in the compilers for performance optimization are presented. Also compiler flags are introduced to use these optimizations during compilation.

*2.2.1 Techniques.* Compilers analyze written code and can perform optimizations [4]. This means editing in such a way that the functionality remains unchanged, while it has some performance impact [4]. Table 1 shows relevant techniques grouped by code structure.

| Code structure | Optimization | Description |
|---|---|---|
| Functions, code blocks | Unused code removal [5] | Code which is never called gets removed. This reduces the applications binary file size and used RAM for execution. [5] |
| | Function inlining [6] | Replaces all function calls with its body and removes it. [6] |
| Loops: while, for | Loop unrolling [7] | Instead of computing a loop iterating consecutive through all indices of the index variable, it is only iterated over every n-th index. Then all items in between are processed via relative indices. [7] |
| | Vectorization [8] | Splits register into even regions and perform operation on them individually (Single instruction multiple data) [8] |
| | Loop-invariant code motion [9] | Code within a loop which does not access the loop variable is considered loop invariant. This optimization moves it out of the loop without changing code semantics. [9] |
| Conditionals: if-else | Branch prediction [10] | Tries to predict the most likely taken path for an if-statement in order to process it faster than sequentially [10] |
| Mathematics | Floating point [11] | Precision of floating points impacts the calculation time. Unnecessary high precision can therefore be reduced to increase calculation speed. [11] |
| | Algebraic identity removal [12] | Simplify equations with removing parts which hold true for every input numbers in order to speed up calculation time. [12] |
| | Scalar replacement [13] | Store intermediate variables as scalar instead of arrays. So they can be stored in registers opposed to main memory to improve access time. [13] |
| Memory | Memory access [14] | Data accesses are first looked up from cache and loaded there from disk if not available. Access patterns can increase the Cache-Hit ratio, such that data is more likely to be in the cache. [14] |

Table 1. Compiler optimization techniques classified by code structure

*2.2.2 Compilation flags.* Both compilers Clang and NVCC can be parameterized with optimization flags in order to control the applied code optimizations during compilation. There are individual parameter flags and bundles which cover a set of these in one flag. An overview over considered optimization flags with associated used techniques is given in table 2.

| Flag | Optimizations | Techniques | |
|---|---|---|---|
| | | Clang | NVCC |
| `-O0` [15] | Nothing [15] | - [16] | - [17] |
| `-O1` [15] | Small [15] | Unused code removal [16] | Algebraic identity removal [17] |
| `-O2` [15] | Moderate [15] | Function inlining [16]<br>Loop unrolling [16] | Function inlining [17]<br>Invariant code motion [17] |
| `-O3` [15] | Aggressive [15] | Memory access [16]<br>SIMD [16]<br>Branch prediction [16] | Loop unrolling [17]<br>Scalar replacement [17] |
| `-Ofast` [15] | Mathematics [15] | Floating point precision [16] | Floating point precision [11] |

Table 2. Optimization flags for Clang and NVCC with associated techniques

# 3 Related Work

This chapter outlines related work, following on the fundamentals in the previous chapter. This includes benchmarking performance in HPC systems and compiler optimization.

## 3.1 HPC benchmarking

Benchmarking performance for HPC systems was researched concerning sample construction and size.

*3.1.1 Sample size.* Marjanović et. al. (2016) states that the sample size is considered to be an important factor measuring performance on HPC systems. When measuring too small problem sizes, the execution overhead dominates the benchmark. Benchmarks should therefore adapt the compute unit speed, memory size and bandwidth, latency and message rate of the interconnection network in order to produce meaningful results. [18]

*3.1.2 Sample selection.* Marjanović et. al. (2016) distinguishes between two workload types: collection of applications and application kernels and single applications sensitive to most relevant system properties. Simple kernels benchmarks measure specific isolated application properties. With benchmarking real-world applications which contain all relevant properties of an HPC application, one can identify side effects of single kernels and performance effects evolving from combination of multiple kernels which are supposed to be executed together. Therefore kernels and real-world applications are both considered relevant for measuring HPC application performance. [18]

## 3.2 Compiler performance

Compiler optimization was regarding classification and compiler comparisons.

*3.2.1 Optimizations.* Schneck et. al. (1973) suggests to classify compiler optimization techniques into machine dependent, architecture dependent and architecture independent. If optimizations rely on specific hardware properties, they are considered machine dependent. If they expect an specific instruction set, they belong to the category architecture dependent. Otherwise they are architecture independent. [19]

*3.2.2 Comparisons.* Li et. al. (2018) compared the performance of CUDA and OpenACC in particular for underlying compilers. They selected 19 kernels in 10 benchmarks from the Rodinia suite and NVIDIA, including matrix multiplication and jacobi solver. Performance measures were execution time and data sensitivity. The benchmarks were executed with nvprof from NIVIDA Toolkit 2015 on a server consisting of host processor (10-core Intel Xeon E5-2650 at 2.3 GHz / 16 GB DDR4 RAM / 25 MB cache) and GPU (NVIDIA Tesla K40c at 0.75 GHz / 11,520 MB global memory). They found that CUDA code runs significantly faster than OpenACC code. [20]

## 4 Methodology

Following on the related work, this chapter explains the design of the conducted experiment in detail. Addressed topics are sample selection, performance measures and benchmark planning.

### 4.1 Samples

In this subsection, the criteria used for sample selection and the experiment samples are presented.

*4.1.1 Criteria.* The samples are classified into their algorithmic complexity. This means how many code they contain and how complex the call structure is organized. Beside its complexity, the performance impact is also suspected to be influenced by how often the code is invoked. Simple code may also impacts performance, if it is called very often.

*4.1.2 Classification.* Here an ordinal scale with categories basic, medium and complex is used for classification. Basic samples overall have low algorithmic complexity and therefore consist of few code lines and/or linear code structure without many functions, calls or recursions. Complex samples overall have high algorithmic complexity and medium samples lay in between.

*4.1.3 Selection.* The selection for the experiment is presented in table 3. All samples from categories basic, medium and complex were taken from the official NVIDIA CUDA samples repository which aims to demonstrate CUDA API functionalities [21]. The real-world example was provided by Dr. Andreas Herten (FZ Jülich).

| Category | Samples |
|---|---|
| Basic | Printf<br>Scalar product |
| Medium | Matrix multiplication<br>Vector addition |
| Complex | Reduction |
| Real-world example | Jacobi solver |

Table 3. Selected samples for the experiment

### 4.2 Performance measures

This subsection presents the performance measures used in the experiment.

*4.2.1 Compilation time.* Compilation time is considered as the time which the used compiler needs in total to generate a binary output file from all related input files. It is benchmarked as the time elapsed until the compiler complete generating the binary output file. Here only limited execution are considered due to comparison later on.

*4.2.2 Execution time.* Execution time is considered to be the time an application takes after being started until it terminates. It is benchmarked as time elapsed between start and completion of the whole application. Here only limited execution times are taken into account due to comparison later on. For HPC applications, this measure is most important because they are often computational-intensive.

*4.2.3 Binary size.* The binary size of a compiled sample is benchmarked in kilobytes which it occupies on disk.

### 4.3 Experiment

This subsection focuses on the experiment design to benchmark the selected samples.

*4.3.1 Overview.* The overall plan of the experiment consisting of the consecutive phases preprocessing, benchmark and postprocessing is depicted in figure 1.

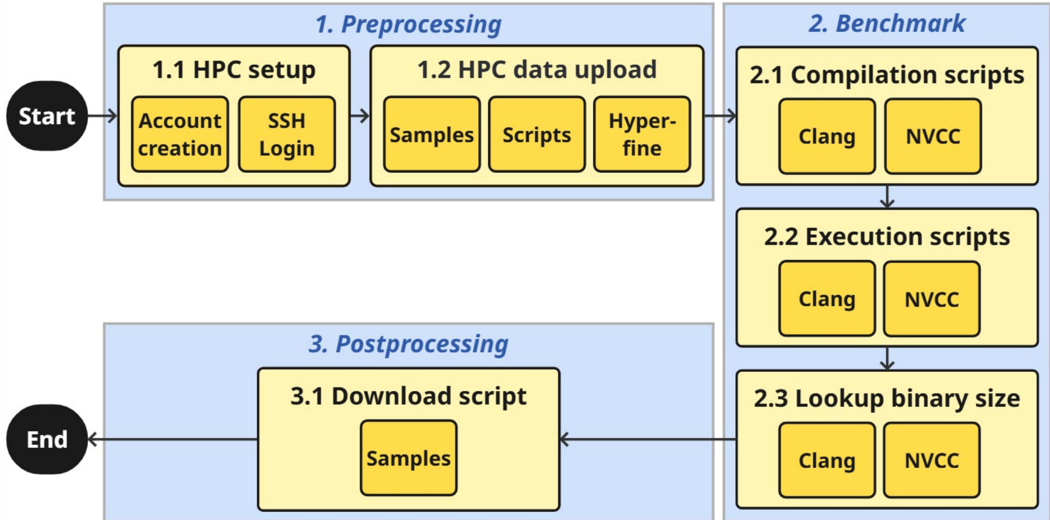



Fig. 1. Overall experiment plan

The preprocessing phase starts with connecting to the HPC system. Also all necessary data is uploaded there via command-line over SCP using SSH. In the benchmark phase the scripts are executed on the HPC system to generate the results. Finally, the postprocessing phase ends the experiment, in which all sample projects are downloaded back.

*4.3.2 Preprocessing.* For preprocessing (step 1) of the experiment the HPC setup is done (step 1.1). First a new account is registered on JURECA which required an username, password and SSH public key as credentials [22]. With Cisco Duo as authenticator app for Android, a second factor was utilized to further improve security. Then a SSH connection was established over command-line authenticated with the corresponding private key and an one-time passkey from Cisco Duo. Finally, all data needed for the experiment was uploaded into JURECA storage space via SCP (step 1.2) which includes

- All 6 sample project folders
- 4 Benchmark scripts: Compilation and execution scripts for Clang and NVCC
- Hyperfine: Command-line benchmarking executable, which determines the execution time of another program [23, 24]

*4.3.3 Benchmark.* After preprocessing, all benchmark scripts are executed on JURECA (step 2) via command-line over SSH. This includes one compilation and execution script for each compiler which benchmarks all samples. The compilation scripts have to be executed first to generate the binary executable file (step 2.1). Next, the execution benchmark is launched (step 2.2) and the file sizes obtained via console (step 2.3). In the end, the results have been written in the build folders of the respective projects.

*4.3.4 Postprocessing.* The experiment ends with the postprocessing phase (step 3). Here the download script is executed on the local system (step 3.1). It calls SCP for each sample for downloading the associated project folder from JURECA to obtain the results.

## 4.4 Compilation scripts

This subsection explains in detail the conception of the compilation scripts.

*4.4.1 Overview.* Both scripts first define the sample directory paths and load the modules Clang, CMake and CUDA. After that, the function `Benchmark` is defined which takes sample directory path and optimization flag as parameters. Figure 2 visualizes the overall conception, including the call structure and the function `Benchmark`. Finally this function is called for all combinations of samples and optimization flags. It clears the build subfolder, generates a makefile and benchmarks the make call. In the following, these steps are presented in depth including used parameters.

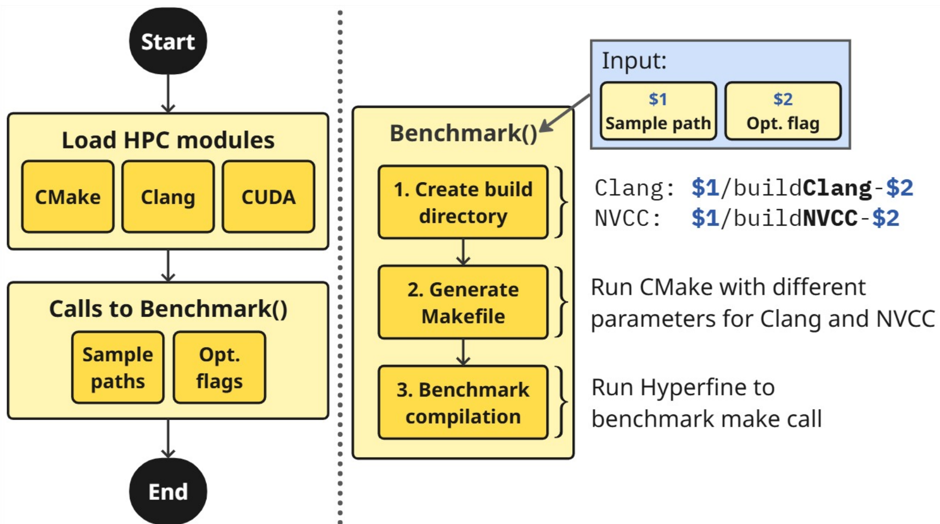



Fig. 2. Conception of compilation benchmark scripts. Left: call structure, right: benchmark function.

*4.4.2 Build directory.* At first, the destination for compilation output is created. A unique empty build directory is created in the sample directory. It is named `build[Clang/NVCC][Opt. flag]` based on the actual configuration. If this directory already exists, it will be deleted.

*4.4.3 Makefile.* Then CMake is invoked to generate correct makefiles for the current configuration. For Clang and NVCC different parameters are passed, both calls are shown in listings 3 and 4. [25]

```
cmake -DCMAKE_CUDA_ARCHITECTURES=80
↪ -DCMAKE_CUDA_FLAGS=$2 ..
```

Listing 3. CMake call in NVCC compilation script, line 35

```
cmake -DCMAKE_CUDA_ARCHITECTURES=80
↪ -DCMAKE_CXX_COMPILER=clang++
↪ -DCMAKE_CUDA_COMPILER=clang++
↪ -DCMAKE_CUDA_HOST_COMPILER=clang++
↪ -DCMAKE_CUDA_FLAGS=$2 ..
```

Listing 4. CMake call in Clang compilation script, line 35

Both scripts first set the architecture compatible to JURECA[26]. Also the current optimization flag stored in $2 is passed to the parameter `DCMAKE_CUDA_FLAGS` [27, 28]. The last argument is the build source which is the sample directory (parent of build folder). For clang compilation, these parameters also set to `clang++`:

- `DCMAKE_CUDA_COMPILER`: Compiler used for device code [29]
- `DCMAKE_CXX_COMPILER`: Compiler used for host code [29]
- `DCMAKE_CUDA_HOST_COMPILER`: Compiler used from device code compiler to compile host code [30]

*4.4.4 Benchmark.* Finally, hyperfine is called to benchmark the compilation as shown in listing 5. In the following, the parameters of this call are explained in depth.

```
hyperfine 'make' --warmup 5  --prepare 'make clean' --export-csv
↪ './benchmarkCompilation.csv' --export-json
↪ './benchmarkCompilation.json' #Benchmarked make call
```

Listing 5. Hyperfine benchmark call in Clang/NVCC compilation scripts, line 36

(1) The first parameter is the file path to the application to benchmark, here the `make` program.
(2) The `--warmup` parameter allows to perform a specified number of runs before the program is benchmarked [31]. This excludes potential statistical outliers in the first runs caused by unfilled HPC disk caches [31]. Here 5 warmup runs are performed for the experiment.
(3) The `--prepare` parameter enables to run another program before each benchmarking run [31]. This can be used to ensure preconditions before each benchmarking run [31]. The program `make` potentially only recompiles changed input files. After the first make call which builds everything from scratch, every call could perform faster reusing previously created output files. With `--prepare 'make clean'` all compile output files are deleted before each benchmarking run. This also ensures that the binary files remain in the build folders after the last run for further analysis.
(4) The last parameters `--export-csv` and `--export-json` specify the filename where the benchmark results are saved after completion as CSV and JSON files. [31]
(5) The number of runs can be specified with `--runs` [31]. If not specified, it is automatically choosen such that the results are statistically significant [31]. For the compilation scripts, 9-10 runs have been executed for each benchmark.

## 4.5 Execution scripts

In this subsection, the conception of the execution scripts is explained in detail.

*4.5.1 Preconditions.* There have to be ensured two preconditions before the execution scripts can be runned on JURECA.

(1) *The compilation scripts have been executed*
This is necessary because the compilation scripts generate the binaries which execution time is benchmarked by the execution scripts. Therefore if the precondition is violated, there would exists no binary which the execution script could benchmark.
(2) *GPU utilization is enabled for JURECA*
The second is needed because HPC often can accelerate execution time using GPUs if the application supports this. Therefore if this precondition is violated, the benchmark results would not show the realistically achievable speed using GPUs. To enable GPU utilization for JURECA, a back-end node have to be allocated first [32]. This is done with `salloc` as shown in listing 6 [32]. Here one backend node is allocated, with one job per node and 4 GPU kernels [32]. After completion, another shell opens [32] in which the execution script can be started.

```
salloc --nodes=1 --ntasks-per-node=1 --gpus-per-task=4
↪  --partition=dc-gpu-devel --account=exalab
```

Listing 6. Allocation of back-end node resources on JURECA

*4.5.2 Overview.* Both scripts also define the sample directory paths and load the modules Clang, CMake and CUDA. It then defines the function `Benchmark` which takes sample directory path, optimization flag and sample binary filename as parameters. The overall conception is visualized in figure 3, including the call structure and function `Benchmark`. Finally this function is called for all combinations of samples and optimization flags. It first opens the build directory. Then it benchmarks the execution of the sample binary file which is explained in the following.

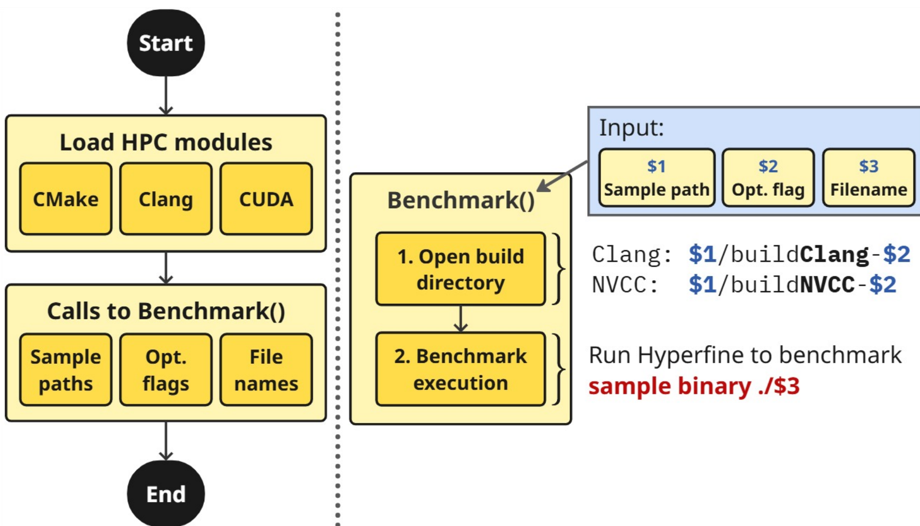



Fig. 3. Conception of execution benchmark scripts. Left: Call structure, right: Benchmark function.

*4.5.3 Benchmark.* The `Benchmark` function executes hyperfine for benchmarking the execution time of all sample binaries. The call is shown in listing 7.

```
srun hyperfine --warmup 5 ./$3 --export-csv './benchmarkExecution.csv'
↪  --export-json './benchmarkExecution.json'
```

Listing 7. Hyperfine benchmark call from Clang/NVCC execution script, line 35

(1) To ensure it runs on the back-end node and utilizes GPUs, the call is prefixed with `srun` [32].
(2) Here also 5 warmup runs are performed before benchmarking [31].
(3) The results are also exported into the build folders as CSV and JSON files [31].

## 5 Findings

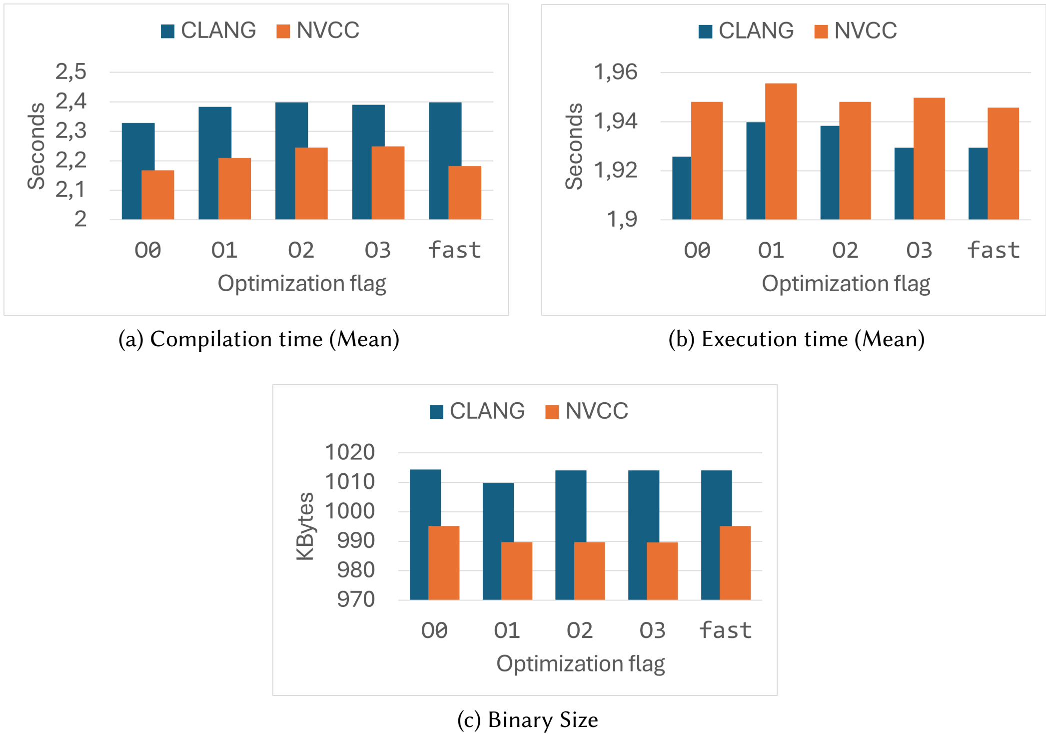
This chapter shows the benchmark results of the experiment, grouped in subsections by sample selection criteria. It also addresses analysis of the results.

*5.0.1 Important details.* Because benchmarking `-Ofast` gives an error using NVCC, it was benchmarked the flag `-ffast-math` for NVCC instead. Because of that the diagrams contain the description "`fast`" for this flag. All scales range around minimum and maximum average benchmark values.

### 5.1 Basic

This subsection begins with results of the basic samples.

*5.1.1 Printf.* First, the benchmarks for printf are presented. NVCC is significantly faster than Clang for compilation and execution. Also NVCC generates significantly smaller binary files. The benchmarks are depicted in figure 4.




(a) Compilation time (Mean)

(b) Execution time (Mean)

(c) Binary Size

Fig. 4. Benchmarks for printf

The diagram scale for compilation ranges from 2 up to 2,5 seconds. Clang compiles up to 9% slower than NVCC. For execution, the diagram scale ranges from 1,9 up to 1,96 seconds. The execution time for binaries compiled with Clang is 1% faster than for those compiled with NVCC. The scale for binary size ranges between 970 and 1020 kilobytes. The Clang binary files are 2% larger than those compiled with NVCC.

*5.1.2 Scalar product.* Second, the scalar product benchmarks are shown. NVCC compiles faster and also generates smaller binary files. On the other hand, Clang executables run faster. In figure 5 the results are collected.

(a) Compilation time (Mean)

(b) Execution time (Mean)

(c) Binary Size

Fig. 5. Benchmarks for scalar product

The diagram scale for compilation ranges from 4,2 up to 5 seconds. Clang compiles up to 6% faster than NVCC. For execution, the diagram scale is between 1,94 and 2 seconds. Clang executables run 1% faster than those generated with NVCC. The scale for the binary size diagram lays between 980 up to 1060 kilobytes. Binary files created with Clang are 2% larger than those created with NVCC.

## 5.2 Medium

After the basic samples, in this section the medium samples are introduced.

*5.2.1 Matrix multiplication.* Figure 6 shows the benchmarks for matrix multiplication. NVCC compiles faster than Clang and also creates smaller binaries. The execution time is significantly faster for Clang in O0 without optimizations, but slower in all other levels.

Fig. 6. Benchmarks for matrix multiplication

For `-O0` up to `-O3`, NVCC compilation time is up to 8% faster than Clang compilation time. For `-Ofast` and `-ffast-math` it is x% faster. The execution time ranges in the diagram from 1,8 up to 2,4 seconds. For `-O0`, Clang is 12% slower than NVCC. For all other optimization flags, Clang is 1% faster than NVCC. Binaries created with Clang are up to 3% smaller than those created with NVCC.

*5.2.2 Vector addition.* Figure 7 shows the benchmark results for vector addition. NVCC is faster with compilation and execution of the sample and creates smaller binary files than Clang.

(a) Compilation time (Mean)

(b) Execution time (Mean)

(c) Binary Size

Fig. 7. Benchmarks for vector addition

For compilation, the diagram scale ranges between 1,8 and 2 seconds. The compilation time differs up to 3% between Clang and NVCC. The execution time diagram scale is between 1,9 and 1,96 seconds. The execution time of Clang executables is up to 3% faster than those of NVCC. For binary size, the diagram scale lays between 960 and 1020 kilobytes. The Clang executable file sizes are up to 3% larger than those of NVCC.

### 5.3 Complex

Following on the medium samples, this chapter presents the complex samples.

*5.3.1 Reduction.* Next, the reduction sample results are presented. They are depicted in figure 8. For `-O0` up to `-O3` Clang compiles faster, and for `-Ofast` and `--fast-math` NVCC compiles slightly faster. The execution time for NVCC binary is significantly faster for `-O0` and also faster for `-O3` and `--fast-math` compared to Clang. For the other flags `-O1` and `-O2` Clang executable is slightly faster. Clang creates significantly smaller binaries than NVCC.

(a) Compilation time (Mean)

(b) Execution time (Mean)

(c) Binary Size

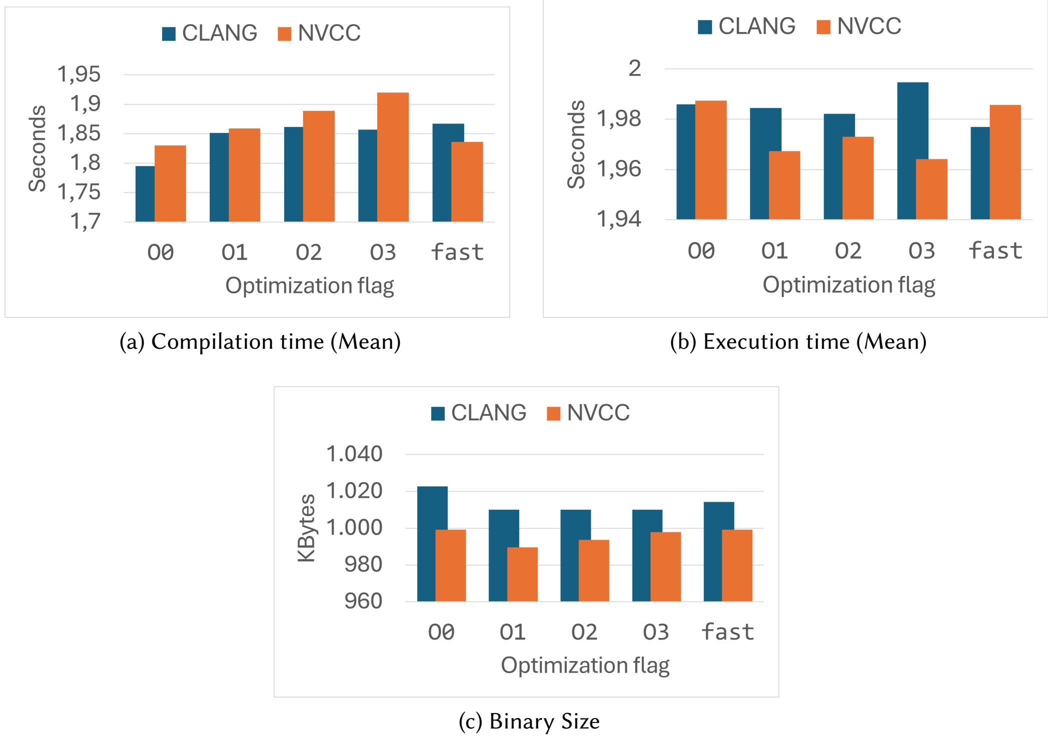
Fig. 8. Benchmarks for reduction

The scale range for compilation time is between 6 and 12 seconds. The compilation time of Clang for `-O0` is 11% faster than NVCC. For `-O1` up to `-O3` compilation takes 10 up to 13% longer than with NVCC. For `-Ofast` compared with `-ffast-math`, compilation time using Clang is 3% faster than NVCC. For execution time, the diagram scales from 2,38 up to 2,44 seconds. The diagram scale for binary size is between 0 and 4000 kilobytes. The percentage differences are very significant, for `-O0` and `-O1` Clang executables are up to 58% bigger than those generated with NVCC. For `-O2` and `-O3` Clang executables are 45% bigger than those created with NVCC. And for `-Ofast` compared to `-ffast-math` binary files created with Clang are 49% bigger than NVCC generated ones.

### 5.4 Real-world

Finally, in this subsection the results for the real-world example are presented.

*5.4.1 Jacobi solver.* Finally, the benchmarks for the jacobi solver real-world application are shown in figure 9. NVCC compiles faster for `-O0` up to `-O3`. The execution time is for `-O0` and `-Ofast` faster with Clang, but for `-O1` up to `-O3` faster for NVCC. NVCC also creates smaller binary files.




(a) Compilation time (Mean)

(b) Execution time (Mean)

(c) Binary Size

Fig. 9. Benchmarks for jacobi solver

The compilation time for `-O0` up to `-O3` is up to 3% faster when using Clang opposed to NVCC. For `-Ofast` compared to `-ffast-math` Clang is 2% slower than NVCC. The execution time takes up to 2% longer when using Clang rather than NVCC. The binary size is up to 2% larger using Clang compared to NVCC.

### 5.5 Analysis

After the benchmarks for all samples have been shown, in this subsection the results are analyzed to answer the research questions of this seminar thesis. Also recommendations are given when to use which compiler.

*5.5.1 Research questions.* Clang and NVCC differ in performance of compilation time, execution time and binary size of the selected samples. For simple and medium samples, this difference is between 1-2 and 8-9%. The complex sample reduction and the real-world example jacobi solver benchmarks revealed even larger differences. Reasons for this difference could be the different compilation methods of both compilers as described in the fundamentals, split compilation versus merged parsing. Also the different single optimization techniques collected in the compared optimization bundle flags are suspected to have an performance impact.

*5.5.2 Recommendations.* No clear recommendation for one of both compilers can be given, it depends on the compiled program, the used compiler and optimization flag. The basic samples printf and scalar product are executed faster when compiled with Clang. For the medium samples, matrix multiplication should definitely compiled at least with `-O1` to execute fast and there is no

big execution time difference between Clang and NVCC. For vector addition, Clang should be used for compilation because created executables run faster than those compiled with NVCC. For the complex sample and the real-world example, there are highly different benchmark values for each optimization stage. The fastest execution of reduction can be achieved when compiling with NVCC using `-ffast-math`. For the jacobi solver, NVCC should be used for compilation with `-O3`.

# 6 Future work

The seminar thesis concludes with future work which recommends related topics relevant for being studied in further research.

## 6.1 Hardware dependence

The experiment was tied to specific hardware, more precise the HPC system JURECA and NVIDIA GPUs.

*6.1.1 HPC system.* The experiment was runned on the HPC cluster JURECA and the results are therefore based on this hardware. To understand the HPC system impact, the experiment should be repeated on different HPC clusters as well. Hereby it can be observed if the benchmarks behave different or similiar on other HPC systems.

*6.1.2 GPU vendor.* Also the experiment runned with CUDA which is used to program NVIDIA GPUs. In the future, other experiments are recommended to research the problem for other GPU vendors.

## 6.2 Compiler properties

The properties of both compared compilers could have also an performance impact, namely compiler optimization flags and the technique used to compile CUDA code.

*6.2.1 Optimization levels.* Because the compiler optimizations of both compilers are only analyzed for optimization level bundles, the benchmark results cannot be tracked back to causing optimizations. The experiment should be repeated for single optimization flags for a better understanding of these single optimizations to applications performance.

*6.2.2 Compilation technique.* Beside optimizations, also the technique of both compilers used for compilation could have an impact on performance itself. Hence it is recommended to shed the light on performance differences caused by split compilation versus merged parsing CUDA compilation techniques in future research.

## 6.3 Samples

Concerning samples, future work is recommended related to their scope and input size.

*6.3.1 Scope.* Because of the limited number of analyzed samples, the findings may not be representative. The experiment should be repeated with a larger amount of samples.

*6.3.2 Input size.* The samples were executed with fixed size data inputs. Small input sizes can introduce higher benchmark variability due to higher influence of the execution overhead [18]. The experiment should be repeated with inputs of different sizes for each sample. Hereby the effects of scaling input sizes can be studied.

## 7 Summary

This chapter finally summarizes the experiment plan and the main findings.
In the context of HPC computing, high performance is important for applications because they often contain computational-intensive operations. Compiler can apply several optimizations and thereby are suspected to impact the application performance. This seminar thesis focuses on NVIDIA CUDA applications and compares the compilation time when using NVCC or Clang, as well as execution time and file size of the generated binaries. 5 samples from NVIDIA CUDA Samples repository and a real-world application are selected and their performance was benchmarked for five optimization stages `-O0` up to `-O3`, `-Ofast` and `-ffast-math` on the JURECA HPC system at the Jülich supercomputing centre (Germany). To automate this process, compilation and benchmark scripts have been written for Clang and NVCC and a download script to download the results. The results show that there is no general recommendation that can be given in order to achieve fastest execution time for the samples. Simple samples, which have low algorithmic complexity, run fastest when compiled with Clang. In future work, hardware dependence should be researched like the impact of the HPC system and GPU vendor. Also further investigation should be taken of compiler properties like optimization levels and compilation techniques. The sample scope and input size should be varied in future research.

## References


[1] 2025. Configuration — jureca user documentation documentation. Retrieved May 03, 2025 from https://apps.fz-juelich.de/jsc/hps/jureca/configuration.html.

[2] David B Kirk and W Hwu Wen-Mei. 2016. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.

[3] 2025. Compiling cuda with clang — llvm 21.0.0git documentation. Retrieved July 30, 2025 from https://llvm.org/docs/CompileCudaWithLLVM.html.

[4] Jeffrey D. Ullman, Monica S. Lam, Ravi Sethi, and Alfred V. Aho. 2008. *Compiler Prinzipien, Techniken und Werkzeuge*. Pearson Deutschland, 1296. ISBN: 9783827370976. https://elibrary.pearson.de/book/99.150005/9783863265748.

[5] 2025. Dead code elimination. Retrieved May 04, 2025 from https://cran.r-project.org/web/packages/rco/vignettes/opt-dead-code.html.

[6] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 977–989.

[7] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. 2017. Chapter 5 - source code transformations and optimizations. In *Embedded Computing for High Performance*. João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz, (Eds.) Morgan Kaufmann, Boston, 137–183. ISBN: 978-0-12-804189-5. doi:https://doi.org/10.1016/B978-0-12-804189-5.00005-3.

[8] Jing Ge Feng, Ye Ping He, and Qiu Ming Tao. 2021. Evaluation of compilers' capability of automatic vectorization based on source code analysis. *Scientific Programming*, 2021, 1, 3264624.

[9] 2025. Assignment 5. Retrieved May 04, 2025 from https://www.cs.utexas.edu/~pingali/CS380C/2020/assignments/assignment5/index.html.

[10] Brian L Deitrich, Ben Chung Chen, and WW Hwu. 1998. Improving static branch prediction in a compiler. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*. IEEE, 214–221.

[11] 2025. 1. introduction — floating point and ieee 754 13.0 documentation. Retrieved May 04, 2025 from https://docs.nvidia.com/cuda/floating-point/index.html.

[12] Bruno Buchberger and Rüdiger Loos. 1982. Algebraic simplification. In *Computer algebra: symbolic and algebraic computation*. Springer, 11–43.

[13] Xavier Lapillonne, Katherine Osterried, and Oliver Fuhrer. 2017. Chapter 13 - using openacc to port large legacy climate and weather modeling code to gpus. In *Parallel Programming with OpenACC*. Rob Farber, (Ed.) Morgan Kaufmann, Boston, 267–290. ISBN: 978-0-12-410397-9. doi:https://doi.org/10.1016/B978-0-12-410397-9.00013-5.

[14] Wen-Mei W Hwu and Pohua P Chang. 1989. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 242–251.

[15] 2025. Clang - the clang c, c++, and objective-c compiler — clang 22.0.0git documentation. Retrieved May 04, 2025 from https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options.

[16] 2025. Clang optimization levels - stack overflow. Retrieved May 04, 2025 from https://stackoverflow.com/questions/15548023/clang-optimization-levels.

[17] 2025. Nvidia hpc compilers reference guide — nvidia hpc compilers reference guide 25.7 documentation. Retrieved May 04, 2025 from https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-ref-guide/index.html.

[18] Vladimir Marjanović, José Gracia, and Colin W. Glass. 2016. *Proceedings of PMBS 2016: 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems: Held in conjunction with SC16: the International Conference for High Performance Computing, Networking, Storage and Analysis : Salt Lake City, Utah, November 13-18, 2016*. IEEE, Piscataway, NJ. ISBN: 9781509052189. http://ieeexplore.ieee.org/servlet/opac?punumber=7835831.

[19] Paul B Schneck. 1973. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference*, 106–113.

[20] Xuechao Li and Po-Chou Shih. 2018. An early performance comparison of cuda and openacc. *MATEC Web of Conferences*, 208, 05002. doi:10.1051/matecconf/201820805002.

[21] 2025. GitHub - nvidia/cuda-samples: samples for cuda developers which demonstrates features in cuda toolkit. Retrieved July 30, 2025 from https://github.com/NVIDIA/cuda-samples.

[22] 2025. Access — jureca user documentation documentation. Retrieved May 03, 2025 from https://apps.fz-juelich.de/jsc/hps/jureca/access.html.

[23] 2025. GitHub - sharkdp/hyperfine: A command-line benchmarking tool — github.com. Retrieved July 30, 2025 from https://github.com/sharkdp/hyperfine.

[24] [SW] David Peter, hyperfine version 1.16.1, Mar. 2023. URL: https://github.com/sharkdp/hyperfine.

[25] 2025. Cmake(1) — cmake 4.1.0-rc4 documentation. Retrieved July 30, 2025 from https://cmake.org/cmake/help/latest/manual/cmake.1.html.

[26] 2025. Retrieved May 04, 2025 from https://cmake.org/cmake/help/latest/variable/CMAKE_CUDA_ARCHITECTURES.html.

[27] 2025. Retrieved May 04, 2025 from https://cmake.org/cmake/help/latest/variable/CMAKE_LANG_FLAGS.html.

[28] 2025. Retrieved May 04, 2025 from https://cmake.org/cmake/help/latest/envvar/CUDAFLAGS.html.

[29] 2025. Retrieved May 04, 2025 from https://cmake.org/cmake/help/latest/variable/CMAKE_LANG_COMPILER.html.

[30] 2025. Retrieved May 04, 2025 from https://cmake.org/cmake/help/latest/variable/CMAKE_LANG_HOST_COMPILER.html.

[31] 2025. Ubuntu manpage: hyperfine - hyperfine. Retrieved May 04, 2025 from https://manpages.ubuntu.com/manpages/jammy/man1/hyperfine.1.html.

[32] 2025. Batch system — jureca user documentation documentation. Retrieved May 03, 2025 from https://apps.fz-juelich.de/jsc/hps/jureca/batchsystem.html.

## A Benchmark scripts

In this appendix the measurement scripts are presented in full detail.

### A.1 Clang: Compilation script

```
#!/bin/bash -x

# Sample directory paths
intro='/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples/Samples/0_Introduction'
concepts='/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples/Samples/2_Concepts_and_Techniques'

simplePrintf=$intro/simplePrintf
matrixMul=$intro/matrixMul
vectorAdd=$intro/vectorAdd
scalarProd=$concepts/scalarProd
reduction=$concepts/reduction
jacobi='/p/home/jusers/ctistis1/jureca/Data/Jacobi'

# Add Hyperfine to PATH
export PATH=$PATH:/p/home/jusers/ctistis1/jureca/Data/Hyperfine

# Run benchmarks for all optimization levels
# $1: Project directory path
benchmarkSample() {
  benchmark $1 -O0
  benchmark $1 -O1
  benchmark $1 -O2
  benchmark $1 -O3
  benchmark $1 -Ofast
}

# Compilation time benchmark
# $1: Project directory path
# $2: Optimization level
benchmark() {
  cd $1
  rm -rf ./buildClang$2
  mkdir ./buildClang$2
  cd ./buildClang$2
  cmake -DCMAKE_CUDA_ARCHITECTURES=80 -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_CUDA_COMPILER=clang++ -DCMAKE_CUDA_HOST_COMPILER=clang++ -DCMAKE_CUDA_FLAGS=$2 ..
  hyperfine 'make' --warmup 5  --prepare 'make clean' --export-csv './benchmarkCompilation.csv' --export-json './benchmarkCompilation.json' #Benchmarked make call
}
```

```
38
39 # Load used modules
40 module use /p/project1/cexalab/easybuild/jurecadc/modules/all/Compiler
   ↪ /sidecompiler/GCCcore/13.3.0/
41 module load Clang/19.1.7 CUDA CMake
42
43 # Call compilation time benchmarks for each project:
44 benchmarkSample $simplePrintf
45 benchmarkSample $matrixMul
46 benchmarkSample $vectorAdd
47 benchmarkSample $scalarProd
48 benchmarkSample $reduction
49 benchmarkSample $jacobi
```

### A.2 Clang: Execution script

```
1  #!/bin/bash -x
2
3  # Sample directory paths
4  intro='/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples/Samples
   ↪ /0_Introduction'
5  concepts='/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples
   ↪ /Samples/2_Concepts_and_Techniques'
6
7  simplePrintf=$intro/simplePrintf
8  matrixMul=$intro/matrixMul
9  vectorAdd=$intro/vectorAdd
10 scalarProd=$concepts/scalarProd
11 reduction=$concepts/reduction
12 jacobi='/p/home/jusers/ctistis1/jureca/Data/Jacobi'
13
14 # Add Hyperfine to PATH
15 export PATH=$PATH:/p/home/jusers/ctistis1/jureca/Data/Hyperfine
16
17 # Run benchmarks for all optimization levels
18 # $1: Project directory path
19 # $2: Executable filename
20 benchmarkSample() {
21   benchmark $1 -O0 $2
22   benchmark $1 -O1 $2
23   benchmark $1 -O2 $2
24   benchmark $1 -O3 $2
25   benchmark $1 -Ofast $2
26 }
27
28 # Execution time benchmark
29 # $1: Project directory path
30 # $2: Optimization level
```

```
# $3: Executable filename
benchmark() {
  cd $1
  cd buildClang$2
  srun hyperfine --warmup 5 ./$3 --export-csv './benchmarkExecution.csv'
   ↪ --export-json './benchmarkExecution.json'
}
# Load used modules
module use /p/project1/cexalab/easybuild/jurecadc/modules/all/Compiler⌋
↪ /sidecompiler/GCCcore/13.3.0/
module load Clang/19.1.7 CUDA CMake

# Call execution time benchmarks for each project:
benchmarkSample $simplePrintf simplePrintf
benchmarkSample $matrixMul matrixMul
benchmarkSample $vectorAdd vectorAdd
benchmarkSample $scalarProd scalarProd
benchmarkSample $reduction reduction
benchmarkSample $jacobi jacobi
```

### A.3 NVCC: Compilation script

```
#!/bin/bash -x

# Sample directory paths
intro='/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples/Samples⌋
↪ /0_Introduction'
concepts='/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples⌋
↪ /Samples/2_Concepts_and_Techniques'

simplePrintf=$intro/simplePrintf
matrixMul=$intro/matrixMul
vectorAdd=$intro/vectorAdd
scalarProd=$concepts/scalarProd
reduction=$concepts/reduction
jacobi='/p/home/jusers/ctistis1/jureca/Data/Jacobi'

# Add Hyperfine to PATH
export PATH=$PATH:/p/home/jusers/ctistis1/jureca/Data/Hyperfine

# Run benchmarks for all optimization levels
# $1: Project directory path
benchmarkSample() {
  benchmark $1 -O0
  benchmark $1 -O1
  benchmark $1 -O2
  benchmark $1 -O3
  benchmark $1 '-ffast-math'
```

```
25 }
26
27 # Compilation time benchmark
28 # $1: Project directory path
29 # $2: Optimization level
30 benchmark() {
31   cd $1
32   rm -rf ./buildNVCC$2
33   mkdir ./buildNVCC$2
34   cd buildNVCC$2
35   cmake -DCMAKE_CUDA_ARCHITECTURES=80 -DCMAKE_CUDA_FLAGS=$2 ..
36   hyperfine 'make' --warmup 5 --prepare 'make clean' --export-csv
   ↪  './benchmarkCompilation.csv' --export-json
   ↪  './benchmarkCompilation.json' #Benchmarked make call
37 }
38
39 # Load used modules
40 module use /p/project1/cexalab/easybuild/jurecadc/modules/all/Compiler
   ↪  /sidecompiler/GCCcore/13.3.0/
41 module load Clang/19.1.7 CUDA CMake
42
43 # Call compilation time benchmarks for each project:
44 benchmarkSample $simplePrintf
45 benchmarkSample $matrixMul
46 benchmarkSample $vectorAdd
47 benchmarkSample $scalarProd
48 benchmarkSample $reduction
49 benchmarkSample $jacobi
```

### A.4 NVCC: Execution script

```
1 #!/bin/bash -x
2
3 # Sample directory paths
4 intro='/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples/Samples
  ↪  /0_Introduction'
5 concepts='/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples
  ↪  /Samples/2_Concepts_and_Techniques'
6
7 simplePrintf=$intro/simplePrintf
8 matrixMul=$intro/matrixMul
9 vectorAdd=$intro/vectorAdd
10 scalarProd=$concepts/scalarProd
11 reduction=$concepts/reduction
12 jacobi='/p/home/jusers/ctistis1/jureca/Data/Jacobi'
13
14 # Add Hyperfine to PATH
15 export PATH=$PATH:/p/home/jusers/ctistis1/jureca/Data/Hyperfine
```

```
16
17 # Run benchmarks for all optimization levels
18 # $1: Project directory path
19 # $2: Executable filename
20 benchmarkSample() {
21   benchmark $1 -O0 $2
22   benchmark $1 -O1 $2
23   benchmark $1 -O2 $2
24   benchmark $1 -O3 $2
25   benchmark $1 -Ofast $2
26 }
27
28 # Execution time benchmark
29 # $1: Project directory path
30 # $2: Optimization level
31 # $3: Executable filename
32 benchmark() {
33   cd $1
34   cd buildNVCC$2
35   srun hyperfine --warmup 5 ./$3 --export-csv './benchmarkExecution.csv'
    ↪  --export-json './benchmarkExecution.json'
36 }
37 # Load used modules
38 module use /p/project1/cexalab/easybuild/jurecadc/modules/all/Compiler⌋
   ↪  /sidecompiler/GCCcore/13.3.0/
39 module load Clang/19.1.7 CUDA CMake
40
41 # Call execution time benchmarks for each project:
42 benchmarkSample $simplePrintf simplePrintf
43 benchmarkSample $matrixMul matrixMul
44 benchmarkSample $vectorAdd vectorAdd
45 benchmarkSample $scalarProd scalarProd
46 benchmarkSample $reduction reduction
47 benchmarkSample $jacobi jacobi
```

## A.5 Download script

```
1 ::Sample directory paths
2 set intro="/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples⌋
  ↪  /Samples/0_Introduction"
3 set concepts="/p/home/jusers/ctistis1/jureca/Data/CUDAsamples/cuda-samples⌋
  ↪  /Samples/2_Concepts_and_Techniques"
4 set dest="C:\Users\tctis\Benchmarks"
5
6 set simplePrintf=%intro%/simplePrintf
7 set matrixMul=%intro%/matrixMul
8 set vectorAdd=%intro%/vectorAdd
9 set scalarProd=%concepts%/scalarProd
```

```
set reduction=%concepts%/reduction
set jacobi="/p/home/jusers/ctistis1/jureca/Data/Jacobi"

::HPC server configuration
set scpConfig=-i ~/.ssh/juelich-key -o MACs=hmac-sha2-512-etm@openssh.com
set hpcLogin=ctistis1@jureca.fz-juelich.de

@echo off
:copyBenchmarks
  scp %scpConfig% -r %hpcLogin%:%simplePrintf% %dest%
  scp %scpConfig% -r %hpcLogin%:%matrixMul% %dest%
  scp %scpConfig% -r %hpcLogin%:%vectorAdd% %dest%
  scp %scpConfig% -r %hpcLogin%:%scalarProd% %dest%
  scp %scpConfig% -r %hpcLogin%:%reduction% %dest%
  scp %scpConfig% -r %hpcLogin%:%jacobi% %dest%

```

## B Benchmark results

This appendix contains the obtained benchmark results produced by the conducted experiment in full detail. This includdes minimum, maximum, average and standard deviation of each benchmarking run.

### B.1 Compilation

All values in seconds.

| CLANG-O0 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,32838748 | 0,01164245 | 2,329314481 | 2,30922937 | 2,343596144 |
| MatrixMul | 2,357230675 | 0,012751377 | 2,358641957 | 2,338806055 | 2,384333575 |
| VectorAdd | 1,868145646 | 0,009431557 | 1,866402367 | 1,855261762 | 1,891040427 |
| ScalarProd | 4,657997206 | 0,012300339 | 4,656122577 | 4,644132794 | 4,681579513 |
| Reduction | 10,28589757 | 0,044980102 | 10,27358612 | 10,23486349 | 10,38643425 |
| Jacobi | 1,794832452 | 0,010842605 | 1,793242124 | 1,810830834 | 1,781224146 |

| NVCC-O0 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,167718859 | 0,007030259 | 2,166366335 | 2,158269184 | 2,182056528 |
| MatrixMul | 2,223605831 | 0,00969267 | 2,222977059 | 2,210830936 | 2,238619007 |
| VectorAdd | 1,931905563 | 0,012719606 | 1,931628424 | 1,915033281 | 1,950925457 |
| ScalarProd | 4,496500607 | 0,017638059 | 4,498632867 | 4,468322514 | 4,522409971 |
| Reduction | 9,127819587 | 0,666999704 | 8,91709381 | 8,895657745 | 11,02564311 |
| Jacobi | 1,830078132 | 0,008360979 | 1,830888549 | 1,817333309 | 1,839930946 |

| CLANG-O1 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,38247863 | 0,012051655 | 2,379816547 | 2,368377196 | 2,40571662 |
| MatrixMul | 2,489203031 | 0,007803331 | 2,490026278 | 2,478244494 | 2,501132806 |
| VectorAdd | 1,904708745 | 0,015651582 | 1,902042072 | 1,88342372 | 1,933557237 |
| ScalarProd | 4,786089435 | 0,09255532 | 4,765782473 | 4,709076101 | 5,035708333 |
| Reduction | 8,373641522 | 0,051156217 | 8,366589143 | 8,298676493 | 8,484014854 |
| Jacobi | 1,851476098 | 0,010178644 | 1,85008826 | 1,837522551 | 1,876081761 |

| NVCC-O1 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,208821706 | 0,008430935 | 2,209315918 | 2,198058984 | 2,221763758 |
| MatrixMul | 2,300171867 | 0,011267246 | 2,299006749 | 2,283440446 | 2,323245181 |
| VectorAdd | 1,935722323 | 0,021567288 | 1,932513829 | 1,903744569 | 1,975508211 |
| ScalarProd | 4,560020179 | 0,016175356 | 4,5541166 | 4,541311419 | 4,585682214 |
| Reduction | 9,244829802 | 0,010482934 | 9,247922227 | 9,221577085 | 9,256443077 |
| Jacobi | 1,859180511 | 0,008811074 | 1,860687391 | 1,841723051 | 1,869781446 |

---

| CLANG-O2 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,398227974 | 0,010701524 | 2,396709493 | 2,383587633 | 2,41655971 |
| MatrixMul | 2,530781179 | 0,013004529 | 2,53153846 | 2,511678929 | 2,555165387 |
| VectorAdd | 1,898969456 | 0,009794532 | 1,903468066 | 1,885935921 | 1,909150991 |
| ScalarProd | 4,780307423 | 0,024374759 | 4,772888508 | 4,739525893 | 4,82243464 |
| Reduction | 9,106163454 | 0,050709124 | 9,100344414 | 9,021430592 | 9,1863216 |
| Jacobi | 1,861637069 | 0,010287967 | 1,860297058 | 1,849709449 | 1,882977152 |

| NVCC-O2 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,244596227 | 0,012310585 | 2,242823797 | 2,227013328 | 2,269811306 |
| MatrixMul | 2,364492158 | 0,011205101 | 2,361148794 | 2,35160047 | 2,384906289 |
| VectorAdd | 1,935619383 | 0,019635728 | 1,929646239 | 1,921338809 | 1,989173203 |
| ScalarProd | 4,631202409 | 0,031884473 | 4,621760501 | 4,591641919 | 4,68787179 |
| Reduction | 9,80216025 | 0,018928219 | 9,803284115 | 9,768141902 | 9,831636677 |
| Jacobi | 1,888856115 | 0,009300225 | 1,888847514 | 1,87134118 | 1,902853711 |

---

| CLANG-O3 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,39021589 | 0,010070374 | 2,387358454 | 2,378770801 | 2,407146685 |
| MatrixMul | 2,539196186 | 0,014379442 | 2,536921152 | 2,522200604 | 2,56804622 |
| VectorAdd | 1,909400189 | 0,008814883 | 1,909068496 | 1,897084403 | 1,925874077 |
| ScalarProd | 4,805518395 | 0,020884205 | 4,804953473 | 4,782314962 | 4,853190615 |
| Reduction | 9,175162217 | 0,063534422 | 9,173495564 | 9,098425595 | 9,306795642 |
| Jacobi | 1,857168052 | 0,011591495 | 1,853280259 | 1,844372738 | 1,882681733 |

| NVCC-O3 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,249033187 | 0,011776816 | 2,249733568 | 2,23070054 | 2,269372109 |
| MatrixMul | 2,397113578 | 0,02893231 | 2,388496571 | 2,37788776 | 2,476134377 |
| VectorAdd | 1,950077134 | 0,008841248 | 1,949265159 | 1,932433601 | 1,963733596 |
| ScalarProd | 4,633076244 | 0,015867149 | 4,627837555 | 4,610091622 | 4,664693872 |
| Reduction | 10,35264168 | 0,097389066 | 10,33099205 | 10,27078791 | 10,60705261 |
| Jacobi | 1,919896639 | 0,068522935 | 1,900787888 | 1,87663331 | 2,11296 |

---

| CLANG-Ofast | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,398188998 | 0,009981683 | 2,396927287 | 2,384808352 | 2,421069995 |
| MatrixMul | 2,590833012 | 0,016612559 | 2,589785698 | 2,569377271 | 2,628301491 |
| VectorAdd | 1,897331322 | 0,011973744 | 1,894686818 | 1,874999788 | 1,921509792 |
| ScalarProd | 4,798982112 | 0,019063642 | 4,798960501 | 4,769258351 | 4,830438673 |
| Reduction | 9,188675654 | 0,082843749 | 9,152577962 | 9,085774186 | 9,329339234 |
| Jacobi | 1,867121727 | 0,011481438 | 1,862291066 | 1,851585484 | 1,884649082 |

| NVCC-ffast-math | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 2,182109391 | 0,010512796 | 2,180385025 | 2,168615868 | 2,198574039 |
| MatrixMul | 2,236753497 | 0,015637847 | 2,235552309 | 2,218035042 | 2,275133201 |
| VectorAdd | 1,920626854 | 0,029252912 | 1,911117473 | 1,888301803 | 1,983003329 |
| ScalarProd | 4,512150171 | 0,023901335 | 4,508827135 | 4,477006752 | 4,565730965 |
| Reduction | 8,921541281 | 0,067694843 | 8,902021827 | 8,871449807 | 9,10886589 |
| Jacobi | 1,836296255 | 0,01108813 | 1,834618594 | 1,820642456 | 1,859759211 |

### B.2 Execution

All values in seconds.

| CLANG-O0 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,925763695 | 0,013669354 | 1,923804947 | 1,907046051 | 1,947163124 |
| MatrixMul | 2,216891758 | 0,005221127 | 2,21603484 | 2,209138427 | 2,227814065 |
| VectorAdd | 1,923541891 | 0,009143719 | 1,924758662 | 1,908114732 | 1,935967697 |
| ScalarProd | 1,968365849 | 0,007133873 | 1,96798778 | 1,955223119 | 1,980714861 |
| Reduction | 2,424854554 | 0,022665525 | 2,427094661 | 2,39369863 | 2,455946273 |
| Jacobi | 1,98594546 | 0,013596452 | 1,98496808 | 1,967921856 | 2,019385123 |

| NVCC-O0 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,948055328 | 0,008331605 | 1,94877 | 1,935014791 | 1,958469564 |
| MatrixMul | 1,957975608 | 0,011663492 | 1,961351903 | 1,93028725 | 1,96740842 |
| VectorAdd | 1,942607348 | 0,007029444 | 1,94306007 | 1,928452789 | 1,952921771 |
| ScalarProd | 1,98937059 | 0,018962206 | 1,986150067 | 1,967773828 | 2,035130521 |
| Reduction | 2,410433835 | 0,008494373 | 2,410471885 | 2,396858212 | 2,421588326 |
| Jacobi | 1,987414686 | 0,028655585 | 1,978368294 | 1,950233826 | 2,029415751 |

| CLANG-O1 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,939760454 | 0,009542438 | 1,93962541 | 1,92545773 | 1,954098911 |
| MatrixMul | 1,939147049 | 0,009222594 | 1,943207265 | 1,923673137 | 1,947503317 |
| VectorAdd | 1,917732984 | 0,016835144 | 1,921546732 | 1,886703043 | 1,945296792 |
| ScalarProd | 1,962992974 | 0,006488571 | 1,963099176 | 1,954505163 | 1,977614999 |
| Reduction | 2,406416598 | 0,01716782 | 2,404042023 | 2,381744632 | 2,443663674 |
| Jacobi | 1,984538121 | 0,047042218 | 1,965488874 | 1,931022756 | 2,05651439 |

| NVCC-O1 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,955619746 | 0,009541223 | 1,955628386 | 1,938003927 | 1,968360469 |
| MatrixMul | 1,964830717 | 0,007948803 | 1,965508539 | 1,949222603 | 1,974470601 |
| VectorAdd | 1,944182355 | 0,006056668 | 1,944299125 | 1,929721092 | 1,953995167 |
| ScalarProd | 1,986336558 | 0,013161593 | 1,979835094 | 1,977790617 | 2,016454651 |
| Reduction | 2,40800003 | 0,008937118 | 2,406347965 | 2,393302372 | 2,422829027 |
| Jacobi | 1,967371324 | 0,01153612 | 1,965921825 | 1,952430368 | 1,987387031 |

| CLANG-O2 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,938356599 | 0,009749262 | 1,940858601 | 1,921819947 | 1,953015463 |
| MatrixMul | 1,940441718 | 0,012228756 | 1,942734817 | 1,911703188 | 1,954187395 |
| VectorAdd | 1,915724005 | 0,008325306 | 1,91370139 | 1,900564907 | 1,929838272 |
| ScalarProd | 1,963096834 | 0,013530417 | 1,963149099 | 1,943870262 | 1,99244284 |
| Reduction | 2,401475584 | 0,015157911 | 2,403144566 | 2,362442493 | 2,417986854 |
| Jacobi | 1,982245083 | 0,033924786 | 1,994354009 | 1,928643878 | 2,015470327 |

| NVCC-O2 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,948034306 | 0,014355179 | 1,947889009 | 1,927541173 | 1,975673849 |
| MatrixMul | 1,952141481 | 0,012263482 | 1,951634486 | 1,930609829 | 1,968829089 |
| VectorAdd | 1,942019971 | 0,006783224 | 1,940071845 | 1,934115513 | 1,95223592 |
| ScalarProd | 1,985459136 | 0,00670793 | 1,985186571 | 1,974411072 | 1,994099373 |
| Reduction | 2,405288609 | 0,005870607 | 2,404339118 | 2,394909972 | 2,414982705 |
| Jacobi | 1,973057206 | 0,022003178 | 1,963187126 | 1,952470034 | 2,019329094 |

| CLANG-O3 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,929427053 | 0,00850342 | 1,929701173 | 1,917513501 | 1,94761768 |
| MatrixMul | 1,942302292 | 0,005495708 | 1,941477635 | 1,935194131 | 1,95258659 |
| VectorAdd | 1,91493601 | 0,010788849 | 1,916584991 | 1,900695181 | 1,92898519 |
| ScalarProd | 1,965009945 | 0,008671946 | 1,965564298 | 1,952284669 | 1,97772139 |
| Reduction | 2,40691586 | 0,020236348 | 2,408333159 | 2,36037877 | 2,445022725 |
| Jacobi | 1,994718682 | 0,034698229 | 2,007676682 | 1,933659413 | 2,048029746 |

| NVCC-O3 | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,949752219 | 0,014959625 | 1,952266643 | 1,923775997 | 1,972299646 |
| MatrixMul | 1,957643033 | 0,009596867 | 1,957733644 | 1,942639124 | 1,975370536 |
| VectorAdd | 1,941063866 | 0,008434533 | 1,939628909 | 1,924371642 | 1,952597287 |
| ScalarProd | 1,97832441 | 0,010578381 | 1,977001714 | 1,968126432 | 2,006442773 |
| Reduction | 2,404086986 | 0,01304219 | 2,400369583 | 2,391910434 | 2,436005866 |
| Jacobi | 1,964199116 | 0,010439101 | 1,965804514 | 1,945365586 | 1,982518929 |

---

| CLANG-Ofast | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,929445417 | 0,008656111 | 1,930259889 | 1,909753745 | 1,941793866 |
| MatrixMul | 1,940984596 | 0,006843563 | 1,940234564 | 1,931288937 | 1,949732794 |
| VectorAdd | 1,916341817 | 0,013988548 | 1,917243818 | 1,895157534 | 1,945644262 |
| ScalarProd | 1,963099092 | 0,007586498 | 1,963781593 | 1,949865706 | 1,974622812 |
| Reduction | 2,399350807 | 0,018133114 | 2,403258494 | 2,364248642 | 2,427765793 |
| Jacobi | 1,976998802 | 0,037107251 | 1,958477668 | 1,937837933 | 2,031759647 |

| NVCC-ffast-math | MEAN | STDDEV | MEDIAN | MIN | MAX |
|---|---|---|---|---|---|
| SimplePrintf | 1,945722478 | 0,014285281 | 1,946365232 | 1,925983367 | 1,965985121 |
| MatrixMul | 1,960325429 | 0,005461363 | 1,960043324 | 1,951707293 | 1,967925983 |
| VectorAdd | 1,933021435 | 0,011533037 | 1,932897736 | 1,914274145 | 1,948165268 |
| ScalarProd | 1,982040871 | 0,008326815 | 1,982245953 | 1,968006351 | 1,99809255 |
| Reduction | 2,395406689 | 0,015127511 | 2,39627483 | 2,373879111 | 2,421653584 |
| Jacobi | 1,985755623 | 0,026585844 | 1,980337288 | 1,95002314 | 2,025595406 |

---

### B.3 Binary size

All values in Bytes.

| CLANG | O0 | O1 | O2 | O3 | Ofast |
|---|---|---|---|---|---|
| SimplePrintf | 1.014.352 | 1.009.848 | 1.014.048 | 1.014.048 | 1.014.048 |
| MatrixMul | 1.047.464 | 1.026.480 | 1.026.584 | 1.026.584 | 1.026.584 |
| VectorAdd | 1.009.376 | 1.005.232 | 1.005.232 | 1.005.232 | 1.005.232 |
| ScalarProd | 1.041.112 | 1.019.160 | 1.019.264 | 1.023.360 | 1.023.360 |
| Reduction | 1.909.168 | 1.851.824 | 2.027.952 | 2.036.144 | 2.036.144 |
| Jacobi | 1.022.784 | 1.010.112 | 1.010.112 | 1.010.112 | 1.014.264 |

| NVCC | O0 | O1 | O2 | O3 | ffast-math |
|---|---|---|---|---|---|
| SimplePrintf | 995.160 | 989.680 | 989.680 | 989.592 | 995.160 |
| MatrixMul | 1.016.448 | 1.010.560 | 1.010.520 | 1.014.448 | 1.016.448 |
| VectorAdd | 981.896 | 980.704 | 980.704 | 980.704 | 981.896 |
| ScalarProd | 1.021.160 | 1.011.408 | 1.011.480 | 1.011.480 | 1.021.160 |
| Reduction | 3.032.240 | 2.922.384 | 2.939.408 | 2.949.960 | 3.032.240 |
| Jacobi | 999.176 | 989.656 | 993.704 | 997.824 | 999.176 |