Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2023

Evaluation of FPGA-based high performance computing platforms

Martin F Lundgren

Master of Science Thesis in Electrical Engineering

Evaluation of FPGA-based high performance computing platforms

Martin F Lundgren

LiTH-ISY-EX–23/5544–SE

Supervisor: **Jose Nunez-Yanez**
ISY, Linköpings universitet

Examiner: **Oscar Gustafsson**
ISY, Linköpings universitet

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

High performance computing is a topic that has risen to the top in the era of digitalization, AI and automation. Therefore, the search for more cost and time effective ways to implement HPC work is always a subject extensively researched. One part of this is to have hardware that is capable to improve on these criteria. Different hardware usually have different code languages to implement these works though, cross-platform solution like Intel's oneAPI framework is starting to gaining popularity.

In this thesis, the capabilities of Intel's oneAPI framework to implement and execute HPC benchmarks on different hardware platforms will be discussed. Using the hardware available through Intel's DevCloud services, Intel's Xeon Gold 6128, Intel's UHD Graphics P630 and the Arria10 FPGA board were all chosen to use for implementation. The benchmarks that were chosen to be used were GEMM (General Matrix Multiplication) and BUDE (Bristol University Docking Engine). They were implemented using DPC++ (Data Parallel C++), Intel's own SYCL-based C++ extension. The benchmarks were also tried to be improved upon with HPC speed-up methods like loop unrolling and some hardware manipulation.

The performance for CPU and GPU were recorded and compared, as the FPGA implementation could not be preformed because of technical difficulties. The results are good comparison to related work, but did not improve much upon them. This because the hardware used is quite weak compared to industry standard. Though further research on the topic would be interesting, to compare a working FPGA implementation to the other results and results from other studies. This implementation also probably has the biggest improvement potential, so to see how good one could make it would be interesting. Also, testing some other more complex benchmarks could be interesting.

Acknowledgments

I would like to thank everyone that has helped me write this thesis. My examiner and supervisor owes special thanks for all the good feedback and endless answering to questions. A big thanks to all my friends in my class for makes this process, even though done separately, feel fun which have taken me through the slumps of motivation that comes with this kind of project. Lastly I want to thank my family which have been my pillar to lean on, not only through this project but all of my education. Thank you!

Linköping, January 2023
Martin F Lundgren

Contents

List of Figures

List of Tables

Notation

Abbreviations

Abbreviations	Meaning
GEMM	General Matrix Multiplications
HPC	High Performance Computing
FPGA	Field-programable gate array
GPU	Graphics processing unit
CPU	Central processing unit
DPCPP	Data parallel c++
SYCL	Cross-platform abstraction layer for c++
ALUT	Adaptive look up table
FF	Flip-flops
RAM	Random access memory
MLAB	Memory logic array block
DSP	Digital signal processors

1 Introduction

In this day and age, something that is getting more and more relevant is High Performance Computing (HPC). It is used in everything from solving complex maths problems in machine learning and AI[23], to simple matrix operations on a gigantic scale. We use this in every part of our very tech reliant society, which is why it is very important that we have hardware and code that can keep up with the demand. To do this, the use of normal CPUs have been slowly phased out for the more effective GPUs and FPGAs[15]. GPUs with its easy parallelization have been the go-to choice for many, but FPGAs with its low latency and low power consumption have slowly been rising in popularity[25]. The difference in code and hardware structure has made it a problem for many developers to jump in between different hardware platforms. So more and more single source platforms like Intel's oneAPI DevCloud system are gaining popularity. The oneAPI DevCloud offers the Jupyter IDE to work in, SYCL based DPC++ compilers and other tool kits to easily start your own HPC development career[19].

1.1 Aim

The aim of this thesis is to, with the help of the oneAPI framework, analyse how fast and power efficient the different hardware platforms are with some HPC benchmarks. To accomplish this two benchmarks were chosen to implement: GEMM or General matrix multiplication and BUDE or Bristol University Docking Engine. The aim can then be separated into questions to be answered by this thesis.

1.1.1 Question

The questions to be answered by this thesis:

- Which hardware platforms performs better for the two benchmarks with the oneAPI system?
- Can the different algorithms be improved with some HPC speed-up methods?

1.2 Delimitations

This project is specifically made with the limitations of the oneAPI toolkit. Which means that the hardware the benchmarks will be run on is the ones given access to through the DevCloud environment. The hardware used will therefore be the Arria 10 FPGA board, the UHD Graphics P630 GPU and the Xeon Gold 6128 CPU. Other delimitations are the use of only two benchmarks, and sizes of data used to be kept to reasonable sizes. These limits were chosen for time limitations purposes.

2

Background

To start off this thesis, some concepts and terms needs to be explained. To do that, this section will make a deep dive into some of the things this thesis will touch upon.

2.1 Hardware

In order to understand why there is a difference in performance between the different hardware, you have to understand how the different hardware are structured and works. So in this section they will be explained in more detail to give more insight on the ones that will be used during this thesis.

2.1.1 CPU

A CPU, or a Central Processing Unit, is what is known in common tongue as a processor. It is the central point of all general purpose computers, easily said it is the brain of the computer. With some input it will do a task which then produces an output. A flowchart model can be seen in Figure 2.1. A CPU usually has very complex computational logic and instructions, so that it can do everything. Examples may be handling different sort of number representations, like integers and floating-point numbers, and complex mathematical operations like square root and division[10]. These number representations behave very differently from each other, so if, for example, you want to multiply integers there are different instructions compared to multiplying floats. The many different instructions, complex logic and high clock rate makes the CPU drain a lot of power and take up a lot of area. The high resource consumption (larger area uptake) and the heat accumulation are therefore the driving factors on why the CPUs parallelization options are limited, which is the primary method to speed up computers

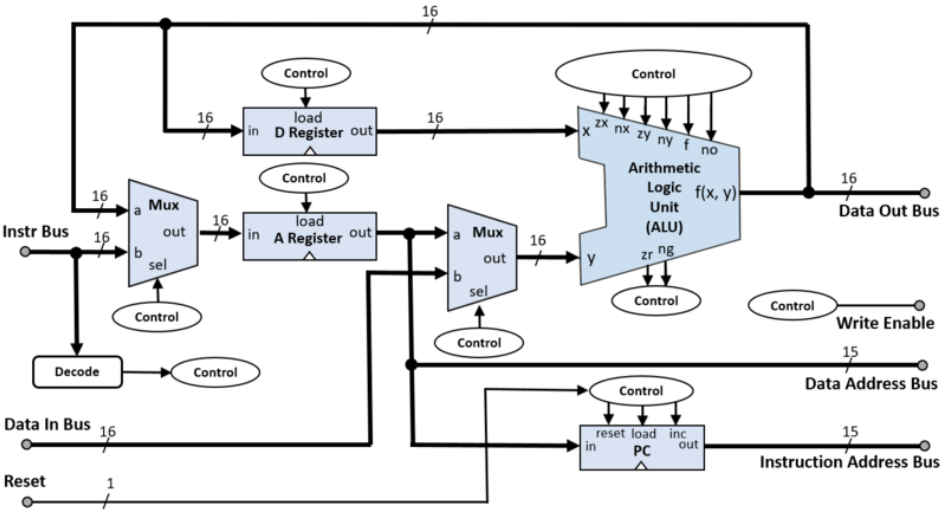

Figure 2.1: *A Basic CPU model from* `https://commons.wikimedia.org/wiki/File:Hack_Computer_CPU_Block_Diagram.png`*, Credit: Rleininger / $CC_BY - SA - 4.0$*

now that we are reaching the physical limit on both clock frequency and size of transistors.

One solution to do this problem is a multicore design[9]. With a multicore design, you have more than one computing node or core, so you can do more than one calculation at a time. This means that the CPU can, for example, do many small operations independent of each other in the same clock cycle. This technique, and a few others like hyperthreading and instruction pipelining, have increased the performance of high-end computing significantly in the last decades.

2.1.2 GPU

The GPU, or Graphical Processing Unit, is a so-called accelerator. As said in the name, it was made for one purpose, which was speeding up graphical processes in computers. This is to offload these heavy computational processes from the CPU. The fundamental differences between a GPU and a CPU is that while a CPU has a few very complex cores that focus on task parallelism, a GPU consists of thousands of small simple cores with a few well tuned instructions[5, 16]. A comparison can be seen in Figure 2.2.

This means that a GPU cannot preform all the complex computations a CPU can, but can make many simple tasks in parallel. The GPUs usually operate on the SIMD, or Single Instruction Multiple Data, architecture. Which means every single core gets the same instructions to do, but gets different data to do it on. This makes GPUs very fast on simple but computational heavy tasks like linear alge-

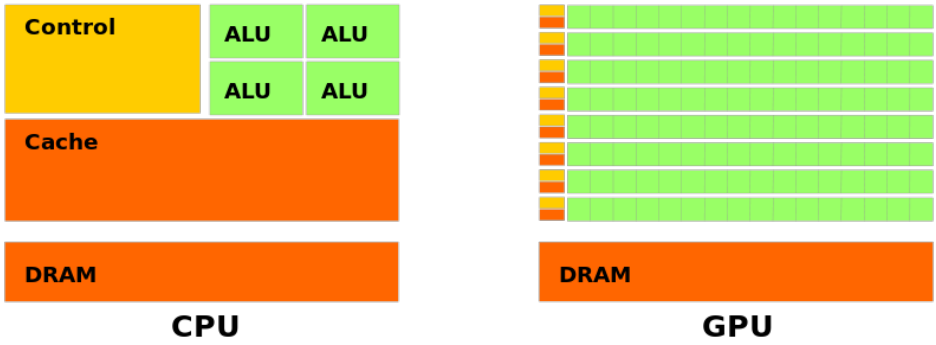

Figure 2.2: *GPU vs CPU architecture compared* `https://commons.wikimedia.org/wiki/File:Cpu-gpu.svg`*, Credit: PavelNajman / CC-BY-3.0*

braic projections, rotations and mirroring. But lately people realize that GPUs can also be used to do the computational heavy lifting in HPC, for example in matrix multiplication and machine learning, by utilizing these features.

2.1.3 FPGA

FPGAs, or Field Programmable Gate Arrays, are a more flexible ASIC (Application Specific Integrate Circuit). ASICs are hardware specifically made for one purpose and by doing that are very well-equipped and fast for that certain task. But they are therefore both slow to produce and very rigid in their design. FPGAs were developed to fill that hole between, easier to produce and more flexible during the design stage[25]. They were marketed to both help with development when designing new ASICs, but were also shipped in products before fabrication of the ASICs were done, to later be replaced by the new chips. Later on, when FPGAs got faster, cheaper and less power hungry, they stopped replacing them for new ASICs and just used the FPGAs as the main designs instead.

FPGAs are an interconnected network of configurable logic blocks and I/O connections. FPGAs also have features like memory, ALUs (Arithmetic Logic Unit) and decoders available. Each logic block contains a set of multiplexers, flip-flops and logic gates, enough to make up a small state machine by its own. With this set up, the FPGA can be configured to have almost any complex function by first configuring the logic block for different tasks in the function, then configuring the interconnection to fit the design pattern. By making a circuit with minimal functionality, memory and area usage outside the intended purposes, the FPGA can minimize both power usage and heat generation and maximize speed and I/O bandwidth. These qualities make FPGAs more suitable for HPC purposes compared to other hardware, like GPUs, which have a very high power usage and heat generation.

2.2 Hardware in HPC

In modern times our performance needs have only become higher and higher with the advancements in AI, biomedical engineering, and more[23]. These trends put more and more pressure for better hardware to keep up with the demand. Because of its parallelism capabilities, GPUs surpassed conventional CPUs in HPC implementations in the early 2000s [11]. The GPU has since then been the benchmark accelerator to compare all others to. Lately, the power consumption and heat dissipation become problems which have the HPC community in rising concerns. The accelerator rising to the occasion is the FPGA. With its customizable logic blocks and interconnections, it can minimize power consumption without losing computing power [22]. Studies show that depending on which HPC benchmark is considered, GPUs perform better or worse than FPGAs [24]. This depending on how the implementations utilize different strengths of different hardware. Determining the of best performing hardware has to be done on a case by case basis, though a good hypothesis can be done by analysing how the implementations use the different hardware strengths.

2.3 oneAPI DPCPP

Intel's oneAPI system is made for cross-platform development, using its DPCPP SYCL-based language. SYCL is a C++ open source cross-platform programming model for heterogeneous computing [3]. SYCL is based on a few simple models for execution and memory handling. The model uses a host, usually a CPU, for controlling the flow of execution and a device which is the hardware that will execute the submitted task. For execution, SYCL uses a queue based model where the program submits a task to a queue for execution, then in sequence the device selected for the queue will execute the kernel in the queue. This model enables many queues execute in parallel, but the programmer has to make sure that dependencies are satisfied. This can be done in many ways, like using **depends_on** statements or by using buffers and accessors. Buffers and accessors are a part of the memory handling in SYCL, where buffers handle the memory while the accessors handle access to the buffers them and dependencies to each of them. First there are two types of memory in SYCL-based programming, USM or Unified Shared Memory where the host share memory with a device, and second by using buffers (a memory object which can be shared between devices). When using buffers, you access them through something called accessors. These accessors can be specified to **read**, **write** or **read and write** to buffers and are specified in each task. Accessors are only defined in the scope of the task, but by specifying **read** or **write** action that will be done to the buffers, the run-time can be optimized for which tasks in the same queue can be executed in parallel and which have data dependencies on each other.

Tasks that are submitted are usually in one of two forms. **Parallel_for** tasks are for-loops that can be executed so that each iteration of the loop is executed in

parallel. These loops are run most successfully on GPUs because of their unique capabilities for parallelism, but can also be executed on CPUs. In **Parallel_for**-loops, a range must be included, which can be n-dimensional (nd-range). There can also be subgroups where we can divide each part into smaller nd-ranges and execute the problem in even smaller sizes with a divide and conquer method. The other form of task is the **single_task** form. **single_task** executes just once and is not using any form of parallelism. It is very useful for FPGAs which can utilize its fast logic to build deep pipelines for efficient data reuse and more optimized compute density but without using too much area. It is also useful for some CPU tasks[6].

Based on SYCL Intel developed Data parallel C++. It is a language and compiler for their related processor systems. This means it can be written and compiled to any of the chosen Intel hardware of CPUs, GPUs, FPGAs, and other accelerators. DPCPP is an extension of SYCL which can be downloaded to Linux, MAC, or Windows for private use or be used through the Intel oneAPI DevCloud. This is a huge leap for heterogeneous programming, to use one code for many different platforms[18]. DPCPPs goals of giving the best performance for every hardware with the same code structure makes it easy for any developer with some understanding of C/C++ structure to pick up and use.

DPCPP uses three different scopes. The first and highest level of abstraction is the application scope where the code for the host is executed, like memory allocation and printing results. The second is the command group scope, where kernel functions and host code are used to launch it on the device. Lastly, the kernel scope, where the kernel function is executed[4]. In the application and command group scopes all standard C++ functionality is kept, but some restrictions are set in the kernel scope.

2.4 GEMM

General Matrix Multiplication, or GEMM for short, is a benchmark based in linear algebra. It is a very suitable benchmark for HPC evaluations on different hardware for its easy scalability and for use with different number representations such as floats, integers and even complex numbers. GEMM is also optimizable through memory allocation by having the matrices transposed or not. This can make kernels execute faster by faster memory access [12].

$$C = \alpha \times op(A) \times op(B) + \beta C \tag{2.1}$$

In equation 2.1 a mathematical representation of GEMM is shown. Where A, B and C are matrices and α and β are scalars. Op() is specifying if the matrices are transposed or not by inputs set by the caller [12].

In previous studies done [17] on this benchmark has shown that Intel's HARPv2 FPGA hardware Outperformed NVIDIA's Pascal GPU when implementing GEMM

[17]. In some of the experiments of the study, it shows that the FPGA outperform the GPU by 50 times. Another study made with Zynq UltraScale+ MPSoC device made improvements on both dense and sparse matrix multiplication with almost a 20 time improvement, compared to Google's own NEON RUY library made for optimized GEMM calculations[14].

Because of its wide use in everything from neural networks and artificial intelligence to gaming, GEMM is very interesting for this study. The benchmark is both easy to implement and interesting, with deep roots in HPC.

2.5 Bristol University Docking Engine (BUDE)

With the rise of HPC, many new applications have been discovered. In the biomedical field, one such thing is molecular docking. Molecular docking is a method where the structure of a molecule is modelled when interacting with different proteins, shown in Figure 2.3. This process has had a gigantic impact in the medical industry, especially when discovering new drugs and using this method to predict different molecules interactions with the body[20]. This method is very computational heavy, so doing it on serial CPUs can take days, but with the emergence of GPUs and FPGAs this time could be cut down by as much as 36 times[21].

Bristol University Docking Engine, or BUDE, is a docking algorithm developed at Bristol University for research purposes. It was originally developed in OpenCL, but have since then been made available in many different implementations [2]. One of these implementations is in SYCL which is the one this thesis will build upon.

Earlier studies on this have been done on Intel's Arria10 HARP FPGA, where some promising results have been made, which is one of the reasons for this study [1]. With the similarities of the hardware run on DevCloud it could be interesting to see how it will compare to the other studies. With the focus on suitable benchmarks to test Intel's DevCloud service on, this benchmark has merit.

2.6 Speed-up options

There are many ways to speed up an HPC process which utilizes all the properties of the used hardware. We could improve on the benchmarks from their original forms with some of these theories.

2.6.1 Loop unrolling

The first method is loop unrolling. This is the process of doing more than one iteration of a process per iteration of the loop. Say that a loop has the purpose

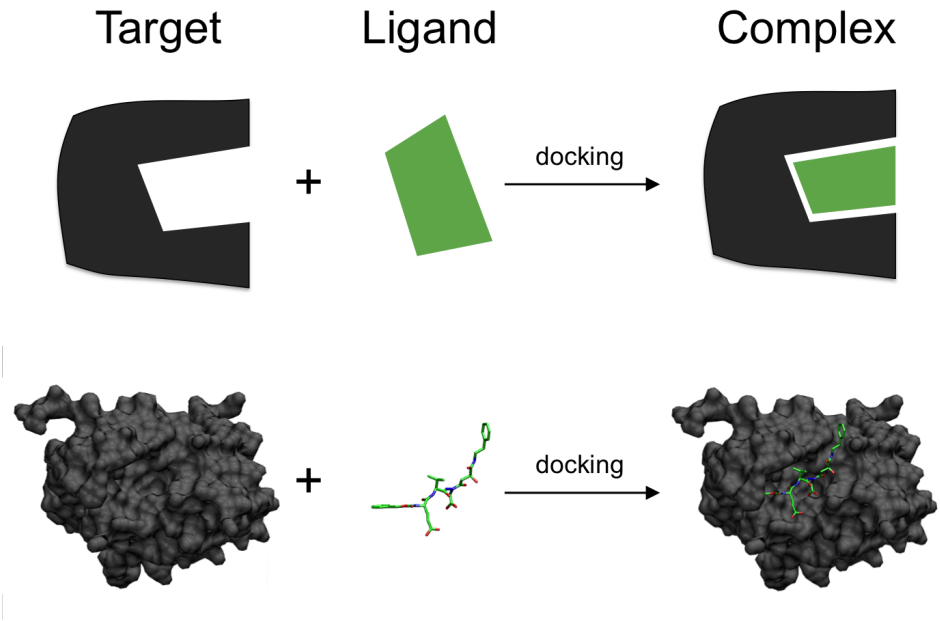

Figure 2.3: *Illustrations of molecular docking from* `https://commons.wikimedia.org/wiki/File:Docking_representation_2.png`, *Credit: Scigenis / $CC_BY - SA - 4.0$*

of assigning every element of a matrix to zero. A conventional loop would be written something like this:

Listing 2.1: *Loop example*

```
vector<int> A(64);
for( int i = 0, i< sizeof(A), i++){
    A[i] = 0;
}
```

But when doing and unrolling, it would look something like this:

Listing 2.2: *Unrolled loop example*

```
vector<int> A(64);
for( int i = 0, i< sizeof(A)/2, i++){
    A[2*i] = 0;
    A[2*i+1] = 0;
}
```

Unrolling loops benefits processors who can utilize pipe lining techniques. The performance boost from this method comes from that every iteration of the loop can be pipelined, and more processes can be done in the same clock cycle. It also reduces the amount of compares that has to be done for each iteration to check if it should step out of the loop. Utilizing this method would use more of the hardware in question, so one has to see to not unroll a loop too many times to see that it does not use more resources than the hardware provides.

2.6.2 Subdividing groups

Another method to heighten the performance is to subdivide data groups. This is a method where the functions divide the data to be manipulated in smaller groups, which each core/thread gets to handle. This method can reduce the amount of memory fetches each processor has to make, which heightens the performance.

2.6.3 Manipulating device restriction

Sometimes, by restricting the capabilities of the device used, one can enhance its performance. This is because when a CPU can have more than one core and threads active, and as a general rule tries to use all its resources. Even though this may not be the most optimal way of using them. For example, if it has to be executed 3 times, dividing the loops instructions in to 16 threads may not be as efficient as dividing into three. This optimizes for data locality and reuse, which usually is one of the biggest consumers of time when executing a larger program.

3
Method

This chapter is split in different parts to explain the methodology of which has been used to accomplish the goals of this thesis. These parts are split up into:

- Intel DevCloud and how to use it
- The implementation of the GEMM benchmark
- The implementation of the BUDE benchmark

Each part will explain how it has been implemented so that the results can be replicated in the easiest way possible

3.1 Intel DevCloud development methodology

In this chapter, an explanation on how to use Intel DevCloud will be made.

3.1.1 How to get started

To get started with development on Intel's DevCloud an account has to be made, this is easiest accomplished by getting on to the website (`https://devcloud.intel.com/oneapi/get_started/`) and following the instructions when you have clicked sign in. It is open source, so all you have to do is provide an email-address and the purpose of your account, and then it is ready to go. You only have four months of enrolment, though, so make sure to request for an extension before that time runs out if you need it.

For making and executing project one needs to connect to the servers, this can be accomplished in many different ways. For this an SSH, or Secure Shell, client

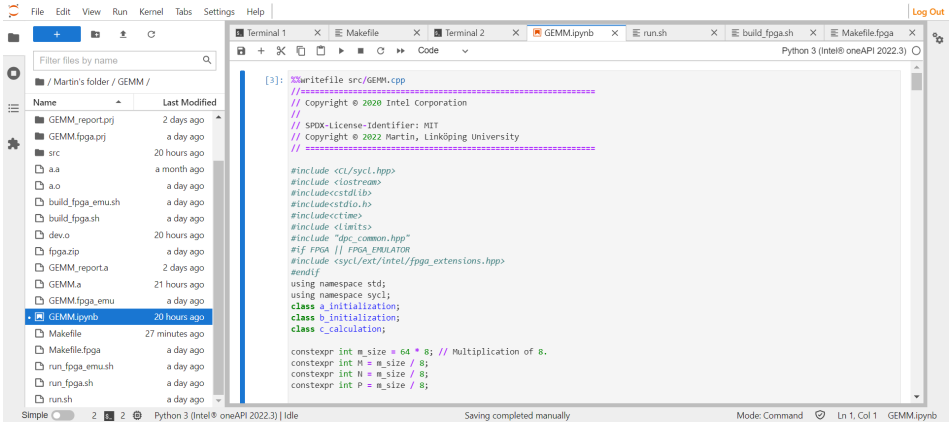

Figure 3.1: *Screenshot from Jupyterlab**

must be used. For Linux and macOS users, one can download an automated installer. This installer will set up SSH configurations and a private SSH key. The installer is found here: `https://devcloud.intel.com/oneapi/documentation/connect-with-ssh-linux-macos/`. For Windows, the user first has to download Cygwin. This can be done through the official web page by following the instruction on: `https://devcloud.intel.com/oneapi/documentation/connect-with-ssh-windows-cygwin/`. When Cygwin is downloaded, there are several ways to go. One can create an SSH connection with Cygwin with the instructions from the installation page above. One can also connect via Windows own IDE Visual studios CODE, or vsCODE for short. This connection requires vsCODE obviously and can be setup by following the instructions on this website: `https://devcloud.intel.com/oneapi/documentation/connect-with-vscode/`. The last way to connect oneAPI environment is with Jupyterlab. This way is easily accessed via Intel's web page `https://devcloud.intel.com/oneapi/get_started/`. At the bottom of the page, there is a link named "connect to Jupyterlab". Click this link while logged in and one is connected within a few seconds. Once in, we can run everything from this web page. The only drawback here is that Jupyterlab requires one to reconnect to the server every four hours, which means saving files continuously is a must to not lose progress. For convenience's sake, this project was done in the Jupyterlab* environment.

3.1.2 How to build a project

To build, compile and run programs in Jupyterlab you need a few things. Firstly, you need a CPP file which contains the code to be run. Secondly, a Makefile is needed. A Makefile contains all the commands to build and execute the code.

Listing 3.1: *Makefile example*

```
DPCPP_CXX = dpcpp
DPCPP_CXXFLAGS = -std=c++17
```

```
DPCPP_LDSFLAGS =
DPCPP_EXE_NAME = GEMM
DPCPP_SOURCES = src/GEMM.cpp
all:
$(DPCPP_CXX) $(DPCPP_CXXFLAGS) -o $(DPCPP_EXE_NAME) $(DPCPP_SOURCES)

run:
    ./$(DPCPP_EXE_NAME)

clean:
    rm -rf  *.out
```

In Listning 3.1 we can see how a normal Makefile usually looks like. This example help one compile any DPCPP code to CPUs and GPUs. To do this a .sh or shell file, is needed. In the .sh file the commands that will be executed on the computing node is defined. Listing 3.2 shows how a shell file can look.

Listing 3.2: *Shell file example*

```
#!/bin/bash
source /opt/intel/oneapi/setvars.sh
make clean
make all
make run
```

In the example, we can see that first some environment variables are set before removing old files with the clean command. Later we compile the code for the relevant platform and then execute it. To submit a job to a computing node on Intel DevCloud one opens a terminal by click the blue plus symbol. In the launcher menu that pops up, chose "terminal", then a Linux terminal will pop up where the usual Linux terminal commands work. Jump to the working directory with the "cd" command. When in the working directory one can submit a job to compute nodes with the "qsub" command. The following Listing 3.3 shows how this can look.

Listing 3.3: *Job submitting example*

```
qsub -l nodes=1:gpu:ppn=2 -d . run.sh
```

When submitting the job one can specify which kind of compute node should do the work. In Listing 3.3 GPU is specified, this works the same for CPU. When compiling and running FPGA some things have to be added.

3.1.3 How to compile and run FPGA jobs

To run your code on FPGA some more things have to be done. First one needs a separate Makefile, Makefile.fpga. Listing 3.4 shows how one of these file may look like.

Listing 3.4: *Makefile.fpga example*

```
CXX := dpcpp
CXXFLAGS = -O2 -g -std=c++17

SRC := src/GEMM.cpp
```

```
.PHONY: fpga_emu run_emu clean

fpga_emu: GEMM.fpga_emu

hw: GEMM.fpga

report: GEMM.a

GEMM.fpga_emu: $(SRC)
$(CXX) $(CXXFLAGS) -fintelfpga $^ -o $@ -DFPGA_EMULATOR=1

a.o: $(SRC)
$(CXX) $(CXXFLAGS) -fintelfpga -c $^ -o $@ -DFPGA=1

GEMM.fpga: a.o
$(CXX) $(CXXFLAGS) -fintelfpga $^ -o $@ -Xshardware
-Xsboard=/opt/intel/oneapi/intel_a10gx_pac:pac_a10

run_emu: GEMM.fpga_emu
./GEMM.fpga_emu

run_hw: GEMM.fpga
./GEMM.fpga

dev.o: $(SRC)
$(CXX) $(CXXFLAGS) -fintelfpga -c $^ -o $@ -DFPGA=1

GEMM.a: dev.o
$(CXX) $(CXXFLAGS) -fintelfpga -fsycl-link $^ -o $@ -Xshardware
-Xsdont-error-if-large-area-est

clean:
rm -rf *.o *.d *.out *.mon *.emu *.aocr *.aoco
*.prj *.fpga_emu *.fpga_emu_buffers GEMM.fpga *.a
```

When compiling for FPGA, there are three different steps of compilation. The first is the FPGA emulations, which is a simulation done on a CPU, to check that the code has the right function. This can be done on any node and takes just a few minutes. The shell file can look something like in the Listing 3.5.

Listing 3.5: *.sh file FPGA exampel*

```
#!/bin/bash
source /opt/intel/oneapi/setvars.sh
make fpga_emu -f Makefile.fpga
```

The second step is the report. This compilation makes a report, which contains information on area estimations, clock frequency and more seen in Figure 3.2. Here one can see if any optimization has to be done to fit everything on the board. This report generation can take between a few minutes to a few hours depending on how big the design is. Lastly, there is the actual bit stream compilation for the FPGA board. This compilation takes a few hours and has to be done on a FPGA compile capable node. One submits a job to such a node with a command like Listing 3.6. After the compilation is done, one can run the program. Once again we need to specify a FPGA capable node but this time for running. So instead of fpga_compile, one uses fpga_runtime.

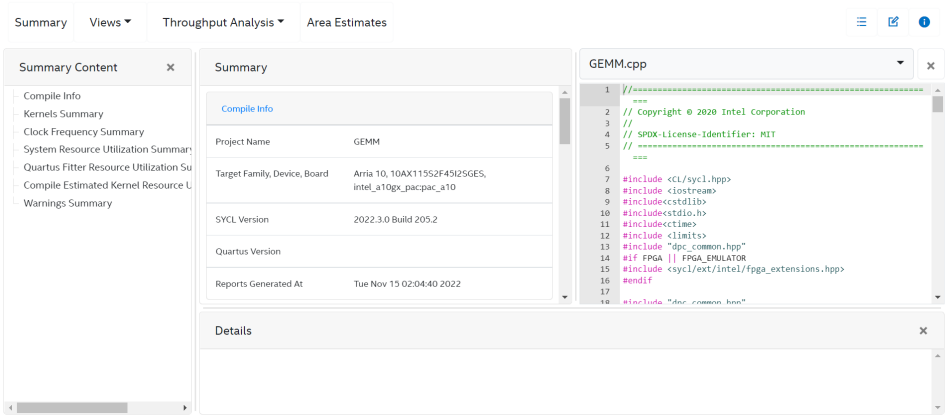

Figure 3.2: *Screenshot from the early image report for the FPGA*

Listing 3.6: *Job submitting example*

```
qsub -l nodes=1:fpga_compile:ppn=2 -d . build_fpga.sh
```

3.1.4 Analysing outputs

To analyse the output of the jobs, one can look in the *.o and *.e files created after the job is run. In the .o file one can find a copy of the output stream, prints from the host etc. The .e file contains the error messages that the job gets. This is very useful when debugging. This can also be done through the terminal by using the -I flag queuing a job.

3.2 GEMM

To implement the GEMM Benchmark the oneAPI samples were used as an outline which we then modified to fit our needs. The source code can be found here: https://github.com/oneapi-src/oneAPI-samples/blob/master/DirectProgr C%2B%2BSYCL/DenseLinearAlgebra/matrix_mul/src/matrix_mul_sycl.cpp. The parts that were added to the original code were a runtime if (if) to determine if the target hardware was FPGA or not. This was because we needed to add a few header files and switch to an FPGA selector, seen in code Listing 3.7.

Listing 3.7: *FPGA selector snippet*

```
#if FPGA_EMULATOR
 // FPGA emu selector
  ext::intel::fpga_emulator_selector DeviceSelector;
#elif FPGA
 // FPGA hardware selector
  ext::intel::fpga_selector DeviceSelector;
#else
  // selecting default the device
```

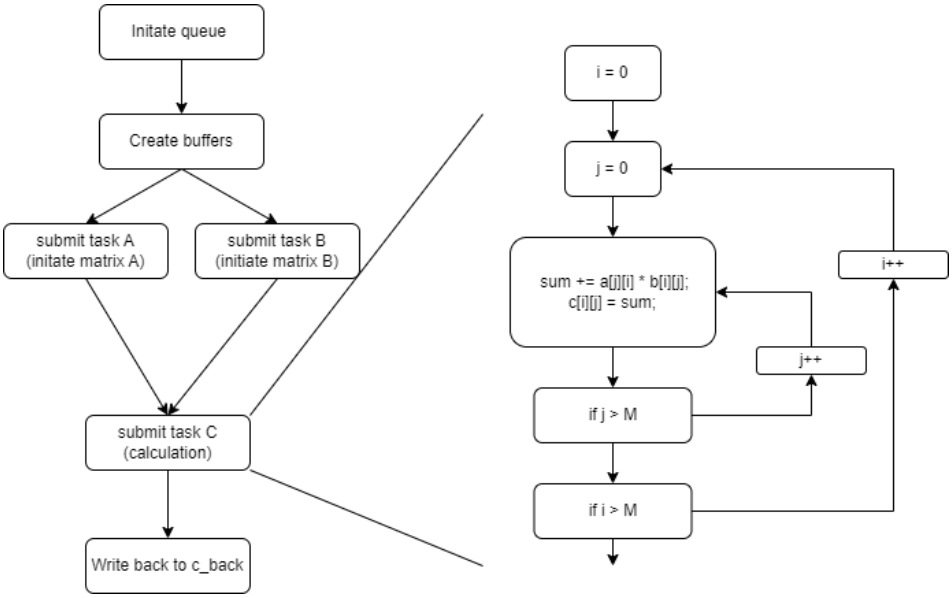

Figure 3.3: *Flowchart of the GEMM benchmark*

```
  default_selector DeviceSelector;
#endif
```

To the source code was also a single task kernel added, a copy of the original kernel. This kernel would be the one the FPGA would execute and the other types of hardware would execute the **parallel_for** loop. A FPGA is not capable of executing the **parallel_for** loop and instead gets to execute the single task loop. Both loops work the same way only that the single task loop will execute sequentially with a nested loop while the other will do it in parallel, hence its name.

In the single task kernel is also where you find the unrolls to enhance its performance via the ***#pragma unroll*** command. The ***#pragma unroll*** command unrolls the loop the amount of times written. In Listing 3.8 it is shown how to unroll a loop three times. This is how the command will be used to find the best way to unroll the single task loop for the FPGA hardware.

Listing 3.8: *Unroll example*

```
#pragma unroll 3
for( int i = 0, i < 9, i++){
    sum += i
}
```

A flowchart for the code can be found in Figure 3.3. The code works by first by initiating things like the result matrix *c_back* and the queue *q*. After this buffers for all the matrices are made. Then depending on which hardware is chosen the code either goes in to the single task or **parallel_for** kernel, then by submitting

all the tasks to be executed to the queue. The initiating tasks (A and B) will and can be executed in parallel whereas their accessors have no concurrency conflicts. But task C, the calculating task, will not be executed until both task the other tasks have been performed. Task C then calculates the matrix multiplication and puts the answer in the *c_back* matrix. Every task is individually clocked to then be added together to a total time. After this a result versifier verify the results and then end the program.

3.3 BUDE

With the BUDE benchmark the code was created with the help of the miniBUDE repository made by University of Bristol: `https://github.com/UoB-HPC/miniBUDE` and the help of previous works on the same topic by Wei-Chih Huang at University of Bristol[13]. To first set up the code some data files had to be downloaded from the previously mentioned github. These are .in files called ligands.in, protein.in, forcefield.in and pose.in who are the input data to try this benchmark on. The other file to download is the ref_energies.out file which is the correct out data that the code will compare to at the end.

The code itself works a lot like the previous benchmark. Though this benchmark has a lot more to initialize before it starts the function **loadParameters()** does this. When the kernel is run the code makes the same separation depending on the hardware, single task kernel for FPGA, **parallel_for** for everything else. The difference between them is that the single task have to do each work group separately while the parallel does each work group in parallel.

The computation in itself is to calculate the total energies from the interactions of each ligand (*l_atom*) with each protein (*p_atom*) by their placement in space to each other and the strength of their force field. After iterating through each atom, the energies are added to a matrix and returned.

The BUDE benchmark has a few speed-up methods. First there was the unrolling of the single task loop which would work the same as in the GEMM benchmark. Second is the manipulations of work group sizes. To manipulate these the constants **workgroupsizeDEFAULT** was varied to look for differences in performance. The last method was the manipulation of how many threads the CPU uses when executing the benchmark which was set with the command DPCPP_CPU_NUM_CUS=xx in the terminal.

4

Results

This chapter is where the results of this thesis are presented and discussed. Some things did not go as planned. This is because of some technical errors on Intel DevCloud's part, the compilation to the FPGA board could not be done. With the time constraint put on this project the results were finished with just the CPUs and GPUs results for timed events, and the early reports for the FPGA board will give some insight on how the FPGA would perform.

4.1 GEMM

First of we have the results from the GEMM benchmark. In Figure 4.1 it is shown how the GPU and the CPU performed on the GEMM benchmark depending on the size of the matrix. The x-axis shows the M value of the $M \times M$ matrix used in the tests. All tests were made on homogeneous matrices between the sizes of 2^6 to 2^{12}. As seen the very small matrices gives a higher execution time for the GPU to then go down and slowly rise while the CPUs execution time holds steady. This behaviour is probably seen because the complex structure of the CPU lets it use its complexity very efficiently to almost remove any added computation time, but its complexity also impedes its bottom line (the fastest it could go, even for low sizes). While the GPU is faster at low sizes (except for the smallest at 64) one can observe an exponential growth depending on the size of the matrix. This would seem very inline with the amount of operations needed to be done also exponentially grows with each increase of n^2, where n is the increase of M. Some estimation can be made from the early report for the FPGA. Counting the clock cycles and adding them together and dividing it by the clock rate we get:

$$(2616 + 204 + 192)/240000 = 0,01255s \tag{4.1}$$

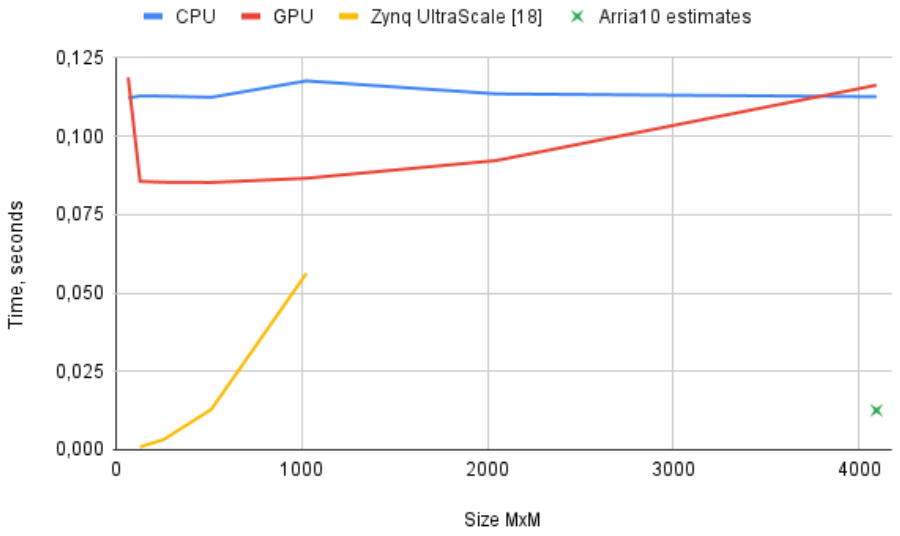

Figure 4.1: *Kernel execution time depending on size of the matrix*

Hardware	Clock frequency (MHz)	Cores
Intel Xeon gold 6128	3400	6
Intel UHD Graphics P630	350	24

Table 4.1: *Specifications for the different hardware*

This shows a significant performance increase against both the CPU and GPU. We can also compare all of these results against the results of previous studies on the Zynq device[14]. Here one can see that it performs better than both CPU and GPU for the smaller matrices, but it has a lot larger increase in execution time the bigger the matrix is, so it will probably surpass the CPU and GPU at larger data amounts.

One would usually think that the GPU would be faster than the CPU, but when comparing them the most apparent difference is that the clock frequency is about 10 times higher for the CPU, as seen in Table 4.1. Even though the GPU has more cores, the difference in clock frequency would still make a bigger impact on performance. Also comparing the Intel GPU UHD P630 GPU to different GPUs than the one used on Intel's DevCloud, it performs very poorly compared to the industry standard right now and their benchmark test [8]. This would explain why the results would skew in the favour of the CPU which seems to compare better to its industry counterpart[7].

One thing that was also tested was the amount of difference when restricting

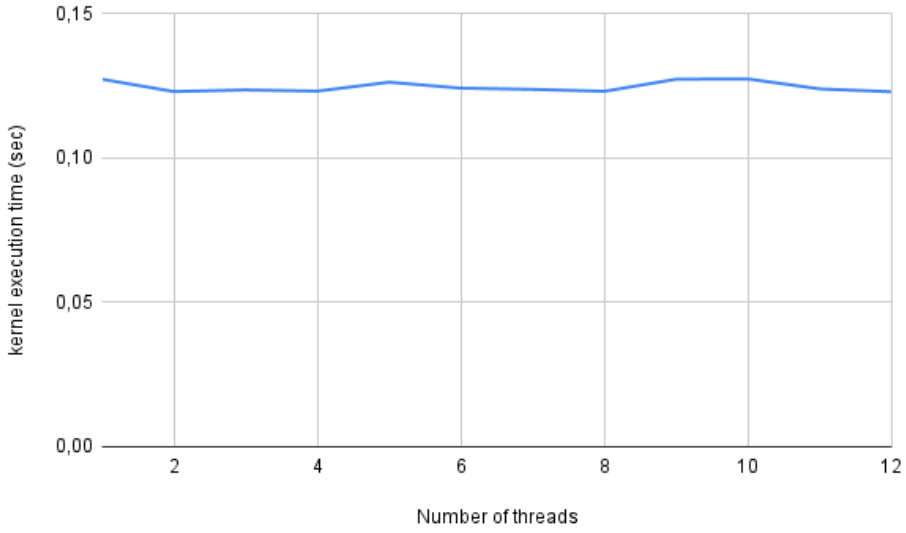

Figure 4.2: *Kernel execution time depending on amount of threads*

the amount of threads the CPU could use, these results can be seen in Figure 4.2. As seen in the Figure, this seems to have no noticeable impact on the overall performance of the CPU.

The last thing to be looked at for the GEMM benchmark is the early report estimations of the Aria10 FPGA board. The resource utilization estimates are the best data to compare to other tests and how much speed-up the unrolls could have given. In Table 4.2, the resource utilization for the different kernel designs for the FPGA can be seen, one not unrolled and one unrolled every loop 10 times. The 10 times unroll was chosen because it had the highest utilization of RAM one could get with a homogeneous unroll (all unrolls are equal in size). The not unrolled results compare pretty well to similar studies on the topic [13], while for the unrolled, one can see a much higher resource utilization in especially RAM usage. Higher utilization of resources for an FPGA like this usually corresponds with a better performance, so we could see that to unroll would probably speed up the benchmark test on the FPGA. However, utilization is still quite low on all things except for RAM, so the Arria10 Board is maybe not the most cost-efficient choice of board for this kind of task, where a less capable board in terms of resources might have the same performance as long as it has a big RAM.

4.2 BUDE

From the results of the BUDE benchmark, the graphs in Figure 4.3 were made. Here one can see that the CPU outperforms the GPU by about 12 times. The ex-

Table 4.2: *Resource utilization, early image FPGA report for GEMM*

Kernel design	ALUTs	FFs	RAMs	MLABs	DSPs
Not unrolled	<1%	22%	22%	1%	10%
Unrolled 10	<1%	44%	96%	4%	35%

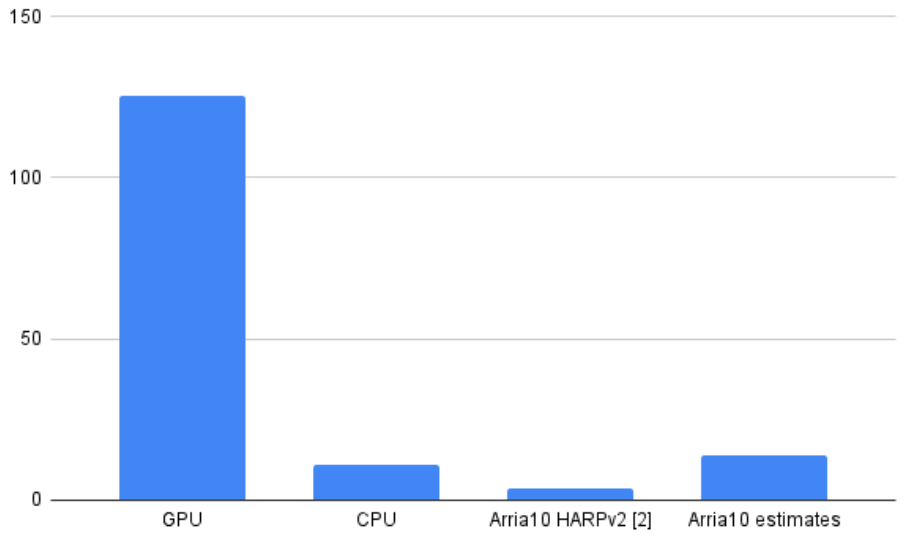

Figure 4.3: *Kernel execution time (ms) for different hardware*

planation could be that the BUDE benchmark is built with four loops that will be executed at the same time, which means that the GPU can not utilize its larger capacity for parallelization. That would give the clock frequency difference between the two hardware, shown in table 4.1, an even bigger impact on performance. In Figure 4.3, one can also see the result from another study done on the Arria10 FPGA board[1]. Comparing the result for the FPGA, the Arria10 seem to outperform even the CPU. This would point to that if the FPGA on DevCloud could have been used, it could have outperformed the CPU too. Estimating the time for the FPGA implementation in the same way for BUDE as for the GEMM implementation gives us:

$$3371/240000 = 0,01404583333s \tag{4.2}$$

Which is similar to the CPU performances. This could point to that with some more specific adjustments could have a similar performance as Arria10 HARPv2.

A test of the impact of enabled threads was also done on the CPUs performance. Figure 4.4 shows how the impact of using different amounts of threads can have. The graphs show that a low amount of threads will impact the performance badly,

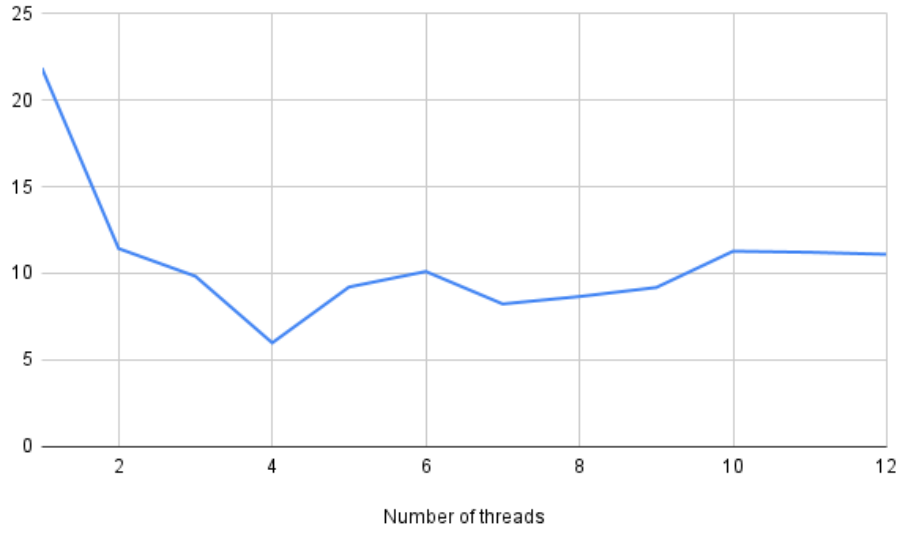

Figure 4.4: *Kernel execution time depending on amount of threads enabled.*

Table 4.3: *Resource utilization, early image FPGA report for BUDE*

Kernel design	ALUTs	FFs	RAMs	MLABs	DSPs
Not unrolled	<1%	25%	30%	1%	13%
Unrolled 4	<1%	43%	79%	9%	68%

but the benchmark hits its peak at four threads and then goes down after and stabilize. This could be explained by that fact that with four threads, where each thread handles one loop each. Above this the threads get cluttered when different threads have two shared loops. This would also explain why the performance is worse, even though they have a bigger resource budget.

In Table 4.3, the resource utilization of the BUDE kernel is shown. One for a not unrolled version and one for all loops unrolled four times, four chosen by the fact that more than that would not give any more performance boost. When one compare the two it is easy to see that the unrolled version have a better resource utilization. This would point to better performance as said before. One can also see that in general, the BUDE benchmark utilizes the board better than the GEMM where all resources seem to go up when unrolling. The explanation for this could be that the benchmark is more computationally heavy and fewer data heavy, so a more complex board like the Arria10 would be needed to implement this benchmark.

5

Conclusion

This thesis aimed at looking at how viable to use a high performance computing platform to implement different benchmarks on different hardware. Intel's DevCloud oneAPI was chosen for this thesis. Here, by using the SYCL based C++ extension DPC++ to implement two different HPC benchmarks, GEMM and BUDE. After implementation the two benchmarks were supposed to be run on three different platforms, a CPU (Intel Xeon Gold 6128), a GPU (Intel UHD Graphics p630) and a FPGA (Intel Arria10 board). With some technical difficulties and time constraints the FPGA results could not be followed through on, because the bit-stream generation was not working, so the results were incomplete. Therefore, the results were estimated instead. The result yielded an overall better performance for the CPU for both benchmarks, this probably because the CPU used in the test is closer to the industry standard than the GPU and other result could have yielded if run on a more powerful GPU. By comparing the result of other studies we could see that the FPGA board could probably be comparable or even be better than the CPU for the benchmarks.

After this some methods to improve on the benchmarks were used to see if performance could be improved upon. One method used was loop unrolling which was used for the FPGA part of the code. When using this speed-up method no direct improvement could be observed because the code could not be run. However, the FPGA board had a higher resource utilization using this method, which would point to a higher performance. Another method was to restrict the amount of threads that the CPU could use during execution. This saw practically no improvement for GEMM but could improve the performance of the BUDE benchmark when only using four threads.

In conclusion, the platform worked well for implementing HPC benchmarks for

both CPU and GPU platforms and with more time and less human and technical errors could even be used for FPGA HPC development and research. The benchmark was also improvable with some HPC speed-up methods.

5.1 Future work

For some future works on this topic would be for a start to get the FPGA compilation to work, to see how it actually performs compared to the other hardware, both using speed-up methods and not. It would also be interesting to test more specialized speed-ups where for example all unrolls are not homogeneous or see if changing the work group size in the BUDE kernel makes any difference. Also, testing other benchmarks could be interesting.

Appendix

A

Code for GEMM benchmark

```
//==============================================================
// Copyright    2020 Intel Corporation
//
// SPDX-License-Identifier: MIT
// Copyright    2022 Martin, Link ping University
// ==============================================================

#include <CL/sycl.hpp>
#include <iostream>
#include<cstdlib>
#include<stdio.h>
#include<ctime>
#include <limits>
#include "dpc_common.hpp"
#if FPGA || FPGA_EMULATOR
#include <sycl/ext/intel/fpga_extensions.hpp>
#endif
using namespace std;
using namespace sycl;
class a_initialization;
class b_initialization;
class c_calculation;

constexpr int m_size = 4096; // Multiplication of 8.
constexpr int M = m_size;
constexpr int N = m_size;
constexpr int P = m_size;

//verify the results
int VerifyResult(float (*c_back)[P]);

int main() {
    // Host memory buffer that device will write data back before destruction.
    float(*c_back)[P] = new float[M][P];

    // intialize c_back
    for (int i = 0; i < M; i++)
        for (int j = 0; j < P; j++) c_back[i][j] = 0;

    #if FPGA_EMULATOR
     // FPGA emu selector
     ext::intel::fpga_emulator_selector DeviceSelector;
    #elif FPGA
     // FPGA hardware selector
     ext::intel::fpga_selector DeviceSelector;
    #else
     // selecting default the device
     default_selector DeviceSelector;
    #endif

    queue q(DeviceSelector, dpc_common::exception_handler);
    cout << "Device:␣" << q.get_device().get_info<info::device::name>() << "\n";

    // Host memory buffer that device will write data back before destruction.
```

```
// device write data back to buffer of host memory before be destroyed

//auto data = malloc_host<float>(M*P, q);
//auto c = malloc_device<float>(M*N, q);
//auto b = malloc_device<float>(N*P, q);

 // buffers for matrices, buffer c is bound with host memory data
 buffer<float,2> a_buf(range(M, N));
 buffer<float,2> b_buf(range(N, P));
 buffer c_buf(reinterpret_cast<float *>(c_back),range(M, P));

 cout << "matrices size: c(" << M << "," << P << ") = a(" << M << "," << N << ") * b(" << N << "," << P << ")\n";
 #if FPGA || FPGA_EMULATOR
     dpc_common::TimeInterval kernel_execution_a_runtime;

     auto execution_a = q.submit([&](handler& h)
     {
         accessor a(a_buf, h, write_only);
         h.single_task<a_initialization>([=]() [[intel::kernel_args_restrict]] {
             // every element of A matrix is set to 1.
             for (int iway = 0; iway < M; iway++) {
                #pragma unroll 10
                for (int jway = 0; jway < N; jway++)
                {
                    a[iway][jway] = 1;
                }
             }
         });
     });
     double elapsed_execution_a_time = kernel_execution_a_runtime.Elapsed();
     dpc_common::TimeInterval kernel_execution_b_runtime;
     // Initializing B matrix by submitting command group to queue
     auto execution_b = q.submit([&](handler& h)
     {
         // access (write) to buffer on the device
         accessor b(b_buf, h, write_only);
         h.single_task<b_initialization>([=]() [[intel::kernel_args_restrict]] {
             // Every column of B (sequence 1,2,...,N)
             for (int iway = 0; iway < N; iway++) {
                 #pragma unroll 10
                 for (int jway = 0; jway < P; jway++)
                 {
                     b[iway][jway] = iway + 1;
                 }
             }
         });
     });
     double elapsed_execution_b_time = kernel_execution_b_runtime.Elapsed();
     dpc_common::TimeInterval kernel_execution_c_runtime;
     // Multiplying matrices: c = a * b by submitting command group to queue

     auto execution_c = q.submit([&](handler& h)
     {
         accessor a(a_buf, h, read_only);
         accessor b(b_buf, h, read_only);
         accessor c(c_buf, h, write_only);
         h.single_task<c_calculation>([=]() [[intel::kernel_args_restrict]] {
             for (int iway = 0; iway < M; iway++) {
                 #pragma unroll 10
                 for (int jway = 0; jway < P; jway++)
                 {
                     // Getting y direction global position.
                     int row = jway;
                     // Getting x direction global position.
                     int col = iway;
                     int width_a = M;
                     float sum = 0;
                     // Calculating result of an element of c
                     #pragma unroll 10
                     for (int iway = 0; iway < width_a; iway++)
                     {
                         sum += a[row][iway] * b[iway][col];
                     }
                     c[iway][jway] = sum;
                 }
             }
         });
     });
 double elapsed_execution_c_time = kernel_execution_c_runtime.Elapsed();
 #else
     dpc_common::TimeInterval kernel_execution_a_runtime;

     auto execution_a = q.submit([&](handler& h)
     {
         accessor a(a_buf, h, write_only);
         h.parallel_for(range(M, N), [=](auto index) {
             // every element of A matrix is set to 1.
             a[index] = 1.0f;
         });
     });
     double elapsed_execution_a_time = kernel_execution_a_runtime.Elapsed();
```

```
        dpc_common::TimeInterval kernel_execution_b_runtime;
        // Initializing B matrix by submitting command group to queue
        auto execution_b = q.submit([&](handler& h)
        {
            // access (write) to buffer on the device
            accessor b(b_buf, h, write_only);
            h.parallel_for(range(N, P), [=](auto index) {
                // Every column of B (sequence 1,2,...,N)
                b[index] = index[0] + 1;
            });
        });
        double elapsed_execution_b_time = kernel_execution_b_runtime.Elapsed();
        dpc_common::TimeInterval kernel_execution_c_runtime;
        // Multiplying matrices: c = a * b by submitting command group to queue

        auto execution_c = q.submit([&](handler& h)
        {
            accessor a(a_buf, h, read_only);
            accessor b(b_buf, h, read_only);
            accessor c(c_buf, h, write_only);
            h.parallel_for(range(M, P), [=](auto index) {
                // Getting y direction global position.
                int row = index[0];
                // Getting x direction global position.
                int col = index[1];
                int width_a = M;
                float sum = 0;
                // Calculating result of an element of c
                for (int iway = 0; iway < width_a; iway++)
                {
                    sum += a[row][iway] * b[iway][col];
                }
                c[index] = sum;
            });
        });
    double elapsed_execution_c_time = kernel_execution_c_runtime.Elapsed();
    #endif

    double input_size_kb = (2 * N) * sizeof(float) / (1024);

    host_accessor A{c_buf};

    std::cout << "Initializing␣kernel␣A␣throughput:␣"
    << (input_size_kb / elapsed_execution_a_time) << "␣KB/s␣\n";
    std::cout << "Initializing␣kernel␣B␣throughput:␣"
    << (input_size_kb / elapsed_execution_b_time) << "␣KB/s␣\n";
    std::cout << "Initializing␣kernel␣C␣throughput:␣"
    << (input_size_kb / elapsed_execution_c_time) << "␣KB/s␣\n";
    cout << "Total␣execution␣time:" << elapsed_execution_a_time + elapsed_execution_b_time + elapsed_execution_c_time << "sec\n";

    int result;
    cout << "Result␣of␣GEMM␣using␣DPC++:␣";
    result = VerifyResult(c_back);
    //delete[] c_back;
    return result;
    //return 1;
}

bool ValueSame(float a, float b)
{
    return fabs(a - b) < numeric_limits<float>::epsilon();
}

int VerifyResult(float (*c_back)[P]) {
  // Check that the results are correct by comparing with host computing.
  int i, j, k;

  // 2D arrays on host side.
  float(*a_host)[N] = new float[M][N];
  float(*b_host)[P] = new float[N][P];
  float(*c_host)[P] = new float[M][P];

  // Each element of matrix a is 1.
  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++) a_host[i][j] = 1.0f;

  // Each column of b_host is the sequence 1,2,...,N
  for (i = 0; i < N; i++)
    for (j = 0; j < P; j++) b_host[i][j] = i + 1.0f;

  // c_host is initialized to zero.
  for (i = 0; i < M; i++)
    for (j = 0; j < P; j++) c_host[i][j] = 0.0f;

  for (i = 0; i < M; i++) {
    for (k = 0; k < N; k++) {
      // Each element of the product is just the sum 1+2+...+n
      for (j = 0; j < P; j++) {
        c_host[i][j] += a_host[i][k] * b_host[k][j];
      }
    }
```

```
  }

  bool mismatch_found = false;

  // Compare host side results with the result buffer from device side: print
  // mismatched data 5 times only.
  int print_count = 0;

  for (i = 0; i < M; i++) {
    for (j = 0; j < P; j++) {
      if (!ValueSame(c_back[i][j], c_host[i][j])) {
        cout << "Fail - The result is incorrect for element: [" << i << ", "
             << j << "], expected: " << c_host[i][j]
             << ", but found: " << c_back[i][j] << "\n";
        mismatch_found = true;
        print_count++;
        if (print_count == 5) break;
      }
    }

    if (print_count == 5) break;
  }

  delete[] a_host;
  delete[] b_host;
  delete[] c_host;

  if (!mismatch_found) {
    cout << "Success - The results are correct!\n";
    return 0;
  } else {
    cout << "Fail - The results mismatch!\n";
    return -1;
  }
}
```

B

Code for BUDE benchmark

```
//Copyright 2020 University of Bristol High Performance Computing Group
//Copyright 2022 University of Bristol Wei-Chih Huang_Advanced Microelectronic Systems Engineering_MSc
//Copyright 2022 Link ping University Martin Frick Lundgren Advanced Electric Engineering_MSc
#include <cmath>
#include <memory>
#include <vector>
#include <chrono>
#include <iostream>
#include <fstream>
#include <functional>
#include <algorithm>
#include <CL/sycl.hpp>
#include <cstdint>
#include <string>
#include <iomanip>
#include "CL/opencl.h"
#include "dpc_common.hpp"
#if FPGA || FPGA_EMULATOR
#include <sycl/ext/intel/fpga_extensions.hpp>
#endif
typedef std::chrono::high_resolution_clock::time_point TimePoint;
#define SettingZERO 0.0f
#define SettingQUARTER 0.25f
#define SettingHALF 0.5f
#define SettingONE 1.0f
#define SettingTWO 2.0f
#define SettingFOUR 4.0f
#define SettingCNSTNT 45.0f
// Energy evaluation parameters
#define SettingHBTYPEF 70
#define SettingHBTYPEE 69
#define SettingHARDNESS 38.0f
#define SettingNPNPDIST 5.5f
#define SettingNPPDIST 1.0f

using namespace std;
using namespace sycl;
using namespace cl;
using namespace __spv;
using namespace access;
using namespace detail;
constexpr cl::sycl::access::mode sycl_read = cl::sycl::access::mode::read;
constexpr cl::sycl::access::mode sycl_write = cl::sycl::access::mode::write;
constexpr cl::sycl::access::mode sycl_read_write = cl::sycl::access::mode::read_write;
constexpr cl::sycl::access::mode sycl_discard_write = cl::sycl::access::mode::discard_write;
constexpr cl::sycl::access::target sycl_lmem = cl::sycl::access::target::local;
constexpr cl::sycl::access::target sycl_gmem = sycl::access::target::device;
#ifndef PPWIDEFAULT
#define PPWIDEFAULT 16
#endif
#ifndef workgroupsizeDEFAULT
#define workgroupsizeDEFAULT 4
#endif
#ifndef NUMTDPERTHREAD
```

```
#define NUMTDPERTHREAD PPWIDEFAULT
#endif
#define ITERSDEFAULT 8
#define NPOSESDEFAULT 65536
#define NPOSESREF 65536
//#define DIRDATA "C:/Users/user/Desktop/BUDE/data/bm1" // for laptop
#define DIRDATA "miniBUDE/data/bm1" //for devcloud jupyterla
#define LIGANDFILE "/ligand.in"
#define PROTEINFILE "/protein.in"
#define Force_FieldFILE "/forcefield.in"
#define POSESFILE "/poses.in"
#define FILEREFENERGIES "/ref_energies.out"
namespace clsycl = cl::sycl;
static constexpr clsycl::access::mode R = clsycl::access::mode::read;
static constexpr clsycl::access::mode W = clsycl::access::mode::write;
static constexpr clsycl::access::mode RW = clsycl::access::mode::read_write;
static constexpr clsycl::access::target GlobalSize = clsycl::access::target::device;
static constexpr clsycl::access::target Local = clsycl::access::target::local;

typedef struct
{
    float x, y, z;
    int32_t type;
} TheAtom;

typedef struct
{
    int32_t hbtype;
    float radius;
    float hphb;
    float elsc;
} TheFFParams;

struct Params {
    size_t natlig;
    size_t natpro;
    size_t ntypes;
    size_t nposes;
    std::vector<TheAtom> protein;
    std::vector<TheAtom> ligand;
    std::vector<TheFFParams> Force_Field;
    std::array<std::vector<float>, 6> poses;
    size_t iterations;
    size_t workgroupsize;
    std::string deckDir;
     friend std::ostream& operator<<(std::ostream& os, const Params& params)
    {
        os <<
        "natlig: " << params.natlig << "\n" <<
        "natpro: " << params.natpro << "\n" <<
        "ntypes: " << params.ntypes << "\n" <<
        "nposes: " << params.nposes << "\n" <<
        "iterations: " << params.iterations << "\n" <<
        "posesPerWI: " << NUMTDPERTHREAD << "\n" <<
        "workgroupsize: " << params.workgroupsize << "\n";
        return os;
     }
};

void FastenMain(
    handler& h,
    size_t workgroupsize,
    size_t ntypes,
    size_t nposes,
    size_t natlig,
    size_t natpro,
    accessor<TheAtom, 1, sycl_read, sycl_gmem> protein_molecule,
    accessor<TheAtom, 1, sycl_read, sycl_gmem> ligand_molecule,
    accessor<float, 1, sycl_read, sycl_gmem> Trans_FormsZreo,
    accessor<float, 1, sycl_read, sycl_gmem> Trans_FormsOne,
    accessor<float, 1, sycl_read, sycl_gmem> Trans_FormsTwo,
    accessor<float, 1, sycl_read, sycl_gmem> Trans_FormsThree,
    accessor<float, 1, sycl_read, sycl_gmem> Trans_FormsFour,
    accessor<float, 1, sycl_read, sycl_gmem> Trans_FormsFive,
    accessor<TheFFParams, 1, sycl_read, sycl_gmem> Force_Field,
    accessor<float, 1, sycl_discard_write, sycl_gmem> Energytotals)
    {
        constexpr const auto FloatMax = std::numeric_limits<float>::max();
        size_t GlobalSize = ceil((nposes) / static_cast<double> (NUMTDPERTHREAD));
        GlobalSize = workgroupsize * ceil(static_cast<double> (GlobalSize) / workgroupsize);
        accessor<TheFFParams, 1, sycl_read_write, sycl_lmem> local_Force_Field(range<1>(ntypes), h);
        #if FPGA || FPGA_EMU
        h.single_task<class bude_kernel>([=]() [[intel::kernel_args_restrict]]
        {
            #pragma unroll 4
             for (size_t id = 0; id < workgroupsizeDEFAULT; id++) {
             const size_t lid = id;
             const size_t gid = id;
             const size_t lrange = 1;
             float etot[NUMTDPERTHREAD];
             cl::sycl::float3 lpos[NUMTDPERTHREAD];
```

```
    cl::sycl::float4 Trans_Form[NUMTDPERTHREAD][3];
    size_t ix = gid * lrange * NUMTDPERTHREAD + lid;
    ix = ix < nposes ? ix : nposes - NUMTDPERTHREAD;
   #pragma unroll 4
    for (int iway = lid; iway < ntypes; iway += lrange) local_Force_Field[iway] =
   Force_Field[iway];
   #pragma unroll 4
    for (size_t iway = 0; iway < NUMTDPERTHREAD; iway++) {
    size_t index = ix + iway * lrange;
    const float sx = cl::sycl::sin(Trans_FormsZreo[index]);
    const float cx = cl::sycl::cos(Trans_FormsZreo[index]);
    const float sy = cl::sycl::sin(Trans_FormsOne[index]);
    const float cy = cl::sycl::cos(Trans_FormsOne[index]);
    const float sz = cl::sycl::sin(Trans_FormsTwo[index]);
    const float cz = cl::sycl::cos(Trans_FormsTwo[index]);
    Trans_Form[iway][0].x() = cy * cz;
    Trans_Form[iway][0].y() = sx * sy * cz - cx * sz;
    Trans_Form[iway][0].z() = cx * sy * cz + sx * sz;
    Trans_Form[iway][0].w() = Trans_FormsThree[index];
    Trans_Form[iway][1].x() = cy * sz;
    Trans_Form[iway][1].y() = sx * sy * sz + cx * cz;
    Trans_Form[iway][1].z() = cx * sy * sz - sx * cz;
    Trans_Form[iway][1].w() = Trans_FormsFour[index];
    Trans_Form[iway][2].x() = -sy;
    Trans_Form[iway][2].y() = sx * cy;
    Trans_Form[iway][2].z() = cx * cy;
    Trans_Form[iway][2].w() = Trans_FormsFive[index];
    etot[iway] = SettingZERO;
}
size_t il = 0;
do {
    const TheAtom l_TheAtom = ligand_molecule[il];
    const TheFFParams l_params = local_Force_Field[l_TheAtom.type];
    const bool lhphb_ltz = l_params.hphb < SettingZERO;
    const bool lhphb_gtz = l_params.hphb > SettingZERO;
    const cl::sycl::float4 linitpos(l_TheAtom.x, l_TheAtom.y, l_TheAtom.z, SettingONE);
   #pragma unroll 4
    for (size_t iway = 0; iway < NUMTDPERTHREAD; iway++) {
    // Trans_Form ligand TheAtom
    lpos[iway].x() = Trans_Form[iway][0].w() +
    linitpos.x() * Trans_Form[iway][0].x() +
    linitpos.y() * Trans_Form[iway][0].y() +
    linitpos.z() * Trans_Form[iway][0].z();
    lpos[iway].y() = Trans_Form[iway][1].w() +
    linitpos.x() * Trans_Form[iway][1].x() +
    linitpos.y() * Trans_Form[iway][1].y() +
    linitpos.z() * Trans_Form[iway][1].z();
    lpos[iway].z() = Trans_Form[iway][2].w() +
    linitpos.x() * Trans_Form[iway][2].x() +
    linitpos.y() * Trans_Form[iway][2].y() +
    linitpos.z() * Trans_Form[iway][2].z();
    }
    size_t ip = 0;
do {
    const TheAtom p_TheAtom = protein_molecule[ip];
    const TheFFParams p_params = local_Force_Field[p_TheAtom.type];
    const float radij = p_params.radius + l_params.radius;
    const float r_radij = 1.f / (radij);
    const float elcdst = (p_params.hbtype == SettingHBTYPEF && l_params.hbtype ==
   SettingHBTYPEF) ? SettingFOUR : SettingTWO;
    const float elcdst1 = (p_params.hbtype == SettingHBTYPEF && l_params.hbtype ==
   SettingHBTYPEF) ? SettingQUARTER : SettingHALF;
    const bool type_E = ((p_params.hbtype == SettingHBTYPEE || l_params.hbtype ==
   SettingHBTYPEE));
    const bool phphb_ltz = p_params.hphb < SettingZERO;
    const bool phphb_gtz = p_params.hphb > SettingZERO;
    const bool phphb_nz = p_params.hphb != SettingZERO;
    const float p_hphb = p_params.hphb * (phphb_ltz && lhphb_gtz ? -SettingONE :
   SettingONE);
    const float l_hphb = l_params.hphb * (phphb_gtz && lhphb_ltz ? -SettingONE :
   SettingONE);
    const float distdslv = (phphb_ltz ? (lhphb_ltz ? SettingNPNPDIST :
   SettingNPPDIST) : (lhphb_ltz ? SettingNPPDIST : -FloatMax));
    const float r_distdslv = 1.f / (distdslv);
    const float chrg_init = l_params.elsc * p_params.elsc;
    const float dslv_init = p_hphb + l_hphb;
   #pragma unroll 2
    for (size_t iway = 0; iway < NUMTDPERTHREAD; iway++) {
    const float x = lpos[iway].x() - p_TheAtom.x;
    const float y = lpos[iway].y() - p_TheAtom.y;
    const float z = lpos[iway].z() - p_TheAtom.z;
    const float distij = cl::sycl::sqrt(x * x + y * y + z * z);
    const float distbb = distij - radij;
    const bool zSettingONE1 = (distbb < SettingZERO);
    etot[iway] += (SettingONE - (distij * r_radij)) * (zSettingONE1 ? 2 *
   SettingHARDNESS : SettingZERO);
    float chrg_e = chrg_init * ((zSettingONE1 ? 1 : (SettingONE - distbb * elcdst1))
   * (distbb < elcdst ? 1 : SettingZERO));
    const float neg_chrg_e = -cl::sycl::fabs(chrg_e);
    chrg_e = type_E ? neg_chrg_e : chrg_e;
    etot[iway] += chrg_e * SettingCNSTNT;
```

```
            const float coeff = (SettingONE - (distbb * r_distdslv));
            float dslv_e = dslv_init * ((distbb < distdslv&& phphb_nz) ? 1 : SettingZERO);
            dslv_e *= (zSettingONE1 ? 1 : coeff);
            etot[iway] += dslv_e;
            }
        } while (++ip < natpro);
    } while (++il < natlig);
     const size_t td_base = gid * lrange * NUMTDPERTHREAD + lid;
     if (td_base < nposes) {
        #pragma unroll 4
         for (size_t iway = 0; iway < NUMTDPERTHREAD; iway++) {
            Energytotals[td_base + iway * lrange] = etot[iway] * SettingHALF;
         }
     }
}
});
    #else
    h.parallel_for<class bude_kernel>(nd_range<1>(workgroupsize, 0), [=](nd_item<1> item) {
            const size_t lid = item.get_local_id(0);
            const size_t gid = item.get_group(0);
            const size_t lrange = item.get_local_range(0);
            float etot[NUMTDPERTHREAD];
             cl::sycl::float3 lpos[NUMTDPERTHREAD];
             cl::sycl::float4 Trans_Form[NUMTDPERTHREAD][3];
             size_t ix = gid * lrange * NUMTDPERTHREAD + lid;
             ix = ix < nposes ? ix : nposes - NUMTDPERTHREAD;
            #pragma unroll 4
             for (int iway = lid; iway < ntypes; iway += lrange) local_Force_Field[iway] =
            Force_Field[iway];
            #pragma unroll 4
             for (size_t iway = 0; iway < NUMTDPERTHREAD; iway++) {
             size_t index = ix + iway * lrange;
             const float sx = cl::sycl::sin(Trans_FormsZreo[index]);
             const float cx = cl::sycl::cos(Trans_FormsZreo[index]);
             const float sy = cl::sycl::sin(Trans_FormsOne[index]);
             const float cy = cl::sycl::cos(Trans_FormsOne[index]);
             const float sz = cl::sycl::sin(Trans_FormsTwo[index]);
             const float cz = cl::sycl::cos(Trans_FormsTwo[index]);
             Trans_Form[iway][0].x() = cy * cz;
             Trans_Form[iway][0].y() = sx * sy * cz - cx * sz;
             Trans_Form[iway][0].z() = cx * sy * cz + sx * sz;
             Trans_Form[iway][0].w() = Trans_FormsThree[index];
             Trans_Form[iway][1].x() = cy * sz;
             Trans_Form[iway][1].y() = sx * sy * sz + cx * cz;
             Trans_Form[iway][1].z() = cx * sy * sz - sx * cz;
             Trans_Form[iway][1].w() = Trans_FormsFour[index];
             Trans_Form[iway][2].x() = -sy;
             Trans_Form[iway][2].y() = sx * cy;
             Trans_Form[iway][2].z() = cx * cy;
             Trans_Form[iway][2].w() = Trans_FormsFive[index];
             etot[iway] = SettingZERO;
        }
        size_t il = 0;
        do {
             const TheAtom l_TheAtom = ligand_molecule[il];
             const TheFFParams l_params = local_Force_Field[l_TheAtom.type];
             const bool lhphb_ltz = l_params.hphb < SettingZERO;
             const bool lhphb_gtz = l_params.hphb > SettingZERO;
             const cl::sycl::float4 linitpos(l_TheAtom.x, l_TheAtom.y, l_TheAtom.z, SettingONE);
            #pragma unroll 4
             for (size_t iway = 0; iway < NUMTDPERTHREAD; iway++) {
             // Trans_Form ligand TheAtom
             lpos[iway].x() = Trans_Form[iway][0].w() +
             linitpos.x() * Trans_Form[iway][0].x() +
             linitpos.y() * Trans_Form[iway][0].y() +
             linitpos.z() * Trans_Form[iway][0].z();
             lpos[iway].y() = Trans_Form[iway][1].w() +
             linitpos.x() * Trans_Form[iway][1].x() +
             linitpos.y() * Trans_Form[iway][1].y() +
             linitpos.z() * Trans_Form[iway][1].z();
             lpos[iway].z() = Trans_Form[iway][2].w() +
             linitpos.x() * Trans_Form[iway][2].x() +
             linitpos.y() * Trans_Form[iway][2].y() +
             linitpos.z() * Trans_Form[iway][2].z();
             }
             size_t ip = 0;
        do {
             const TheAtom p_TheAtom = protein_molecule[ip];
             const TheFFParams p_params = local_Force_Field[p_TheAtom.type];
             const float radij = p_params.radius + l_params.radius;
             const float r_radij = 1.f / (radij);
             const float elcdst = (p_params.hbtype == SettingHBTYPEF && l_params.hbtype ==
            SettingHBTYPEF) ? SettingFOUR : SettingTWO;
             const float elcdst1 = (p_params.hbtype == SettingHBTYPEF && l_params.hbtype ==
            SettingHBTYPEF) ? SettingQUARTER : SettingHALF;
             const bool type_E = ((p_params.hbtype == SettingHBTYPEE || l_params.hbtype ==
            SettingHBTYPEE));
             const bool phphb_ltz = p_params.hphb < SettingZERO;
             const bool phphb_gtz = p_params.hphb > SettingZERO;
             const bool phphb_nz = p_params.hphb != SettingZERO;
             const float p_hphb = p_params.hphb * (phphb_ltz && lhphb_gtz ? -SettingONE :
```

```
            SettingONE);
             const float l_hphb = l_params.hphb * (phphb_gtz && lhphb_ltz ? -SettingONE :
            SettingONE);
             const float distdslv = (phphb_ltz ? (lhphb_ltz ? SettingNPNPDIST :
            SettingNPPDIST) : (lhphb_ltz ? SettingNPPDIST : -FloatMax));
             const float r_distdslv = 1.f / (distdslv);
             const float chrg_init = l_params.elsc * p_params.elsc;
             const float dslv_init = p_hphb + l_hphb;
            #pragma unroll 2
             for (size_t iway = 0; iway < NUMTDPERTHREAD; iway++) {
             const float x = lpos[iway].x() - p_TheAtom.x;
             const float y = lpos[iway].y() - p_TheAtom.y;
             const float z = lpos[iway].z() - p_TheAtom.z;
             const float distij = cl::sycl::sqrt(x * x + y * y + z * z);
             const float distbb = distij - radij;
             const bool zSettingONE1 = (distbb < SettingZERO);
             etot[iway] += (SettingONE - (distij * r_radij)) * (zSettingONE1 ? 2 *
            SettingHARDNESS : SettingZERO);
             float chrg_e = chrg_init * ((zSettingONE1 ? 1 : (SettingONE - distbb * elcdst1))
            * (distbb < elcdst ? 1 : SettingZERO));
             const float neg_chrg_e = -cl::sycl::fabs(chrg_e);
             chrg_e = type_E ? neg_chrg_e : chrg_e;
             etot[iway] += chrg_e * SettingCNSTNT;
             const float coeff = (SettingONE - (distbb * r_distdslv));
             float dslv_e = dslv_init * ((distbb < distdslv&& phphb_nz) ? 1 : SettingZERO);
             dslv_e *= (zSettingONE1 ? 1 : coeff);
             etot[iway] += dslv_e;
             }
         } while (++ip < natpro);
    } while (++il < natlig);
     const size_t td_base = gid * lrange * NUMTDPERTHREAD + lid;
     if (td_base < nposes) {
        #pragma unroll 4
         for (size_t iway = 0; iway < NUMTDPERTHREAD; iway++) {
             Energytotals[td_base + iway * lrange] = etot[iway] * SettingHALF;
         }
     }
 });
 #endif
}

double elapsedMillis(const TimePoint& start, const TimePoint& end) {
 auto elapsedNs = static_cast<double>(
 std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count());
 return elapsedNs * 1e-6;
}

void printTimings(const Params& params, double millis) {
 double msecond = (millis / params.iterations);
 double runkerneltime = msecond * 1e-3;
 double ops_per_wg = NUMTDPERTHREAD * 27 + params.natlig * (3 + NUMTDPERTHREAD * 18 + params.natpro * (11 + NUMTDPERTHREAD * 30) ) + NU
 double total_ops = ops_per_wg * ((double)params.nposes / NUMTDPERTHREAD);
 double flops = total_ops / runkerneltime;
 double gflops = flops / 1e9;
 double interactions = (double)params.nposes * (double)params.natlig * (double)params.natpro;
 double interactions_per_sec = interactions / runkerneltime;
 std::cout.precision(3);
 std::cout << std::fixed;
 std::cout << "- AverageKernelTime: " << msecond << " msecond\n";
 std::cout << "- InteractionPerSecond (billion): " << (interactions_per_sec / 1e9) << " billion\n";
 std::cout << "- FloatingPointoperations (Giga): " << gflops << "\n";
}

template<typename T>
std::vector<T> readNStruct(const std::string& path) {
 std::fstream s(path, std::ios::binary | std::ios::in);
 if (!s.good()) {
 throw std::invalid_argument("BadFile: " + path);
 }
 s.ignore(std::numeric_limits<std::streamsize>::max());
 auto len = s.gcount();
 s.clear();
 s.seekg(0, std::ios::beg);
 std::vector<T> xs(len / sizeof(T));
 s.read(reinterpret_cast<char*>(xs.data()), len);
 s.close();
 return xs;
}

Params loadParameters(const std::vector<std::string>& args) {
 Params params = {};
 params.iterations = ITERSDEFAULT;
 params.nposes = NPOSESDEFAULT;
 params.workgroupsize = workgroupsizeDEFAULT;
 params.deckDir = DIRDATA;
 const auto readParam = [&args](size_t& current,
 const std::string& arg,
 const std::initializer_list<std::string>& matches,
 const std::function<void(std::string)>& handle) {
 if (matches.size() == 0) return false;
 if (std::find(matches.begin(), matches.end(), arg) != matches.end()) {
```

```
    if (current + 1 < args.size()) {
        current++;
        handle(args[current]);
    }
    else {
        std::cerr << "[";
        for (const auto& m : matches) std::cerr << m;
        std::cerr << "]␣specified␣but␣no␣value␣was␣given" << std::endl;
        std::exit(EXIT_FAILURE);
    }
    return true;
}
return false;
};
const auto bindInt = [](const std::string& param, size_t& dest, const std::string& name) {
try {
    auto parsed = std::stol(param);
    if (parsed < 0) {
        std::cerr << "positive␣integer␣required␣for␣<" << name << ">:␣`" << parsed << "`" << std::endl;
        std::exit(EXIT_FAILURE);
    }
    dest = parsed;
}
catch (...) {
    std::cerr << "malformed␣value,␣integer␣required␣for␣<" << name << ">:␣`" << param << "`" << std::endl;
    std::exit(EXIT_FAILURE);
}
};
for (size_t iway = 0; iway < args.size(); ++iway) {
    using namespace std::placeholders;
    const auto arg = args[iway];
    if (readParam(iway, arg, { "--iterations", "-iway" }, std::bind(bindInt, _1, std::ref(params.iterations),
   "iterations"))) continue;
    if (readParam(iway, arg, { "--numposes", "-n" }, std::bind(bindInt, _1, std::ref(params.nposes),
   "numposes"))) continue;
    if (readParam(iway, arg, { "--workgroupsize", "-w" }, std::bind(bindInt, _1,
    std::ref(params.workgroupsize), "workgroupsize"))) continue;
    if (readParam(iway, arg, { "--deck" }, [&](const std::string& param) { params.deckDir = param; }))
    continue;
    if (arg == "--help" || arg == "-h") {
        std::cout << "\n";
        std::cout << "Usage:␣./main␣[OPTIONS]\n\n"
        << "Options:\n"
        << "␣-h␣--help␣Print␣this␣message\n"
        << "␣-iway␣--iterations␣iway␣Repeat␣kernel␣iway␣times␣(default:␣" << ITERSDEFAULT <<
       ")\n"
        << "␣-n␣--numposes␣N␣Compute␣energies␣for␣N␣poses␣(default:␣" << NPOSESDEFAULT
      << ")\n"
       // << " -p --poserperwi PPWI Compute PPWI poses per work-item (default: " <<
      //PPWIDEFAULT << ")\n"
        << "␣-w␣--workgroupsize␣workgroupsize␣Run␣with␣work-group␣size␣workgroupsize␣using␣nd␣range,␣set␣to␣0␣for␣plain␣range␣(default:␣" <
        << "␣--deck␣DECK␣Use␣the␣DECK␣directory␣as␣input␣deck␣(default:␣" << DIRDATA << ")"
        << std::endl;
        std::exit(EXIT_SUCCESS);
    }
    std::cout << "Unrecognized␣argument␣'" << arg << "'␣(try␣'--help')" << std::endl;
    std::exit(EXIT_FAILURE);
}
params.ligand = readNStruct<TheAtom>(params.deckDir + LIGANDFILE);
params.natlig = params.ligand.size();
params.protein = readNStruct<TheAtom>(params.deckDir + PROTEINFILE);
params.natpro = params.protein.size();
params.Force_Field = readNStruct<TheFFParams>(params.deckDir + Force_FieldFILE);
params.ntypes = params.Force_Field.size();
auto poses = readNStruct<float>(params.deckDir + POSESFILE);
if (poses.size() / 6 != params.nposes) {
    throw std::invalid_argument("BadPoses:␣" + std::to_string(poses.size()));
}
for (size_t iway = 0; iway < 6; ++iway) {
    params.poses[iway].resize(params.nposes);
    std::copy(
    std::next(poses.cbegin(), iway * params.nposes),
    std::next(poses.cbegin(), iway * params.nposes + params.nposes),
    params.poses[iway].begin());
}
return params;
}

template<typename T>
static clsycl::buffer<T> mkBuffer(clsycl::queue& queue, const std::vector< T>& xs) {
 clsycl::buffer<T> buffer(xs.size());
 queue.submit([&](clsycl::handler& h) {
 h.copy(xs.data(), buffer.template get_access<RW>(h));
 });
 return buffer;
}

std::vector<float> runKernel(Params params) {
 std::vector<float> energies(params.nposes);
#if FPGA_EMULATOR
 ext::intel::fpga_emulator_selector d_selector;
```

```
#elif FPGA
 ext::intel::fpga_selector d_selector;
#else
 default_selector d_selector;
#endif
 try {
     sycl::queue q(d_selector, dpc_common::exception_handler);
     cout << "Device:␣" << q.get_device().get_info<info::device::name>() << "\n";
     buffer<TheAtom, 1> protein(params.protein.data(), params.natpro);
     buffer<TheAtom, 1> ligand(params.ligand.data(), params.natlig);
     buffer<float, 1> Trans_FormsZreo(params.poses[0].data(), params.nposes);
     buffer<float, 1> Trans_FormsOne(params.poses[1].data(), params.nposes);
     buffer<float, 1> Trans_FormsTwo(params.poses[2].data(), params.nposes);
     buffer<float, 1> Trans_FormsThree(params.poses[3].data(), params.nposes);
     buffer<float, 1> Trans_FormsFour(params.poses[4].data(), params.nposes);
     buffer<float, 1> Trans_FormsFive(params.poses[5].data(), params.nposes);
     buffer<TheFFParams, 1> Force_Field(params.Force_Field.data(), params.ntypes);
     buffer<float> results(energies.size());
     q.submit([&](handler& h) {
         FastenMain(h,
         params.workgroupsize,
         params.ntypes,
         params.nposes,
         params.natlig,
         params.natpro,
         protein.get_access<sycl_read>(h),
         ligand.get_access<sycl_read>(h),
         Trans_FormsZreo.get_access<sycl_read>(h),
         Trans_FormsOne.get_access<sycl_read>(h),
         Trans_FormsTwo.get_access<sycl_read>(h),
         Trans_FormsThree.get_access<sycl_read>(h),
         Trans_FormsFour.get_access<sycl_read>(h),
         Trans_FormsFive.get_access<sycl_read>(h),
         Force_Field.get_access<sycl_read>(h),
         results.get_access<sycl_discard_write>(h));
     });
     q.wait();
     auto kernelStart = std::chrono::high_resolution_clock::now();
     for (size_t iway = 0; iway < params.iterations; ++iway) {
         q.submit([&](handler& h) {
             FastenMain(h,
             params.workgroupsize,
             params.ntypes,
             params.nposes,
             params.natlig,
             params.natpro,
             protein.get_access<sycl_read>(h),
             ligand.get_access<sycl_read>(h),
             Trans_FormsZreo.get_access<sycl_read>(h),
             Trans_FormsOne.get_access<sycl_read>(h),
             Trans_FormsTwo.get_access<sycl_read>(h),
             Trans_FormsThree.get_access<sycl_read>(h),
             Trans_FormsFour.get_access<sycl_read>(h),
             Trans_FormsFive.get_access<sycl_read>(h),
             Force_Field.get_access<sycl_read>(h),
             results.get_access<sycl_discard_write>(h));
         });
     }
     q.wait();
     auto kernelEnd = std::chrono::high_resolution_clock::now();
     q.submit([&](handler& h) {
         auto acc = results.get_access<sycl_read>(h);
         h.copy(acc, energies.data());
     });
     q.wait();
     printTimings(params, elapsedMillis(kernelStart, kernelEnd));
     return energies;
 }
 catch (sycl::exception const& e) {
     cout << "An␣exception␣is␣caught␣while␣running.\n";
     terminate();
 }
}
int main(int argc, char* argv[]) {

 auto args = std::vector<std::string>(argv + 1, argv + argc);
 auto params = loadParameters(args);
 std::cout << "PosesNUMBER␣:␣" << params.nposes << std::endl;
 std::cout << "IterationsNUMBER␣:␣" << params.iterations << std::endl;
 std::cout << "LigandsNUMBER␣:␣" << params.natlig << std::endl;
 std::cout << "ProteinsNUMBER␣:␣" << params.natpro << std::endl;
 std::cout << "DeckPLACE␣:␣" << params.deckDir << std::endl;
 std::cout << "WGNUMBER␣:␣" << params.workgroupsize << std::endl;
 auto energies = runKernel(params);
 std::ifstream refEnergies(params.deckDir + FILEREFENERGIES);
 size_t nRefPoses = params.nposes;
 if (params.nposes > NPOSESREF) {
     std::cout << "Only␣validating␣the␣first␣" << NPOSESREF << "␣poses.\n";
     nRefPoses = NPOSESREF;
 }
 std::string line;
```

```
 float maxdifference = 0.0f;
 for (size_t iway = 0; iway < 16; iway++) {
 //for (size_t iway = 0; iway < nRefPoses; iway++) {
     if (!std::getline(refEnergies, line)) {
         throw std::logic_error("ran out of ref energies lines to verify");
     }
     float e = std::stof(line);
     if (std::fabs(e) < 1.f && std::fabs(energies[iway]) < 1.f) continue;
     float diff = std::fabs(e - energies[iway]) / e;
     if (diff > maxdifference) maxdifference = diff;
 }
 std::cout << "Largest difference was " << std::setprecision(4) << (100 * maxdifference) << "%.\n\n";
 refEnergies.close();
 return 0;
}
```

Bibliography

[1] Bude-harp. URL https://github.com/eejlny/BUDE-HARP.

[2] minibude. URL https://github.com/UoB-HPC/miniBUDE.

[3] Sycl™ 2020 specification (revision 5). 2020. URL https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#_sycl_execution_model.

[4] Anatomy of a dpc++ program. 2020. URL https://docs.oneapi.io/versions/latest/model/sample-program.html.

[5] Exploring the gpu architecture. 2022. URL https://core.vmware.com/resource/exploring-gpu-architecture.

[6] Intel oneapi base training modules. 2022. URL https://devcloud.intel.com/oneapi/get_started/baseTrainingModules/.

[7] Intel xeon gold 6128, 2023. URL https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+Gold+6128+%40+3.40GHz&id=3104.

[8] Intel uhd graphics p630, 2023. URL https://www.videocardbenchmark.net/gpu.php?gpu=Intel+UHD+Graphics+P630&id=3924.

[9] Stephen J. Bigelow. What is a multicore processor?, 2022. URL https://www.techtarget.com/searchdatacenter/definition/multi-core-processor.

[10] David Both. The central processing unit (cpu): Its components and functionality. 2020. URL https://www.redhat.com/sysadmin/cpu-components-functionality.

[11] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 47–47, 2004. doi: 10.1109/SC.2004.26.

[12] Rahul Garg and Laurie Hendren. A portable and high-performance general matrix-multiply (gemm) library for gpus and single-chip cpu/gpu systems. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 672–680, 2014. doi: 10.1109/PDP.2014.40.

[13] Wei-Chih Huang. Evaluation of the intel cloud fpga platforms for high performance computing. 2022.

[14] Eduardo de la Torre Jose Nunez-Yanez, Andres Otero. Dynamically reconfigurable variable-precision sparse-dense matrix acceleration in Tensorflow Lite. *Microprocessors and Microsystems*, page 104801, February 2023. doi: 10.1016/j.micpro.2023.104801.

[15] Srinidhi Kestur, John D. Davis, and Oliver Williams. Blas comparison on fpga, cpu and gpu. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pages 288–293, 2010. doi: 10.1109/ISVLSI.2010.84.

[16] Mantas Levinas. Gpu vs cpu: What are the key differences? 2020. URL https://www.cherryservers.com/blog/gpu-vs-cpu-what-are-the-key-differences.

[17] Duncan J.M Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 107–116, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356145. doi: 10.1145/3174243.3174258. URL https://doi.org/10.1145/3174243.3174258.

[18] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. 01 2021. ISBN 978-1-4842-5573-5. doi: 10.1007/978-1-4842-5574-2.

[19] Germán Castaño Roldán. Intel-oneapi for heterogeneous computing. 2021. URL https://eprints.ucm.es/id/eprint/66814/1/CASTA%C3%91O%20ROLD%C3%81N%2090031_GERMAN_CASTANO_ROLDAN_Intel-oneAPI_for_heterogeneous_computing_784051_962915113.pdf.

[20] Francesca Stanzione, Ilenia Giangreco, and Jason C. Cole. Chapter four - use of molecular docking computational tools in drug discovery. volume 60 of *Progress in Medicinal Chemistry*, pages 273–343. Elsevier, 2021. doi: https://doi.org/10.1016/bs.pmch.2021.01.004. URL https://www.sciencedirect.com/science/article/pii/S0079646821000047.

[21] Bharat Sukhwani. Accelerating molecular docking and binding site mapping using fpgas and gpus. 2011.

[22] Prasanna Sundararajan. High performance computing using fpgas. 2010.

[23] A Tekin, A Tuncer Durak, C Piechurski, D Kaliszan, F Aylin Sungur, F Robertsén, and P Gschwandtner. State-of-the-art and trends for computing and interconnect network solutions for hpc and ai. *Partnership for Advanced Computing in Europe, Available online at www. praceri. eu*, 2021.

[24] Qiang Wu, Yajun Ha, Akash Kumar, Shaobo Luo, Ang Li, and Shihab Mohamed. A heterogeneous platform with gpu and fpga for power efficient high performance computing. In *2014 International Symposium on Integrated Circuits (ISIC)*, pages 220–223, 2014. doi: 10.1109/ISICIR.2014.7029447.

[25] Bob Zeidman. All about FPGAs. *EDN*, 2006. URL `https://www.edn.com/all-about-fpgas/`.