Vrije Universiteit Amsterdam

University of Amsterdam

Master Thesis

Developing a BLAS library for the AMD AI Engine

Author: Tristan Laan (2752022)

1st supervisor: dr. Tiziano De Matteis
2nd reader: dr. Ana Lucia Varbanescu

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

20th September 2024

Abstract

Spatial dataflow computer architectures allow us to remove control logic overhead seen in traditional computer architectures such as central processing units (CPUs), by determining the control at compile-time. Many of the spatial dataflow architectures are marketed as artificial intelligence (AI) accelerators, and offer integration with common AI frameworks. However for general computations, they require the programmer to learn new complex device specific programming languages. The lack of support from high-level libraries make them hard to use as high-performance computing (HPC) accelerator.

In this thesis we present AIEBLAS, a Basic Linear Algebra Subprograms (BLAS) library for the AMD/Xilinx AI Engine (AIE), which can compile full dataflow designs consisting of chained BLAS routines for the AIE. The library offers an expandable core which makes it possible for new operations and optimizations to be implemented, and the main design principles of the library apply to other spatial dataflow architectures as well.

We evaluate the designs generated by AIEBLAS by comparing the performance to OpenBLAS, a popular BLAS implementation for CPUs, as well as evaluating the performance of several optimizations integrated into the library.

We conclude that our library provides a new way to program AMD/Xilinx AI Engines, without requiring a deep understanding of the underlying programming model.

Contents

Acronyms

ACAP Adaptive Compute Acceleration Platform. 5

ADF adaptive dataflow. 10–12, 15, 32, 35, 51, 54

AI artificial intelligence. ii, 1, 5, 6, 16, 53

AIE AI Engine. ii, 2, 3, 5–7, 9–12, 15, 16, 19–21, 23, 25–27, 30–35, 37–44, 47–49, 51–54

API application programming interface. 7, 9, 10, 12, 19, 25, 38, 52

AXI 4 Advanced eXtensible Interface 4. 6, 7, 9

BLAS Basic Linear Algebra Subprograms. ii, 1–3, 15, 16, 19, 20, 22, 23, 25, 27, 31–35, 37, 38, 47, 48, 52–54

BRAM block random access memory. 5

CPU central processing unit. ii, 1–3, 15, 37–39, 44, 47–49, 52–54

DRAM dynamic random access memory. 5

DSP digital signal processing. 5

FF flip flop. 5

FPGA field-programmable gate array. 2, 5, 7, 12, 15, 16, 37, 38, 40, 42, 43

GeMM General Matrix Multiply. 16, 21

GeMV General Matrix-Vector Multiplication. 21, 52

GPU graphics processing unit. 1, 15

Acronyms

HDL hardware description language. 12

HLS high-level synthesis. 12, 15, 20

HPC high-performance computing. ii, 53

JSON JavaScript Object Notation. 22, 23, 52, 54

LUT look-up table. 5

ML machine learning. 1

PL programmable logic. 5, 7, 9–12, 15, 16, 20, 21, 23, 27, 31, 32, 35, 38, 39, 42, 44, 47, 51, 52, 54

PRNG pseudorandom number generator. 48

VLIW very long instruction word. 6

XO Xilinx object. 15, 35

XRT Xilinx Runtime library. 23, 38, 40, 51, 52

1

Introduction

For decades we have relied on Moore's law [1] (which predicts that the number of transistors in integrated circuits approximately doubles every two years) to improve the performance of computation on central processing units (CPUs). This has largely been possible due to Dennard scaling [2], a scaling law which states the power density should stay constant as transistors get smaller. However researchers have observed that Dennard scaling no longer applies to current technological advancements and that we have reached the end of Moore's law [3, 4]. Horowitz shows that modern CPUs have a very high energy overhead due to their programmable architecture, spending over 50% of the energy on control logic [5]. To combat this, hardware manufacturers have started moving to more specialized computer architectures. An example of this are spatial dataflow architectures, which aim to move some of the logic that can be computed in advance to compile-time [6].

The new dataflow architectures often specifically aim the acceleration of artificial intelligence (AI) workloads, such as machine learning (ML). We would however ideally also be able to use these architectures for more general computational use-cases as well. To program the devices, manufacturers often offer integration with popular high-level AI frameworks (e.g. PyTorch [7] and TensorFlow [8]), however these frameworks are hard to use for non-AI use-cases. The manufacturers also tend to offer a custom programming platform which provide specialized functionality to utilize the full capabilities of the device. While these programming platforms are often capable to program the device for general use-cases, they are often hard to use and require in-depth knowledge from the program. It would be useful to have a more generalized library than the AI frameworks, without the programmer having to learn the custom programming platform.

The Basic Linear Algebra Subprograms (BLAS) specification is a popular generalized linear algebra library [9]. There are many BLAS implementations available for CPUs and graphics processing units (GPUs), such as OpenBLAS [10], BLIS [11] and cuBLAS [12]. A BLAS library would offer a more generalized library than most AI frameworks, while still being easy for a programmer to use. However currently no BLAS libraries exist for most dataflow architectures.

The AI Engine (AIE) developed by AMD is an example of a dataflow architecture. AMD has released several devices with an embedded AIE, including laptop and desktop CPUs, as well as field-programmable gate arrays (FPGAs). In this thesis we will investigate whether it is possible to develop an effective BLAS library for the AIE using an AMD Versal VCK5000. Our goal for the BLAS library is to (1) use a dataflow approach to utilize the benefits provided by the dataflow architecture of the AIE, (2) be easy for users to use without requiring deep knowledge of the underlying tools, (3) be easily expandable with new functionality and optimizations, and (4) have the concepts behind the library be reusable for other dataflow architectures.

1.1 Research questions

To develop our BLAS library (AIEBLAS) we will first have to determine a set of design choices that ensure the libraries usability and expandability. Thus the first research question we will investigate is:

[RQ1] Which design choices ensure a usable and expandable BLAS library targeting a spatial dataflow architecture?

To accomplish this, we look at how we can map the BLAS routines to a dataflow model, and determine important design choices to make sure we can fully utilize the dataflow architecture. We will also determine an input interface for the BLAS library, which is both easy to use for users of the library, and can be easily expanded.

With the design requirements in place we can start creating the AIEBLAS library. To guide us through this process, we will investigate the next research question:

[RQ2] How can we automatically generate a dataflow program consisting of BLAS routines for an AI Engine from a high-level specification?

Here we will focus on how we can automate the process, such that the only input required from the user is the high-level specification, but also how we can follow the design goals and keep the design expandable and eaasy to use.

To ensure the capabilities of the dataflow architecture are properly utilized, we will research what optimizations we can apply to the code generation, by answering the research question:

[RQ3] What optimizations can we apply to the kernel generation of BLAS routines to make the BLAS routines more performant on an AIE?

Here we will investigate both general optimizations that can be applied to any BLAS routine, as well as routine-specific optimizations.

Lastly to evaluate the effectiveness of AIEBLAS, we will compare the performance of the AIEBLAS library to the performance of other CPU BLAS libraries, and answer the question:

[RQ4] How performant is the AIEBLAS library compared to other BLAS libraries when performing common routines?

We will use the CPU BLAS library OpenBLAS, to compare the performance of AIEBLAS routines running on the AIE to the same routines running on the CPU.

1.2 Main contributions

In this thesis we make the following contributions:

1. We explain the AIE toolchain and how BLAS routines can be mapped to the AIE tiles in a dataflow program.
2. We present a new open-source BLAS library AIEBLAS for the AIE, which allows programmers to calculate general numerical computations on the AIE without deep knowledge of the hardware and software tools.
    (a) The library is easily expandable with new functionality and optimizations, and the concepts behind the library are reusable for other dataflow architectures.
3. We propose several optimizations that can be applied to BLAS routines running on the AIE.

4. We have evaluated the performance of AIEBLAS against OpenBLAS, a popular BLAS library for CPUs.
    (a) We have additionally evaluated the performance of some of the proposed optimizations by comparing them to their unoptimized counterparts.

1.3 Plagiarism declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, internet or machine), and has not been submitted elsewhere for assessment.

1.4 Acknowledgement

This work was supported in part by AMD under the Heterogeneous Accelerated Compute Clusters (HACC) program.

2

Background

2.1 VCK5000 and AI Engine

2.1.1 Hardware overview

The AMD Versal VCK5000 [13] is a heterogeneous hardware platform, part of the Versal Adaptive Compute Acceleration Platform (ACAP) device series. The VCK5000 consists of an FPGA with programmable logic (PL) and dynamic random access memory (DRAM), and a VC1902 AI Core with 400 AI Engine cores.

FPGA specifications

The resources available on the PL of the FPGA in the VCK5000 are show in Table 2.1. The look-up table (LUT)and flip flop (FF) units are used to store small amounts of data and logic, while the block random access memory (BRAM) units are used to store larger blocks of data on the FPGA. The digital signal processing (DSP) units are used for computing floating-point operations on the FPGA. [14]

Resource	Count
LUT	899,840
FF	1,799,680
DSP	1,968
BRAM	967

Table 2.1: Logic blocks available on the FPGA part of the VCK5000.

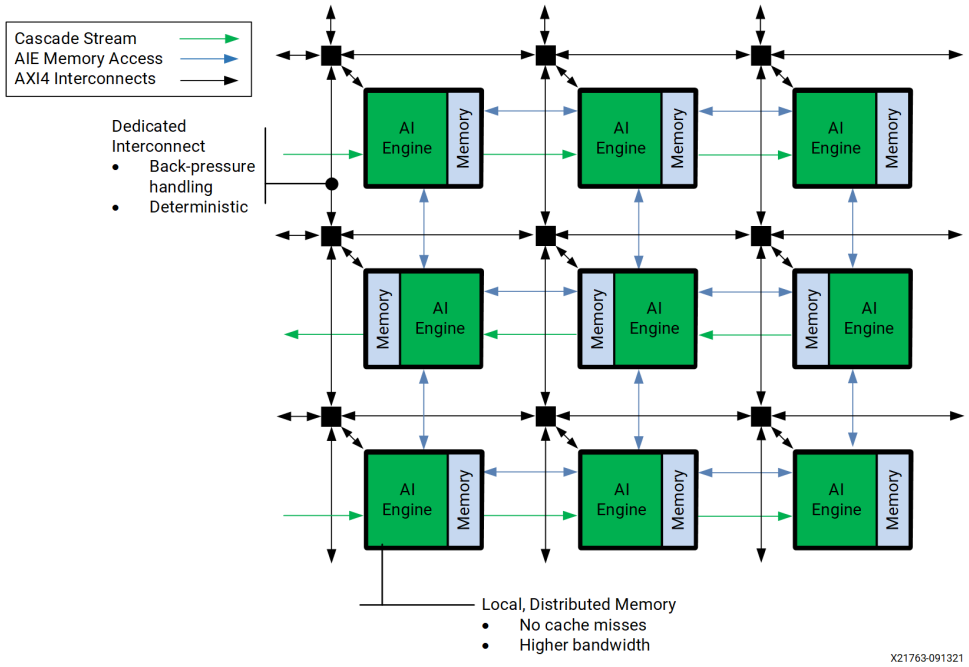


Figure 2.1: AI Engine array overview showing the AIE tiles in a 2D grid, with connections to neighbouring tiles. Figure from *Versal Adaptive SoC AI Engine Architecture Manual (AM009)*. [15]

The FPGA also has 16 GB of DRAM memory, which can be used to store large amounts of data, and can be accessed from the host device. This is mostly useful to store input and output data to be communicated with the host.

AI Engine specifications

The VC1902 AI Core contains an AIE array, consisting of a 2D 8×50 grid with AIE tiles. Figure 2.1 shows an overview of a part from the AIE array, showing nine of the AIE tiles. Each tile contains a very long instruction word (VLIW) processor (reffered to by AMD as an AI Engine), which contains one scalar unit, one vector unit, two load units, and one store unit. Each tile additionally has a memory module, containing eight memory banks of 4 KB, totalling to 32 KB of local memory on each AIE tile.

To send and receive data, each AIE tile contains three types of connections: (1) two cascade streams to connect to the accumulator of the left and right neighbouring AIE tiles, (2) four memory connections to the neighbouring memory blocks, and (3) Advanced eXtensible Interface 4 (AXI 4) interconnects connecting all the tiles in the AIE array.

The cascade streams are arranged in a snake like pattern going through all the tiles in the array. On the even rows it connects the AIE tiles from left to right, and on the odd rows

from right to left. Add the edges of the grid, where the last AIE tiles have no outgoing cascade stream, they are connected to the tile below them. Only the top left AIE tile has no incoming cascade stream, and the bottom right AIE tile has not outgoing cascade stream. The cascade stream allows for a 384-bit wide connection between the tiles, but it can only send data from the accumulator.

As can be seen in figure 2.1, the memory module is positioned on the right side of the tile in the even rows, and on the left side in the odd rows. Each AIE tile can access the memory of its own tile, and three neighbouring tile. It cannot access the memory from the tile on the other side of the memory module, meaning the even row tiles cannot access neighbours on the right, and the odd row tiles cannot access neighbours on the left. Each memory port supports up to a 256-bit wide connection.

The AXI 4 interconnects allow two input streams and two output streams to be set up in each AIE tile, where each stream is 32-bit wide.

2.1.2 Toolchain overview

To program the VCK5000, we need to take three parts into account. (1) Designing a kernel which will be mapped to a single AIE tile and performs a computation on a small chunk of data; (2) creating a data flow graph to create a chain of AIE kernels and providing a specification for the external data connections; and (3) creating PL kernels to move data between the host and the AIE array. [16]

Figure 2.2 shows how the software constructs map to the hardware platform described in section 2.1. Each instance of an AIE kernel is mapped to a single AIE tile, while the data flow graph corresponds to the whole AIE array and describes in which tiles the AIE kernels will be placed and which tiles will be connected with data streams. Lastly the PL kernel is mapped to the PL on the FPGA, connected to the AIE array by the PLIO.

AI Engine kernel

The AIE kernel is written in C++ with special intrinsics provided by an application programming interface (API), the AIE API. The API includes special vector data types and vector arithmetic functions to target the vector units on the AIE processors, which AMD notes as being important to achieve the highest performance on the AI Engine [16]. The data types and functions in the API are implemented in a C++ header-only library, which get translated into optimized intrinsic functions. The supported vector types and sizes are

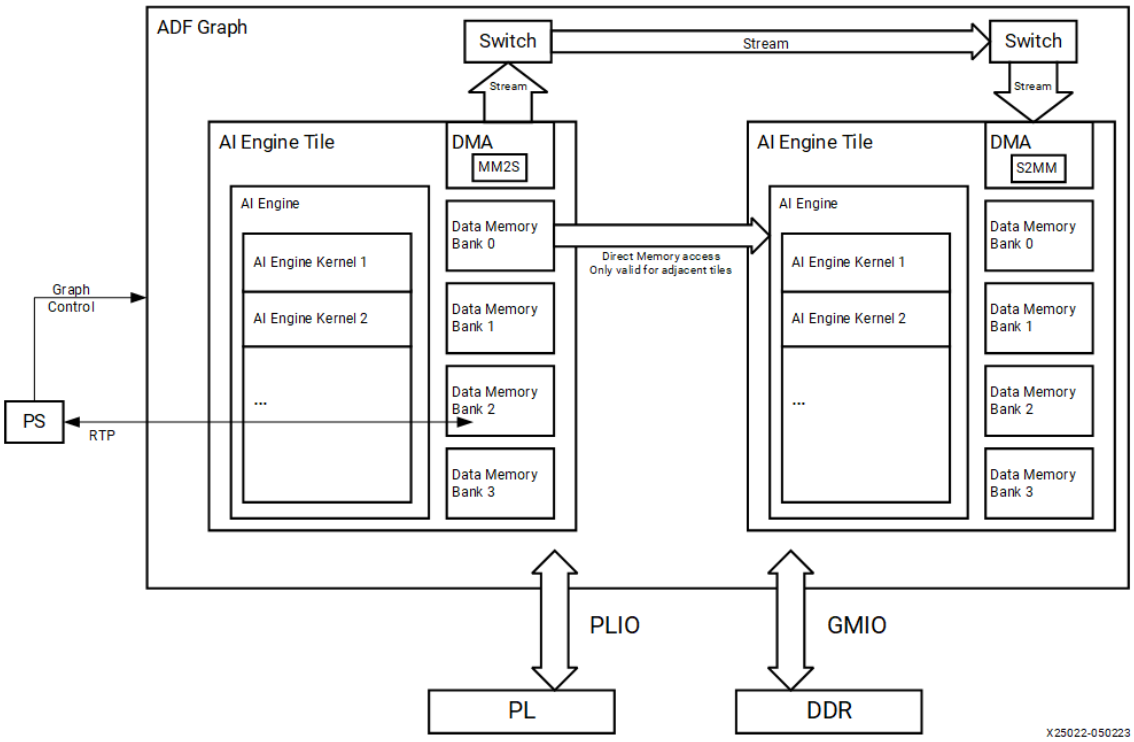


Figure 2.2: Diagram showing how the software constructs used by the AIE toolchain map to the AIE hardware. Figure from *AI Engine Kernel and Graph Programming Guide (UG1079)*. [16]

type	size	type	size
(u)int8	16/32/64/128	float	4/8/16/32
int16	8/16/32/64	cint16	4/8/16/32
int32	4/8/16/32	cint32	2/4/8/16
int64*	2/4/8/16	cfloat	2/4/8/16

* Emulated type which is not supported by all arithmetic functions

Table 2.2: Vector types and sizes supported by the AI Engine.

listed in table 2.2. We can see that 64-bit data types are not natively supported, and that vectors have a minimum size of 128-bits.

The `aiecompiler`, which compiles the AIE program to a design that can run on the AIE array, internally uses the Chess compiler by Synopsys, which provides additional special compiler pragmas that offer more control over loop scheduling, memory alignment, memory sequencing, and conditional tuning. [17]

Exchanging data To exchange data between AIE tiles in the array and to exchange data over PLIO, windows or streams can be added as arguments to the kernel. The streams provide direct synchronized access to external data using the AXI 4 interconnects described in section 2.1. A stream can either be an input stream to read data, or an output stream to write data. If the kernel kernel tries to read from an empty input stream it can stall, and likewise it can also stall if the kernel tries to write to a fully buffered output stream.

Windows on the other hand use buffers in the memory of the AIE tiles and provide the kernel with a chunk of data, where the size of the chunk is declared in the data flow graph. If possible, the windows will use the direct memory connections described in section 2.1 to access the data in the buffer. If the tiles exchanging data do not have a direct memory connection, a ping-pong model is used, where both tiles have their own buffers and once a buffer gets filled it uses the AXI 4 interface to send the data to the buffer of the receiving tile.

Like the streams, a window can either be an input window to read data, or an output window to write data to. If the kernel contains one or multiple input windows as argument, the kernel will wait until all the input windows are filled with a chunk of data before it starts executing. Once the kernel starts executing it can read and write data from the window, and only once the kernel has finished the data in the output windows is send to the external connection. The windows are designed to be accessed sequentially, and the

kernel keeps track of the current position in a given window. Using the AIE API, we can use special window functions to increment or decrement this position.

The main difference between the stream-based and window-based approach for streaming data to the AIE kernel is that in the window-based approach, the AIE kernel gets incoming blocks of data from the input windows, from which it produces new blocks of data to the output windows, while the AIE kernel can run indefinitely when using the stream-based approach.

```
1  // Number of elements in a window based on the width defined in the graph
2  #define NUM_SAMPLES 64
3  void vector_add(input_window<int32> *x, input_window<int32> *y,
4                  output_window<int32> *out) {
5     for (unsigned i = 0; i < NUM_SAMPLES / 4; ++i) {
6         aie::vector<int32, 4> vx = window_readincr_v<4>(x);
7         aie::vector<int32, 4> vy = window_readincr_v<4>(y);
8         aie::vector<int32, 4> vout = aie::add(vx, vy);
9         window_writeincr(out, vout);
10    }
11 }
```

Listing 1: Example vector add kernel using the window-based approach.

Listing 1 shows an example kernel using the window-based approach to calculate a vector-add. On lines 6 and 7 we see the `window_readincr` function being used to read a vector of 4 elements from the input windows, and the `incr` parts means this function will increment the position in the window by 4. On line 8 the AIE API is used to perform a vector add on two vectors of size 4, which is stored in vout. Lastly on line 9 `window_writeincr` is used to write the 4 elements of the vector to the output window, again incrementing the position by 4 elements afterwards.

Data flow graphs

The adaptive dataflow (ADF) graph defines how the external interfaces of the AIE kernels are connected. The AIE API provides a special header for the ADF graph containing a base specification for a Graph class. To create our own ADF graph, we have to subclass this Graph class. This class defines the kernel instances that will be mapped into the AIE array,

the external PLIO connections the AIE kernels will connect to, and the internal data streams between the AIE kernels. This graph definition will be parsed by the `aiecompiler` to extract the graph definition to create a floorplan for the AIE array, but also as an entry point to extract information such as the source code location for the kernels that need to be compiled.

Kernel declaration To add a kernel to our graph definition, we need to add a kernel object provided by the ADF header to the graph definition. In Listing 2 we show a basic definition providing a single instance of the vector add kernel defined in Listing 1. In the class constructor on lines 7–8, we can see that we initialize the kernel object by calling the `kernel::create` function and passing the C++ function prototype of the vector add kernel as an argument. We also specify the source file location containing the kernel implementation itself. Lastly we specify the runtime ratio of the kernel. Multiple kernels can be mapped onto the same AIE tile, but each tile can only execute a single kernel at the same time. The runtime ratio is used to determine how many cycles the kernel is allowed to run before it is swapped out for the next kernel. The runtime ratio must be between zero and one.

```
1  using namespace adf;
2  class simpleGraph : public graph {
3  private:
4      kernel vadd;
5  public:
6      simpleGraph() {
7          vadd = kernel::create(vector_add);
8          source(vadd) = "kernels/vadd.cpp";
9          runtime<ratio>(vadd) = 0.9;
10     }
11 }
```

Listing 2: An incomplete graph definition containing a single vector add kernel.

Connecting the kernels To specify the data flow in the AIE array we have to specify data connections in the graph definition. In Listing 3 we extend the graph definition from

Listing 2 by connecting the inputs and outputs of the vector add kernel to PLIO. On lines 5–7 we add PLIO objects to the kernel, which we initialize in the constructor on lines 14–16. For each PLIO object we specify the bus width of the connection, which can be 32-, 64-, or 128-bits [18], as well as a path to a data file. This data file is used during simulation to feed data to the AIE in the absence of an actual PLIO connection.

On lines 17–19 connect objects are used to connect the PLIO to the kernels. The template argument specifies the type of connection is used, this can be `stream` when using input/output streams, or `window` when using input/output windows. When using windows, the window size is also defined using the template argument. The constructor of the connect object takes two arguments; the first argument specifies the outgoing data connection, while the second argument specifies the incoming connection. Each kernel and PLIO object has an `in` and `out` array containing a list of inputs and outputs, which are used as arguments to the connect constructor. The PLIO objects only contain a single input for input PLIOs and a single output for output PLIOs, while for kernels the inputs and outputs are determined by the kernel arguments. For example `vadd.in[1]` on line 18 refers to the second input argument of the vector add kernel, which corresponds to the input window `y` in Listing 1.

Kernel mapping By default the `aiecompiler` will automatically create a floorplan for the AIE array based on the graph definition. The ADF API however also provides mechanisms for manually adding location constraints in the graph definition. For example to specify that the vector add kernel should be placed on tile $(25, 0)$, we could add the assignment `location<kernel>(vadd) = tile(25, 0)` to the graph constructor. Similarly the function `location<buffer>` can be used to limit a buffer placement to a specific memory bank in an AIE tile.

Data movement between host and AI Engine

To move data to the AIE array on the VCK5000, PL kernels can be used. PL kernels live on the programmable logic of the FPGA inside the VCK5000, and can be written in hardware description language (HDL) languages such as Verilog and VHDL, or when using high-level synthesis (HLS) tools such as Xilinx Vitis in C++. It is possible to use the PLIO interface to integrate an AIE design into a larger FPGA program, but for sending data from the host to the AIE array, a simple HLS PL kernel can be used.

Listing 4 shows a simple PL kernel for sending an array `mem_x` to a HLS stream, which can later be connected to a PLIO connection in the AIE design. On lines 10–12 an element is

```
using namespace adf;
class simpleGraph : public graph {
private:
    kernel vadd;
    input_plio x, y;
    output_plio out;
public:
    simpleGraph() {
        vadd = kernel::create(vector_add);
        source(vadd) = "kernels/vadd.cpp";
        runtime<ratio>(vadd) = 0.9;

        x = input_plio::create(plio_32_bits, "data/x.txt");
        y = input_plio::create(plio_32_bits, "data/y.txt");
        out = output_plio::create(plio_32_bits, "data/out.txt");
        connect<window<256>> net0(x.out[0], vadd.in[0]);
        connect<window<256>> net1(y.out[0], vadd.in[1]);
        connect<window<256>> net2(vadd.out[0], out.in[0]);
    }
}
```

Listing 3: A graph definition containing a single vector add kernel with data connections.

```
1  void mm2s(ap_int<32> *mem_x, int size,
2            hls::stream<qdma_axis<32, 0, 0, 0>> &stream_x) {
3  #pragma HLS interface m_axi port = mem_x offset = slave
4  #pragma HLS interface axis port = stream_x
5  #pragma HLS interface s_axilite port = mem_x bundle = control
6  #pragma HLS interface s_axilite port = size bundle = control
7  #pragma HLS interface s_axilite port = return bundle = control
8      for (int j = 0; j < size; j++) {
9          #pragma pipeline II = 1
10         qdma_axis<32,0,0,0> x;
11         x.data = mem_x[j];
12         x.keep_all();
13         stream_x.write(x);
14     }
15 }
```

Listing 4: A PL kernel that can send data from the host to a PLIO connection in the AIE array.

prepared to be send over the stream on line 13. The statement on line 11 sets the content of the element to the current data point in the `mem_x` array, while the `keep_all` call on line 12 marks that all 32-bits of the element are valid and should be read on the other end of the stream.

Compilation scheme

To compile an AIE design to be run on the VCK5000, first the ADF graph needs to be compiled by `aiecompiler`. Only the graph source file where an instance of the graph design is initialized needs to be provided. The compiler will automatically extract the relevant kernel files that are part of the design. The `aiecompiler` will first compile the kernel files to byte code the AIE tiles can understand. The compiler will then run a global placement algorithm to determine the mapping of the kernels to AIE tiles, the placement of buffers on the memory banks in the tiles, and the stream connections between the tiles. The whole specification, together with the byte code for the AIE tiles then gets packaged to a single static library called `libadf.a` by default.

The HLS PL kernels can be compiled conventionally using the `v++ compiler` [19] to Xilinx object (XO) objects. The AIE specification in `libadf.a` and the XO objects can then be linked into an FPGA design for the VCK5000 using the `v++` compiler together with a connectivity specification. The connectivity specification specifies which PL kernels will be part of the design, where the PL kernels will be located on the FPGA, and how the PL kernels will be connected to the AIE.

2.2 BLAS

The BLAS specification [20] describes a set of routines containing commonly used low-level dense linear algebra operations. The aim of the routines is to provide a pre-defined set of basic building blocks to compute basic vector and matrix operations, which can be used to develop higher-level linear algebra libraries, such as LAPACK [21]. This design has made it possible for software vendors to develop BLAS implementations optimized for specific computer architectures, which can be used as backend for the higher-level libraries. [9]

The BLAS routines are divided in three levels: Level 1 contains scalar, vector and vector-vector operations, Level 2 contains matrix-vector operations, and Level 3 contains matrix-matrix operations.

There are many BLAS implementations available for different computer architectures. For CPUs there are popular open-source implementations such as OpenBLAS by Zhang et al. [10], as well as implementations from hardware vendors, like Intel oneMKL [22] and AMD AOCL [23]. For GPUs, NVIDIA provides a CUDA implementation called cuBLAS [12]. For FPGAs, AMD provides the Vitis BLAS [24] library for AMD Alveo FPGAs and De Matteis et al. present a BLAS implementation available for Intel FPGAs called FBLAS [25].

2.3 Related work

There are multiple state of the art implementations of the Level 3 General Matrix Multiply (GeMM) routine. Lei et al. show a design utilizing both the PL and AIE array of the VCK190[1], and reaching up to 70% of the AIE tiles peak performance [26]. Taka et al. similarly show an AIE design achieving a throughput of up to 77 TOPs on the AIE [27]. However these works only cover a single BLAS routine and do not aim to automatically generate the routines based on the user requirements, differentiating them from the work in this thesis. They can however prove useful to integrate into the AIEBLAS library.

Zhuang et al. propose the *CHARM* framework, which can automatically generate AIE designs that can compute matrix-matrix multiplications. The original version of *CHARM* uses a text configuration file to specify the user configuration for the code generation [28], while *CHARM* 2.0 offers a Python API to configure the code generator [29]. Using 32-bit floating-point datatypes, they reach up to 2.94 TFLOPS inference throughput for multiple machine learning tasks. While the project aims to be easy to use without requiring a deep understanding of the AIE toolchain, it only focuses on matrix-matrix multiplications, heavily focusing on AI workloads. It cannot be used as a full numerical library like BLAS.

Zhang et al. describe a framework called *EA4RCA* to generate AIE designs for communication avoiding applications, a specific set of applications which have a minimal amount of data movements to accomplish the task [30]. While this described framework also generates AIE code, the project has a different focus than AIEBLAS. *EA4RCA* aims to maximize the performance of AIE code, and provide a different method for programming the AIE. It does not aim to provide a high-level library that can be integrated into existing code, like AIEBLAS.

[1] The VCK190 is an embedded board that uses the same PL area and AIE array that is on the VCK5000.

Heinz et al. provide integration for the VCK5000 in the existing *TaPaSCo* framework, providing support for the AIE as well. The *TaPaSCo* framework partially automates the development FPGA designs, and allows the integration of a AIE graph definition. However the framework does not automate any of the development of the AIE graph itself.

3

Design

In this chapter we explore the design considerations involved in developing our BLAS implementation AIEBLAS using the AMD AI Engine.

3.1 BLAS interface

We will have to decide how to adapt the BLAS interface to fit in an AIE design. We will have to make sure that we are able to utilize the spatial dataflow architecture of the AIE. Additionally we will have to decide which routines to include in the initial version of AIEBLAS.

3.1.1 Mapping a BLAS routine to an AI Engine kernel

All BLAS routines follow a general format where the first arguments describe the dimensions of the operation, the input vectors are specified by a memory location paired with the stride of the elements, and the result will be stored in one of the incoming vectors or matrices. For example if we look at the `axpy` kernel computing the operation $\alpha x + y$, the routine signature is `XAXPY(N, ALPHA, X, INCX, Y, INCY)`, where the result will be stored in the parameter `Y`.

In the dataflow model of the AI Engine, kernels are expected to read incoming blocks of data and produce new outgoing blocks of data; it is not possible to write the output of a routine back to an input stream or input window. In our AIE design we will thus have to deviate from the BLAS specification and produce a new data stream for the output result

of a routine. Note that a host API could be adapted to store this new result buffer into the correct buffer specified by the BLAS specification.

Separating the outgoing data from the incoming data comes with the additional benefit that it will make it easy for us to create chains of routines in a large pipeline, by chaining the output of a routine to the input of another routine.

We will need to choose between streams and windows to send data to the AIE kernels. Since some of the second level BLAS routines will require us to use data in a vector multiple times, we will use windows to send vectors and matrices to the routines on the AIE. For the scalar values we will also need to use streams or windows, since the VCK5000 does not allow us to directly pass parameters from the host to the AIE itself, and we can only write HLS streams from PL kernels to the AIE. Windows have a minimum size of 16 bytes, so if we used windows for scalar values we would have to pad the values to 16 bytes. Since we can store the scalars inside the kernels itself, and thus we do not require the feature of windows being able to re-read values from the buffer, it is more sensible to use streams to send scalar values to the AIE, which do not have a minimum size.

It is not good for performance in a dataflow architecture to have stride in the vector arrays, since this means that unused data is streamed to the kernels, so it is sensible to not implement stride support in the AIE itself. If stride support is required this would be better to implement in the PL kernels before the data gets send to the AIE array. This has the additional benefit of limiting the amount of scalar parameters that we need to add to the AIE kernels, since we can only use two input streams per kernel at most.

3.1.2 Supported BLAS routines

As specified in section 2.2, the BLAS interface contains three levels. In this subsection, we describe which routines will be supported in the initial version of AIEBLAS.

First level

The first BLAS level consists of 16 routines limited to vector and scalar operations. Most of the functions can be implemented on the AIE, however there are some routines which are not useful to implement on the AIE. Firstly there are two scalar only routines, `rotg` and `rotmg` that generate plane rotations. This can not be accelerated and is more sensible to compute on the host device. There is also a `copy` routine, which is intended to copy a data buffer. Since we use a data-flow model on the AIE the data lives as streams, so we

always output the results of a routine to a new output stream, defeating the purpose of a copy routine.

Second level

Due to time constraints we will only implement the `gemv` routine which computes a General Matrix-Vector Multiplication (GeMV) operation. This will demonstrate the ability of running second level functions on the AIE, while the expandable design of AIEBLAS allows the library to be extended with more second level functions if required.

Third level

The third level consists only of matrix-matrix routines which require complex designs to run on spatial architectures. We will not cover these complex designs in this thesis. However as mentioned in section 2.3 there are already multiple implementations of GeMM routines available for the AIE. If this routine is desired, it would be possible to integrate the existing work into AIEBLAS.

3.2 AIEBLAS design

In this section we describe the general design of the AIEBLAS code generation process, and highlight the important design choices to keep the code generation adaptable and make AIEBLAS easy to use.

The code generation will have to create a complete AIE design as described in section 2.1.2, which will require us to generate AIE kernels to execute the routines, a dataflow graph to connect the routines, PL kernels to interface with the host, and a build system to build the design.

3.2.1 User configuration

We need a specification for how the user will be able configure the AIEBLAS generated design. Here we should take three things into consideration:

1. The user configuration should be easy to write and should not require deep knowledge of the AIE architecture.
2. The user configuration should still allow details of the underlying implementation to be changed, such as the window sizes and vector instructions.

3. The design should be adaptable, such that it is easy for new options to be added to the configuration.

A natural fit for these requirements is the JavaScript Object Notation (JSON) format. The JSON format is easy for users to write, we can have optional fields for advanced options, and we can add new fields to the specification if we want to add new options to the configuration.

```
{"kernels": [
    {"blas_op": "scal",
     "user_name": "scale",
     "type": "int32",
     "vector_size": 8,
     "window_size": 256,
     "extra": {"alpha": 2}
    },
    {"blas_op": "dot",
     "user_name": "dot",
     "type": "int32"
    }
 ],
 "connections": [
    {"in": {"kernel": "scale", "parameter": "out"},
     "out": {"kernel": "dot", "parameter": "x"}
    }
 ],
 "platform": "xilinx_vck5000_gen4x8_qdma_2_202220_1",
 "profile": false
}
```

Listing 5: An example AIEBLAS JSON configuration containing a `scal` and a `dot` kernel.

In Listing 5 we show an example of a JSON file containing an AIEBLAS configuration for a `scal` and a `dot` kernel, where the output of the `scal` routine is used as one of the input vectors of the `dot` kernel.

To keep the configuration easy to use, the only required fields for each kernel are: (1) the `blas_op` to specify the BLAS routine to generate, (2) `user_name` to specify a unique

name for the kernel, and (3) `dtype` to specify the data type of the input and output data. However we still allow access to more advanced settings through optional fields, such as the window and vector sizes. If these fields are not set, they will use their default values. Each kernel can also use a special `extra` field to specify BLAS routine specific settings. For example in the example configuration, we set the value of the scalar α in the `scal` routine to the value 2.

The connection section of the specification is used to chain the outputs of kernels to inputs of other kernels in the dataflow graph. In the example configuration the output of the `scal` routine is set as input for the `dot` routine. If a connection is not listed in this specification, a PL kernel will need to be generated by AIEBLAS to handle this connection from the host.

3.2.2 Code generation

With the input format and the target output defined, we can create a code generator. We will write AIEBLAS in C++, since the object-oriented design of C++ is useful as a basis for our generator, and it makes it easy to integrate the library with the Xilinx Runtime library (XRT) which also uses C++. We specifically will use the C++23 standard, which allows us to use the new `print` and `format` libraries that are useful for our code generation. To ensure compatibility with older compilers, we use a special compatibility header to provide definitions for the `print` and `format` functions if they are not supported by the compiler. Since the `format` library is especially complex to implement, we will use the *{fmt}* library [32] as a fallback, which uses the same syntax as the C++20 `format` library.

The general structure of the code generator is shown in Figure 3.1. First the user specification will be parsed using a JSON parser. Since C++ does not have a native JSON parser, we use the *JSON for Modern C++* library [33] to create our parser.

The parsed user configuration gets passed to the main generator. To prevent duplication of code, the generator is split into two parts.

1. A main generator which implements most of the boilerplate code, which either is mostly static, or does not depend on a specific BLAS routine. Examples are the CMake build system and the graph specification.

2. A sub generator specification which specifies functions for generating routine-specific code. For each BLAS routine, we create a subclass of this specification, and implement the highly routine-specific code, such as the AIE kernel.

The sub generator design allows us to easily implement routine-specific optimizations, since the important routine-specific code is contained in a single subclass which does not influence the rest of the project. It also allows us to easily implement new BLAS-routines into the AIEBLAS library by implementing a new sub generator.

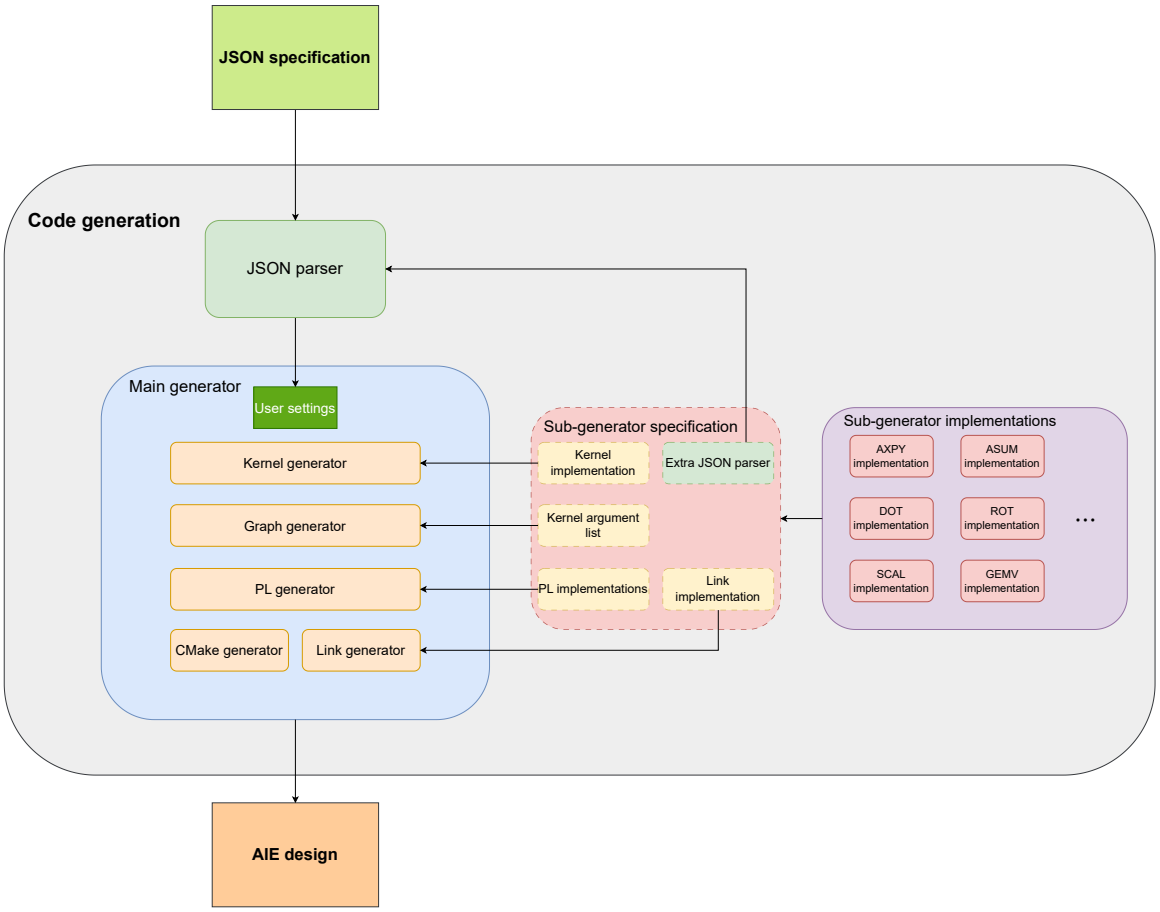


Figure 3.1: Overview of the code generation design in AIEBLAS.

4

Implementation

In this chapter we describe the implementation of AIEBLAS based on the design described in chapter 3.

4.1 Kernel Generation

We start with the AIE kernel generation for the BLAS operation. To store the requested kernel settings, we use a C `struct` that stores the requested BLAS operation, a user-defined name for the kernel, the datatype of the kernel, the size of the AIE vector units, and the window size. Based on these settings we can generate the BLAS routines.

4.1.1 Blueprints

Before we implement a code generator for the BLAS routines, we start by creating blueprint implementations of each BLAS operation. These blueprint implementations will be used as a basis for the code generation. We will highlight a few blueprints that show different challenges in porting BLAS operations to an AIE kernel.

Scale kernel

The first blueprint we present is a blueprint for the `scal` BLAS routine, which scales a vector by multiplying a scalar α with a vector x.

The blueprint for the `scal` routine is shown in Listing 6. Starting with the function signature on lines 6–7, we see that we have an input stream to stream the scalar α to the

```
1  #define NUM_SAMPLES 64
2  // 64 / 8
3  #define NUM_LOOPS 8
4  int32 chess_storage(%chess_alignof(v4int32)) alpha_storage[4] = {0,0,0,0};
5
6  void scal(input_stream<int32> *alpha, input_window<int32> *x,
7            output_window<int32> *out) {
8      int32 *alpha_store = &alpha_storage[0], *alpha_set = &alpha_storage[1];
9      if (*alpha_set == 0) {
10         *alpha_set = 1;
11         *alpha_store = readincr(alpha);
12     }
13     aie::vector<int32, 8> vx, vout;
14     aie::vector<int32, 8> scalar = aie::broadcast<int32, 8>(*alpha_store);
15     for (unsigned i = 0; i < NUM_LOOPS; i++) {
16         vx = window_readincr_v<8>(x);
17         vout = aie::mul(vx, scalar).to_vector<int32>();
18         window_writeincr(out, vout);
19 } }
```

Listing 6: `scal` routine, which scales a vector by multiplying a scalar α with a vector x.

kernel, an input window x to receive the incoming vector, and an outgoing vector *out* to write the result to.

Skipping over the initialization for now, we can see that the core of the function consists of a for-loop from lines 16–20. In each iteration a vector of eight integers is read from the input window, the `mul` function is used to multiply the vector with the scalar, and the result is written to the output window. We need to manually convert the output of the multiply function to a vector, since all multiplication functions of the AIE API return an accumulator. Accumulators use accumulator registers of the AIE which contain more bits than the vector registers, meaning intermediate results will have a higher precision before the end result is converted to a vector again. Since we are using 32-bit integers in this blueprint, the AIE will be using the 48-bit accumulator registers to store the result of the multiplication.

We can see that the for-loop runs for eight iterations, and a total of 64 integers will be

read from the input window, and written to the output window. Since we are using 32-bit integers, this means that the input and output windows will need to have a window size of $64 \times 32/8 = 256$ bytes. If we want to compute a scale operation on a vector of a 1024 elements, the kernel would need to be run for $1024/64 = 16$ invocations. This poses a challenge for reading the scalar α, since if we were to read a single element from the input stream on each invocation, we would need to re-stream the scalar to the kernel for each invocation. To prevent this we use the local memory of the AIE tile to cache the scalar value and only read from the stream on the first kernel invocation, as can be seen on lines 8–12. On line 4 we declare a global array that we use to cache the values. We use a special chess compiler directive to align the array to a 128-bit boundary, which is required if we want to use vector instructions on the array [16].

Dot kernel

The next interesting blueprint to look at is the `dot` BLAS routine, computing the dot product of two vectors of the same size.

The blueprint for the `scal` routine is shown in Listing 7. In the core of the function from lines 17–21, the kernel reads eight elements from each of the two incoming vectors, multiplies the corresponding elements of each vector, and adds the results to a global result variable.

The major difference between the `scal` and `dot` routine is that the `dot` routine reduces the output to a single scalar. This means the output can only be written to the output stream once the whole dot product is computed. To accommodate for this we will need to keep track of the amount of times the kernel has been executed, and how large the incoming vectors actually are. To keep track of the amount of kernel invocations we use a counter stored in the local memory which is incremented on each kernel invocation, as can be seen on line 22. To determine the size of the incoming vectors we add an additional input stream, such that we can stream the vector size from the host to the kernel via PLIO. On lines 23–25 we can see that the output is only written to the output stream on the last kernel invocation.

One more thing to note is that an aligned array of eight integers is used in vector calculations by casting the array to a vector on line 15, and storing it as reference in a new variable. This new variable can be used like a regular AIE vector as can be seen on line 20. At the end of the kernel on line 24 the result vector gets summed to a scalar using the `reduce_add` function.

```
#define NUM_SAMPLES 64
// 64 / 8
#define NUM_LOOPS 8
uint64 chess_storage(%chess_alignof(v4int64)) counter[4] = {0,0,0,0};
int32 chess_storage(%chess_alignof(v8int32)) result_storage[8]
        = {0, 0, 0, 0, 0, 0, 0, 0};

void dot(input_stream<uint64> *in_size_n, input_window<int32> *x,
         input_window<int32> *y, output_stream<int32> *out) {
    uint64 *num_cycles = &counter[0];
    uint64 *cycle = &counter[1];
    if (*num_cycles == 0) {
        *num_cycles = readincr(in_size_n) / NUM_SAMPLES;
    }
    aie::vector<int32, 8> &result = *(aie::vector<int32,8>*)result_storage;
    aie::vector<int32, 8> vx, vy;
    for (unsigned i = 0; i < NUM_LOOPS; i++) {
        vx = window_readincr_v<8>(x);
        vy = window_readincr_v<8>(y);
        result = aie::add(aie::mul(vx, vy).to_vector<int32>(), result);
    }
    *cycle += 1;
    if (*cycle == *num_cycles) {
        writeincr(out, aie::reduce_add(result));
    }
}
```

Listing 7: `dot` routine, which calculates the dot product of a vector x and y.

GeMV kernel

The last blueprint to look at is for the `gemv` routine, which calculates the the matrix-vector calculation $z \leftarrow \alpha Ax + \beta y$, where A is a matrix, x, y and z are vectors, and α and β are scalar values.

```
#define NUM_SAMPLES 64
// 64 / 8
#define NUM_LOOPS 8
uint64 chess_storage(%chess_alignof(v4int64)) counter[4] = {0, 0, 0, 0}

void gemv(input_window<int32> *A, input_window<int32> *x,
          output_window<int32> *out) {
    uint64 *cycle = &counter[0];
    if (*cycle == 0)
        window_acquire(x);
    if (*cycle % NUM_SAMPLES == 0)
        window_acquire(out);
    aie::vector<int32, 8> vx, vA;
    aie::vector<int32, 8> vout = aie::zeros<int32, 8>();
    for (unsigned i = 0; i < NUM_LOOPS; i++) {
        vA = window_readincr_v<8>(A);
        vx = window_readincr_v<8>(x);
        vout = aie::add(vout, aie::mul(vA, vx).to_vector<int32>());
    }
    window_decr_v4(x, NUM_LOOPS);
    window_writeincr(out, aie::reduce_add(vout));
    *cycle += 1;
    if (*cycle % NUM_SAMPLES == 0)
        window_release(out);
}
```

Listing 8: Shortened `gemv` routine, which calculates a matrix-vector multiplication, limited to matrices with a width of 64 elements.

A shortened version of the blueprint for the `gemv` routine is shown in Listing 8. Note that the implementation is limited to matrices that have the same width as the defined window size, which in this case matches 64 elements. Only the matrix-vector multiplication itself

is shown for brevity. While the core of the function from lines 15–18 is similar to the `dot` routine, this kernel is interesting to look at because in each kernel invocation a different amount of data is read from and written to the input and output windows.

In each kernel invocation a full row of the matrix A is read, advancing to the next row after the kernel is finished, and the row is multiplied by the vector x, with the same vector being used for the next kernel invocation. At the end of the kernel only a single value of the output vector has been computed, meaning the output window will only be filled after 64 iterations.

To summarize, we would want the input window A to advance by 64 elements after each kernel invocation, the input window x to not move at all[1], and the output window to move after each 64th invocation. However by default each window will advance to the next elements between each kernel invocations. To solve this, we can mark the windows we do not want to advance each iteration as asynchronous in the graph specification, which allows us to take control of when the window will advance to the next slice of data. Once we have marked a window as asynchronous, we need to manually acquire a window lock before we start reading or writing data from or to the window, and release the window once we want to advance to the next slice of data. On line 10 we can see that we acquire a lock for x on the first invocation and never release it, meaning the data from the vector x will always stay in the window. We can also see on lines 12 and 24 that the output window is acquired and released every 64th invocation, making sure the output window is filled before it advances.

4.1.2 Implementation

To automatically generate the AIE kernels we create a add three functions to the subgenerator specification described in section 3.2.2, which will be called by the main generator class. We create function to: (1) generate the global kernel code, (2) generate the kernel arguments, and (3) generate the kernel body. By splitting the kernel generation in three parts we can us the kernel argument generation to generate both the header function declaration, as well as the source function definition.

We have to slightly adapt the blueprints for the code generator to be able to generate different kernels based on the user settings. For example, if the user requests to use scalar

[1]Note that an alternative solution would be to stream the vector x to the kernel again for each invocation, but this would make it harder to chain kernels.

operations by setting the vector size to zero, the AIE specific datatypes and vector operations have to be replaced with native C++ scalar types and normal arithmetic operators.

4.1.3 Optimization

Restrict qualifier

AMD recommends using the C restrict qualifier when two pointers will not point to the same memory locations, to allow the compiler to perform more aggressive optimizations [18]. For example, in the `scal` routine in Listing 6 the compiler can not assume by default that the input window `x` and the output window `out` refer to separate memory locations, which would add additional memory dependencies in the main for-loop limiting the pipelining possibilities. If we add the restrict keyword to our kernel arguments, we can guarantee to the compiler that `x` and `out` will never point to the same memory location. In our code generation we thus make sure that the restrict qualifier is always added to the kernel arguments.

Compile-time scalar values

Most BLAS operations include scalar values to scale vectors or matrices, but in a lot of computations it is either not necessary to scale a vector or matrix, or the scale factor is known at compile time. To account for this we add additional kernel options for all kernels that have scalar inputs to embed the scalar value into the kernel, removing the stream required to send the scalar to the kernel and reducing the amount of resources required for the design. This is especially important because there is a limit to the amount PL kernels that can be placed on the FPGA, and each unused scalar value would add an extra PL kernel to the design.

Tiling `gemv` implementation

To calculate the matrix-vector product $y = \alpha Ax + \beta y$ in the `gemv` routine, the vector x will have to be re-iterated over for every row in A. If we have an x larger than the size of a window, we cannot keep the entire vector of x in view of the AIE kernel. If we use a simple calculation pattern, where we iterate through the matrix A row-wise, we would have to resend the vector for every row of A. This causes unnecessary data movement, which can impact the performance of the AIE routine. If we were to iterate through A column-wise, we would have a similar issue, but with the y vector.

A solution to this is to use a tiling interface, where instead of iterating through the matrix row-wise or column-wise, we divide the matrix up into tiles, where the height and width

of the tiles are equal to the window size of the kernel parameters. We can then calculate a the tiles row-wise to produce a single window of output data for each row of tiles. For each row inside a tile, we use the same part of the vector x, so instead of having to re-send vector x to the AIE for every row of A, we now only have to resend it for every row of tiles. If we use a 1024×1024 matrix and a window size of 64 as an example, we would have to resend x a 1024 times for the non-tiling approach, and only $1024/64 = 16$ times for the tiling approach.

To implement this tiling approach from the non-tiling approach, two changes have to be made. (1) The AIE kernel now has to store the intermediate results in an array of the same size as the window, instead of in a single value, since we are now computing a row of blocks at a time instead of a single row of A. (2) The PL kernel has to be modified to send the matrix A in a block-wise pattern, instead of the normal row-wise pattern.

4.2 Graph generation

In the ADF graph generation we need to declare and initialize all the kernels, declare PLIO connections, and connect the kernel parameters. We have a single function in the main generator class which generates graph. It first loops over the kernels in the kernel list and declares and initializes the kernels based on the user-specified kernel name.

For creating the PLIO connections and connecting the kernel parameters, we first need to know how many and which arguments each kernel has. To achieve this, the BLAS operation specific generators contain a function for obtaining an ordered list of the kernel arguments. The items in the list store information of the kernel argument, such as whether it is an input or an output, the name of the argument, whether it is a stream or window, and if the argument is asynchronous. While this does give us information on the general arguments, we do not know yet if the window or stream will be connected to PLIO or to another kernel. For this we use the connection map stored in the kernel struct. For each argument we can look up the name of the argument, retrieve whether it is connected to another kernel, and if so retrieve the name of the connecting kernel and argument. We also have a special disabled state which is used when scalar values are set at compile-time, and thus the input stream is unused.

For each kernel argument, if it is not connected to another we create a new PLIO object, which we initialize with a unique name by combining the kernel user name with the name of the argument.

chain length	Compilation time	
	manual partitioning	automatic partitioning
5	1m 49s	1m 50s
10	2m 17s	2m 18s
15	2m 20s	2m 36s
20	2m 48s	5m 55s
25	3m 17s	10m 43s
30	3m 22s	33m 22s
40	3m 55s	33m 57s
50	4m 58s	–

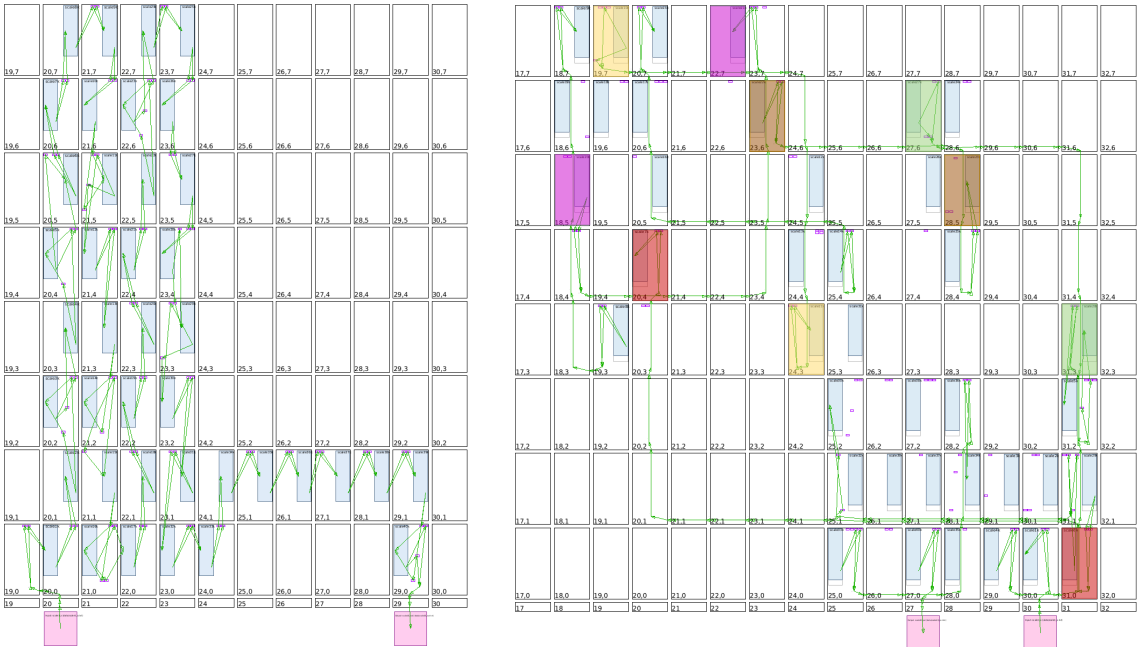
Table 4.1: AIE compilation time of chains of BLAS `scal` operations with and without a manually partitioned layout. A dash means the global placer of the compiler could not find a solution to generate a floorplan and the compilation failed.

To connect the kernel parameters we need to know the index of the input and output arguments. To keep track of this we keep a counter of the amount of inputs and outputs we have encountered, and we iterate through the argument list sequentially. If the argument is connected to PLIO, we can directly connect it to the previously declared PLIO object. In the case where the argument is connected to another kernel, we first need to know the index of the connecting argument of the other kernel. To achieve this, we do a lookup in the kernel list based on the kernel user name. Once we have found the other kernel we iterate through the arguments until we find the corresponding argument, and we keep track of the index in the input and output list of the external kernel.

Once we have the index of both the incoming and outgoing data connection, we can declare the connection, marking the arguments as asynchronous when applicable. To avoid duplicate declarations, we only declare connections between two kernels if the current argument is an output window or stream.

4.2.1 Kernel placement

Since we have not specified any location constraints on the kernels in the graph so far, the floorplanning will be automatically performed by the compiler. However if we have a long chain of operations, this can heavily increase compilation times. In Table 4.1 we show the compilation time of chains of `scal` operations, where one version does not place any location constraints and uses the automatic partitioning, while the other version uses location constraints to manually place each kernel into a specific AIE tile. We can see



(a) manually partitioned AIE floorplan

(b) automatically partitioned AIE floorplan

Figure 4.1: Comparison of two AIE programs that compute the same chain of 40 BLAS `scal` operations. The first program uses a manually partitioned layout, while the second program uses an automatically partitioned layout by the `aiecompiler`. The colored boxes highlight kernels that follow each other in the compute chain, but which are not placed in adjacent tiles by the compiler.

that the compilation time of the manually partitioned version steadily increases as the chain becomes longer, mostly because of the extra kernels that have to be compiled, the automatic partitioning increases much faster. At a length of 40 operations, the automatically partitioned program takes around ten times as long to compile than the manually partitioned program, spending most of the time running the global placer. We can also see that once we reach a chain length of 50, the compiler is no longer able to find a solution for the automatically partitioned program and the compilation fails.

If we look at the comparison of the automatically and manually partitioned floorplans in figure 4.1, we can also see that the compiler does not always find an ideal solution. In this case the colored boxes indicate kernels that are directly connected, but which are not placed in adjacent tiles. Since kernels can only access memory from adjacent cells, this means that the compiler will have to place duplicate buffers for both kernels, and copy the data from the first buffer to the second buffer before the second kernel can start executing.

To allow us to create a manual partitioning in AIEBLAS, we add an optional field in the

kernel struct to store a tile location for each kernel. This field is used by the graph generator to put a location constraint in for the desired kernel.

4.3 PL kernel generation

For kernels that use PLIO connections to send or receive data from the host, we need to generate PL kernels. The amount of required PL kernels needed can differ between BLAS operations, and the connections setup by the user. To account for this we use the BLAS operation specific generators to produce a list of PL kernels that need to be generated, together with a lambda function to generate the PL kernel. The main code generator will then iterate through this list and call the lambda functions to generate the PL kernels.

4.4 Build system generation

4.4.1 Connectivity configuration

To build the AIE design we first need to generate a configuration file describing how the PL kernels are linked to the AIE. We first generate general settings such as the platform file of the VCK5000, and profile options. Then we iterate through the BLAS kernel list, and call a function of the BLAS operation specific generators, which will generate the connectivity configuration of the PL kernels related to that BLAS operation.

4.4.2 CMake project

Now that we have generated all the source files required for the AIE design, we need to generate a build system to compile the source files into a complete design to load onto the VCK5000. To accomplish this we generate a CMake project that contains steps to (1) build the ADF graph into a static library, (2) build all the PL kernels to XO objects, and (3) link and package the ADF graph and the PL kernels into an `xclbin` file containing the whole design. The cmake project contains a single target called `aie` to generate the `xclbin`, which compiles the `xclbin` file in the CMake build directory. This allows the CMake project to be integrated into the parent project containing the user program using AIEBLAS.

The cmake project is mostly statically generated, we only need to take the list of source files from the generator and place the sources into the placeholders inside the cmake project.

5

Evaluation

5.1 Performance evaluation

To evaluate the performance of BLAS routines generated by AIEBLAS, we measure the execution time of several designs, and compare the performance to OpenBLAS running on the CPU. We will run the following experiments:

1. Single routines; We run two designs only containing a single generated BLAS routine to set a baseline of the performance of the routines. One design uses the level 1 `axpy` routine, and the other design uses the level 2 `gemv` routine.
2. Tiling optimization; We evaluate the performance of the tiling optimization for the `gemv` routine, as described in section 4.1.3, compared to a non-tiling implementation.
3. Dataflow optimization; To test the capabilities of the spatial dataflow architecture of the AIE, we compare the performance of an AIE design containing two BLAS routines chained together to another AIE design containing the same two BLAS routines, but storing the intermediate result in the memory of the FPGA instead of chaining the routines together.
4. Sum reduction; We run a larger design computing the sum of 32 vectors using a 5-level reduction to test the pipeline performance of the AIE.

5.1.1 General experiment setup

We run our performance experiments on a single node of the AtLarge cluster at the Vrije Universiteit. The node has a 10-core Intel Xeon Silver 4210R CPU running at 2.4GHz,

256 GB of DDR4 memory, and a single VCK5000 accelerator card.

All the code for the performance experiments have been compiled using GCC 11.4.0 for the host code, together with the compiler flags `-O3 -DNDEBUG` to enable compiler optimizations. The VCK5000 designs are compiled using Xilinx Vitis v2022.2, and we use XRT 2.14.0 to control the device from the host.

For the CPU benchmarks we use OpenBLAS 0.3.27, optimized for the CPU architecture using the following configuration: `NO_AFFINITY SKYLAKEX MAX_THREADS=20`.

5.1.2 Single routines

To set a baseline of the performance of single BLAS routines running on the AIE, we measure the performance of a single level 1 `axpy` BLAS routine, and a single level 2 `gemv` routine. For each routine we implement two AIE designs. The first design uses PL kernels for sending and receiving the input and output data, which will measure the performance of a complete AIE design as generated by AIEBLAS. The second design generates the data in a separate AIE kernel, and only uses a PL kernel to send a start signal to the data generating AIE kernel and a PL kernel to receive a finish signal from the AI Engine. This second design will provide us insights into how fast the AIE itself performs without the overhead of the PL kernels.

We measure the execution time from the host from the moment the first PL kernel starts and until the last PL kernel has finished receiving data from the AIE. For the designs generating data on the AIE, we additionally measure the amount of cycles executed on the AI Engine itself, using the intrinsic `get_cycles` function provided by the AIE API.

We compare the execution time of the AIE designs to the execution time of the same BLAS routines run on the CPU using OpenBLAS. For the CPU designs we only measure the time it takes to compute the BLAS routine itself.

All the results are averaged over three runs.

Results

Figure 5.1 shows the results of the experiments with the time measured from the host. We observe that the CPU consistently outperforms the complete AIE design with PL by up to 60× for the `axpy` routine and 10-20× for the `gemv` routine, with the exception of a single outlier in the CPU `axpy` results with a vector size of 2^{14}.

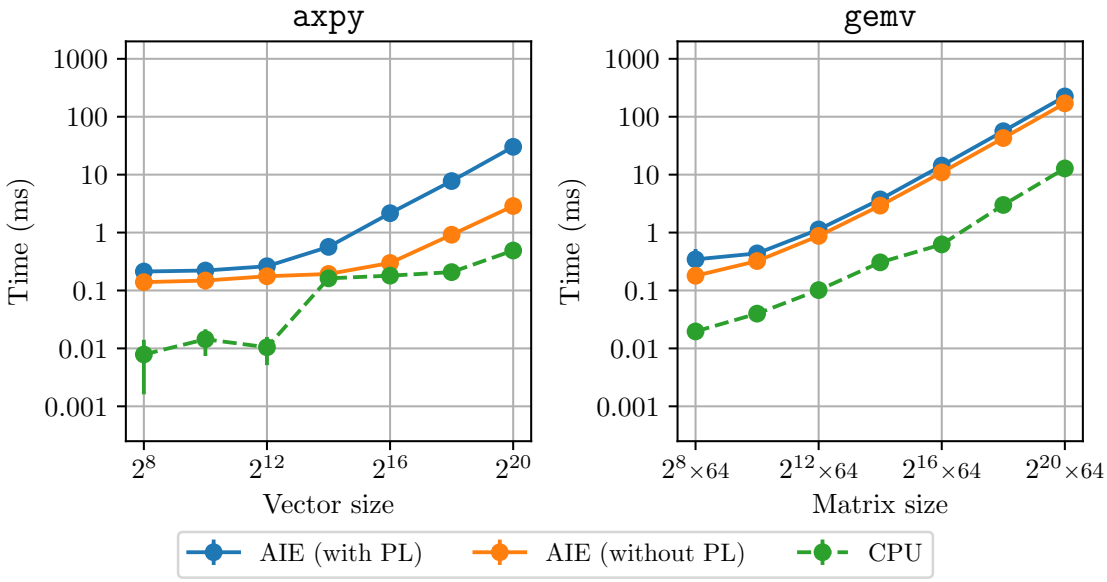


Figure 5.1: Execution time of the single routine experiments, measured from the host application.

It is to be expected that the CPU is faster than the AI Engine when executing a single routine, since we are only utilizing a single tile of the AIE array and not using the dataflow architecture of the device and we have overhead from sending the data from FPGA memory to the AIE array. However this does not explain why the `axpy` routine takes significantly longer to complete for a vector of 2^{20} elements than the `gemv` routine takes to complete for a matrix of size $2^{16} \times 64$, while the latter requires more data transfers and computations. If we look at the results of the AIE designs without PL kernels, we can see that the `gemv` implementation is roughly 30% faster than with PL kernel, since we now do not have to transfer memory to and from the FPGA, but the `axpy` implementation without PL kernel is more than 10× as fast for the largest tested vector size than with PL kernel. This indicates that there is an unexpected amount of time spend in the PL kernel of the `axpy` routine. We suspect that this might be a bug in the older driver or compiler we have to use for the VCK5000, since the latest driver and compiler do not support this device.

If we focus on the results without PL kernel, we can see that the CPU is between 8-18× faster than the AIE implementations, with the exception of the outlying CPU results for the `axpy` routine, which we suspect might be caused by OpenBLAS switching to a multithreaded implementation which has an overhead. Assuming that adding more routines to the AIE design in a pipeline has a negligible influence on the execution time, the AIE

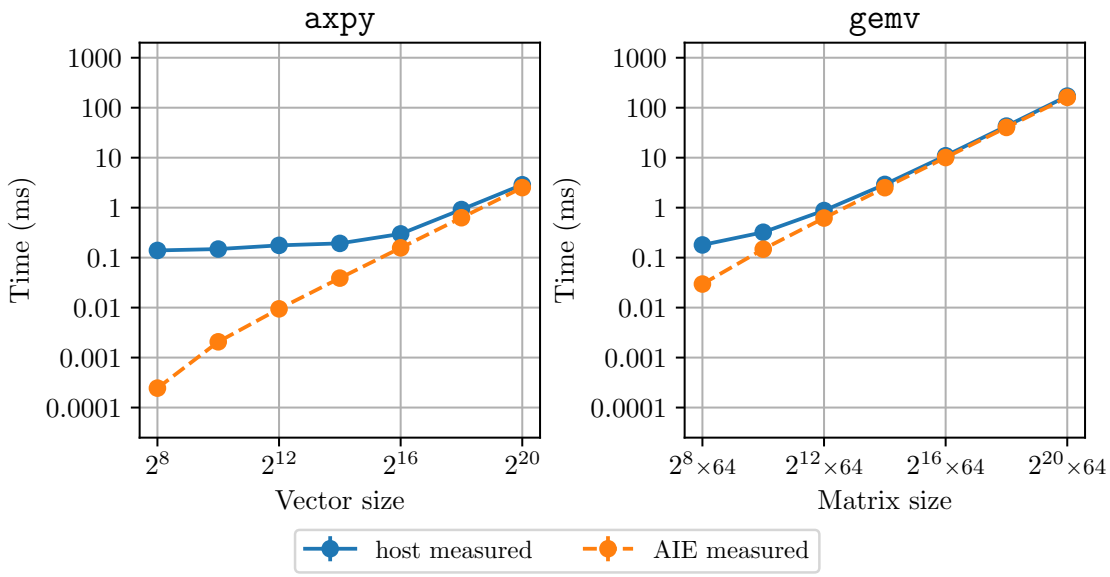


Figure 5.2: Execution time measured from the host and AIE for the single routine implementations not using PLIO to communicate data.

should be able to outperform the CPU when using a larger computation which can be pipelined.

Figure 5.2 shows the cycle measurements which we performed on the AIE compared to the time measurements on the host. The cycles have been converted to execution time by dividing it with the clock frequency of the AIE, which is 1.25 GHz. We can see that the host measurements contains an overhead of around 0.1 ms, which is likely spent in XRT, which makes the measurements less accurate for the smaller data sizes. However for larger data sizes this inaccuracy becomes less significant, since the overhead remains constant.

5.1.3 Tiling optimization

To test the performance of the tiling optimization for the `gemv` routine described in section 4.1.3, we compare an AIE design using a tiled `gemv` implementation to an AIE design which does not use tiling. We run the experiments with different matrix dimensions.

Since there is only a difference in the amount of transfers to the AIE when changing the amount of columns of the matrix, we expect the speedup of the tiling implementation compared to the non-tiling implementation to scale with the amount of columns in the matrix.

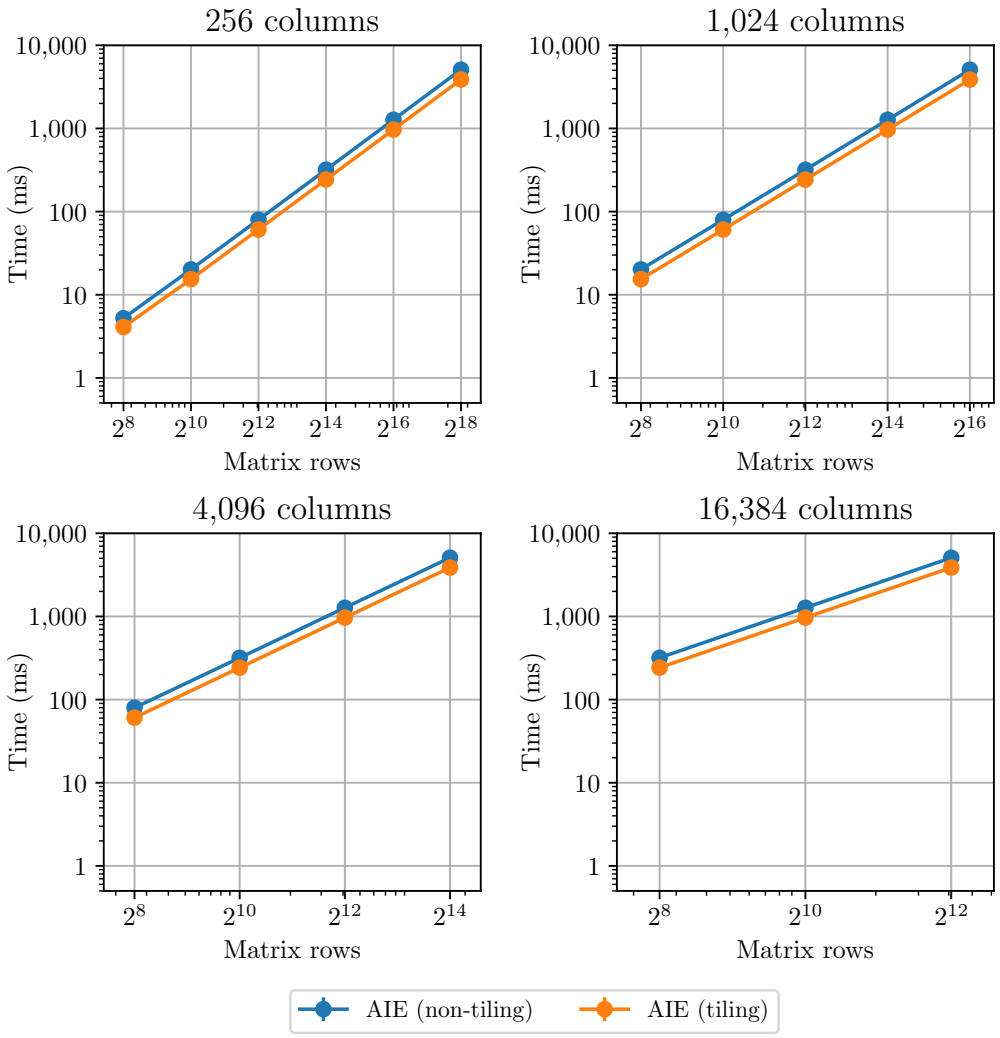


Figure 5.3: Execution time of the tiling `gemv` experiments for different matrix sizes, measured from the host application. Note that all experiments were run with up to 2^{18} rows in the matrix, but the VCK5000 could not allocate enough memory when running with the larger matrices not shown in the figure.

Results

Figure 5.3 shows the results of the tiling experiments. The experiments with matrices of at least 268 million elements did not complete, with the XRT driver reporting that not enough memory was available. In total we allocate around 1 GB for this matrix size, so there should be enough memory available on the FPGA's 16 GB of memory. We can however still analyse the other results.

We see that the tiling implementation is consistently circa 31% faster than the non-tiling implementation. This is a larger speedup than expected, especially for the matrices with

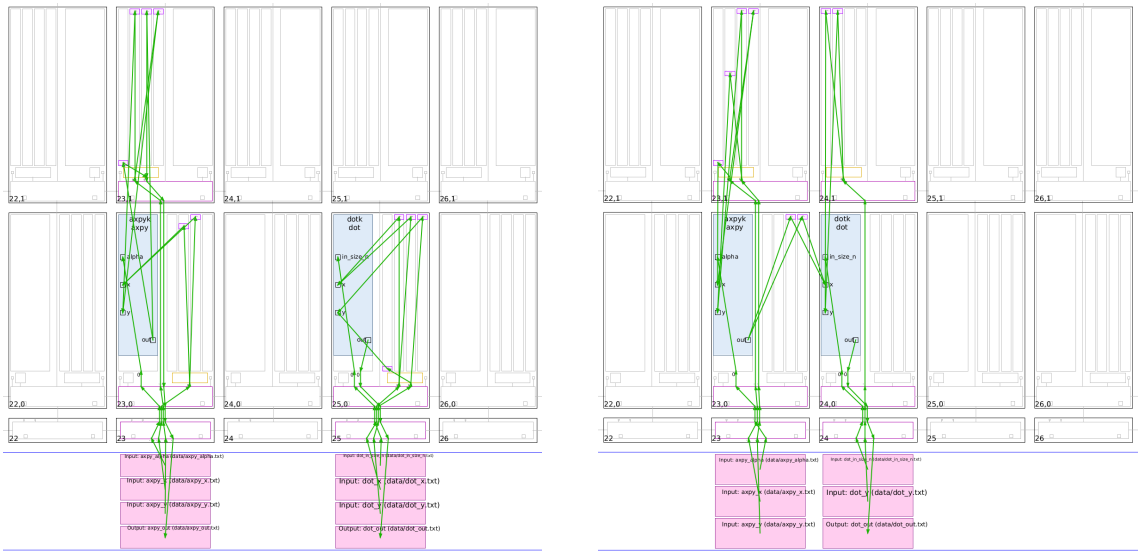
fewer columns, where there is only a small difference in the amount of data transfers to the AIE.

Since we already saw in the single routine experiments that the PL kernels do not seem to run entirely stable, the larger than expected speedup could be attributed to a problem in the PL kernel performance of the non-tiling implementation.

5.1.4 Dataflow optimization

To test the pipelining capabilities of the spatial dataflow architecture of the AIE, we create two AIE designs computing a compound `axpydot` routine, which consists of an `axpy` routine computing $z = w - \alpha$ and a `dot` routine computing $\beta = z^T u$, where u, w and z are vectors and α and β are scalar values. The first design, as shown in Figure 5.4a, places the `axpy` and `dot` routines seperately in the `aie` array, and stores the intermediate results z in the FPGA memory. The second design, shown in Figure 5.4b, instead chains the output of the `axpy` routine to the input of the `dot` routine, creating a pipeline containing the two routines.

We test the two designs with differing vector sizes and we measure the execution time from the host from the moment the first PL kernel starts and until the last PL kernel has finished receiving data from the AIE. The results are averaged over three runs.



(a) Separate axpy and dot kernels

(b) Chained axpy and dot kernels

Figure 5.4: Two different implementations of the combined `axpydot` program computing $z = w - \alpha v$ and $\beta = z^T u$.

We expect the pipelined implementation using a dataflow design to execute significantly faster than the implementation without dataflow. The pipeline should only add a few more cycles to the computation, while the implementation without dataflow needs to wait for the `axpy` routine to finish before being able to start the `dot` routine, which should roughly double the execution time.

Results

Figure 5.5 shows the results of the dataflow experiment. We can see that the pipelined dataflow implementation, which chains the output of the `axpy` routine to the input of the `dot` routine, consistently executes around 1.5-2× faster than the implementation without dataflow. With a vector size of 2^8 the non-dataflow implementation takes an average of 0.20 ms to complete, while the dataflow implementation takes 0.14 ms. For the largest tested vector size of 2^{20} the difference is even larger, with 60.07 ms for the non-dataflow implementation and 30.28 ms for the dataflow implementation.

This result confirms our hypothesis that the implementation without dataflow will take around 2× as long as the pipelined dataflow implementation, since it will have to completely finish the `axpy` routine and store the results in the FPGA memory, before it cab start

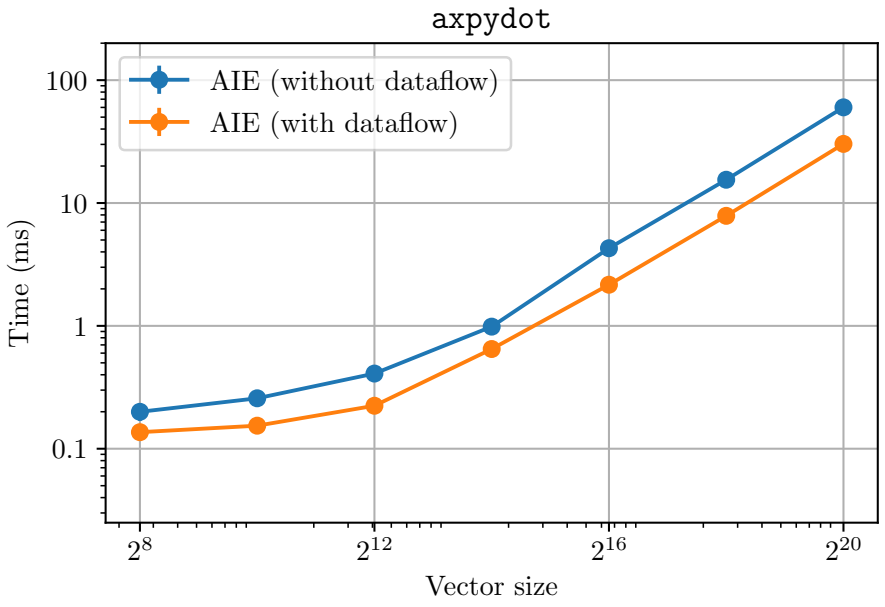


Figure 5.5: Execution time of the dataflow experiment, comparing two AIE designs computing the compound `axpydot` routine. One design uses a dataflow design chaining the `axpy` and `dot` routines, while the other design stores the intermediate result of the `axpy` routine in the FPGA memory. Execution time is measured from the host application.

executing the `dot` routine. The dataflow implementation on the other hand executes as a pipeline, and only takes a few more cycles to compute the `dot` routine after the `axpy` routine. In fact, if we compare the results to the single `axpy` routine experiments, we can see that the dataflow `axpydot` implementation only takes a negligible 0.03 ms longer to complete than the `axpy` routine for a vector size of 2^{20}.

5.1.5 Sum reduction

To fully test the capabilities of the spatial dataflow architecture, we use AIEBLAS to generate a pipelined 5-level sum reduction on the AIE, computing the sum of 32 vectors using `axpy` routines. The dataflow graph of the sum reduction can be seen in Figure 5.6 with 32 incoming vectors going into the graph on the left, and one result vector going out of the graph on the right. We compare two implementations, one using data sent from PLIO, and one using AIE kernels to generate data on the AI Engine itself. The latter uses a separate AIE kernel to generate each input vector.

Figure 5.7 shows the floorplan of the AIE design using PLIO. To be able to compile this design, we have to manually specify the kernel locations in the AIEBLAS configuration, since the design is too large for the global placer of the `aiecompiler` to find a solution.

As a comparison to the CPU, we also implement an OpenBLAS design which executes 32 `axpy` routines in succession to calculate the sum of the vectors. Additionally we reuse the results of the `axpy` routine from section 5.1.2 computing the sum of two vectors to compare the execution time increase of the CPU and AIE when calculating a larger computation.

Results

Figure 5.8 contains the results of the sum reduction experiments. The graph on the left shows the results from section 5.1.2 of a single `axpy` routine computing the sum of two vectors, and the graph on the right shows the results of the new implementations computing a sum of 32 vectors.

If we focus on the designs using PLIO, we can see a major increase in execution time when calculating the sum of 32 vectors compared to the sum of 2 vectors, with a large spike when going to a vector size of 2^{16}. This is unexpected, it should not take over half a second to compute the sum of roughly 2 million elements on a device which can compute 1.25 billion instructions per second on each tile. Additionally we do not observe the same behaviour if we look at the designs which do not use PLIO to receive data from the host. We suspect

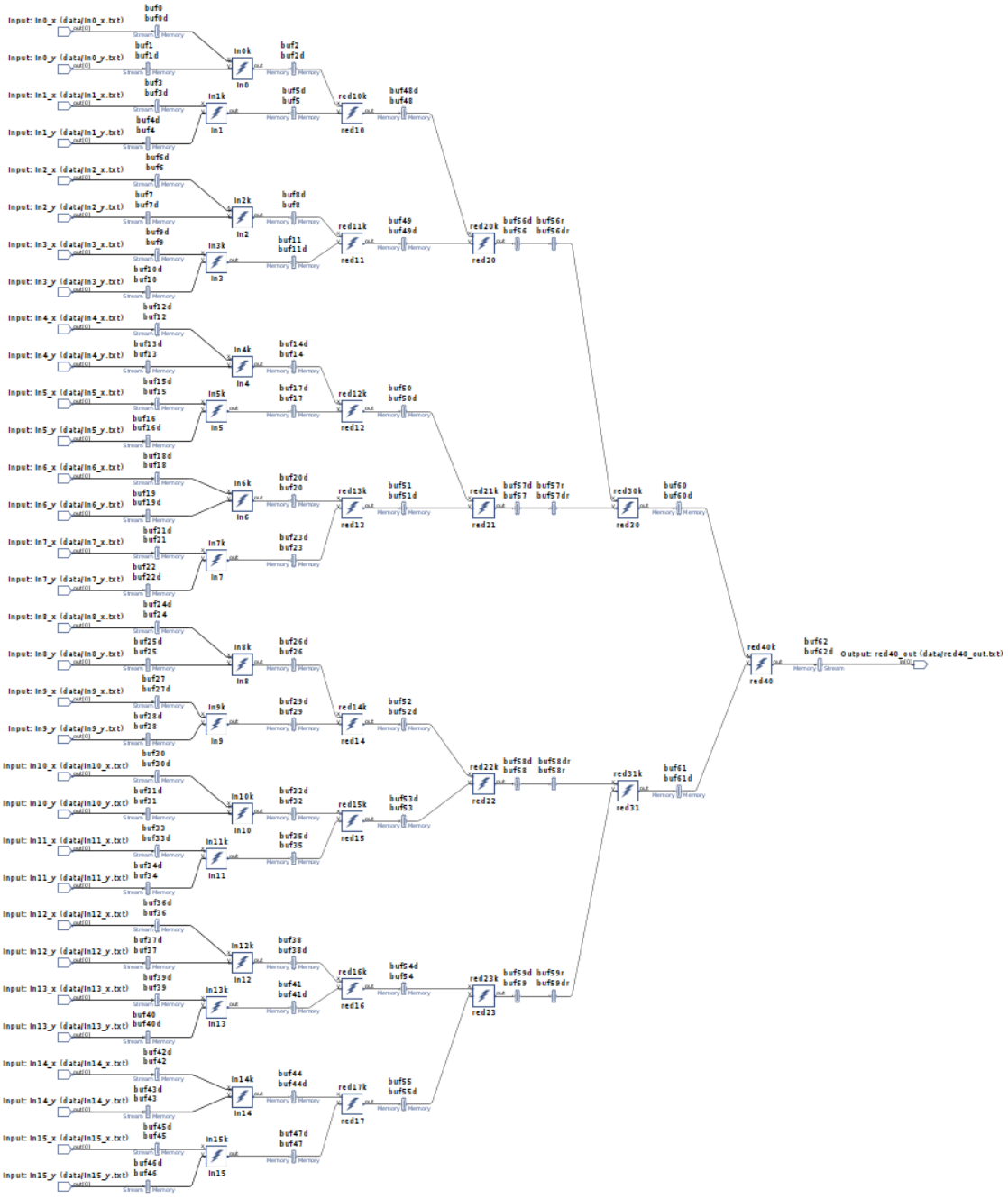


Figure 5.6: Dataflow graph of an AIE design calculating the sum of 32 vectors.

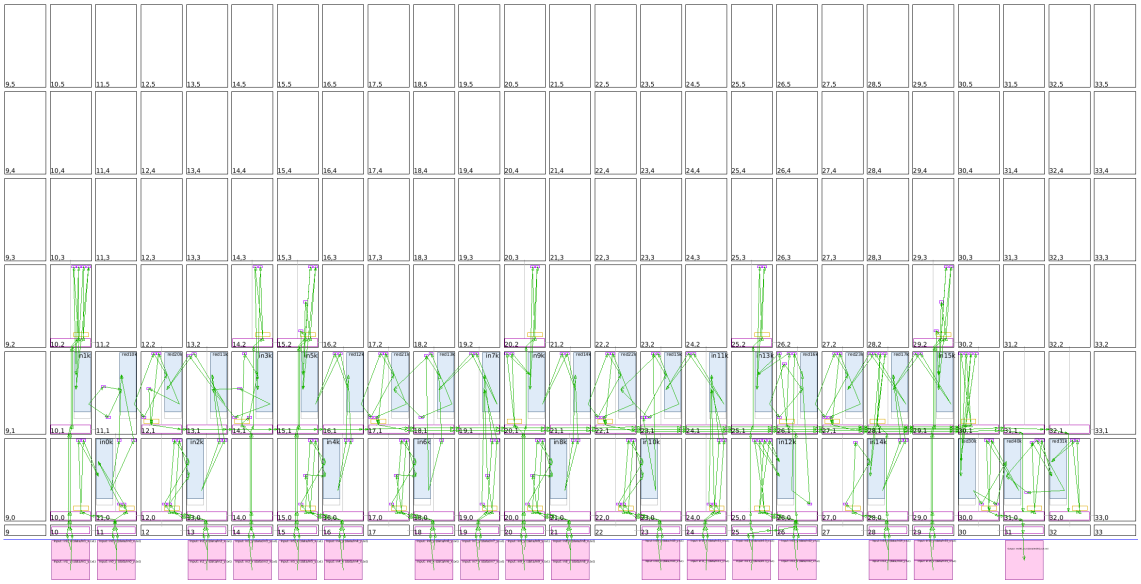


Figure 5.7: Floorplan of an AIE design calculating the sum of 32 vectors.

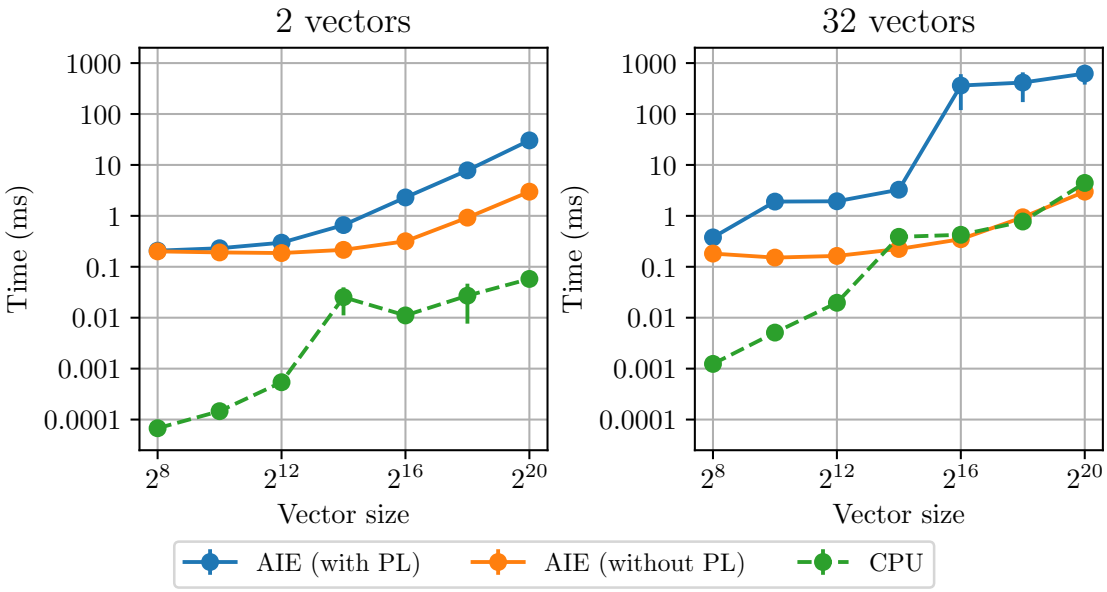


Figure 5.8: Execution time of the sum experiment, the left graph containing programs calculating the sum of two vectors, and the right graph containing programs calculating the sum of 32 vectors. Execution time is measured from the host application.

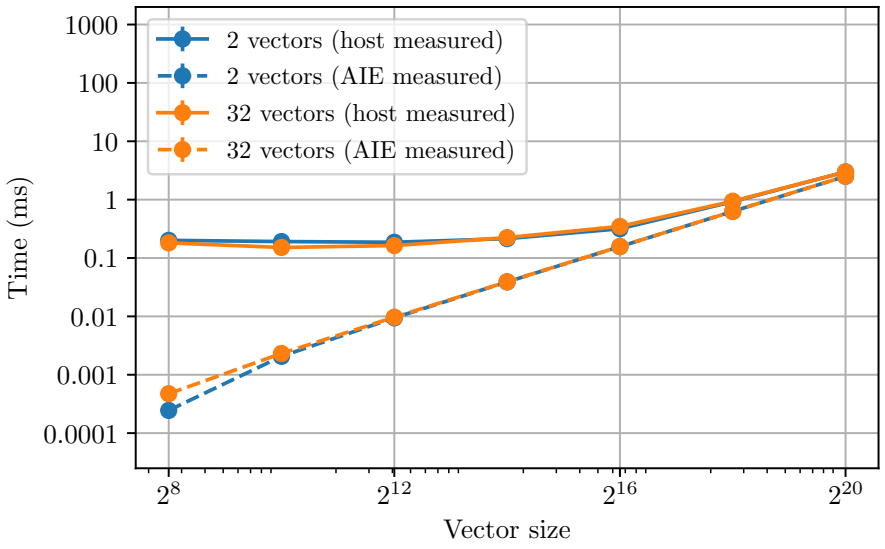


Figure 5.9: Execution time measured from the host and AIE for the sum implementations not using PLIO to communicate data, showcasing the minimal overhead from the pipelined sum reduction.

this is likely a bug with the PL kernels that is present in the old version of the drivers or compiler, as described in section 5.1.2.

If we compare the AIE designs without PL kernels to the CPU implementations, we can see that the AI Engine only observes a negligible execution time increase, while the CPU takes between 15-28× longer to compute the sum of 32 vectors than the sum of 2 vectors.

Figure 5.9 shows us a closer look at the AIE implementations which do not receive data from PLIO. If we look at the AIE measurements we again see a negligible increase in execution time for the sum reduction compared to the single `axpy` kernel. In fact the measured cycles show that the pipelined reduction only takes roughly 300 cycles longer than the single `axpy` kernel, resulting in a 240 nanoseconds time increase.

5.2 Result verification of BLAS routines

To verify the correctness of the BLAS routines generated by AIEBLAS, we compare the results to the same routines provided by OpenBLAS, a well tested CPU implementation of BLAS [10].

Routine	Passed test		
	$m = 256$	$m = 1,024$	$m = 65,536$
asum	✓	✓	✓
axpy	✓	✓	✓
dot	✓	✓	✓
gemv	✓	✓	✗ (0.096% off)
iamax	✓	✓	✓
nrm2	✓	✓	✓
rot	✓	✓	✗ (0.013% off)
scal	✓	✓	✓

Table 5.1: Results of the verification tests, which test if the results of AIEBLAS routines differ at most by 0.01% of OpenBLAS routines for random input data and vectors of size m. The failed tests show the largest observed deviation from the reference results.

Experiment setup

We test all the routines provided by AIEBLAS in a separate AIE design. For each routine we use 512-bit vector instructions on the AIE, and have the window size fixed to 256 bytes. We only use 32-bit floating-point datatypes, since the BLAS specification does not require integer implementations, and there are no integer routines available in OpenBLAS.

Ideally we would also test every permutation of different vector and window sizes together with chained BLAS routines, however each AIE design takes multiple hours to compile, and thus we try to limit the amount of designs we have to generate.

We run each design with input vectors of the following lengths: 256, 1,024 and 65,536. We fill the input data with random numbers generated by the mt19937 pseudorandom number generator (PRNG) [34].

Since we use floating-point datatypes, and the CPU and AIE have different floating-point units, we cannot directly compare if the floating-point numbers are exactly the same, since the numbers can slightly differ based on the underlying precision of the arithmetic implementations. Instead we verify whether the AIEBLAS results differ at most 0.01% from the reference OpenBLAS results. We discard smaller differences as insignificant rounding errors, and not implementation errors.

Results

Table 5.1 shows the results of the correctness verification. We see that most routines pass the tests, and the tests that didn't pass only have a small deviation from the OpenBLAS

results. For the `gemv` routine, 40 out of the 65,536 elements deviated more than 0.01% from the OpenBLAS results, and for the `rot` routine only 1 element out of each of the two output vectors deviated.

For the `rot` routine, we suspect the deviation comes from the fact that the value is calculated using a single multiplication, followed by a single multiply-add instruction. The AIE stores multiplication results in an accumulator of 80-bits, which has a higher precision than the 32-bit floating point numbers. This accumulator is directly passed to the multiply-add instruction, meaning the multiply-add instruction uses this higher-precision intermediate value as input, without converting it to a lower precision floating-point number first. The small deviation from the CPU results can reasonably be caused by the AIE using higher-precision intermediate values.

For the `gemv` routine, each element of the output vector is constructed from a summation of multiplication results. Since we use a matrix consisting of 64 columns, the results are constructed using 64 multiplications, and 63 additions. Each of these operations have a small relative error, but when accumulated, this relative error can grow larger than our threshold of 0.01%. Since the largest error observed was only 0.096%, we suspect this is the case.

We would expect the same to happen to the `dot` routine, since this also is a summation of multiplication results, but the `dot` routine did pass the test. This can be explained by the fact that the `dot` routine only produces a single element, while the `gemv` routine generates 65,536 elements in the output vector. We see that only about 0.06% of the elements in the output vector of the `gemv` routine contained results that did not pass the tests, we would need to run the `dot` kernel a very large amount of times to observe this 0.06% chance of the test to fail.

Overall the two failed tests have a small enough error for us to say the designs containing a single routine produce the correct results, and behave as expected.

6

Discussion

6.1 Hardware limitations

During the development of AIEBLAS we have been hindered by multiple limitations of the VCK5000 hardware and software support. We will list these limitations in this section.

XRT offers numerous runtime functions to control the ADF graph, however these functions are only supported on embedded platforms, and the application crashes with a runtime error if you try to use them with the VCK5000.

The XRT driver also offers abilities to profile the AIE at runtime, however these profiling options do not work with designs built using Xilinx Vitis v2022.2, and require a newer version of the compiler to be used. Xilinx Vitis v2022.2 is the latest version of the compiler which can be used by the VCK5000 platform, since the newer versions v2023.1, v2023.2 and v2024.1 all do not support the VCK5000 platform. This also limits us in the development of AIEBLAS, since Xilinx Vitis v2023.1 and v2024.1 introduces several new features to the AIE toolchain.

While can use the `aiecompiler` to simulate an AIE design, this simulation uses dummy data for the PLIO connection, and can not be used to simulate a design together with the PL kernels. This means that if we want to test a full design generated by AIEBLAS, we need to compile a full hardware design for the VCK5000, which can take up to three hours to complete. Only then are we able to test if the design works as expected. If it does not work we have a limited ability to debug our design, since the profiling options are not supported by XRT, we can not get a clear view of why certain aspects of the design take longer than expected, or why output results are incorrect. For example, we have not been

able to debug why the PL kernel design of the `axpy` routines does not provide the expected performance in chapter 5.

During the experimentation with the device we have had multiple occasions where the AIE array became unstable, the device was unable to be hot reset, and the server containing the VCK5000 had to be completely rebooted before the VCK5000 worked again.

6.2 Future work

In this section we describe various possible continuations of the work in this thesis. To make AIEBLAS complete, the missing second and third level BLAS routines could be added to the library, by implementing the missing sub generators. As a start one of the existing GeMV routines [26–29] could be integrated in the library.

While AIEBLAS currently makes it easy to generate an AIE design, users still need to write their own host application, interfacing with XRT. It would be useful to generate a host API for AIEBLAS, which hides away the details of XRT, and provides standard BLAS function definitions that once called, launch the requested BLAS operations from the AIE design.

With a host API added to AIEBLAS, it would be possible to automate the JSON user configuration process, by performing a dry-run of the host application and keeping track of the function calls to the host API. The JSON configuration could then be generated based on the called functions.

Further optimizations for the existing BLAS routines in AIEBLAS could be implemented. For example it would be interesting to investigate optimized implementations of the existing routines which can span over multiple AIE tiles, to utilize more of the computational power of the device. Related work has shown that much higher performance could be achieved on the AIE. [29]

Currently AIEBLAS only targets the VCK5000, but recently AMD has started releasing Ryzen CPUs with an integrated AI Engine. If AIEBLAS was adapted to be able to target these devices, it would create a much larger pool of target devices. The CPUs do not offer PL, so the library would have to be adapted to use another method available on the CPUs to communicate with the AI Engine array, but in principle the aie design itself should be usable without further changes.

Alternatively the design choices for implementing a BLAS library on a spatial dataflow architecture could also be used to create a library targeting other devices with a similar architecture.

7

Conclusion

Devices with a spatial dataflow architectures are being released as AI accelerators. For these devices to be used for general numerical problem solving in the high-performance computing (HPC) community, we need a high-level numerical library to program the device. In this thesis we propose AIEBLAS, a high-level BLAS library targeting the AMD VCK5000 with an AIE array. We determine important design requirements for developing a BLAS library targeting a spatial dataflow architecture, and use the design requirement to develop a code generator that generates AIE designs containing BLAS routines. We evaluate the performance of the generated design using multiple experiments and compare the performance to OpenBLAS, a CPU BLAS implementation. In this chapter we summarize our findings, and answer the research questions as presented in section 1.1.

7.1 Main findings

[RQ1] Which design choices ensure a usable and expandable BLAS library targeting a spatial dataflow architecture?
In chapter 3 we proposed several design choices to ensure a BLAS library can properly use the spatial dataflow architecture of the AIE, while at the same time keeping the library expandable. The design choices we propose are:

- The BLAS routines should be mapped to to an AIE kernel, where the kernel arguments should stay close the the routine arguments. However small changes have to be made to create separate output arguments for the kernels, to ensure we can properly use the dataflow architecture.

- It is important that we can chain BLAS routines in the BLAS library, such that we can properly utilize the dataflow architecture.
- To keep the design expandable, the generator should be split in two parts: a main generator which handles most of the static and non-routine dependent code, and sub generators which implement the routine dependent code.
- The library should take a JSON configuration file as input, containing the requested BLAS routines to be generated with kernel and graph settings.

[RQ2] How can we automatically generate a dataflow program consisting of BLAS routines for an AI Engine from a high-level specification?
In chapter 4 we describe the implementation of the code generation in AIEBLAS, which takes a JSON file as input and automatically generates a full AIE design. The generated AIE design consists of (1) AIE kernels implementing the requested BLAS routines, (2) an ADF graph specification, specifying how the kernels are connected in the dataflow graph, (3) PL kernels to connect the AIE array to the host, and (4) a build system to build the generated files into a full design for the VCK5000.

[RQ3] What optimizations can we apply to the kernel generation of BLAS routines to make the BLAS routines more performant on an AIE?
We proposed several optimizations in chapter 4 that we implement in the kernel generation of AIEBLAS. The optimizations are:

- The restrict qualifier can be used for the input arguments to allow the `aiecompiler` to use more aggressive optimizations.
- If scalar values of BLAS routines are known at compile time, they can be embedded into the source code of the AIE design, saving overhead from transferring the scalar from the host to the AIE array at runtime.
- For the `gemv`, a tiling implementation can reduce the amount of data send to the AIE array, by improving the data access pattern of the kernel.

[RQ4] How performant is the AIEBLAS library compared to other BLAS libraries when performing common routines?

We performed multiple experiments on BLAS routines generated by AIEBLAS, as described in chapter 5. When running a single routine on a single AIE tile, the AIE was significantly slower than the same routine run on the CPU using the BLAS library OpenBLAS. However when running a larger design consisting of multiple routines, the AIEBLAS design gets comparable performance to the CPU using OpenBLAS, if we do not use PL kernels.

Bibliography

[1] G. E. Moore. 'Cramming more components onto integrated circuits'. In: *Electronics* 38.8 (1965), pp. 1–14. DOI: 10.1109/N-SSC.2006.4785860.

[2] R. Dennard et al. 'Design of ion-implanted MOSFET's with very small physical dimensions'. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[3] H. Esmaeilzadeh et al. 'Dark silicon and the end of multicore scaling'. In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 365–376. ISSN: 0163-5964. DOI: 10.1145/2024723.2000108.

[4] T. N. Theis and H.-S. P. Wong. 'The End of Moore's Law: A New Beginning for Information Technology'. In: *Computing in Science & Engineering* 19.2 (2017), pp. 41–50. DOI: 10.1109/MCSE.2017.29.

[5] M. Horowitz. '1.1 Computing's energy problem (and what we can do about it)'. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[6] A. Rico et al. 'AMD XDNA™ NPU in Ryzen™ AI Processors'. In: *IEEE Micro* (2024), pp. 1–10. DOI: 10.1109/MM.2024.3423692.

[7] J. Ansel et al. 'PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation'. In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024. DOI: 10.1145/3620665.3640366.

[8] M. Abadi et al. *TensorFlow, Large-scale machine learning on heterogeneous systems*. July 2024. DOI: 10.5281/zenodo.12726004.

[9] L. S. Blackford et al. 'An updated set of basic linear algebra subprograms (BLAS)'. In: *ACM Trans. Math. Softw.* 28.2 (June 2002), pp. 135–151. ISSN: 0098-3500. DOI: 10.1145/567806.567807.

[10] X. Zhang et al. *OpenBLAS: An optimized BLAS library*. URL: https://www.openblas.net/ (visited on 25/07/2024).

[11] F. G. Van Zee and R. A. van de Geijn. 'BLIS: A Framework for Rapidly Instantiating BLAS Functionality'. In: *ACM Trans. Math. Softw.* 41.3 (June 2015). ISSN: 0098-3500. DOI: 10.1145/2764454.

[12] *cuBLAS*. NVIDIA Corporation. URL: https://developer.nvidia.com/cublas (visited on 25/07/2024).

[13] *VCK5000 Versal Development Card*. Advanced Micro Devices, Inc. URL: https://www.xilinx.com/products/boards-and-kits/vck5000.html (visited on 26/07/2024).

[14] *Versal™ Architecture and Product Data Sheet: Overview (DS950)*. Advanced Micro Devices, Inc. URL: https://docs.amd.com/v/u/en-US/ds950-versal-overview (visited on 26/07/2024).

[15] *Versal Adaptive SoC AI Engine Architecture Manual (AM009)*. Advanced Micro Devices, Inc. URL: https://docs.amd.com/r/en-US/am009-versal-ai-engine (visited on 26/07/2024).

[16] *AI Engine Kernel and Graph Programming Guide (UG1079)*. Advanced Micro Devices, Inc. URL: https://docs.amd.com/r/2022.2-English/ug1079-ai-engine-kernel-coding (visited on 26/07/2024).

[17] *ASIP Programmer Chess Compiler User Manual*. Version R-2021.03. Synopsys, Inc. 2021.

[18] *AI Engine Tools and Flows User Guide (UG1076)*. Advanced Micro Devices, Inc. URL: https://docs.amd.com/r/2022.2-English/ug1076-ai-engine-environment (visited on 26/07/2024).

[19] *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)*. Advanced Micro Devices, Inc. URL: https://docs.amd.com/r/2022.2-English/ug1393-vitis-application-acceleration (visited on 26/07/2024).

[20] *BLAS (Basic Linear Algebra Subprograms)*. Basic Linear Algebra Subprograms Technical (BLAST) Forum. URL: https://www.netlib.org/blas/ (visited on 25/07/2024).

[21] *LAPACK — Linear Algebra PACKage*. The University of Tennessee et al. URL: https://www.netlib.org/lapack/ (visited on 25/07/2024).

[22] *Accelerate Fast Math with Intel® oneAPI Math Kernel Library*. Intel Corporation. URL: https://software.intel.com/en-us/intel-mkl (visited on 25/07/2024).

[23] *AMD Optimizing CPU Libraries (AOCL)*. Advanced Micro Devices, Inc. URL: https://developer.amd.com/amd-aocl (visited on 25/07/2024).

[24] *Vitis BLAS Library*. Advanced Micro Devices, Inc. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-blas.html (visited on 25/07/2024).

[25] T. De Matteis, J. de Fine Licht and T. Hoefler. 'FBLAS: Streaming Linear Algebra on FPGA'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986. DOI: 10.1109/SC41405.2020.00063.

[26] J. Lei et al. 'GEMM-Like Convolution for Deep Learning Inference on the Xilinx Versal'. In: *High Performance Computing*. Ed. by A. Bienz et al. Cham: Springer Nature Switzerland, 2023, pp. 593–604. ISBN: 978-3-031-40843-4.

[27] E. Taka et al. *Efficient Approaches for GEMM Acceleration on Leading AI-Optimized FPGAs*. 2024. DOI: 10.48550/arXiv.2404.11066. arXiv: 2404.11066 [cs.AR].

[28] J. Zhuang et al. 'CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture'. In: *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '23. Monterey, CA, USA: Association for Computing Machinery, 2023, pp. 153–164. ISBN: 9781450394178. DOI: 10.1145/3543622.3573210.

[29] J. Zhuang et al. 'CHARM 2.0: Composing Heterogeneous Accelerators for Deep Learning on Versal ACAP Architecture'. In: *ACM Trans. Reconfigurable Technol. Syst.* (August 2024). Just Accepted. ISSN: 1936-7406. DOI: 10.1145/3686163.

[30] W. Zhang et al. 'EA4RCA: Efficient AIE accelerator design framework for regular Communication-Avoiding Algorithm'. In: *ACM Trans. Archit. Code Optim.* (July 2024). Just Accepted. ISSN: 1544-3566. DOI: 10.1145/3678010.

[31] C. Heinz et al. 'TaPaS Co-AIE: An Open-Source Framework for Streaming-Based Heterogeneous Acceleration Using AMD AI Engines'. In: *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2024, pp. 155–161. DOI: 10.1109/IPDPSW63119.2024.00041.

[32] V. Zverovich et al. *{fmt}*. Version 10.2.1. January 2024. URL: https://fmt.dev/ (visited on 20/08/2024).

[33] N. Lohmann. *JSON for Modern C++*. Version 3.11.3. November 2023. URL: https://json.nlohmann.me (visited on 20/08/2024).

[34] M. Matsumoto and T. Nishimura. 'Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator'. In: *ACM Trans. Model. Comput. Simul.* 8.1 (January 1998), pp. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995.