DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Performance Assessment of using OpenCL on FPGA Systems for ODE Solvers

Patrick Tobias Haft

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Performance Assessment of using OpenCL on FPGA Systems for ODE Solvers

# Leistungsbewertung von OpenCL auf FPGA Systemen für ODE Löser

| | |
|---|---|
| Author: | Patrick Tobias Haft |
| Supervisor: | Prof. Dr. Martin Schulz |
| Advisor: | Dr. Martin Schreiber |
| Submission Date: | 15.04.2021 |

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.04.2021 Patrick Tobias Haft

# Acknowledgments

First of all, I want to thank my advisor, Dr. Martin Schreiber, for his outstanding support during the thesis. He was always available for any questions and provided me important feedback throughout my work.

Furthermore, I would like to thank Prof. Dr. Martin Schulz, who made it possible to write my thesis at the Chair of Computer Architecture and Parallel Systems at TUM.

A special word of thanks goes to the LMU and, in particular, to Pascal Jungblut, who provided me access to a FPGA system. Without this support, this thesis would not have been possible.

# Abstract

Parameter optimization is a common task in various fields such as computational biology. In these scientific fields, optimization can be, e.g. based on ordinary differential equations with the computational task getting increasingly computation-intensive for increasing complexity of ODE and the parameters to determine. Hence, this raises requirements for an efficient treatment on high-performance computing architectures. In HPC, it is essential, besides an efficient implementation, to choose the best fitting architecture for each problem. Latest research has shown that FPGAs are a better accelerating device than GPUs or multi-core CPUs for specific issues. Therefore, this thesis deals with the implementation and assessment of ODE solvers optimized for FPGAs.

Since FPGAs are relatively new in HPC, the thesis first explains the essential components of FPGAs and how to program them. Subsequently, the thesis expounds on the FPGA implementation of an efficient ODE solver and how it integrates into an automatic code generation to support different ODE systems. Furthermore, it illustrates the effect of the different optimizations on hardware utilization and execution time. The results are finally compared to those of CPUs and GPUs. The comparison reveals that for small problem sizes, the FPGA performs better than the CPU and almost as good as the GPU. For larger problems, both other architectures outperform the FPGA. The results give a first tendency when which architecture fits best. Consequently, this thesis builds a good foundation for further research about the usability of FPGAs for ODE solvers. This thesis did not take energy efficiency into account.

# Glossary

| | |
|---|---|
| **ALM** | Adaptive Logic Modules |
| **ALUT** | Adaptive Look-up-Table |
| **AOC** | Altera Offline Compiler |
| **CLB** | Configurable Logic Block |
| **DDR4-SDRAM** | Double Data Rate 4 Synchronous Dynamic Random Access Memory |
| **DSP** | Digital Signal Processing element |
| **FPGA** | Field Programmable Gate Array |
| **HBM2** | High-Bandwidth Memory 2 |
| **HDL** | Hardware Description Language |
| **HPC** | High-Performance Computing |
| **IP** | Intellectual Properties |
| **IVP** | Initial Value Problem |
| **LUT** | Look-up-Table |
| **RAM** | Random Access memory |
| **ODE** | Ordinary Differential Equation |
| **OpenCL** | Open Computing Language |
| **PCIe** | Peripheral Component Interconnect Express |
| **SIMD** | Single Instruction Multiple Data |
| **TFLOPS** | Tera Floating Point Operations Per Second |
| **TIDOWA** | Time Integration DOmain-specific Wicked Awesome stuff |
| **TMACS** | Tera Multiply-Accumulate Instructions per Second |

# Contents

# 1 Introduction

In biology and other scientific fields, the behavior of a system is often modeled in terms of changes over time. An example is the feedback loop of the human eye movement following an moving object in [2]. These models typically contain a number of parameters which determine temporal changes. Small changes in the parameters of a model may result in a completely different behavior. In experiments, a number of measurements of the system exist, and from those observations the values of the parameters of the underlying model need to be found. The calculation of the best fitting parameter values is called *parameter optimization* [26]. It is a task commonly used in computational science. Because of the sensitive behavior of many systems with respect to small changes in parameter values, parameter optimization needs to be very precise and is computationally intensive. It requires highly optimized implementations. Therefore, it is important to find the best performing hardware architecture for each problem.

## 1.1 ODEs, Integration and IVPs

For a mathematical representation of the previously mentioned models, systems of differential equation are commonly used. A differential equation defines the relation between a unknown function and its derivatives. Beside the derivatives a differential equation also contains an independent variable, for example $t \in \mathbb{R}$ as the time. This system would describe the change of the system over time. There are two types of differential equation, ordinary and partial. PDE contain multiple of the independent variables, ordinary only one. This thesis will only deal with ODE. An ODE is written mathematically:

$$\frac{dy(t)}{dt} = f(t, y(t)) \quad \text{abbreviated} \quad \frac{dy}{dt} = f(t, y) \text{ [26]} \tag{1.1}$$

while $f$ is a known function. The linear part of an Ordinary Differential Equation (ODE) can also be described as:

$$\frac{dy(t)}{dt} = \lambda y(t) \text{[33]} \tag{1.2}$$

Therefore, $\lambda$ is a parameter of the ODE. ODE can be extended to systems of ODEs. They are describe as the following:

$$\begin{aligned} \frac{dy_1}{dt} &= f_1(t, y_1, y_2, ..., y_n) \\ \frac{dy_2}{dt} &= f_2(t, y_1, y_2, ..., y_n) \\ ... &= ... \\ \frac{dy_n}{dt} &= f_n(t, y_1, y_2, ..., y_n) \end{aligned} \tag{1.3}$$

each function $y_i$ is now also dependent on the other functions [26]. Furthermore, a system of ODEs has multiple parameters $\lambda$. A simple example for an liner ODE system is the oscillatory system [33]:

$$\frac{du(t)}{dt} = \lambda u(t) - \lambda \bar{u} \tag{1.4}$$

This ODE system is used for all performance tests in this thesis.

**Integration**

For simple differential equations the solution of a ODE can be found analytical by separation of variables [6]:

$$\begin{aligned} \text{For the ODE:} \quad \frac{dy}{dt} &= t \cdot y(t) \quad \text{a solution can be found as follows.} \\ \frac{1}{y(t)} \cdot \frac{dy}{dt} &= t \\ \frac{1}{y} \cdot dy &= t \cdot dt \\ \int_{y_0}^{y} \frac{1}{\eta} \cdot d\eta &= \int_{t_0}^{t} \tau \cdot d\tau \\ \ln y - \ln y_0 &= \frac{t^2}{2} - \frac{t_0^2}{2} \\ y(t) &= y_0 \cdot e^{\frac{t^2}{2}} \cdot e^{-\frac{t_0^2}{2}} \end{aligned} \tag{1.5}$$

For more complex equations or ODE systems this solution approach is not feasible. Therefore, the solution needs to be found numerically. An example for a solution approach is Runge-Kutta 4 integration method [6]. This integration method is used exclusively for all implementations in this thesis.

**Initial Value Problem**

An Initial Value Problem (IVP) consists of a ODE system $\frac{dy}{dt} = f(t, y)$ and an vector $y_0 = (y_{0,1}, y_{0,2}, ..., y_{0,n})$. The i-te element of the vector $y$ is the starting value for each function. Therefore:

$$\forall i \in \{1, 2, ..., n\} : y_i(0) = y_{0,i} \tag{1.6}$$

Consequently:

$$y(0) = y_0 \tag{1.7}$$

The goal is to find all solution of y with a defined set of parameters $\lambda$ for a fixed number of integration steps $k \in \mathbb{Z}_0^+$ at all points $t = k \cdot \delta t$. This solution for one IVP can be found numerically by an ODE solver. The parameter optimization uses the results for a large number of IVPs with different parameters to find the best fitting value for a given measurement.

## 1.2 Goal of the thesis: FPGA relevance for ODE Solvers

This thesis focuses mainly on the Open Computing Language (OpenCL) code generation to implement efficient ODE solvers for Field Programmable Gate Arrays (FPGAs). It's a continuation of the work of Severin Bals's bachelor thesis "Development of a domain-specific language for the efficient time integration of ODEs" [3], Anna Mittermeier's project during her master studies at TUM, and Taylor Lei's bachelor thesis "Optimization of Parametrized High-Dimensional ODE Simulation" [26]. They developed a system named Time Integration DOmain-specific Wicked Awesome stuff (TIDOWA) at the Technical University of Munich. During this thesis TIDOWA was extended to a continuous integration system by Hoang-Trieu Tong. The TIDOWA system consists of two main parts. The first one is the general numerical time integration part that automatically generates ODE solver code for C, OpenMP, OpenCL, and CUDA based on systems of parametrized ODEs. These ODEs are represented in a domain-specific language. The second part is the generic parameter optimization. It extends the time integration part to a backend to call the ODE solvers for different parameter sets. Therefore, the TIDOWA system can optimize the parameters for any given ODE system and measurement results. The key fact is that it is using only the computational results of the IVPs. Therefore, it is constrained to black-box optimization [26]. For a good fitting result of the parameter set, the optimizer needs the solution of many IVPs. Therefore the performance critical task is the solving of the ODE System for multiple parameter sets. In addition to efficient and highly parallelized execution, it is necessary to run different ODE systems on the best suitable architecture or, in other words, use the best fitting part of a heterogeneous architecture. The TIDOWA system already supports

code generation and execution for the CPU with C, OpenMP, OpenCL, and GPU with CUDA and OpenCL. This thesis aims to extend the hardware coverage by adding OpenCL FPGA support to TIDOWA. Furthermore, it tests the relevance of this new architecture for ODE solvers by comparing the execution times of the FPGA to those of the GPU and CPU.

# 2 FPGA Systems

The primary focus of this thesis is to research how FPGAs perform relative to CPUs and GPUs for solving ODEs. Therefore, this chapter gives a brief introduction on FPGAs. In the following the essential building components of FPGAs are explained. Further, we will compare two of the most prominent FPGA vendors to select a sufficient suitable broad for the thesis. In addition, this chapter shows how FPGAs are programmed and why OpenCL and not a hardware description language is used for the implementation. Finally, we will take a brief look at compilation and execution in case of a simple OpenCL FPGA design example. With this design example, a better understanding is conveyed of how to work with FPGAs. This will help to understand the main differences in areas like compilation and execution.

## 2.1 Introduction to FPGA Systems

To better understand what FPGA Systems are, this chapter provides basic knowledge about the fundamental hardware architecture and the resulting advantages and disadvantages as compared to the more commonly used hardware systems like CPU and GPU. This knowledge is necessary to get a good comprehension of the optimization applied to the OpenCL ODE solver.

### 2.1.1 Architecture and Basic Components

A FPGA consists of three main parts. The first one is the *Configurable Logic Block (CLB)*. It is the smallest compute element of the FPGA, which can be programmed to perform different logic operations on the input data, like NAND XOR and AND. A CLB is built from a Look-up-Table (LUT), a multiplexer, and a register. Modern FPGAs contain several hundred thousand of CLBs [7] [5] [28].
To create more complex functions, these CLBs are connected by the *Programmable Interconnects*, which comprise of wire segments terminating in a programmable switch within a switch box. The routing switches can be activated to create more significant data paths between the CLBs [4] [7] [5].
The *I/O Blocks* form the last central part. They are responsible for connecting the inner components to the external hardware to pass data for computation to the FPGA [7] [5].

For a better pictorial representation of the three main parts, take a look at Figure 2.1. To increase the performance or reduce the area usage of a complex FPGA design, the vendors introduced *Intellectual Properties (IP)* to perform specific operations. For example *Digital Signal Processing elements (DSPs)* are dedicated units for multiplication. They are able to perform an 18x25 bit multiplication in one clock cycle. Further on, a DSP contains an accumulator, which enables optimized multiply-accumulation operations [4] [28] [27]. Such hardware structure is, for instance, useful for computational intensive applications like matrix-matrix multiplication to multiply the row and column elements and sum up the results. Because of their good ability to perform floating-point operations, theses IPs come to use for the ODE solvers. This becomes clear in the diagrams of the used hardware in Chapter 5.
In addition, *memory blocks* are built into the gate array architecture to store values tight to the design. The FPGA does not need to store every value on the global memory or in logic cells, which reduces the latency and the area usage [27] [45]. Moreover, such IPs offer space for hardware optimization. For example, Intel invented the "double pumping" configuration for the memory blocks, whereby the memory is clocked twice as high compared to the rest of the design. This unlocks two read or write operations per block and clock cycle of the main part [22]. Furthermore, there exist FPGA boards with IPs like Ethernet connections to communicated directly to external hardware and not through main memory and the CPU, enabling significant latency reduction [27] [12].

Newer FPGAs extend the CLBs to *Adaptive Logic Modules (ALM)*. These kinds of logic modules can implement more than one logic function. An ALM in an Intel Stratix 10 FPGA consists of two adaptive LUTs, two full-bit adders, and four registers. Various types of functions can be implemented by the two Adaptive Look-up-Tables (ALUTs) with up to eight inputs. Due to the adaptability, these ALM are completely backward compatible to the four input LUTs architectures implemented in previous generations of FPGAs [45] [25].

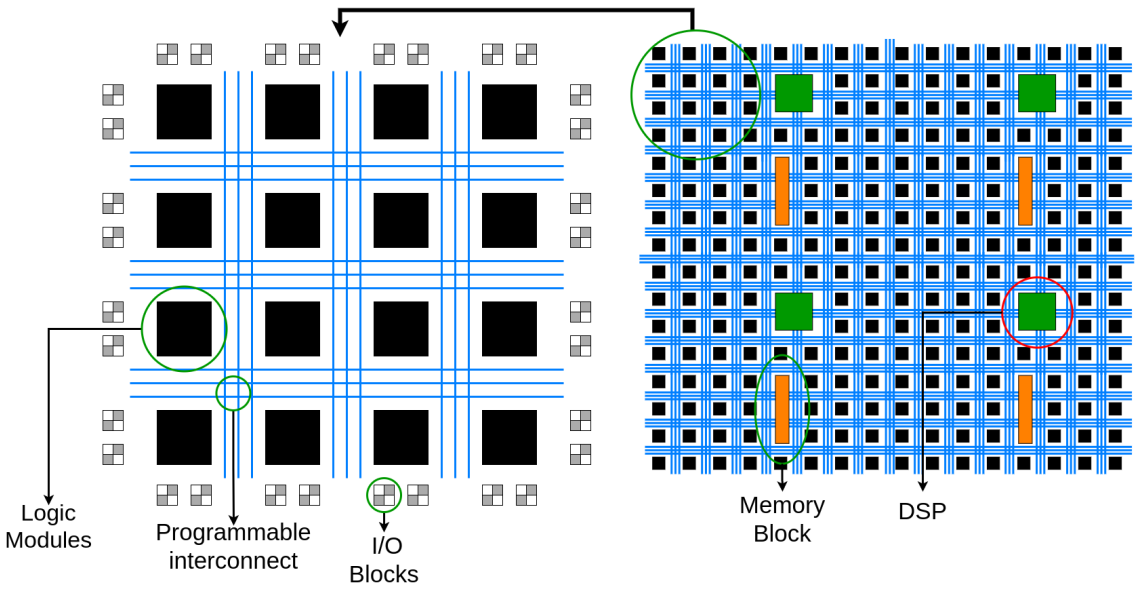


Figure 2.1: FPGA Basic Architecture: The right side shows an overview of the FPGA components and how they are connected to each other. The black rectangles are the Logic Modules, which are connected by the blue programmable interconnect. These array structures are interrupted by the green DSP blocks and the orange memory bank blocks also connected to the routing [4] [7] [28] [5] [45].

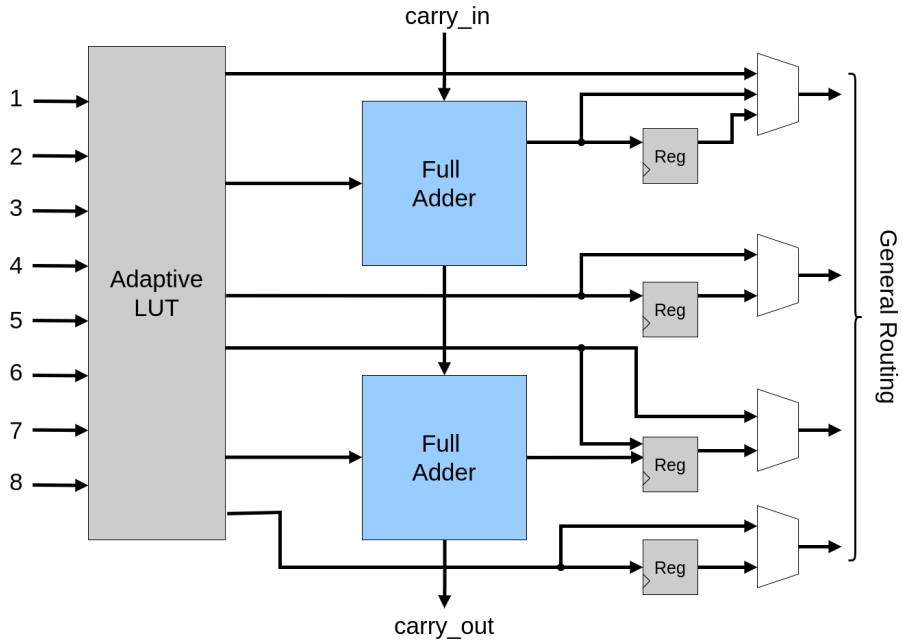


Figure 2.2: High-Level Block Diagram of Intel Stratix 10 Adaptive Logic Module: The left shows the adaptive LUTs. It consists internally of four 4-LUTs, which are feed by the eight inputs. The results can be passed to two full-bit adders or bypass the adders and send directly to the resisters or multiplexers. The outputs of the multiplexers are reconnected to the general routing [45] [25].

### 2.1.2 Comparison with CPU, GPU and ASIC

**Central Processing Unit (CPU):** Compared to CPUs, FPGAs have much lower latency. External hardware can be directly integrated into the FPGA and does not need to communicate with the compute unit via a generic bus system like Peripheral Component Interconnect Express (PCIe). Furthermore, latency is much more deterministic and does not depend on an operating system to release computing resources [31]. Overall compute power and energy efficiency are also advantages of FPGAs [37]. These advantages become evident in the results of the related work in Section 2.2. The architecture of FPGAs is also more flexible as compared to CPUs. Therefore, parallelism can not only be archived by multiple compute units. FPGAs are also able to supplement a pipelining computation. Consequently, FPGAs have a finer granularity in parallelization.
The most obvious advantage of CPUs is the programmability. CPUs support a wide range of programming languages, in contrast to FPGA, which can only be programmed by hardware description languages, OpenCL or C/C++. Therefore, the development time for CPU applications is drastically lower. The fixed CPU architecture also has a big advantage when it comes to compiling. Already the simplest FPGA designs needs hours for a successful compilation. In contrast, CPU compilation is finished in most cases in a matter of seconds.
**Application-Specific Integrated Circuit (ASIC):** The main difference between these two device architectures is that the logic of FPGA can be reprogrammed, while that of ASIC remains permanently wired. If you need a huge amount of devices with the exact same logic, which does not need to be changeable, ASICs are always more cost efficient and consume less power. Nevertheless, for a single device application or application that needs to reprogram the logic of the device, FPGA is the only suitable option [5].
**Graphics Processing Unit (GPU):** In terms of the core floating-point compute power, GPUs still have a clear advantage. The recently released NVIDIA GeForce RTX 3090 has 35.58 Tera Floating Point Operations Per Second (TFLOPS) for single-precision and therefore almost three times more floating-point performance than the recently released Intel Agilex F-Series FPGA with only 12.8 TFLOPS. Also, memory bandwidth is a huge advantage for GPUs because it is integrated into the board, while most FPGAs use external memory like Double Data Rate 4 Synchronous Dynamic Random Access Memory (DDR4-SDRAM) connected by PCIe [34] [8]. For applications with small memory allocations, the FPGAs have a big advantage. Modern FPGAs contain several thousand M20K memory blocks that are integrated into the array structure. Thus the largest systems have more than 250 MB of on-chip memory [8]. When it comes to power efficiency, FPGAs also have a significant advantage. This can be observed in the results of the related work in Section 2.2. Due to different threading models, FPGAs perform better for applications with large branching. FPGAs use a pipelining model, while

GPUs execute different threads in parallel on multiple compute units. The performance can be significantly reduced if different threads execute different branches. For the pipelining architecture, this is not a huge performance drawback [30]. Detailed results for the different performance tests for execution time and power efficiency is discussed in 2.2.

## 2.2 Related Work: FPGA in HPC

In the last decade, High-Performance Computing (HPC) commonly uses GPUs as accelerating devices to perform computationally intensive tasks [36]. Thanks to the application optimized hardware in modern FPGA systems, especially the larger on-chip memory, immense numbers of adaptive logic modules, and DSPs, FPGAs come closer to GPUs' core compute performance [32]. FPGAs' significant advantage over GPUs is the tremendous less power consumption and, therefore, better performance in GFLOPS/Watt [30]. The latest research shows how well FPGAs perform in comparison to GPUs. In the following, we will take a look at the results of some papers.

The paper [45] provides results of two performance evaluations for FPGAS in comparison to CPUs and GPUs. The first part discusses the implementation and optimization of Rodinia benchmarks for FPGA programmed in OpenCL. They show different optimization levels and compare the performance and power efficiency results to the latest implementation for CPUs and GPUs. The second evaluation discusses high-performance stencil computation on FPGA also using OpenCL. The results are again compared for performance and power efficiency of CPUs and GPUs. It should be mentioned that for the stencil computation, significantly more time was used for the implementation and optimization. For the FPGA test, an Intel Stratix V and Arria 10 were used. To get a fair comparison CPUs and GPUs of the same age were utilized. Intel i7-3930K and E5-2650 v3 for CPU tests and Nvidia K20X and GTX 980 Ti for GPU tests. The results show that CPUs are outperformed in every HPC benchmark by the both FPGAs. Furthermore, CPUs have even worse power efficiency. In one of the power efficiency tests, the Stratix V FPGA performs 16.7 times better than the Intel i7-3930K. For the GPU comparison, the FPGAs do not compete that well. Except in one case, the same age GPU produces better results than the FPGAs in every performance benchmark. Nevertheless, FPGAs have an advantage regarding power efficiency. The FPGAs perform clearly better than the power hungry GPUs in nearly every power consumption benchmark.
In the highly optimized stencil computation, FPGAs showed their real potential. They achieved up to 700 GFLOP/s in 2D stencil computation on the Intel Arria 10 GX 1150

board, nearly 50% of the maximal compute power. With these results, FPGAs are competitive to CPUs and even GPU devices of the same age.

The authors of [32] also researched the usability of FPGA acceleration for HPC compared to CPUs and GPUs. The scenario for their tests was a 3D Fast Fourier Transformation. An Intel Aria 10, Intel Xeon E5-2680v4, and an NVIDIA TESLA P100 device were used for the FPGA, CPU, and GPU execution. The results showed that the OpenCL total execution time archived an average speedup of 29 compared to the CPU and 4.1 compared to the GPU execution.

In [30], the authors evaluated the performance of FPGA against GPUs for three different design test cases. For the FPGA execution, a Xilinx Virtex-7 690t was used, and for the GPUs, two Nvidia devices were used, the GTX960 and Quadro K4200. The GPU kernels were all written in OpenCL. For the FPGAs, OpenCL was used with the Xilinx SDAccel and Vivado HLS high-level design tools, except for one algorithm. K-Nearest Neighbor (KNN) forms the first test case. Two implementation variations were implemented for this algorithm. The first one only uses the accelerating devices for the distance calculation, while the nearest neighbor is computed on the CPU. In contrast to that, the second version used the FPGA and GPU for both tasks by calling two kernels communicating to each other. The first evaluation results showed a clear advantage for the GPUs in terms of execution time due to the significantly higher bandwidth to memory. This observation changed for the second implementation. While the GPU needs to use the slower external memory for kernel communication, the FPGA can utilize on-chip memory and therefore eliminates the external memory access. In both cases, the FPGA out performs the GPUs in power consumption. The power consumption difference is more drastic for the second case because the off-chip memory access consumes the major part of the used energy in the first variant.
The second test algorithm was the Monte Carlo method for financial models. The authors explain that C/C++ with HLS-specific pragmas were used for this test case because of the efficient implementation of trigonometric functions. In every model, the FPGA beat GPUs in terms of execution time as well as power consumption.
For the last test suite, the bitonic sorting algorithm was utilized. NVIDIA provided the source code for this algorithm, which was optimized for GPU execution. The application was executed on the FPGA with and without FPGA specific directives. For the not optimized version, the FPGA clearly performed worse than both GPUs in execution time and even used more energy than the GTX960. With the FPGA specific directives, the Virtex-7 was able to beat the K420 and performed just as well as the GTX960. Furthermore, for the optimized implementation, the FPGA has an advantage in power consumption.

For further related research, see [44], [37], and [35].

## 2.3 Intel vs. Xilinx FPGAs

Two of the largest FPGA manufactures are Intel (originally Altera) and Xilinx (currently taken over by AMD). To find an appropriate board for the purpose of this thesis, those vendors are compared. First of all the different product divisions are shown. Secondly, for a better understanding of the key data sheet values for FPGAs and for choosing the right system, one Intel and one Xilinx system with different versions are compared. Finally, the reasons why the Intel devices and their programming tools were chosen is explained.

### 2.3.1 Device Family Comparison

Both vendor's product range can be divided into three categories. *Low performance or cost optimized* broads, which are the entry level models. For Xilinx, this includes Spartan-7, Spartan-6, Artix-7, Zynq-7000. Intel overs in this category the Cyclone Series and the MAX Series. Such systems are build of several ten thousand up to 400 thousand logic elements and several hundred DSPs [9] [39]. The second category is the *mid range or best performance per cost* devices like Kintex-7 and Virtex-7 for Xilinx, and the Arria Series for Intel. Those contain up to over a million logic cells and over a thousand DSPs [9] [39]. Intel, with the Stratix and Agilex Series and Xilinx with Kintex and Virtex, both in the UltraScale and UltraScale+ version, represent the last category: *high-performance system*. Those FPGAs belong to the best the market currently offers. The most powerful systems of these categories are capable of computing over 10 single-precision TFLOPS [9] [39]. The comparison uses the Intel Stratix 10 and the Xilinx Virtex UltraScale+. Though Stratix 10 is not the best performing Intel system because Agilex is built on newer transistor size and contains more logic elements and DSPs. However, Agilex got just released during the thesis, and currently, not all versions with their full data sheet are available.

### 2.3.2 Stratix 10 vs. Virtex UltraScale+

For each device family of the vendors, the comparison is split into two main parts, and each of them in several subsections for the different versions. The first central part is a comparison between the features that different models support. It will also be explained for what kind of application each model is optimized. The second part is a pure comparison of the data sheets provided by the manufacturers. It should be previously mentioned that all subversion of Stratix 10 and Virtex UltraScale+ (except

VU19P) can be configured in different versions, like the number of DSPs or logic elements. All referenced values are always the maximized configurations. For detailed information about a specific model configuration take look at the data sheets at the vendor's web page.

#### Stratix 10 Features

The feature supported by all Stratix 10 devices is that they are all built based on the *Intel Hyperflex FPGA architecture*. This architecture's centerpiece is the "register everywhere" design that adds Hyper-Registers to the interconnect routing. These registers are distinct from the conventional registers that are built into adaptive logic modules. This allows bypassing every routing segment in the FPGA core and functional blocks like ALMs, embedded memory blocks, and DSPs [12]. Through this new hardware design, two optimizations are available:

- Hyper-Retiming: eliminates critical paths by using registers in the interconnect and not the one located in the ALMs, which allows running at a faster clock frequency.
- Hyper-Pipelining: adds additional pipeline stages between ALMs to eliminate long routing delays.

According to Intel, Stratix 10 with Hyperflex can achieve two times the performance and up to 70% lower power consumption at the same performance compared to previous high performance FPGA Stratix V [1].

Furthermore, all Stratix 10 boards contain a *Secure Device Manager*, which is the entry point for all JTAG commands and data for device configuration. The SDM block manages all security and configuration functions while not affecting the user design [12]. Stratix 10 boards also support new transceiver technology, capable of up to 28.3 GB/s across the backplane. Besides, all of the variations offer parallel memory support up to 2,666 Mbps for DDR4-SDRAM. Moreover, they can use a wide range of external memory protocols [12]:

- hard memory controllers: DDR4-SDRAM, DDR3/ DDR3L, LPDDR3
- soft controllers: RLDRAM 3, Intel Optane DC persistent memory, QDR II+ / QDR II + Xtreme / QDR IV

#### Intel Stratix 10 FPGAs Specific Variation Features

The *GX* version is optimized for applications that require the highest transceiver bandwidth and core fabric performance [13] [14].

The system on a chip devices of the Stratix 10 family is the *SX Soc*, which contains a quad-core ARM Cortex-A53 MPCore hard processor system, therefore the best fitting FPGA for embedded applications. The processor supports several features like hardware virtualization, system management, monitoring capabilities, acceleration preprocessing, 64-bit architecture (ARMv8), 32-bit execution mode, and has board support packages for popular operating systems like Linux [18] [14].
The board that meets the bandwidth demands for 5G communication, cloud computing, and network virtualization is the *TX* FPGA. It is capable of up to 57.8 GB/s in up to 144 transceiver lanes and contains an Ethernet hard IP solution. Furthermore, it is optimized for networking infrastructures that support 50GE, 100GE, 200GE, and 400GE applications, and a quad-core ARM Cortex-A53 MPCore hard processor system can be integrated into the FPGA [19] [20].
The *MX* version to Intel Startix 10 is the best accelerator for HPC, data center, and virtual networking functions. It integrates two High-Bandwidth Memory 2 (HBM2) with a memory bandwidth of up to 512 GB/s. The DRAM is physically connected to the FPGA using Intel EMIB. Through the significantly shorter interconnect between the core fabric and the memory, the FPGA uses a lower system power, resulting in an optimum performance per watt [16] [15].
With the Intel Ultra Path Interconnect, a direct coherent connection to Intel Xeon Scalable processors, the Intel Stratix 10 *DX* FPGA fits the best for designs, which need high bandwidth. In addition, it is the only Stratix 10 version supporting PCI Express Gen4x16 with up to 16 GT/s. Moreover, it supports Intel Optane persistent memory and has the option to built-in a quad-core ARM Cortex-A53 MPCore hard processor system or 8 GB integrated HBM2 DRAM with 512 GB/s [10] [11].
Intel Stratix 10 *NX* is the optimized FPGA for AI applications that require high bandwidth and low latency. Its AI Tensor blocks are tuned for matrix-matrix or vector-matrix multiplications, commonly used in areas like Machine Learning. These optimized hardware blocks have fifteen times more INT8 throughput than a standard DSP block. Additionally, the enlarged integrated memory stacks allow to store persisted AI models in the on-chip memory, enabling lower latency with larger memory bandwidth and therefore preventing memory bound performance challenges [17].

#### Xilinx VIRTEX UltraSCALE+ FPGA Specific Variation Features

In this Section we compare directly the version specific features, because unlike Stratix 10 Xilinx VIRTEX UltraSCALE+ does not share some basic features over the hole lineup. The first version we take a closer look at is the base *VIRTEX UltraSCALE+* with no name extension. It supports DDR4-SDRAM with up to 2,666 Mb/s and can be configured

with up to 500 Mb of on-chip memory. The maximal amount of transceivers integrated into the board is 128, which runs at a speed of 32,75 GB/s. In addition, this device supports PCI Express Gen3 x16. It is the best Xilinx board in terms of optimized performance in fixed and floating-point computation with up to 22 Tera Multiply-Accumulate Instructions per Second (TMACS) [40].
The second board, which is mostly the Xilinx equivalent to Intel Stratix 10 MX and DX, is named *Xilinx Virtex UltraScale+ HBM*. It supports, like the two Intel FPGAs, High-Bandwidth Memory Gen2, at a data rate of 460 GB/s. Furthermore, the HBM version has build-in transceivers with 58 GB/s and PCIe Gen4 x8 connection with CCIX support [42].
The next variant in the lineup is the *58G*, which is optimized for applications that require high connection speed from external components to the core fabric. Therefore, it supports 48 transceivers at 58 GB/s and 32 at 32.75 GB/s. It has integrated blocks for PCIe Gen3 x16 in all devices and PICe Gen4 x8 with CCIX in selected versions [41].
*VU19P* is the model of Virtex UltraScale+ with the highest amount of built-in logic elements and, consequently best fitting board for prototyping and emulation of advanced ASIC and SoC designs [43].

| Version | max Logic Elements (million) | maximal DSP blocks | peak fixed-point performance (TMACS) | peak float point performance (TFLOPS) | max adaptive logic modules |
|---|---|---|---|---|---|
| Intel Stratix 10 default | 2.753 | 5,760 | 23 | 9.2 | 933,120 |
| GX | 10.2 | - | 13.8 | 5.5 | 3,466,000 |
| SX SoC | - | - | - | - | - |
| TX | - | - | - | - | - |
| MX | 2.753 | 3,326 | 15.8 | 8.0 | 702,720 |
| DX 1100: quad-core ARM processor | 1.325 | 2,592 | 10.4 | 4.1 | 449,280 |
| DX 2100: 8 GB HBM2 | 2.073 | 3,960 | 15.8 | 6.3 | 702,720 |
| DX 2800 | 2.753 | - | 23 | 9.2 | - |
| NX | all not specified | | | | |

"-" is used to refer the default value at the top

Table 2.1: Intel Stratix 10 FPGAs Data Sheet Comparison [14] [20] [15] [11]

| Version | max System Logic Elements (million) | maximal DSP Slices | HBM DRAM (GB) | Transceivers 32.75/58 GB/s | PCIe Gen3 x16/Gen4 x8/CCIX | on Chip Memory (Mb) |
|---|---|---|---|---|---|---|
| Virtex UltraScale+ | 3.780 | 12,288 | 0 | 128/0 | 4/0 | 455 |
| Virtex UltraScale+ HBM | 2.853 | 9,024 | 16 | 96/0 or 32/32 | 0/4 | not specified |
| Virtex UltraScale+ 58G | 3.780 | 12,288 | 0 | 32/48 | 4/0 | 455 |
| Virtex UltraScale+ VU19P | 8.938 | 3,840 | 0 | 80/0 | 0/8 | 224 |

Table 2.2: Xilinx VIRTEX UltraSCALE+ FPGAs Data Sheet Comparison [38]

### 2.3.3 Design Decision for Intel

The computation power in floating-point operations per second is the critical data sheet values because floating-point calculation is the main task performed in the ODE solver. Unfortunately, both vendors do not provide data sheets with the same values. Moreover, the data they have in common describes not always the exact same hardware elements. For example, Intel uses the term DSP blocks. In contrast, Xilinx describes this as DSP Slices. Additionally, Xilinx does not mention the overall compute power in TMACs or TFLOPs, except for the base model Virtex UltraScale+, but only the fixed-point performance. Also, this value does not outreach the performance of the Stratix 10 GX, which is the best Intel FPGA for core fabric performance. Furthermore, Intel provides good written documentation about their FPGA SDK for OpenCL and useful tutorials to getting started with FPGA programming. The documentation is split into three parts: "Getting Started Guide" [23], which introduces how to install all necessary components. Furthermore, it explains how to compile OpenCL Code with their Offline-Compiler for emulation and FPGAs. In addition, it shows how to run an example design in emulation mode and on the FPGA. The "Programming Guide" [24] takes a closer look at the different compiler configurations and the OpenCL kernel and host program structure. The last part is the "Best practice guide" [22], which explains how to tune different parts of your design to get the best performance. Intel also provides some interesting examples to better understand good performing OpenCL design, for example, Intel's matrix-matrix multiplication. Due to the well written documentation with tutorials and examples, the better comparable data sheets with performance values over almost all devices, the decision was made to go with Intel FPGAs and its SDK for OpenCL.

## 2.4 Programmability of FPGA Systems

This section gives an overview of how FPGAs can be programmed. The first part explains the basics of the programming idea of OpenCL. The second part looks at the more original way to program FPGAs with hardware description languages. The OpenCL section takes the major part of the paragraph because it is the programming language used in this thesis.

### 2.4.1 OpenCL Architecture

OpenCL is not only a programming language. It consists of an API, libraries, a run time environment, and a language called OpenCL for kernel programming. It is an

industry standard for programming different types of hardware architecture like CPU, GPU, and FPGA [45].

### OpenCL Host and Device Side

The execution of the OpenCL program can be split into two parts. The first one is the host side code that typically executes on the CPU. This code is written in regular C or C++. The second part is the device side code, also called the kernel, which executes on the accelerating devices. This device is typically the FPGA, GPU, or the CPU itself. The kernel code is written in OpenCL, a C-like programming language. The host can call the OpenCL API to start a new context, transfer or send data to the accelerator, get information about the devices or the execution state of the kernel, and of course, start the kernel execution. It is the part that initializes all necessary data and manages the execution of the kernel. The kernel is almost always the performance critical part of the program. It is a standard C-like method that receives all necessary data through the function arguments [29] [45]. To better understand how this kernel method is executed on the devices, the next section closely considers the execution or also called the threading model.

### OpenCL Execution Model

There are two different execution modes of the kernel. The first one is *NDRange* model. When the host starts the kernel, the host needs to create an index space in one, two, or three dimensions. The three dimensions can be any size. Each element in this space gets a unique ID, called global ID. Depending on the number of dimensions that the host defined, the global ID has one, two, or three components. For each such element, named work-item, the kernel is started once. The number of all work-items is called global work size. These work-items are packed into work-groups, which is also an index space with the same number of dimensions as the global space, but with a significantly smaller size. Why they are assigned to groups becomes evident in the memory model. A fitting example for this kind of execution mode is a matrix-matrix multiplication, where each element in the resulting matrix represents one work-item. Its position in the matrix is defined by the two parts of the global ID [29] [45].
The second Programming model is called *Single Work-Item*. In contrast to NDRange, the host program does not define a global index space. The kernel called by the host is executed exactly once, therefore single work-item execution, is also called a task execution. This one kernel does all computation. Parallelism is achieved through pipelining and vectorization of the loops [29] [45].

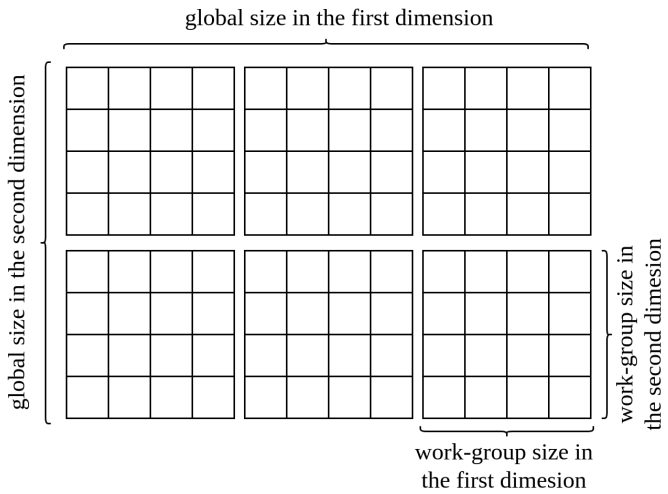


Figure 2.3: NDRange example model with two global dimensions of the size 12 in the first and 8 in the second dimension. Furthermore, this example has a work-group size of 4 in both dimensions. That results overall in 96 work-items packed into 6 different work-groups with 16 work-item each.

#### OpenCL Memory Model

The memory model of the OpenCL standard defines four different types of memory.
**Global Memory:** The data in this memory area is visible to all work-items in every work-group. It allows read and write access. The global memory is the largest but also the slowest memory because it is generaly located in off-chip memory like DDR4-SDRAM. In most cases, it has a capacity of several Gigabytes, therefore good comparable to the main memory in the Von Neumann architecture. When the host program transfers data to the FPGA by calling the OpenCL API, it is stored in this memory type. Depending on the device, specific areas are cached in the on-chip memory [29] [45].
**Constant Memory:** It is also accessible for all work-item in every work-group but only allows read accesses. Like the global memory, it is stored on external memory and can be cached at run time in on-chip memory. The host creates data in this memory when it transfers memory buffers with a read-only flag to the device. The constant memory arguments of the kernel are signaled to the compiler by adding the keyword "constant" before the variable name. For FPGA, this helps the compiler to optimizes the memory access [29] [45].
**Local Memory:** This kind of memory is shared across all work-items in one work-group. Work-items can not access the local memory of other work-groups. The host has no access to this memory because it is typically stored in the on-chip memory and initialized by the kernels. For FPGA the on-board memory blocks as explained in 2.1.1 are used for this memory type. Consistency for all work-items in one work-group is only guaranteed after a barrier call in the kernel [29] [45].
**Private Memory:** Any variable, array, or buffer declared in the kernel code without the "local" keyword is in the private memory. It is only accessible for the work-item, which allocated it. On FPGAs, this memory is normally stored in registers or memory blocks for larger data. If it exceeds the on-chip memory of the FPGA, it can leak to global external memory. This creates huge performance drawbacks [29] [45].

### 2.4.2 Hardware Description Languages

Hardware Description Languages (HDLs) have compared to standard programming languages a completely different structure. Originally, they were used to describe a hardware design in text format to develop devices like ASICs. Therefore, every data path and computation needs to be manually developed.

### 2.4.3 Reasons for Design-Decision using OpenCL

OpenCL with the Intel FPGA SDK for this thesis is used for this thesis for several reasons. First of all, a C-like language is better fitting for automatic code generation. Additionally, the more familiar language with the significantly better debugging is important for a working end result. Furthermore, the TIDOWA system already supports OpenCL for CPU execution. Therefore, much of the host source code can be reused, and the kernels only need to be adjusted and optimized for FPGAs.

## 2.5 FPGA Development Tools and Design Example

This chapter gives a short introduction of how to use the Intel FPGS SDK for OpenCL by explaining the necessary steps to compile and run a FPGA example design. With this basic example, it is easier to understand how the workflow is integrated into the TIDOWA system. First of all, we take a look at two important commands to interact with the SDK.

### 2.5.1 Work Flow of Intel FPGA SDK for OpenCL

For a normal OpenCL design, the compilation of the kernel would take place at run time. Due to the extremely high compile time of several hours, the kernel code is compiled offline by the SDK Offline Compiler with the command:

```
$ aoc
```

In addition to the aoc command, Intel also offers a utility command called "aocl", to get information about the SDK. For example:

```
$ aocl version #prints the SDK version
$ aocl list-devices #prints all installed devices
$ aocl compile-config #Shows the flags for compiling your host program
$ aocl link-config #Shows the flags for linking your host
                   #program with the runtime libraries
$ aocl diagnose #Run your vendor's test program for the board.
```

In contrast to the kernel, host side source code is compiled with a normal C or C++ compiler. The host binary that executes on the CPU loads the precompiled bit stream on the FPGA and starts the execution [23].

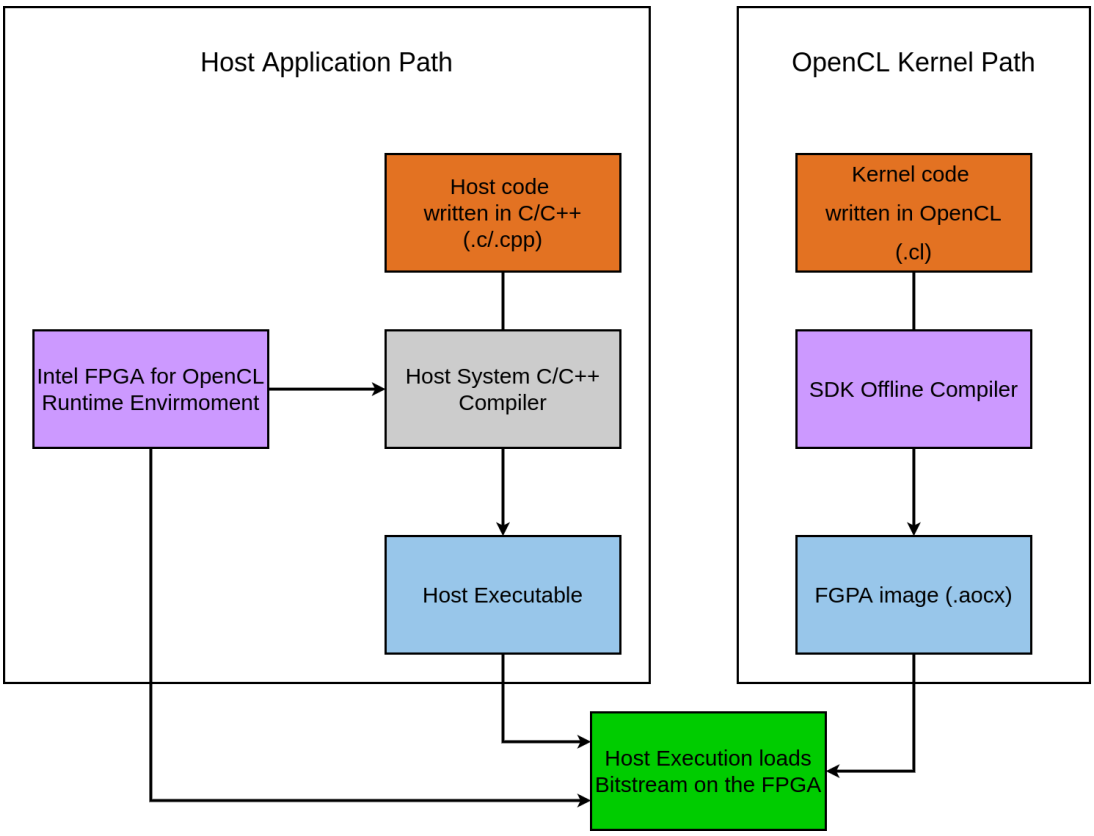


Figure 2.4: The left side shows the compilation path of the host code, which uses the C/C++ compiler install on the host system. The host program is linked with the run time environment provided by the SDK. The right side represents the path of the kernel compilation. The OpenCL source code stored in a .cl file is compiled by the SDK offline compiler that creates a FPGA image saved in an .aocx file [24].

### 2.5.2 Intel Matrix Multiplication

#### File Structure

The project consists of two parts. The common folder stores useful .cpp files for the main program. For example, it contains functions to check error codes returned by calls of the OpenCL API or functions that make it easier to get the right OpenCL platform. The second directory is named matrix_mult, which itself is divided into two subfolders: device for the .cl files and host for files like main.cpp executed in the host machine. The compilation command will later create a third directory called bin for all executables and binaries.

#### Compilation and Execution in Emulation Mode

To ensure that the kernel and host work properly, we first compile and run the program in emulation mode. The emulation device is installed with the SDK, and therefore it is not necessary to have a real FPGA plugged into the system to ensure that the program satisfies functional correctness. To compile the .cl file for emulation, we use the aoc command:

```
$ aoc -march=emulator -legacy-emulator -v device/matrix_mult.cl
-o bin/matrix_mult.aocx
```

-march=emulator -legacy-emulator are used to tell the compiler to compile the kernel for the emulation device. -v reports the compilation progress. -o defines the path to the output file. The host is compiled by executing the Makefile in the matrix_mult folder. This creates an executable in the bin folder. To start the program in emulation mode, type the command:

```
$ env CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=1 ./bin/host
-ah=256 -aw=256 -bw=256
```

The environment variable is necessary to tell the host program to use the emulation device. ah, aw and bw flags specifies the size of the matrix. They need to be set because the default matrix size is pretty large, and therefore the execution in emulation mode would take some time. The result of the program is printed to the console.

#### Compilation and first Execution on FPGA

After a first successful run of the matrix_mult in emulation mode, it is time to execute the FPGA kernel to test it on the real hardware system. Therefore you need to recompile it for a real FPGA board. Use the following command if you want to use the default board:

```
$ aoc device/matrix_mult.cl -o bin/matrix_mult.aocx -v
```

If you want to use a specific board use:

```
$ aoc device/matrix_mult.cl -o bin/matrix_mult.aocx -v -board=<board name>
```

To get the name list of all installed boards call:

```
$ aoc -list-boards
```

This compilation needs several hours to finish. After a successful compilation, run the host program again, but without the environment variable:

```
$ aoc ./bin/host -ah=512 -aw=512 -bw=512
```

#### Optimized Compilation and Execution

To increase the performance, you can add two additional flags to the Offline Compiler.

```
$ aoc device/matrix_mult.cl -o bin/matrix_mult.aocx -fp-relaxed -fpc
```

-fp-relaxed directs the OpenCL Offline Compiler to relax the order of arithmetic floating-point operations using a balanced tree hardware implementation. -fpc removes intermediary floating-point rounding operations. For even better performance, increase the number of items in the one work-group. This improves the load operations from the global memory. The third optimization is to specify the number of SMID work-items. This attribute vectorizes the kernel. It is explain in detail in Section 4.3.4 . The optimized compiling command is:

```
$ aoc device/matrix_mult.cl -o bin/matrix_mult.aocx -fp-relaxed -fpc
-DBLOCK_SIZE=256 -DSIMD_WORK_ITEMS=16
```

**Important:** The default block size (work-group size) in the kernel and the host is set to 64. If you recompile the kernel with a new block size, it is necessary to recompile the host program with the equivalent value. Just set BLOCK_SIZE with the desired value as an environment variable.

**Note:** The step explained above to compile and run the kernel is specific to the installation version of the Intel FPGA SDK for OpenCL (19.2). In newer versions, some flag and environment variables can change.

# 3 TIDOWA

To create and run a solver for any given ODE system, the TIDOWA system implements four steps discretization, code generation, compilation, and execution. All four steps are implemented in python. The next sections explain the different TIDOWA steps.

## 3.1 Discretisation

The discretization step is implemented in a python file called main_disc.py. At first, it loads a python configuration file. This configuration file uses sympy to define the ODE system in a symbolic representation. Furthermore, it stores the integration method used to discretize the ODE system. With the ODEs and integration method, this TIDOWA step creates a discretized version of the ODE system and stores it in a python file called discretization record. The discretization is now finished, and the code generation can start. For detailed information on how the discretization works, see [3].

## 3.2 Automatic Code Generation

The python file main_codegen.py is responsible for the dynamic code generation. Before we can understand how the code is generated, we need to know which files are used to implement the ODE solver. Depending on the code type, the solver is implemented in one or two files. The first one, which is contained in every code type, is the main C or C++ file. It manages the execution. For example, it loads all IVPs stored in a .csv file and saves them in buffers. The main program calls the solve_system method ones for each of the IVPs. After the code generation, this function implements the time integration for a specific ODE system and integration method. Currently, the code of solve_system contains placeholders for the implementation of the integrating method. Depending on the code type, this method is stored in a second or is implemented in the main file. The solve_system method writes the results to an array. The main program gets the result array and saves the results in a .csv file. For each of the code types, there exists one such implementation called template. The actual code generation is done by a python class called Transpiler. There exists one subclass for each of the code types, for instance, C_Transpiler for the automatic C code generation. The main_codegen.py

program creates the corresponding transpiler instance and calls the generate method of the transpiler. This method takes the solve_system function of the Template and replaces the placeholders with the implementation of the integration. The discretization record defines the ODE system and integration method. This TIDOWA step stores all files in a new code_built directory. After the code generation step the generated code is syntactically correct. All placeholder were superseded. Furthermore, the generated files implement a semantically correct ODE solvers. See [3] for further information.

## 3.3 Compilation

TIDOWA compilation step takes the generated code and compiles it to create an executable. The main_compile.py python file implements the automatic compilation. TIDOWA compilation supports different compilers like GNU, Intel, or NVIDIA CUDA compiler. The code type defines the compiler and the flags that are used for the compilation. With the compiler, the flag, the code input files, and the output file's name, the compilation step generates and executes a compilation command. Initially, in [3] this step was implemented in Makefiles. Former work has changed it to a python-based build system.

## 3.4 Execution

This part only starts the executable with the necessary arguments like the CSV input and output file paths.

# 4 FPGA in TIDOWA

This part of the thesis explains the FPGA OpenCL ODE solver's implementation and how the code is integrated into the TIDOWA system. We will first take a look at the host and kernel side implementation. It is important to know the kernel and host structure to understand the kernel-optimization, covered in the following section. After getting a deep understanding of implementation, the last part of this section shows how to use the automatic TIDOWA code generation and how the different optimizations are applied.

## 4.1 Host Side Implementation

The host side implementation is written in standard C++. It consists of two parts. The first one is the common directory with the AOCLUtils .cpp files. It is provided by Intel and implements useful functions to simplify things like the finding the correct OpenCL platform. The second part is the main.cpp. It manages the execution by setting up all the necessary variables, starting the kernel, and finishing the calculation correctly. The initialization takes on the following tasks: Creates arrays for the integration starting values, the different parameter values, and the results.

### 4.1.1 OpenCL Initialisation

The function that implements the initialization is called opencl_init. It calls functions to get the right platform and device id. Further on, it creates an OpenCL context with the device id. Beside, this function generates and builds a program object from the .aocx binary file. All variables are stored globally to have access to them in the other OpenCL functions.

### 4.1.2 Start OpenCL Kernel Execution

This function first creates a new command queue and all OpenCL buffers for the kernel. After the allocation of the buffers, the host writes the corresponding array or variable on it. For example, the previously created parameter array is written into the parameter OpenCL buffer. The host calls the clCreateKernel method with the program object and

kernel name as arguments to create a new kernel instance. The initialized buffers are now set as kernel arguments. Before the host side kicks of the kernel execution, the last step is to create the global and local work-group size. This implementation only uses one dimension to define the global ID because more dimension would only cause more complicated indexing in the kernels while gaining no profits. The host now starts the kernel by executing the NDRange method. After the kernel finishes successfully, the host writes the results stored in one of the OpenCL buffers into a global array. Furthermore, it calls an OpenCL profiling method to get the kernel's execution time. The last part finishes the command queue and releases all OpenCL objects created in this method, like the kernel and the buffers.

### 4.1.3 Finish OpenCL

The method opencl_cleanup calls the release method for the globally stored program and context object to finish the OpenCL part correctly.

The OpenCL part is split into three methods to accelerate the execution and prevent memory leaks for the optimizer. The optimizer often calls the solve method implemented in the kernel. It would create a huge overhead, and unnecessary memory allocations if the initialization and finish part would be in the same function that starts the kernel.

## 4.2 Kernel Side Implementation

### 4.2.1 Kernel Function

The function solve_system is marked with the keyword __kernel to tell the compiler that this method is a kernel. The solve_system kernel has 6 arguments:

- Two pointers to double arrays for the different parameter and initial value sets.
- One pointer to a double array for the results.
- One double pointer for the time integration step size.
- Two int pointers for the number of integration steps and the position when the kernel starts saving the results.

All arguments except for the result array are marked with the __constant keyword to tell the compiler that these values are stored in the read-only memory. Therefore, the compiler can perform optimizations for these variables.

### 4.2.2 Index Initialisation

The kernel first stores all constant single value arguments in the private variables. Then it calls the get_id function for global and local item work space. With these values, the kernel calculates the offset in the parameter, initial value and result array.

### 4.2.3 Array and Variable Initialisation

In this step of the execution, the kernel allocates two privates array, one for initial values and one for the parameters. In two for loop, the kernel initializes the arrays by reading the corresponding data with the calculated offset from global memory. For each equation in the ODE system, the solve_system method created one private variable. Furthermore, it allocates the necessary variables for the integration method.

### 4.2.4 Main Calculation Loop

In every loop iteration, the kernel calculates the results for one time integration step. It first increases the time variable by the integration step size. Depending on the integration method, the kernel computes the necessary variables. For example, if the implementation uses Runge-Kutta-4, four variables are calculated for each equation in the ODE system. With the calculated variables for each equation, the kernel computes the integration results and stores it in the variable. This loop is the most computationally intensive task and accounts for a large part of the execution time and hardware resources.

### 4.2.5 Result Write-Back to Global Memory

The result's write-back is integrated into the integration loop, explain in Section 4.2.4. With an if statement, the kernel determines if the integration loop already reached the start saving point. After the integration loop reaches this point, the kernel writes the previous iteration results back to global memory. The number of equations defines the size of the inner loop. For example, if the host sets the start saving point to 0, the kernel writes the results into the global memory in every loop iteration.

## 4.3 Kernel optimization Techniques

The following sections explain different optimization techniques that are applied to the solve_system kernel. For each technique, the corresponding section shows what this optimization is about, on which part of the code it can be used, and how it can lead to better execution times. Furthermore, it takes a look at its restrictions and downsides.

### 4.3.1 Loop unrolling

Loop comes with the downside that the control logic can cost performance, especially if the termination condition is complex. To reduce this overhead, OpenCL allows unrolling loops. The compiler replicates the loop body multiple times and decreases the number of iterations. If there are no loop-carried dependencies, the iterations pack into one run, can be performed in parallel. To direct the compiler to use loop unrolling, a pragma is applied directly in front of the loop [24]. For example:

```
#pragma unroll <unroll_factor>
 for (int i = 0; i < N; ++i) {
     <loop_body>
 }
```

The unroll factor defines the number of loop body replications in one iteration. This optimization can be applied to the initialization of the initial values and parameter arrays. Furthermore, it can be used for the write-back loop of the results. We can fully unroll all of these loops because the trip count is very small and known at compile time. The main calculation loop can also be unrolled but only by a factor. Complete unrolling is not feasible because the size of the loop is typically not set at compilation time. Moreover, a complete unrolled integration loop probably does not fit on the FPGA. The technique's downside is the larger area usage because the compiler needs more logic to implement the larger loop body. Besides, loop unrolling comes with the restriction that the loop size must be evenly dividable by the specified unroll factor.

### 4.3.2 Specifying required or maximal Work-Group Size

Intel recommends specifying the size of one work-group whenever possible. There are two variants to do so, defining the maximal or required work-group size. The maximal only sets an upper bound for each dimension of the work-group size, while the required work-group size sets strict values for each dimension. If these values are not specified, a default value is assumed by the compiler. This can lead to inefficient hardware if the actual size deviates significantly from the compiler's assumption [24] [22]. The restriction is that if the host calls the kernel with an invalid work-group size, the computation will result in an error. A kernel with a required work-group size of 256 in the first dimension would look like:

```
__attribute__((reqd_work_group_size(256,1,1)))
__kernel void kernel_name()
```

### 4.3.3 Specifying Compute Units

One option to add data parallelism to the FPGA execution is to specify the number of compute units. To set the number of compute units, we add the following argument to the kernel:

```
__attribute__((num_compute_units(<number_of_compute_units>)))
 __kernel void kernel_name()
```

The compiler replicated the kernel logic by the number of compute units. Each compute unit can execute several work-groups concurrently though pipelining. The work-groups are dynamically distributed to the different compute units. Two work-items in the same work-group always execute on the same compute unit. The implementation downside is that the several compute units use more FPGA resources and the parallel execution of work-groups need more bandwidth [24] [22].

### 4.3.4 Specifying SIMD Work-Items

The second version that applies data parallelism to the FPGA execution is the vectorization of the kernel over the work-item. The FPGA executes with this optimization multiple work-items in a single instruction multiple data manner. The compiler archive this by a widening of the pipeline. It increases the data throughput at the cost of more area usage. To apply vectorization, add the SIMD attribute in front of the kernel:

```
__attribute__((num_simd_work_items(16)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void kernel_name()
```

The restriction is that the required work-group size must also be specified if the SIMD attribute is added. Furthermore, the work-group size must be evenly dividable by the number of SIMD work-items. The vectorization factor must be a power of 2 and equal to or less than 16 [24].

The last two sections dealt with two different options to apply data parallelism to FPGAs. Intel recommends preferring the vectorization over the multiple compute units because the compiler can generate more efficient hardware while archiving the same goal. For example, one memory access in four different compute units are distinct from each other, while a kernel with a SIMD factor of four performs only one memory access with a larger width [22].

### 4.3.5 Floating-Point Optimization

#### Relaxed Order of Floating-Point Operations

Long calculation with an unbalanced structure have a long critical path. To create more efficient hardware, the calculation can be reordered by generating a balanced tree structure. The compiler applies the relaxed order of floating-point operation if following flag is added to the kernel compilation:

```
$ aoc -fp-relaxed <file_name>.cl
```

Because the order of floating-point operations has changed, the results can differ minimally from the original computations. Therefore, this optimization suits only in application with a tolerance [22].

#### Rounding Floating-Point Optimization

In some kernels that implement complex calculations, many intermediate rounding operations are performed. These rounding operations consume much of the hardware resources. The flag -fpc directs the compiler to reduce the number of rounding operations. This reduces the required hardware resources. Similar to the optimization explained in Section 4.3.5, this technique violates the IEEE Standard 754-2008 and therefore produces slightly different results [22]. The compilation command is:

```
$ aoc -fpc <file_name>.cl
```

### 4.3.6 Avoid Pointer Aliasing

Pointer aliasing means that two arguments of the kernel can address the same location in the memory. The compiler needs to create extra hardware to prevent invalid read and writes to the potential same memory location. If there is no aliasing between pointers, direct the compiler to get rid of the extra control logic by applying the keyword “restrict” in front of the argument name. Therefore, all kernel arguments can be marked with this keyword because in this design there is no aliasing between the arguments.

## 4.4 Integration of OpenCL for FPGA into TIDOWA

The chapter 3 explained the four TIDOWA steps to generate and execute an ODE solver for any interagteion method and ODE system. To support OpenCL for FPGAs these steps are extended or slightly changed. Furthermore, to be able to generate different versions of the optimization, a fifth step is added. This new part is not directly

integrated into the TIDOWA system. The following sections explain the changes applied to the different TIDOWA step to add OpenCL for FPGAs. The discretization was left out because it remained the same.

### 4.4.1 Automatic Code Generation

To support OpenCL for FPGAs, a new template was added with the main.cpp file, the kernel written in OpenCL and the utility files explained in Section 4.1. Furthermore, a new subclass called OpenCL_FPGA_Transpiler of the transpiler class was implemented. The generated code of this code type differs slightly. While the generated code is for the other languages already syntactically correct and ready for compilation, the OpenCL code for FPGAs contains placeholders used later in the optimization part.

### 4.4.2 Compilation

The Altera Offline Compiler (AOC) was added to the set of compilers to support kernel compilation for FPGAs. Moreover, new compilation commands were implemented into the main_compile.py program to compile the OpenCL host and kernel program.

### 4.4.3 Execution

The execution stage stays the same, only an environment variable is set if the OpenCL program should run in emulation mode. This environment variable is explained in Section 2.5.2

### 4.4.4 Automatic FPGA Optimization

The optimization takes place between the TIDOWA code generation and compilation step. As explained in Section 3.2, the .cl file still contains markers. For example:

```
%%OPTI_NUM_COMPUTE_UNITS%%
%%OPTI_SIMD_WORK_ITEMS%%
%%OPTI_REQUIRED_WORK_GROUP_SIZE%%
__kernel void solve_system(...)
```

The optimizes take the .cl file with the markers, replace them and writes a new syntactically correct file called calc_func_opti.cl. To tell the optimizer which optimization values are applied, the optimizer gets also a Python file as input. This file contains a Python dictionary. The keys of the dictionary are the name of the optimization, and the values to each key are the optimization factors.

For example, the following dictionary:

```
{"OPTI_NUM_COMPUTE_UNITS": 2,
    "OPTI_REQUIRED_WORK_GROUP_SIZE": 128,
    "OPTI_SIMD_WORK_ITEMS": 16,
   }
```

would create this code:

```
__attribute__((num_compute_units(2)))
_attribute__((num_simd_work_items(16)))
_attribute__((reqd_work_group_size(128,1,1)))
_kernel void solve_system(...)
```

The code is then ready for the TIDOWA compilation step. To apply compiler optimizations like the reordering of the floating-point operations, we add an array with flags to the dictionary with the key "additional_flags". Theses flags are added to the TIDOWA compilation step of the kernel. Thus it is possible to create any optimized code with any optimization factor of the optimization techniques. Of course, the restrictions of OpenCL must be adhered to, otherwise the compiler will raise an error.

# 5 Evaluations of Optimizations based on Kernel Reports

This part of the thesis first explains the different tests for the optimizations. There are two types of tests. The first one is the "default" test type, where similar optimizations are packed into one test. The second type of test is the "combined" test. These tests implement almost all optimizations with different optimization factors. With these implementations, we try to achieve the best possible performance. After the various tests have been defined, we look at the kernel report and compare the test cases' used hardware resources. We will compare the performance of these tests in Section **??**. The best performing implementation is compared to the run times of other hardware architectures.

## 5.1 Default Optimization Techniques

For the performance testing, the most meaningful test would be to allow the default implementation to compete against each optimization technique. Moreover, we could apply different optimization factors or pack several optimizations into one group. The benchmarks would then show which combination of optimization factors and techniques would produce the best results. Unfortunately, this would create an immense amount of compilation, and due to the long compilation time, this brute-force-method is not feasible for this thesis. Therefore, optimizations that are very similar or depend on each other are packed into one test. Which optimizations are grouped together is explained in the following sections. .

### 5.1.1 Default

The default implementation has almost no optimization. It only uses the "restrict" keyword in front of every kernel argument to tell the compiler there exists no pointer aliasing between the variables. For advantages of this keyword, see Section 4.3.6.

### 5.1.2 Static Loop Unrolling

There are three loops for which the number of iteration is known at compile time:

- Initialization of the initial value array
- Initialization of the parameter array
- Write-back of the results to global memory

Because of the small size of the loops, they are unrolled completely. Therefore, the loop overhead is eliminated.

### 5.1.3 Integration Loop Unrolling

The integration loop unrolling was placed in a separate test because it differs from the static unrolled loops. First of all, the trip count of the integration loop is typically much larger. For example, the oscillator equation has two equations and two parameters. Therefore, the initialization and write-back loops have a size of two. In contrast, the integration loop has, in some test cases, several thousand iterations. Moreover, while the size of the static loops is known at compile time, the integration loop can be of any size. For these reasons, we test the performance and hardware usage separately.

### 5.1.4 Kernel Attributes

The test combine all optimizations that are an attribute in front of the kernel. Therefore, the optimization for this test suite specify:

- Number of compute unis (see Section 4.3.3)
- Number of Single Instruction Multiple Data (SIMD) work-items (see Section 4.3.4)
- the required work-group size in section (see Section 4.3.2)
- global work offset

### 5.1.5 Floating-Point Optimization

The floating-point test suite combines the relaxed order of floating-point operations explained in Section 4.3.5 and the floating-point rounding optimization explained in 4.3.5. Compared to the default implementation, no changes were applied to the code, only the two flags “-fp-relaxed” and “-fpc” were added to the compilation command.

## 5.2 Combined Optimization Techniques

The following sections describe the tests that try to archive the best performance. For each test the sections explain which optimizations and optimization factors are used. In addition, it explicates the goal or reason of the test beside the performance.

### 5.2.1 Maximal Vectorization

For this test, the focus was on vectorization. With a SIMD factor of 16 the maximal possible value allowed by OpenCL was used. The number of compute units is set to 1, because for other values the compilation aborted with the error message that the design exceeds the hardware resources. Furthermore, to support vectorization we also need to specify a required work-group size. It was also adjusted to 128. Additionally, the global work offset is set to 0. The static loop unrolling explained in Section 5.1.2 is applied too. To fit the vectorization onto the FPGA we need to reduce the hardware usage. Therefore, the floating-point optimization of Section 5.1.5 is added to the compilation command.

### 5.2.2 Vectorization and Compute Units

This test try to compare the vectorization and the use of multiple compute units. It will be compared to the test of Section 5.2.1. While the other only use a vectorization with a factor of 16, this implementation has two compute units with a vectorization of 8 to keep the degree of parallelization the constant. Furthermore, to have a fair comparison the work-group size is also set to 128, the global work offset is 0, it uses the floating-point optimization and implements the static loop unrolling.

### 5.2.3 Unrolling Integration Loop with Floating-Point Optimization

In this test integration loop unrolling of Section 5.1.3 and the floating-point optimization is combined. The goal of this test is not to get a better performance than the default integration loop unrolling, the focus is to find out how much hardware can be saved for the unrolling implementation by using the the two compilation flags.

### 5.2.4 Vectorization, Compute Units and Unrolling

This test uses all optimizations but with smaller factors to fit into the FPGA. There are two versions of this test. The first one has two compute units, a vectroization of four and two as integration loop unrolling factor. In contrast the second version, it has three compute units. Both version also implement the static unrolling and use the

floating-point optimization. There are two implementations, because only of a large number of IVPs the second version can take advantage of the extra compute unit. For smaller problems sizes the overhead of the third compute unit could be larger than the additional computing power.

## 5.3 Kernel Report Comparison

During the compilation, the AOC creates a HTML report file. The kernel report has three main analysis types. Throughput analysis, area analysis and system viewers. In this section takes a deeper look at each of the kernel area analysis part. It describes the used hardware resources of each part of the kernel. Four hardware components are used for the analysis: Adaptive logic-up tables (ALUT), registers (FF), on-chip memory banks (RAMs) and the digital signal processors (DSP). The first diagram shows the static partition, which does not change for any compilation. The second diagram illustrates the used hardware resources for the default implementation. In the remaining bar chart graphs this default implementation serves as a comparison. The bars of the default version are always shaded.

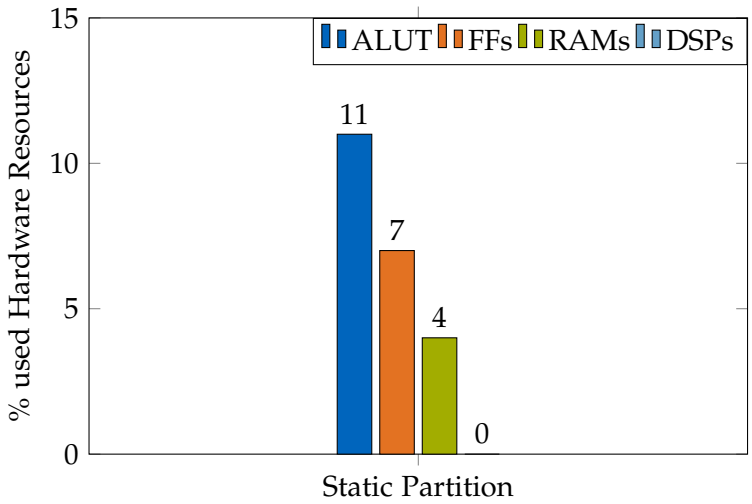


Figure 5.1: This diagram shows the portion of the total hardware resources used by the static partition. The static partition implements the platform interface logic. The size of it is independent of the compiled OpenCL kernel. Therefore, it stays the same for all compilations of the different optimizations and is left out in all other diagrams.

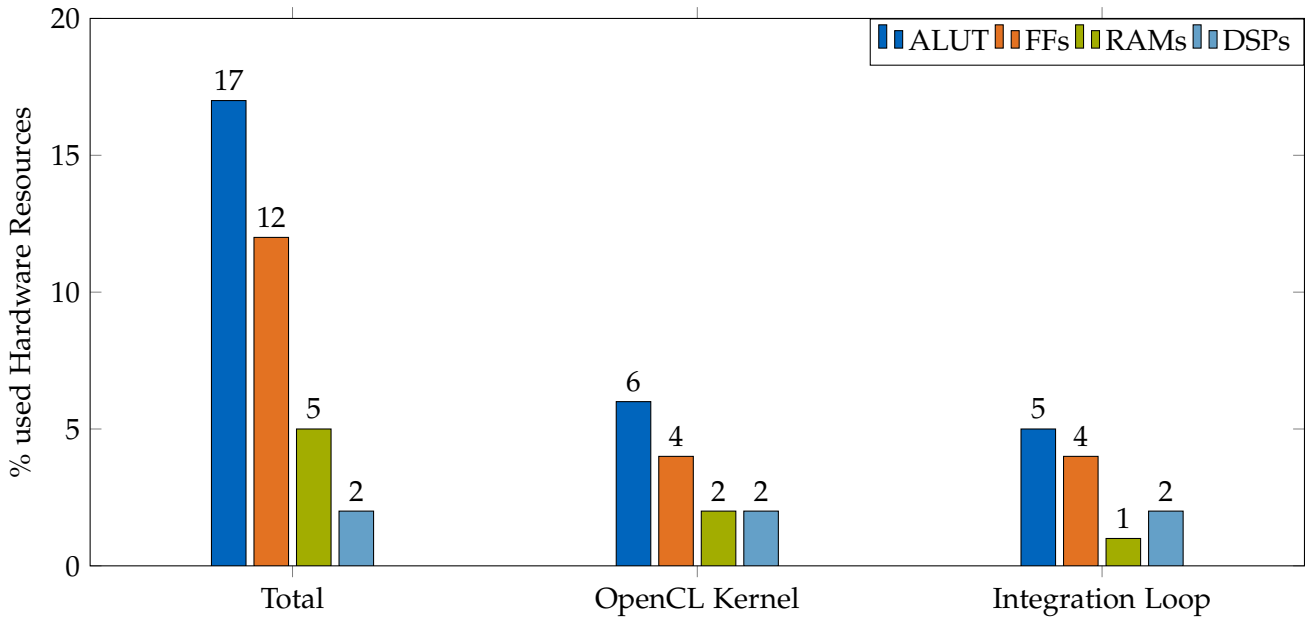


Figure 5.2: This bar chart diagram describes the used hardware in % for the default implementation. The total part is consisting of the static partition (see Figure 5.1) and the OpenCL kernel. The OpenCL kernel contains the following parts. The global interconnect, the indexing and initialization part and the integration loop. All diagrams only show the Integration Loop hardware utilization, because it consumes the major part of the kernel hardware. The largest part is made up of the static partition, but the relation between the hardware components of the OpenCL kernel shows that the critical resource are the ALUTs.

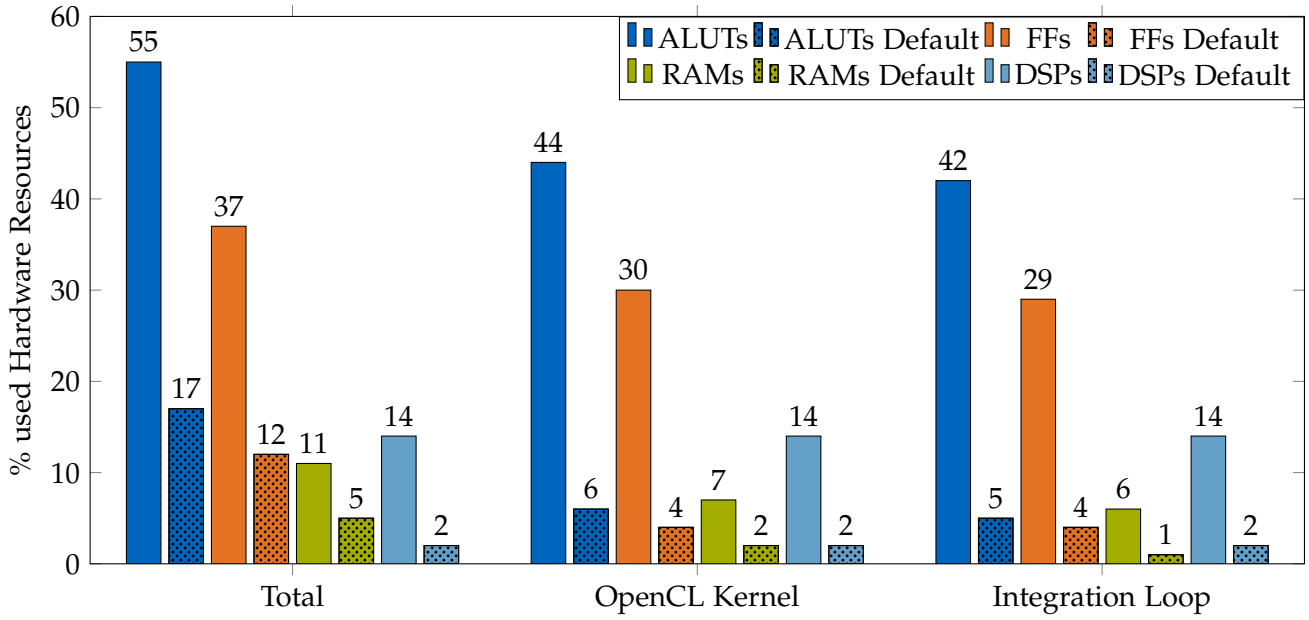


Figure 5.3: This bar chart diagram shows the used hardware resource in % of the following optimization: The number of compute units is set to one. The factor of the SIMD work-item attribute is 8. The global work offset is set to 0. Furthermore, to support vectorization we also need to specify a required work-group size. In this case this is also adjusted to 8. Without any other optimization this was the maximal verctorization that could have been archived. For a verctorization of 16 the compilation aborted with the error message that the design exceeds the hardware resources of the FPGA.

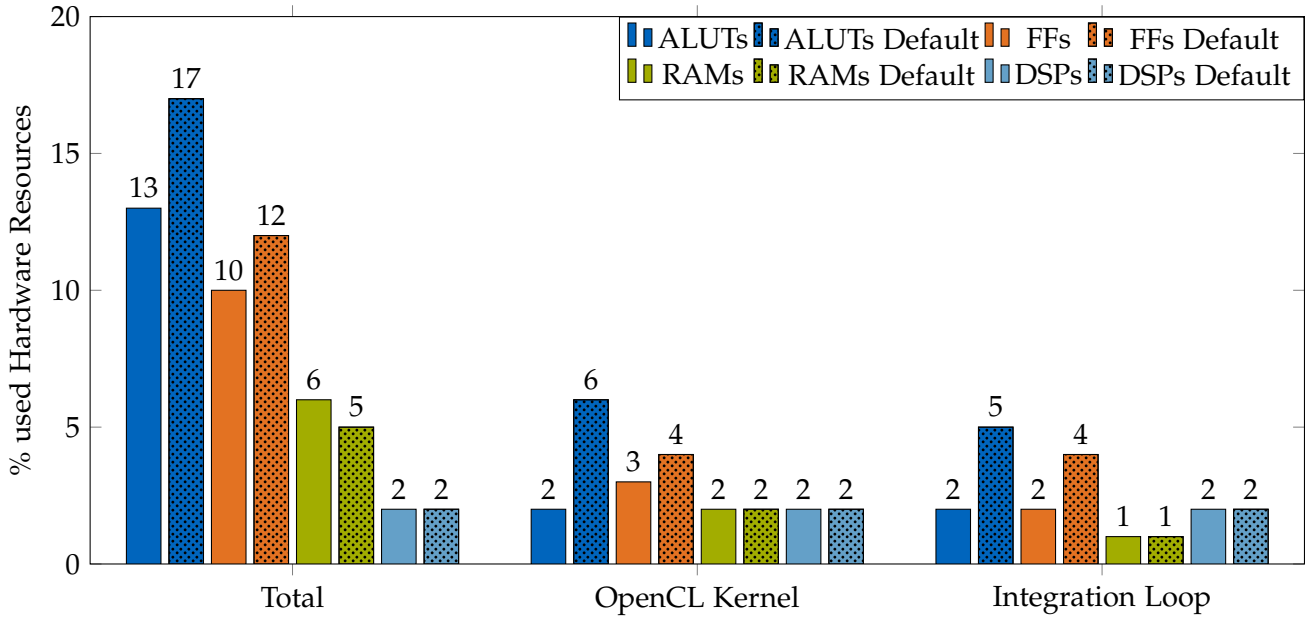


Figure 5.4: This bar chart diagram compares the hardware resource usage of floating-point optimization of Section 5.1.5 with the default implementation. It shows several interesting observations. The used hardware in total did not change significantly. This can be explained by the fact that the static partition makes up the most prominent part (see Figure 5.2), which stays the same for any optimization. For the OpenCL kernel and the integration loop, this observation changes. The integration loop only consumes a third of the ALUTs and half of the FF resources compared to the default implementation. The number of RAMs and DSPs is equal for the default and optimized implementation. These two results arise from the fact that the intermediate floating-point rounding mainly consumes logic like ALUTs and FF and not DSPs. DSPs are used for the floating-point operation itself. Furthermore, the percentage of used RAMs does not change, because they only store value and are not part of the calculation hardware. Since we only optimized the calculation part, the number of RAMs is equivalent.

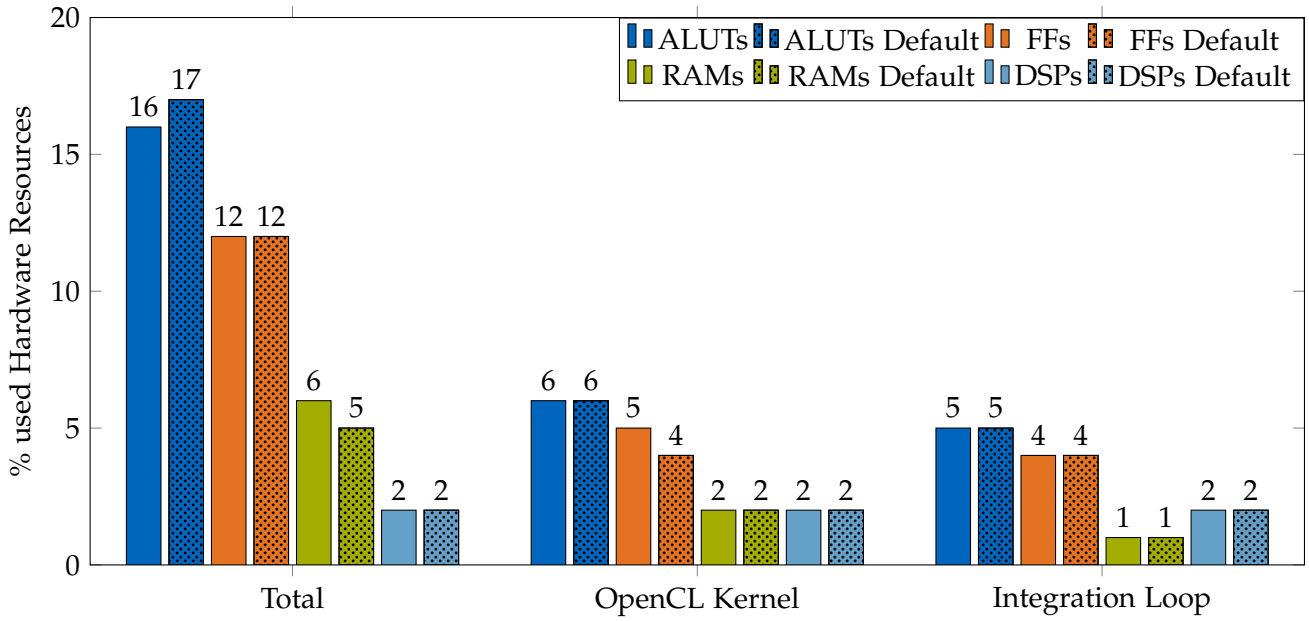


Figure 5.5: This bar chart diagram compares the hardware resource usage of floating-point optimization of Section 5.1.2 with the default implementation. The hardware usage for the unrolling of the static implementation did not changed very much. Only the total number of ALUTs decreased slightly. This small improvement is achieved in side the kernel. Unfortunately, due to the rounding of the percentage, we can not see the differences. In fact the unrolled version used 5,76%, while the default variant consumes 5,79%. Normally unrolling of loops increase the hardware usage, because the loop body is replicated. It is not the case here, because the loops are all unrolled completly and have a size of two. Therefore, the extra hardware of the second loop body, consumes slightly less resources than the control logic of the loop.

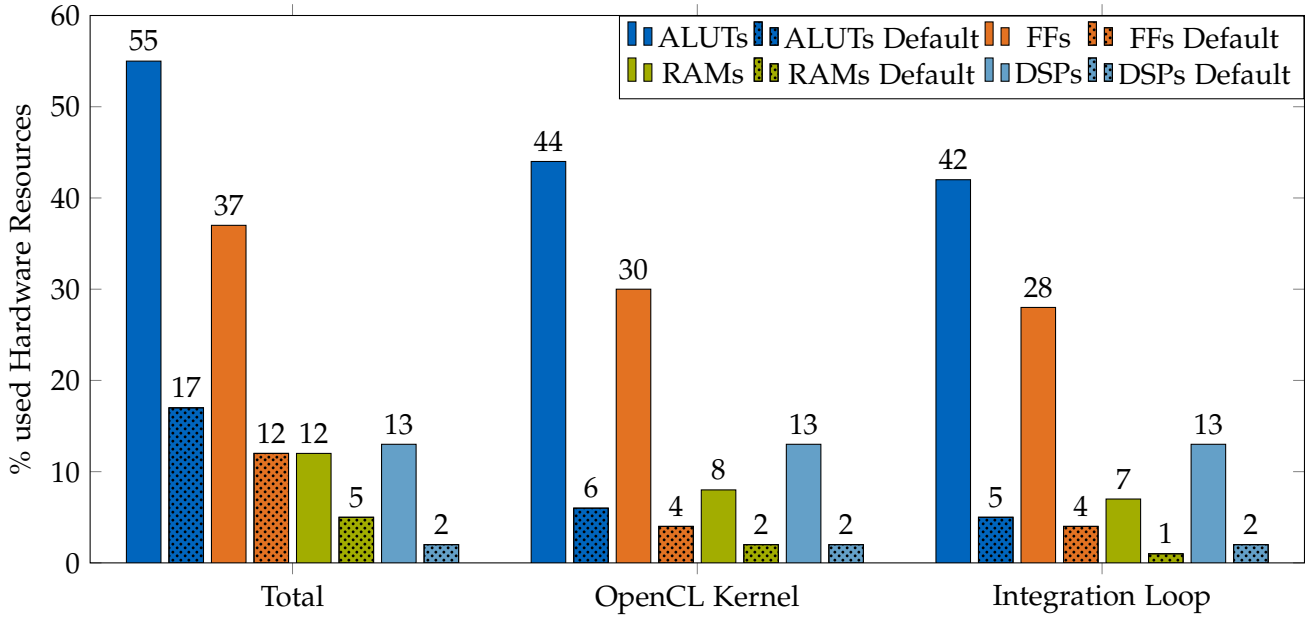


Figure 5.6: This bar chart diagram describes the change of used hardware resources if the integration loop is unrolled (see Section 5.1.3) with the default version as a comparison. The unrolling factor is set to 8. Therefore, the hardware structure of the loop body is replicated eight times. Due to the complex calculations of the integration step implemented in the loop body, the consummation of all resources increases clearly. The OpenCL kernel part shows that the new design uses fewer resources than eight times the default version. The fewer consummation has two reasons. First of all, only the loop body is replicated, and parts like the indexing stay the same. Besides, the computation potential can share some logic. A downside of this implementation is that for all tests, the loop's trip count needs to be a multiple of 8.

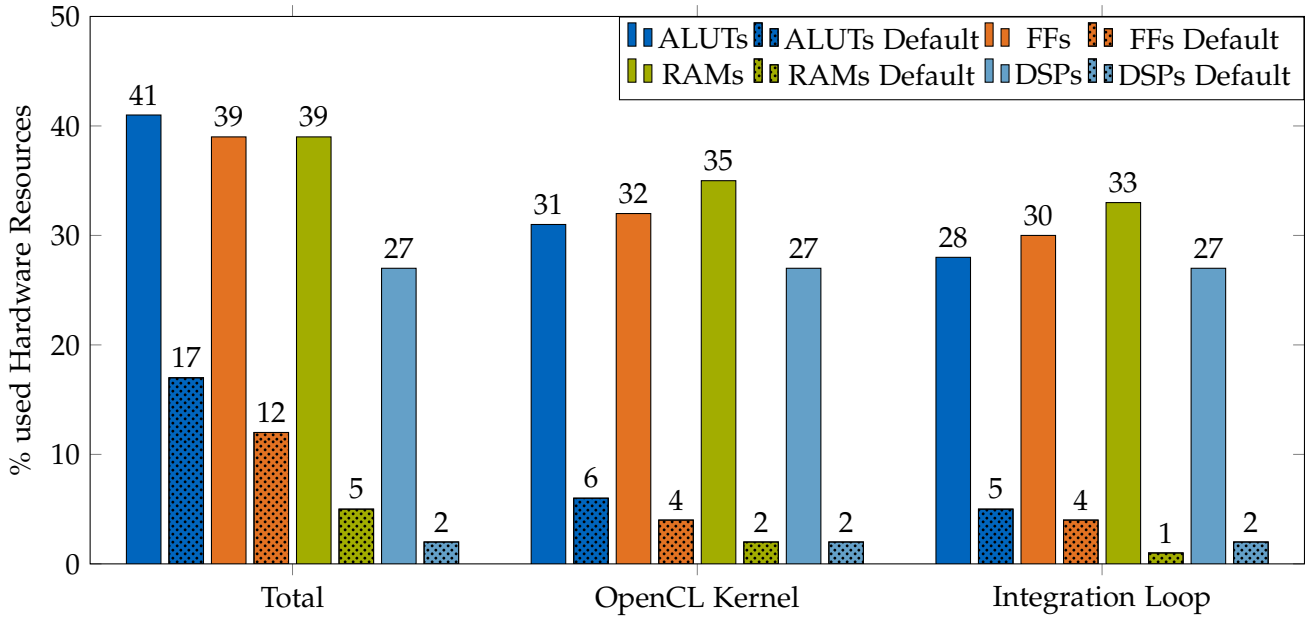


Figure 5.7: This diagram illustrates the hardware consumption of the optimized version described in Section 5.2.1. The key optimizations are the vectorization of 16, unrolling of all static loops, and the floating-point compilation flags. Because this design implements a large vectorization, it needs many more resources, but significantly less than 16 times the default version. This is possible due to the floating-point optimizations and the vectorization that allows the different hardware paths to share some logic. It is particularly noticeable that the overall hardware consumption is more evenly divided across the different components. Thereby the available ALUTs are no longer the critical resource.

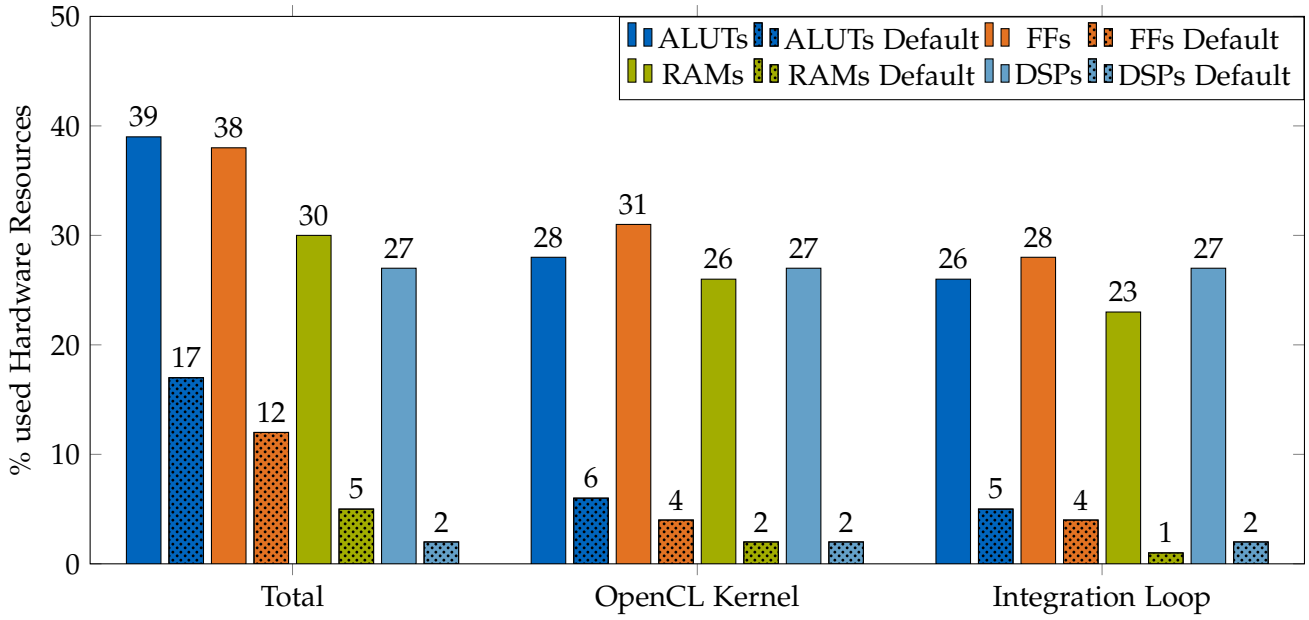


Figure 5.8: This bar chart shows the hardware consummation of the optimized version explained in Section 5.2.2. Like the version of Figure 5.7 it consumes less than 16 times the hardware of the default version for all three analysis points, except for the RAMs in the integration loop. In comparison to the resource usage of the version with vectorization only (illustrate in Figure 5.7), it consumes, except for DSPs fewer hardware resources, even if the parallel degree is the same. Therefore, vectorization only, is in terms of FPGA resource reduction, not the best option.

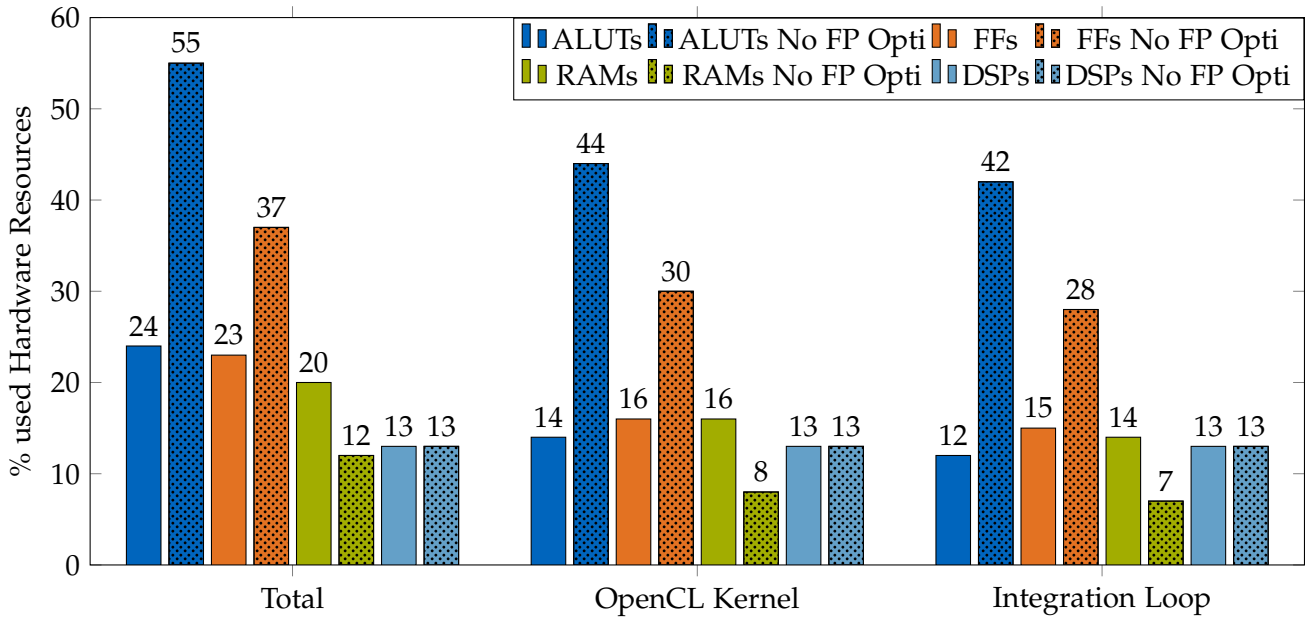


Figure 5.9: This bar chart diagram differs a bit from the others. It compares the hardware usage of the integration loop unrolling with and without the floating-point optimization. These two versions are explained in Section 5.2.3 and Section 5.1.3. The graph shows three significant observations. The floating-point optimization reduces the amount of used ALUTs and FFs tremendously. For the OpenCL kernel, the optimization more than thirds the number of ALUTs and almost halves the amount of FFs. In contrast, for the OpenCL kernel and the integration loop, the quantity of RAMs doubles, and the amount of DSPs stays precisely the same. In terms of hardware resource management, these observations show that it is helpful to apply the floating-point optimization because the ALUTs are the bottleneck of this design. On the other hand, for applications that already integrate many RAMs in the default design, unrolling with floating-point optimization could increase the critical portion.

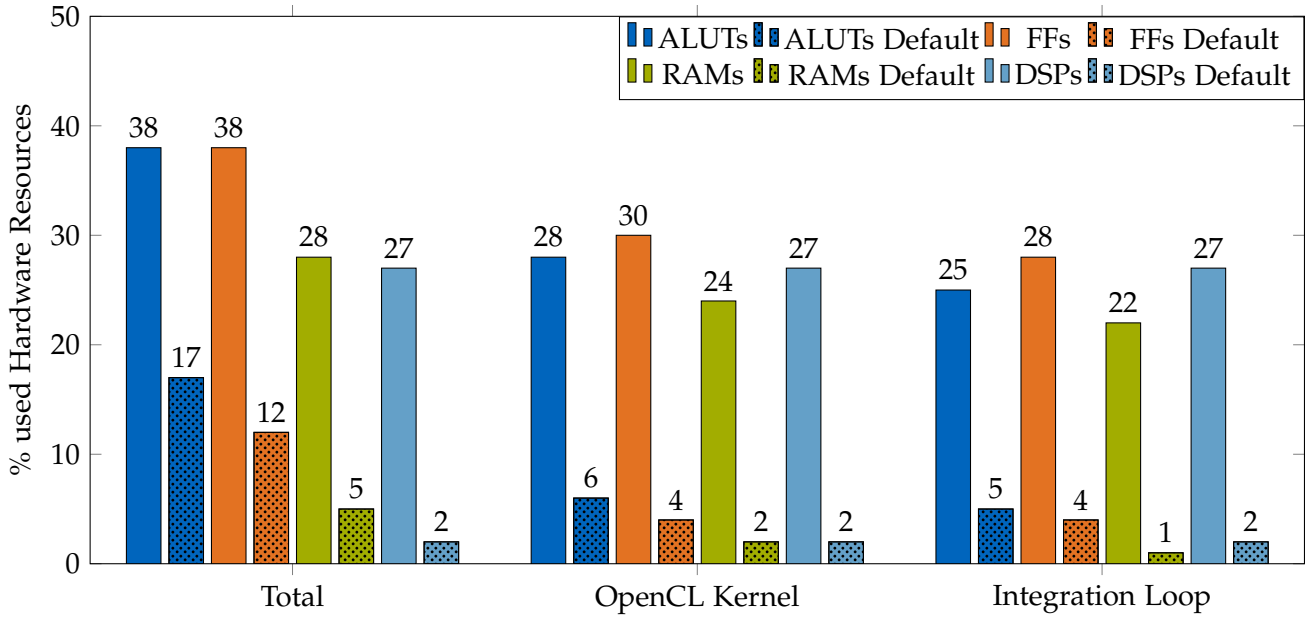


Figure 5.10: This diagram illustrates the hardware resources for the optimization version of Section 5.2.4 with two compute units. For the OpenCL Kernel part the diagram shows that used resources are very well distributed among all hardware components.

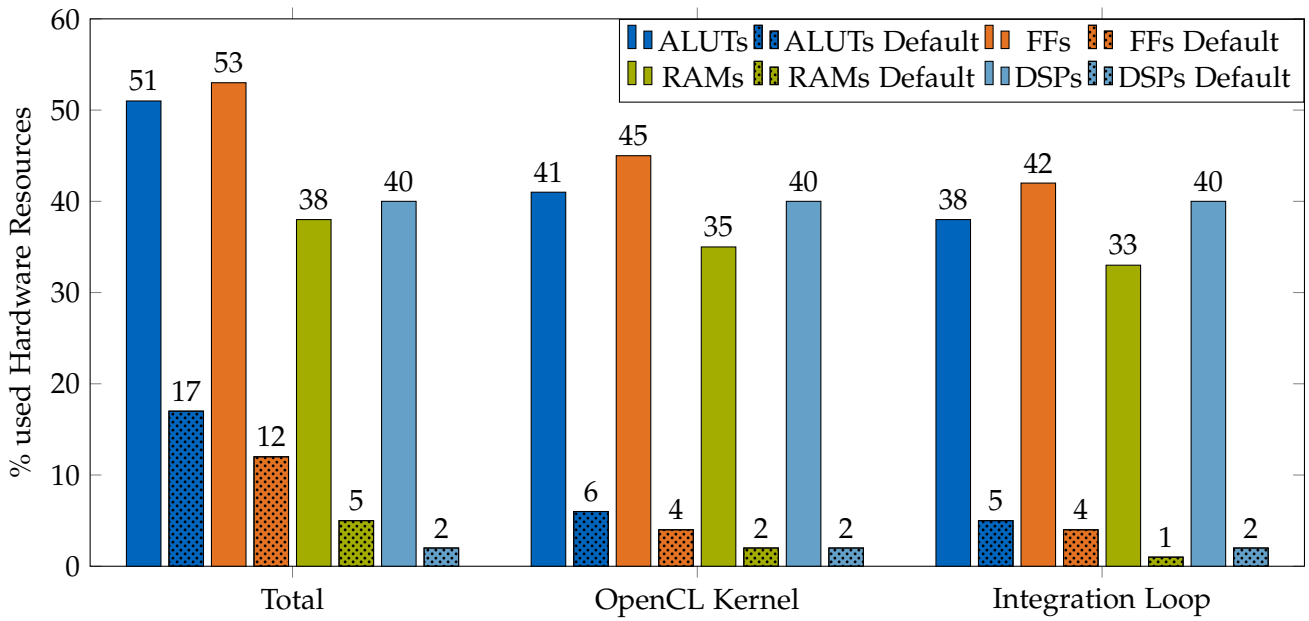


Figure 5.11: This diagram illustrates the hardware resources for the optimization version of Section 5.2.4 with three compute units. Compared to the other optimization the high DSP utilization is very significant.

We can draw several conclusions with the help of the diagram. First of all, it is interesting that the static partition always remains the same. The section observation is that the integration loop constantly consumes a major part of the hardware resources due to the complex floating-point operations. It is also noticeable that every optimization did not consume significantly more than 50% of any hardware component. Besides, the floating-point optimizations have a considerable impact on hardware utilization. Without the floating-point optimizations, many designs would not have compiled successfully because the distribution among the hardware components would have been very uneven.

# 6 Performance Testing

This chapter of the thesis deals with performance testing. First, we take a look a the testing environment that is used for the execution. The second section explains how these tests are performed in a repeatable, standardized way. The next part shows the different test cases. In the last chapter, the performance results are demonstrated for the different optimization techniques and run configurations.

## 6.1 Utilized FPGA System

For the performance testing, the FPGA System of the Ludwig-Maximilians-University was used. This system implements the Intel FPGA Programmable Accelerating Card D5005 with the device number BD-ACD-1SX280H2DES. The core is an Intel Stratix 10SX FPGA (1SX280HN2F43E2VG). The accelerating card contains four DDR4-SDRAMs memory banks with 8 GB each and a data rate of 2400 MT/s. The PCIe Gen3 x16 interface connects the accelerating device to the rest of the system. For further information about this accelerating card, see [21]. The system has two Intel(R) Xeon(R) CPUE5-2630L as CPUs with six cores and two threads each at a clock rate of @ 2,4GHz. As Random Access memory (RAM) it contains eight of the Hynix HMT41GR7AFR4A DDR3 memory with 8 GB each. The operating system is CentOS Linux.

## 6.2 Testing Procedure

A Python script realizes the automatic testing. All run configurations are stored in an array that contains Python dictionaries. These dictionaries have the following structure:

```
{ "num_combinations": 16,
"num_steps": 4096,
"deltaT": 0.01,
"begin_start_saving": 4096}
```

We execute the tests in different directories. The execution stores all results in subfolders. The script collects all execution times. It stores the values in separate arrays in a Python dictionary. The key is a string composed of the test name and the run configuration

values. This procedure is repeated for the specified number of runs. This dictionary is serialized and stored in a file with the Python package pickle. Therefore, it is later easier to transfer, load, and analyze the results.

## 6.3 Test Cases

Two variables define a run configuration—the number of initial values and parameter combinations and the number of integration steps. The size of the time step is equal for all tests, and the start saving variable always has the same values as the number of integration steps. Therefore, the solver only stores the values of the last integration step. This modification is applied because we only want to analyze FPGAs' core compute performance in this thesis. First tests have shown that external memory access with this design is a bottleneck. Consequently, the write-back of the results needs to be kept at a minimum. Optimizing the write operations would have been beyond the scope of this thesis. For a fair comparison, all GPUs' tests are also started with equal values for the number of steps and start saving point.
For the tests, we use the following values.

$$\#\text{IVPs} \in \{128, 1024, 4096, 8192, 16384\} \quad \#\text{ISs} \in \{128, 4096, 32768, 131072\} \tag{6.1}$$

Each value of the IVPs is combined whit every value of the ISs. Therefore, the overall number of run configurations is 20. Combining these values builds a wide range of tests. The different optimization techniques of sections 5.1 and 5.2 are executed with each run configuration. Each execution is started several times. With the different executions, we can calculate the average, maximal and minimal execution times.

## 6.4 Results

This section shows the execution time results for the different run configurations. The tables are arranged with an increasing number of IVPs and of ISs. Not every combination of IVPs and Integration step is presented because the difference for some run configurations are shallow. The diagrams only represent significant changes. Each bar graph is labeled with the used optimization. To fit into the diagram, the following abbreviations are used. Furthermore, the best performing version is marked in green and the worst in orange.

| Abbreviation | Explanation |
|---|---|
| CP=<value> | The number of used compute units (see Section 4.3.3 ) |
| VEC=<value> | Used vectorization factor (see Section 4.3.4) |
| IUR=<value> | The unrolling factor of the integration loop (see Section 5.1.3) |
| SUR | Static loop unrolling (see Section 5.1.2 ) |
| FP OPTI | Floating-Point optimization by applying the two compilation flags explained in 5.1.5 |
| DEFAULT | This references the default implementation |

Table 6.1: Abbreviations of the Optimizations

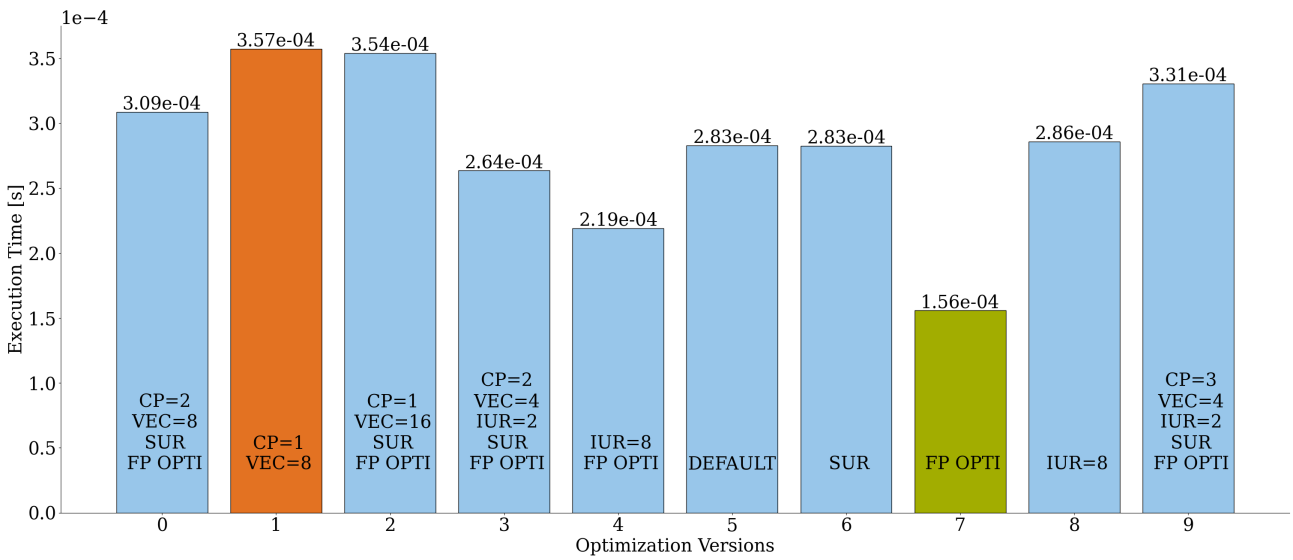


Figure 6.1: Execution Times of 128 Initial Value Problems and 128 Integration Steps

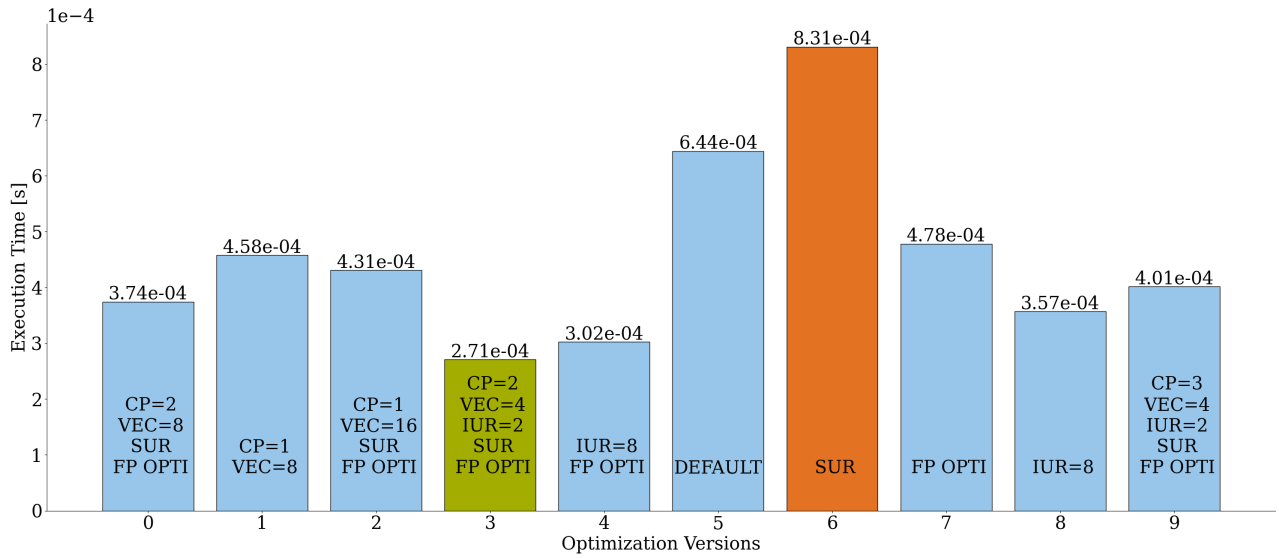


Figure 6.2: Execution Times of 1024 Initial Value Problems and 128 Integration Steps

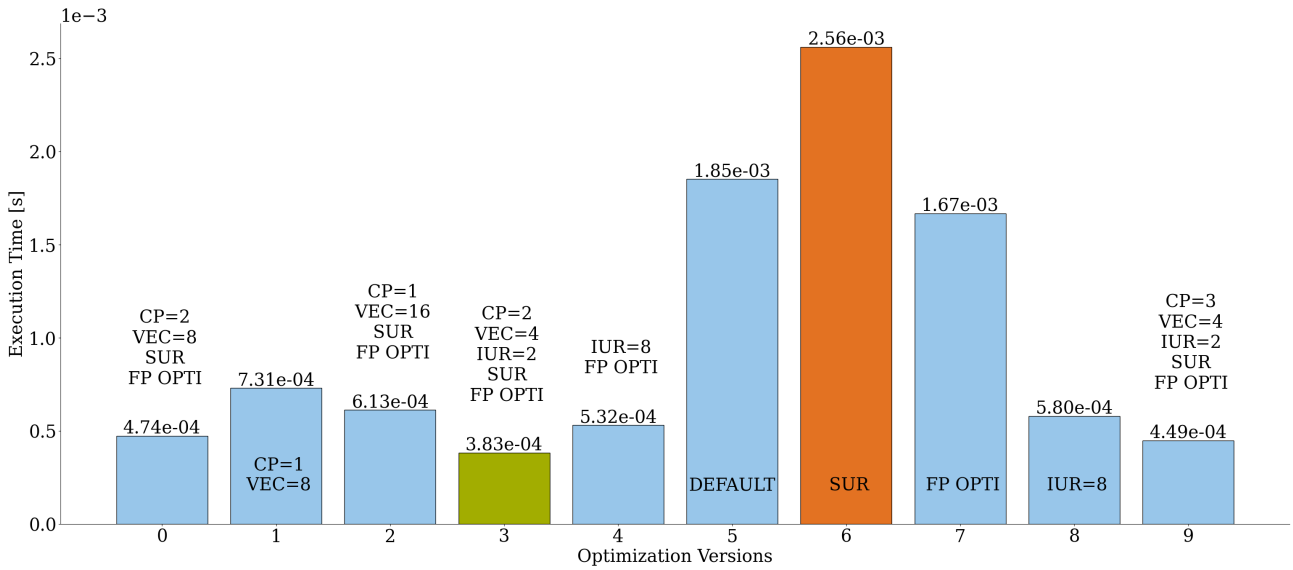


Figure 6.3: Execution Times of 4096 Initial Value Problems and 128 Integration Steps

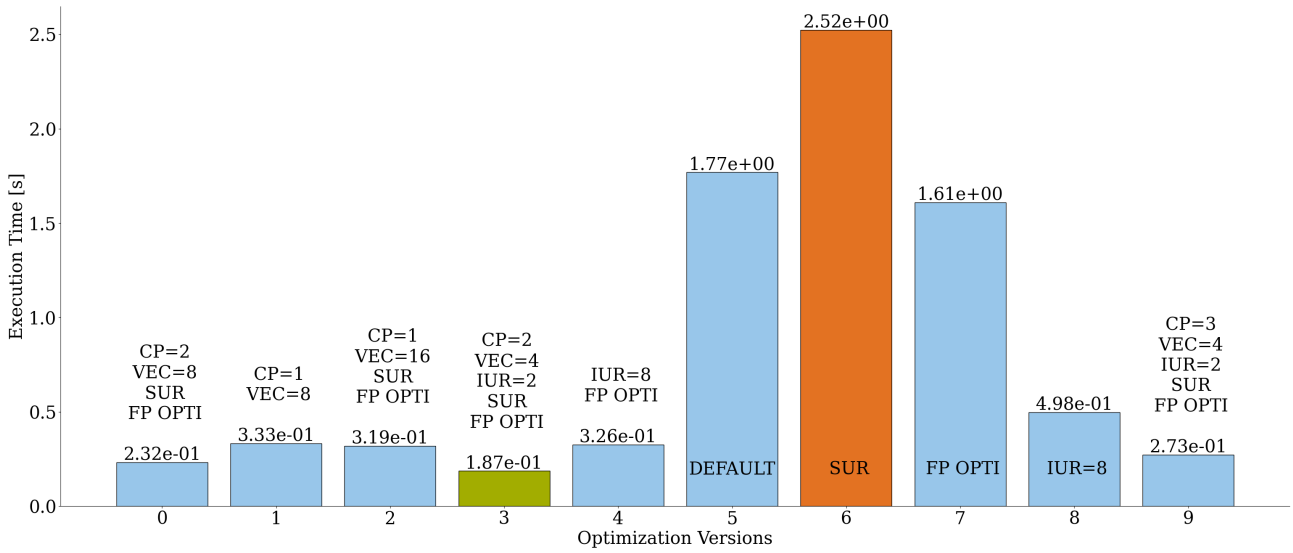


Figure 6.4: Execution Times of 4096 Initial Value Problems and 131072 Integration Steps

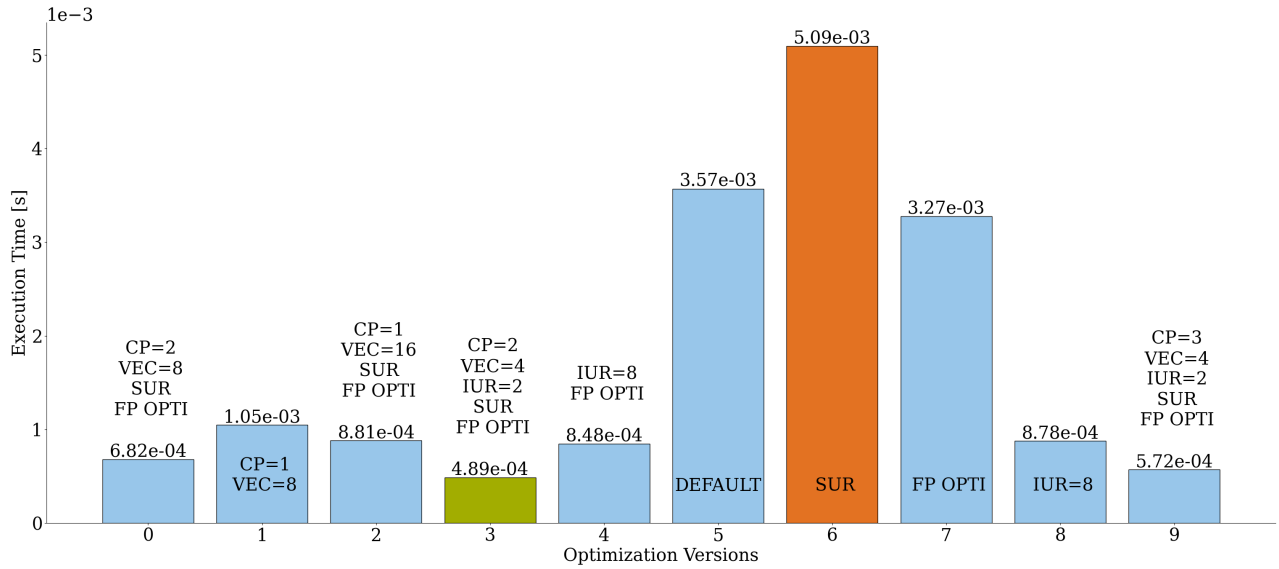


Figure 6.5: Execution Times of 8192 Initial Value Problems and 128 Integration Steps

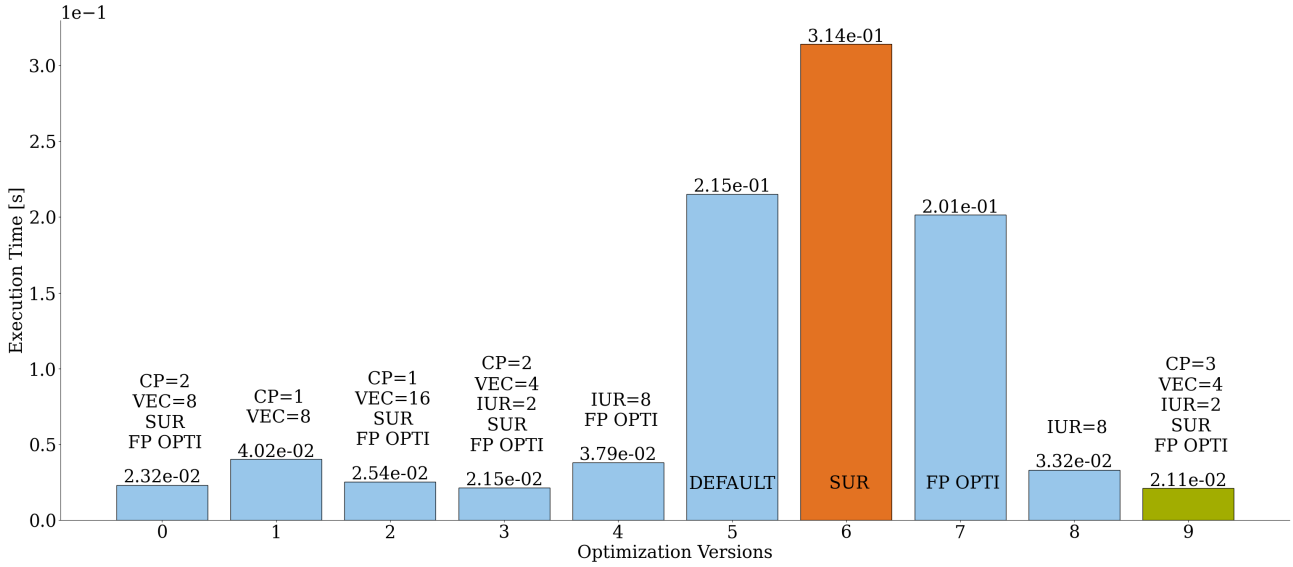


Figure 6.6: Execution Times of 16384 Initial Value Problems and 4096 Integration Steps

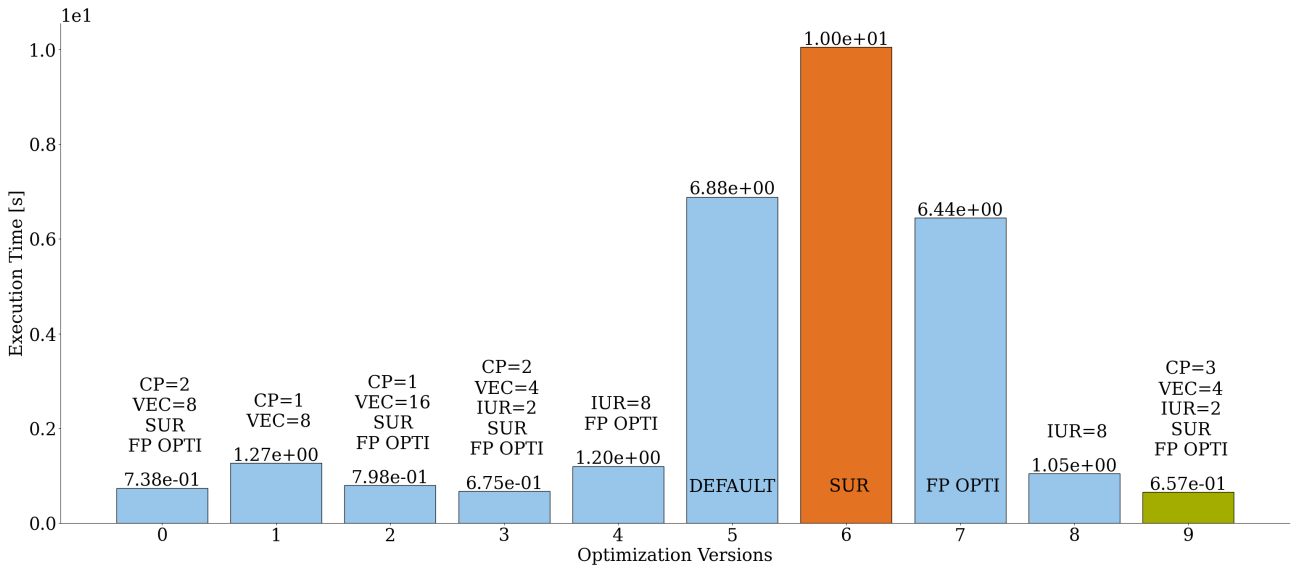


Figure 6.7: Execution Times of 16384 Initial Value Problems and 131072 Integration Steps

## 6.5 Discussion

This section analyses the run times of the different optimization test cases demonstrated in Section 6.4. Each of the following sections described one observation of the execution time bar graphs.

### Initial Observation

The first bar graph diagram of figure 6.1 shows the execution time for the smallest size with 128 Initial Value Problems and 128 Integration Steps. For this run configuration, the version that implements only the floating-point optimization performs the best. This stays the same for all tests with 128 IVPs . The diagram shows clearly that multiply compute units, vectorization, and unrolling increase the run time for this magnitude of IVPs and ISs. The vectorization of 8 has the highest execution time. For this small problem set, the version of $\{0, 1, 2, 3, 9\}$ can not take advantage of the added parallelism. The pipeline length is probably larger than the number of global work-items. Therefore, widening the pipeline with vectorization or adding more pipelines by specifying multiple compute units increases the pipeline's management overhead and empty states. This results in a longer execution time. The bad performance of these hardware configurations is equal for all tests with 128 IVPs. Only some of the versions with floating-point optimization can perform slightly better than the default implementation.

### First significant Change

The first significant change appears with the increase of IVPs to 1024, see figure 6.2. The problem size is now large enough to take advantage of the extra parallelism. The versions of $\{0, 1, 2, 3, 9\}$ performed previously worse than the default implementation, have now an immense shorter execution time. The new best version is number 3 with two compute units, vectorization of 4, integration loop unrolling of 2, static loop unrolling, and floating-point optimizations. Even if number 9 has more compute units, number 3 performs better for all run configurations with up to 16384 IVPs and 128 ISs.

### Static Loop Unrolling

The static loop unrolling is for this implementation with no global memory write-back unambiguously the worst optimization. Only for the first run configuration, it performs just as well as the default implementation. For every other number of IVPs and ISs, the static loop unrolling has a significantly larger execution time.

### Relative Differences for equal IVPs

This part looks at the run times for an equal amount of IVPs, but an increasing number of ISs. For this observation figure, 6.3 and 6.4 are relevant. The execution time for 5, 6, and 7 increased almost by the increasing factor of ISs. For example, in figure 6.3, the static unrolling has an execution time of 2.56e−03 seconds. 1024 is the increasing factor of the ISs between figure 6.3 and 6.4. The time multiplied with this factor is 2.62e+00. This value only deviates by 4% from the actual execution time of static loop unrolling in figure 6.4. The linear increase does not hold for the rest of the optimization versions. Therefore, for an equal number of IVPs, the relative difference increases with an increasing number of ISs.

### Relative Differences for increasing IVPs

The previous section exposed that the relative difference between $\{0, 1, 2, 3, 8, 9\}$ and $\{5, 6, 7\}$ increases with a larger number of IVPs and equal ISs. That this also applies the other way around shows the comparison of figures 6.3 and 6.5. This time the ISs keep equal, and the number of IVPs is doubled. The factor between the execution time of $\{5, 6, 7\}$, is almost equal to the incensing factor of the number of IVPs . Like in the previous section, this does not hold for version in $\{0, 1, 2, 3, 8, 9\}$. This results again in an increase of the relative differences. Therefore, with this two observation, it is clear that the performance difference increase with larger problem sizes.

### Best Version for maximal Problem Size

At a size of 16384 IVPs and 4096 ISs the best performing version changes to number 9. This version can take advantage of the extra compute unit at this problem size compared to number 3. For the rest of the run configuration, this version remains the best and is consequently the best variant for the hugest problems.

# 7 Cross-Architecture Comparison

## 7.1 CPU, GPU Description

The used hardware for the performance benchmarks is described in this section. For information about the FPGA system, see Section 6.1. The CPU and GPU tests were executed on the same system, made available by the Technical University of Munich. The system implements the AMD Ryzen Threadripper 2990WX as CPU and the NVIDIA QuadroP6000, and the NVIDIA Tesla K20 as GPUs. The CPU has 34 cores with 64 threads. Only the Quadro P600 was utilized for the GPU benchmarks because it has by far more compute power. Furthermore, the system implements 64 GB main memory.

## 7.2 Results

This section compares the execution times on different hardware systems. Each plot illustrates the run times for an increasing number of initial value problems with a constant size of integration steps. All run times for the FPGA are of one optimization version to keep the comparison fair. The implementation with two compute units, vectorization of 4, unrolling of 2, floating-point optimization, and static loop unrolling, was used. Of course, it is not the best performing version for every run configuration. For example, the default implementation outperforms it for small problem size, and the optimization with three compute units runs faster for the largest problems. Nevertheless, it is, on average, the best overall run configurations. We did not change the used versions for different problems because the CPU and GPU implementations were also not optimized based on the problem size. For the CPU execution, the OpenMP implementation was used. The GPU was programmed with OpenCL. Each point of the graph has three components a minimal, maximal, and average value. The minimal and maximal values are illustrated as the shaded area behind the graph. The average value is marked with a dot.

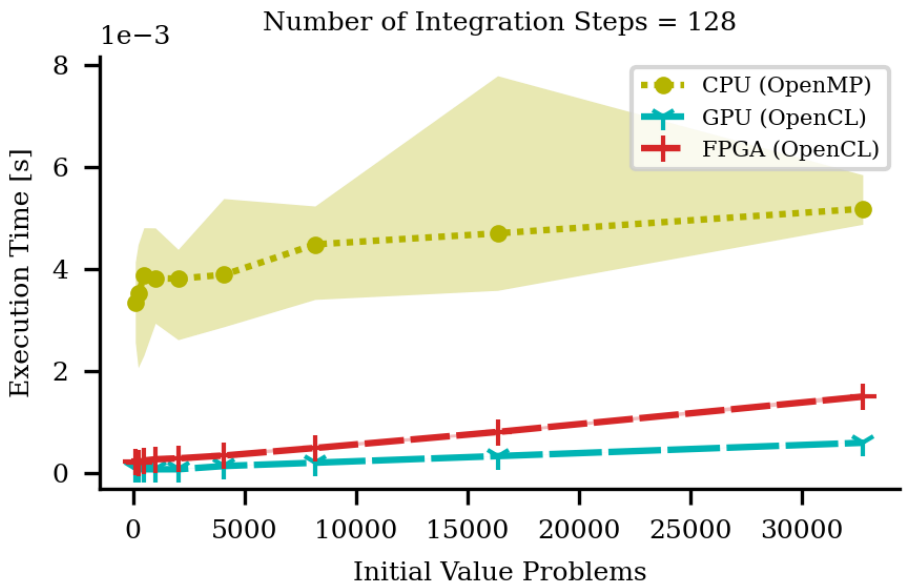


Figure 7.1: GPU, CPU and FPGA Execution Times for 128 Integration Steps

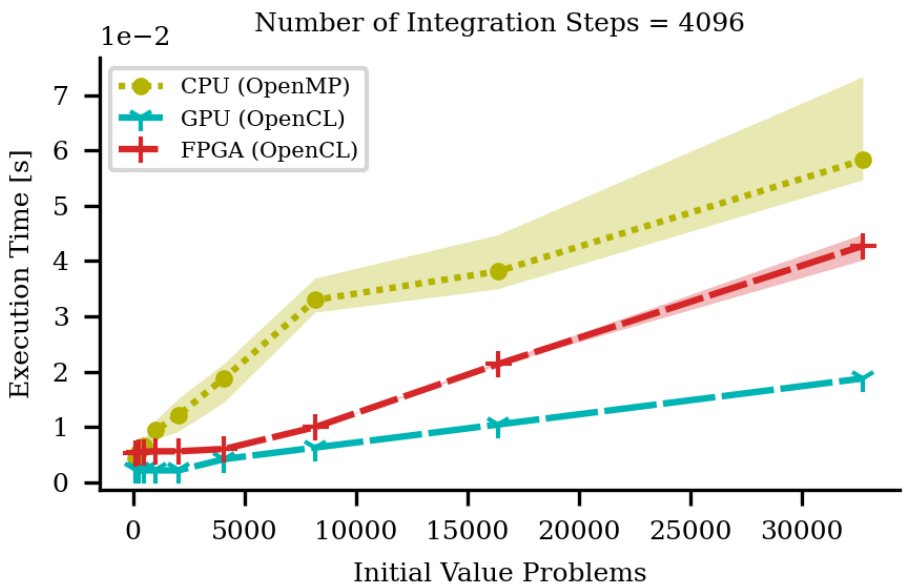


Figure 7.2: GPU, CPU and FPGA Execution Times for 4096 Integration Steps

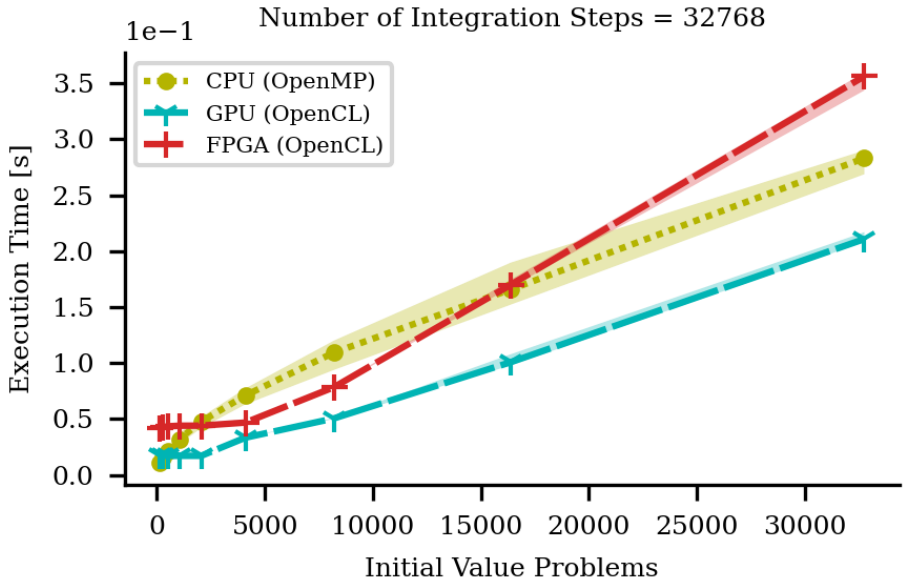


Figure 7.3: GPU, CPU and FPGA Execution Times for 32768 Integration Steps

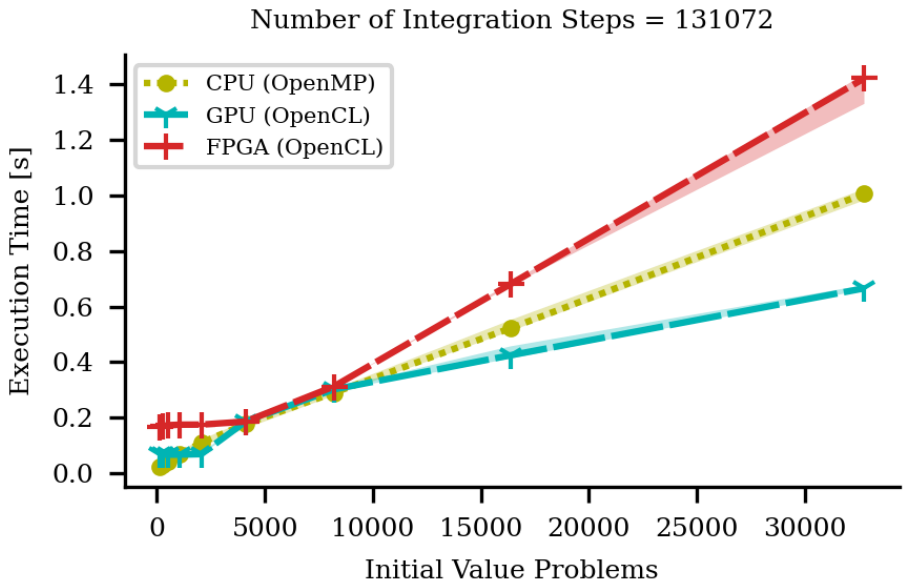


Figure 7.4: GPU, CPU and FPGA Execution Times for 131072 Integration Steps

## 7.3 Discussion

This part of the thesis discusses the performance results of section 7.2 for the different hardware architectures. It is important to know that an out-of-the-box study created the results. Therefore, these results show only the first tendencies for choosing the best fitting architecture for each problem size. In order to be able to draw clear conclusions, further analyzes are essential.

### 7.3.1 Deviations

The execution time graphs show clearly that the CPUs have the most significant deviation for almost every run configuration. Only for 131072 integration steps and 32768 IVPs the FPGA execution time is a bit more volatile. Very uncommon are the immense differences of the CPU run time in minor problems in Figure 7.1, mainly because the FPGA and GPU have almost no deviation. The GPU has, on average, the most consistent execution times. Only for a larger number of integration steps, like in Figure 7.3 or 7.4 the results slightly differ.

### 7.3.2 Performance

For a small or middle sized number of integration steps, like 128 or 4096, the GPU and FPGA performance better than the CPU, see Figures 7.1 and 7.2. For 128 integration steps, this performance difference is even more significant. The CPU can only keep up at this problem size for the 4096 integration steps and a very tiny number of IVPs like 128. For a larger number of integration steps, the CPU performance increases relative to the FPGA and GPU. Figure 7.3 shows that for 32768 integration steps, the CPU outperforms the FPGA for all problem sizes smaller than 2048 or greater than or equal to 16384. For an even larger number of integration steps, there is no interval at which the FPGA runs faster than the CPU. Only for 4096 and 8192 ISs both perform equally. This observation is illustrated in figure 7.4. The GPU-CPU comparison for this larger size of integration steps shows that the CPU can only perform slightly better than the GPU for a very small number of IVPs. For an increasing number of IVPs, the GPU runs again faster than the CPU. For almost every run configuration, the GPU has shorter execution times than the FPGA. While the performance differences are very significant for a larger number of IVPs and integration steps, this is not true for smaller problem sizes. For 131072 integration steps with 4096 or 8192 IVPs the FPGA performs as well as the GPU.
In general, the result shows that for almost every problem size the GPUs have the best performance or are very close to the best performing system. Furthermore, the results

point out that the FPGA can keep up with the other system for some run configurations in core compute power. Unfortunately, we did not manage to get an execution time of the FPGA which is significantly better than those of the other hardware architecture. A deeper investigation of this will be part of future work including a comparison with other performance tests like energy efficiency.

# 8 Further Research

The main focus of this thesis was to extend the TIDOWA system to support FPGAs. Furthermore, it tested the performance of different optimization techniques and compared the results with CPU and GPU implementations. The results of the comparison form a reasonable basis for further research. FPGAs are a relatively new topic in this research area. Therefore, there are many opportunities to continue this work. Which topics future work can address are explained in the following sections.

## 8.1 Optimizing Memory Access

The current performance testing only stores the last value of the integration. This restriction was made to test the core compute performance. Initial tests have shown that the global memory access in every iteration of the integration loop is a clear bottleneck. The version with vectorization and multiple compute units could not perform significantly better than the default implementation. To provide good run times for executions that store every value, the global memory access needs to be optimized. For example, this could be achieved by adding a local buffer. The calculated values are not written back directly to global memory. The local buffer stores them temporally until it is full. The buffer is then written back to global memory. This modification reduces the access to global memory significantly and could increase the performance.

## 8.2 NDRange vs. Single Work-Item

The implementing of OpenCL for FPGA uses the NDRagne model. Future work could extend this to support also the single work-item model. It would be interesting to research the best fitting model for each ODE system and run configuration.

## 8.3 Fixed- vs. Floating-Point Calculation

The kernel report comparison in Section 5.3 has shown that the major part of hardware resources is consumed by the floating-point operations in the integration loop. This

limits the factor of vectorization, unrolling, or the number of compute units that can be applied to the kernel. There are two options to reduce this resource consumption. The current implementation uses a double-precision representation. This could be changed to single-precision. The second option is to switch to a fixed-point representation. Both versions would use tremendously less hardware than the double-precision implementation. This opens up the opportunity to increase the factor of vectorization, unrolling, or the number of compute units. Of course, it is indispensable to compare the accuracy of the different versions.

## 8.4 Detailed Performance Profiling

The Intel FPGA SDK for OpenCL offers the option to implement performance counts into the design. These counts collect data during the kernel execution. With the Intel VTune Profiler, this data can be analyzed to find the bottleneck of the implementation. Profiling the performance in detail becomes especially important for the global memory access optimization explained in Section 8.1.

# 9 Conclusion

Heterogeneous architectures become increasingly important for HPC. To solve a problem as quickly and efficiently as possible, it is necessary to select the best fitting component of a heterogeneous system. Which architecture is best suited heavily depends on the type of problem. Therefore, it is essential to research this.
The focus of this thesis was to develop an ODE solver optimized for FPGAs and integrate the implementation into the TIDOWA system. Furthermore, it was the goal to test the relevance of these solvers compared to other systems.
This thesis initially explained how FPGAs are constructed and showed what the market for FPGAs currently offers. Further, it demonstrated how development tools like the Intel FPGA SDK for OpenCL can be used. It expounded which optimizations can be applied to FPGA specific OpenCL code. Furthermore, the thesis illustrated how these optimizations affect the used hardware resources and accelerate the execution. The results showed that a combination of different techniques performs better than using only one optimization with a high optimization factor. These outcomes build a sound basis for further implementations and research. By analyzing the used hardware resources, it is now easier to predict whether a compilation of a new design completes successfully.
The results were then compared to those of the CPU and GPU execution. The comparison revealed that for smaller problems, the FPGA implementation performed better than the CPU and almost as good as the GPU implementation. In contrast, for larger problem sizes, the FPGA gets outperformed by the two other architectures. The results were crated by an out-of-the-box study, therefore to draw clear conclusions, further research is required. The results only show a first tendency to which problem each architecture fits the best. This thesis has built an essential foundation in developing efficient ODE solvers for FPGA systems.

# Bibliography


[1] *A New FPGA Architecture and Leading-Edge FinFET Process Technology Promise to Meet Next-Generation System Requirements*. WP-01220-1.4. Intel. 2019.

[2] E. Avramidis and O. E. Akman. "Optimisation of an exemplar oculomotor model using multi-objective genetic algorithms executed on a GPU-CPU combination." In: *BMC systems biology* 11.1 (2017), pp. 1–23.

[3] S. Bals. "Development of a domain-specific language for the efficient time integration of ODEs." In: (2019).

[4] V. Betz. *FPGA Architecture for the Challenge*. URL: https://www.eecg.utoronto.ca/~vaughn/challenge/fpga_arch.html. (accessed: 02.03.2021).

[5] P. Biswas. *Introduction to FPGA and its Architecture*. URL: https://towardsdatascience.com/introduction-to-fpga-and-its-architecture-20a62c14421c. (accessed: 02.03.2021).

[6] H.-J. Bungartz. *Lecture notes in "Numerisches Programmieren" (IN0019)*. 2019.

[7] M. Fuchs. "Geschichte und Einführung in Aufbau und Arbeitsweise von FPGA." In: (2003).

[8] Intel. *Intel Agilex I-Series SoC FPGA Product Table*. URL: https://www.intel.de/content/dam/www/programmable/us/en/pdfs/literature/pt/intel-agilex-i-series-product-table.pdf. (accessed: 14.04.2021).

[9] Intel. *Intel FPGAs*. URL: https://www.intel.de/content/www/de/de/products/programmable/fpga.html. (accessed: 10.04.2021).

[10] Intel. *Intel Stratix 10 DX FPGAs*. Intel. URL: https://www.intel.de/content/www/de/de/products/programmable/sip/stratix-10-dx.html. (accessed: 19.03.2021).

[11] Intel. *Intel Stratix 10 DX Product Table*. 2019. URL: https://www.intel.de/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-dx-product-table.pdf. (accessed: 19.03.2021).

[12] Intel. *Intel Stratix 10 FPGA Features*. Intel. 2019. URL: https://www.intel.de/content/www/de/de/products/programmable/fpga/stratix-10/features.html. (accessed: 17.03.2021).

[13] Intel. *Intel Stratix 10 GX FPGAs*. Intel. URL: https://www.intel.de/content/www/de/de/products/programmable/fpga/stratix-10/gx.html. (accessed: 19.03.2021).

[14] Intel. *Intel Stratix 10 GX/SX Product Table*. URL: https://www.intel.de/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-product-table.pdf. (accessed: 17.03.2021).

[15] Intel. *Intel Stratix 10 MX (DRAM System-In-Package) Product Table*. URL: https://www.intel.de/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-mx-product-table.pdf. (accessed: 19.03.2021).

[16] Intel. *Intel Stratix 10 MX FPGAs*. Intel. URL: https://www.intel.de/content/www/de/de/products/programmable/sip/stratix-10-mx.html. (accessed: 19.03.2021).

[17] Intel. *Intel Stratix 10 NX FPGAs*. Intel. URL: https://www.intel.de/content/www/de/de/products/programmable/fpga/stratix-10/nx.html. (accessed: 19.03.2021).

[18] Intel. *Intel Stratix 10 SX SoC FPGAs*. Intel. URL: https://www.intel.de/content/www/de/de/products/programmable/soc/stratix-10.html. (accessed: 19.03.2021).

[19] Intel. *Intel Stratix 10 TX FPGAs*. Intel. URL: https://www.intel.de/content/www/de/de/products/programmable/fpga/stratix-10/tx.html. (accessed: 19.03.2021).

[20] Intel. *Intel Stratix 10 TX Product Table*. 2019. URL: https://www.intel.de/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-tx-product-table.pdf. (accessed: 19.03.2021).

[21] *Intel FPGA Programmable Acceleration Card D5005*. Intel. 2021. URL: https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/intel-fpga-pac-d5005/documentation.html. (accessed: 06.04.2021).

[22] *Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide*. UG-OCL003. Intel. 2020.

[23] *Intel FPGA SDK for OpenCL Pro Edition Getting Started Guide*. UG-OCL001. Intel. 2020.

[24] *Intel FPGA SDK for OpenCL Pro Edition Programming Guide*. UG-OCL002. Intel. 2020.

[25] *Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide*. UG-S10LAB. Intel. Apr. 2020.

[26] T. Lei. "Optimization of Parametrized High-Dimensional ODE Simulations." In: (2020).

[27] mikrocontroller. *FPGA*. URL: https://www.mikrocontroller.net/articles/FPGA. (accessed: 02.03.2021).

[28] R. Miron. *FPGA-Grundlagen: Funktionsweise und Einsatzmöglichkeiten*. URL: https://www.elektronikpraxis.vogel.de/fpga-grundlagen-funktionsweise-und-einsatzmoeglichkeiten-a-906696/. (accessed: 02.03.2021).

[29] A. Munshi, ed. *The OpenCL Specification*. Khronos Group, 2009.

[30] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno. "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis." In: *IEEE Access* 5 (2017), pp. 2747–2762.

[31] A. van der Ploeg. *Why use an FPGA instead of a CPU or GPU?* URL: https://blog.esciencecenter.nl/why-use-an-fpga-instead-of-a-cpu-or-gpu-b234cd4f309c. (accessed: 02.03.2021).

[32] A. Sanaullah and M. C. Herbordt. "Fpga hpc using opencl: Case study in 3d fft." In: *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. 2018, pp. 1–6.

[33] M. Schreiber. *Lecture notes in Selected Topics in Algorithms and Scientific Computing (IN3400, IN3480)*. 2020.

[34] techpowerup. *NVIDIA GeForce RTX 3090*. 2020. URL: https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622. (accessed: 18.03.2021).

[35] M. Vestias and H. Neto. "Trends of CPU, GPU and FPGA for high-performance computing." In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2014, pp. 1–6.

[36] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson. "Comparing hardware accelerators in scientific applications: A case study." In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (2010), pp. 58–68.

[37] Q. Wu, Y. Ha, A. Kumar, S. Luo, A. Li, and S. Mohamed. "A heterogeneous platform with GPU and FPGA for power efficient high performance computing." In: *2014 International Symposium on Integrated Circuits (ISIC)*. IEEE. 2014, pp. 220–223.

[38] Xilinx. *UltraScale+ FPGAs Product Tables and Product Selection Guide*. URL: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf#VUSP. (accessed: 19.03.2021).

[39] Xilinx. *Xilinx FPGAs*. URL: https://www.xilinx.com/products/silicon-devices/fpga.html. (accessed: 10.04.2021).

[40] Xilinx. *Xilinx Virtex UltraScale+*. Xilinx. URL: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html. (accessed: 19.03.2021).

[41] Xilinx. *Xilinx Virtex UltraScale+ 58G*. Xilinx. URL: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-58g.html. (accessed: 19.03.2021).

[42] Xilinx. *Xilinx Virtex UltraScale+ HBM*. Xilinx. URL: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-hbm.html. (accessed: 19.03.2021).

[43] Xilinx. *Xilinx Virtex UltraScale+ VU19P*. Xilinx. URL: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-vu19p.html. (accessed: 19.03.2021).

[44] C. Yang, J. Sheng, R. Patel, A. Sanaullah, V. Sachdeva, and M. C. Herbordt. "OpenCL for HPC with FPGAs: Case study in molecular electrostatics." In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2017, pp. 1–8.

[45] H. R. Zohouri. "High performance computing with FPGAs and OpenCL." In: (2018).