Designing a Communication Library for Xilinx Versal Devices Using the Window-Based API

Designing a Communication Library for Xilinx Versal Devices Using the Window-Based API

Bachelor Thesis

Sven Zanetti

Advisors: Prof. Dr. T. Hoefler[1], Dr. T. De Matteis[2], L. Gianinazzi[1]

[1] ETH Zürich, [2] Vrije Universiteit Amsterdam

Scalable Parallel Computing Laboratory, ETH Zürich

August 20, 2023

Abstract

The Adaptive Compute Acceleration Platform (ACAP) developed by AMD/Xilinx is a novel architecture which combines three parts: An ARM-based CPU, an FPGA and a CGRA. The CGRA is implemented as a configurable grid of powerful vector processors called AI Engines (AIEs), whose communication patterns can be programmed using dataflow graphs.

Programming the AI Engines is difficult because it requires extensive knowledge of the underlying hardware. To the best of our knowledge, there are currently no open source libraries which add abstraction layers over the API provided by the manufacturers. Collective operations known from the Message Passing Interface (MPI) could simplify the programming experience, but only point to point communication and broadcasting are supported out of the box.

Using the VCK190 Evaluation Kit, which is part of the Versal AI Core Series, we build upon exploratory work done by [4] and extend it with a latency benchmark of inter-AIE communication for various distances. In the main part of the thesis, we adapt MPI's Reduce function to this architecture as a particular example of a collective operation, and we implement our design. We perform various scientific benchmarks and develop a cost model to predict the cycle count of our implementation. Furthermore, we analyse several limitations which arise when scaling the implementation to larger data sizes and numbers of AIEs, which limits the usability of our prototype in real applications. Finally, we present theoretical considerations concerning the design of another collective operation called Gather.

We provide our implementation of the Reduce function and the necessary connection logic as a proof-of-concept on Gitlab [11].

Contents

CHAPTER 1

Introduction

The field of high performance computing is changing more rapidly than ever due to the end of Moore's Law and Dennard Scaling. At first, the solution was simply to add more cores to CPUs, but this is only feasible and sensible to a certain degree. Adding more cores has diminishing returns, in particular if programs are not highly parallelizable; also, cache coherency and other data movement protocols exacerbate these diminishing returns. The industry moved towards more and more highly specialised processors, such as GPUs and, recently, TPUs. These are very efficient but due to their specificity cannot be used once requirements change.

This has resulted in an interest in reconfigurable hardware such as FPGAs. FPGAs offer fine-grained control over low-level hardware primitives such as logic gates and look-up tables. If used correctly, they provide the specificity and efficiency of more specialised processors while also being flexible enough to adapt to different tasks. However, as FPGA programming is done very close to the hardware, it is often difficult and requires a lot of time. Coarse-grained reconfigurable arrays (CGRAs) have been proposed as a compromise between the low-level control offered by FPGAs and the programmability of more traditional devices. They consist of an interconnected grid of full-blown processors, but the processors as well as the connections can be freely programmed.

AMD/Xilinx has released the Versal Adaptive Compute Acceleration Platform (ACAP) which combines several different paradigms. It consists of a CPU part, an FPGA part and a CGRA part, connected via a Network on Chip (NoC). The CGRA is realized using a grid of Very Long Instruction Word (VLIW) vector processors called *AI Engines* (AIEs). Each AI Engine contains local data memory and is connected to the remaining engines and to the other parts of the architecture in different ways. However, to the best of our knowledge,

there are as of yet no open source libraries apart from those distributed directly by AMD/Xilinx to help with programming the AI Engines. Programmers have to consider many low-level hardware details, in particular regarding the communication between AI Engines.

In high performance multi-core environments, a common approach to communication is to use the Message Passing Interface (MPI). Cores communicate not by writing into shared memory, but by sending each other messages instead. We wanted to explore whether it is possible to use a similar approach on the AI Engine architecture. In particular, we wanted to find out whether it is possible and efficient to implement collective operations on this hardware. With point to point communication and broadcasting capabilities provided out of the box, we focus on the Reduce collective operation for our implementation and analysis.

In chapter 2, we introduce the hardware and the programming model. Chapter 3 describes our first steps with the architecture, showcasing point to point communication as we extend the work of [4] with a benchmark of inter-AIE latency. The main part of the thesis is chapter 4, where we design the Reduce function for this architecture. We provide the implementation of a prototype, evaluate it using benchmarks and develop a cost model able to predict the latency of the operation. A detailed analysis of limitations regarding how well our implementation scales finishes this chapter off. In chapter 5, we briefly touch on a second collective operation known from MPI which is called Gather and provide certain design considerations. Finally, chapter 6 mentions some implementation issues and gives an outlook on further work.

1.1 Related Work

Another bachelor thesis completed recently at ETH did much of the foundational work on which our thesis builds [4]. Initially, we used their code to get used to the programming model before adapting it to suit our own needs. The thesis gives an excellent overview of the hardware specifications and the programming model. It is mainly concerned with throughput benchmarks, concluding that they mostly match the manufacturer's claims. It suggests as further work that latency benchmarks could be done or that one could attempt to implement collective operations for the architecture, so it has significantly inspired our thesis.

Other work published earlier this year has used the Versal VCK190 to implement a kernel commonly used in deep learning for the AI Engine architecture [1]. An evaluation of the Versal architecture for computing in space has likewise been published this year, concluding that the heterogeneity of the device is a good fit for the varying requirements in space [3]. It also highlights the energy efficiency of the AI Engines.

CHAPTER 2

Background

This chapter gives an overview of the hardware of the AMD/Xilinx Versal device and introduces the programming model. The structure of the hardware overview has been inspired by [4].

2.1 Hardware

We are using the VCK190 Evaluation Kit, which is the first board released as part of the AMD/Xilinx AI Core series. The AI Core series is based on the Adaptive Compute Acceleration Platform (ACAP). This is a heterogeneous architecture, consisting of three main parts: Scalar Engines, Adaptable Engines and Intelligent Engines.

Scalar Engines

The board contains two scalar engines, namely an ARM Dual-Core Cortex-A72 application processor and an ARM Dual-Core Cortex-R5 real-time processor. The application processor runs Petalinux, which is used as a host and can, for example, start programs which should run on the Intelligent Engines. We are running Petalinux 2022.1. The real-time processor is not considered in our thesis.

Adaptable Engines

Adaptible engines are an umbrella term to refer to a set of different RAM blocks, DSP engines and a traditional FPGA, which is also called programmable logic (PL). In this thesis, none of the adaptible engines are used.

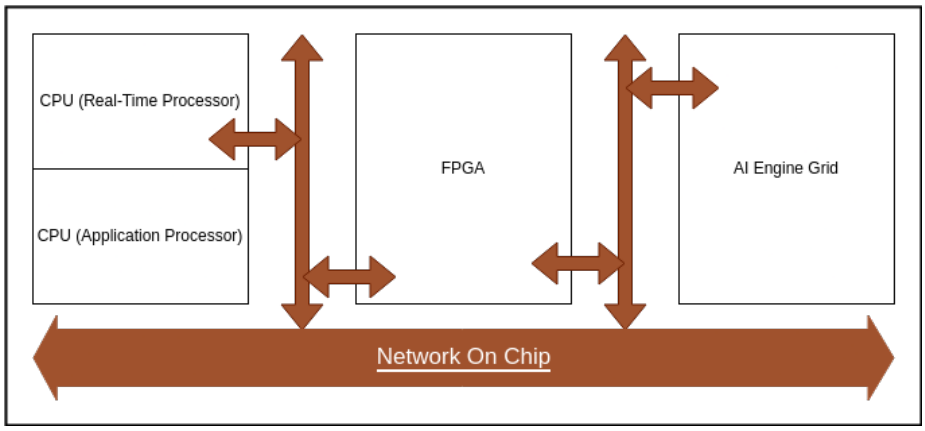

Figure 2.1: Abstracted architecture of AMD/Xilinx ACAP. Some details are omitted if they are not necessary for this thesis.

Intelligent Engines

Intelligent engines, which are of most interest for our thesis, refer to a two-dimensional array of so-called AI Engine tiles which we will call the *AIE grid*. In the case of our board, the VCK190, the grid has a dimension of 8x50 AI Engine tiles. Each AI Engine tile contains the AI Engine itself as well as its $32KB$ local data memory, along with several different types of connections. The AI Engine is a 7-way very long instruction word (VLIW), single instruction multiple data (SIMD) vector processor. Per cycle, it can perform one scalar operation, one vector operation, one vector store, two vector loads and two moves. In all of our programs, the processor runs at a frequency of $1.25GHz$, but for more complicated programs it may run at $1GHz$.

AI Engines are interconnected in various ways. Most important for our thesis are two types of connections: *Tile memory access* and *DMA*.

- In addition to their own data memory, each AI Engine tile can access the data memory of their north and south neighbouring tiles, as well as the data memory of either their east or west neighbouring tile. As seen in figure 2.3, in each row the tiles alternate between having the AI Engine on the left or the right side. If the engine is on the right side of a tile, it can access the data memory of its east neighbour — vice versa, if it is on the left side of a tile, it can access that of its west neighbour.

- Connection between non-neighbouring tiles happens via a DMA (Direct Memory Access) engine. The DMA engine internally uses the AXI4 stream interconnects to reach any tile on the grid.

In addition, there is the possibility of using AXI4 streams regardless of whether the communicating engines are neighbours and there are so-called cascade streams, but these have not been investigated for our thesis.

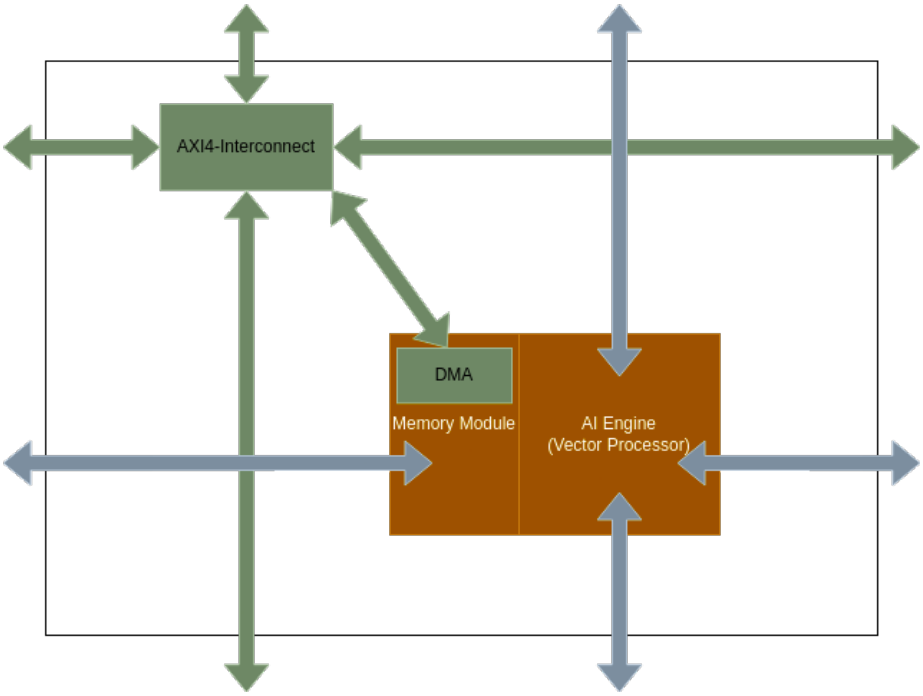

Figure 2.2: One AI Engine tile, containing the AI Engine, its local data memory and two types of connections. Green connections are used for DMA-based communication, while blue connections are used for Tile Memory Access.

2.2 Programming Model

The programming paradigm of the AI Engine grid follows that of a dataflow graph. In a dataflow graph, kernels are connected with directed edges which symbolise that one kernel produces some data, sends it along the edge to another kernel which can then use the data for its computations. The sending kernel is usually called a producer, the receiving one a consumer.

When programming the AI Engines, a programmer usually first defines the computations a kernel performs by defining the kernel as a function in a particular file. In a second step, the programmer can create an abstract dataflow graph. To do so, the programmer instantiates kernels to create nodes of the graph. It is worth pointing out that a single kernel function f can be instantiated multiple times, that is, many graph nodes can execute the same logical function. These nodes can then be connected. Finally, the abstract dataflow graph is mapped onto the physical AIE grid. This can be left to the compiler which tries to find a mapping such that resource utilization is minimized, or it can be done manually by the programmer. Note that when the number of nodes becomes larger, the compiler takes longer to find a a solution and may eventually fail to find one, in which case the user is forced to manually create a mapping.

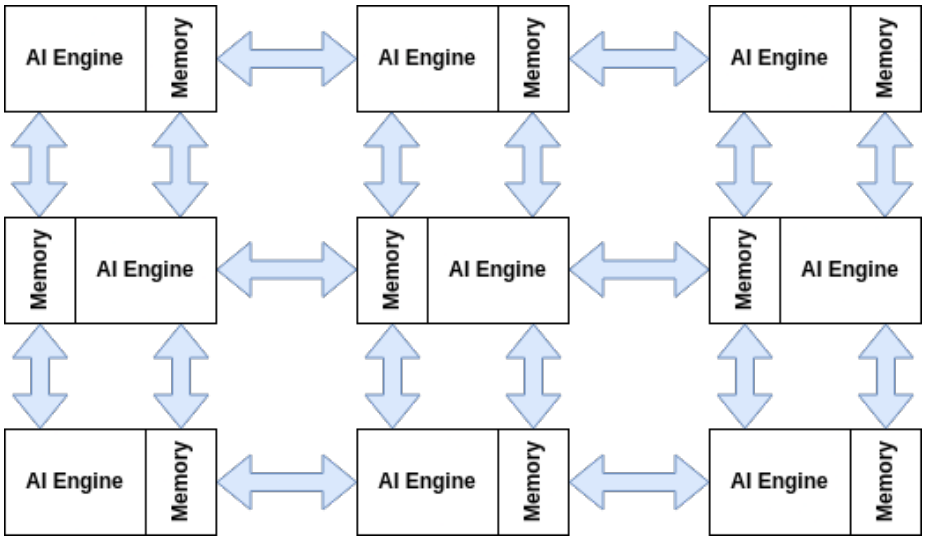

Figure 2.3: The AI Engine Tile Array. Connections in blue symbolise access to local data memory of neighbouring tiles (Tile Memory Access).

From a software perspective, the API offers two different ways of connecting kernels: streams and windows. In streaming connections, a kernel has access to input stream objects and/or an output stream objects. Within the kernel code, the programmer is free to pop data from input streams and push data to output streams arbitrarily often, element by element. All streaming connections are mapped onto the hardware as AXI4 streams, regardless of the relative position of the producer and consumer kernels. However, streaming connections will not be considered for the remainder of this thesis.

In window connections, which we shall exclusively consider, a kernel has access to input window and/or output window objects. Such a window has a predetermined size (in bytes), which can be fixed individually for each window connection. Within the kernel code, the programmer can assume that all input windows are completely filled with data, and a programmer can randomly access this data. Likewise, a programmer can fill each output window in a random-access way. How window connections are mapped onto hardware depends on the relative position of the producer and consumer kernels. If possible, the connection makes use of tile memory access. Recall that tile memory access is only possible if the kernels are mapped onto neighbouring AI Engine tiles. Recall further that neighbours in north/south direction can always use tile memory access, but whether a tile can access its east or its west neighbouring tile depends on the row on the AIE grid. If it is not possible to use tile memory access (so in particular, if the kernels are mapped onto non-neighbouring tiles), window connections use the DMA engine which is present on all AIE tiles.

Furthermore, window connections make use of so-called *double buffering* (also called *ping-pong buffering*) to increase performance. Double buffering refers

to the technique that each logical window object is mapped onto two physical memory regions (called the ping buffer and the pong buffer), located on different memory banks to avoid address conflicts. This is useful because producer kernels can only write to output windows if any previously written data has already been read by the consumer kernel. Consider the following thought experiment, where we assume a producer and a consumer both take only a single cycle to produce/consume some window data: If the consumer writes some data to an output window during the first cycle, then the receiver can read this data during the second cycle. But during that same second cycle, the producer cannot produce another window of data because at that moment, its previous window has not been consumed yet; it would overwrite the data as it is being consumed. Only in the third cycle can it produce its next window; so in summary, every second cycle is wasted. Double buffering solves this issue by alternating between the ping and the pong buffer each time the kernel is invoked. Hence, the producer kernel writes to the ping buffer in the first cycle and to the pong buffer in the second; and in the third cycle, the ping buffer is available again as its data has been consumed during the second one. Ping-pong buffers are invisible to the kernel programmer — from a software perspective, all writes to the output window happen to the same object.

As stated before, a programmer can assume in the kernel code that all input windows are filled with data and all output windows are empty. This behaviour is called *synchronous window access*. To guarantee this behaviour, the API needs to ensure that a kernel function is only invoked once these conditions are fulfilled. To this end, the API maintains *window locks*, either one or two per window connection depending on whether or not the connection is double buffered. The window locks of a connection are shared between its consumer and its receiver — e.g., while the producer of a given connection holds the window lock on its ping buffer, the consumer cannot acquire the window lock on its own ping buffer. The API ensures that a kernel is only invoked after its has acquired the window locks of all its connections — thus guaranteeing that data does not change in the middle of the kernel's execution.

There is one exception to this rule, namely if a window is marked as *asynchronous*. To be more specific, for a given window connection, either the producer or the consumer or both have the freedom to mark their end of the connection as asynchronous. If a window connection is asynchronous for a kernel, it is ignored in that kernel's invocation conditions; in other words, the window lock for this connection is not acquired behind the scenes before the kernel is invoked. This has the result that inside the kernel code, the programmer can no longer assume that data is available to read (if it is an input window) or that it is safe to write (if it is an output window). The programmer is required to manually call a method which acquires the window before they are allowed to perform any read or write operation on it. This acquisition call blocks until the window becomes available, i.e. until the other end of the connection releases their lock on it. The programmer is further required to release the window lock manually inside the kernel after they are done, with the effect that the other end of the connection is then free to acquire the lock on that window again (potentially even before the

kernel has finished the current invocation). Asynchronous windows are useful in a situation where a kernel might have to perform a lot of compute-intensive work before it requires the data from an input window; manually acquiring the window lock later in the kernel allows it to perform that work at the same as the producer is generating data instead of only starting with the intensive work after the producer is done, leading to more parallelism. Furthermore, asynchronous windows allow a kernel to read or write several windows of data to a connection within a single kernel invocation. Usually, as window locks are only acquired and released before a kernel starts and after it has finished, there can be only one window read or written per invocation. This gives the programmer more freedom in designing their programs. Finally, as we will see, there are situations where it is necessary to use asynchronous windows to avoid cyclic dependencies.

To recap, the locks of synchronous windows are handled behind the scenes, which is the default behaviour of window connections. The locks of asynchronous windows must be handled manually by the programmer.

CHAPTER 3

Latency Benchmark

Previous work has already been done to benchmark the throughput of the device for various communication patterns [4]. Their results on window-based communication show that neighbouring AI Engines can achieve a throughput of 29 GB/s using tile memory access (page 45), while non-neighbouring AI Engines only achieve a throughput of 4.4 GB/s (page 48). This section will discuss our attempt at benchmarking the latency of window connections. This is interesting as it can provide a more complete picture of the capabilities of the device — in particular, as we will see, the latency is quite high even for tile memory access. Furthermore, this chapter serves as an introduction to point to point communication which is provided by the API out of the box. It also showcases how asynchronous windows must be used in some situations to avoid deadlocks. Finally, it shows how cycle counters can be used for performance measurements, which will be reused later when we evaluate the performance of the Reduce implementation.

3.1 Setup

We measure the latency of a window connection between two engines indirectly by measuring the round-trip time. In particular, we transfer the smallest possible unit of data from engine A to engine B, then B transfers the data back to A (we will refer to one such sequence as a *ping-pong*). Since we use a window-based approach to communication, the smallest possible unit of data we can send corresponds to the smallest window allowed by the architecture, which has a size of 16 bytes. Therefore, in all the following experiments, we will fill the window with four 32-bit integers. We will consider the case where the kernels are placed on non-neighbouring AI Engines (with varying distances) in section 3.3.3. For

now, we consider the case where the kernels are placed on neighbouring engines such that they can make use of tile memory access.

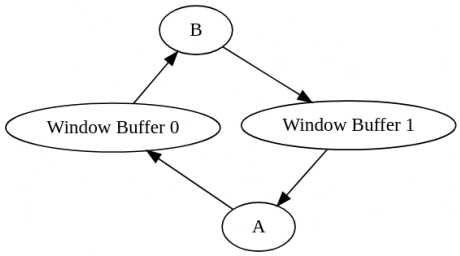

Figure 3.1: *Vitis Analyzer* allows us to inspect the abstract dataflow graph. This graph shows the cyclic dependency between kernels A and B. Vitis Analyzer can generate a graph similar to the one depicted here automatically.

Unfortunately, this design introduces a cyclic dependency which leads to a deadlock at runtime if implemented using the default (synchronous) windows. As explained in section 2.2, a kernel is only invoked after it has acquired the window lock on all its window connections. In this design, A can acquire the lock on its output window, but it cannot acquire the one on its input window. This is because a consumer can only acquire the lock on an input window after the producer has acquired and released the corresponding lock — that is, after the producer has written something to the connection. Therefore, in this design, kernel B has to finish before kernel A can be invoked.[1] But, likewise, kernel B is not invoked because that would require kernel A to write something to its output window first. Overall, both kernels stall.

This issue is resolved by using asynchronous windows. As asynchronous window connections are ignored in the invocation conditions of a kernel, both A and B start executing without waiting for each other. B tries to acquire the lock on its input window but this call blocks until the window has been produced by A and A releases the window lock. Meanwhile, A can acquire the lock on its output window, fill the window and release the lock, at which point B can acquire its input window lock. A blocks waiting for its own input window lock, which it can only acquire once B releases the lock on its output window. See the pseudocode in listing 3.1 for the code which is executed on engines A and B.

[1]Note that in this chapter, A and B may sometimes refer to the kernel as a node in the abstract dataflow graph and sometimes to the AI Engine running this kernel. It should be clear from the context which meaning is intended.

```
a(input window, output window):
    acquire output window lock
    write to output window
    release output window lock

    acquire input window lock
    read from input window
    release input window lock

b(input window, output window):
    acquire output window
    acquire input window

    read from input window
    write to output window

    release input window
    release output window
```

Listing 3.1: Pseudocode of the kernel code executed on engine A and B.

A different question is how we can get the kernels to perform several ping-pongs. In the host code which is responsible for starting execution of the AI Engine grid, an argument n can be passed to the method which runs the graph (`mygraph.run(n)`), with the effect that the graph ends once each kernel has been executed precisely n times (as opposed to running forever). Between invocations of the kernel function, each engine runs the corresponding kernel's main function, which requires some time. To get rid of this overhead, we call `mygraph.run(1)`, create a loop inside each kernel and pass the number of loop iterations as a runtime parameter to the kernel function.[2] This ensures the kernels are only invoked once, thus avoiding the overhead of their main functions, while still allowing us to perform several ping-pongs.

3.2 Timing Methods

In the following, we present two different approaches to benchmark the latency in the order in which they were implemented. The first one, using a wall-clock timer in the host code, has several drawbacks, which is why we finally discarded it. For future reference it may still be worth describing. The second approach makes use of cycle counters which are local to each AI Engine.

3.2.1 Wall-Clock Timer

This initial approach uses the `chrono` library to start and end a wall-clock timer in the host code. This approach was chosen first because it has been used previously in the work of [4] to obtain the bandwidth benchmarks. The following listing provides an excerpt of our host code.

[2]Runtime parameters are kernel arguments the host can change between kernel invocations. How they work precisely is not relevant to this thesis.

```
long long round_trip_time = 0;
Timer timer;

ret = mygraph.run(1);
mygraph.wait();

round_trip_time = timer.stop();
latency = static_cast<double>(round_trip_time)/(2*i);
```

Listing 3.2: Excerpt of the host code starting the AIE graph execution. The variable i should hold the number of iterations to perform.

The main idea is that starting the graph incurs some constant overhead, but as we increase the number of iterations i, this overhead should become negligible. Thus, by setting i to a large number and then taking the average latency over i iterations, we should be able to cancel the constant overhead. Indeed, we have seen that the latency converges to a value of 85.5 nanoseconds (for $i = 10^6$), which when multiplied with the constant processor frequency of $1.25GHz$ results in an average latency of 107 cycles.

This method has the serious drawback that we do not have access to the latency values of each individual data transfer, meaning we cannot compute statistical properties apart from the average. In particular, it would be helpful to know the standard deviation or the interquartile range. Furthermore, it remains unclear how much the starting overhead influences the converged average latency.

3.2.2 Cycle Counter

A superior approach is to measure the number of passed cycles directly. Each AI Engine contains a local cycle counter which can be inspected from within the kernel code. As before, both kernels contain a loop which performs i iterations of the ping-pong. In kernel A, we read the counter's value at cycle c_1, perform one ping-pong and read the counter again at cycle c_2. This method removes the startup overhead that was measured in section 3.2.1 by design, as we start the counter inside the kernel, i.e. on the AI Engine itself (as opposed to in the host). We write the round-trip time $c_2 - c_1$ to an output window after each individual ping-pong, which later allows us to compute the latency of each ping-pong and deduce from this the interquartile range.

In the following, the final kernel code of both A and B is provided. Notice that we use vectorization in the form of `v4int32`. This datatype can hold four 32-bit integers and it ensures that the addition in B (which is necessary to check for correctness) and the `window_readincr` and `window_writeincr` calls are executed as fast as possible.

```
void a(input_window<int32>* in, output_window<int32>* data_out,
    output_window<int64>* cycles_out, output_window<int32>*
    check_out, int32 iterations)
{
  v4int32 rcv = null_v4int32();

  aie::tile tile = aie::tile::current();
  unsigned long long time_start;
  unsigned long long time_end;

  for (int i = 0; i < iterations; i++) {
    time_start = tile.cycles();

    window_acquire(data_out);
    window_writeincr(data_out, rcv);
    window_release(data_out);

    window_acquire(in);
    window_readincr(in, rcv);
    window_release(in);

    time_end = tile.cycles();
    int64 cycle_dif = time_end-time_start;
    window_writeincr(cycles_out, cycle_dif);
  }
  window_writeincr(check_out, rcv);
}

void b(input_window<int32>* in, output_window<int32>* out, int32
    iterations)
{
  v4int32 basisOne = null_v4int32();

  for (int i = 0; i < 4; i++)
  {
    basisOne = upd_elem(basisOne, i, 1);
  }

  for(int i = 0; i < iterations; i++){
    v4int32 rcv;

    window_acquire(out);
    window_acquire(in);

    window_readincr(in, rcv);
    rcv += basisOne;
    window_writeincr(out, rcv);

    window_release(in);
    window_release(out);
  }
}
```

Listing 3.3: Final benchmark code of kernels A and B using a cycle counter.

3.3 Results and Discussion

3.3.1 Latency

We use the method described in section 3.2.2 with 1024 iterations. We ensure kernels A and B are placed on neighbouring engines. We visualize the result in Python with a box plot provided by `matplotlib`. As shown in figure 3.2, the median latency for neighbouring AI Engines is 98.5 cycles with an interquartile range of 0.5 cycles, which means the measurement is highly consistent. It is also worth noting that the difference to the measurement using `chrono` is almost 10 percent (there, we got 107 cycles). This suggests that using wall-clock time in the host might not be an accurate way of benchmarking the AI Engines.

The cycle count of 98.5 is higher than we expected. We have reported this finding to AMD/Xilinx engineers asking them whether this value meets their expectations, but so far we have not received a response.

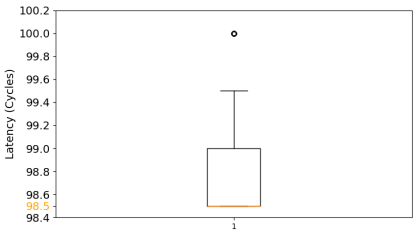

Figure 3.2: Latency when kernels are placed on neighbouring AI Engines and use tile memory access. The median latency is highlighted in orange.

3.3.2 Simulator Trace

Using *Vitis Analyzer*, we can inspect a detailed trace view of a run of the AIE simulator. The AIE simulator models timing and resources of the AIE grid and provides cycle-approximate timing information [9]. For this run, both kernels A and B run for two iterations, though only the first iteration is discussed. The goal of this section is to investigate where the 98.5 cycles reported as median latency in section 3.3 are spent. The trace view reports events based on nanoseconds, which we have converted to cycles by multiplying with the constant processor frequency of $1.25GHz$.

We see that engine B starts the kernel function on cycle 706, slightly before engine A starts its kernel on cycle 740. After 58 cycles, A writes (0,0,0,0) to its output window. 38 cycles later, this data arrives as an input window at B, which then needs only 4 cycles before it writes (1,1,1,1) to its own output window. 125.5 cycles later, this window arrives as an input window at A. This gives a total of 225.5 cycles, which would result in a latency of 112.25 cycles, slightly overestimating the latency measured in hardware. However, the cycle counter in kernel A is started right before data is written to the output window, so it is

likely that not all of the 58 cycles between A starting and A writing to its output window are measured in the hardware measurement. If we remove the 58 cycles A takes before writing its window, we get a corrected total of 167.5 cycles, thus a latency of 82.75 cycles, now slightly underestimating the hardware latency.

This simulation shows that a significant proportion of the time is spent while the data is "in transit" i.e. written to A's output window but not yet available in B's input window. Since these kernels transfer data simply by using a shared memory buffer (tile memory access), this time is likely spent acquiring and releasing window locks. If this is true, it is unclear why acquiring the window locks takes only 38 cycles for the transfer from A to B but 125.5 cycles for the transfer from B to A.

3.3.3 Latency versus Tile Distance

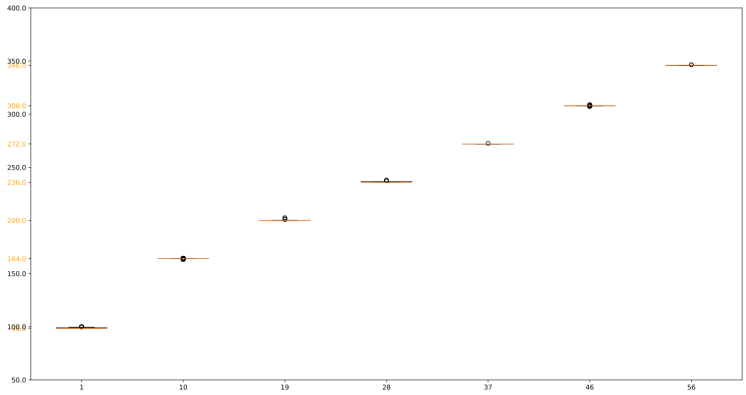

Figure 3.3: The x-axis represents the Manhattan distance between engines A and B. The y-axis represents the latency in cycles. We report the median out of 1024 iterations.

In the previous sections, kernels A and B were mapped onto neighbouring tiles in the AIE grid. If the kernels are placed further away from each other, the latency is expected to increase. This is because the kernels can no longer use tile memory access but have to use the DMA engine instead, which transfers the window data using an AXI4 stream.

In a grid, one possible notion of distance is the L1 norm, also known as Manhattan distance. As the AIE grid has a dimension of 8x50, the maximum Manhattan distance is 56 if the kernels are placed in opposing corners. We measure the latency for seven different Manhattan distances (1, 10, 19, 28, 37, 46, 56) and find that the median latency ($N = 1024$) depends linearly on the distance between the engines. Linear regression gives the line $y = 4.32x + 110$ ($R^2 = 0.989$), i.e. increasing the distance by one tile increases the median latency

by 4.32 cycles. Note that this interpretation is not perfect because only in the case where the distance is 1, tile memory access is used, while all other distances use DMA. We expect tile memory access to have a lower latency than a linear regression based on the distances which use DMA would predict for a distance of 1. Therefore we perform another linear regression using only those distances which are greater than 1. This gets us the line $y = 3.97x + 125$ ($R^2 = 1$), suggesting that non-neighbour window connection latency scales linearly with the distance between the engines. Plugging in $x = 1$, we get a predicted latency for neighbours if they also used the DMA: 128.97 cycles.

We conclude that for neighbouring AI Engines, if they can use tile memory access, their median window latency has been measured to be 98.5 cycles. Based on linear regression, we expect that if neighbouring AI Engines instead used the DMA engine, they would have a window latency of roughly 129 cycles.

Table 3.1: Window Latency Measurements

Distance	Median Latency
1	98.5
10	164
19	200
28	236
37	272
46	308
56	346

CHAPTER 4

Reduce

This chapter describes our design of the Reduce collective function known from MPI, adapted to fit the API provided by AMD/Xilinx for the AI Engine architecture. It provides the implementation of a prototype, an evaluation with cycle counter benchmarks and a cost model able to predict the performance of the implementation. It also discusses some limitations of our design.

4.1 Motivation

As stated in the introduction, it seems currently hard to program the AI Engines. It might be helpful if users could apply concepts already known from multicore programming, such as message passing. For communication between kernels, point to point communication (sending windows from one producer to one consumer) as well as broadcasting (sending windows from one producer to several consumers) operations are already provided by AMD/Xilinx. We choose one operation defined in the MPI (Message Passing Interface) standard, called *Reduce*, to test whether it is possible to implement such a collective function on this hardware given the provided API.

The Reduce collective operation works as follows: Several cores each produce some local data. Each core's data gets combined ("reduced") using a particular function (the reduction function) to end up with a final result. One predetermined core (called the root of the operation) should end up with the final result. The reduction function must be associative and commutative. If, for example, each core produces a single integer and the reduction function is addition, the root should end up with the sum of each core's integer.

We adapt this definition to the AI Engine architecture, where we work with windows of data instead of single integers. Each kernel can provide an array of

local data and the root kernel should end up with a result which is the element-wise application of the reduction function to each core's local data array. All communication between kernels must happen using input and output windows.

4.2 Goals

The goal is to implement a prototype of the Reduce function, benchmark the function and create a cost model for it. Internally, the Reduce function needs to use *window-based communication* to transfer its data to other kernels. We assume the user has n kernel functions (in the following called "user kernels") which they wish to be part of the reduction and which might already. Theoretically, the user kernels might already communicate with each other in some way — we assume this is not the case, but the implementation could be adapted to take preexisting connections into account rather easily. Each user kernel needs to manually call Reduce and pass it some local data in the form of an array. In the root, the call should block until the final result is available. It should be possible to call Reduce several times within a single kernel.

Note that the most straight-forward approach to solve this issue is to use point to point communications which are provided out of the box to connect $n-1$ producer kernels to a root kernel. The producer kernels would send their local data arrays to the root as windows, and the consumer kernel could look at each input window in turn and perform an element-wise reduction operation. This does not work! The reason is that there is a limitation of 14 window connections per kernel (input windows and output windows together), which means that this design would only allow at most 15 kernels to participate in the reduction operation. In the following, we propose a design which allows us to scale to more kernels.

4.3 Design

Let us now discuss the design used to connect the AI engines containing the user kernels. To this end, we ignore all connections the user kernels may already have and focus only on additional connections necessary to enable the Reduce function. We arrange the user kernels in a directed tree of arity M, where each node corresponds to a user kernel and each edge from A to B is a window connection, such that the window is an output window at kernel A and an input window at kernel B. Note that due to the constraint that a kernel can have at most 14 window connections, the arity of this tree can be at most 13 (so each node has at most 13 input windows and 1 output window).

Functionally, leaves only forward their local data in the form of an output window to an interior node. Interior nodes perform the bulk of the work: Given m input windows (for $m \leq M$), $in_1, ..., in_m$, as well as a local data array L and a (commutative and associative) function f, an interior node computes

$$res[i] = in_1[i] \ f \ in_2[i] \ f \ ... \ in_m[i] \ f \ L[i].$$

In other words, the data of all its input windows and its own local data is reduced element-wise according to the reduction function. This produces an array res of the same length as the input windows and the local data. This array is forwarded to the next level closer to the root in the tree. The root of the tree ends up with the final result of the reduction, which can be returned to the user.

Note that going forward (for the implementation and evaluation), we consider only a binary tree, i.e. a tree with an arity of $M = 2$, and more specifically only perfect binary trees (i.e. those which contain $2^L - 1$ nodes). Our implementation of the Reduce function allows the tree to have an arbitrary arity, but the connection logic (see section 4.4.3) works only for perfect binary trees. It could reasonably be expanded to work for higher arities.[1] Note that one consideration is that when using higher arities of the tree, the window size would need to be decreased. This is because all windows of a kernel must be stored in its local data memory (which is limited to at most $128KB$). Previous benchmarks have shown that smaller window sizes allow smaller throughputs of window-based inter-AIE connections [4] (page 46), so it might make sense to reduce the arity of the tree to allow larger window sizes. Furthermore, increasing the arity of the tree means that fewer window connections are available for an end user (since there are only 14 window connections per kernel available in total). So using a binary tree allows us to use the highest possible window size and keeps as many window connections as possible free for the end user's own kernel connections.

4.4 Implementation

Now that we have defined the goals and high-level design choices of the Reduce function, let us discuss some lower-level problems that arose during implementation.

4.4.1 Synchronous or Asynchronous

It was not initially clear whether the communication along the tree should happen with synchronous or asynchronous windows. Usually, synchronous windows should be the the first choice, since they are the default and much better documented by AMD/Xilinx. However, there are at least two important reasons why we ended up choosing asynchronous windows.

The first reason is a limitation which arises when using synchronous windows: Reduce can only be called once per kernel. This is because after the first Reduce consumes the input windows, they cannot be read again within the same kernel invocation. Any attempt to do so will stall the user kernel. Asynchronous windows solve this issue: Once Reduce is called, the input windows and the output window are acquired (this blocks until all input windows have arrived and any output windows from previous calls have been consumed by the next kernel), then the result is produced, and finally the input windows and the

[1] We would have liked to investigate the behaviour of trees with different arities, but debugging and analysing the errors discussed in section 4.6 cost too much time.

output window are released again (this instantly sends the output window to the next kernel).

The second reason has to do with performance: Asynchronous windows allow computation before data arrives. Assume a user kernel (which is an interior node in the binary tree) does some expensive work and at the end calls Reduce. If the connection were synchronous, the kernel could only start executing once both input windows are available. It would perform the expensive computation and only send the output window once the user kernel returns. On the other hand, if the connection were asynchronous, the kernel could start executing right away, without waiting for the input windows. It could perform the expensive operation, and once it gets to the Reduce call it is likely that the input windows have arrived (otherwise, it would block here until they arrive). Looking at the binary tree as a whole, this means that there is more parallelism across levels.

4.4.2 Code Details

Connections along the Reduce tree must be asynchronous. To make this easier to work with, the implementation of Reduce is divided into two functions: We shall refer to these as the *inner Reduce function* and the *outer Reduce function* for the remainder of this thesis. The inner Reduce function (see listing 4.1) assumes all windows are available and handles the main computation of the element-wise reduction. The outer function (see listing 4.2) handles the acquisition and release of all relevant window locks (because the window connections are asynchronous) and then calls the inner function as a helper. It can also handle the case where the user data size is bigger than the window size. The inner function should not be called directly by the user — instead, the user should call the outer Reduce function.

```
template <typename MPI_Op, typename T>
void const_reduce_elementwise_and_send(unsigned n,
  T *local_data,
  std::initializer_list<input_window<T> *> windows,
  output_window<T> *out)
{
  MPI_Op op;
  auto it = aie::cbegin(local_data, n);
  for (; it < local_data + n; it++)
  chess_prepare_for_pipelining {
    T acc = *it;
    for (auto window : windows)
    chess_prepare_for_pipelining {
      acc = op(acc, window_readincr(window));
    }
    window_writeincr(out, acc);
  }
}
```

Listing 4.1: Inner Reduce function, which assumes all windows are available.

Template Arguments

The function call to Reduce requires two template arguments, MPI_Op and T. In general, template arguments should be used sparingly, since they can lead to much bigger code sizes because the function is compiled to separate assembly code for each template which is instantiated. The following should briefly justify the use of these.

The argument T defines the type of each window element. This makes it possible to call our implementation with any data type supported by the API. In our analyses, we have only considered the 32-bit integer type `int32`.

MPI_Op defines which function is used to reduce the input windows and the local data and can be either MPI_Sum or MPI_Max. These are structs we have defined ourselves in a header file and can be included by the user, which each define an $operator()$ to either add their two inputs or take the their maximum. Depending on the template argument, the variable op is an instance of one of these two structs. Calling $op(acc, window\_readincr(window))$ invokes either the addition or the maximum function of the two inputs. The set of possible functions can easily be extended, which is the reason why this template parameter exists. In our analyses, we only consider MPI_Sum, so we only perform element-wise addition.

Initializer List

An early implementation decision was how to pass the input windows. The most straight-forward way is that Reduce accepts exactly two input windows. However, what happens if we wanted to extend the design to 3-ary trees, or to cases where the binary tree is not complete, i.e. there is some interior node with only one child? That node could not call the Reduce function, which means the function would need to be implemented twice: Once with two input windows and once with a single one.

A simpler and more scalable approach seemed to pass a variable number of input windows to Reduce. The obvious choice was to use a $std :: vector$, but this led to a compiler error. A workaround is to use an initializer list, which can have a variable length, as long as it is known at compile time. The user passes their input windows inside curly braces. As stated before, as we only consider binary trees, in retrospect it might have made more sense to reduce the flexibility and complexity here.

Inner Function

The core part which performs the accumulation is located in the inner Reduce function (see 4.1). The outer loop iterates over each element in local data. For each local data element, the inner loop iterates over all input windows. Each input window reads its next element e and calls the reduction function on the partially reduced variable acc and on e. After iterating over all windows, the resulting element is written to the output window. This implements an element-wise reduction.

Outer Function

The previously discussed inner Reduce function assumes all windows are available, but as explained in section 4.4.1, the connections need to be asynchronous. Therefore, an outer Reduce function (see 4.2) handles all necessary window locks, calls the inner function once all windows are available, and releases all window locks afterwards. To avoid deadlocks, all locks are released in the reverse order of acquisition.

```
1  template <typename MPI_Op, typename T>
2  void async_const_reduce_elementwise_and_send(unsigned n, T *
       local_data, std::initializer_list<input_window<T> *> windows,
       output_window<T> *out)
3  {
4    unsigned n_connection = out->winsize / sizeof(T);
5    unsigned n_chunks = n / n_connection;
6
7    for (unsigned cur_chunk = 0; cur_chunk < n_chunks; cur_chunk++)
8    chess_prepare_for_pipelining {
9
10     for (auto window : windows)
11     chess_prepare_for_pipelining {
12       window_acquire(window);
13     }
14
15     window_acquire(out);
16
17     const_reduce_elementwise_and_send<MPI_Op, T>(n_connection,
       local_data+cur_chunk*n_connection, windows, out);
18
19     window_release(out);
20     for (auto it = std::rbegin(windows); it != std::rend(windows);
       ++it)
21     chess_prepare_for_pipelining {
22       window_release(*it);
23     }
24
25   }
26 }
```

Listing 4.2: Outer Reduce function, which handles locks and splits user data into chunks.

For a node which calls Reduce, this means that the call blocks until all inputs have arrived, the result has been computed and written to the output window, and any previously written result has been processed by the parent.

One requirement was that it should be possible to call Reduce several times inside a kernel. Assume that a child sends two consecutive windows to its parent which belong to *different* operations. Does this outer function guarantee that the operations are not mixed up?

To see that this is indeed guaranteed, consider two interior nodes A and B and the root R. A calls Reduce to send its first window to R. If it now tries to send its second window to R, the call $window\_acquire(out)$ blocks until the first window is consumed by R - or more specifically, until R calls $window\_acquire$ and $window\_release$ on the connection with A. But R only calls $window\_release$

on the connection with A at the end of the outer function, so in particular after calling *window_acquire* on both its inputs. So R consumes the first output window of both A and B before A can compute its second output window and send it to R. This guarantees a consistent ordering even if Reduce is called several times per kernel, as long as the number of calls is equal for all kernels.

Chunking User Data

We wanted to decouple the user's data size from the window size. For example, it should be possible to reduce $2KB$ of data over a $512B$ connection by sending four separate windows (in the following called *chunks*), all of which are sequentially reduced element-wise. To the end user, this should not change anything in the function call.

The outer function achieves this by first computing the number of chunks by dividing the total number of elements (to be provided by the user as an argument n) by the number of elements which can be sent in one window, *n_connection*. For each chunk, all locks are acquired, data is reduced and sent, and all locks are released again. Chunking can therefore be viewed as n_{chunks} separate Reduce calls in series, each of which reduces *n_connection* elements. The ordering correctness proof sketched out in section 4.4.2 can be applied to this series of Reduce calls, which shows that chunking preserves the correctness of the final result.

In other words, this means that we always have pipelining across successive calls to the Reduce function, but as long as the window size matches the data size (so there is only one chunk), there is no pipelining within a single Reduce call. If the data size is chosen larger than the window size, there is pipelining within a single Reduce call due to our chunking mechanism: The outer function splits the user data into smaller windows and calls the inner Reduce function on each window separately — to the inner function, this looks like several independent calls. Even though the chunking mechanism works, in the evaluation we will not consider it due to time constraints. We will assume that the data size is always equal to the window size. However, in the limitation analysis, we will again assume that chunking is possible.

Vectorisation

The current implementation only uses the scalar part of the AI Engines to compute the reduction. This is inefficient as each AI Engine contains a vector processor. Chunking would allow us to use a window size which exactly corresponds to a vector size (as specified by the AMD/Xilinx API), which means the outer loop of the inner Reduce function (listing 4.1) can be removed if the scalar addition operation is replaced by a vector addition. This has not been implemented to keep the benchmarks consistent and the Reduce code simple.

4.4.3 Automatic Tree Connection

Even though the Reduce function is only meant to be a proof-of-concept, we still want to make it as simple as possible for a user to incorporate this functionality into their code. As described in the current chapter up to this point, from the perspective of a kernel programmer, this simplicity has been achieved: The programmer only needs to add one function call to Reduce to each kernel which participates in the reduction. However, this assumes that all relevant kernel instances are already connected in an appropriate binary tree.

To connect kernel instances A and B, one needs to insert the following line in the code of the graph constructor:

```
connect<window<WINSIZE>> con(async(A.out[i]), async(B.in[j]));
```

Here, con is an arbitrary name we give this connection and `async` is needed to mark both ends of the connection as asynchronous window connections. If a user had 63 kernel instances which they wished to participate in a Reduce, they would need to write the above line 62 times (as in a binary tree there is one edge per node, except for the root), keeping track of exactly which instances they are connecting and which input and output ports have or have not been used already. This quickly gets cumbersome and reduces the usability of the project. Even for benchmarking and testing, this would be very error-prone.

For these reasons, we have implemented a function which accepts kernels and connects them in a binary tree, which can be called in the graph constructor.

```
void connect_tree(kernel* root, unsigned n_interiors, kernel*
    interiors, unsigned n_leaves, kernel* leaves){
  unsigned n = n_interiors + n_leaves;

  for (int i = 1; i <= n; i++){
    printf("Connecting element %d / %u\n", i, n);
    kernel* cur = k(i, root, n_interiors, interiors, n_leaves,
    leaves);
    unsigned partner_index = static_cast<unsigned>(std::ceil(
    static_cast<double>(i)/2) - 1);
    kernel* partner = k(partner_index, root, n_interiors, interiors
    , n_leaves, leaves);

    connect<window<MPI_CON_SIZE>> temp(async(cur->out[0]), async(
    partner->in[(i-1) % 2]));
  }
}
```

Listing 4.3: This function can connect kernels given to it as arguments in a binary tree for the user.

Abstractly, this function uses a contiguous array of all nodes which contains first the root, then all interior nodes, and finally all leaves. In this array, any node at index i can find its parent at index $\lceil \frac{i}{2} \rceil - 1$. For reasons explained in 4.6.5, the root, interior nodes and leaves must be passed to the function in separate arrays. The helper function k returns a pointer to the appropriate kernel instance, treating all kernels as a contiguous array as described above.

What about ports? In this implementation, we assume the first output port of each kernel is used to send the output window to a node's parent, and the first two input ports are used to receive input windows from a node's children. These ports cannot be used by the kernels for anything else. In particular, this leads to compatibility issues if a programmer already has a graph and later decides to use the Reduce function. They are forced to change their existing connections in order to free up output port 0 and inputs ports 0 and 1 of all relevant kernels to make use of this connection function. This issue could be solved by passing a map to connect_tree, which maps kernel instances to port indices. This tells the function which input and output ports it should use for particular kernel instances. For simplicity, this has not been implemented.

4.5 Evaluation

4.5.1 Cost Model

It can be useful to be able to roughly predict how many cycles our implementation of Reduce takes for a particular configuration of window sizes, data sizes and tree depths. We have therefore developed a simple cost model. We denote by n_{window} the number of elements per window and by n_{data} the number of elements in each node's local data array. Note that we always send 4-byte integers, so we have the relations

$$n_{window} = \frac{WS}{4}$$
$$n_{data} = \frac{DS}{4},$$

where WS is the window size and DS the data size. We will assume for simplicity that $WS = DS$, so we do not take the chunking mechanism into account, since it would introduce pipelining within a single Reduce operation, making it more difficult to model accurately.

Our model makes several more assumptions:

1. We consider a perfect binary tree, that is, it contains exactly $n = 2^L - 1$ nodes for some $L \geq 3$. L is the depth of the tree, or in other words, the number of levels.

2. Each level of the tree executes completely in parallel. Therefore, the cost of one level is equal to the cost of one node.

3. Leaves do not process data, they only send all of their local data elements.

4. A node can only start executing once both of its children are done. In other words, levels of the tree are executed sequentially.

5. All non-leaf nodes iterate over n_{window} elements. For each element, they perform one accumulation operation per input window.

6. Each node performs some additional operations, such as acquiring and releasing locks, which do not depend on the number of elements.

To deduce a cost model from these assumptions, we have relied on the AIE compiler report which, for some loops, reports the latency of the loop body as well as the initiation interval (II). The latency of the loop body tells us how many cycles a single iteration takes to execute. The II, on the other hand, is a measure of pipeline parallelism, as it tells us after how many cycles of a certain iteration the next iteration can start executing. Unfortunately, these numbers are only reported for certain loops and not all of them, and we have been unable to ascertain which conditions a loop must fulfill to be reported. Furthermore, for all loops the compiler has reported on, the latency of the loop body equals the II, so these loops cannot be pipelined.

Note that in the cost model as well as in the evaluation, we will use the terms *Leaf Time*, *Level Time* and *Tree Time*. We define the leaf time to be the number of cycles required by each leaf node to execute one call of the Reduce function, by level time the number of cycles required by any non-leaf node to execute one call of the Reduce function assuming all inputs are available, and by tree time the number of cycles required by the root to execute one call of the Reduce function assuming the whole tree has to be executed.

Let us proceed in the following way: We first take note of all the IIs we get from the compiler report. We use these to estimate the cycle count of the leaves and the non-leaves separately. Then we combine them to estimate the cycle count of the whole Reduce tree. As we will see, using only the data provided by the compiler report results in a crude underestimate because for many operations, there are no cycles reported — in particular for all operations which are not part of a loop. Therefore we introduce various correction parameters. After measuring the real cycle count of several configurations, we are able to fit these parameters such that the final cost model becomes quite accurate.

Initiation Intervals

The first II we consider is that of the inner loop of the inner Reduce function (shown in listing 4.4), which is reported to be 30 cycles. This is surprising because *op* is a scalar addition, which should take one cycle, and the remaining operations are reads and writes from and to local data memory[2]. In particular, all of the input windows must already be in local data memory by the time this loop is executed because window synchronization happens in the outer Reduce function. We would have expected an II of approximately 3 cycles: 1 to read both `acc` and the next window element, 1 to add them and 1 to write the result to `acc`. As for the cost model, we now know that in assumption 5, the cycle count per window element per input window is (at least) 30.

```
1 for (auto window : windows)
2   chess_prepare_for_pipelining {
3     acc = op(acc, window_readincr(window));
```

[2] `window_readincr` is a read from a window buffer and `acc` is stored on the kernel's stack.

```
4 }
```

Listing 4.4: The inner loop of the inner Reduce function has an II of 30 cycles.

From the compiler report, we further know that the II (and the loop body latency) of the loop which acquires windows in the outer Reduce function is 48 cycles, while the II (and the loop body latency) of that which releases them is 45 cycles. All other loops do not have a reported II — in particular, the outer loop of the inner Reduce function (which iterates over the elements of local data) and the main loop of the outer Reduce function (which iterates over chunks).

Leaf Time

Now we try to model the cycle count required by each leaf. Leaves do not have input windows, so the inner loop of the inner Reduce function is not executed. The outer loop, for which the compiler does not report the II, iterates over all elements of local data, loading them and then writing them to the output window unmodified. The cycle count required to do so is clearly proportional to the number of window elements, but the proportionality parameter is not known to us because we do not know for sure how many cycles the operations take. Let us introduce a parameter $T_{leaf,element}$ here to account for the per-element cycle count of a leaf.

Considering now the outer Reduce function, we see that leaves do not need to acquire and release any input windows because there are none. They must only acquire one output window and release it. Acquiring the window takes 48 cycles and releasing it takes 45 cycles, which sums to 93 cycles.[3]

Taken together, we expect the leaves to take 93 cycles to deal with window locks, and then $T_{leaf,element}$ cycles per window element:

$$T_{leaf} = n_{window} \cdot T_{leaf,element} + 93 \tag{4.1}$$

Level Time

Predicting the level time, i.e. the time of a non-leaf node, is quite similar to the prediction of leaf time, with the difference that now there are two input windows. In the inner Reduce function, the II of the inner loop is 30 and there are two input windows, so the inner loop should in total take 60 cycles. The outer loop, iterating over all of the window elements, executes some additional instructions (loading, storing) apart from the inner loop. As in the leaf time, we capture this with the parameter $T_{level,element}$ which we expect to be similar to $T_{leaf,element}$.

In the outer Reduce function, there are three windows in total, all of which must be acquired and released. The cycle count required for this is $3 \cdot 48 + 3 \cdot 45 = 279$.

Together, we expect a single level of non-leaves to take 279 cycles for locks and $60 + T_{level,element}$ cycles to reduce each window element:

$$T_{level} = n_{window} \cdot (60 + T_{level,element}) + 279 \tag{4.2}$$

[3] This is based on the loop body latency of the acquisition loop (48) and the release loop (45). We assume acquisition/release as independent operations take the same number of cycles.

Tree Time

For the tree time, we make use of our assumptions 2, 3 and 4. The leaves are executed first and the whole level takes time T_{leaf} because we assume all leaves can be executed in parallel. After they are done, all levels of non-leaves are executed sequentially, each level taking time T_{level}. If the depth of our tree is L, we have one level of leaves and $L-1$ levels of non-leaves. Therefore, we estimate the total tree time as

$$T_{tree} = T_{leaf} + (L-1) \cdot T_{level} \tag{4.3}$$

As we will see, all of these are underestimates because we are neglecting the cycle counts of several instructions. Let us now collect measurements and return to the cost model afterwards to revise it. In order to do so, we first need to clarify how the measurements have been obtained.

4.5.2 Methods

This section discusses the methods used to obtain benchmarks of the Reduce implementation. The approach based on measuring the wall-clock time using `chrono` was already unreliable when we used only two kernels (see section 3.3.1), so it was clear that cycle counters should be used. Cycle counters are local to an AI Engine, so the question became which engine we should use for the measurement. Under the assumption that all kernels start simultaneously, it makes most sense to measure the cycle count on the root. Since the Reduce call must first wait for all data from the children to arrive, the counter in the root measures the cycles between the moment when the leaves start and the moment Reduce returns the final result in the root.

This is only accurate if all involved kernels are invoked at the same time. Luckily, there is a compiler flag to ensure this which is called `--broadcast-enable-core`. To quote from the user manual: *"Enable all AI Engines associated with a graph using broadcast. This option reserves one broadcast channel in the array for core enabling purpose. The default is true."* [8]. As far as we know, there is no better guarantee to ensure the kernels are synchronized, so for our purposes, we shall assume the kernel functions are invoked simultaneously.

Each kernel starts by initializing an array of local data with numbers from 1 to n_{data}, allocated on the stack. After this, each kernel enters a loop which calls Reduce several times (so there is pipelining across the several Reduce calls). The root measures the start cycle and end cycle inside this loop, right before and after the Reduce call. We are most interested in the cycle count required for the first iteration (recall that we call this *tree time*, denoted by T_{tree}). However, the subsequent iterations may provide useful evidence that our measurements are reasonable and that the cost model is sensible. Recall that in the first iteration, the Reduce call in the root has to wait for all other levels of the tree to produce their results and forward them until they arrive at the root, before it can start executing. Due to pipeline parallelism, we expect that in subsequent iterations, data from the root's children should already be available once the root calls

Reduce again. What we measure in the root in subsequent iterations is therefore only the cycle count required by the root to compute the result, which we have called *level time*. Note further that the initialization of local data (which is necessary for the correctness check) should not influence the result because the same initialization code is executed on all kernels and thus should take the same number of cycles (or differ negligibly compared to the cycle count of the Reduce operation). The root starts the cycle counter only after its initialization has been done, and at that moment all remaining kernels should also have finished initialization.

We expect the following: For fixed n_{window} and n_{data}, the level time should stay constant when we scale the depth of the tree. This would match the cost model prediction of the level time, which assumes it to be independent of the depth.

The benchmarks have been obtained by choosing different $(L, n_{window}, n_{data})$ configurations. Each configuration had to be compiled individually. Unfortunately, we could not scale L as much as we would have liked (for details see section 4.6).

The level time plot of a configuration is computed by running the AIE graph with that configuration once, with 1024 iterations, and discarding the first cycle count, leaving 1023 values. The tree time plot of a configuration is computed by running the AIE graph 10 times (by completely restarting the host program to ensure there is no pipelining), each time considering only the first cycle count, giving 10 values. All box plots have been created in Python 3 with `matplotlib`. In tables, we will report the median values (of 1023 level time values and 10 tree time values, respectively).

4.5.3 Results and Cost Model Evaluation

In the following, we present the Reduce benchmarks and compare them to the estimates given by the cost model. Table 4.1 presents all tested configurations of tree depth and window size (L, WS). Tree time is always denoted as TT, level time as LT. The columns report the following values:

- TT Prediction and LT Prediction report the tree time and the level time estimated by the cost model.
- TT Measurement and LT Measurement report the tree time and the level time which have been measured by running the design on hardware.
- Leaves reports the leaf time estimated by the cost model.
- TT Error and LT Error report the percentage error of the model estimates rounded to three decimal points, computed as

$$\frac{T_{measured} - T_{predicted}}{T_{measured}}.$$

Table 4.1: Reduce Measurements with Initial Cost Model Predictions

L	WS	TT Prediction	TT Measurement	TT Error	LT Prediction	LT Measurement	LT Error	Leaves
3	16	1155	1246.5	0.073	527	729	0.277	101
4	16	1682	1783.5	0.057	527	729	0.277	101
5	16	2209	2247.5	0.017	527	729	0.277	101
6	16	2736	2780	0.016	527	729	0.277	101
3	8192	258699	379373	0.318	127255	170384	0.253	4189
4	8192	385954	551618.5	0.3	127255	170384	0.253	4189
3	4096	129675	189946.5	0.317	63767	85391	0.253	2141
4	4096	193442	276165.5	0.3	63767	85404	0.253	2141
5	4096	257209	362667.5	0.291	63767	85392	0.253	2141
6	4096	320976	449123	0.285	63767	85391	0.253	2141
3	2048	65163	94189	0.308	32023	42895	0.253	1117
3	1024	32907	47343	0.305	16151	21647	0.254	605
3	512	16779	23907	0.298	8215	11023	0.255	349
3	256	8715	12208	0.286	4247	5709	0.256	221
3	128	4683	6344	0.262	2263	3053	0.259	157
3	64	2667	3408	0.217	1271	1725	0.263	125
3	32	1659	1949	0.149	775	1061	0.27	109

Before we compare the measurements to the cost model, let us consider some figures which visualise the measurements reported in table 4.1. Figures 4.4, 4.5 and 4.6 show the level time as a function of the depth L, for three different window sizes ($WS = 16, 4096, 8192$). As we can see, regardless of the window size, the level time stays constant as we increase the depth of the tree. Figures 4.1, 4.2 and 4.3 show the tree time as a function of the depth L, again for the three window sizes ($WS = 16, 4096, 8192$). The plots strongly suggest a linear relationship between the tree time and the depth of the tree. Finally, figures 4.7 and 4.8 show the level time and the tree time as functions of the window size, for a fixed depth of 3. Note that the x-axis in the last two figures is logarithmic, so the plots suggest that both the tree time and the level time depend linearly on the window size, assuming the depth is fixed.

Even though all of these observations support the general structure of our cost model, it is evident from table 4.1 that the model severely underestimates both the tree time and the level time. The reason for this is that our current model ignores the cycle counts of those instructions which are not in a loop for which the compiler reports the II. Most importantly, operations such as loading and storing in our estimation of the level time (see equation 4.2) might take more than our arbitrary 2 cycles, or there might be instructions or NOPs inserted by the compiler. If the cycle count we estimate for the outer loop of the inner Reduce function is wrong, this error scales linearly with both the window size and the depth of the tree. To give us the highest possible degree of freedom to correct errors such as these, we introduce several new parameters into our cost model. The corrected estimation equations are as follows:

$$T_{leaf} = n_{window} \cdot T_{leaf,element} + 93 + T_{leaf,constant} \tag{4.4}$$

$$T_{level} = n_{window} \cdot (60 + T_{level,element}) + 279 + T_{level,constant} \tag{4.5}$$

$$T_{tree} = T_{leaf} + (L-1) \cdot T_{level} + (L-1) \cdot (n_{window} \cdot T_{tree,element} + T_{tree,constant}) \tag{4.6}$$

These correction parameters are expected to capture the following missing

cycle counts in our model:

- $T_{leaf,element}$ and $T_{level,element}$ capture cycles spent in the outer loop of the inner Reduce function, except for those spent in the inner loop.
- $T_{leaf,constant}$ and $T_{level,constant}$ capture cycles spent outside of the outer loop of the inner Reduce function, except for those spent acquiring and releasing locks.
- The correction term of the tree time lets us decouple our prediction of the tree time from that of the level time. This compensates for the fact that as the depth increases, the tree time might not grow exactly linearly with the level time but instead with a slight variation of it. $T_{tree,element}$ and $T_{tree,constant}$ are the offset per element per level and the offset per level, respectively. Without this correction, we have not managed to fit the tree time accurately.

We fit the correction parameters by hand by trying to minimize the average of the percentage errors of first the level time and then the tree time. This was done using only the first 10 rows shown in table 4.2. As we can see in that table, using these correction parameters we can perfectly predict the level time. As for the tree time, even though we cannot predict it as well as the level time, all of the errors are below 4% and usually even below 1%.

The parameters we have found are as follows:

- $T_{leaf,element} = 17$
- $T_{leaf,constant} = 0$
- $T_{level,element} = 23$
- $T_{level,constant} = 120$
- $T_{tree,element} = 1$
- $T_{tree,constant} = -214$

Using these corrections, we get the final closed form solutions of the cost model:

$$
\begin{aligned}
T_{leaf} &= 17 \cdot n_{window} + 93 \\
T_{level} &= 83 \cdot n_{window} + 399 \\
T_{tree} &= 17 \cdot n_{window} + 93 + (L-1) \cdot (84 \cdot n_{window} + 185)
\end{aligned}
$$

Table 4.2: Reduce Measurements with Final Cost Model Predictions

L	WS	TT Prediction	TT Measurement	TT Error	LT Prediction	LT Measurement	LT Error	Leaves
3	16	1203	1246.5	0.035	731	729	-0.003	161
4	16	1724	1783.5	0.033	731	729	-0.003	161
5	16	2245	2247.5	0.001	731	729	-0.003	161
6	16	2766	2780	0.005	731	729	-0.003	161
3	8192	379343	379373	0	170383	170384	0	34909
4	8192	551560	551618.5	0	170383	170384	0	34909
3	4096	189903	189946.5	0	85391	85391	0	17501
4	4096	276104	276165.5	0	85391	85404	0	17501
5	4096	362305	362667.5	0.001	85391	85392	0	17501
6	4096	448506	449123	0.001	85391	85391	0	17501
3	2048	95183	94189	-0.011	42895	42895	0	8797
3	1024	47823	47343	-0.01	21647	21647	0	4445
3	512	24143	23907	-0.01	11023	11023	0	2269
3	256	12303	12208	-0.008	5711	5709	0	1181
3	128	6383	6344	-0.006	3055	3053	-0.001	637
3	64	3423	3408	-0.004	1727	1725	-0.001	365
3	32	1943	1949	0.003	1063	1061	-0.002	229

4.6 Limitations

This section discusses problems that arise when compiling the benchmark code for hardware. In particular, we have encountered difficulties when scaling the number of kernels participating in the Reduce operation, scaling the size of the arrays to be reduced, and scaling the window size of the connections between the kernels. Optimally, we would like to be able to scale the number of kernels to 400, which is the total number of engines in the AIE grid. Regarding the data size and the window size, it is not initially clear what the theoretical maximum is, but intuitively, we want to scale the data size to a value as large as possible and we want the window size to freely scale anywhere between 16 bytes (which is the minimum) and the data size. All of these scaling issues arise only when compiling for *hardware*. The *x86simulator*, which is used to verify the logical correctness of a design, has no problems when scaling these values. Rather, the issues are related to memory limits of the AI Engines, placement of the kernels and routing between them.

4.6.1 Placer Timeout

We first notice that as n increases, the compiler spends much more time in the placement stage. It is not fully clear what happens during this stage, but it appears that the compiler tries to find hardware locations for all buffers, connections, kernels etc. such that all constraints imposed by the user and the graph layout are fulfilled. Such user constraints can be that a kernel should be placed on a specific engine, or that the stack of an engine should be placed in a particular buffer, and so on. Initially, we only provide the constraint that each kernel should be placed on a separate engine, in which case the placement stage successfully finishes for $n \leq 63$ (considering only perfect binary trees, that is, $n = 2^L - 1$). However, with $n = 127$, the placement stage of the compiler times

out with the following error message:

```
WARNING: [aiecompiler 47-1001] Placer has timed out.
The problem space is likely too large to solve.
```

We need to help the placer to find a solution before timing out, so we must reduce the problem space. To this end, we introduce artificial location constraints for each kernel. For lack of a better strategy, these constraints are chosen in the following way: We fill the AIE grid column-wise, starting in the bottom left corner, with all leaves, then all interior nodes and finally the root. We only place a kernel on every second engine, leaving every other engine empty. Intuitively, we expect this to help the placer and the router to find space for connections and buffers. This is the setup which has been used for all Reduce benchmarks for consistency.

4.6.2 Memory Background

The scaling issues we will describe are mostly manifestations of memory constraints, hence this section addresses some background knowledge. Most importantly, each physical AI Engine contains $32KB$ of local data memory. Each AI Engine can also access the local memory of at most three neighbouring engines. This means that an AI Engine has an absolute limit of $4 \cdot 32KB = 128KB$ of local memory it can use. Trying to exceed this limit leads to a compilation error.[4]

The local data memory of an engine must provide enough space for all of the input/output windows of the corresponding kernel, its stack and its heap. Note that by default, all input/output windows are double-buffered. This means that each logical connection with a window size of W bytes requires $2 \cdot W$ bytes of data memory. The stack is *"[u]sed as a standard compiler calling convention including stack-allocated local variables and register spilling."* [8], while the heap is *"[u]sed for allocating any remaining file-scoped data that is not explicitly connected in the user graph."* [8]. The stack and the heap both have a default size of $1KB$, respectively, but these sizes can be modified using compiler flags. The user guide states the following important constraint on the size of the heap and the stack: *"The stack, heap, and sync buffer (32 bytes, includes the graph run iteration number information) are allocated up to 32768 bytes of data memory."* [8]. For simplicity, we shall refer to the total memory required by the **s**tack, **h**eap, and **s**ync buffer as *SHS memory*.[5]

4.6.3 Limitations due to Memory Constraints

Now we present three distinct limitations which cause the memory-related errors we have encountered when compiling the benchmark code for hardware.

[4]The error message is: `ERROR: [aiecompiler 77-5387] The total memory used by node mygraph.ROOT (i14) exceeds the per-core memory limit of 131072 KB`. We strongly suspect this to be a spelling mistake, instead referring to $131072B$, which equals $128KB$ and matches all the descriptions in the documentation.

[5]This is our own terminology, but we find it helpful to avoid cluttering the text with repetitions of its definition.

Window Size

In the graph used for the Reduce benchmark, the root is the node with the largest number of input/output windows, so we will focus on the root. It has four windows in total, namely two input windows to receive the partially reduced arrays from its children, one output window to send the final result of the reduction to the Linux host, and one output window to send the cycle count of each iteration to the host. Each window connection is double-buffered. With a window size of $8KB$, this means the root uses $4 \cdot 2 \cdot 8KB = 64KB$ just to buffer windows. Additionally, it needs some non-zero amount of SHS memory, made up primarily of stack memory to store local variables. As elaborated in section 4.6.2, each kernel has a hard limit of $128KB$ of data memory it can use. As SHS memory is limited to $32KB$, we expect compilation to succeed with a window size of $8KB$, since the kernel uses at most $64KB + 32KB = 96KB$ of data memory, which is below the hard limit — and it indeed compiles successfully. Once we increase the window size to $16KB$, the root requires $128KB$ just to buffer windows, which leaves no space for SHS memory. Because a kernel always requires some non-zero amount of SHS memory, this should not be able to compile. Trying to do so indeed raises the error described in section 4.6.2.

This means there is an upper bound of $8KB$ on the window size. This bound could be improved by enforcing single buffering instead of double buffering or by decreasing the number of windows in the root. The first option is not optimal because double buffers ensure that a window can be continually written to (compare section 2.2). In other words, enforcing single buffering would free up space allowing us to increase the window size, but in turn it could decrease the throughput due to wasted cycles. The second option is infeasible for benchmarks as we would have to give up either the correctness check or the cycle count information.

Data Size

We have already established that a kernel's SHS memory is limited to $32KB$. This clearly implies that a kernel's stack space is limited to $32KB$. Each kernel which is part of the Reduce operation has to allocate an array of local memory on its stack in order to pass the array to the Reduce function. Since this array is allocated on the kernel's stack, we have a theoretical upper bound on the size of the array (which we call data size) of $32KB$. As a reminder, the window size and the data size do not necessarily have to be equal in our implementation due to the chunking mechanism. As long as the data size is a multiple of the window size, the Reduce function automatically splits the local data arrays into several chunks (each of size equal to the window size).

This bound cannot be improved because the limit of $32KB$ is imposed by the hardware. If a user requires to reduce data of a size larger than this, our recommendation would be to generate the local data sequentially in blocks of less than $32KB$, each time reducing the current block before generating the next one.

Number of Kernels

We have seen that a kernel might need more memory than a single engine can provide. We can use this knowledge to estimate an upper bound on the number of kernels we can place on the AIE grid, depending on the memory requirements of the kernels. Let WS be the window size and DS the data size.

Let us assume that the total memory requirement of kernel k is $M_k = WIN_k + SHS_k$, where WIN_k is the memory needed for all window buffers and SHS_k is the SHS memory. Further, assume that $SHS_k = DS + \epsilon$, where DS is the data size — this states that a kernel's SHS memory is mainly comprised of the local data array[6]. Leaves have one output window, interior nodes have two input windows and one output window and the root has four windows in total. Therefore, due to double buffering, $WIN_{leaf} = 2 \cdot WS$, $WIN_{interior} = 3 \cdot 2 \cdot WS$, and $WIN_{root} = 4 \cdot 2 \cdot WS$. We assume that the parameters (WS, DS, ϵ) are chosen in such a way that the memory requirement of each kernel is below the per-kernel memory limitation of $128KB$. Then we can compute the total memory requirement of the whole graph, assuming a perfect binary tree of depth $L \geq 3$, as

$$M_{total} = M_{root} + (2^{L-1} - 2) \cdot M_{interior} + 2^{L-1} \cdot M_{leaf} \tag{4.7}$$

Since each engine provides $32KB$ of local data memory, we can compute a lower bound on the required number of engines as

$$N_{engines} \geq \lceil \frac{M_{total}}{32KB} \rceil \tag{4.8}$$

The lower bound is only met if the total memory can be appropriately distributed on the available engines and the kernels can be placed in such a way that they have access to all their required memory. More engines might be needed because of the spatial nature of the device. As the AIE grid contains 400 engines, solving this inequality for L provides a limit on the depth of the tree and therefore on the number of kernels, for given window size and data size. We start by noting that equation 4.8 implies

$$400 > \frac{M_{total}}{32KB} - 1$$

Plugging in the definition of M_{total}, it is straight-forward to find that

$$L < \log_2(\frac{12832KB - M_{root} + 2 \cdot M_{interior}}{M_{interior} + M_{leaf}}) + 1$$

Using the definitions of the different kernel memories, we finally get a limit on the depth of the tree, depending on the window size, the data size and the SHS remainder term ϵ:

$$L < \log_2(\frac{12832KB + 4 \cdot WS + DS + \epsilon}{8 \cdot WS + 2 \cdot DS + 2 \cdot \epsilon}) + 1 \tag{4.9}$$

[6] ϵ compensates for the facts that the stack contains other local variables apart from the local data array, and that the SHS memory also contains the sync buffer. We assume the heap is not used.

For example, plugging in $(WS = 8KB, DS = 24KB, \epsilon = 1KB)$ yields a limit of $L < 7.8$, which (since we are dealing with perfect binary trees) implies $L \leq 7$. To use a configuration which was part of the benchmarks, $(WS = 8KB, DS = 8KB, \epsilon = 1KB)$ yields a limit of $L \leq 8$.

It is worth reiterating that even below this theoretical limit, it is possible (and, in fact, likely) that the compiler does not find a way to place the kernels in such a way that all of them can satisfy their memory requirement within neighbouring tiles, even though the AIE grid as a whole would provide enough memory for the configuration. These computations only take into account the pure memory requirements, without regards for the locality requirements.

Conclusion to Memory-Related Limitations

In summary, these are the limitations which are related to memory:

1. As the root has many window connections, increasing the window size quickly leads to the root exceeding the maximum per-kernel memory limit of $128KB$ just to buffer the windows. This limits the window size to $8KB$.

2. Each kernel's local data must be allocated on the stack. Since the SHS memory of each kernel is limited to $32KB$ and the stack is part of SHS memory, the data size is limited to $32KB - \epsilon$ for some small but non-zero ϵ.

3. If the window size and data size are chosen large enough, each kernel needs more memory than a single engine can provide. Even if locality constraints are ignored, i.e. all memory is assumed to be freely divisible and accessible from everywhere, this limits the number of kernels as a function of window size and data size.

4. Each engine can only access the local data memory of itself and at most three of its neighbouring engines. This locality constraint can imply that even if there is enough total memory for a certain number of kernels according to limitation 3, the kernels cannot be placed in such a way that the memory a kernel requires is close enough.

Limitation 4 is particularly difficult as we increase the number of kernels, because as explained in section 4.6.1, we cannot rely on the compiler to find a placement even if one exists as the problem space is too large — instead, we have to provide location constraints ourselves. But constraining the locations of kernels without an elaborate system makes it highly unlikely that each kernel has enough memory within three neighbouring engines to satisfy its resource requirements. As mentioned in section 4.6.1, we fixed the locations of the kernels in such a way that in each column on the AIE grid, only every second engine contains a kernel, leaving every other engine empty to provide memory for neighbouring kernels. This is clearly a crude approach, which can help explain the discrepancy between the limitations on the window size, data size and in particular the number of kernels which we predict in theory and the ones we

measure when really compiling. It would be an interesting direction for future work to find location constraints which better take the spatial topology into account.

4.6.4 Limitations due to Routing Constraints

As we have seen, increasing the window size or the data size leads to memory constraints, limiting the depth of the Reduce tree. Now we go in the other direction. If both the window size and the data size are set to the minimum of 16 bytes, we would expect that each engine on which a kernel is placed provides enough space to store all of the kernel's data. Therefore, it should be possible to compile the program with a tree containing up to 400 kernels. As our implementation uses a perfect binary tree, we would expect compilation to succeed for a tree of depth 8, i.e. containing 255 kernels. In reality, however, we have only been able to scale the depth to 6 (63 kernels). Trying to compile a configuration with a depth of 7 (127 kernels), $WS = 16B$, and $DS = 16B$, yields the following error:
`ERROR: [aiecompiler 35-3253] AIE Router found too many nets on Column: 8 going EAST, 37/36.`

We do not have a definitive answer why this error occurs, but we have a hypothesis (whose correctness we cannot guarantee). To understand it, it is necessary to recall how the AI Engines are connected in hardware. We have already seen several times that each engine can directly access the local data memory of up to three neighbouring engines (tile memory access). To be more specific, each engine can always access the local data memory of the engine above and below it. Additionally, each engine can access the local data memory of the engine to its right or its left, but not both. Recall the reason for this is that within a tile, if the AI Engine is placed on the left side and the memory module on the right side, only the data memory of the left neighbouring engine can be accessed directly (and vice versa if the directions are reversed). Each row alternates between placing the AI Engine on the left or right side of a tile. Hence, in each column there can be at most four instances of tile memory access to the east direction (we will see why this is important).

Window communication between non-neighbouring AI Engines always happens using the DMA engine [6]. The DMA has a S2MM module to store stream data to memory and an MM2S module to write memory contents to a stream and if we understand the source correctly, these streams also use the AXI4 interconnects [7]. Each engine's AXI4 interconnect can establish up to six connections from south to north, and up to four connections in each of the three remaining directions [5]. In particular, for each engine there can be at most four AXI4 streams in the east direction. Hence, assuming our understanding is correct, there can be at most four window connections with DMA in the east direction per AI Engine tile (this might include window connections between distant tiles which are routed across this tile's AXI4 interconnect).[7]

[7]The author freely admits this paragraph should be taken with a grain of salt as he cannot

To get back to the error message: As there can be at most four instances of tile memory access in the east direction per column and four instances of DMA-based window access in east direction per engine (and there are eight engines per column), there can be at most 36 connections of either kind in the east direction per column. This would match the report we get from the error message. It is trying to convey that the way we have placed the kernels leads to some column which has to send more different streams of data to the east than available. Solving this error for one particular configuration might be feasible by trial and error, but solving it in general would require us to develop an algorithm which keeps track of all AXI4 streams and tile memory access instances of each engine and then places the kernels in such a way that the maximum number of connections for either type of connection is never exceeded. But even computing how many AXI4 streams in a given direction a particular interconnect has or how many tile memory access instances a certain engine has, would require us to know many of the low-level placement and routing choices the AIE compiler internally makes, before it makes them. It is possible to influence these choices by passing user constraints, but at the point of writing it is not clear to what extent. In conclusion, this seems like a very difficult issue to solve, in particular if the placement algorithm should simultaneously take into account the neighbour constraints mentioned in section 4.6.3. Developing such a placement algorithm could be a direction for future work, though the work would likely be considerable if at all feasible.

4.6.5 Kernel Signatures

It would be desirable if the tree structure could be completely hidden from an end user. After all, to them, Reduce is simply an operation to be called in several kernels, such that a specific one - the root - ends up with the result. There should be no reason for them to know how data flows internally.

Unfortunately, with the presented design this is not possible as far as we have found. The fundamental reason is that a kernel's signature needs to contain all input and output windows. We have three different types of nodes in a binary tree: Leaves which have zero inputs and one output, interior nodes which have two inputs and one output, and the root which has two inputs and zero outputs. Thus, a kernel which has no input windows according to its signature, cannot be used as an interior node. A kernel's signature decides where in the tree it can be used. This couples the tree structure we wish to hide from the user to the kernel signature they explicitly have to write. So, even though we can hide the process of *connecting* the kernels from the user (see section 4.4.3), it is up to them to decide the node type of each kernel.

guarantee its correctness.

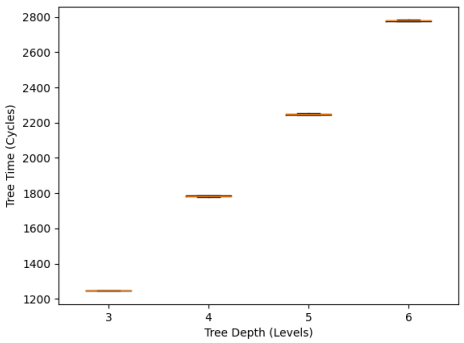

Figure 4.1: Tree time for different tree depths, for $n_{data} = 4, n_{window} = 4$.

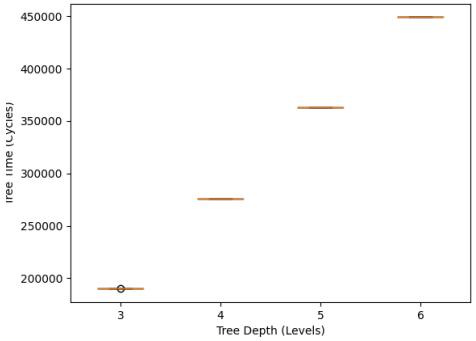

Figure 4.2: Tree time for different tree depths, for $n_{data} = 1024, n_{window} = 1024$.

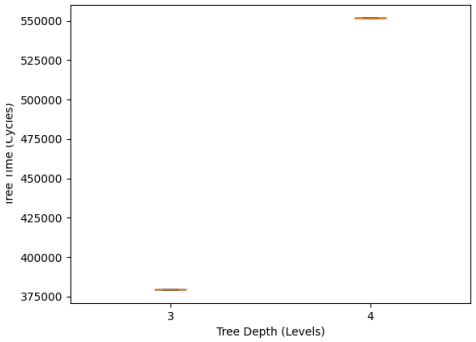

Figure 4.3: Tree time for different tree depths, for $n_{data} = 2048, n_{window} = 2048$.

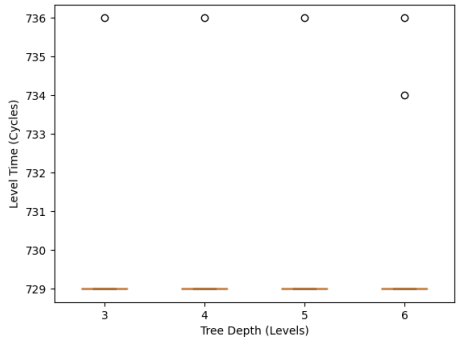

Figure 4.4: Level time for different tree depths, for $n_{data} = 4, n_{window} = 4$.

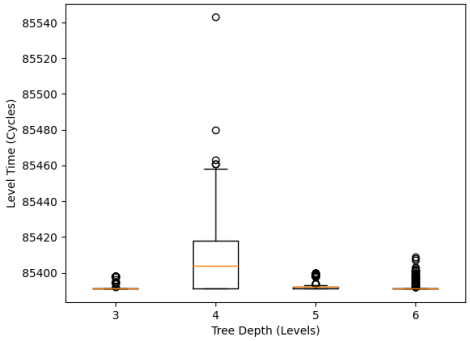

Figure 4.5: Level time for different tree depths, for $n_{data} = 1024, n_{window} = 1024$.

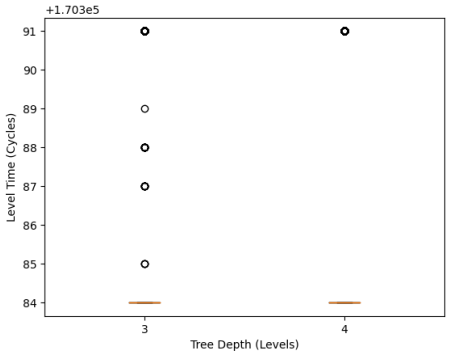

Figure 4.6: Level time for different tree depths, for $n_{data} = 2048, n_{window} = 2048$.

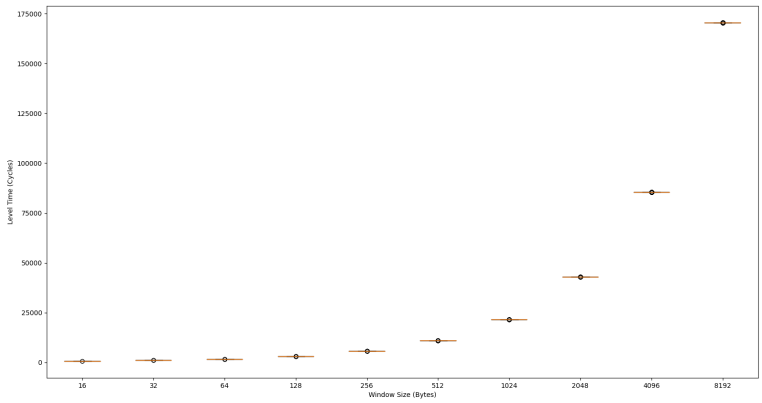

Figure 4.7: Level time for different window sizes, for a constant depth of $L = 3$.

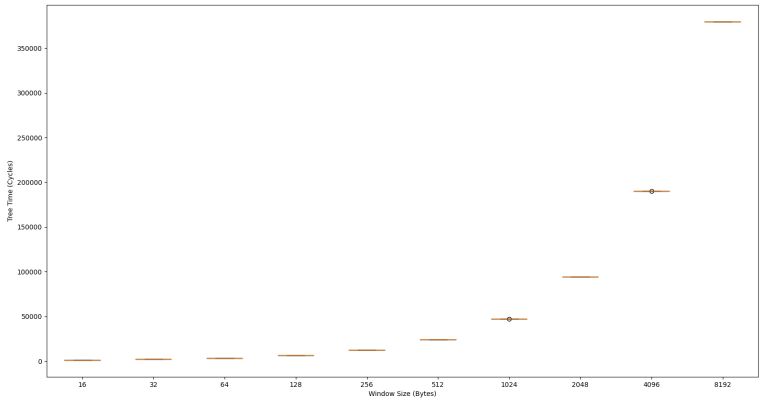

Figure 4.8: Tree time for different window sizes, for a constant depth of $L = 3$.

CHAPTER 5

Gather

The focus of this thesis is the Reduce function, which has been designed, implemented and analysed in detail. Many of the design choices and limitations, such as the use of a tree to forward intermediate results and the considerations regarding memory limitations, can be expected to also apply to other collective operations. Nonetheless, we would like to briefly address some design considerations for a collective called Gather. Note that this collective has not been implemented, so these considerations are purely theoretical.

The Gather collective is described by the MPI standard as a function which *"gathers together values from a group of processes"* [2]. As in the Reduce collective, there is one process dedicated to be the root which is supposed to hold the result of the collective operation once it returns. Each process can provide several values, the count of which can be specified upon calling the function.

Adapting the standard to the AIE architecture implies the following:

- Each kernel provides a *window* of data. As there is a minimum window size (16 bytes), it is not possible for a kernel to provide a single value (unless the window is padded with zeroes, which could not be distinguished from a window where the zeroes were actually significant).
- The window size must be fixed at compile time, so it is neither possible nor necessary to specify the count of values when calling Gather.
- The root cannot be specified when calling the function. Rather, the connections decide which kernel acts as the root. The connection logic must be defined separately and connections are fixed at compile time.

5.1 Design

The most straight-forward approach to designing this collective is to decide one kernel which acts as the root and directly connect all other kernels to the root. The Gather call in the root could then simply wait for data from all senders to arrive and return once they have. Unfortunately, this does not work for the same reason as in the Reduce collective: There can be at most 14 window connections per kernel. Therefore, we adapt the approach from our Reduce design and use a tree. It is theoretically possible to use a tree with any arity up to 13 (since 13 input windows and one output window per kernel is the upper bound of window connections per kernel). However, precaution must be taken to ensure that the per-kernel memory limitation of $128KB$ is never exceeded (see section 4.6). This means that higher arities force us to use smaller window sizes, such that all of the window buffers fit into the kernel's local data memory. Also, one should keep in mind that the more windows our collective function uses per kernel, the fewer windows an end user has left available. In a realistic setting, it might make sense to use the smallest possible arity such that the user is as unconstrained as possible in the number of windows they can use for their own purposes per kernel. Let us assume the design makes use of a tree with arity n. The tree does not need to be perfect, we only require that all nodes have at most n children and all nodes except for the root have exactly one parent.

As in the Reduce design, each user kernel acts as one node of the tree. As a first approach, let us assume that leaves simply send their data window to their parents. The reason why this does not fully work becomes evident once we consider what happens at an interior node, i.e. one with at least one input window. Interior nodes need to somehow combine the data they receive from all input windows as well as the data they provide themselves and send the combination on to their parents. In comparison to the Reduce design, where an interior node performs an element-wise operation on all input windows and its own data to produce an output window of the same size as each input window, we are now in a situation where the interior node's output is (potentially much) larger than the individual input windows. Let us consider an interior node which has m input windows ($m \leq n$) and it provides its own window of data which has a size of WS. We see two possible ways how it could combine all of this data and send it on to its parent:

1. We assume each of its input windows has a size of WS_i for $i \in \{1, ..., m\}$ and fix the size of its output window to be $WS + \sum_{i=1}^{m} WS_i$. Then it can send its own data and all of the data contained in the various input windows as part of a single output window.

2. We keep the window size constant across all connections in the tree. The node forwards all of the input windows it receives on to its parent and sends its own data as a separate window as well.

Both solutions can be implemented, but both have drawbacks. We argue that the drawback of the first solution is so severe that it should not be considered

and recommend to implement the second solution instead. The reason for this is that solution 1 requires the window size to grow exponentially with the depth of the tree, resulting in huge window connections close to the root even for trees of small depth. Since all of a kernel's windows have to be stored in its local data memory, these large window sizes could force nodes which are close to the root to exceed their per-kernel memory limits (see section 4.6 for a detailed discussion of memory issues).

We now specify how solution 2 works in detail. The difficulty of this approach becomes evident as we turn to interior nodes deeper in the tree. Assume for example a perfect binary tree with a depth of 3 and a constant window size WS. The leaves send exactly one window, containing their local data. Each interior node behaves as explained in the bullet point, i.e. it sends three different output windows each of size WS, containing the data sent to it by its first child, its second child and its own data, respectively. But now consider the root — it sequentially receives three windows from each of its two children. How is it supposed to know how many windows it receives from each child without knowing the graph topology?[1] In particular, it would need to know how many tree levels are above it in order to compute the number of windows it receives from each child, and this is only valid assuming a perfect tree. If it is not perfect, knowledge of the whole graph topology would be required. This seems hardly feasible.

Our alternative approach is inspired by networking, where it is common to send header packets which provide information about the content which is about to arrive. In our situation, such a header packet is a window which we require each node to send before sending any actual data (we call this the *header window*). The first element of the header window is the number of data windows this node is about to send. For leaves, the number of data windows is clearly 1. Each interior node can compute the number of data windows by waiting for the header window of all its children, adding the first element of each one and finally adding 1 (because it sends an additional window with its own data). After that, it can send the header window containing this result and follow it up directly with a first data window containing this node's data. Finally, the node can iterate over all input windows and wait for the specified number of data windows to arrive from each connection. As soon as an input window is available, the node can forward it to its parent. For clarification, we provide a pseudocode for this procedure in listing 5.1.

A disadvantage of this design is that the number of windows which have to be sent between two levels in the tree grows exponentially with the distance to the leaves. We know that acquiring and releasing asynchronous windows takes a lot of cycles, so this method is potentially very slow for deep trees. It seems like this is a classic space-time tradeoff — either the window size (and thus the memory requirements) or the number of messages (and thus the runtime)

[1] This problem becomes even more significant if we want to be able to call the Gather function several times per kernel. Each kernel needs to know how many incoming windows from a given child belong to the first Gather call in order to determine which window is the first belonging to the second call.

increase exponentially.

If one were to implement this function for the AIE architecture, it would be absolutely necessary to use asynchronous windows because a single kernel invocation needs to read and write multiple windows per window connection. One implementation choice we leave open is the order in which data windows are forwarded. The pseudocode in listing 5.1 forwards all windows arriving from the first child before moving on to the second child. It would also be possible to forward the first window arriving from each child, then forwarding the second window from each child etc. The impact this ordering has on the performance could be interesting to investigate.

```
function Gather(input_windows[m], output_window):

    n_data_windows[m] //empty array of integers

    for i = 0; i < m; i++:
        w = input_windows[i]
        acquire(w)
        d <- read first element of w
        release(w)
        n_data_windows[i] = d

    header_value = 1 + sum of all entries of n_data_windows
    send(output_window, {header_value, ...})
    send(local_data)

    for i = 0; i < m; i++:
        repeat n_data_windows[i] times:
            acquire(w)
            send(output_window, w)
            release(w)
```

Listing 5.1: Pseudocode of the Gather collective operation.

CHAPTER 6

Discussion

6.1 Lessons Learned

This section mentions some of the bugs we have encountered while working on the thesis.

6.1.1 GMIO Transactions

There has been great confusion concerning one line in the host code which is relevant for both the latency benchmark and the Reduce implementation. In the host code, to access the data a kernel produces we have to explicitly send it from the kernel to GMIO. From the kernel's perspective, this appears to be a normal write to an output window. In the host, after allocating enough memory to store the kernel's data once it arrives, we have to call a specific function `GMIO::aie2gm_nb` to *"initiate memory-mapped AXI4 transactions for the AI Engine to write to DDR memory spaces"* [10]. After running the graph, we call `GMIO::wait`, which supposedly waits for the AXI4 transaction to finish. The documentation claims that after this function call has returned, we should be able to access the data in the host: *"gr.gmioOut.wait() is to ensure that data has been migrated to DDR memory. After it is done, the PS can access output data for post-processing."* [10]. However, we have found that this does not work: For some executions of a single compiled program, the data is available as expected; for other executions, it is not available and we read arbitrary initialization values; and for some executions, we even get data which has partially arrived, i.e. the first few entries match what we expect to receive from the kernel and the remaining entries are still the initialization values. The workaround we have found is to call the `wait` method on our graph instance, in addition to the

`GMIO::wait` call (see listing 6.1). We do not have a justification for why this should fix the bug, nor why the bug appears in the first place.

```
simpleGraph mygraph;
int main(int argc, char *argv[]) {
    mygraph.init();
    int32* block = (int32*)GMIO::malloc(block_size);
    mygraph.gmioOut.aie2gm_nb(block, block_size);
    mygraph.run(1);
    mygraph.wait(); //<-- This should not be necessary!
    mygraph.gmioOut.wait(); //<-- This should be sufficient.
    //... can access data in 'block'.
}
```

Listing 6.1: Slightly simplified host code showcasing that we need the line `mygraph.wait()` to successfully access the transferred data.

6.1.2 Using std::vector

When defining the graph, it is necessary to store all kernels as private members of the graph class. One solution would be to use a `std::vector` of kernels as shown in listing 6.2. However, this has led to inexplicable compilation errors when used in the context of the full Reduce code. The solution was to use fixed-size C-style arrays instead. We do not know why using `std::vector` does not work as there are examples provided by AMD/Xilinx which make use of them, but we have not investigated this error further since we have found a workaround. We would generally recommend to use C-style arrays instead of `std::vector` in order to avoid this error.

```
class simpleGraph : public graph {
    private:
        //std::vector<kernel> kernels;
        kernel kernels[n];
    public:
        simpleGraph() {
            for (int i = 0; i < n; i++) {
                //kernels.push_back(kernel::create(k));
                kernels[i] = kernel::create(k);
            }
            // ...
        }
};
```

Listing 6.2: Comparison of the usage of `std::vector` (commented out) and C-style arrays.

6.1.3 Conclusion

Working on this device has been interesting, but often frustrating. The guides provided by AMD/Xilinx are relatively well written and answer most questions, but information is scattered rather unpredictably across several guides. The support forum is not very active and is the only online resource apart from the official guides. We have encountered inexplicable bugs which seem to contradict statements in the guides, and this has cost us valuable time.

The AIE architecture is promising, but only if it is used in a very particular way which makes use of all its intricate hardware details.

6.2 Further Work

This thesis has hopefully shown that it is possible to implement collective operations for the AIE architecture, using the example of the Reduce function. There are many opportunities for further work in this direction.

Comparison of Arities

The Reduce design we have implemented makes use of a binary tree to forward partially reduced intermediate results. It would be interesting to see how trees of different arities influence the performance and the scaling behaviour of the implementation. The same could be done for the Gather collective once it is implemented.

Placement and Routing Problems

We have extensively discussed the limitations due to limited per-kernel memory and a limited number of inter-AIE connections. As we have seen, the practical upper bound we could achieve on the depth of the tree is significantly below the theoretical upper bound. This is not surprising because we did not model spatial constraints:

- Each kernel can access the local data memory of at most three neighbouring engines.
- Only a limited number of AXI4 streams per direction are available at each engine.

It would be interesting to see if one can model the upper bound on the window size, data size and tree depth more accurately, taking into account these two constraints. Optimally, one could even implement a proper placement algorithm. However, this seems like a hard problem given that the AIE compiler internally attempts to solve essentially the same problem (using an ILP solver), but it either times out due to the size of the problem space or it does not find a solution because it is over-constrained.

Remaining Collectives

It should be relatively simple to implement the Gather collective based on the implementation of Reduce and the design considerations done in chapter 5. One could try to improve on the design such that the number of windows does not grow exponentially with the tree depth.

The MPI standard defines more collective operations, such as Scatter and Allreduce, which could also be designed and implemented. For Scatter, it might

be possible to use a similar approach as Gather, just inverted. Allreduce could be implemented by performing Reduce and then broadcasting the result to all senders using the built-in broadcast capabilities. Alternatively, one could experiment with more advanced designs such as butterfly networks.

Bibliography

[1] Jie Lei, Jose Flich, and Enrique S. Quintana-Orti. "Toward Matrix Multiplication for Deep Learning Inference on the Xilinx Versal". In: 2023. DOI: 10.1109/PDP59025.2023.00043.

[2] MPI. *Gather*. URL: https://www.mpich.org/static/docs/v3.3/www3/MPI_Gather.html (visited on 08/20/2023).

[3] Noah Perryman, Christopher Wilson, and Alan George. "Evaluation of Xilinx Versal Architecture for Next-Gen Edge Computing in Space". In: vol. 2023-March. 2023. DOI: 10.1109/AERO55745.2023.10115906.

[4] Max Wierse. *Evaluation of Xilinx Versal Device*. 2023.

[5] Xilinx. *AM009 AXI4 Stream Interconnects*. URL: https://docs.xilinx.com/r/en-US/am009-versal-ai-engine/AXI4-Stream-Interconnect (visited on 08/20/2023).

[6] Xilinx. *AM009 Communication via DMA*. URL: https://docs.xilinx.com/r/en-US/am009-versal-ai-engine/AI-Engine-Tile-to-AI-Engine-Tile-Data-Communication-via-Memory-and-DMA (visited on 08/20/2023).

[7] Xilinx. *AM009 Memory Module*. URL: https://docs.xilinx.com/r/en-US/am009-versal-ai-engine/AI-Engine-Memory-Module (visited on 08/20/2023).

[8] Xilinx. *UG1076 AIE Compiler Options*. URL: https://docs.xilinx.com/r/2022.1-English/ug1076-ai-engine-environment/AI-Engine-Compiler-Options (visited on 08/20/2023).

[9] Xilinx. *UG1076 AIE Simulation*. URL: https://docs.xilinx.com/r/2022.1-English/ug1076-ai-engine-environment/Simulating-an-AI-Engine-Graph-Application (visited on 08/20/2023).

[10] Xilinx. *UG1076 GM2AIE*. URL: https://docs.xilinx.com/r/2022.1-English/ug1076-ai-engine-environment/Programming-Model-for-AI-Engine-DDR-Memory-Connection (visited on 08/20/2023).

[11] Sven Zanetti. *Gitlab Repository*. URL: https://spclgitlab.ethz.ch/szanetti/versal_window (visited on 08/20/2023).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Designing a Communication Library for Xilinx Versal Devices Using the Window-Based API

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Zanetti	Sven

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 20 August 2023	S. Zanetti

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.