Informatica — Universiteit van Amsterdam

Bachelor Informatica

University of Amsterdam

From VexCL to FPGAs

Tristan Laan

June 18, 2021

Supervisor(s): dr. ir. A.L. Varbanescu

Abstract

Field-programmable gate arrays (FPGAs) are capturing the interest of the high performance computing community. However, there are many computing frameworks that only support the use of GPUs as accelerators, and, therefore, are not FPGA-ready.

In this thesis we investigate such a framework, VexCL, and propose a porting process to turn VexCL code into a Xilinx Vitis application, which can run on a Xilinx FPGA. Moreover, we expand this process to include FPGA-specific optimizations.

We develop and demonstrate the process on an affine transformation application. To validate the generality of the process, we have also tested it on a second VexCL application. This second case-study demonstrates that the process and a part of the optimizations are portable.

We conclude that out porting process is systematic in nature and applicable for many VexCL applications. These promising results indicate that our process can be (partially) automated in a compiler, a task left for future work.

Contents

Abbreviations **7**

1 Introduction **9**
1.1 Context 9
1.2 Research question and approach 9
1.3 Ethics 10
1.4 Thesis Outline 10

2 Background **11**
2.1 High-Level Synthesis 11
2.2 VexCL 11
2.3 FPGA architecture 11
2.3.1 Xilinx Alveo U250 Architecture 12
2.4 Compilation process 12
2.5 Related Work 12
2.5.1 Porting 13
2.5.2 Optimizing 13

3 Porting a VexCL application **15**
3.1 Program description 15
3.2 VexCL implementation 15
3.3 VexCL kernel 17
3.4 Xilinx Vitis implementation 18
3.4.1 Device code 19
3.4.2 Host code 19
3.5 Verifying correctness 22
3.6 Compiling and running the ported application 22
3.6.1 Compilation and execution time 22
3.6.2 Compiled FPGA kernel 23
3.6.3 Application timeline 25
3.7 Porting guidelines 25

4 Improving the performance of a ported application **27**
4.1 Optimizations 27
4.1.1 Memory alignment 27
4.1.2 Fixed-point arithmetic 27
4.1.3 Burst transfers 28
4.1.4 Saturating data width 28
4.2 Applying the optimizations 28
4.2.1 Memory alignment 28
4.2.2 Fixed-point arithmetic 29
4.2.3 Burst transfers 29

4.2.4 Saturating data width . . . 31
4.3 Testing the optimizations . . . 31
4.3.1 Implementations . . . 32
4.3.2 Input sizes . . . 33
4.3.3 Hardware . . . 33
4.3.4 Profiling tools . . . 33
4.3.5 Method . . . 33
4.3.6 Results . . . 33
4.3.7 Discussion . . . 37
4.4 Summary . . . 37

5 The effectiveness of the porting process **41**
5.1 Sparse matrix representation . . . 41
5.1.1 CSR format . . . 41
5.1.2 ELL format . . . 42
5.1.3 Hybrid ELL–CSR format . . . 42
5.2 VexCL implementation . . . 42
5.2.1 OpenCL kernel . . . 44
5.3 Xilinx Vitis implementation . . . 45
5.3.1 Porting process . . . 45
5.3.2 Compiled application . . . 46
5.3.3 Verifying correctness . . . 46
5.4 Performance study . . . 47
5.4.1 Implementations . . . 47
5.4.2 Method . . . 48
5.4.3 Results . . . 48
5.5 Summary . . . 52

6 Conclusion **55**
6.1 Main findings . . . 55
6.2 Contributions . . . 56
6.3 Limitations . . . 56
6.4 Future Work . . . 57

Bibliography **59**

A Testing result data **61**

Abbreviations

BRAM block random access memory. 11, 12, 23, 33, 34, 37

CPU central processing unit. 22, 33, 46

CSR compressed sparse row. 41, 42, 44, 46

DDR double data rate. 12, 18, 23, 28

DSP digital signal processing. 11, 12, 23, 33

ELL ELLPACK. 42, 44

FPGA field-programmable gate array. 3, 9–13, 18–20, 22, 23, 27, 28, 32–34, 37, 45, 46, 48, 50, 52, 55–57

GPU graphics processing unit. 3, 9, 11, 13, 22, 27, 33, 34, 37, 42, 48, 50, 57

HDL hardware description language. 11

HELL hybrid ELL–CSR. 41, 42, 44, 46, *see* CSR & ELL

HLS high-level synthesis. 11, 22, 55

HPC high-performance computing. 9, 55

JSON JavaScript Object Notation. 22, 33

LUT look-up table. 11, 12, 23, 27, 33

MT/s megatransfers per second. 12

nvidia-smi NVIDIA System Management Interface. 33

PCIe Peripheral Component Interconnect Express. 12

SLR super logic region. 12

SpMV sparse matrix-vector multiplication. 41, 42, 44, 46, 48, 50, 52, 55, 56, 61, 62

XO Xilinx object. 22, 45

CHAPTER 1

Introduction

1.1 Context

Field-programmable gate arrays (FPGAs) are capturing the interest of the high-performance computing (HPC) community to use as accelerator, because of their favourable energy consumption compared to other accelerators, like graphics processing units (GPUs) [11, 12]. It is also becoming easier than ever to program FPGAs, due to ongoing efforts to enable OpenCL[1], C or C++ programs to target FPGAs [23, 12]. Xilinx currently provides the Xilinx Vitis[2] platform to run C++ and OpenCL programs on Xilinx FPGAs.

However, many frameworks in the HPC community, like Halide[3] and PyTorch[4], aim to make it easier to write high-performance applications, and target GPU-like accelerators without explicitly writing OpenCL or CUDA. Yet most such frameworks only support GPUs. With the emergence of FPGAs in the HPC community, adding support for FPGAs in such frameworks can be highly beneficial to understand whether FPGAs can indeed become HPC accelerators.

VexCL is a C++ library that adds support for vector arithmetic on a GPU using standard C++ operators [3]. Just like Halide and PyTorch, VexCL only supports using GPUs as accelerator. In this project we will focus on (1) designing and developing a strategy to run VexCL code on Xilinx FPGAs, using Xilinx Vitis as target language, (2) proposing optimizations to this strategy to improve the performance of the application on FPGAs, and (3) evaluating this strategy using representative applications.

1.2 Research question and approach

Our main research question is:
How can VexCL code be effectively compiled into code for FPGAs?

To determine how we can compile code from VexCL to Xilinx Vitis for FPGAs, we first need to select which intermediate representation that VexCL can produce (e.g., OpenCL or OpenMP) to use. Thus, our first subquestion is:
[SQ1] What language supported by VexCL is a convenient intermediate representation for compiling VexCL to Xilinx Vitis code for FPGAs?
To answer this question, we study which intermediate representations VexCL can produce, experiment with these intermediate representations, and study previous work to determine what intermediate representations have already been used in different case-studies.

[1]OpenCL – https://www.khronos.org/opencl/
[2]Xilinx Vitis – https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html
[3]Halide – https://halide-lang.org/
[4]PyTorch – https://pytorch.org/

Once both the input and output languages are selected, we can proceed designing our translation process, driven by the subquestion:
[SQ2] How can we design a step-by-step guide to convert VexCL code to Xilinx Vitis code that targets FPGAs?
Here we will determine the requirements for our compiler and study design practices for FPGA compilers. Following our design, we will provide a first prototype compiler based on this design.

Next we evaluate possible general optimizations that can be applied to ported applications. Thus, we propose the following subquestion:
[SQ3] What optimizations can we apply to applications ported from VexCL to Xilinx Vitis code that improve the performance of the code?
To answer this question, we will research several optimizations for Xilinx FPGAs and apply them to a ported application to test how they perform.

Finally, we plan to validate the step-by-step compilation guide. To do so, we formulate one final subquestion:
[SQ4] How effective is the compilation guide?
To find out how effective the guide is, we use a more complicated VexCL application, and port it to Xilinx Vitis following our guidelines. We use this application to assess the correctness, completeness, and limitations of our guide.

1.3 Ethics

To support the computer science community, and improve innovation in this field, all the code used in this thesis is available open-source[5]. Furthermore, we ensure, to the best of our abilities, that all design, implementation, data collection, and analysis included in this thesis are correct and transparent.

Indiscriminate testing and exhaustive experiments with FPGAs can consume significant resources. Therefore, for all our development, testing, and validation, we made use of simulation and emulation for the early stages, and only used the expensive hardware design process for the final sets of experiments.

1.4 Thesis Outline

Chapter 2 provides background information about VexCL, Xilinx FPGAs, and on previous work in this research area. Then, in chapter 3, we present our step-by-step porting guide with a case study of porting a VexCL application to Xilinx Vitis, here we also evaluate the compilation of the ported application and execution of the application on a Xilinx FPGA. In chapter 4 we propose several optimizations that can be applied to a ported application, and evaluate the performance of the optimizations applied to the application we used in our case study. We further evaluate the effectiveness of the porting guide in chapter 5 by testing it on another VexCL application. Lastly, in chapter 6, we conclude the effectiveness of our porting guide and propose future work that can be done to improve the porting process of VexCL applications to Xilinx FPGAs.

[5]Source code – `https://gitlab.com/tristanlaan/vexcl2fpgas`

CHAPTER 2

Background

2.1 High-Level Synthesis

Historically, special hardware description languages (HDLs), like Verilog and VHDL, have been used to program FPGAs. Such languages provide an abstract way to describe computer circuits by describing what a module does, instead of its logical implementation [14]. A module in this case is the HDL equivalence of a function in a programming language, and represents a logical unit on the FPGA. A module for example describes which logical gates are used and how they are used together in a high-level description. The HDL will then compile this module into a hardware circuit that can be programmed into the FPGA. To do this, the HDL tool has to allocate resources on the FPGA, schedule operations to clock cycles, and bind them to the resources, bind variables to storage units, and bind transfers to buses.

To make programming FPGAs easier and more wide-spread, high-level synthesis (HLS) tools have been introduced. A HLS tool takes a source language as input and compiles it either to a HDL, or directly into a hardware circuit [2]. The first HLS tools were created in the 1970s and became widely used in the mid-2000s [9]. HLS tools have been using C/C++ languages or derivations of it (such as SystemC) as input language; in more recent years, research has been put in using languages developed specifically for heterogeneous computation, like OpenCL, as source language [13]. FPGA manufacturers, like Xilinx and Intel, have developed toolchains that support OpenCL programs making it easier for new users to start using FPGAs [20, 7]. The advantage of using OpenCL is that there are many applications that already use OpenCL kernels for GPU acceleration.

2.2 VexCL

VexCL is an OpenCL library that aims to make general purpose computing on GPUs easier by providing intuitve notation for vector and matrix arithmetic [3]. Although there are alternative libraries that provide better performance, its ease-of-use and the provided wrapper functions stand out [10]. VexCL currently supports compiling to OpenCL, OpenMP and C++, but also has support for custom backends.

2.3 FPGA architecture

FPGA architectures differ from more traditional computer designs in the fact that FPGAs consist of different components. The most important components are look-up tables (LUTs), digital signal processing (DSP) slices, registers and block random access memories (BRAMs) [21].

A LUT takes a set amount of bits as input and produces a smaller amount of bits based on the input. This means that a LUT of k bits can execute an arbitrary Boolean circuit consisting of k bits by programming the truth table of the circuit into the LUT. LUTs generally have one or two output bits and five or six input bits, but it is possible to chain multiple LUTs to create circuits which support an arbitrary amount of input and output bits. [17]

A DSP slice consists of more complex components that are specifically targeted towards DSP applications. They have more input and output bits than LUTs and can apply more complex operations like multiplication, single instruction, multiple data arithmetic of up to 48 bits and applying digital filters, like a Gaussian or Bloom filter. These DSP targeted operations are also useful for complex arithmetic in more general applications. [18]

Registers and BRAMs provide a way store data in the process. Registers can store a few bits and BRAMs can store larger amounts of data. [17]

The components in a FPGA will, once programmed, be interconnected into a pipeline which executes the design provided by the user.

2.3.1 Xilinx Alveo U250 Architecture

The Xilinx Alveo U250 accelerator card consists of four super logic regions (SLRs). A SLR can store multiple kernels provided by the user, but a kernel can only target one SLR. Each SLR contains general FPGA components that can be programmed in to a pipeline based on a kernel provided by the user. Each SLR also connects to 16 GB of DDR4 memory running at 2400 megatransfers per second (MT/s). Additionally, the first SLR is also connected to the host device via a Peripheral Component Interconnect Express (PCIe) Gen 3 connection using 16 lanes with a maximum speed of 8000 MT/s, and the third SLR has two network interfaces which both support speeds up to 100 Gb/s. In total, all the SLRs consist of 1,728,000 LUTs, 3,456,000 registers, 12,288 DSPs slices and 1,280 memory blocks. [15]

2.4 Compilation process

Xilinx provides three compilation targets, namely software emulation, hardware emulation and hardware execution. Xilinx provides these emulation targets to be able to test the code without access to a FPGA, but also because it is significantly faster to compile the code for the emulators than to compile the code for hardware execution. [21]

The software emulator runs the kernel sequentially on the CPU and allows to check the code for correctness, while having a very short compilation time.

The hardware emulator runs the code on a simulated FPGA, and can be used to estimate the performance of the kernel on a FPGA and to check if the code would run correctly on a FPGA. Compiling for the hardware emulator than for the software emulator, and the software emulator is faster than the hardware emulator.

With hardware execution the kernel will be run on the FPGA, and it can be used to verify that the code runs correctly and to check how fast it performs on the FPGA. Compiling for hardware execution takes longer than compiling for both the software and hardware emulator.

2.5 Related Work

In this section we describe relevant related work. Specifically, we focus on porting applications to other programming languages, and applying optimizations to OpenCL kernels for FPGAs. We note that, because the Xilinx Vitis platform is still very new (it was only released in October of 2019 [22]), the research work using the Xilinx Vitis platform is still very limited.

2.5.1 Porting

Gozillon et al. are currently working on allowing SYCL[1] code to target Xilinx FPGAs, using their triSYCL framework[2]. SYCL is a C++ library that, like VexCL, aims to make it possible to write code for heterogeneous computation without the need to write the kernel and host code separately from each other. Early work from Gozillon et al. has shown that they have been successful in incorporating the Vitis compiler in the SYCL platform, allowing them to run SYCL applications using a Xilinx FPGA as accelerator [5]. Currently they have only verified correctness for a single Xilinx FPGA, and the performance does not yet reach the performance of Xilinx Vitis, but they are still working to improve this. This work is similar to our work, we both aim to add support for a C++ framework which does not currently support using FPGAs as accelerator, and add possible optimizations to this compilation process. VexCL differs from SYCL however, in the sense that VexCL not only aims to be able to write code for heterogeneous computation in a single file, but also to simplify the code and minimize the knowledge required about heterogeneous computing to be able to write the code. Due to how tied this work is to the SYCL framework, we can not directly use their work to port VexCL applications, but we can take inspiration from their compilation pipeline to possibly integrate the Vitis compiler as VexCL back-end.

2.5.2 Optimizing

Recent works have shown that GPU-optimized OpenCL kernels can be altered using certain strategies to be optimized for FPGAs [23, 12]. These strategies consist of optimizations such as loop unrolling, shift registers and sliding windows. Optimized programs using such optimization techniques can be up to 66× faster than running a non-optimized program on the FPGA [23]. Although these works are not specifically targeted to the Xilinx Vitis platform, they could be adapted and integrated into our porting process.

[1] SYCL – `https://www.khronos.org/sycl`
[2] triSYCL – `https://github.com/triSYCL/triSYCL`

CHAPTER 3

Porting a VexCL application

3.1 Program description

The program we will use for our case-study implements a simple affine transformation: it calculates the expression given in equation 3.1, where $\vec{y}$ and $\vec{t}$ are vectors of length m, $\vec{x}$ is a vector of length n and A is a $m \times n$ matrix.

$$\vec{t} = \vec{y} + A\vec{x} \tag{3.1}$$

3.2 VexCL implementation

To start writing a VexCL program, we need some boilerplate code first, as can be seen in listing 1. The code simply includes the VexCL library and specifies that we want to use double precision math on our accelerator. We also set the `VEXCL_SHOW_KERNELS` flag because we want to see the OpenCL kernel that VexCL uses.

```
1  #define CL_TARGET_OPENCL_VERSION 120
2  #define VEXCL_SHOW_KERNELS
3  #include <vexcl/vexcl.hpp>
4
5  int main(int argc, char **argv) {
6      vex::Context ctx(vex::Filter::DoublePrecision);
7      ...
8      return 0;
9  }
```

Listing 1: The boilerplate VexCL code.

We further need to initialize the input data of our equation, which can be seen in listing 2. We first need to create the input data on the host device, and then the data can be copied to the accelerator device. For the host side vectors we can simply use the std::vector implementation of C++. We implement the matrix A as a flattened vector, because VexCL does not support dense matrices without external libraries. Once the host vectors are initialized, we can specify the device-side vectors using vex::vector and copy the host-side data into the vectors. We have to cast the doubles of the host-side vectors to cl_doubles for the device-side vectors. Note that

we leave the output vector $\vec{t}$ uninitialized on the accelerator, because the data in the vector will later be overwritten by the result of the equation.

```
size_t m = 7, n = 5;
std::vector<double> a(m * n), x(n), y(m), t(m);

// Initialize matrix + vectors
...

// Transfer host-side doubles into device-side cl_double vectors
vex::vector<cl_double> A(ctx, a.size(), reinterpret_cast<cl_double*>(a.data()));
vex::vector<cl_double> X(ctx, x.size(), reinterpret_cast<cl_double*>(x.data()));
vex::vector<cl_double> Y(ctx, y.size(), reinterpret_cast<cl_double*>(y.data()));
vex::vector<cl_double> T(ctx, t.size();
```

Listing 2: The VexCL data initialization.

Once the vectors are initialized we can start calculating the equation. The relevant code can be seen in listing 3. The addition is very simple as VexCL simply overloads the addition operator to support device-side vector addition. VexCL does not support matrix multiplication out of the box however, so we have to implement this ourselves with VexCL commands. To perform the matrix vector multiplication $A\vec{x}$, we extent the vector $\vec{x}$ to a $m \times n$ (7×5) matrix X by repeating $\vec{x}^\intercal$ for each row of X. Then we will perform an element-wise multiplication between A and X and reduce the result to a vector by summing the columns of the result together. Note that all the matrices are still implemented as flattened vectors, but the `reshape` and `reduce` functions of VexCL behave like the arguments are matrices and will handle the index conversion for us.

```
template <class M, class V>
auto prod(size_t m, size_t n, M &&A, V &&x) {
    using namespace vex;
    // Specify M×N matrix shape.
    auto MxN = extents[m][n];
    // Reshape x to a matrix by copying x into each row of X.
    auto X = reshape(x, MxN, extents[1]);
    // Multiply A with X element-wise.
    auto E = A * X;
    // Reduce matrix E to a vector of size M by summing over dimension 1.
    return reduce<SUM>(MxN, E, 1);
}

int main(int argc, char **argv) {
    ...
    T = Y + prod(m, n, A, X);
    ...
}
```

Listing 3: Equation 3.1 calculated in VexCL.

When the calculations are finished, we can copy the results back to the host-device vector $\vec{t}$, as can be seen in listing 4. Note that we have to cast the cl_doubles back to normal doubles.

```
1 vex::copy(T.begin(), T.end(), reinterpret_cast<cl_double*>(t.data()));
```

Listing 4: VexCL copying back the results.

Original parameter name	**Reference name**
n	parameter 0
prm_1	buffer 1
prm_2	buffer 2
prm_3_1	buffer 3
prm_3_2_expr_1	buffer 4
prm_3_2_slice_1	slice 1
prm_3_2_slice_2	slice 2
prm_3_2_slice_3	slice 3
prm_3_2_slice_4	slice 4
prm_3_start	reduce start
prm_3_length0	reduce length 0
prm_3_stride0	reduce stride 0
prm_3_length1	reduce length 1
prm_3_stride1	reduce stride 1

Table 3.1: Reference naming of VexCL kernel parameters to improve the readability.

3.3 VexCL kernel

Because we set the `VEXCL_SHOW_KERNELS` flag in our program, VexCL will output the OpenCL kernels it produced, which we can use to port the application to Xilinx Vitis. The produced kernel can be seen in listing 5. At the top of the file we have two pragmas that enable the double precision floating point numbers on supported hardware. Then we find an automatically generated function to sum two doubles. The actual kernel is generated as the kernel function `vexcl_vector_kernel` with 14 parameters. To improve the readability of this section, we will reference the parameters with the names defined in table 3.1. The first parameter, *parameter 0*, is the size of the output vector $\vec{t}$, which was equal to $m = 7$. The following parameters are the input and output buffers, which are ordered based on where they appear in the equation $\vec{t} = \vec{y} + A\vec{x}$. So *buffer 1* corresponds to $\vec{t}$, *buffer 2* corresponds to $\vec{y}$, *buffer 3* corresponds to A and *buffer 4* corresponds to $\vec{x}$.

Let us now take a look at the last five parameters, which are generated by the `vex::reduce` command, *reduce start*, *reduce length 0*, *reduce stride 0*, *reduce length 1* and *reduce stride 1*. The parameters *reduce length 0* and *reduce stride 0* are the size of the first and second dimension of the input matrix respectively, so in our case reduce length $0 = m$ and reduce length $1 = n$. The parameter *reduce length 1* corresponds to how many values need to be summed together and the parameter *reduce stride 1* indicates how many elements we have to skip in the underlying array of the input matrix to get to the next value to be summed. Because we are summing over the columns, reduce length $1 = n$ and reduce stride $1 = 1$. Lastly *reduce start* is a global offset in the array, which we do not need so reduce start $= 0$.

We are now only left with the parameters *slice 1*, *slice 2*, *slice 3* and *slice 4*, which are generated by the `vex::reshape` command. These parameters are used to convert a index into the flattened matrix X to an index into the vector $\vec{x}$, and are easier to understand if we look to the code where they are used. In the expression `(slice_1 * (((slice_2 + idx) / slice_3) % slice_4))`, `idx` is the index in the flattened matrix and the outcome of the expression is the index in the vector $\vec{x}$. If we start from `idx` we see that a global offset *slice 2* is added, this is only used if you slice an array, not when you reshape it, so in our case slice $2 = 0$. The next operation

in the expression is an integer division by *slice 1*. This division is used to repeat the several index multiple times and is useful to replicate a vector along the columns of a matrix, but that is not necessary in our case, so slice $1 = 1$. Then we see a modulo operation with *slice 4*, which is used to wrap around the index. In our case we wanted to replicate the vector along the rows of a matrix, so at each row of the matrix we want to start at index 0 again. This means slice $4 = |\vec{x}| = n$. Lastly we have a multiplication by *slice 1*, which can be used to skip every ith index if set to i. That is not needed in our case so slice $1 = 1$. The expression `(slice_1 * (((slice_2 + idx) / slice_3) % slice_4))` thus simplifies to (idx $\mod n$) in our case.

Now that we have defined all the parameters, we can see that the kernel consists of a parallelized loop that runs over the elements of $\vec{t}$, which has an inner loop to calculate $A\vec{x}$, then adds the sum to $\vec{y}$ and stores the result in $\vec{t}$.

```
#if defined(cl_khr_fp64)
#  pragma OPENCL EXTENSION cl_khr_fp64: enable
#elif defined(cl_amd_fp64)
#  pragma OPENCL EXTENSION cl_amd_fp64: enable
#endif

double SUM_double(double prm1, double prm2) {
  return prm1 + prm2;
}

kernel void vexcl_vector_kernel(ulong n, global double *prm_1,
  global double *prm_2, global double *prm_3_1, global double *prm_3_2_expr_1,
  ulong prm_3_2_slice_1, ulong prm_3_2_slice_2, ulong prm_3_2_slice_3,
  ulong prm_3_2_slice_4, ulong prm_3_start, ulong prm_3_length0,
  long prm_3_stride0, ulong prm_3_length1, long prm_3_stride1)
{
  for (ulong idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
    double prm_3_sum = 0;
    {
      size_t pos = idx;
      size_t ptr1 = prm_3_start + (pos % prm_3_length0) * prm_3_stride0;
      for (size_t i1 = 0, ptr2 = ptr1; i1 < prm_3_length1; ++i1,
           ptr2 += prm_3_stride1) {
        size_t idx = ptr2;
        prm_3_sum = SUM_double(prm_3_sum,
          (prm_3_1[idx] * prm_3_2_expr_1[(prm_3_2_slice_1 * (
          ((prm_3_2_slice_2 + idx) / prm_3_2_slice_3) % prm_3_2_slice_4))]));
      }
    }
    prm_1[idx] = (prm_2[idx] + prm_3_sum);
  }
}
```

Listing 5: The OpenCL kernel produced by VexCL.

3.4 Xilinx Vitis implementation

The Vitis implementation of the affine transformation needs two different files, where only one file is needed with VexCL, a file with the host code and a file with the device code.

3.4.1 Device code

For the device code we port the OpenCL kernel, generated by VexCL, to a Xilinx Vitis C kernel. Note that we could have used a OpenCL kernel as well for the Xilinx Vitis implementation, but not all Vitis features are available in OpenCL [21]. To convert the OpenCL kernel to a C kernel, we can mostly copy the OpenCL kernel with little changes needed, as is visible in listing 6. First we need to wrap the code in a `extern "C"` block to avoid name mangling issues between C and C++ [21]. We also need to change some of the data types from OpenCL types to C types, so we remove the `kernel` and `global` keyword and replace the `ulong`, `long` and `size_t` keywords with `int`. There are two OpenCL specific functions too that make sure that the loop is parallelized in OpenCL, `get_global_id` and `get_global_size`. The Vitis compiler does not have such a construct, so we can replace the for loop to simply start at zero, and increment by one. We specify the read-only buffers as `const` and rename the kernel to be more easily distinguishable. We need to add pragmas for the vectors to specify which interface protocol the FPGA should use, we will use the Advanced eXtensible Interface 4 protocol, as recommended by Xilinx [16]. Lastly we also have to create a create a configuration file for the specific FPGA to specify the main function and which memory interfaces the ports should be connected to. We will use DDR interface 1 for all the parameters in our case, meaning all data transfers will happen over the same memory interface. This configuration file can be seen in listing 7.

```
extern "C" {
  // Unchanged defines
  ...

  double SUM_double( double prm1, double prm2) {
    return prm1 + prm2;
  }

  void affinetrans(int n, double *prm_1, const double *prm_2,
    const double *prm_3_1, const double *prm_3_2_expr_1, int prm_3_2_slice_1,
    int prm_3_2_slice_2, int prm_3_2_slice_3, int prm_3_2_slice_4,
    int prm_3_start, int prm_3_length0, int prm_3_stride0, int prm_3_length1,
    int prm_3_stride1)
  {
    #pragma HLS INTERFACE m_axi port=prm_1 bundle=aximm1
    #pragma HLS INTERFACE m_axi port=prm_2 bundle=aximm1
    #pragma HLS INTERFACE m_axi port=prm_3_1 bundle=aximm1
    #pragma HLS INTERFACE m_axi port=prm_3_2_expr_1 bundle=aximm1

    for(int idx = 0; idx < n; ++idx)
    {
        // Unchanged computations
        ...
    }
}
```

Listing 6: A shortened version of the ported Xilinx Vitis kernel that can run on a FPGA.

3.4.2 Host code

The first step in creating the Vitis host code is to replace the boilerplate VexCL code with boilerplate code that enables a connection with the FPGA. As can be seen in listing 8, the defines change, and, instead of including VexCL, we include OpenCL. Setting up the context is also a bit different, where we only needed a single command for VexCL, we now need to write

```
platform=xilinx_u250_gen3x16_xdma_3_1_202020_1
debug=1
save-temps=1

[connectivity]
nk=affinetrans:1:affinetrans_1
sp=affinetrans_1.prm_1:DDR[1]
sp=affinetrans_1.prm_2:DDR[1]
sp=affinetrans_1.prm_3_1:DDR[1]
sp=affinetrans_1.prm_3_2_expr_1:DDR[1]

[profile]
data=all:all:all
```

Listing 7: A Xilinx configuration file for the affine transformation kernel on a U250 Xilinx FPGA.

our own code to connect to the FPGA, and to load the Vitis kernel to the FPGA.

```
#define CL_HPP_CL_1_2_DEFAULT_BUILD
#define CL_HPP_TARGET_OPENCL_VERSION 120
#define CL_HPP_MINIMUM_OPENCL_VERSION 120
#define CL_HPP_ENABLE_PROGRAM_CONSTRUCTION_FROM_ARRAY_COMPATIBILITY 1
#define CL_USE_DEPRECATED_OPENCL_1_2_APIS
#include <CL/cl2.hpp>
...
int main(int argc, char **argv) {
    // Get Xilinx device
    const cl::Device device = get_xilinx_devices().front();
    ...
    // Load OpenCL kernel
    cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE, &err);
    cl::Kernel krnl_affine_transform(program, "affinetrans", &err);
    ...
    return 0;
}
```

Listing 8: The shortened boilerplate Xilinx Vitis code.

The declaration and initialization of the host vectors is the same in the Vitis implementation as it was in the VexCL implementation, but copying the data to the accelerator does change, as shown in listing 9. Instead of creating `vex::vector` objects, we need to create `cl::Buffer` objects and specify that we want to copy the data from a host pointer and if the data will be readable, writable or both.

We can now remove the VexCL vector computations, as that is already done in the FPGA kernel. We do however need to call the kernel ourselves in the Vitis implementation. To do this, we first need to set the arguments of the kernel. In section 3.3 we already specified which values corresponded to the arguments, so we can simply set the arguments to those values. In listing 10 we set the arguments to the kernel and execute the kernel. We however first need to transfer the input buffers to the kernel and after the kernel is finished, we can transfer the output buffer back to the host device.

```
std::vector<double> a(m * n), x(n), y(m), t(m);

// Initialize matrix + vectors
...

// Create the buffers and allocate memory
cl::Buffer A(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
             sizeof(double) * a.size(), a.data(), &err);
cl::Buffer X(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
             sizeof(double) * x.size(), x.data(), &err);
cl::Buffer Y(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
             sizeof(double) * y.size(), y.data(), &err);
cl::Buffer T(context, CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
             sizeof(double) * t.size(), t.data(), &err);
```

Listing 9: The data initialization in Xilinx Vitis.

```
// Set kernel arguments
krnl_affine_transform.setArg(0, (int) m);
krnl_affine_transform.setArg(1, T);
krnl_affine_transform.setArg(2, Y);
krnl_affine_transform.setArg(3, A);
krnl_affine_transform.setArg(4, X);
krnl_affine_transform.setArg(5, 1);
krnl_affine_transform.setArg(6, 0);
krnl_affine_transform.setArg(7, 1);
krnl_affine_transform.setArg(8, (int) n);
krnl_affine_transform.setArg(9, 0);
krnl_affine_transform.setArg(10, (int) m);
krnl_affine_transform.setArg(11, (int) n);
krnl_affine_transform.setArg(12, (int) n);
krnl_affine_transform.setArg(13, 1);

// Schedule transfer of inputs to device memory, execution of kernel,
// and transfer of outputs back to host memory
q.enqueueMigrateMemObjects({Y, A, X}, 0);
q.enqueueTask(krnl_affine_transform);
q.enqueueMigrateMemObjects({T}, CL_MIGRATE_MEM_OBJECT_HOST);

// Wait for all scheduled operations to finish
q.finish();
```

Listing 10: Calling a kernel in Xilinx Vitis.

Target	Compilation time (hours)	Execution time (minutes)
Software emulation	00:00:51	00:01.244
Hardware emulation	00:06:38	08:25.781
Hardware execution	01:31:11	00:01.376

Table 3.2: Compilation and execution time of the Vitis kernel with different targets using a 12-core Intel Xeon Gold 6128 GPU with 187 GiB of RAM capacity and a Xilinx Alveo U250 FPGA. A random 256×256 matrix and three random vectors with a length of 256 are used as input data to measure the execution time.

3.5 Verifying correctness

```
template <typename Vec>
std::vector<double> calculate_results(size_t m, size_t n, Vec A, Vec x,
                                      Vec y) {
    auto res = std::vector<double>(m * n);

    for (size_t i = 0; i < m; ++i) {
        double sum = 0;

        // Calculate matrix multiplication of current row
        for (size_t j = 0; j < n; ++j) {
            sum += A[i * n + j] * x[j];
        }

        // Store results
        res[i] = y[i] + sum;
    }

    return res;
}
```

Listing 11: A sequential implementation of the affine transformation.

To test both the VexCL and the Xilinx Vitis implementation, listing 11 presents a sequential version of the algorithm, to be used as reference implementation. We check if the output of the accelerated version matches the output of this sequential version. To account for floating point errors, we check if $|(acc[i] - ref[i])/ref[i]| < 0.01$ holds for each element in the output vectors of the accelerated implementation (acc) and the reference implementation (ref).

To be able to use the same input data for both the VexCL implementation and Vitis implementation, we present a JavaScript Object Notation (JSON) format that describes the size and data of the matrix and vectors, as can be seen in listing 12. This file can then be parsed by both implementations to read the same input data, and we can now change the input data without recompiling the program.

3.6 Compiling and running the ported application

3.6.1 Compilation and execution time

As described in chapter 2, there are three compilation targets for the Vitis kernel, software emulation, hardware emulation and hardware execution. To compile the kernel, we must first

```
{
    "A": {
        "size": [2, 2],
        "data": [[0, 1], [2, 3]]
    },
    "x": {
        "size": [2],
        "data": [4, 5]
    },
    "y": {
        "size": [2],
        "data": [6, 7]
    }
}
```

Listing 12: JSON format to describe input data.

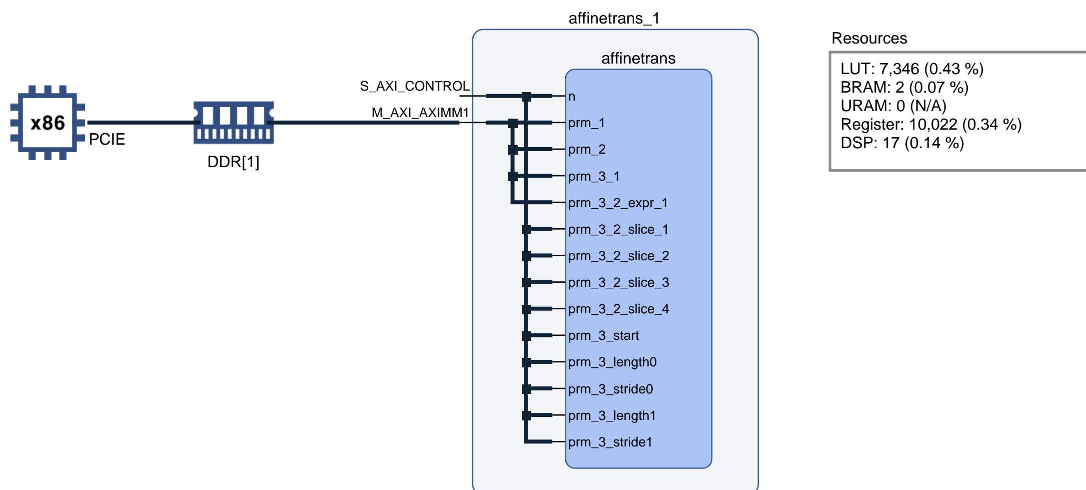
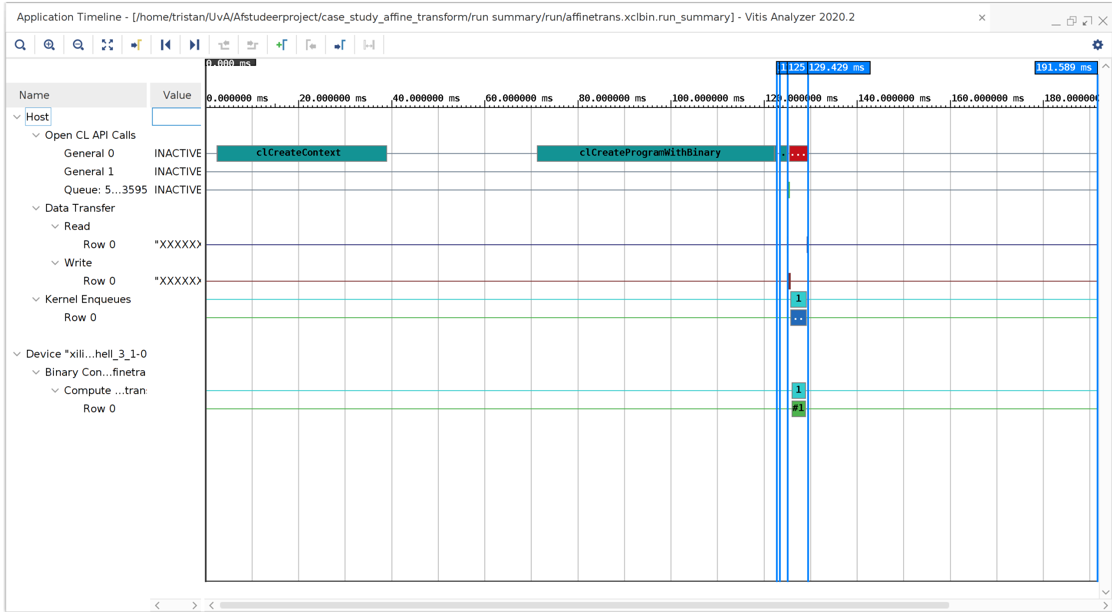


Figure 3.1: System diagram of the affine transformation implementation on Xilinx Vitis FPGA.

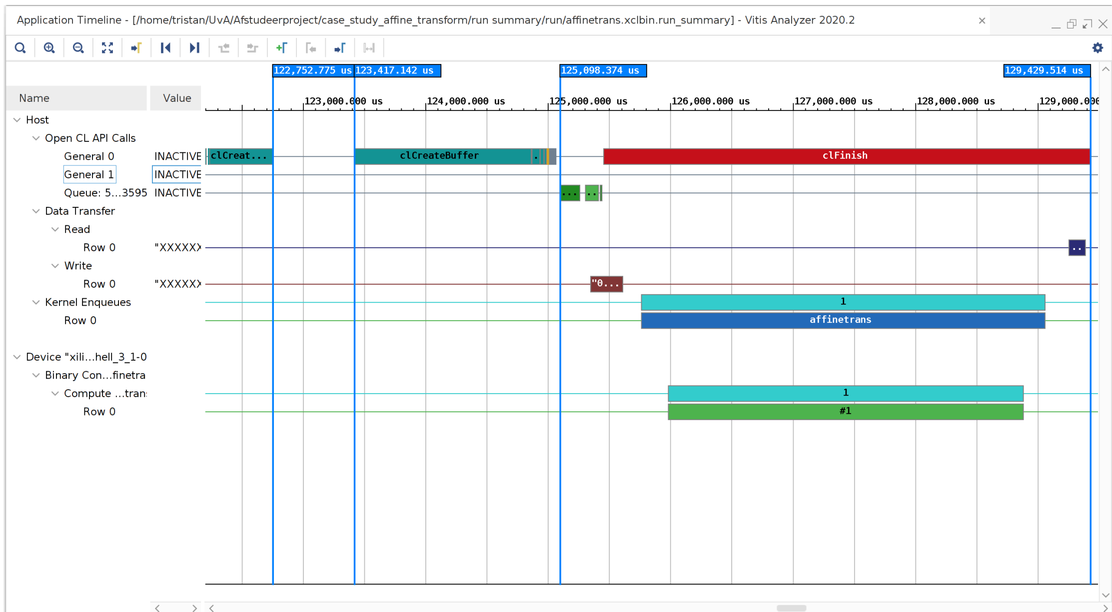
run the HLS compiler to turn the kernel into a Xilinx object (XO) file, and then run the linker of the Vitis compiler to turn the XO file into a binary for the FPGA device. In table 3.2 we show the time it takes to compile the kernel for these three targets using a 12-core Intel Xeon Gold 6128 central processing unit (CPU), and how long it takes each version to compute the affine transformation of a random 256×256 matrix and three random vectors with a length of 256. We can see that compiling for hardware execution indeed takes much longer than compiling for emulation. We can also see that running the hardware emulation is very slow in comparison to software emulation and hardware execution.

3.6.2 Compiled FPGA kernel

Once we compile the kernel, we can let the Vitis Analyser produce a system diagram. This diagram can be seen in figure 3.1. We can see that all the buffers are connected to the DDR memory interface, and that the other parameters are directly fed by a control bus. We can also see that our design uses 7,356 LUTs, 2 BRAMs, 10,022 registers and 17 DSP slices.



(a)



(b)

Figure 3.2: Application timeline of ported Vitis application to calculate an affine transformation, showing the OpenCL calls, data transfers and kernel executions. A random 256×256 matrix and three random vectors with a length of 256 are used as input data for the application. Sub-figure (a) shows the full timeline, while sub-figure (b) shows the timeline zoomed in on the kernel execution and data transfers.

Task	Time	Percentage of total time
Loading kernel to FPGA	122.8 ms	64.0%
Generating input data	0.7 ms	0.4%
Memory buffer creation and setting kernel args	1.6 ms	0.8%
Data transfers and kernel execution	4.4 ms	2.3%
Verifying results	62.2 ms	32.5%
Full application	191.6 ms	100.0%

Table 3.3: Analysis of the application timeline in figure 3.2, showing how long each part of the ported affine transformation application approximately takes. A random 256×256 matrix and three random vectors with a length of 256 are used as input data for the application.

VexCL	Xilinx Vitis

// Entire kernel	#include <stddef.h>
	extern "C" {
	// Entire kernel
	}

kernel void vexcl_vector_kernel(...) {...}	void *kernel_function_name*(...) {...}
ulong	size_t
idx = get_global_id(0)	idx = 0
idx += get_global_size(0)	++idx
global *type* **prm*	const *type* **prm* // if *prm* is input buffer *type* **prm* // if *prm* is output buffer ... #pragma HLS INTERFACE m_axi port=*prm* bundle=aximm<*memory interface No.*>

Table 3.4: Kernel code changes from VexCL to Xilinx Vitis.

3.6.3 Application timeline

Using the Xilinx profiler we can create an application timeline with information about OpenCL API calls, data transfers and kernel execution time from a run of our application. In figure 3.2 we show a timeline of our ported Vitis application that is run using a randomly generated 256×256 matrix and three randomly generated vectors of size 256. In table 3.3, we show how long each part of the application takes, as derived from the timeline. We can see that most time is spend loading the kernel to the FPGA, and verifying the results. That verifying the results using a sequential implementation of the algorithm is slow compared to computing the results on the FPGA is expected and shows that our implementation is faster than the sequential version. That loading the kernel takes the longest time in this case is also expected, since the input data is very small and is thus easy to compute. The time it takes to load the kernel to the FPGA is constant, while the time it takes to do the other tasks is dependent on the size of the input.

3.7 Porting guidelines

The steps needed to convert the VexCL code to Vitis code are shown in table 3.5 for the host code, and in table 3.4 for the kernel. Users can use these tables to match patterns in the VexCL and OpenCL code, and then replace them with the Xilinx Vitis counterpart. This simplifies the process of porting a VexCL application to Xilinx Vitis. The tables can also serve as a basis to describe which tasks are necessary to automate the process of converting a VexCL application to Xilinx Vitis.

VexCL	**Xilinx Vitis**
ctx(vex::Filter::DoublePrecision);	device = get_xilinx_devices().front(); . . . cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE, &err); cl::Kernel *krnl_name*(program, *kernel_function_name*, &err);

vex::vector<cl_type> *A*(ctx, *a*.size(),
a.data());
 	
vex::vector<cl_type> *T*(ctx, *t*.size());
 	
...
vex::copy(*T*.begin(), *T*.end(), *t*.data());
	cl::Buffer *A*(context, CL_MEM_READ_ONLY |
	CL_MEM_USE_HOST_PTR, sizeof(type) * *a*.size(), *a*.data(),
	&err);
	cl::Buffer *T*(context, CL_MEM_WRITE_ONLY |
	CL_MEM_USE_HOST_PTR, sizeof(type) * *t*.size(), *t*.data(),
	&err);

$T = X + Y$;	*krnl_name*.setArg(0, *output_vector_size*);
	krnl_name.setArg(1, *T*);
	krnl_name.setArg(2, *X*);
	krnl_name.setArg(3, *Y*);

reshape(*X*, extents[*m*][*n*], extents[1])	*krnl_name*.setArg(0, *output_vector_size*);
	krnl_name.setArg(1, *X*);
	krnl_name.setArg(2, 1); // skip indices
	krnl_name.setArg(3, 0); // offset
	krnl_name.setArg(4, 1); // repetitions
	krnl_name.setArg(5, *n*); // modulo

reduce<OP>(extents[*m*][*n*], *X*, 1)	*krnl_name*.setArg(0, *output_vector_size*);
	krnl_name.setArg(1, *X*);
	krnl_name.setArg(2, 0); // offset
	krnl_name.setArg(3, *m*); // first dimension matrix
	krnl_name.setArg(4, *n*); // second dimension matrix
	krnl_name.setArg(5, *n*); // amount of values to reduce each time
	krnl_name.setArg(6, 1); // distance between values

// Finished computations	q.enqueueMigrateMemObjects(*input_buffers*, 0);
	q.enqueueTask(*krnl_name*);
	q.enqueueMigrateMemObjects(*output_buffers*,
	CL_MIGRATE_MEM_OBJECT_HOST);
	q.finish();

Table 3.5: Host code changes from VexCL to Xilinx Vitis.

CHAPTER 4

Improving the performance of a ported application

In chapter 3 we provide a porting process to create a Xilinx Vitis application based on a VexCL application. In this chapter we extend this process with several optimizations that can be applied on a newly-ported Vitis application.

4.1 Optimizations

4.1.1 Memory alignment

The GNU memory allocator, used by the compiler of the host code, aligns the memory by eight or sixteen bytes depending on the specific system [8], but Xilinx FPGAs align memory to 4000 bytes internally. So if the host code allocates buffers to transfer data to the FPGA using the default allocator, the program will internally have to copy the data to a new buffer that is aligned to 4000 bytes [21]. To prevent this from happening, we have to use a special allocator that aligns the data to 4000 bytes from the start. This can be done using the `posix_memalign` function, which allows the user to specify the memory alignment to use.

4.1.2 Fixed-point arithmetic

On GPUs it is common to use floating-point numbers to represent decimal numbers. In this representation one bit is reserved to represent a sign (s), a fixed amount of bits is reserved to represent an exponent (e) and a fixed amount of bits is reserved to represent a fraction (d), the number is then represented as $(-1)^s \times 2^e \times (1.d)_2$. Xilinx FPGAs also support floating-point numbers, but in contrast to GPUs, Xilinx FPGAs additionally support a fixed-point representation for decimal numbers. This representation uses the format $n + \frac{1}{f}$, where n and f are numbers of a fixed amount of bits.

The biggest drawback of using fixed-point numbers is that while the accuracy of fixed-point numbers is fixed, no matter the size of the number, the limits of the numbers it can represent are lower than a floating-point number of the same amount of bits. So if both very large or very small numbers have to be represented, floating-point numbers are a better choice.

An advantage of fixed-point numbers is that fixed-point arithmetic is more efficient than floating-point arithmetic on Xilinx FPGAs. Finnerty and Ratigner show that a fixed-point number implementation of their algorithm, in comparison to a floating-point number implementation, has more than seven times lower latency, more than 80% lower power requirements and requires eleven times less LUTs in the logic circuit on a Xilinx FPGA [4].

Another Xilinx FPGAs implementation specific advantage is that while floating-point numbers can only be used with single- or double-precision on the FPGA, Xilinx allows user to specify how many bits are used to store the numbers n and f of the fixed-point number.

4.1.3 Burst transfers

To get data from the DDR memory interface, the FPGA has to make a read request, which causes a read latency, similarly after writing data the FPGA has to wait for a write acknowledgment, which causes a write latency. To reduce the amount of time waiting for the total amount of read requests and write acknowledgments, we can increase the amount of data we request or send in one read or write request. This is called bursting. Bursting can give up to a four to five times performance improvement [19].

4.1.4 Saturating data width

The DDR memory interface of the FPGA support a maximum data width of 512 bits, but by default the data width will be matched by the size of the data type. So if you use an `int` as data type the data width will be 32 bits, and if you use a `double` as data type the width will be 64 bits. To improve the maximum data transfer rate, it is possible to widen the data width and allow more data to be transferred at the same time. We can for example store sixteen 32-bit integers in a single 512-bit integer and retrieve the original sixteen 32-bit integers using bit shifts.

4.2 Applying the optimizations

4.2.1 Memory alignment

The memory alignment optimization can be easily integrated into the vectors we use, because the `std::vector` class has support for a custom allocator. In listing 13 we show this custom allocator, which makes use of the `posix_memalign` function mentioned before. To use this custom allocator we replace `std::vector` calls in the host code that was in the form of `std::vector<T> x(m);`, from the porting process in chapter 3, to `std::vector<T, aligned_allocator<T>> x(m);`.

```
1  template <typename T>
2  struct aligned_allocator
3  {
4    using value_type = T;
5    T* allocate(std::size_t num)
6    {
7      void* ptr = nullptr;
8      if (posix_memalign(&ptr,4096,num*sizeof(T)))
9        throw std::bad_alloc();
10     return reinterpret_cast<T*>(ptr);
11   }
12   void deallocate(T* p, std::size_t num)
13   {
14     free(p);
15   }
16 };
```

Listing 13: A custom aligned allocator that can be used by `std::vector`.

4.2.2 Fixed-point arithmetic

Xilinx provides fixed-point number support through the `ap_fixed` and `ap_ufixed` data type for signed and unsigned numbers respectively. We can create our own custom fixed-point by declaring it using `typedef ap_fixed<`W`, `I`, `Q`, `O`, `N`> fix_t;`.

W specifies the total size of the number in bits, and I the number of bits used for the non-fraction part of the number. The amount of bits used for the fraction part of the number is automatically calculated by $W - I$. To find the correct number for W and I, we have to determine the range of numbers we want to represent. If we for example want to represent numbers ranging from -250 to 240 with a precision of at least 0.01. then we need to reserve at least $\lceil \log_2(\max(250, 240)) \rceil + 1 = 9$ bits for the non-fraction part of the number. Note that we had to add one for the sign, and choose the maximum between 250 and 240 because the amount of numbers that can be represented is symmetric around zero. For the fraction part of the number we need at least $\lceil \log_2(\frac{1}{0.01}) \rceil = 7$ bits. So in this case we would declare $W = 9 + 7 = 16$ and $I = 9$.

Q represent the quantization mode and describes how numbers should be rounded if the fraction is too small for the fixed-point number. Here you can choose modes to round or truncate to $\pm\infty$ or zero, by default it rounds to $+\infty$.

Lastly, O and N define what happens in case of an overflow. O specifies the overflow mode, by default this is to wrap around like the C `int` data type, but it is also possible to use saturation, which means that numbers that are too large to be represented will be set to a specific value. For this specific value it is possible to choose the closest number to the represented number or zero. N specifies the amount of saturation bits that are used in the wrap around mode, and specify how many bits will be copied when wrapping around. If $N = 1$ for example, positive numbers will stay positive and negative numbers will stay negative. In contrast, if $N = 0$ than positive numbers will wrap around to negative values and vice versa, a graphical example of this behaviour can be seen in figure 4.1. By default $N = 0$.

Once we have specified our data type, we can simply replace the original `double` declarations from our porting process in chapter 3 with matching `fix_t` declarations.

4.2.3 Burst transfers

Typically our kernels have a structure similar to algorithm 1, first we read the data, then we do some computation and lastly we store the results. Sometimes we do this in one step, i.e. $out[i] \leftarrow Compute(A[i], B[i])$, but then we have to first split it up in multiple steps before we can apply this optimization.

1: **for** $i \leftarrow 0$ **to** n **step** 1 **do**
2: $\quad a \leftarrow A[i]$
3: $\quad b \leftarrow B[i]$
4: $\quad c \leftarrow Compute(a, b)$
5: $\quad out[i] \leftarrow c$
6: **end for**

Algorithm 1: A typical kernel structure.

To make use of burst transfers in algorithm 1, we increase the step size of the for-loop, and add new for-loops around the read part, compute part and write part. If we add `max_read_burst_length` and `max_write_burst_length` to the memory pragmas we specified in chapter 3, Xilinx can automatically use burst transfers for the new small for-loops. In algorithm 2 we show the new optimized kernel, which uses a burst size of 16. Note that the temporary variables have become temporary arrays, and that we have to add a boundary check for the case that n is not divisible by $burst$.

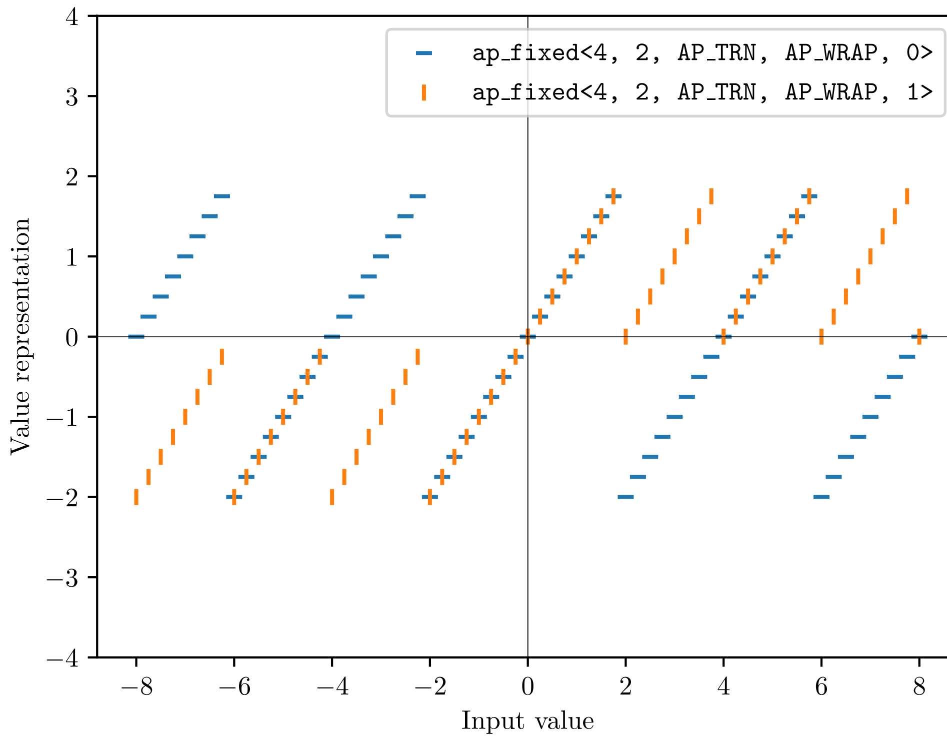


Figure 4.1: Graphical representation of the difference between the amount of saturation bits in fixed-point numbers with wrap around overflow.

1: $burst \leftarrow 16$
2: **for** $i \leftarrow 0$ **to** n **step** $burst$ **do**
3: $\quad chunk \leftarrow burst$
4: $\quad$ **if** $(i + burst) > n$ **then** ▷ Boundary check
5: $\quad\quad chunk \leftarrow n - i$
6: $\quad$ **end if**

7: $\quad$ **for** $j \leftarrow 0$ **to** $chunk$ **step** 1 **do**
8: $\quad\quad a[j] \leftarrow A[i + j]$
9: $\quad\quad b[j] \leftarrow B[i + j]$
10: $\quad$ **end for**

11: $\quad$ **for** $j \leftarrow 0$ **to** $chunk$ **step** 1 **do**
12: $\quad\quad c[j] \leftarrow Compute(a[j], b[j])$
13: $\quad$ **end for**

14: $\quad$ **for** $j \leftarrow 0$ **to** $chunk$ **step** 1 **do**
15: $\quad\quad out[i + j] \leftarrow c[j]$
16: $\quad$ **end for**
17: **end for**

Algorithm 2: A typical kernel structure with burst transfers of 16 elements.

4.2.4 Saturating data width

To take advantage of saturated data widths, we have to make sure that our kernel can take advantage of being able to read multiple data elements at once, i.e. we have to make sure that our algorithm already makes use of burst transfers. So to show how to implement this optimization, we will start with the kernel specified in algorithm 2. If we assume that A, B and out are arrays of doubles, then the current data width will be 64-bits. To increase the data width we can create a `struct` that consists of multiple doubles, then the `struct` has a larger data width than 64-bits. If we want to achieve a data width of 512-bits, we can create a `struct` that consists of $\frac{512}{64} = 8$ doubles, as shown in listing 14. Now we can use this new data type in our algorithm, as can be seen in algorithm 3. We add a new $width$ variable to denote the amount of elements in the struct, and use bit shifts and moduli to retrieve the original data. Note that we do not need to change the host code, as the `struct` data is still in the same location when using the structs.

```
1  typedef struct {
2      double data[8];
3  } double_v;
```

Listing 14: Example `struct` consisting of eight doubles to saturate the data width.

1: $burst \leftarrow 16$
2: $width \leftarrow 8$
3: $logwidth \leftarrow 3$
4: **for** $i \leftarrow 0$ **to** n **step** $burst$ **do**
5: $\quad chunk \leftarrow burst$
6: $\quad$**if** $(i + burst) > n$ **then** ▷ Boundary check
7: $\quad\quad chunk \leftarrow n - i$
8: $\quad$**end if**

9: $\quad$**for** $j \leftarrow 0$ **to** $chunk$ **step** 1 **do**
10: $\quad\quad k \leftarrow i + j$
11: $\quad\quad a[j] \leftarrow A[k >> logwidth].data[k \mod width]$
12: $\quad\quad b[j] \leftarrow B[k >> logwidth].data[k \mod width]$
13: $\quad$**end for**

14: $\quad$**for** $j \leftarrow 0$ **to** $chunk$ **step** 1 **do**
15: $\quad\quad c[j] \leftarrow Compute(a[j], b[j])$
16: $\quad$**end for**

17: $\quad$**for** $j \leftarrow 0$ **to** $chunk$ **step** 1 **do**
18: $\quad\quad k \leftarrow i + j$
19: $\quad\quad out[k >> logwidth].data[k \mod width] \leftarrow c[j]$
20: $\quad$**end for**
21: **end for**

Algorithm 3: Typical kernel structure with burst transfers and higher data width.

4.3 Testing the optimizations

To see how the optimizations perform, we will apply them to the ported affine transformation application as described in chapter 3, and compare their performance and power usage by running

them with different input sizes and using different profiling tools to extract the execution time and power usage.

4.3.1 Implementations

To test the optimizations we propose five optimized versions of the affine transformation application from chapter 3 and compare them to the unoptimized vitis port and the original VexCL application from chapter 3.

Common optimizations

The first optimization we propose, optimization 1, applies the memory alignment strategy described in subsection 4.2.1 to the unoptimized version of the affine transformation application. This optimization also inlines the unnecessary `SUM_DOUBLE` function. Lastly this optimization rearranges the memory interfaces, t keeps using the first interface, y and A will use the second interface and x will use the third interface. This rearrangement makes sure that data that can be read at the same time does not use the same interface. We use this optimization to test how much influence applying common optimizations have on the performance of the application.

Fixed-point arithmetic optimization

The second optimization, optimization 2, builds upon optimization 1 by replacing the double-precision floating-point data types with fixed-point data types as described in subsection 4.2.2. We specify two new data types, `fix_t` = `ap_fixed<18, 7, AP_RND, AP_SAT>` and `lfix_t` = `ap_fixed<64, 54, AP_RND, AP_SAT>`. The `fix_t` data type can represent numbers from -64 to 64 with a precision of $\frac{1}{2048} \approx 3.88 \cdot 10^{-4}$ and the `lfix_t` data type can represent numbers from -2^{53} to 2^{53} with the same precision as `fix_t`. We will replace the data types of the input buffers A, x and y with the `fix_t` data type, and replace the data types of the output buffer t and internal variable `prm_3_sum` with the `lfix_t` data type. We use this optimization to test if fixed-point numbers indeed have higher performance than floating-point numbers on Xilinx FPGAs.

Burst transfer optimization

Optimization 3 applies the burst transfer strategy, as described in subsection 4.2.3, to optimization 2. We choose a burst size of 256, because the memory bus supports bursts of up to 16 kib. We include this optimization to test if burst transfers have an impact on the performance of the application.

Data width saturation optimization

Optimization 4 adapts optimization 3 to saturate the data width of the burst transfers, as described in subsection 4.2.4. To accomplish this, we present two new data types, `fix_v` = `struct {fix_t data[16]}` and `lfix_v` = `struct {lfix_t data[8]}`. We change the data types of A, x and y from `fix_t` to `fix_v` and the data type of t from `lfix_t` to `lfix_v`. We add this optimization to see if data width saturation improves the performance of burst transfers in the application.

Smaller fixed-point numbers optimization

The last optimization, optimization 5, is the same as optimization 4, but with a small change to the `lfix_t` data type. `lfix_t` = `ap_fixed<64, 54, AP_RND, AP_SAT>` is redefined to `lfix_t` = `ap_fixed<44, 33, AP_RND, AP_SAT>`, so the precision stays the same, but the data type can now only represent numbers from -2^{32} to 2^{32}, instead of numbers from -2^{53} to 2^{53}. We include this optimization to see how much impact the size of the fixed-point numbers has on the performance of the application.

4.3.2 Input sizes

As described in chapter 3, the application calculates an affine transformation defined as $\vec{t} = \vec{y} + A\vec{x}$, so if the matrix A is a $m \times n$ matrix, then $\vec{t}$ and $\vec{y}$ should be of size m and $\vec{x}$ should be of size n. So to describe the input sizes, we only have to define the size of matrix A, and the sizes of the vector can be derived from the size of the matrix. We will test the application with matrix sizes 32×32, 256×256, 1024×1024, 5000×5000 and $10^4 \times 10^4$ to see how the applications perform as the size increases. Note that we moved away from powers of 2 with the two largest matrices, because this might be a disadvantage for optimization 3, 4 and 5, as they all have burst sizes of 256 and both 5000×5000 and $10^4 \times 10^4$ are not divisible by 256. We also test matrix sizes of 2000×2000, $2 \times 2 \cdot 10^6$ and $2 \cdot 10^6 \times 2$ to see how good the applications perform when the size of the matrix is unbalanced, compared to a square matrix.

4.3.3 Hardware

All the testing will be done on a machine with a 12-core Intel Xeon Gold 6128 CPU, 187 GiB of RAM capacity, a NVIDIA RTX 2080Ti GPU and a Xilinx Alveo U250 FPGA.

4.3.4 Profiling tools

In all implementation we measure the time it takes to execute the application, excluding the input data initialization and correctness verification, which we refer to as the *wall time* for simplicity. To measure the wall time, we use the steady clock from the C++ standard library. On the FPGA implementations we also use the Xilinx profiler to measure the kernel execution time, and the power usage of the FPGA. For the VexCL implementation we only measure the power usage by running the NVIDIA System Management Interface (nvidia-smi) program[1] in the background, which can measure the power draw of a NVIDIA GPU, while running the application.

Both the Xilinx profiler and the nvidia-smi program measure the power usage over time, this will generally stay the same while running the application. However because we run the nvidia-smi program in the background, the profiler of the NVIDIA GPU will start a little earlier than the program we want to measure, so to account for this, we drop values that are smaller or equal to 1 watt.

4.3.5 Method

To compare all the implementations, we first generate input data of different sizes, as described in subsection 4.3.2. We then run all the different implementations with the same input data and the profiling tools active to retrieve the measurement data. We repeat this experiment ten times, with different input data each run. We then calculate the average and standard deviation of the wall time, kernel time and power usage from all ten runs. Note that the power usage will be averaged over more than ten values, because the power usage is measured multiple times during each run. We can also approximate the total energy usage from the wall time and power usage, by calculating $E = P \cdot t$, where E is the approximate total energy usage in watt-hours (Wh), P is the average power usage in watts (W) and t is the execution time in hours (h).

The input data of the larger matrices can be very large, the largest matrix has a size of $\frac{10^4 \cdot 10^4 \cdot 8}{10^6} =$ 800 MB, if it is efficiently encoded, if we use the JSON format as described in chapter 3, it will be approximately 2 GB. To be able to load this data efficiently into the application we need a faster solution than our JSON implementation, so we use a Python script to generate C code with the input data hard-coded into arrays, and then compile this C code to a shared object that is linked to all the implementations.

4.3.6 Results

All the raw averaged data from the results of the experiment can be found in appendix A.

[1] nvidia-smi – `https://developer.nvidia.com/nvidia-system-management-interface`

Version	LUTs	BRAMs	Registers	DSP slices
Unoptimized	7,346 (0.43%)	2 (0.07%)	10,022 (0.34%)	17 (0.14%)
Optimization 1	8,074 (0.47%)	3 (0.11%)	11,138 (0.38%)	17 (0.14%)
Optimization 2	7,285 (0.42%)	2 (0.07%)	9,317 (0.32%)	7 (0.06%)
Optimization 3	8,921 (0.52%)	17 (0.63%)	10,344 (0.35%)	7 (0.06%)
Optimization 4	11,152 (0.62%)	26 (0.97%)	19,116 (0.65%)	7 (0.06%)
Optimization 5	8,734 (0.51%)	17 (0.63%)	10,202 (0.35%)	7 (0.06%)

Table 4.1: Resource utilization of Xilinx Vitis kernels compiled for a Xilinx Alveo U250 accelerator card.

Resource utilization

In table 4.1 we show the resource utilization of the compiled kernels on the FPGA. We can see that optimization 1 uses more resources than the unoptimized kernel. We can also see that optimization 2 uses less resources than both optimization 1 and the unoptimized version. Optimization 3 has a large increase in resources compared to the previous versions, especially in BRAM usage. Optimization 4 uses even more resources, and uses the most resources of all implementations. Lastly optimization 5 uses less resources than the version it was based off, optimization 3.

Wall time

In figure 4.2 we show the average wall time of each implementation per dimension.

We can see that the VexCL implementation is generally faster, especially as the matrix grows bigger, except for the wide $2\times2\cdot10^6$ matrix.

Optimization 1 is always faster than the unoptimized Vitis implementation, generally around 25%, except for the tall matrix, there optimization 1 is only about 5% faster than the unoptimized version.

Optimization 2 is mostly between 5% and 20% faster than optimization 1, except for the 1024×1024 matrix, where it is 3% slower.

Looking at optimization 3, we can see that, while it is 18% faster than optimization 2 for the 32×32 matrix and 8% faster for the 256×256 matrix, it gets slower as the matrices grow. Optimization 3 is 3% slower than optimization 2 for the 1024×1024 matrix and 17% slower for the $10^4\times10^4$ matrix.

Optimization 4 is almost always between 25% and 30% slower than optimization 3, except for the tall $2\cdot10^6\times2$ matrix, where it is 14% faster than optimization 3.

Optimization 5 is only slower than optimization 3 with the 32×32 matrix, where it is 20% slower. With the tall $2\cdot10^6\times2$ matrix Optimization 5 is about as fast as optimization 3, and with the other matrices it is between 7% and 9% faster.

Power usage

In figure 4.3 we show the average power usage of each implementation per dimension.

We can see that the VexCL version, running on the GPU, always uses more power than the Vitis versions, running on the FPGA. Generally the VexCL version uses about 50 W of power, which is between 50% and 75% more power than the unoptimized Vitis version uses. Only with the wide $2\times2\cdot10^6$ matrix it uses more power, at the start of the computation it uses about 50 W of power, but at the end of the computation it uses about 80 W of power. On average it uses 69 W of power for the wide matrix.

The power consumption of the FPGA is very close for all Vitis versions, between 27 W and 32 W. Optimizations 2, 3, 4 and 5 all use about the same amount of power, 32 W. Optimization 1 generally uses a little less power, between 28 and 30 W of power, except for the largest two

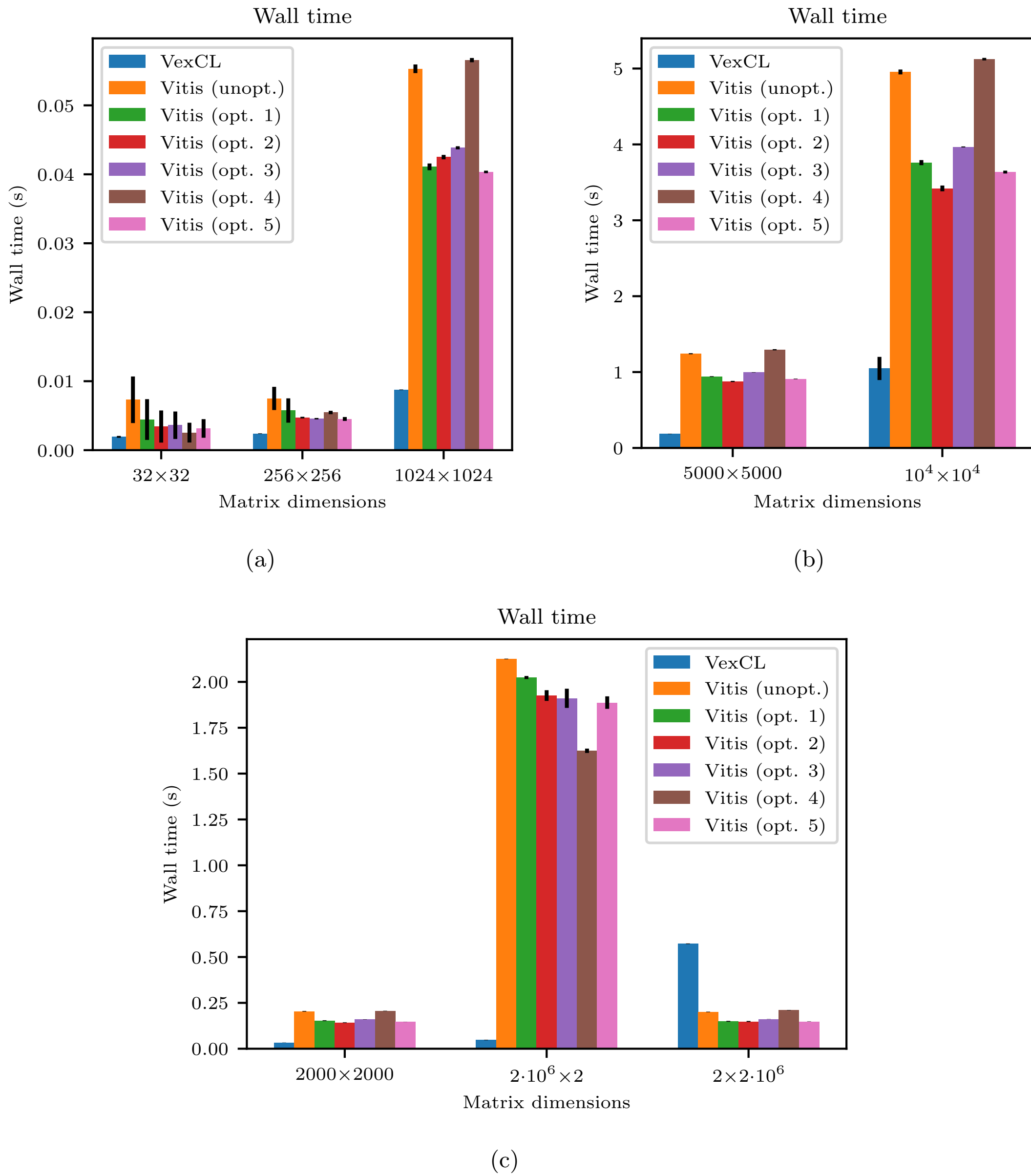


Figure 4.2: The wall time of the different affine transformation implementations with different matrix configurations, averaged over ten runs.

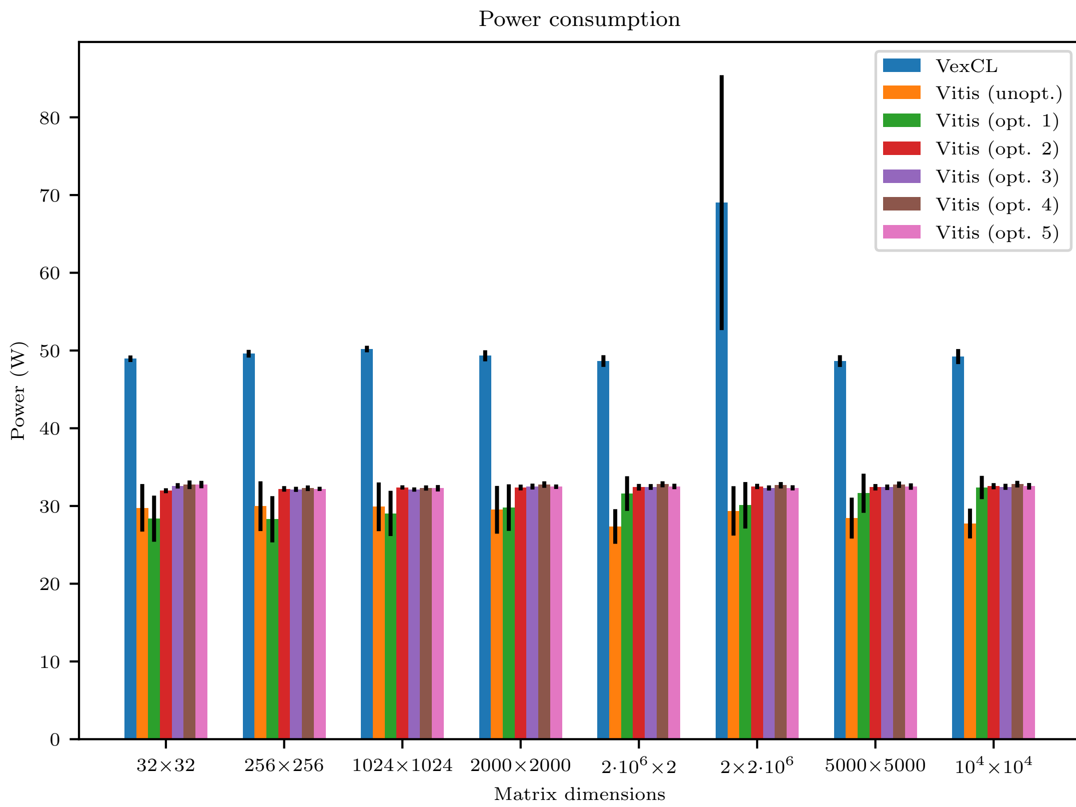


Figure 4.3: The power usage of the different affine transformation implementations with different matrix configurations, averaged over ten runs.

matrices and the wide matrix, where it also uses about 32 W of power. The unoptimized version always uses between 27 W and 28 W of power.

Approximate energy usage

In figure 4.4 we show the approximate energy usage of each implementation per dimension. We can see that even though the VexCL implementation uses more power, it still generally has a lower energy usage, because it has a shorter execution time. Only for the wide $2{\times}2{\cdot}10^6$ matrix the VexCL version has a higher energy usage than the Vitis versions, because of the higher execution time. Because the power usage of the Vitis implementations was very close, the differences in energy usage between the implementation is about the same as for the execution time.

4.3.7 Discussion

Looking at the resource utilization of optimization 4, we could have predicted that this optimization would be slower than the other optimization due to the high BRAM usage. A probable cause for this is that the Xilinx Vitis compiler can not handle the `structs` we used very efficiently. We can also see that the fixed-point numbers from optimization 2 indeed use less resources than the floating-point numbers from optimization 1, and that the fixed-point number with a smaller range in optimization 5 use less resources than the fixed-point numbers in optimization 3.

The results indicate that the common optimizations help with increasing the performance of the application.

The fixed-point arithmetic used in optimization 2 seems to indeed be faster than the floating-point arithmetic used in optimization 1. It also, however, seems to use more power than the floating-point arithmetic: the power usage of optimization 2 is higher than that of optimization 1. We can also see that fixed-point numbers with a smaller range seem to have higher performance than fixed-point numbers with a larger range, because our results indicate that optimization 5 is generally faster than optimization 3.

The wall-time results comparison between optimizations 2 and 3 indicate that the burst transfers from optimization 3 only improve the performance of the application when the matrix is small, and seem to hurt the performance when the matrix is large. The performance penalty of the burst transfers could be caused by the increased computation complexity, and because the results of the summation for the matrix multiplication must be written to the same variable.

The data width saturation technique likely degrades the performance of the application, as we see that optimization 4 is, in almost all, cases slower than optimization 3. This behaviour was to be expected, given the measured resource utilization.

In general, the Vitis implementations do not reach the performance of the VexCL implementation running on the GPU. Especially as the input size gets larger, the relative difference in execution time between the VexCL version and the Vitis version increases. This indicates that the VexCL version can better parallelize the application on the GPU than the Vitis versions can on the FPGA. Only the very wide matrix performs worse in the VexCL implementation, likely due to the fact that the VexCL version only parallelizes the outer loop, and runs the inner loop sequentially.

4.4 Summary

Our empirical analysis indicates that, even with the *successful* proposed optimizations, our Vitis implementation running on the FPGA is neither faster, nor more energy efficient than the VexCL implementation running on the GPU.

We can also conclude that using aligned memory allocation, inlining small functions, and using more memory interfaces generally improves the performance of the application.

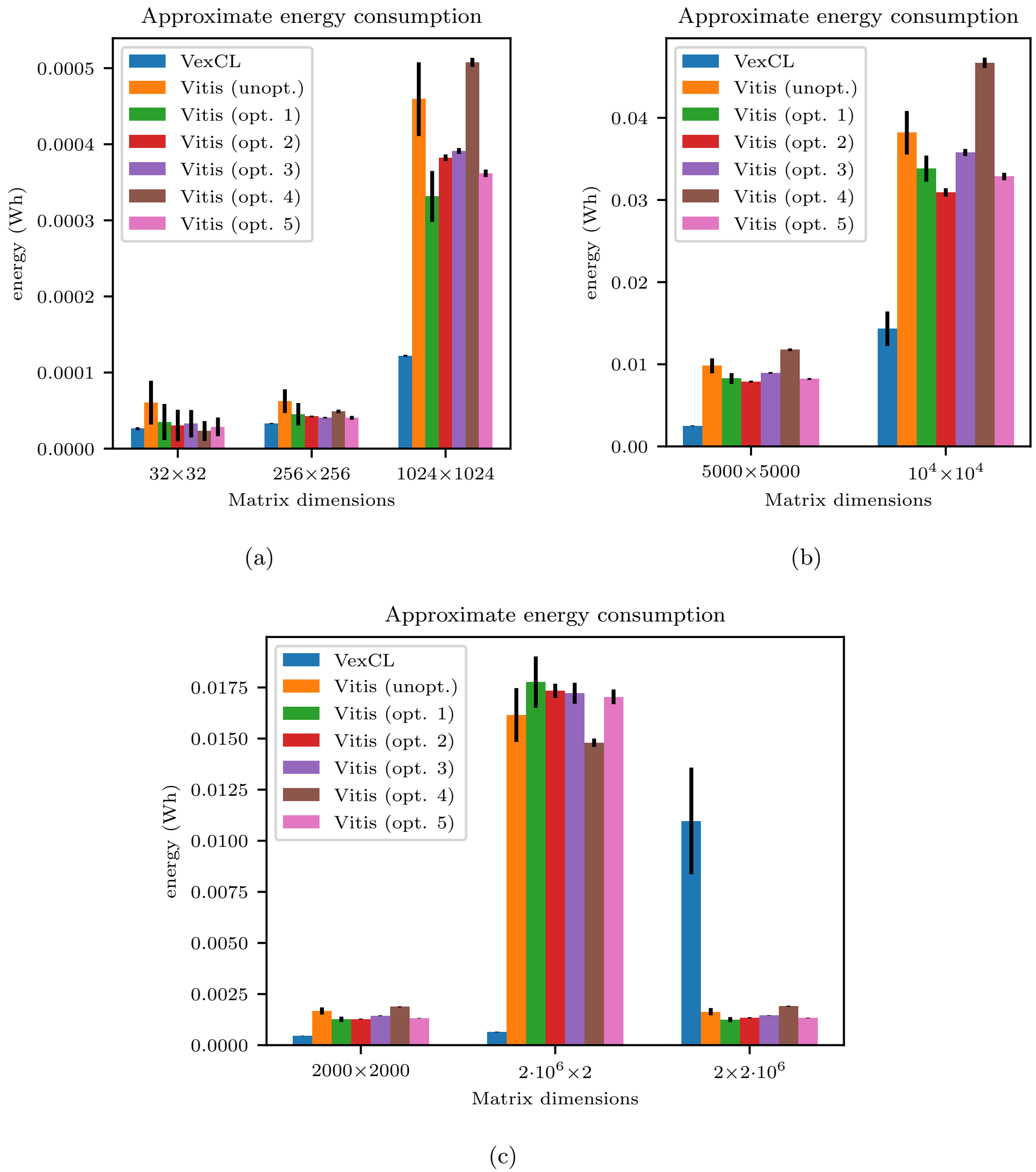


Figure 4.4: The approximate energy usage of the different affine transformation implementations with different matrix configurations, derived from the wall time and power usage results.

If the disadvantages of fixed-point numbers are acceptable for the application, the performance of the application can be improved by replacing floating-point numbers with fixed-point numbers, and to get the most performance out of the fixed-point numbers, the range should be set as small as the application requirements allow.

The burst transfer optimization and the data saturation optimization both generally do not increase the performance of the applications and most of the time decrease the performance of the application, so these optimizations should not be applied in their current form.

CHAPTER 5

The effectiveness of the porting process

In this chapter we apply the porting guide proposed in chapter 3 to a more advanced application, thus testing its effectiveness; we further improve the porting guide, as necessary. The application we use is sparse matrix-vector multiplication (SpMV). SpMV uses sparse matrix representations to more efficiently store large matrices with many zero values, by only storing the non-zero values. Sparse matrices are very common in scientific computation, in applications from graph processing, like PageRank [6], to physics calculations [1].

In this chaper, we specifically look at the calculation defined in equation 5.1 and 5.2, where A is a sparse matrix of size $m \times n$, $\vec{t}$ is a vector of size m, $\vec{u_1}$, $\vec{u_2}$, and $\vec{u_3}$ are vectors of size n, $\odot$ is the element-wise multiplication, and $\oslash$ is the element-wise division.

$$\vec{t} = A\varphi(\vec{u_1}, \vec{u_2}, \vec{u_3}) \tag{5.1}$$

$$\varphi(\vec{u_1}, \vec{u_2}, \vec{u_3}) = \left(\vec{u_1} - \vec{u_2} + \log^2(\vec{u_3}) \odot \sin(\vec{u_1})\right) \oslash (\vec{u_1} \odot \vec{u_2}) \tag{5.2}$$

5.1 Sparse matrix representation

There are several formats that can be used to represent a sparse matrix, all with their own advantages and disadvantages. VexCL only uses the compressed sparse row (CSR) and hybrid ELL–CSR (HELL) formats, so we will focus on those two.

5.1.1 CSR format

The CSR format stores the sparse matrix using three arrays, a row array (R), a column array (C), and a data array (V). V stores all the non-zero values of the array, and has a length of nnz. C has the same length as V, and specifies the column at which each value is located in matrix, i.e. $C[i]$ stores the column of $V[i]$. The row array has a length of $(m + 1)$, where m is the amount of rows, and $(R[i] - R[i-1])$ specifies the amount of elements in the ith row of the matrix. More specifically, $R[i-1]$ specifies at which index the first element of row i starts in the arrays C and V, and $R[i]$ specifies what the last index of row i is in C and V. The arrays V and C must be grouped by row to allow this representation, and to improve cache performance, the arrays should be sorted by row and column. Note that because of this, the first value of R will always be zero, and the last value nnz. In equation 5.3 an example is shown of how the matrix A on the left is stored in the CSR format on the right.

$$A = \begin{pmatrix} 0 & 6 & 9 & 7 \\ 0 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 \end{pmatrix} \qquad \begin{aligned} V &= [6,9,7,3,3] \\ C &= [1,2,3,0,1] \\ R &= [0,3,3,5] \end{aligned} \tag{5.3}$$

5.1.2 ELL format

In the ELLPACK (ELL), two matrices are used to store the sparse matrix: a a data matrix V, which stores the non-zero values, and a column matrix C, which stores the columns of the values in the sparse matrix. Each row of the two matrices represent a row in the sparse matrix, and the entry $C_{i,j}$ specifies in which column the entry $V_{i,j}$ is stored in the sparse matrix. The matrices V and C have the same size, the height is equal to the height of the sparse matrix, and the width is equal to highest amount of non-zero values in a row of the sparse matrix. When not all the rows in the sparse matrix contain the same amount amount of non-zero values, the smaller rows will be padded in the matrices V and C. The ELL format requires more space than the CSR format, but the values can more easily be retrieved. Equation 5.4 shows an example of how the matrix A on the left is stored in the ELL format on the right, with padded values represented as $*$.

$$A = \begin{pmatrix} 0 & 6 & 9 & 7 \\ 0 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 \end{pmatrix} \qquad C = \begin{pmatrix} 1 & 2 & 3 \\ * & * & * \\ 0 & 1 & * \end{pmatrix} \quad V = \begin{pmatrix} 6 & 9 & 7 \\ * & * & * \\ 3 & 3 & * \end{pmatrix} \tag{5.4}$$

5.1.3 Hybrid ELL–CSR format

If a few rows are larger than the rest in the ELL format, the wasted space from padded value increases significantly. To mitigate this, the HELL format stores the bulk of the data using the ELL format, but uses the CSR format for the outliers. An example of this is shown in equation 5.5, where the matrix A on the left is stored using the HELL format on the right, where the first two columns are stored using the ELL format, while the last column is stored using the CSR format.

$$A = \begin{pmatrix} 0 & 6 & 9 & 7 \\ 0 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 \end{pmatrix} \qquad \begin{aligned} &C_{ell} = \begin{pmatrix} 1 & 2 \\ * & * \\ 0 & 1 \end{pmatrix} \quad V_{ell} = \begin{pmatrix} 6 & 9 \\ * & * \\ 3 & 3 \end{pmatrix} \\ &V_{csr} = [7] \\ &C_{csr} = [3] \\ &R_{csr} = [0,1,1,1] \end{aligned} \tag{5.5}$$

5.2 VexCL implementation

The VexCL implementation of SpMVs application is quite straightforward, since VexCL has native support for sparse matrices and SpMVs. The sparse matrices must be initialized in the CSR format, and VexCL will then convert them to the HELL format, which is the format that will be used on the GPU. The SpMVs can simply be calculated using the standard multiplication operator in C++. The φ function can also be easily implemented using the log and sin functions provided by VexCL. We do however use a new VexCL construct in the φ function, the tag function. The `tag` function makes sure that if a vector is used multiple times in the same kernel, VexCL will not use duplicate parameters to send the same vector to the kernel multiple times, instead all the duplicate uses of the vector will use the same tag parameter. A shortened version of the VexCL implementation can be seen in listing 15.

```
template <typename Vec>
Vec phi(U1, U2, U3) {
    auto u1 = vex::tag<1>(U1);
    auto u2 = vex::tag<2>(U2);
    auto u3 = vex::tag<3>(U3);

    return (u1 - u2 + vex::log(u3) * vex::log(u3) * vex::sin(u1)) / (u1 * u2);
}

int main() {
    std::vector<double> A_row(m), A_col(nnz), A_data(nnz), u1(n), u2(n), u3(n);
    std::vector<double> t(m);

    // Initialize matrix + vectors
    ...

    // Transfer host-side doubles into device-side cl_double vectors
    vex::SpMat<cl_double> A(ctx, m, n, A_row.data(), A_col.data(),
        reinterpret_cast<cl_double*>(A_data.data()));
    vex::vector<cl_double> U1(ctx, u1.size(),
        reinterpret_cast<cl_double*>(u1.data()));
    vex::vector<cl_double> U2(ctx, u2.size(),
        reinterpret_cast<cl_double*>(u2.data()));
    vex::vector<cl_double> U3(ctx, u3.size(),
        reinterpret_cast<cl_double*>(u3.data()));
    vex::vector<cl_double> T(ctx, t.size();

    T = A * phi(U1, U2, U3);

    vex::copy(T.begin(), T.end(), reinterpret_cast<cl_double*>(t.data()));

    // Verify results
    ...
}
```

Listing 15: A VexCL implementation of the sparse matrix-vector multiplication calculation.

Parameter	Value
ell_w	height of the ELL matrices
ell_pitch	width of the ELL matrices
ell_col	C_{ell} matrix
ell_val	V_{ell} matrix
csr_row	R_{csr} array
csr_col	C_{csr} array
csr_val	V_{csr} array

Table 5.1: Values of the parameters that correspond to the HELL sparse matrix representation in the third OpenCL kernel generated by VexCL.

5.2.1 OpenCL kernel

When we run the VexCL applicatio to retrieve the OpenCL kernel, we can see that three kernels are generated.

The first kernel, seen in listing 16, simply copies the vector prm_2 to vector prm_1. This kernel is called three times, for each of the calls to the tag function from VexCL. VexCL does this to make a copy of the vector to be used by the tag, so that if the vector passed to the tag function is changed, the data from the tag vector stays the same. Since we do not modify the vectors $\vec{u_1}$, $\vec{u_2}$, and $\vec{u_3}$, these temporary copies of the vectors are unnecessary.

```
1 kernel void vexcl_vector_kernel(ulong n, global double *prm_1,
2   global double *prm_2) {
3   for(ulong idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
4     prm_1[idx] = prm_2[idx];
5   }
6 }
```

Listing 16: The first OpenCL kernel generated by VexCL, which copies a vector.

The second kernel that is generated is the kernel that executes the φ function, and can be seen in listing 17. We can see that, by using the tag function, only three input buffers are passed to the kernel, i.e., the three buffers created by the first kernel. The main body of the code simply calculates the expression as specified in the φ function, using the builtin log and sin functions from OpenCL.

```
1 kernel void vexcl_vector_kernel(ulong n, global double *prm_1,
2   global double *prm_tag_1_1, global double *prm_tag_2_1,
3   global double *prm_tag_3_1) {
4   for(ulong idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
5     prm_1[idx] = (prm_tag_1_1[idx] - prm_tag_2_1[idx] + log(prm_tag_3_1[idx]) \
6       * log(prm_tag_3_1[idx]) * sin(prm_tag_1_1[idx])) / \
7       (prm_tag_1_1[idx] * prm_tag_2_1[idx]);
8   }
9 }
```

Listing 17: The second OpenCL kernel generated by VexCL, which executes the φ function.

The last kernel generated by VexCL executes the actual SpMV, as can be seen in listing 18. We can see that several parameters are generated for the ELL and CSR part of the HELL sparse

matrix representation. In table 5.1 we show what value each of these parameters represent. A input vector is passed to the kernel for the vector of the multiplication, and an output vector is passed to store the result. Lastly there is a scale variable that can be used if the resulting vector of the SpMV is multiplied by a scalar. The outer loop of the kernel loops over the rows of the matrix, and is parallelized. The first inner loop in the kernel loops over the current row in the ELL matrices, and first retrieves the corresponding column from the C_{ell} matrix, and then calculates the matrix vector multiplication on that element of the matrix. Note that the padding of the ELL matrix is represented by setting the entry in the C_{ell} to the maximum value of an unsigned integer. Then a second for loop iterates over the R_{csr} matrix to calculate the matrix vector multiplication of the CSR part. Here, again, the corresponding column is retrieved first from the C_{csr} matrix, and then the matrix vector multiplication on that element of the matrix is computed. The sum of all those multiplication is then stored in the output vector to complete the SpMV calculation of that row.

```
kernel void hybrid_ell_spmv(ulong n, double scale, ulong ell_w,
  ulong ell_pitch, global const ulong *ell_col,
  global const double *ell_val, global const ulong *csr_row,
  global const ulong *csr_col, global const double *csr_val,
  global const double *in, global double *out) {
  for(ulong i = get_global_id(0); i < n; i += get_global_size(0)) {
    double sum = 0;
    for(size_t j = 0; j < ell_w; ++j) {
      ulong c = ell_col[i + j * ell_pitch];
      if (c != (ulong)(-1)) {
        sum += ell_val[i + j * ell_pitch] * in[c];
      }
    }
    if (csr_row) {
      for(size_t j = csr_row[i], e = csr_row[i + 1]; j < e; ++j) {
        sum += csr_val[j] * in[csr_col[j]];
      }
    }
    out[i] = scale * sum;
  }
}
```

Listing 18: Third OpenCL kernel generated by VexCL, which executes the SpMV calculation.

5.3 Xilinx Vitis implementation

5.3.1 Porting process

When we look at the VexCL and OpenCL code, we see that there are some new constructs, which we have not seen before in the porting process. This means that we have to adapt our porting guide, as described in chapter 3.

First of all, there are now three kernels instead of one. We can ignore the first kernel, since it only makes an unnecessary copy of the $\vec{u_i}$ vectors, which we do not need as we use the $\vec{u_i}$ vectors as read-only vectors, so we do not modify them. Even if we modified the $\vec{u_i}$ vectors, we could simply copy them for the tag on the host-side. The other two kernels are however both necessary. We could combine the two kernels into one kernel, but that can have a big influence on the performance if the kernels are large, or if there are many kernels. Instead we will adapt the porting guide to support multiple kernels. On the kernel side of the porting

process nothing changes, we simply create the kernels separately of each other, we will call the first kernel `phi`, and the second kernel `spmat`. We first compile both kernels seperately into XO files, and then link the XO files to the same binary using the Vitis compiler. The configuration file that we used to specify the connectivity of the kernel for the compiler can combine the configurations of both kernels in the same file. The compiler will then generate a design for the FPGA that contains both the kernels, meaning that both the kernels will be present on the FPGA when we load the binary, and no kernel swapping will be needed. On the host-side we only need to duplicate one line to load an extra kernel, the line `cl::Kernel` *krnl_name*`(program,` *kernel_function_name*`, &err)`. The only thing we need to change to this duplicated line is the *krnl_name* and *kernel_function_name*. We can then set the kernel arguments for both kernels, and enqueue the kernels separately. The last thing that changes is that we can now not only have input and output buffers, but also buffers that are only used to communicate between kernels. To specify such a buffer, we can need to replace the memory flags from `CL_MEM_`*READ/WRITE*`_ONLY | CL_MEM_USE_HOST_PTR` to `CL_MEM_HOST_NO_ACCESS`, and set the host pointer argument of the function to `NULL`.

Another construct we have not encountered before is the `vex::SpMat` class. In section 5.2 we saw that the class changed a sparse matrix in CSR format to a sparse matrix in HELL format. We will need to do the same to be able to call the kernel. To accomplish this we have created our own `SpMat` class, based on the `vex::SpMat`, and changed the code so that it no longer depends on the VexCL code base. We did this by (1) replacing the VexCL vectors with standard vectors, which we will later use in the OpenCL buffers, (2) removing the support to split the sparse matrix over multiple kernels, (3) removing the OpenCL code generation, and (4) making the internal HELL format public, so that we can retrieve the data for the kernel arguments later on. We can then simply call this class with the same parameters as the VexCL class, except for the VexCL context parameter which must be left out, and than create buffers for the C_{ell} and V_{ell} matrices, and for the R_{csr}, C_{csr}, and V_{csr} vectors.

For the first kernel, we have two functions we have not encountered yet, `log` and `sin`. Since C only provide those functions in the math library, we need to add `#include <math.h>` add the top of the kernel. When calling this kernel, we can simply use the $\vec{u_i}$ for the tag parameters. Note that the parameter format is `prm_tag_`n`_1`, where n is the value we gave to the tag in VexCL.

For the last kernel that calculates the SpMV, we also identify some new parameters. Fortunately, they are all provided by the `spmat` class, except for the `scale` parameter, as mentioned in section 5.2.1, which must be set to one in this case, since we do not multiply the SpMV result with a scalar.

5.3.2 Compiled application

In figure 5.1 we show the system diagram of the compiled kernels on a Xilinx U250 FPGA. Note that in this version of the kernels we also implemented the memory interface optimization from chapter 4, and spread the parameters over the memory interfaces. We can see that the phi kernel uses more resources than the `spmat` kernel, this is expected, as the phi function consists of many calculations.

5.3.3 Verifying correctness

To test the correctness of the application, we have again implemented a sequential version of the algorithm that runs on the CPU, and verified the output of the application using random data of different sizes and densities. From this verification process, we conclude that the port was successful.

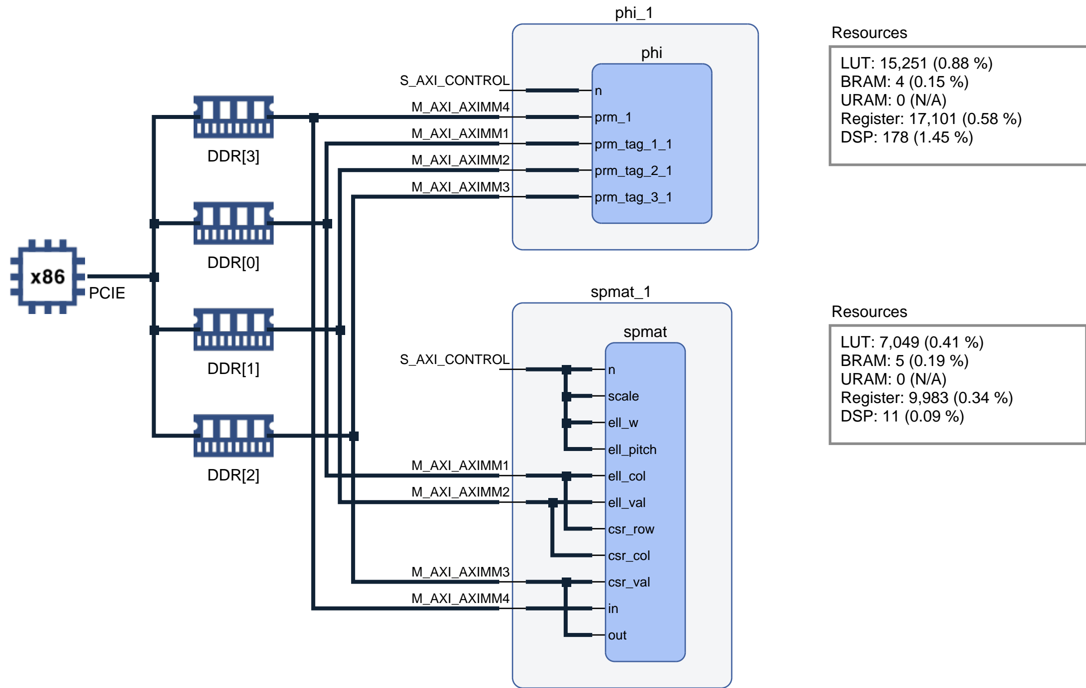


Figure 5.1: System diagram of ported sparse matrix-vector multiplication application from VexCL to Xilinx Vitis on a Xilinx U250 FPGA.

5.4 Performance study

To test how effective the optimizations of chapter 4 are, we also apply them to the SpMV application.

5.4.1 Implementations

In optimization 1, we applied the common optimizations of aligning the host memory and using more memory interfaces. We did have to use the same interfaces multiple times, since there are almost three times more data buffers than memory interfaces. We chose the interface mapping such that parameters that can be read at the same time use different interfaces.

In optimization 2, we replaced the floating-point numbers of optimization 1 with fixed-point numbers. This did create a problem with the `log` and `sin` functions, since the functions are not defined for fixed-point numbers in the standard math header. To fix this we used the hls_math header, provided by Xilinx, which define the functions `hls::log` and `hls::sin` for fixed-point integers. They do however not work with custom quantization and overflow setting, so we had to redefine our fixed-point integers to use the default quantization method of truncating to minus infinity, and the default overflow setting of wrapping around.

In optimization 3, we added burst buffers to the phi kernel of optimization 2. Note that we do not use burst buffers in the spmat kernel, since the kernel does not access the memory in a sequential order.

We do not apply the data width saturation optimization, since the results from chapter 4 indicate that the Vitis compiler does not handle structs well.

Unfortunately we were not able to compile optimizations 2 and 3 for the hardware, since there were timing issues with the phi kernel during the compilation. This means that the kernel is too complicated to be run at the target frequency. This could be solved by lowering the target frequency, but the compilation report indicates that we would have to at least half the target

No.	m×n	density	total matrix elements
1	512×512	0.01	2621
2	5000×5000	0.01	$2.5 \cdot 10^5$
3	$10^5 \times 10^5$	0.01	10^8
4	12500×12500	0.64	10^8
5	$10^6 \times 10^6$	10^{-4}	10^8

Table 5.2: Sparse matrix configurations used for the experiment.

frequency from 300 MHz to 150 MHz, which would have a large impact on the performance. So we choose to drop these two optimizations, as we suspect the performance impact would be too large to give useful results. We suspect the timing issues are caused by current inefficient implementations of the log and sin functions in the hls_math library for fixed-point integers.

5.4.2 Method

We compare three implementations of the SpMV application, the VexCL implementation, the unoptimized Vitis implementation, and optimization 1. The VexCL implementation uses the GPU as accelerator, and the Vitis implementations uses an FPGA. The hardware and profiling tools used for this experiment are the same as in chapter 4.

We generate random input data for the experiment using a Python script. To generate random sparse matrices with a given density, the sparse.random function is used from the SciPy library[1]. We use five different sparse matrix configurations, as specified in table 5.2. We choose the first three configurations to see how the matrix size influences the performance, and the last two configurations to see how the density of the sparse matrix influences the performance. All matrix configurations are run 10 times, all with new random input data.

5.4.3 Results

Wall time

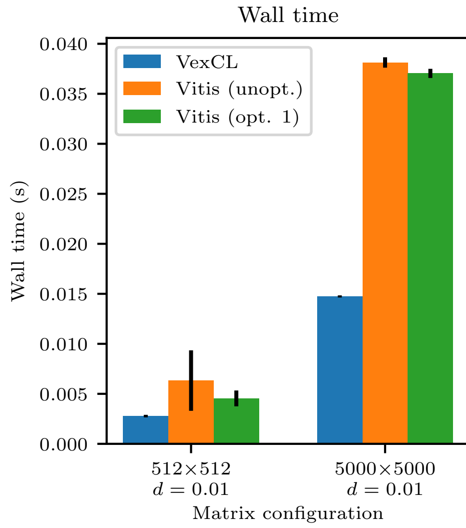
In figure 5.2 we show the wall time of the implementations. We can see that, in line with the results from the affine transformation application in chapter 4, the VexCL implementation on the GPU is always faster than the Vitis implementations. The difference between the VexCL implementation and the unoptimized Vitis implementation is smaller however for the SpMV application than for the affine transformation application. The VexCL implementation is generally only up to 50% faster than the Vitis implementation for the SpMV application, in comparison to generally about 80% faster for the affine transformation application. The optimized version of the Vitis implementation is always faster than the unoptimized version, generally between 5% and 15%.

We can see that the matrix with a density of 0.64 takes the longest time of the matrices of our density experiment, while the matrix with the density of 0.01 is generally slightly faster than the matrix with a density of 10^{-4}.

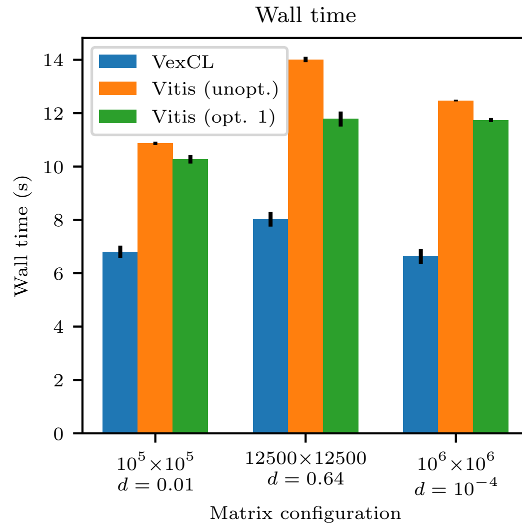
Kernel time

In figure 5.3 we show the kernel time of the phi kernel on the FPGA for the different implementations, and in figure 5.4 we show the kernel time of the spmat kernel. We can see that the optimized version of the phi kernel is much faster than the unoptimized version, especially as the vector size rises, reaching a speedup of up to 95% in comparison to the unoptimized version. This is less the case for the spmat kernel, where the optimized version is up to 15% slower for the smaller matrices, and is only about 25% faster for the larger matrices.

[1] scipy.sparse.random – `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.random.html`

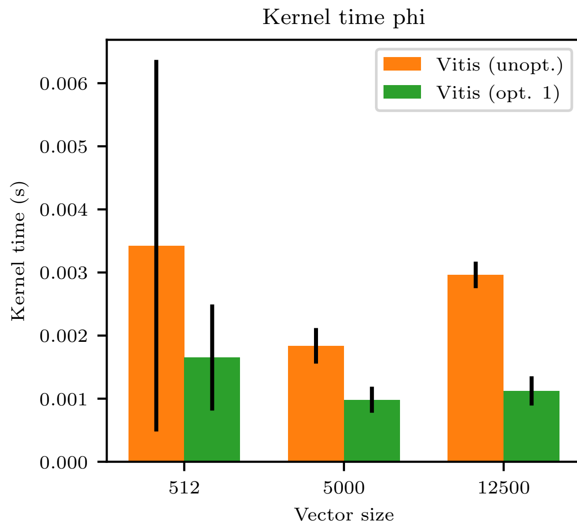


(a)

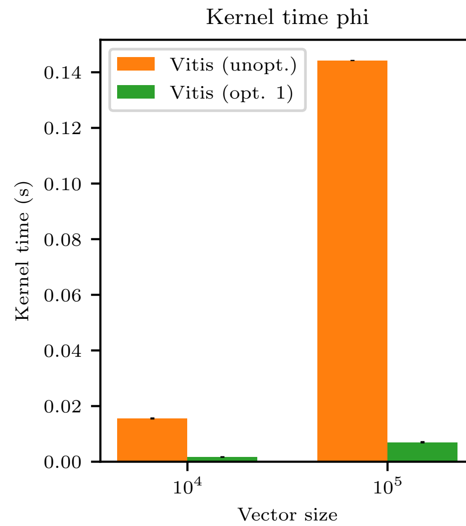


(b)

Figure 5.2: The wall time of the different SpMV implementations with different matrix configurations, averaged over ten runs.

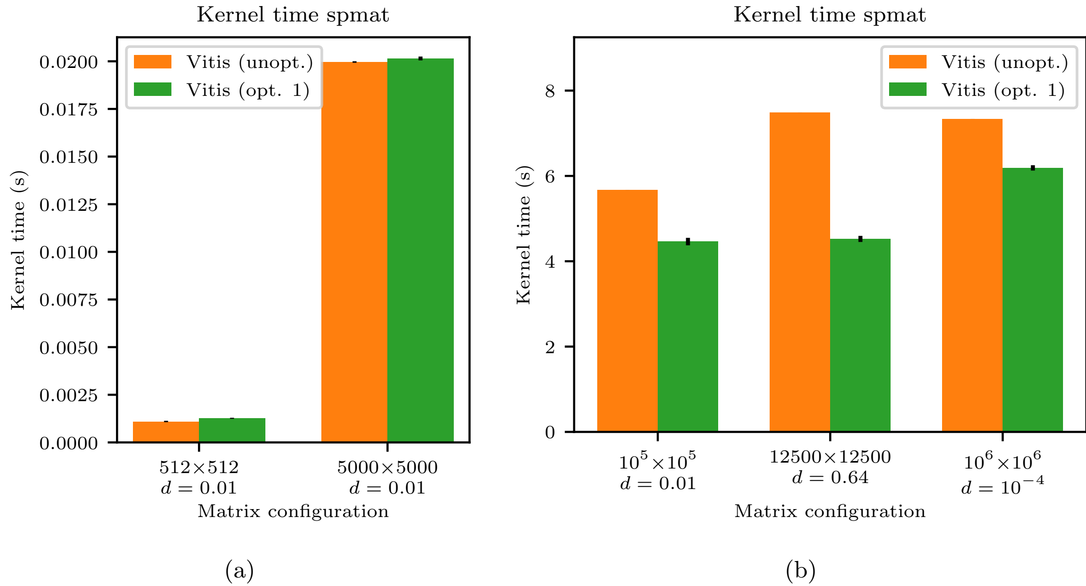


(a)



(b)

Figure 5.3: The kernel time of the phi kernel in different SpMV implementations with different matrix configurations, averaged over ten runs.



(a) (b)

Figure 5.4: The wall time of the spmat kernel in different SpMV implementations with different matrix configurations, averaged over ten runs.

Power usage

In figure 5.5 we show the power usage of the different implementations. We can see that the GPU uses around 50 W of power with the VexCL implementation, while the FPGA uses between 27 W and 37 W of power with the Vitis implementations. We can see that the unoptimized Vitis implementation uses less power on average as the matrix size grows, while the optimized Vitis implementation uses more power as the matrix size grows.

Energy usage

In figure 5.6 we show the approximate energy usage of the implementations, calculated using the wall time and power usage. We can see that for the smallest matrix the VexCL and optimized Vitis version use the least energy, using about 31% less energy than the unoptimized Vitis version. For the 5000×5000 matrix, the VexCL implementation uses the least amount of energy, using 34% less energy than the optimized Vitis version, and 39% less energy than the unoptimized Vitis version. For the larger matrices the unoptimized Vitis implementation uses the least amount of energy, using up to 16% less energy than the VexCL implementation, and up to 22% less energy than the optimized Vitis implementation. The only exemption to this is the largest matrix, where the VexCL implementation uses the same amount of energy as the unoptimized Vitis implementation.

In this application the difference between the energy usage of the Vitis implementations and the VexCL implementation is more favorable for the Vitis implementations than when we looked at the affine transformation application in chapter 4, where the VexCL implementation generally used between 47% and 73% less energy than the Vitis implementation.

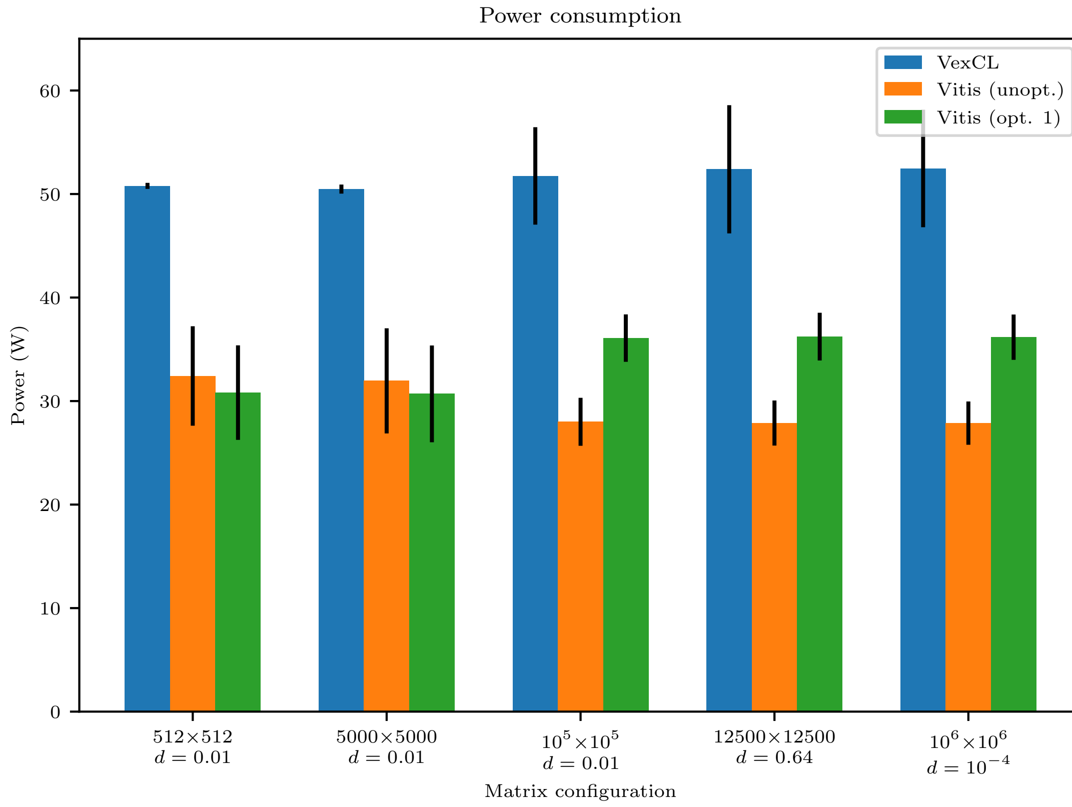


Figure 5.5: The power usage of the different SpMV implementations with different matrix configurations, averaged over ten runs.

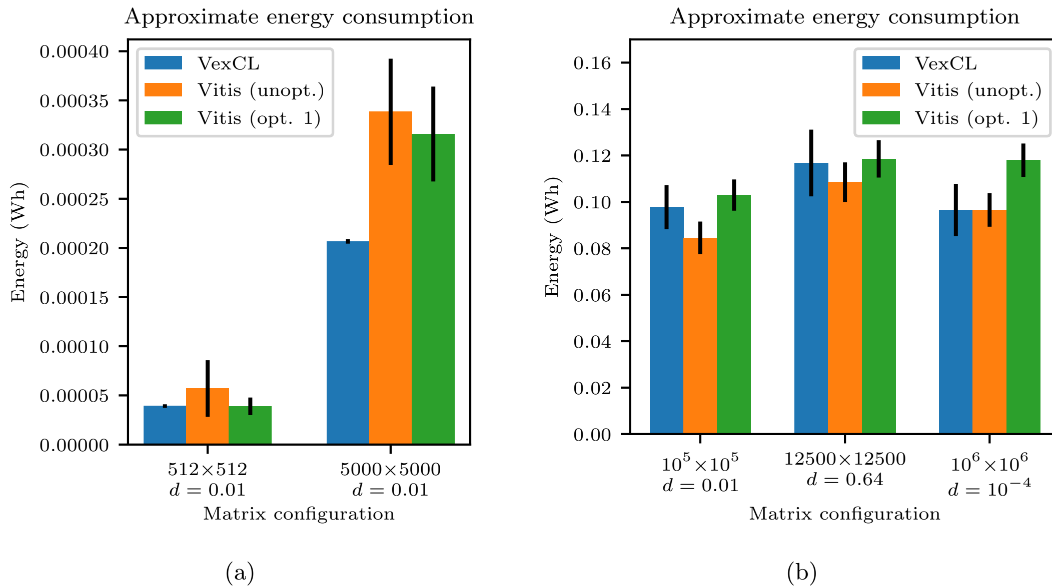


Figure 5.6: The approximate energy usage of the different SpMV implementations with different matrix configurations, derived from the wall time and power usage results.

5.5 Summary

By extending the porting guide to allow more VexCL constructs, we have successfully ported a SpMV application from VexCL to Xilinx Vitis. The porting guide now supports running multiple kernels, which can be useful for many applications. We have also ported the VexCL sparse matrix library to be able to use sparse matrices in Xilinx Vitis. The new porting guidelines are shown in table 5.3 for the kernel code, and table 5.4 for the host code.

Our performance results indicate that this kind of applications is better suited for FPGAs than the affine transformation kernel used in chapter 4. We can see that the energy usage of the Vitis implementations is closer to the energy usage of the VexCL implementation for the SpMV application than for the affine transformation application. The energy usage of the Vitis implementations is sometimes even lower than the energy usage of the VexCL implementation.

The memory alignment optimization, and using more memory interfaces, make the Vitis implementation faster, like in chapter 4. However, for larger matrices, this also increase the power usage in this application. The fixed-point integer optimization could not be implemented in the phi kernel, due to the complexity of the kernel.

VexCL	**Xilinx Vitis**

// Entire kernel	#include <stddef.h>
	extern "C" {
	// Entire kernel
	}

kernel void vexcl_vector_kernel(...) {...}	void *kernel_function_name*(...) {...}
ulong	size_t
idx = get_global_id(0)	idx = 0
idx += get_global_size(0)	++idx
log/sin/cos	#include <math.h> ... log/sin/cos
global *type* **prm*	const *type* **prm* // if *prm* is input buffer *type* **prm* // if *prm* is output buffer ... #pragma HLS INTERFACE m_axi port=*prm* bundle=aximm<*memory interface No.*>

Table 5.3: Kernel code changes from VexCL to Xilinx Vitis.

<table>
<tr><th>VexCL</th><th>Xilinx Vitis</th></tr>
<tr><td>ctx(vex::Filter::DoublePrecision);</td><td>device = get_xilinx_devices().front();
. . .
cl::CommandQueue q(context, device,
CL_QUEUE_PROFILING_ENABLE, &err);
cl::Kernel krnl_name_1(program, kernel_1_function_name,
&err);
cl::Kernel krnl_name_2(program, kernel_2_function_name,
&err);
. . .
cl::Kernel krnl_name_n(program, kernel_n_function_name,
&err);</td></tr>
<tr><td>vex::vector<cl_type> A(ctx, a.size(),
a.data());

vex::vector<cl_type> T(ctx, t.size());

...
vex::copy(T.begin(), T.end(), t.data());</td><td>cl::Buffer A(context, CL_MEM_READ_ONLY |
CL_MEM_USE_HOST_PTR, sizeof(type) * a.size(), a.data(),
&err);
cl::Buffer T(context, CL_MEM_WRITE_ONLY |
CL_MEM_USE_HOST_PTR, sizeof(type) * t.size(), t.data(),
&err);</td></tr>
<tr><td>// Temporary value used between kernels</td><td>cl::Buffer tmp(context, CL_MEM_HOST_NO_ACCESS,
sizeof(type) * a.size(), a.data(), &err);</td></tr>
<tr><td>vex::spMat<cl_type> A(ctx, m, n,
A_row.data(), A_col.data(),
A_data.data());</td><td>#include "spmat.hpp"
. . .
spMat<type> A(m, n, A_row.data(), A_col.data(),
A_data.data());
auto A_ell = A.ell_mat();
// Create input buffers for: A_ell->mat.ell.col,
A_ell->mat.ell.val, A_ell->mat.csr.row, A_ell->mat.csr.col,
A_ell->mat.csr.val</td></tr>
<tr><td>T = X + Y;</td><td>krnl_name.setArg(0, output_vector_size);
krnl_name.setArg(1, T);
krnl_name.setArg(2, X);
krnl_name.setArg(3, Y);</td></tr>
<tr><td>x = vex::tag<n>(X)
T = x + x;</td><td>krnl_name.setArg(0, output_vector_size);
krnl_name.setArg(1, T);
krnl_name.setArg(2, X);</td></tr>
<tr><td>reshape(X, extents[m][n], extents[1])</td><td>krnl_name.setArg(0, output_vector_size);
krnl_name.setArg(1, X);
krnl_name.setArg(2, 1); // skip indices
krnl_name.setArg(3, 0); // offset
krnl_name.setArg(4, 1); // repetitions
krnl_name.setArg(5, n); // modulo</td></tr>
<tr><td>reduce<OP>(extents[m][n], X, 1)</td><td>krnl_name.setArg(0, output_vector_size);
krnl_name.setArg(1, X);
krnl_name.setArg(2, 0); // offset
krnl_name.setArg(3, m); // first dimension matrix
krnl_name.setArg(4, n); // second dimension matrix
krnl_name.setArg(5, n); // amount of values to reduce each time
krnl_name.setArg(6, 1); // distance between values</td></tr>
<tr><td>// spmat A, scalar α, and vector X
T = α * A * X</td><td>krnl_name.setArg(0, output_vector_size);
krnl_name.setArg(1, α);
krnl_name.setArg(2, A_ell->mat.ell.width);
krnl_name.setArg(3, A_ell->mat.ell.pitch);
krnl_name.setArg(4, A_ell_col);
krnl_name.setArg(5, A_ell_val);
krnl_name.setArg(6, A_csr_row);
krnl_name.setArg(7, A_csr_col);
krnl_name.setArg(8, A_csr_val);
krnl_name.setArg(9, X);
krnl_name.setArg(10, T);</td></tr>
<tr><td>// Finished computations</td><td>q.enqueueMigrateMemObjects(input_buffers_krnl_1, 0);
q.enqueueTask(krnl_1_name);
. . .
q.enqueueMigrateMemObjects(input_buffers_krnl_n, 0);
q.enqueueTask(krnl_n_name);
q.enqueueMigrateMemObjects(output_buffers,
CL_MIGRATE_MEM_OBJECT_HOST);
q.finish();</td></tr>
</table>

Table 5.4: Host code changes from VexCL to Xilinx Vitis.

CHAPTER 6

Conclusion

FPGAs are among the newest accelerators used by the HPC community. Because of the effort put into HLS tools, it is currently possible to program FPGAs using programming languages like OpenCL and C++. However, to be completely adopted by the HPC community, tools and libraries that are currently in use in the HPC space must start targeting FPGAs. In this thesis, we focus on this aspect of making FPGAs available to the HPC community, and propose a solution to retarget VexCL code to FPGAs. To this end, we use a Xilinx U250 FPGA, and propose a step-by-step porting process to convert a VexCL application into a Xilinx Vitis application, which we demonstrate on two case-studies, an affine transformation application, and a SpMV application. This chapter summarizes our findings and contributions, and highlights future work directions.

6.1 Main findings

The goal of this research was to propose a systematic method to enable VexCL applications to run on FPGA-accelerated systems. To support this process, we formulated four subquestions. In the following paragraphs, we list our answers for these questions, and the main findings we collected in the process.

[SQ1] What language supported by VexCL is a convenient intermediate representation for compiling VexCL to Xilinx Vitis code for FPGAs?
We first investigated the possible target languages of VexCL, which are discussed in chapter 2. We concluded that VexCL can target several programming languages, including OpenMP, OpenCL and CUDA, and that it was also possible to create a custom back-end to target a new programming language. We decided that OpenCL would be the most convenient intermediate representation for our porting process, since it was the most similar to the C kernels used by Xilinx Vitis, and was already supported by VexCL.

[SQ2] How can we design a step-by-step guide to convert VexCL code to Xilinx Vitis code that targets FPGAs?
We developed a porting guide — presented in chapter 3 — by looking at the steps we had to take to convert a particular VexCL application to Xilinx Vitis. In the guide, we first retrieve the OpenCL kernel that VexCL uses, and port it to a Vitis kernel; next, the original VexCL code is ported to Vitis host code. While the kernel was rather straightforward to port from the OpenCL kernel, translating the host code from the VexCL application required more investigation.

[SQ3] What optimizations can we apply to applications ported from VexCL to Xilinx Vitis code that improve the performance of the code?
In chapter 4 we presented several possible optimizations: using aligned memory, fixed-point integers, burst transfers, and saturating the data-width. When applied to our ported affine

transformation application, these optimization lead to mixed results. For example, using aligned memory, and using fixed-point integers improved the performance of the application on the FPGA. In chapter 5 we applied and tested the aligned memory optimization to a SpMV application, and saw that the performance improved for this application too.

[SQ4] How effective is the compilation guide?
In chapter 5, we applied our porting method to a SpMV application (written in VexCL). This application uses two kernels. With a few additions to the porting guide, we were successful in porting this application as well. This second case-study indicates our porting guide is also effective in porting applications with multiple kernels.

Based on all these findings, we can formulate an answer to our main research question:
How can VexCL code be effectively compiled into code for FPGAs?
By following the step-by-step porting guide presented in this thesis, we are able to port VexCL code to Xilinx Vitis code that is able to run on Xilinx FPGAs. We can also apply several optimizations to this ported application to improve the performance. We have tested this porting guide on two applications — an affine transformation application, and a SpMV application — and both applications were successfully ported. To be able to say that the compilation guide is 100% effective, more testing is required, but the results are promising.

6.2 Contributions

This thesis makes the following contributions:

- We have proposed a step-by-step porting guide, which explains how to port applications written in VexCL to Xilinx Vitis code that can run on a Xilinx FPGA.
- We have verified the correctness of the step-by-step porting guide using two case-studies, the first one being an affine transformation application, and the second one being a SpMV application.
- We have proposed five optimizations that can be applied to ported applications, and applied them to the applications used in both case-studies. The optimizations are:
    1. Aligning the host memory
    2. Using multiple memory interfaces
    3. Replacing floating-point numbers with fixed-point numbers
    4. Using burst-transfers
    5. Saturating the data width of the memory interfaces

  On the affine transformation application we were able to correctly implement all five optimizations, but in the SpMV application we could only apply the first two optimizations.
- In an empirical study on the applications from both the case-studies, we have evaluated the execution time, and energy usage of the Vitis implementations, and the VexCL implementation.

  The VexCL implementation was generally faster, and — for the affine transformation application — also more energy efficient, but with the SpMV application, the Vitis implementation was generally more energy efficient.

6.3 Limitations

Despite the promising results we have seen in our analysis of two different case-studies, there are two limitations to be considered. First, the porting guide currently only supports a subset of all the VexCL functions. For it to be applicable to *any* VexCL code, it would have to be extended to support all the functions. Second, we have only tested a limited amount of applications. It is

therefore possible that the porting guide does not work in every situation. Especially the values and order of the kernel arguments that must be passed from the host to the kernel are hard to determine using the current guide. To make it easier to automate the process and determine the right value of the kernel argument, a custom back-end for VexCL should be created, since then all the information is available.

6.4 Future Work

In this section we propose several future research subjects, building on the research work in this thesis.

- While we have succeeded in porting VexCL applications to FPGAs in this thesis, the process itself still requires a lot of manual work. To improve the usability of this porting process, it would be useful to create a custom VexCL back-end, based on the porting guide, to be able to automate the process described in this thesis, and make the process more effective.
- On top of the optimizations proposed in this thesis, there are also FPGA-specific optimizations proposed by Zohouri et al. and Paulino, Ferreira, and Cardoso that can be applied to OpenCL kernels [23, 12]. It should be possible to adapt those optimizations to be able to apply them to Vitis kernels, and include the optimization in the porting process. Using those optimizations, it might be possible to achieve a lower execution time of the ported application, resulting in less energy usage.
- An interesting case study would be to port more applications using the porting guide, and see if it is possible to extend the known subset of applications that already have a energy usage when using a FPGA, in comparison to using a GPU. It could also be interesting to test the applications on more FPGAs and GPUs, and see how they compare.
- Gozillon et al. are currently working on allowing SYCL to run on Xilinx FPGAs. SYCL is a C++ library that, like VexCL, provides support for heterogeneous computing, without using different code for the host and the kernel. It would be interesting to compare the development process of writing SYCL and VexCL applications that target FPGAs.

Bibliography

[1] Wenwu Chen and Bill Poirier. "Parallel implementation of an efficient preconditioned linear solver for grid-based applications in chemical physics. III: Improved parallel scalability for sparse matrix–vector products". In: *Journal of Parallel and Distributed Computing* 70.7 (2010), pp. 779–782. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2010.03.008.

[2] P. Coussy et al. "An Introduction to High-Level Synthesis". In: *IEEE Design Test of Computers* 26.4 (2009), pp. 8–17. DOI: 10.1109/MDT.2009.69.

[3] Denis Demidov et al. *ddemidov/vexcl: 1.4.2*. Version 1.4.2. Apr. 2021. DOI: 10.5281/zenodo.4722446. URL: https://github.com/ddemidov/vexcl/tree/1.4.2.

[4] Ambrose Finnerty and Hervé Ratigner. *Reduce Power and Cost by Converting from Floating Point to Fixed Point*. Mar. 2017. URL: https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf.

[5] Andrew Gozillon et al. "triSYCL for Xilinx FPGA". In: *Proceedings of the 2020 International Conference on High Performance Computing & Simulation (HPCS)*. United States: IEEE, Dec. 2020. URL: https://research-portal.uws.ac.uk/en/publications/trisycl-for-xilinx-fpga.

[6] Taher H. Haveliwala. *Efficient Computation of PageRank*. Technical Report 1999-31. Stanford InfoLab, 1999. URL: http://ilpubs.stanford.edu/386/.

[7] Intel Corporation. *Intel® oneAPI HPC Toolkit*. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/hpc-toolkit.html.

[8] Sandra Loosemore et al. *The GNU C Library Reference Manual*. Aug. 2020. URL: https://www.gnu.org/software/libc/manual/pdf/libc.pdf.

[9] G. Martin and G. Smith. "High-Level Synthesis: Past, Present, and Future". In: *IEEE Design Test of Computers* 26.4 (2009), pp. 18–25. DOI: 10.1109/MDT.2009.83.

[10] Bruce Merry. "A Performance Comparison of Sort and Scan Libraries for GPUs". In: *Parallel Processing Letters* 25.04 (2015), p. 1550007. DOI: 10.1142/S0129626415500073.

[11] Fahad Bin Muslim et al. "Energy-efficient FPGA Implementation of the k-Nearest Neighbors Algorithm Using OpenCL". In: *8th Workshop on Scalable Computing*. Vol. 9. 2016, pp. 141–145. DOI: 10.15439/2016F327.

[12] N. Paulino, J. C. Ferreira, and J. M. P. Cardoso. "Optimizing OpenCL Code for Performance on FPGA: k-Means Case Study With Integer Data Sets". In: *IEEE Access* 8 (2020), pp. 152286–152304. DOI: 10.1109/ACCESS.2020.3017552.

[13] K. Shagrithaya, K. Kępa, and P. Athanas. "Enabling development of OpenCL applications on FPGA platforms". In: *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*. 2013, pp. 26–30. DOI: 10.1109/ASAP.2013.6567546.

[14] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2002. ISBN: 1-4020-7089-6. DOI: 10.1007/b116662.

[15] Xilinx, Inc. *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. May 2020. URL: https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf.

[16] Xilinx, Inc. *AXI Reference Guide*. July 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[17] Xilinx, Inc. *UltraScale Architecture Configurable Logic Block User Guide*. Feb. 2017. URL: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

[18] Xilinx, Inc. *UltraScale Architecture DSP Slice User Guide*. July 2020. URL: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[19] Xilinx, Inc. *Vitis High-Level Synthesis User Guide*. Mar. 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.

[20] Xilinx, Inc. *Vitis Unified Software Platform*. URL: https://www.xilinx.com/products/design-tools/vitis.html.

[21] Xilinx, Inc. *Vitis Unified Software Platform Documentation: Application Acceleration Development*. Mar. 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1393-vitis-application-acceleration.pdf.

[22] Xilinx, Inc. "Xilinx Announces Vitis – a Unified Software Platform Unlocking a New Design Experience for All Developers". In: *Xilinx* (Oct. 1, 2019). URL: https://www.xilinx.com/news/press/2019/xilinx-announces-vitis--a-unified-software-platform-unlocking-a-new-design-experience-for-all-developers.html.

[23] Hamid Reza Zohouri et al. "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. Salt Lake City, Utah: IEEE Press, 2016. ISBN: 9781467388153. DOI: 10.1109/SC.2016.34.

APPENDIX A

Testing result data

The test results from the experiments on the affine transformation application from chapter 4 can be seen in tables A.1, A.2 and A.3. The percentages changed are relative to the unoptimized Vitis implementation.

The test results from the experiments on the SpMV application from chapter 5 can be seen in tables A.4, and A.5. The percentages changed are relative to the unoptimized Vitis implementation.

Version	Matrix dimensions: 32×32							
	Wall time		Kernel time		Power usage		Energy usage	
	value	change	value	change	value	change	value	change
VexCL	1.93 ms ± 0.13 ms	-73.6%	N/A	-	48.94 W ± 0.42 W	+64.4%	26.23 µWh ± 1.84 µWh	-56.6%
Vitis (unopt.)	7.31 ms ± 3.38 ms	0.0%	6.26 ms ± 3.35 ms	0.0%	29.76 W ± 3.06 W	0.0%	60.41 µWh ± 28.75 µWh	0.0%
Vitis (opt. 1)	4.44 ms ± 2.95 ms	-39.2%	3.55 ms ± 2.93 ms	-43.3%	28.37 W ± 2.97 W	-4.7%	34.99 µWh ± 23.69 µWh	-42.1%
Vitis (opt. 2)	3.43 ms ± 2.33 ms	-53.1%	2.46 ms ± 2.31 ms	-60.7%	31.96 W ± 0.32 W	+7.4%	30.42 µWh ± 20.68 µWh	-49.6%
Vitis (opt. 3)	3.61 ms ± 1.99 ms	-50.5%	2.70 ms ± 1.96 ms	-56.8%	32.59 W ± 0.35 W	+9.5%	32.73 µWh ± 18.02 µWh	-45.8%
Vitis (opt. 4)	2.54 ms ± 1.43 ms	-65.2%	1.64 ms ± 1.38 ms	-73.8%	32.73 W ± 0.55 W	+10.0%	23.12 µWh ± 13.04 µWh	-61.7%
Vitis (opt. 5)	3.15 ms ± 1.35 ms	-56.9%	2.22 ms ± 1.34 ms	-64.5%	32.76 W ± 0.47 W	+10.1%	28.64 µWh ± 12.28 µWh	-52.6%

Version	Matrix dimensions: 256×256							
	Wall time		Kernel time		Power usage		Energy usage	
	value	change	value	change	value	change	value	change
VexCL	2.40 ms ± 41.02 µs	-68.0%	N/A	-	49.59 W ± 0.50 W	+65.4%	33.03 µWh ± 0.65 µWh	-47.1%
Vitis (unopt.)	7.50 ms ± 1.68 ms	0.0%	5.36 ms ± 1.80 ms	0.0%	29.97 W ± 3.20 W	0.0%	62.45 µWh ± 15.59 µWh	0.0%
Vitis (opt. 1)	5.75 ms ± 1.76 ms	-23.3%	4.40 ms ± 1.78 ms	-17.9%	28.29 W ± 2.97 W	-5.6%	45.18 µWh ± 14.73 µWh	-27.6%
Vitis (opt. 2)	4.72 ms ± 0.12 ms	-37.0%	3.69 ms ± 63.22 µs	-31.3%	32.20 W ± 0.36 W	+7.4%	42.24 µWh ± 1.14 µWh	-32.3%
Vitis (opt. 3)	4.57 ms ± 98.57 µs	-39.1%	3.47 ms ± 0.16 ms	-35.3%	32.14 W ± 0.33 W	+7.2%	40.77 µWh ± 0.97 µWh	-34.7%
Vitis (opt. 4)	5.46 ms ± 0.25 ms	-27.2%	4.33 ms ± 0.21 ms	-19.2%	32.28 W ± 0.36 W	+7.7%	48.95 µWh ± 2.28 µWh	-21.6%
Vitis (opt. 5)	4.52 ms ± 0.28 ms	-39.7%	3.43 ms ± 0.24 ms	-36.1%	32.20 W ± 0.27 W	+7.4%	40.46 µWh ± 2.55 µWh	-35.2%

Version	Matrix dimensions: 1024×1024							
	Wall time		Kernel time		Power usage		Energy usage	
	value	change	value	change	value	change	value	change
VexCL	8.76 ms ± 40.45 µs	-84.2%	N/A	-	50.19 W ± 0.42 W	+67.9%	0.12 mWh ± 1.17 µWh	-73.4%
Vitis (unopt.)	55.30 ms ± 0.63 ms	0.0%	42.17 ms ± 0.23 ms	0.0%	29.89 W ± 3.13 W	0.0%	0.46 mWh ± 48.42 µWh	0.0%
Vitis (opt. 1)	41.08 ms ± 0.49 ms	-25.7%	38.49 ms ± 0.74 ms	-8.7%	29.04 W ± 2.91 W	-2.8%	0.33 mWh ± 33.43 µWh	-27.8%
Vitis (opt. 2)	42.49 ms ± 0.32 ms	-23.2%	40.54 ms ± 0.31 ms	-3.9%	32.39 W ± 0.27 W	+8.3%	0.38 mWh ± 4.31 µWh	-16.8%
Vitis (opt. 3)	43.85 ms ± 0.22 ms	-20.7%	41.70 ms ± 0.20 ms	-1.1%	32.12 W ± 0.27 W	+7.4%	0.39 mWh ± 3.87 µWh	-14.8%
Vitis (opt. 4)	56.56 ms ± 0.31 ms	+2.3%	54.30 ms ± 0.20 ms	+28.7%	32.31 W ± 0.34 W	+8.1%	0.51 mWh ± 5.95 µWh	+10.5%
Vitis (opt. 5)	40.35 ms ± 0.18 ms	-27.0%	38.17 ms ± 0.11 ms	-9.5%	32.28 W ± 0.41 W	+8.0%	0.36 mWh ± 4.90 µWh	-21.2%

Table A.1: Test results of smaller matrices from the experiments on the affine transformation application.

Version	Matrix dimensions: 2000×2000							
	Wall time		Kernel time		Power usage		Energy usage	
	value	change	value	change	value	change	value	change
VexCL	33.60 ms ± 0.44 ms	-83.5%	N/A	-	49.32 W ± 0.71 W	+67.1%	0.46 mWh ± 8.97 µWh	-72.5%
Vitis (unopt.)	0.20 s ± 1.76 ms	0.0%	0.16 s ± 0.17 ms	0.0%	29.51 W ± 3.07 W	0.0%	1.67 mWh ± 0.17 mWh	0.0%
Vitis (opt. 1)	0.15 s ± 2.00 ms	-24.9%	0.15 s ± 1.60 ms	-9.9%	29.79 W ± 3.00 W	+1.0%	1.27 mWh ± 0.13 mWh	-24.2%
Vitis (opt. 2)	0.14 s ± 1.73 ms	-30.3%	0.14 s ± 1.64 ms	-15.4%	32.38 W ± 0.37 W	+9.7%	1.28 mWh ± 21.39 µWh	-23.5%
Vitis (opt. 3)	0.16 s ± 0.74 ms	-21.7%	0.16 s ± 0.81 ms	-4.7%	32.50 W ± 0.38 W	+10.1%	1.44 mWh ± 18.04 µWh	-13.7%
Vitis (opt. 4)	0.21 s ± 0.97 ms	+1.1%	0.20 s ± 1.01 ms	+23.8%	32.75 W ± 0.42 W	+11.0%	1.88 mWh ± 25.47 µWh	+12.2%
Vitis (opt. 5)	0.15 s ± 0.29 ms	-28.3%	0.14 s ± 0.16 ms	-13.2%	32.48 W ± 0.29 W	+10.1%	1.32 mWh ± 11.94 µWh	-21.1%

Version	Matrix dimensions: $2 \cdot 10^6 \times 2$							
	Wall time		Kernel time		Power usage		Energy usage	
	value	change	value	change	value	change	value	change
VexCL	47.59 ms ± 1.24 ms	-97.8%	N/A	-	48.65 W ± 0.75 W	+77.8%	0.64 mWh ± 19.52 µWh	-96.0%
Vitis (unopt.)	2.13 s ± 1.46 ms	0.0%	2.05 s ± 0.17 ms	0.0%	27.36 W ± 2.23 W	0.0%	16.15 mWh ± 1.31 mWh	0.0%
Vitis (opt. 1)	2.02 s ± 8.26 ms	-4.8%	2.01 s ± 8.25 ms	-1.9%	31.60 W ± 2.23 W	+15.5%	17.76 mWh ± 1.26 mWh	+10.0%
Vitis (opt. 2)	1.93 s ± 29.17 ms	-9.4%	1.92 s ± 28.91 ms	-6.6%	32.41 W ± 0.43 W	+18.4%	17.34 mWh ± 0.35 mWh	+7.3%
Vitis (opt. 3)	1.91 s ± 52.36 ms	-10.1%	1.90 s ± 52.41 ms	-7.4%	32.45 W ± 0.37 W	+18.6%	17.22 mWh ± 0.51 mWh	+6.6%
Vitis (opt. 4)	1.62 s ± 11.88 ms	-23.6%	1.62 s ± 12.00 ms	-21.3%	32.79 W ± 0.39 W	+19.8%	14.80 mWh ± 0.21 mWh	-8.4%
Vitis (opt. 5)	1.89 s ± 34.10 ms	-11.2%	1.88 s ± 34.49 ms	-8.5%	32.51 W ± 0.35 W	+18.8%	17.04 mWh ± 0.36 mWh	+5.5%

Version	Matrix dimensions: $2 \times 2 \cdot 10^6$							
	Wall time		Kernel time		Power usage		Energy usage	
	value	change	value	change	value	change	value	change
VexCL	0.57 s ± 1.72 ms	+185.2%	N/A	-	69.02 W ± 16.40 W	+135.0%	10.97 mWh ± 2.61 mWh	+570.3%
Vitis (unopt.)	0.20 s ± 1.42 ms	0.0%	0.15 s ± 0.18 ms	0.0%	29.37 W ± 3.18 W	0.0%	1.64 mWh ± 0.18 mWh	0.0%
Vitis (opt. 1)	0.15 s ± 2.42 ms	-25.6%	0.14 s ± 2.19 ms	-4.8%	30.09 W ± 2.99 W	+2.4%	1.25 mWh ± 0.13 mWh	-23.8%
Vitis (opt. 2)	0.15 s ± 2.66 ms	-26.0%	0.14 s ± 2.61 ms	-3.8%	32.51 W ± 0.34 W	+10.7%	1.34 mWh ± 27.85 µWh	-18.1%
Vitis (opt. 3)	0.16 s ± 0.43 ms	-19.4%	0.16 s ± 0.37 ms	+4.9%	32.32 W ± 0.33 W	+10.0%	1.45 mWh ± 15.34 µWh	-11.3%
Vitis (opt. 4)	0.21 s ± 1.02 ms	+4.7%	0.20 s ± 0.75 ms	+37.5%	32.67 W ± 0.41 W	+11.2%	1.91 mWh ± 25.65 µWh	+16.5%
Vitis (opt. 5)	0.15 s ± 0.70 ms	-26.2%	0.14 s ± 0.55 ms	-4.2%	32.34 W ± 0.33 W	+10.1%	1.33 mWh ± 14.98 µWh	-18.7%

Table A.2: Test results of matrices with same amount of elements, but differences in height and width, from the experiments on the affine transformation application.

Version	Matrix dimensions: 5000×5000							
	Wall time		Kernel time		Power usage		Energy usage	
	value	change	value	change	value	change	value	change
VexCL	0.19 s ± 2.17 ms	-85.0%	N/A	-	48.64 W ± 0.76 W	+71.0%	2.51 mWh ± 48.79 µWh	-74.4%
Vitis (unopt.)	1.24 s ± 6.86 ms	0.0%	1.04 s ± 0.18 ms	0.0%	28.44 W ± 2.63 W	0.0%	9.81 mWh ± 0.91 mWh	0.0%
Vitis (opt. 1)	0.94 s ± 4.75 ms	-24.3%	0.91 s ± 5.49 ms	-12.4%	31.64 W ± 2.52 W	+11.2%	8.27 mWh ± 0.66 mWh	-15.7%
Vitis (opt. 2)	0.88 s ± 7.67 ms	-29.5%	0.86 s ± 7.73 ms	-17.6%	32.41 W ± 0.41 W	+14.0%	7.89 mWh ± 0.12 mWh	-19.6%
Vitis (opt. 3)	0.99 s ± 2.47 ms	-19.9%	0.97 s ± 2.53 ms	-6.6%	32.41 W ± 0.35 W	+13.9%	8.95 mWh ± 0.10 mWh	-8.8%
Vitis (opt. 4)	1.29 s ± 7.65 ms	+4.2%	1.27 s ± 7.55 ms	+22.2%	32.74 W ± 0.41 W	+15.1%	11.77 mWh ± 0.16 mWh	+20.0%
Vitis (opt. 5)	0.91 s ± 3.46 ms	-26.7%	0.88 s ± 3.38 ms	-14.8%	32.50 W ± 0.41 W	+14.3%	8.21 mWh ± 0.11 mWh	-16.3%

Version	Matrix dimensions: $10^4 \times 10^4$							
	Wall time		Kernel time		Power usage		Energy usage	
	value	change	value	change	value	change	value	change
VexCL	1.05 s ± 0.15 s	-78.9%	N/A	-	49.22 W ± 0.97 W	+77.4%	14.32 mWh ± 2.11 mWh	-62.5%
Vitis (unopt.)	4.95 s ± 32.18 ms	0.0%	4.02 s ± 0.38 ms	0.0%	27.74 W ± 1.92 W	0.0%	38.19 mWh ± 2.65 mWh	0.0%
Vitis (opt. 1)	3.76 s ± 33.57 ms	-24.1%	3.61 s ± 42.98 ms	-10.0%	32.38 W ± 1.50 W	+16.7%	33.82 mWh ± 1.59 mWh	-11.4%
Vitis (opt. 2)	3.42 s ± 37.95 ms	-31.0%	3.34 s ± 41.25 ms	-16.9%	32.55 W ± 0.40 W	+17.3%	30.92 mWh ± 0.51 mWh	-19.0%
Vitis (opt. 3)	3.97 s ± 6.11 ms	-19.9%	3.87 s ± 6.16 ms	-3.6%	32.47 W ± 0.39 W	+17.0%	35.78 mWh ± 0.43 mWh	-6.3%
Vitis (opt. 4)	5.12 s ± 16.52 ms	+3.4%	5.03 s ± 19.55 ms	+25.2%	32.81 W ± 0.43 W	+18.3%	46.69 mWh ± 0.63 mWh	+22.3%
Vitis (opt. 5)	3.64 s ± 18.83 ms	-26.6%	3.54 s ± 16.35 ms	-11.9%	32.53 W ± 0.42 W	+17.3%	32.85 mWh ± 0.46 mWh	-14.0%

Table A.3: Test results of larger matrices from the experiments on the affine transformation application.

Version	Matrix configuration: size: 512×512 density: 0.01									
	Wall time		Kernel time (phi)		Kernel time (spmat)		Power usage		Energy usage	
	value	change	value	change	value	change	value	change	value	change
VexCL	2.77 ms ± 0.13 ms	-56.1%	N/A	-	N/A	-	50.79 W ± 0.29 W	+56.6%	39.13 µWh ± 1.81 µWh	-31.3%
Vitis (unopt.)	6.32 ms ± 3.02 ms	0.0%	3.42 ms ± 2.94 ms	0.0%	1.10 ms ± 36.48 µs	0.0%	32.42 W ± 4.80 W	0.0%	56.96 µWh ± 28.79 µWh	0.0%
Vitis (opt. 1)	4.54 ms ± 0.79 ms	-28.2%	1.65 ms ± 0.84 ms	-51.7%	1.27 ms ± 27.32 µs	+15.3%	30.82 W ± 4.56 W	-5.0%	38.88 µWh ± 8.95 µWh	-31.7%

Version	Matrix configuration: size: 5000×5000 density: 0.01									
	Wall time		Kernel time (phi)		Kernel time (spmat)		Power usage		Energy usage	
	value	change	value	change	value	change	value	change	value	change
VexCL	14.74 ms ± 0.11 ms	-61.3%	N/A	-	N/A	-	50.48 W ± 0.44 W	+58.0%	0.21 mWh ± 2.33 µWh	-38.9%
Vitis (unopt.)	38.13 ms ± 0.52 ms	0.0%	1.84 ms ± 0.28 ms	0.0%	19.96 ms ± 38.96 µs	0.0%	31.95 W ± 5.08 W	0.0%	0.34 mWh ± 53.99 µWh	0.0%
Vitis (opt. 1)	37.04 ms ± 0.46 ms	-2.9%	0.98 ms ± 0.21 ms	-46.5%	20.15 ms ± 99.43 µs	+0.9%	30.69 W ± 4.67 W	-3.9%	0.32 mWh ± 48.26 µWh	-6.7%

Table A.4: Test results of smaller matrices from the experiments on the sparse matrix-vector multiplication application.

Version	Matrix configuration: size: $10^5 \times 10^5$ density: 0.01									
	Wall time		Kernel time (phi)		Kernel time (spmat)		Power usage		Energy usage	
	value	change	value	change	value	change	value	change	value	change
VexCL	6.80 s ± 0.24 s	-37.4%	N/A	-	N/A	-	51.75 W ± 4.71 W	+84.8%	97.74 mWh ± 9.52 mWh	+15.6%
Vitis (unopt.)	10.87 s ± 65.33 ms	0.0%	15.55 ms ± 0.37 ms	0.0%	5.67 s ± 0.33 ms	0.0%	28.00 W ± 2.32 W	0.0%	84.52 mWh ± 7.01 mWh	0.0%
Vitis (opt. 1)	10.27 s ± 0.16 s	-5.5%	1.70 ms ± 0.29 ms	-89.1%	4.46 s ± 87.70 ms	-21.3%	36.08 W ± 2.29 W	+28.9%	0.10 Wh ± 6.72 mWh	+21.8%

Version	Matrix configuration: size: 12500×12500 density: 0.64									
	Wall time		Kernel time (phi)		Kernel time (spmat)		Power usage		Energy usage	
	value	change	value	change	value	change	value	change	value	change
VexCL	8.02 s ± 0.27 s	-42.7%	N/A	-	N/A	-	52.39 W ± 6.19 W	+87.9%	0.12 Wh ± 14.37 mWh	+7.6%
Vitis (unopt.)	14.01 s ± 0.10 s	0.0%	2.96 ms ± 0.21 ms	0.0%	7.49 s ± 0.26 ms	0.0%	27.88 W ± 2.18 W	0.0%	0.11 Wh ± 8.51 mWh	0.0%
Vitis (opt. 1)	11.78 s ± 0.28 s	-15.9%	1.12 ms ± 0.23 ms	-62.1%	4.52 s ± 71.21 ms	-39.6%	36.22 W ± 2.31 W	+29.9%	0.12 Wh ± 8.06 mWh	+9.3%

Version	Matrix configuration: size: $10^6 \times 10^6$ density: 10^{-4}									
	Wall time		Kernel time (phi)		Kernel time (spmat)		Power usage		Energy usage	
	value	change	value	change	value	change	value	change	value	change
VexCL	6.62 s ± 0.28 s	-46.9%	N/A	-	N/A	-	52.47 W ± 5.68 W	+88.3%	96.53 mWh ± 11.24 mWh	-0.0%
Vitis (unopt.)	12.47 s ± 32.88 ms	0.0%	0.14 s ± 0.22 ms	0.0%	7.34 s ± 0.27 ms	0.0%	27.87 W ± 2.09 W	0.0%	96.56 mWh ± 7.25 mWh	0.0%
Vitis (opt. 1)	11.74 s ± 77.10 ms	-5.9%	7.00 ms ± 0.42 ms	-95.1%	6.19 s ± 62.30 ms	-15.7%	36.17 W ± 2.19 W	+29.8%	0.12 Wh ± 7.18 mWh	+22.2%

Table A.5: Test results of larger matrices from the experiments on the sparse matrix-vector multiplication application.