Evaluating the Efficiency of Neural Network Implementations on AMD Versal AI Engines

Yilin Shen

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2024

Committee:

Scott Hauck

Shih-Chieh Hsu

Program Authorized to Offer Degree:

Electrical and Computer Engineering

University of Washington

Abstract

Evaluating the Efficiency of Neural Network Implementations on AMD Versal AI Engines

Yilin Shen

Chair of the Supervisory Committee:
Scott Hauck
Department of Electrical and Computer Engineering

The AI Engine (AIE) is an optional component of the AMD Versal Adaptive Compute Acceleration Platform (ACAP). It is an innovative device that offers extensive parallelism to enhance compute density and reduce power consumption. However, the performance of the AIE, particularly for small models requiring low latency, remains uncertain.

In this thesis, we mapped three neural network benchmarks to the AIE section of the Versal VCK190. We explored the best coding practices and characteristics of the AIE. Additionally, we mapped these models to the FPGA fabric portion of the VCK190 and compared the cost and performance with our AIE implementation. Based on six metrics, we found that the AIE's efficiency is slightly better than the FPGA fabric in terms of power and silicon area utilization, but worse than the FPGA in terms of performance, resource utilization and price. This discrepancy is due to limitations in interconnection and the inefficiency of hardware units when the vector data path cannot adapt to certain shapes of the input data.

Table of contents

ACKNOWLEDGEMENTS

This research was funded by National Science Foundation (NSF) grant No. 2117997.

I would like to express my deepest gratitude to my advisor, Professor Scott Hauck, for his support, guidance and encouragement throughout my research and study. His expertise and insight were invaluable to me and my work.

Special thanks goes to Professor Shih-Chieh Hsu, for building the A3D3 community and introducing me to it. I learned a lot from this interdisciplinary community and had wonderful opportunities to present my work and get valuable insights from others.

I would also like to thank the members of ACME lab. Thank you to Caroline Johnson for helping me understand VNN and sharing data with me; Thank you to Xiaohan Liu, Atharva Mattam and Pranav Murali for stimulating discussions; Thank you to Geoff Jones for the initial guidance and understanding of the AI Engine.

I am deeply grateful to my family for all of their support and understanding throughout this journey. I would also like to express my heartfelt thanks to my partner, Manze Zhang, for all her love and encouragement.

1 Introduction

In an era where artificial intelligence (AI) continues to be widely deployed, the increasing need of AI performance encounters growing AI complexity. To meet this trend, people came up with various hardware acceleration methods including using existing hardware (such as FPGA and GPU) and building ASIC accelerators. Among these options, the use of FPGAs stands out for its programmability, low latency and moderate cost.

Meanwhile, from the FPGA vendor side, people have started to adopt AI domain-specific architectures and embed them into the FPGA, bridging the gap between the performance needs and the underlying hardware. Recently, AMD released its Versal adaptive SoC product family which integrates CPU, FPGA and high-speed interfaces. In this family, the AI series includes an additional, dedicated hardware region called the AI Engine (AIE)[Gaide19] for AI inference workloads. The AIE is an array of VLIW vector processors with versatile interconnections among each other, forming a dataflow architecture that suits ML workloads. Inside each AIE core, there is a vector unit that supports multiple data formats (8, 16, 32-bit fixed point and floating point), and has a dedicated data path for vector MAC operation. This brand-new architecture potentially offers advantages in performance per watt and higher throughput[Xilinx22], while keeping the programmability. This piqued our interest in evaluating this emerging hardware, and exploring the possibility of migrating low-latency and high-throughput FPGA ML applications to it, if the AIE proves efficient.

In this work, we use the same benchmark model from previous work of our group[Johnson23]. We map models to both the AIE and FPGA fabric of the Versal AI series, measure performance and cost metrics, and analyze the outcome. The structure of this thesis is as follows: Section 2 provides

background information on typical neural network computations. Section 3 introduces the AI Engine, and Section 4 provides AIE characteristics for design considerations. Section 5 introduces metrics that are important to the evaluation. Section 6 demonstrates the benchmark implementations on the AIE, along with the optimization process. Section 7 describes the data acquisition process for the benchmark and presents the results. Section 8 discusses the underlying reasons for the results, possible limitations, and potential enhancement.

1.1 Related Work

Since the Versal AIE has been released, researchers have begun mapping specific computational operations to the Versal chip. CHARM[Zhuang23] mapped the matrix multiply (Matmul) operation to the Versal SoC, while MaxEVA[Taka23] proposed a higher performance Matmul without relying on the FPGA side. Lei et al. mapped parallel Matmul to the Versal exploiting multi-level memory hierarchy[Lei24]. Vyasa explored a high-performance way of convolution on the AIE with 32-bit and 16-bit precision[Chatarasi20]. Chen et al. introduced a heterogeneous way to map graph neural network (GNN) to Versal[Chen23];

Meanwhile, evaluation of Versal AIE is being conducted across multiple fields. Brown et al.[Brown23] evaluated the use of Versal for atmospheric simulation, and SPARTA[Singh23] assessed AIE for weather stencil computation. TaPaSCo-AIE[Heinz24] is a framework leveraging AIE with heterogeneous stream acceleration, and evaluated the AIE performance with a simple feed-forward neural network.

In addition, neural network accelerators have been built upon the AIE. XVDPU[Jia24] is a convolutional neural network accelerator that utilizes both AIE and programmable logic.

However, for our specific need to evaluate standalone AIE versus pure FPGA implementation of machine learning models, there is no direct reference. In this work, we will focus on pure-AIE mapping of ML models and emphasize comparisons to assist FPGA users in selecting next-generation devices for ML applications.

2 Neural Network Computation

2.1 Dense Layer

A dense layer, also known as a fully-connected layer, computes its output by performing matrix multiplication between the input data and the weights, followed by the addition of bias to each output. For example, if we have a dense layer with 10 inputs and 8 outputs, the weight matrix for this layer would be 8×10, and the bias would be a vector of length 8.

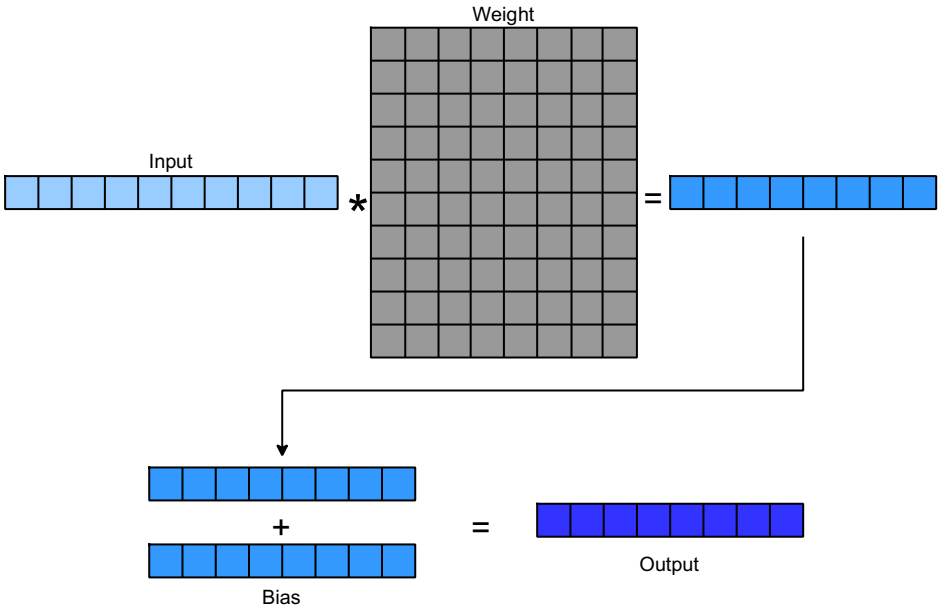


Figure 1. Dense computation example

2.2 Convolution Layer

In general, the input to a convolution layer is a 2D image. Its computation is performed by a filter or kernel that conducts element-wise multiplication, followed by accumulation and addition with a bias. Each time the filter performs this computation, it produces one pixel of the layer's output. The filter slides over the input data horizontally and switches to the next row until it reaches the end of the row. If the window slides one pixel at a time horizontally, the stride of this convolution is one. The window could also slide more than one pixel at a time, the output shape of the conv layer would then be smaller.

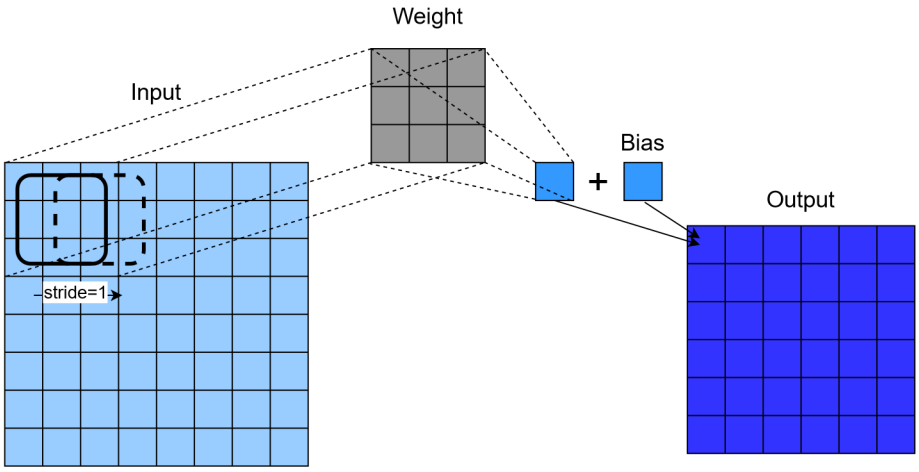


Figure 2. Convolution layer of 8*8 input and 3*3 kernel

2.3 ReLU Layer

ReLU stands for rectified linear unit, and is commonly used for the activation function in neural networks. For a given input, its output complies with the formula:

$$f(x) = \begin{matrix} x & if\ x > 0, \\ 0 & otherwise, \end{matrix}$$

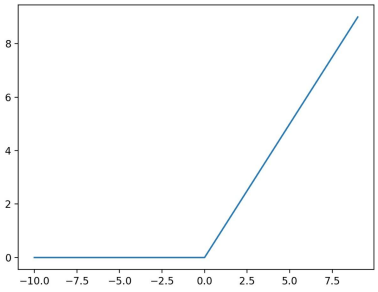


Figure 3. ReLU function line plot

2.4 Sigmoid Layer

The sigmoid layer serves as an activation layer in a neural network, converting any range of input values to a range between 0 and 1. For very negative values, the output will be close to 0, while for large positive input values, the result will be close to 1. The sigmoid function is defined as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

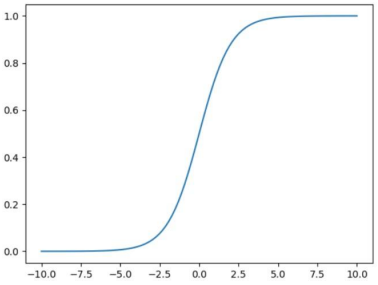


Figure 4. Sigmoid function line plot

2.5 Softmax Layer

The softmax layer is also an activation layer, with outputs guaranteed to be in the range of 0 to 1. Additionally, the softmax function takes a vector of inputs and ensures that the sum of the corresponding output vector is 1. It is commonly used as the final layer in a classification neural network, as the result can be interpreted as a probability distribution.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \; for \; i = 1, \ldots, k$$

2.6 Possible Advantage of the AIE

The dense and convolution layers are computation-intensive, and both of them require the MAC operation. Each AIE core has its vector MAC data path, and the dataflow between each AIE core can be specified. This provides a possibility for efficient mapping of the layer's computation to one or multiple AIEs' hardware units. The vector MAC unit provided better performance and energy efficiency than FPGA solutions in terms of computation, as the FPGA uses scattered resources to support the operation, and the programmable routing between the scattered resources added more delay and introduced more power consumption.

For the Sigmoid and Softmax activation layers, the complex non-linear functions will be implemented in a look-up table fashion for both AIE and FPGA, and the look-up shouldn't take much time compared to the dense/conv layer. The overall performance of a full model should be boosted by the AIE MAC unit if there is no other overhead introduced while mapping the model.

3 Versal Adaptive SoC and AI Engine Introduction

The Versal Adaptive SoC is a next-generation device released by AMD/Xilinx. It provides a heterogeneous combination of CPU, vector processor array (AI Engine) and FPGA. This new category of device allows users to customize the application deployment according to the workload characteristics. For instance, the FPGA can implement arbitrary functionality but with a cost: The FPGA fabric contains fine-grained logic blocks (look-up tables, flip-flops, DSPs) and an interconnection network that connects them. Those logic blocks are often tied to a single operation repetitively, resulting in a large resource usage if the workload is compute-intensive. In contrast,

the AIE, as an array of parallel processors, provides a large density of mathematical operations, and its resources could execute and switch to different operations on a cycle-by-cycle basis. However, the AIE resources are fixed to specific organizations, and computations that do not fit this model can be implemented very inefficiently. The CPU is a better fit for light-weight and general-purpose workloads, as it has reasonable computational resources and supports a variety of operations. There is also a programmable network on chip (NoC) available on Versal, to integrate those three types of resources and creates better connectivity between them.

The AIE is one of the distinct differences between Versal and the previous generation of devices. It aims to accelerate machine learning inference workloads. This could be a possible upgrade to FPGA acceleration of ML inference, as the AIE provides dedicated hardware resources for heavy computation operations, selected operations of ML inference could be faster, and the corresponding power consumption could be smaller. The resource utilization rate of those operations could also be better, as the AIE resource is not tied to specific operation and could be changed between each cycle in contrast to FPGA fabric. However, one of the biggest concerns is, for those operations that are not supported inherently by AIE, will the degradation harm the overall efficiency of ML inference on AIE. This will be evaluated and answered in section 6 and 7.

3.1 AI Engine Core Architecture

Each AI Engine core is a parallel processor, and the parallelism is implemented with a very long instruction word (VLIW) and single instruction multiple data (SIMD). This provides data-level and instruction-level parallelism (DLP and ILP), allowing for better throughput and efficiency.

The VLIW feature of AIE allows 7 operations (2 scalar, 2 load, 1 store, 1 vector and 1 stream operation) per clock cycle. It is essential to balance the workload by evenly distributing different

types of operations to better use the hardware resource. Avoiding data dependencies or structure dependencies is also helpful to maintain efficiency, since the AIE is pipelined and the dependency causes stall or insertion of no operation (NOP). If any operation within the 7 issue slots stalls, the remaining 6 slots might also get stalled.

The SIMD feature of AIE supports arbitrary data type and bit widths. For fixed point data, 32 bits, 16 bits and 8 bits are supported. For floating point data, only 32 bits are supported. No matter which bit width or datatype is used, the supported total data length (vector length) remains the same. The underlying hardware are 16 128 bits-only registers. They could also be regarded as 256, 512 and 1024 bits registers logically, with an exception of the 1024 bits register: They are overlapped because of hardware limitation. If we have two 1024b vectors on one AIE, register spilling could happen and the performance is reduced[Xilinx22].

vector length	128	256	512	1024
bit width	int8/16/32/float32	int8/16/32/float32	int8/16/32/float32	int8/16/32/float32
count	16/8/4/4	32/16/8/8	64/32/16/16	128/64/32/32

Table 1. AIE supported data type

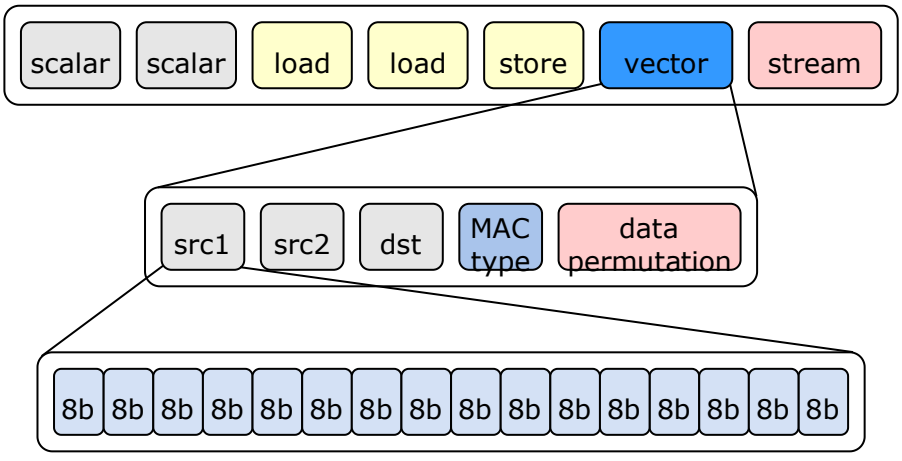


Figure 5. AI Engine VLIW and SIMD

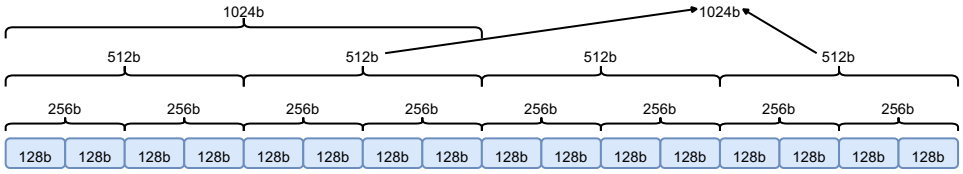


Figure 6. AI Engine register file

3.2 AI Engine Array Architecture

Using a single AIE core, an array of AIE can be built and connected, providing thread-level parallelism (TLP). A typical Versal device with AIEs, for example, the VCK190 board, has an array of 400 AIEs. The TLP is realized by multiple AIEs doing different computations in parallel. However, in contrast to FPGA parallelism, which is using low-cost interconnection to conduct the dataflow among replicated hardware resources, there is an overhead for AIE due to inter-AIE core communications and synchronization. There are three ways available in AIE: Window, stream and cascade data transfer, and the communication way chosen affects the efficiency of TLP. The window transfer uses AIE local data memory as a shared buffer, and the buffer access is controlled by mutex lock. In this way we have a high bandwidth writing/reading the buffer but added an overhead of synchronization. The stream transfer has a much smaller bandwidth but is lock-free. If we have a relatively small amount of data needed to be transferred without synchronization and stall, stream is a good option. There are two pairs (input and output as a pair) of stream ports available on the AIE, making a data reduction structure (see section 4.5) possible. The cascade transfer is also stream-based, high bandwidth, but each AIE has only one pair of cascade ports. Usage of cascade transfer is mainly limited to passing partial results. More details about data transfer will be discussed in section 4.1, section 4.2 and section 4.3.

Each AIE, along with its local data memory and interconnection, forms an AIE tile. This tile-based architecture creates a graph programming model. For each AIE tile we have a kernel function for

it to perform the computation along with the data I/O, and from the top level we specify the connectivity among these tiles.

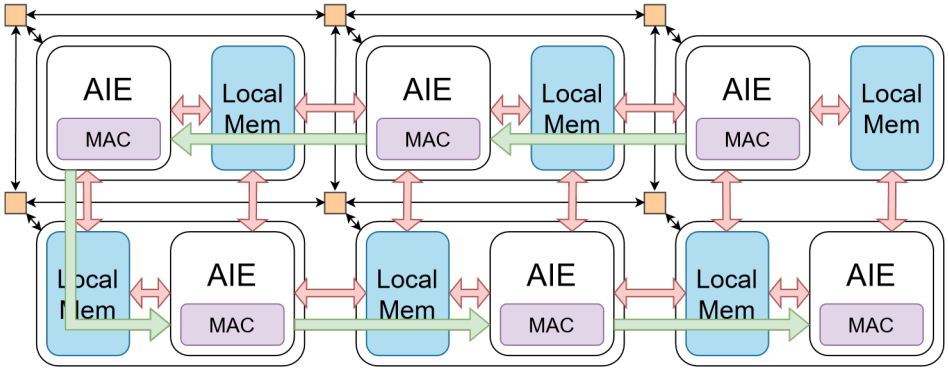


Figure 7. AI Engine array

3.3 AI Engine Interface

The AI Engine communicates with the rest of the parts of Versal by its interface tiles. These tiles can manage two types of interfaces (to the FPGA and to the NoC), and the bandwidth performance is shown below:

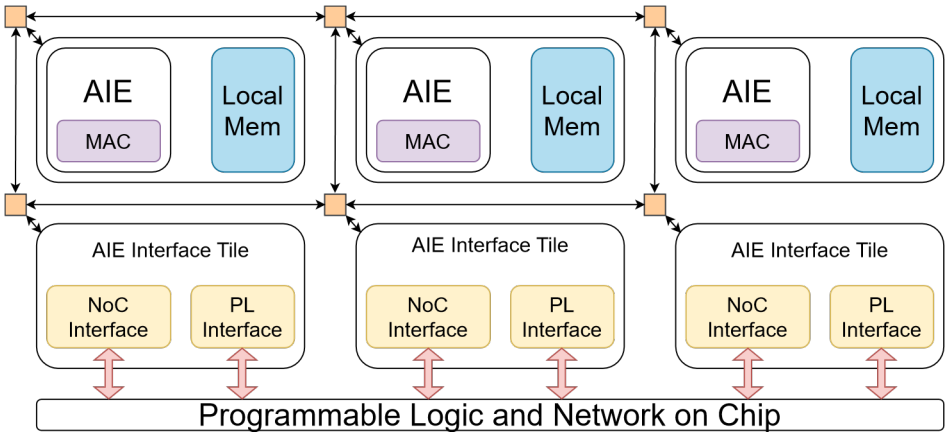


Figure 8. AI Engine interface

	FPGA	NoC
AIE Read (bits/cycle)	8*64	128
AIE Write (bits/cycle)	6*64	128

Table 2. AIE interface bandwidth performance

However, this is not the bandwidth for each of our AIE cores. Each AIE core can only accept 2 words of 32-bit data each cycle from its data switch, and the remaining bandwidth is forwarded to other AIEs by the interconnection network of the AIE array. The mismatch of bandwidth is because we have only one row of AIE Interface tiles but there are multiple rows of AIE cores, one AIE Interface's bandwidth has to accommodate the needs of multiple AIE cores.

4 AI Engine Characteristics

Before we start to discuss the actual neural network mapping on AI Engine, let's first take a look at the AI Engine characteristics, which shows helpful features that AI Engine offers. First we discuss details of data access methods available in the AIE array, then we discuss two fundamental ways of managing the dataflow, and finally, selected characteristics of the AIE core are introduced.

4.1 Window Data Access

Window data access is a high-throughput way (256-bit per cycle) of moving data but with an overhead of lock acquisition/release. For two adjacent AIEs, the window data transfer happens in a shared local memory that is between those two AIEs. When one AIE is interacting with the memory, it acquires the lock and prevents the other AIE from working with the memory until it releases the lock. To prevent performance degradation, the AMD AIE tool implements a double-buffer (ping-pong buffer) in the memory by default, allowing each AIE to work on different memories concurrently.

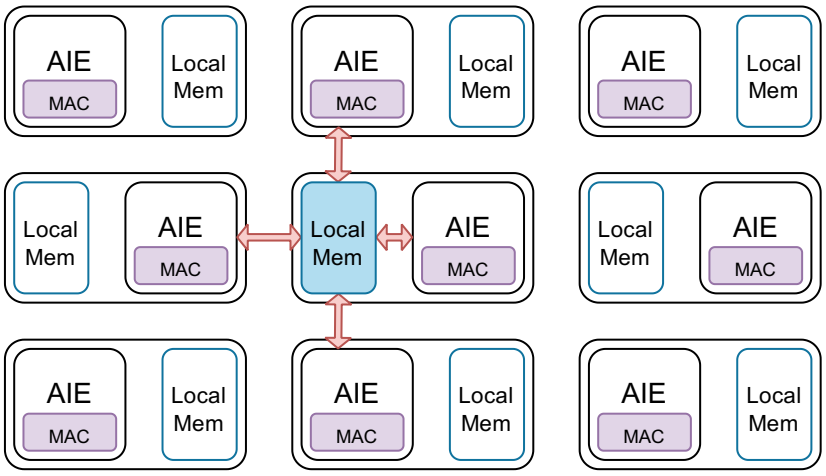


Figure 9. Window data access: local memory to adjacent AIE

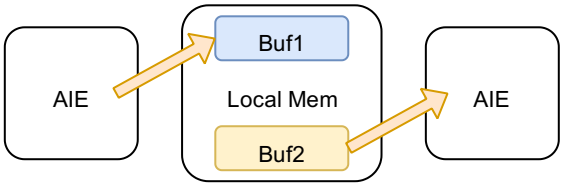


Figure 10. Window data access: double buffer

However, the overhead of lock acquisition/release still hasn't been solved and is caused by the repeated AIE program loading(see section 4.6). As long as we use the window data access method, the lock mechanism is always introduced and will need the compiler-inserted code to conduct the lock. This piece of code is placed outside of the user-defined function for AIE, thus requiring the user function to finish first, then execute the lock code. This behavior is repeated during AIE's execution, and each time the user-defined function needs time to be loaded. The way the AIE compiler inserts the lock conduct code also limits our user function: it cannot be an endless loop, it has to finish and return, to allow the code to be executed afterwards.

4.2 Stream Data Access

In contrast to window data access, stream data access involves reading and writing data from the switch between AIE tiles. It is lock-free and does not require buffer coordination, allowing it to

operate in free-running mode (endless loop). The drawback is that the bandwidth is limited to only 32 bits/cycle per port and each AIE tile has 2 read and 2 write ports. In total we have a 64 bits/cycle bandwidth of either write or read, in contrast to 256 bits/cycle for window data access. Nevertheless, we will rely heavily on this type of access for our models later, because in this way we can avoid the program load overhead and it improves the performance of our models.

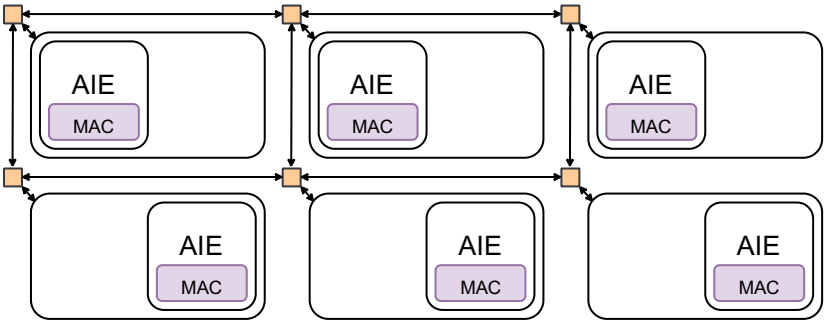


Figure 11. AIE stream data access (top-level)

4.3 Cascade Data Access

The cascade port is another stream-based data access method available, and it is also lock-free, with a width of 384 bits. However, there are some limitations that prevent us from building high-throughput designs using it. First, there is only one cascade read and one cascade write port per AIE, which means we cannot build a reduction tree from it. Second, the cascade port needs to access a special register file called accumulator register instead of the register file used for window and stream access (vector register). To transfer vector register data via the cascade port, we have to perform a data conversion that takes 6 cycles in the data path. Thus, the cascade data access is more beneficial if the computation across multiple AIEs continues to involve the accumulator registers.

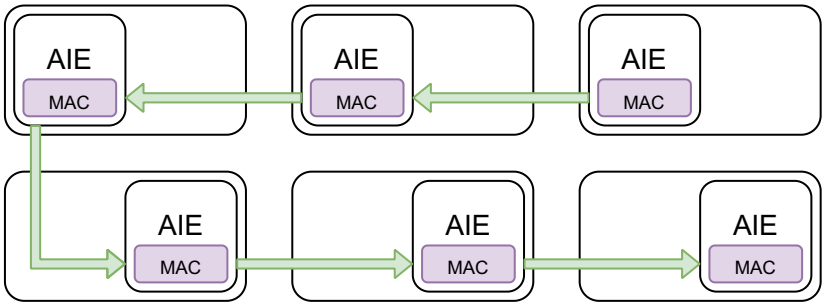


Figure 12. AIE cascade data access

4.4 Broadcast

The output of each AIE tile can be broadcast to multiple receivers. This is particularly useful when there are multiple kernels in a convolutional layer, as it allows multicasting the input to any AIE performing the corresponding kernel's computation, instead of multiple one-to-one data feeders.

However, the broadcast happens only if all receivers are ready. If one receiver is not ready, other receivers are stalled to wait until the broadcast condition is met. This creates a pitfall if we try to implement the broadcast while the receivers have data dependencies. In the example shown below, all AIEs share the same input (broadcast to stream port), and take previous AIE's output as the other input (with cascade port).

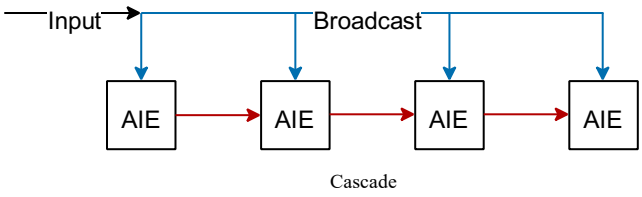


Figure 13. AIE deadlock due to broadcast

Here, each AIE is waiting until both cascade port and stream port's data are ready except the left-most one. Take a closer look at the second AIE from the left, it waits for the cascade port, and the data comes after the left-most AIE takes the broadcast input first, does the computation and puts the output on its cascade port. However, the broadcast cannot execute, because it is still waiting

for the second AIE from the left, which is waiting for the cascade input. This forms a circular dependency and thus a deadlock.

We can still make this AIE arrangement run, by manually specifying FIFO between the broadcast and the AIE. The FIFO is supported by the AIE hardware and is either implemented in the stream interconnection or in the data memory, according to the user's specification. In this way the broadcast does not have to wait for the AIE, since the FIFO is always ready and will take care of the input.

4.5 Reduction

In contrast to broadcast, there is no direct way to construct a multi-sender to single receiver connection. However, it is a typical operation in parallel processing and neural network inferencing: For a computation-heavy workload, it could be broken down to multiple lightweight parts that are done by different AIEs, but then we still need to merge these partial results back together. Sometimes the partial results need to be concatenated to a unified vector, at other times the partial results need to be summed together to form a single value. We will refer to those kinds of operations as reduction. To perform the reduction, there are two approaches:

- packet switch
- reduction tree

Packet switch is primarily used for sharing a single physical channel with multiple data streams. Each packet is bundled with an ID, and the switch has to inspect and redirect the packet using its ID. This adds a layer of complexity to our reduction scenario: the receiver cannot determine which packet should come first without knowing its ID. Checking packet ID and reordering them adds significant overhead, making this method less ideal for our purposes.

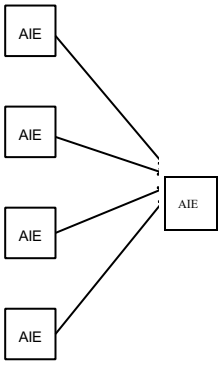


Figure 14. Packet switch

In order to still perform the reduction, we can use some of AIEs as 2-to-1 mergers to assemble a reduction tree. Each AIE has two read stream ports available, allowing it to concatenate and stream out data, functioning as a reducer. The drawback from this approach is the increased latency and resource usage, as the data will flow through multiple layers of AIEs (log2 of n-input) compared to a direct merge.

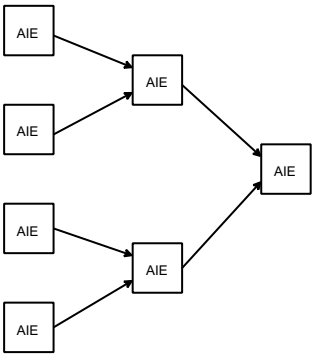


Figure 15. Reduction tree

4.6 Program Loading and Free-running

The AI Engine program loading includes three stages: initialization, the main function, and the computation kernel itself. The AIE user could only implement the computation kernel, but not the init or main function. There is a significant overhead during initialization, but it occurs only once. The overhead of the main function is recurring: each time the computation kernel finishes, the main function has to start, do some buffer synchronization for window data access and then reload

the kernel. This is particularly detrimental if the kernel itself is fast. Free-running mitigates this problem by adding an endless loop around the computation kernel, allowing the kernel to run forever and without overhead (pseudo code shown below). However, as we discussed in section 4.1, window memory access is coordinated by the main function, making it unusable if the kernel is free-running. We still can use stream and cascade data transfer, which is sufficient for most cases.

```
init();
main() {
  lock_and_buffer_init();
    while(true) {
      lock_acquire(); //overhead
      userKernel() {
         //user code
         ...
         ...
      }
      lock_release(); //overhead
      manage_buffer();//overhead
  }
}
```

Default program load sequence

```
init();
main() {
  lock_and_buffer_init();
    while(true) {
      lock_acquire();
        userKernel() {
          while(true) {
             ... //user code
          }
        } //never reach below
        lock_release();
        manage_buffer();
    }
}
```

Free-running sequence

4.7 Vector MAC

Inside each AIE, there is a dedicated vector data path for Multiply-and-Accumulation (MAC) operations. It can perform 128 8-bit MAC operations per cycle, or 32 16-bit MAC operations per cycle, or other precisions. This is the heart of the AI Engine, and understanding the architecture of the data path and its capabilities is crucial for model deployment. The data path includes:

- Two input buffers with 1024 bits and 256 bits
- A shuffle network to retrieve arbitrary data from the respective buffer and feed it to the MAC computation unit
- An output buffer, to store the result and perform partial result accumulation

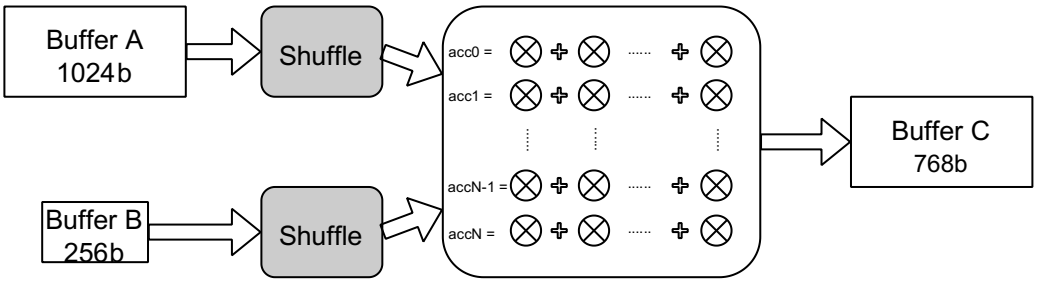


Figure 16. AIE MAC data path

The way we feed those input buffers, and the way we map computations to this vector data path really affects the performance and efficiency. There are two stages of mapping: First determine the chunk of data that we want to operate on (for example a small portion of a large matrix multiplication, or part of the input image for convolution), store them in the buffer, then determine the shuffle network configuration to feed the MAC unit with a specific data layout pattern.

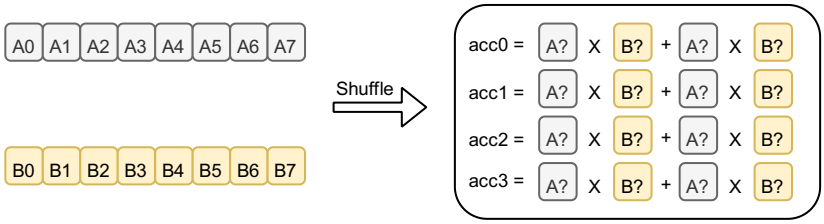


Figure 17. Shuffle network data placement to MAC unit (scaled)

However, the shuffle network has limitations that prevent it from supporting an arbitrary data layout. It fetches data and places them to the MAC unit in a pattern, specified by configuration bits including start, step and offset. A shuffle configuration for B buffer with start=3, step=2 and offset=0,1,1,2 shown in figure below. The top-left element is the first to be settled, with the start parameter. From there, the remaining elements could be filled with the help of offset and step. With lower bit precision there is an additional square parameter that helps further specifying the placement pattern, but there are still patterns which cannot be achieved.

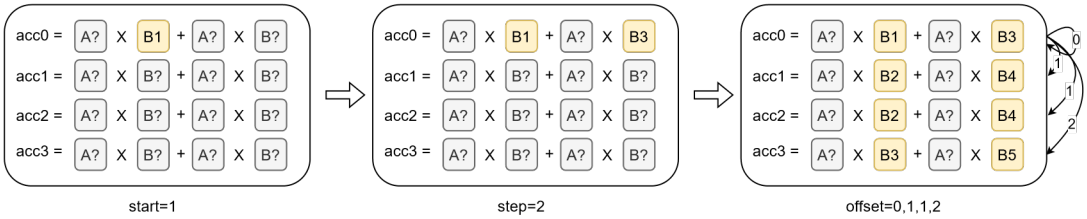


Figure 18. AIE shuffle network pattern for buffer B

5 Metrics for Evaluation

Evaluation of a device could include a broad set of metrics depending on the focus of different users. In this work, we propose the major type of users that represents a specific case in order to have a defined scope: "FPGA user selecting next generation device for ML usage". The evaluation consists of deploying multiple ML models to the AIE device, along with the same set of models deployed to the FPGA device to form a comparison. Several metrics are applied to this evaluation: Initiation Interval (II), latency, power, price, on-chip resource utilization and silicon area utilization. Those together help evaluate how good the AIE (versus the FPGA) is, with aspects of performance, power, cost and ability of replication.

5.1 Initiation Interval

An initiation interval (II) is the time period that must elapse between processing consecutive inputs (two different inputs for inference in our case) of the system. It is inversely proportional to throughput and both characterize the input data consuming speed of a system. It is one of the key performance indicators of the device. Specifically, for a machine-learning data post-processing system of high-energy particle collider, the II, or throughput, determines the minimum time interval between two successive collisions, in order to perform the data analysis in real time.

The reason we are using II instead of throughput is that the AIE operations that we used in our benchmarks are all deterministic, thus II better describes the regularity of the system. Another

reason is, for the remaining metrics, smaller values mean better efficiency. The adoption of II aligns with this pattern and helps us better visualize the data in the analysis section.

5.2 Latency

Latency is the other performance-related metric, describing the time period that elapses between when the data is consumed by the system and the corresponding result is produced by the system. It does not have correlation to II, a system can have both small II and large latency, such as a deep pipelined CPU. Take our example of a particle collider again, its input data cannot be discarded until data post-processing is done, thus the latency matters because it affects the amount of data we have to store or hold. For example if we have two post-processing setups with the same II but different latency; the setup with larger latency would have more input data that is in the process of computation, meaning that more data has to be stored.

5.3 Energy Consumption

Energy is another dimension for evaluating efficiency. The energy consumption of machine learning models is receiving increasing attention, driven by the widespread deployment of AI and the trend toward more complex machine learning models. For mobile use cases, limiting energy consumption is essential due to battery capacity constraints and the heat generated during operation. On the other hand, for data center applications, higher energy consumption can scale significantly, increasing energy costs and potentially exceeding local power limitations. In this work, both the static power and the dynamic power of the chip should be considered, to gain a better understanding of the contributing factors.

5.4 Cost

Cost is referenced when considering the purchase of a device, and it plays an important role when other metrics are worse but the cost is low: For example if the performance of the device is not

competitive but the price is reasonably low, we can buy multiple copies of the device and deploy replicated ML models to have better throughput to compensate the performance.

5.5 Resource Utilization

Resource utilization is the percentage of the on-chip resource usage of our neural network models. It is a direct indicator of the ability to replicate logic. Similar to the example of the cost, if resource utilization of a given model is low, multiple replications could fit to a single chip and compensate for the performance. On the other hand, resource utilization indicates the total capacity of the device. Lower utilization means larger models could be mapped to the chip.

5.6 Area Utilization

Area utilization is the percentage of the silicon area usage of the chip, it unifies the resource utilization result to a common and more comparable level. Other than the resource utilization, we still want to know the "resource efficiency": is this architecture suited for ML workloads and can support the functionality in a reasonably small silicon area? The area utilization metric will help us answer the question.

5.7 Core-Time

Core-Time is a metric that we propose for characterizing the overall model implementation's efficiency on AIE. It is the product of initiation interval and the hardware resource amount, showing the compute resource occupied for processing single input. It is also the inverse of throughput per AIE. The definition of core-time is shown below:

$$Core - Time = II * N_{AIE} = 1/\frac{Total\ throughput}{N_{AIE}}$$

II is the initiation interval of the implementation, and the N_AIE is the number of AIE cores involved in the design. The idea is to show the efficiency of the design and indicate if an

optimization really helped. It also reveals the correlation between II and AIE amount: If we spent 5 AIEs and got II of 100 cycles, then spending double the amount of AIEs (10) for the design should get half the II (50 cycles) theoretically. In the following section the core-time helps us to understand the potential difference between our iterations of attempts at optimization. If the core-time decreases, it means we use AIE cores more efficiently in terms of computation and vice versa.

6 Benchmark Implementations

For our targeting user group "FPGA user selecting next generation device for ML usage", using some example machine learning models is a direct way to demonstrate and benchmark the effectiveness. Here we selected models from our group's previous work[Johnson23]. These models are simple machine learning models that represent generic building blocks of larger deep learning models. We deploy those models on both the FPGA portion and the AIE portion of the same device to have a comparison. The following paragraphs focus on the deployment on the AIE, as the deployment on the FPGA was already done in the previous work.

In this work the model is quantized to 8-bit and 16-bit fixed-point during the deployment, as floating point and higher bit width of fixed-point implementation lead to much more hardware resource usage but bring little performance improvement. According to [Han15], an AlexNet could be quantized to 8-bit without any loss of accuracy. Our implementation flow is, explore the AIE's capabilities and have performance considerations during the 1-D 8-bit model implementation, then complete the 2-D and 2-D stride model 8-bit precision construction, and then finally extend the model to 16-bit precision.

6.1 1-D Model

The 1-D model is the first model of our benchmark, it is trained on the UCI Human Activity Recognition dataset[Reyes-Ortiz13] and consists of 4 layers: dense1, relu, dense2, sigmoid. It serves as a simple starting point for exploring the AIE's basic operations, architectural features and potential pitfalls, while still representing a key machine learning workload.

Taking a closer look at the model, its input count is 10, dense1 has 32 different sets of weights to make 32 outputs. Dense2 has a single 32-element weight to produce 1 output. Dense1 requires 320 MACs and dense2 requires 32 MACs.

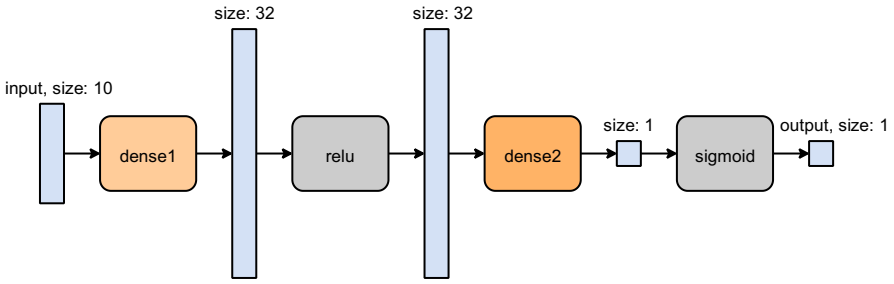


Figure 19. The 1-D model structure

6.1.1 Initial Version

We began mapping the 1-D model by implementing each layer with a dedicated AI Engine and managing the data flow in a streaming manner. For the AI Engine programming model, we need to address two separate parts: the computation kernel and the dataflow graph. This approach allows us to define the scope of each computation kernel intuitively and ensures that the data streaming matches the sequential execution of the model. The overall structure of the AI Engine design is shown below:

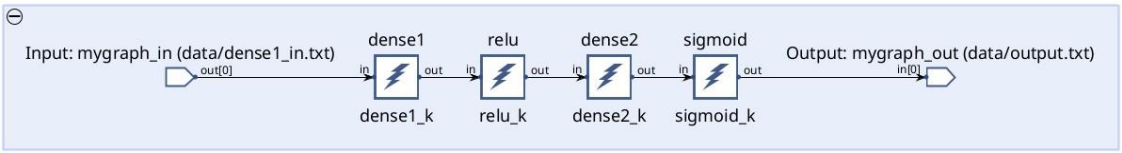


Figure 20. Initial version of AI Engine design

There are three types of layers, and we will discuss the AI Engine mapping of each one individually, with code examples. The dense layer handles most of the computation, and is inherently supported by the AI Engine MAC data path. It is capable of doing 128 int8 MAC operations in one clock cycle, managed by MAC intrinsic functions. We have to first fill the two input buffers of the MAC data path and then call the intrinsic to execute the actual computation. Here we have 10 elements as input but we have to zero pad it to 16 elements to adapt to the limitations of the vector data path, thus the total MAC operations are 16*32 = 512 and we need 4 MAC cycles to finish it.

```
void dense1_k(input_stream_int8* in, output_stream_int8* out) {
    while(1) {
            for(int i=0; i<4; i++) { //tile the computation
                aie::vector<int8,128> weight =
             aie::load_v<128>(weight_table_ptr+i*128); //load weight buffer
                aie::vector<int8,16> data_in = readincr_v16(in);
                aie::vector<int8,32> data_in_pad = aie::concat(data_in, data_in); //get
                data input buffer ready
                aie::accum<acc48,8> acc_partial = mac8(bias[i], weight, 0, 0x1110,
                16, 0x3120, data_in_pad, 0, 0x0000, 2, 0x3210); //the intrinsic, with
                shuffle network configurations
                writeincr(out,  acc_partial.to_vector<int8>());  //convert  data  type
        back and stream out
            }
     }
}
```

For the relu layer, there is a vector API available, specifically vector comparison (aie::max()) and this accepts 16 int8 inputs. We implement the ReLU function by comparing input numbers with zero and keeping the maximum value.

```
void relu_k(input_stream_int8* in, output_stream_int8* out) {
      while(1) {
            aie::vector<int8,16> read_in = readincr_v16(in);
            aie::vector<int8,16> out_data = aie::max(read_in, (int8)0);
            writeincr(out, out_data);
```

```
        }
    }
```

The sigmoid layer computation includes division and exponentiation, neither of which are directly supported by the AI Engine. Fortunately, with only 1 input and 1 output, we can implement this function using a look-up table. We precompute a table of input-output mappings with 256 entries, store the entire table in the AI Engine memory (maximum 32KB, allowing a 32K-entry table with 8-bit data), and return the result by using the input as an index to reference the corresponding value. This approach avoids heavy computation by using memory resources.

```
void sigmoid_k(input_stream_int8 * in, output_stream_int8 * out) {
    while (1) {
            int8 data_in = readincr(in);
            writeincr(out, LUT[data_in+128]); //index should be positive
    }
}
```

On top of these layers, we have to specify the dataflow graph to manage the entire process. This leads to two separate topics: AI Engine communication with the outside world and AI Engine communications with each other.

We limit data coming from the fabric to an int8 data type, to have a fair comparison to our FPGA approach. The only way that AI Engine supports this is to stream the data in. The output follows the same approach. For data transfer between layers, we initially use streaming, as it is relatively easy to implement and keeps the consistency with the interface I/O.

```
class graph_1D : public adf::graph {
private:
        kernel dense1;
        kernel relu;
        kernel dense2;
        kernel sigmoid;
public:
        input_plio in; //interface to the FPGA logic
        output_plio out;
```

```
    graph_1D() {
        connect<stream>(in.out[0], dense1.in[0]);
        connect<stream>(dense1.out[0], relu.in[0]);
        connect<stream>(relu.out[0], dense2.in[0]);
        connect<stream>(dense2.out[0], sigmoid.in[0]);
        connect<stream>(sigmoid.out[0], out.in[0]);
    }
};
```

Before conducting the actual simulation and measurement, we perform a theoretical estimation of the II and latency. The II of 1-D implementation should correspond to the cycles required for the slowest AI Engine computation kernel, as they work sequentially and the data is streamed; Other faster kernels have to stall and wait for data to be streamed in or to be consumed. The required cycle depends on the number of lines of assembly executed by the AIE.

$$II = N_{\text{assembly executed}}$$

The II can be either I/O bound or computation bound. For the 1-D model, the I/O involves streaming in 10 int8 data elements and streaming out 1 int8 data element. Since the data path operates on 16 int8 elements, and there is no efficient API for gathering non-power of two amounts of elements and fitting them to the vector that AIE accepts, we have to pad the input to 16 on the data sender side. An AI Engine can do 32 bits per cycle per stream-in port, so it takes 16*8/32=4 cycles to stream in data per inference. Output requires only 1 cycle since there is only 1 element.

The compute bound II of the 1-D model depends on each kernel's operations. The dense1 kernel requires 4 cycles because it has to do 4 128-MACs and each MAC costs 1 cycle. The relu kernel requires 2 cycles because it can perform a 16-element vector comparison and there are 32 elements per inference. The dense2 kernel is only doing 1 128-MAC and thus requires 1 cycle. Finally the sigmoid kernel also requires 1 cycle because it is only doing a table look-up. To summarize, both the I/O and compute bound II are 4 cycles based on our estimation.

The latency of this 1-D implementation should be proportional to the number of AI Engines and the slowest AI Engine computation time, which is II. It is not the sum of each AIE's specific II, because the slowest AIE would stall the stream data port of the incoming AIE, and propagate to the beginning AIE. For the worst case, the slowest AIE is at the end of the sequential execution, and every AIE involved is stalled. (Note that this rule may no longer hold for more complex implementations, since there could be a large overhead before the while(1) loop starts, thus making the latency even larger. Also, the compiler scheduling of loop unrolling could make latency unstable. For example, it could schedule all of the read operations at the top of the loop and make all of the write operations at the bottom.)

$$Latency_{max} \leq N_{AIE} * II$$

Among all these four computation kernels, the heaviest computation is happening in the first layer/kernel, since 512 (padded from 320) MACs should be done per inference. An AI Engine is capable of doing 128 MACs per cycle so computation needs 4 cycles as well. For Latency, there are 4 AIEs working so the pessimistic latency should be 4*4=16 cycles.

However, after running the simulation and obtaining the trace, the metrics were not as expected: the II is 24 cycles, the latency is 261 cycles, and core-time is 4∗24=96 cycles. To have detailed information of the AIE execution, we dumped the report and inspected it with Vitis Analyzer. One helpful part of the report is the trace, it keeps track of every involved AIE's status throughout the entire simulation timeline, for example what function it is executing, whether it is stalled, etc. The following graph shows a segment of the trace of two AIEs executing dense1 and relu layers respectively. We can observe that the upper row is for dense1 kernel, sometimes it is stalled by the stream, meaning that it is waiting for the data consumer to be ready. In contrast, the second row is for the relu layer, it is always executing the relu computation kernel without any interrupt. From

this we can infer that the relu layer is a potential performance bottleneck, as other layers have to wait for it.

Figure 21. Trace of AIE kernels

Next, we narrowed down our focus to one AIE that we can inspect closely with the help of the assembly profile. The figure below is the original code of the relu kernel and its corresponding assembly. To the best of our knowledge, the VMOV0 and VMOV1 are the stream read and write along with the VLDA.SPIL and VST pair to interface with the stream, and the VCMP is the actual comparison (max function). Other than these we can observe the VST.PACK, VLDA.UNPACK pair, which is doing vector data pack/unpack with the help of data memory but our written code did not ask for it.

```
while(1) {
  aie::vector<int8,16> read_in =
  readincr_v16(in);
  aie::vector<int8,16> out_data =
  aie::max(read_in, (int8)0);
  writeincr(out, out_data);
}
```

```
NOP; NOP; VMOV1 WMS.md0[10], vrh0, #0
NOP
NOP; VCMP xd, r2, ya.s16, r4, c0, r4, c1, c2
NOP
VST.PACK.s8 wd0, [sp, #-16]; NOP; VMOV0 vrh1, WSS.md0[ 8]
NOP; VLDA.SPIL vrh0, [sp, #-16]; NOP
VST vrh1, [sp, #-16]; J #816; NOP /* control_operation: jump
VLDA.UNPACK.s8 wr0, [sp, #-16]; NOP; NOP
NOP
NOP
NOP
NOP
```

Figure 22. Assembly of the relu kernel

This identifies the root cause of the discrepancy: Although the vector comparison API supports 16 lanes of 8-bit vector comparison, the actual hardware only supports 8 lanes of 16-bit comparisons. Consequently, the compiler inserts vector unpack operations (conversion to 16-bit) when reading from memory and vector pack operations (conversion back to 8-bit) when writing to memory. The last four NOPs (no operation) are introduced because of the branch delay slot and the compiler. As

a result, the originally expected 1 cycle per computation becomes 2 (two executions for one inference) *12 (lines of assembly) = 24 cycles.

6.1.2 Relu Enhance

Since we have identified that the relu layer significantly impacts our performance, our goal for this version is to optimize this specific layer. A straightforward approach is to replicate the relu computation across multiple AIEs, assigning workloads to subsequent AIEs in a round-robin fashion while previous ones are still running. Then outputs from those AIEs are gathered in the order of issuing.

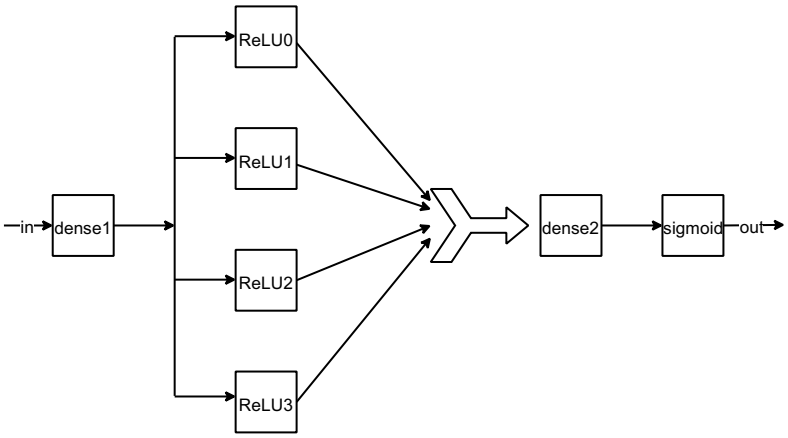


Figure 23. Round-Robin method V.1

However, this approach is not desirable because the kernel performing the reduction constantly checks the packet ID, introducing branch instructions that harms the pipeline. Additionally, the packet-switching method leads to a non-deterministic data path, resulting in different packet sequences between inferences. This necessitates an alternative round-robin approach with a deterministic path, albeit with some overhead:

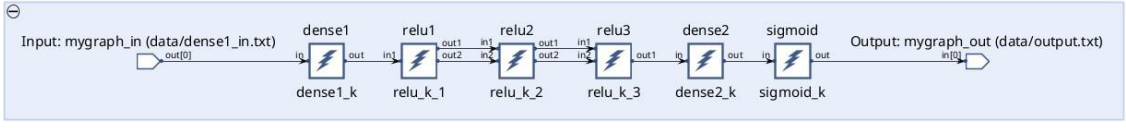


Figure 24. Round-Robin method V.2

The idea of this implementation is that each relu kernel processes its designated data position and passes the remaining data without processing. For example, in the figure above, relu_k_1 passes the first two elements and processes the third one; relu_k_2 passes the first one, takes the second one and passes the last one; relu_k_3 takes the first one and passes the next two.

In this version, the II is 17 cycles, the latency is 322 cycles and the core-time is $6*17=102$ cycles. As observed, to achieve a better II, we added many AI Engine kernels sequentially, which significantly increased the model's latency. Our next step is to balance good II and latency. Some sequential replication is still allowed, but we aim for more parallel replication. We also intend to eliminate some kernels by merging two fast kernels to one without affecting the original II.

6.1.3 Dense Enhance

After enhancing the ReLU layer, we noticed that the next bottleneck is the dense layer. In our previous analysis, we mentioned that each AIE can perform 128 MAC operations per cycle and the dense1 layer requires 512 operations per inference (after padding). However, actual execution shows that it took 17 cycles to finish the dense task instead of 4. This discrepancy is due to the need to load the weights into the vector register file before each MAC operation: The buffer of the MAC data path has a maximum capacity of 1024 bits (128 of 8-bit), the MAC operation of the dense1 layer has to load corresponding part of the weight for partial result computation. With 512 of 8-bit weights and a load bandwidth of 256 bits per cycle, a minimum of $512\times8\div256=16$ cycles required. To mitigate this problem, we distributed the dense layer workload to two AIEs. Originally, one AIE handled 32 outputs each iteration, now each AIE handles 16 outputs.

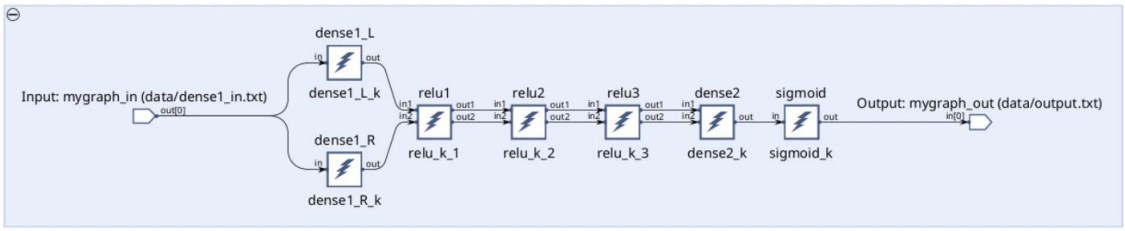


Figure 25. Dense enhance implementation

With this adjustment, the II is now 12 cycles, the latency is 236 cycles and the core-time is 7∗12=84 cycles. Once we doubled the throughput of the dense layer, the entire model's II is again limited by the relu layer. Note that for stream data transfer, each AIE can only accept input data streams if its computation kernel is not stalled, and it can only output data streams if the next stage is not stalled. If the load is not balanced between each stage, upstream faster stages remain idle, waiting for slower stages.

6.1.4 Dense Sigmoid Merge

As we observed in the trace of the sigmoid and the dense2 executions, there were large stalls between each execution. These layers are relatively fast and stalled nearly half of the time. By implementing both layers on a single AIE, we can better utilize the hardware resources and hide the stream stalls. This is the final version of our 1-D model, its II is still 12 cycles but the latency has improved to 216 cycles, and the core-time has improved to 6∗12=72 cycles.

Figure 26. Stalled sigmoid and dense2

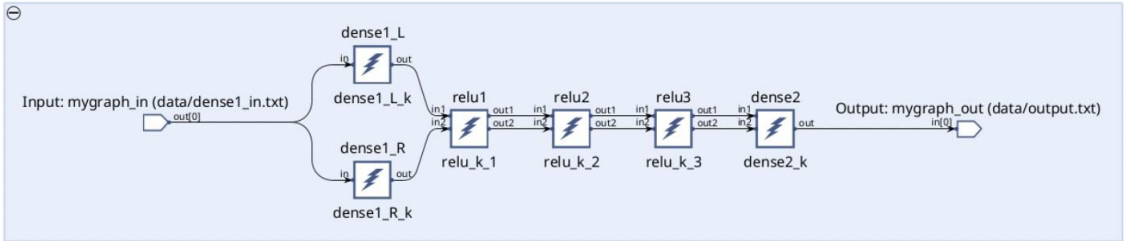


Figure 27. Dense2 sigmoid merge implementation

6.1.5 16-bit Extension

Based on the experience we gained from 8-bit implementation, the 16-bit version of 1-D model is designed with proper performance consideration: To avoid the dense1 layer performance bottleneck, we now use 4 AIEs for the dense1 layer, as the MAC unit could only do 32 of 16-bit MAC per cycle, in contrast to 128 of 8-bit MAC per cycle. Note that we did not use the round-robin method for the 16-bit relu layer, as there is no more pack and unpack involved under this bit precision. The II is 8 cycles and the latency is 156 cycles for this design.

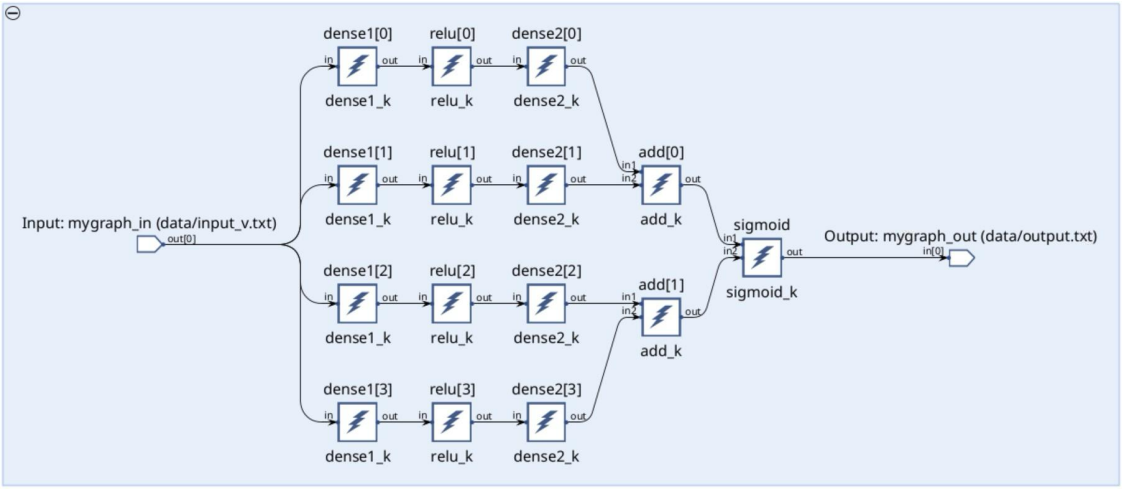


Figure 28. 16-bit implementation of 1-D model

6.2 2-D Model

The 2-D model is our second benchmark, it consists of 4 layers: convolution (conv), relu, dense and softmax. It was trained on the MNIST handwritten digit dataset[Deng12]. The model is still a 4-layer model but consists of a new pattern of heavy computation (conv) and a more complex activation layer (softmax). The conv is a challenge for AIE because the sliding window pattern is not so friendly to the vector datapath, and the softmax is difficult to accommodate because its output elements are relevant to each other and might limit parallelism.

Take a closer look at the size of each layer, the input count is 8x8, and the conv has two 3x3 kernels to make an output of 2x8x8 (with padding). Later, the dense uses10 sets of 2x8x8 weights to

produce 10 outputs. Finally the softmax layer performs its computation and keeps the 10 output elements. For this model, we will primarily focus on the implementation of the convolution layer, as we have already gained substantial knowledge from our experience with the relu and the dense layer.

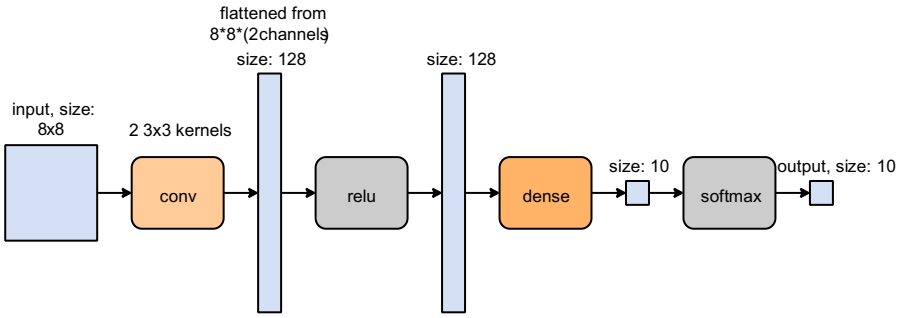


Figure 29. The 2-D model structure

6.2.1 Initial Version

To begin, we initially map each layer to one AI Engine. However, due to the nature of the convolution kernel, it is intuitive to implement separate convolution kernel workload on different AI Engines. Based on our estimation, we expect the conv layer's II to be around 24 cycles per image (to be introduced later). However, if we implement the subsequent dense layer on a single AIE, its II would be at least 1280∗8/256=40 cycles, as it needs to load corresponding 1280 8-bit weight elements for each MAC operation using its 256-bit data path. From this quick estimation, we decided to split the workload of the dense layer across two AIEs, and the relu layer is also split on two AIEs to avoid unnecessary broadcast and reduction.

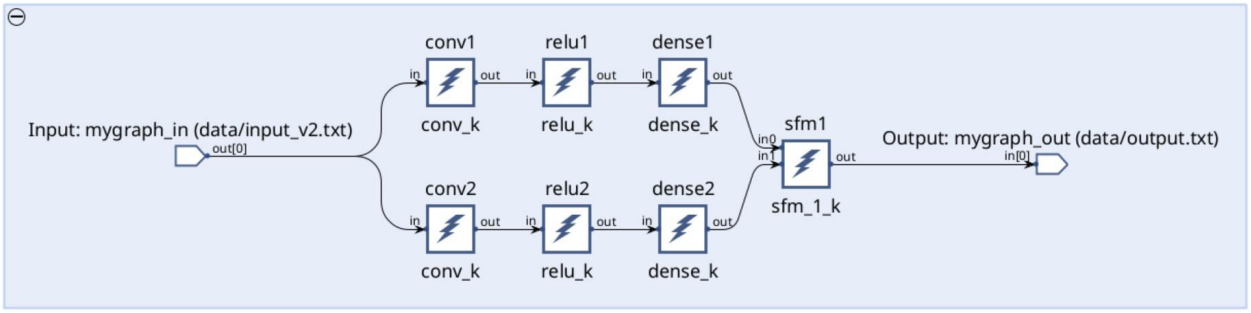


Figure 30. Initial version of 2-D AI Engine mapping

Apart from the relu and dense layer introduced in the 1-D model, the implementation of remaining layers is as follows:

For the conv layer, inspired by [Chatarasi20] and [Ho23], we utilized the vector data path by calculating an entire row of partial results each time. As shown in the figure below, consider the 3x3 convolution of the dashed line-highlighted range in a 8x8 (zero padded to 10x10) input image. The conv kernel performs 8 times of convolution in total and results in a row of 8 elements in the output image.

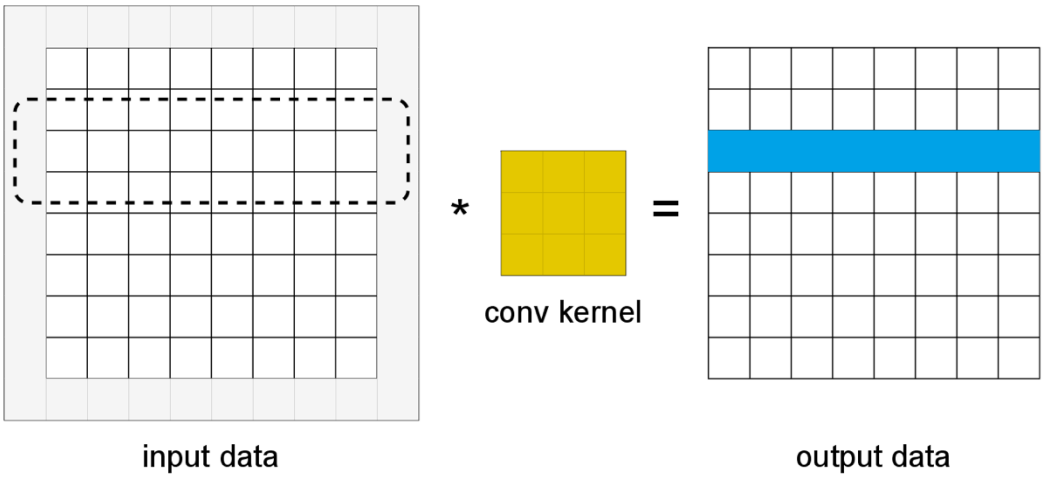


Figure 31. Convolution on a specific row of input image

From there, the computation could be split into three iterations as shown below. The first iteration computes the convolution across the first row of the input image and the first row of the kernel weight, and stores the partial result in the output buffer. Then the second iteration works on the second row, accumulating the result with the previous iterations. Finally the third iteration gets the partial result of the third row and accumulates it with the previous result to get the overall output.

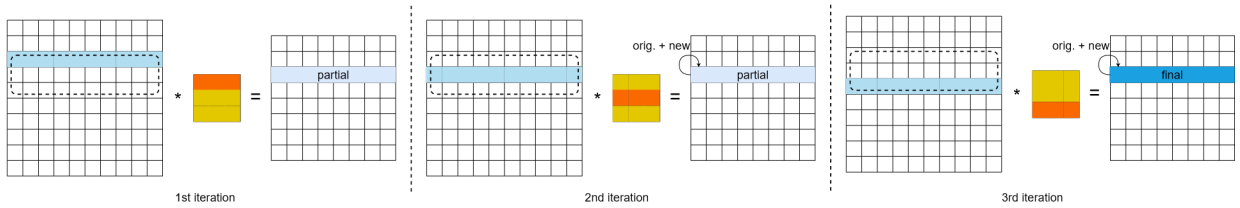
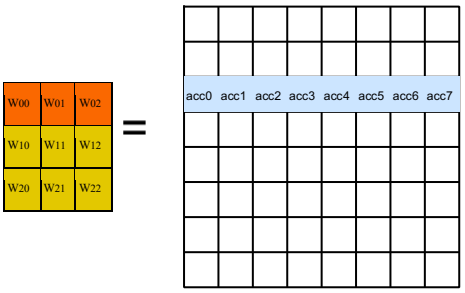


Figure 32. 3 iterations of the convolution of a specific row

For each iteration, the computation could finish in only one cycle with the help of the MAC engine, as long as the input data has already been written to the buffer. The rearrangement of the data from MAC buffer to the MAC computation of the 1st iteration is shown below. Note that we are only using less than half (24 of 128) of the MAC engine capability for this implementation due to the limitation of the shuffle network.



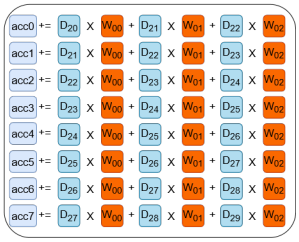


Figure 33. Convolution on a specific row with numbered elements

Figure 34. MAC Detailed mapping

Finally, the conv layer's code could be written as a nested loop, the outer loop gets one row of convolution output data per iteration, and the inner loop works on one row of the partial convolution (one row out of entire rows of the conv kernel) per iteration. In this way, with a 3x3 kernel and a 8x8 padded to 10x10 input, we are expecting a total of 8*3=24 cycles of vector MAC operations.

```
for(int i=0; i<data_row; i++) {
        aie::accum<acc48,16> acc = 0;
        for(int j=0; j<wgt_row; j++) {
                aie::vector<int8,32> row_buf = data[i+j];
                aie::vector<int8,32> wgt_buf = weight[i+j];
```

```
            acc = mac16(acc, wgt_buf, row_buf, SHUFFLE_PARAMS(i, j));
        }
        writeincr(out, acc);
}
```

For the softmax layer, we cannot use an easy look-up table as we did for the Sigmoid layer. For each output we have:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \; for \; i = 1, \ldots, k$$

Due to the nature of this computation, we cannot obtain the output until we have the sum. Our implementation uses a look-up table to translate each input element to its exponent, then adds them together and divides each exponent by the sum.

```
for (int i=0; i<10; i++) {
    int8 res = readincr(in);
    exp[i] = LUT[res+128];
    sum += exp[i];
} //first have the sum
int8 sum_inv = inv(sum); //using inverse and mul to avoid div
for (int i=0; i<10; i++) {
   writeincr(out, exp[i]*sum_inv);
}
```

From the simulation, we found that the II is 127 cycles, the latency is 510 cycles and the core-time is 7∗127=889 cycles. This is far from our estimation, mainly due to the softmax layer. We will apply round-robin optimization to address this issue.

6.2.2 Softmax Optimization

To estimate the number of replications needed for the softmax round-robin optimization, we have to identify the bottleneck among the remaining layers, which is the dense layer with an II of 60 cycles. Therefore, there should be at least 2 AIEs for the softmax layer. However, 2 AIEs won't be sufficient because if we want to use the round-robin method, each original kernel has to help with

accepting and forwarding the extra traffic introduced by this optimization, thus introducing overhead and thus cannot reach the target II of 60. It is preferable to use 3 AIEs to mitigate this issue, and they will function similarly to our 1-D ReLU round-robin implementation. With this setup, the II is now 60 cycles as expected, the latency is 596 cycles and the core-time is 9∗60=540 cycles.

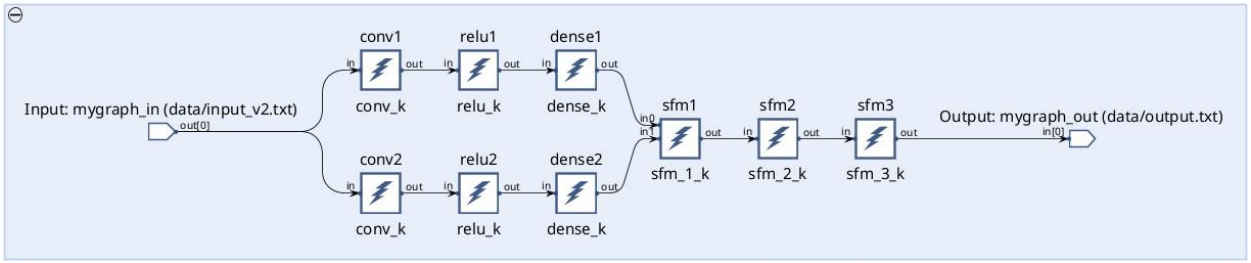


Figure 35. 2D model with softmax round-robin

6.2.3 16-bit Extension

The 16-bit extension of the 2-D model has an identical graph layout as the 8-bit version of the 2-D model. The relu layer implementation remains the same because of the unnecessary pack-unpack, and the softmax layer is still implemented by look up table.

We focus on the convolution layer first: the 8-bit implementation only uses 24 out of 128 MACs per cycle due to the limitation of the shuffle network. For the 16-bit MAC, we can still do 24 MACs because the MAC unit can do 32 MACs per cycle. Thus, the convolution layer's performance will not be downgraded if we keep the same implementation.

As for the dense layer, we still use two AIEs for it, but with different internal designs. As mentioned in section 6.1.3, the 8-bit implementation of dense suffers from long vector data loading time. However, for the 16-bit implementation this problem is alleviated: The length of the operand is reduced from 1024 bits to 512 bits and now loading the operand costs only two cycles. In this way

we could map multiple vector MAC operations to a single MAC unit in a sequential manner without a large overhead. The II is 88 cycles and the latency is 681 cycles for this design.

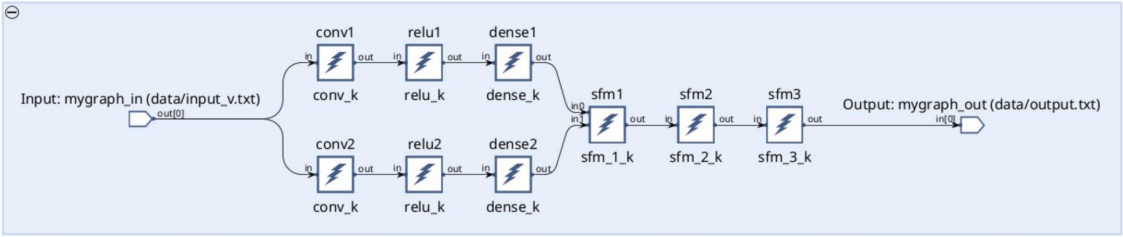


Figure 36. 2D model 16-bit implementation

6.3 2-D Stride Model

The 2-D stride model is our third benchmark. It consists of 4 layers. Convolution (conv), relu, dense and relu. This benchmark is the encoder portion of the Econ-T Autoencoder[Weng23] and it was trained on the data produced by the Compact Muon Solenoid (CMS) Endcap Calorimeter at CERN[CERN17]. The key difference between this model and the 2-D model is the conv layer, the stride parameter of conv is set to two in this model. This new requirement is a potential challenge to the AIE as we have already known that the AIE has limitations on its shuffle network.

Taking a closer look at the size of each layer, the input count is 8x8, and the conv has eight 3x3 kernels to make an output of 4x4x8 (with padding). Later after flatten and relu activation, the dense has 16 sets of 4x4x8 weights to produce 16 outputs and will be activated by relu again. For this model, we still primarily focus on the implementation of the convolution layer with stride, as the remaining layers are already implemented in the 2D model.

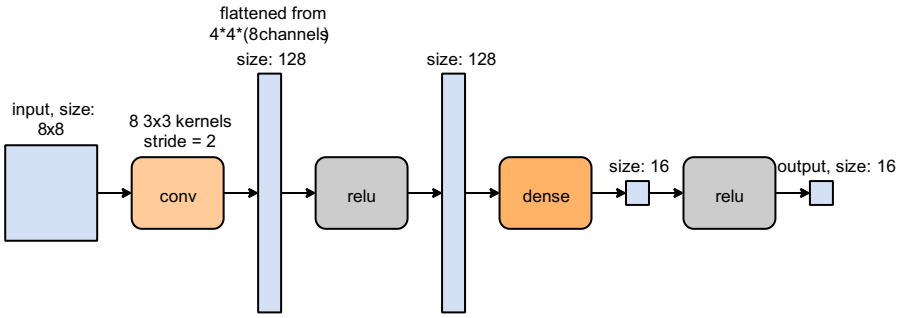


Figure 37. The 2-D stride model structure

6.3.1 8-bit Implementation

As expected, the implementation of conv layer encounters multiple times of the data handling problems. After a moderate amount of design space exploration, we were unable to find a more efficient implementation than the original 2D conv layer's. Now with the row-based convolution, we are only using 12 MACs out of the entire 128 MACs per cycle because of the stride of 2.

The total amount of conv kernel is 8, thus we spent 8 AIEs in parallel for the conv layer. However, the output should be reduced to only one stream. Thus the remaining layers also serve as a reduction tree, accepting two inputs, maintaining the order and producing only one output stream. The II is 44 cycles and the latency is 354 cycles for this design.

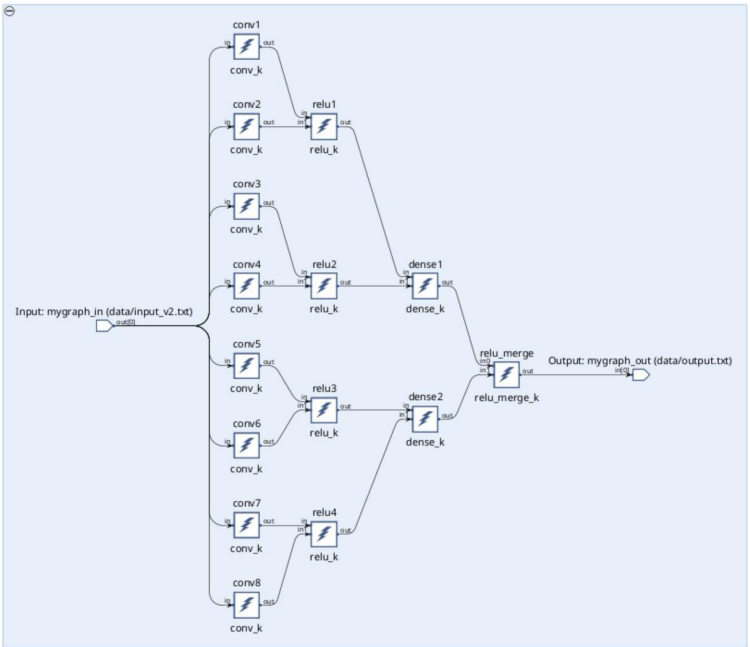


Figure 38. 2D stride model 8-bit implementation

6.3.2 16-bit Extension

The 16-bit extension of the 2D stride model also keeps the original graph layout, for the same reason mentioned in section 6.2.3. The II is 64 cycles and the latency is 271 cycles for this implementation.

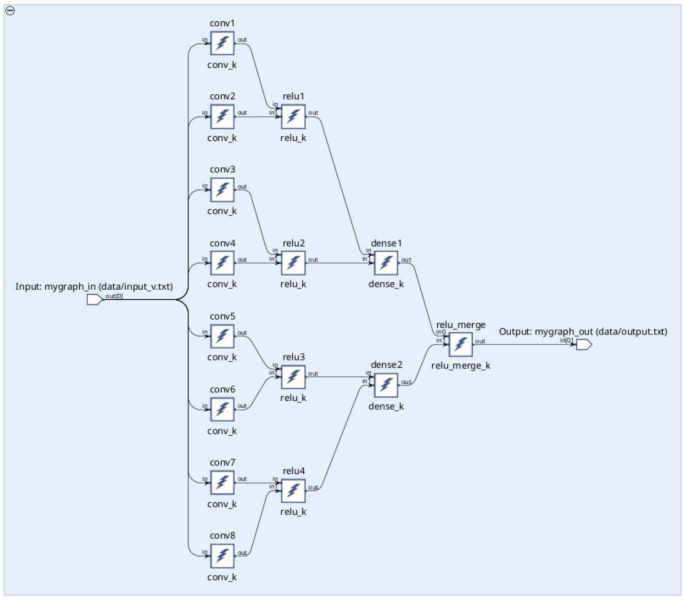


Figure 39. 2D stride model 16-bit implementation

7 Analysis

7.1 1-D Model AIE Roadmap

First from the optimization roadmap of the 1-D AIE 8-bit implementation, we can create a graph showing the relationship between the II and the number of AIE cores. From the graph, we can observe that increasing the number of AIEs results in a better II for the model.

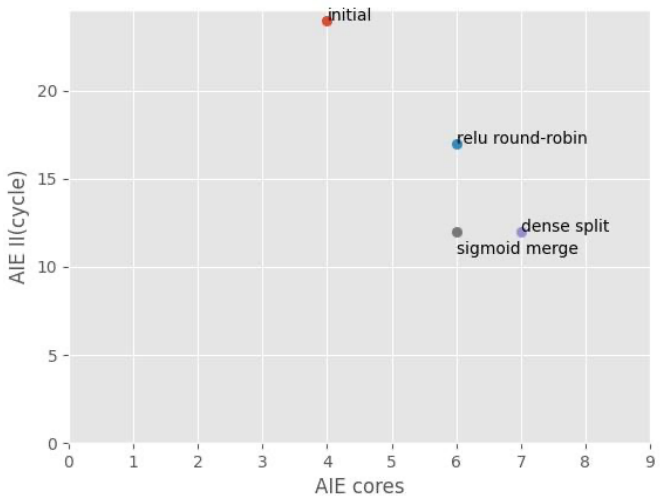


Figure 40. 1-D II vs. hardware resource

Next, we present the latency versus AIE cores graph. Interestingly, the relu round-robin optimization actually made the latency worse than the initial version. This is due to the overhead of data streaming introduced by this mechanism. Another observation is that the sigmoid merge resulted in better latency than the dense split version. This improvement is because the workloads of two AIEs were merged into a single AIE, eliminating idle periods during the execution of those two separate AIEs.

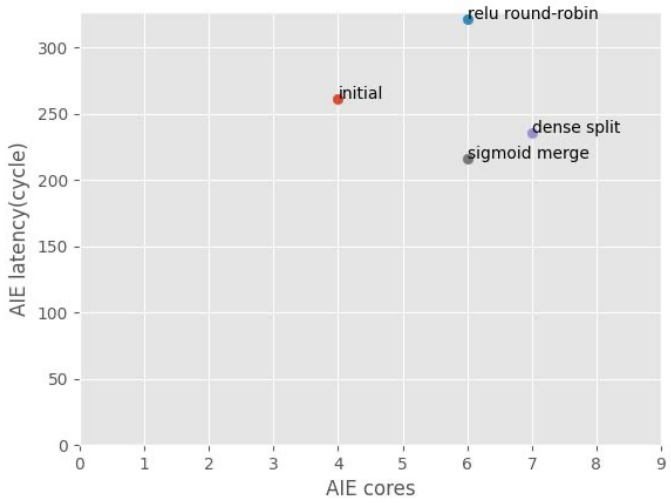


Figure 41. 1-D latency vs. hardware resource

The final part of the 1-D 8-bit optimization roadmap is the graph of core-time versus latency. We can observe that the round-robin method introduced both higher core-time and higher latency while we are trying to get better II. This is because the next performance bottleneck (Dense) falls way behind after the round-robin optimization and limits the entire system, making the core-time inefficient.

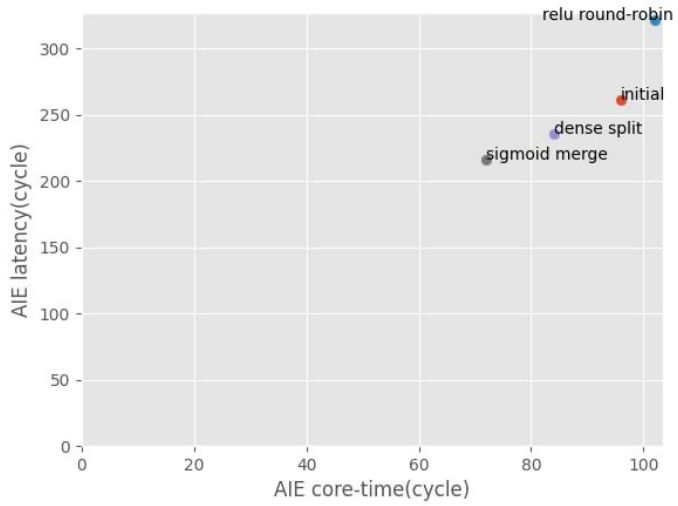


Figure 42. 1-D core-time vs. latency

7.2 1-D Model AIE Bitwidth

From the 8-bit implementation, the 16-bit precision model mapping is also implemented. The comparison of 8-bit last result and 16-bit implementation result is shown below.

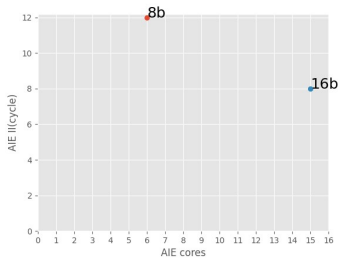


Figure 43. 1-D II vs. cores

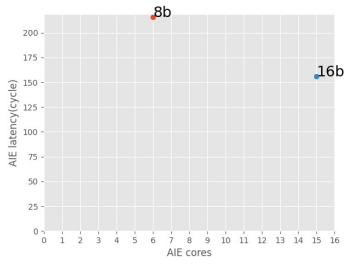


Figure 44. 1-D latency vs. cores

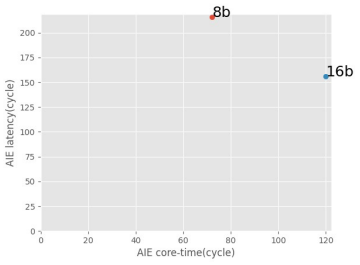


Figure 45. 1-D core-time vs. latency

The II of 16b model became better because it got rid of the Relu layer pack-unpack problem such that the II bottleneck could be further resolved. The latency of the 16b model is also better, as there is no more sequentially connected AIE introduced, and the II has been better. The core-time of the 16b model is more than the 8b model, since the computation amount increases with the bit precision.

7.3 2-D Model AIE Roadmap

The 2-D 8-bit implementation roadmap is simpler than the 1-D, since we have already got lessons learned from our 1-D implementation. Results are shown below.

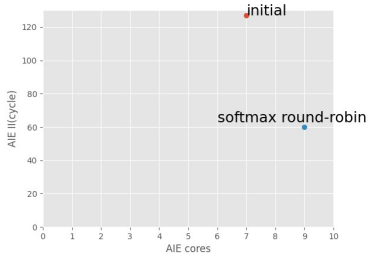


Figure 46. 2-D II vs. cores

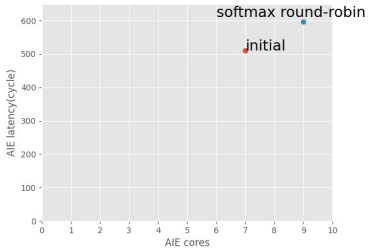


Figure 47. 2-D latency vs. cores

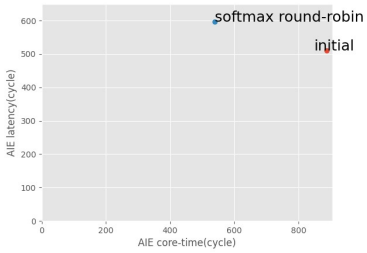


Figure 48. 2-D core-time vs. latency

The pattern is similar to 1-D as well, spending more AIE cores results in better II, but adding AIEs sequentially harms latency. There is a difference in the core-time vs. latency graph, the 2-D core-time goes down after the round-robin optimization. This is because the next bottleneck of the system in terms of II is close to the round-robin's goal and not stalling other kernels excessively.

7.4 2-D Model AIE Bitwidth

The 16-bit extension of the 2-D model from 8-bit has an interesting difference. It uses the same amount of AIEs for 8-b and 16-b implementations.

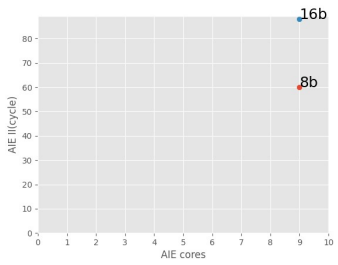


Figure 49. 2-D II vs. cores

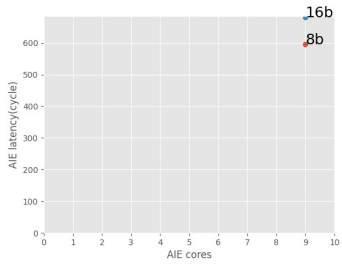


Figure 50. 2-D latency vs. cores

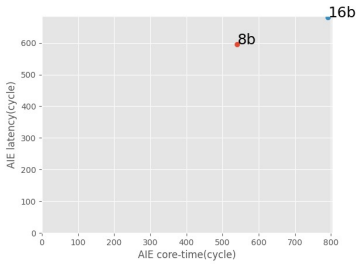


Figure 51. 2-D core-time vs. latency

We can observe that the II of 16b implementation grows about 1.5 times larger and the latency is still close to the 8b implementation while using the same amount of AIEs. This is because the 8b implementation has an inefficient usage of AIE computation resources due to data arrangement limitations. When migrated to 16b precision, the increased computation amount was handled by the remaining hardware resource. This could be also observed from the core-time.

7.5 2-D Stride Model AIE Bitwidth

The 16-bit extension of the 2-D stride model shares almost the same characteristics as the 2-D model. The major difference here is that the 16-bit version has less latency than the 8-bit implementation. This is because of the Relu layer performance again. In this model the relu layer also serves as a 2-to-1 reduction tree and the 8-bit pack-unpack affects the overall latency.

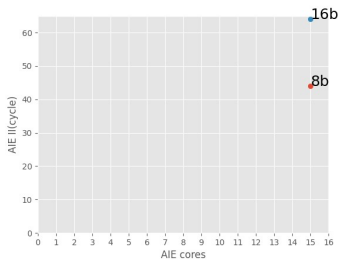


Figure 52. 2-D S II vs. cores

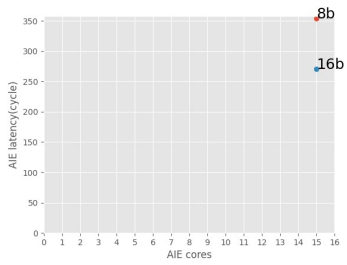


Figure 53. 2-D S latency vs. cores

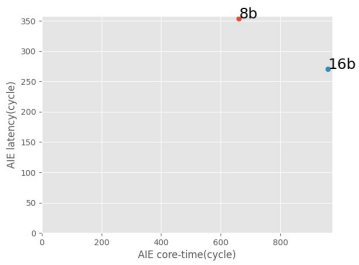


Figure 54. 2-D S core-time vs. latency

7.6 Methodology for AIE vs FPGA

So far, we have mapped our 1-D and 2-D model to AIEs and explored some optimization techniques. It's time to make a comparison with the FPGA implementation. Here, we revisit the major type of users that represents a specific case:“FPGA user selecting next generation device for ML usage”.

For this type of user, the Versal product family from AMD/Xilinx is a reasonable choice. It offers a wide range of capacities and different combinations of functional areas. The primary question explored in this thesis is whether to purchase a device with AIE functionality. This consideration leads us to consider the pricing of Versal devices. Furthermore, for a device containing both fabric and AIE, the user has to decide whether to use the FPGA or the AIE portion of the chip to meet their needs. The following measurements are useful for this decision:

- Initiation interval
- Latency
- Cost
- Energy consumption
- Resource utilization
- Silicon area utilization

For comparison and benchmarking purposes, we selected the VCK190 evaluation board (Versal AI Core chip XCVC1902) as the primary device. This AI Core chip contains both FPGA and AI Engine, providing users with the choice of implementing the ML workload on either portion. Also, there is another chip that contains no AIE but has identical remaining FPGA fabric resources: The Versal Prime chip XCVM1802 from the VMK180 evaluation board. We will use the data from

VMK180 to help calculate the AIE price. Below, we continue to introduce the methodology for comparison of each metric. We will refer to the AIE mapping of the ML model as "AIE", and refer to the FPGA mapping as the 'Verilog Neural Network' (VNN).

7.6.1 Initiation Interval

The II of the AIE is measured using the AI Engine emulator provided by AMD. The process involves compiling and loading the design into the AIE emulator, which is cycle-accurate and can dump timed output data from each AIE core. We then calculate the II with the equation below:

$$II = (T_{last} - T_{first})/(Inference\ count - 1)$$

Where T_{last} is the time stamp of the last output element of the entire test sequence, and T_{first} is the timestamp of the first output element of the test sequence. The inference count is the total number of inferences performed during the test sequence. Note that for this section the II's unit is changed from cycle to ns, in order to make comparison with the VNN.

The II of the VNN is measured in a more straightforward way. For our 1-D model, it consumes an entire 10 input per clock cycle, so the II is the clock cycle. We obtained the clock cycles from the Vivado implementation report.

For the 2-D model, the original VNN design of the conv layer consumes one pixel per clock cycle, meaning that II=input pixels∗clock cycle=64∗clock cycle. This design focuses on resource saving, thus having less competitive performance. To make the VNN more comparable, we replicated the conv layer design for a factor of three, each conv component is responsible for partial input image workload (top conv for row 0~3, middle conv for row 2~5, bottom conv for row 4~7), the overlap is required because each conv operation needs 3 consecutive rows of data to operate on. However, we didn't modify the VNN implementation of the 2-D stride model to make it more comparable,

as the entire design focuses on resource saving with detailed hardware scheduling with regard to the stride, and modifying the implementation requires extensive amounts of work.

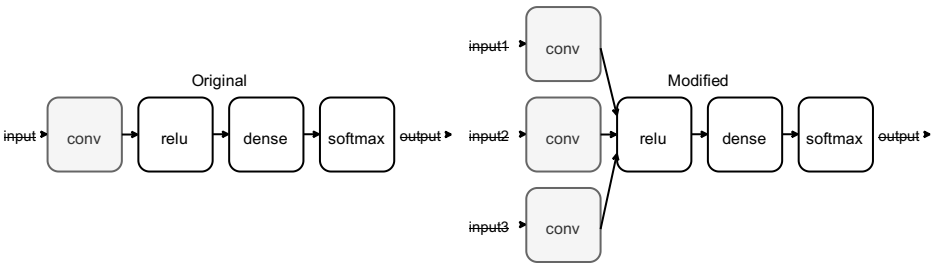


Figure 55. VNN 2-D model modification

7.6.2 Latency

For the latency data of the AIE, we did not use the output data time stamp directly, as there is a one-time overhead of AIE boot and program load, which should not be included in the latency measurement. Instead, we extracted the latency from the trace data dumped by the AIE emulator. The latency is the timestamp of the first output minus the timestamp of the first computation kernel that has been loaded by the main function. The latency of the VNN is measured by Verilog simulation. It is the first output's timestamp minus the time that the first input was consumed.

7.6.3 Cost

Device pricing is a complex process, with details that are trade secrets of the vendors. In this section we do a comparison based on available information but acknowledge there are unavoidable distortions due to hidden pricing information.

When buying a device, the total price is an important consideration. However, for a fair price comparison, we need to consider the corresponding price for specific resource utilization. Although there is no device that contains only AIEs, we can reference the pure AIE price as the price

difference between the FPGA-AIE combined chip (VCK190) and the pure-FPGA chip with the same resource amount (VMK180).

$$Cost_{AllAIE} = Cost_{Chipw/AIE} - Cost_{Chipw/oAIE}$$

As of June 2024, the price of each part (from DigiKey) is:

- VCK190 chip (XCVC1902-2MSEVSVA2197): $29748
- VMK180 chip (XCVM1802-2MSEVSVA2197): $11477

The cost of a model (AIE implementation) can then be obtained by multiplying the percentage of used AIEs, with total cost of AIE:

$$Cost_{AIE\text{-}impl} = \%AIE * Cost_{AllAIE}$$

The total cost of FPGA resources can be obtained directly from the VMK180:

$$Cost_{AllFPGA} = Cost_{Chipw/oAIE}$$

Finally the cost of a model (VNN implementation) is defined as shown below:

$$Cost_{VNN\text{-}impl} = \max(\%LUT, \%FF, \%DSP) * Cost_{AllFPGA}$$

Here in the formula we adopted maximum utilization rate among the FPGA fabric components, since from a replication perspective, if one type of the fabric is exhausted, remaining types cannot be used anyway but the user still has to pay for them.

7.6.4 Energy Consumption

We obtained the power data from the Xilinx Power Estimator (XPE). To ensure a fair and straightforward comparison, we left the ambient temperature and the junction temperature as the

default for both of the devices (25 Celsius ambient temperature, 100 Celsius junction temperature). Then the energy consumption of one neural network inference is calculated as the formula below:

$$\text{Energy consumption} = \text{power} * II$$

The II here is the initiation interval, the time it takes to consume one inference input and ready for another input. The time interval times the power of the device would be the total energy consumption.

7.6.5 Resource Utilization

The AIE resource utilization is obtained directly from the design, as the AIE code determines how many AIEs are used and which function an AIE executes. The FPGA fabric utilization is derived from the resource utilization report, generated after synthesizing the Verilog code to the Versal device in Vivado.

7.6.6 Area Utilization

Area utilization is related to resource utilization and the resource silicon area. However, we cannot access the real-world layout of the VCK190 chip to get the unit area. Instead, we use the layout demonstration from Vivado for our area analysis. It is possible that this layout differs from the actual layout, but we base our analysis on the best available information.

The figure below is the layout of the VCK190 chip from Vivado, where we measured the area of AIE and FPGA fabric. The purple area contains all of the fabric resources (FF, LUT, DSP) and the gray area at the top is for the AIE. We find that the AIE resources occupy 18% of the die size and the fabric occupies 67%. The remaining portions are I/O, CPU, etc. The formula we use to get the

silicon area utilization is shown below, the FPGA area utilization also adopted the maximum utilization among all resources, for the same reason as section 7.6.3.

$$\%Area_{AIE\text{-}impl} = \%Area_{totalAIE} * \%AIEused$$

$$\%Area_{FPGA\text{-}impl} = \%Area_{totalFPGA} * \max(\%LUTused, \%FFused, \%DSPused)$$

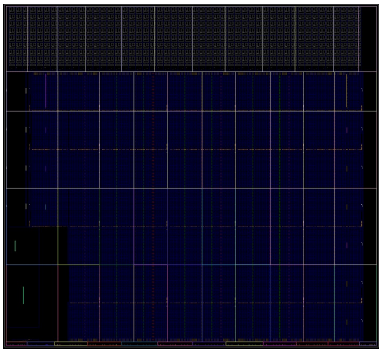


Figure 56. VCK190 layout from Vivado

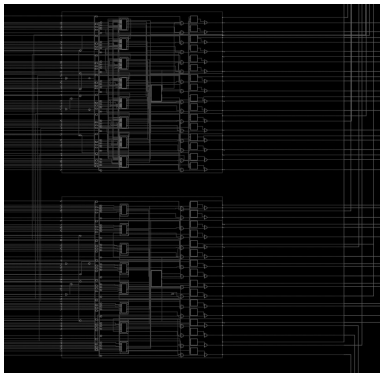


Figure 57. LUT and FF layout (closer look)

7.7 AIE and VNN Results

7.7.1 1-D Model

	II ns	Latency ns	Energy nJ	Cost	FF	LUT	DSP	AIE	Resource%	Area%
8-bit Implementation										
AIE-init	19.2	208.8	113.74	183	/	/	/	4	1	0.18
AIE-relu	13.6	257.6	84.32	274	/	/	/	6	1.5	0.27
AIE-dense	9.6	188.8	61.36	320	/	/	/	7	1.75	0.315
AIE-nal	9.6	172.8	59.55	274	/	/	/	6	1.5	0.27
VNN	1.6	25.6	12.5	52	3191	4045	0	/	0.45	0.301
16-bit Implementation										
AIE	6.4	125.2	48.07	685	/	/	/	15	3.75	0.675
VNN	2.3	36.8	19.13	200	8838	15660	9	/	1.74	1.17

Table 3. 1-D model metrics data

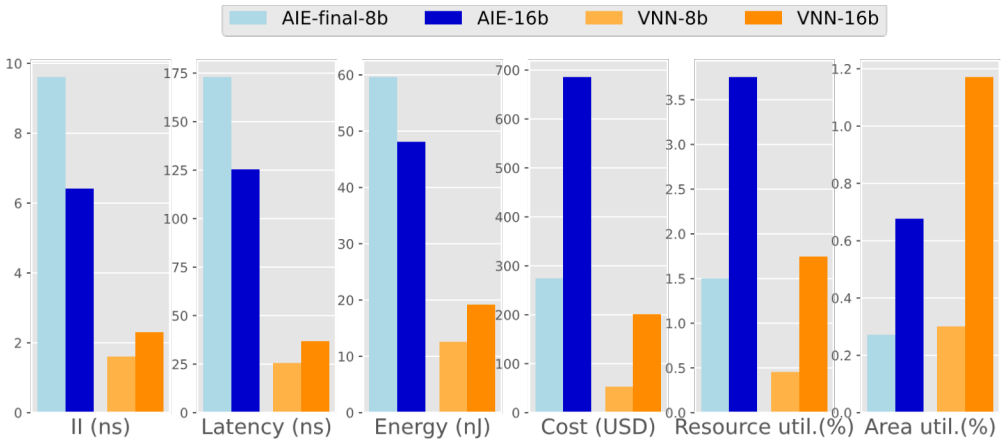


Figure 58. Key metrics of 1-D AIE and VNN comparison

From the data shown above, we can observe three aspects:

The 8-bit AIE design exploration shows an overall characteristic of AIEs: lowering II could also lower latency (as long as there is no large portion of sequential AIEs introduced) and with a cost of power and resource utilization. This correlation between II and latency is introduced by the synchronization mechanism of the AIE, as each AIE has to wait for previous AIE's data and would also be stalled if the subsequent AIE is not ready to receive its output. This is a key difference between AIE and FPGA, as FPGA can have a fine-grained pipeline and manually specify the synchronization to distribute the workload.

The AIE implementation versus VNN implementation shows that, for this specific model, the AIE's II is 6x worse than VNN, AIE's latency is 7x worse than VNN, AIE's energy consumption is 2x worse than VNN, AIE's price is 5x worse than VNN, while AIE has slightly better area utilization over VNN. This inefficiency is because the AIE cannot handle ultra-fast (<10 FPGA cycles) and tiny workload (<5% resource utilization): The AIE has a limitation of lowest II and spending more AIEs cannot help getting it better (detailed discussion in section 8.1. Another possible factor is

that, the VNN binds every operation to a specific fine-grained logic but the AIE is core-based architecture and the designer tends to map multiple operations to one AIE in a mixed (sequential and parallel) manner.

Comparing the 8-bit implementation against the 16-bit implementation also reveals that, for the 1D model, the VNN resource utilization grows 4x as the bit precision doubles. This is because the DSP is not heavily inferred and the multiplication uses 4x resources and it is the dominating operation. The AIE implementation also shows a counterintuitive result. As bit precision doubles, II and latency of the 1D model gets even smaller. One contributing factor is that we spent more AIE cores than in the 8-bit version, but another interesting factor is, the AIE core does not support specific 8-bit operations and using 16-bit operations could avoid data conversion overhead.

7.7.2 2-D Model

	II ns	Latency ns	Energy nJ	Cost	FF	LUT	DSP	AIE	Resource%	Area%
8-bit Implementation										
AIE-init	101.6	408	648.31	320	/	/	/	7	1.75	0.315
AIE-final	48	476.8	320.54	411	/	/	/	9	2.25	0.405
VNN	48.32	83.05	387.96	143	8154	11204	0	/	1.25	0.83
16-bit Implementation										
AIE	70.5	544.8	470.80	411	/	/	/	9	2.25	0.405
VNN	69.76	119.9	634.12	563	24662	44141	89	/	4.91	3.287

Table 4. 2-D model metrics data

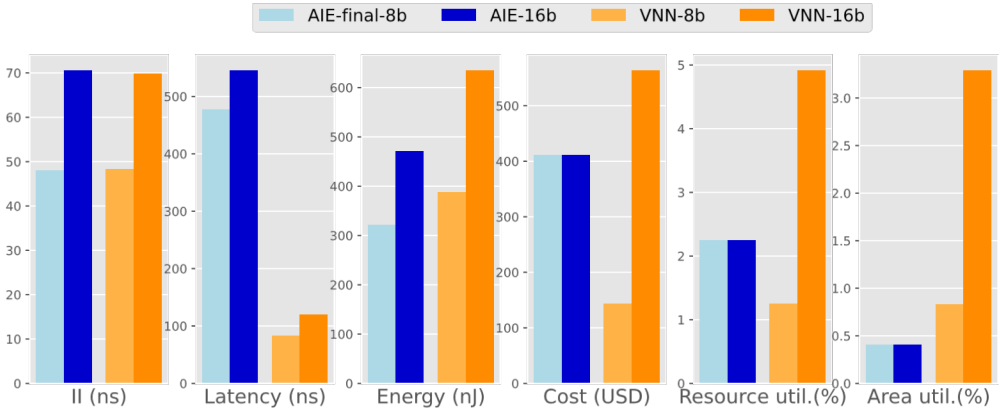


Figure 59. Key metrics of 2-D AIE and VNN comparison

From the plots of the 2D model AIE versus VNN data, we see that for this model, even though the AIE can reach the same II as VNN, the latency is still a gap between AIE and VNN. This is because the parallelization on the AIE side is limited by the data transfer method. If we want to map the workload to AIEs in parallel to reduce the latency, there is always an overhead of joining the data stream back together (detailed discussion in section 8.2).

As bit precision grows from 8-bit to 16-bit, the VNN resource utilization percentage exceeds the AIE percentage. This means AIE is desirable for lowering resource utilization for larger bit width by sacrificing the latency. The gain is from sequentially (time-multiplex) utilizing the AIE core resources instead of using the FPGA fabric in parallel. The area utilization difference between AIE and VNN also grew larger in this case and it is because the AIE has a dedicated and condensed unit for mathematical operation. It does not matter if the operation on the VNN side is mapped as LUT or DSP, the AIE's hardware is more resource efficient when the operation contains massive parallel computation and could be mapped to the MAC unit (for example dense layer).

7.7.3 2-D Stride Model

	II ns	Latency ns	Energy nJ	Cost	FF	LUT	DSP	AIE	Resource%	Area%
8-bit Implementation										

AIE	35.2	283.2	265.37	685	/	/	/	15	3.75	0.675
VNN	224	420	1717.60	95	1044	7469	0	/	0.83	0.556
16-bit Implementation										
AIE	51.2	216.8	385.99	685	/	/	/	15	3.75	0.675
VNN	252.16	472.8	1957.01	420	3307	5982	72	/	3.66	2.45

Table 5. 2-D-Stride model metrics data

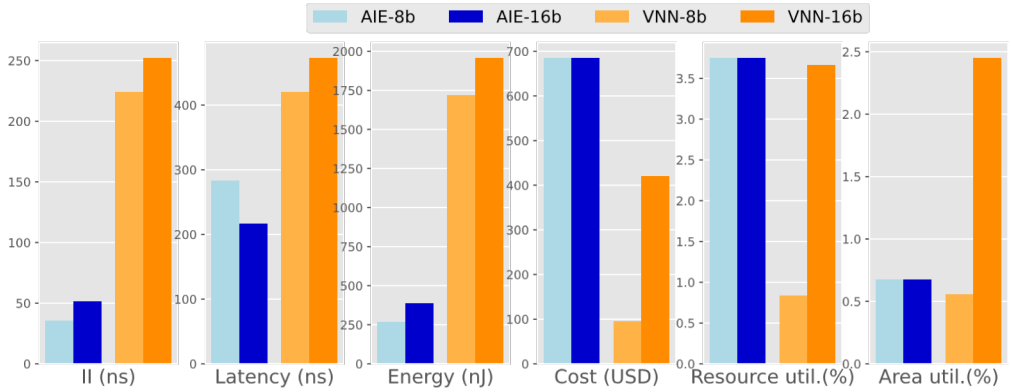


Figure 60. Key metrics of 2-D stride AIE and VNN comparison

For the 2D stride model, since the VNN implementation targets resource saving and sacrifices the II/latency, which is different from the AIE implementation design goal (minimum II and latency with moderate resource utilization), the comparison is less straightforward than previous benchmarks.

From the AIE 8-bit to AIE 16-bit implementation, we can observe that the latency decreased while using the same amount of AIEs. This is because of the avoidance of the pack-unpack conversion.

Secondly, in the AIE 16-bit and VNN 16-bit implementation, spending much more silicon area and using the same power results in worse II/latency on the VNN side. This supports the conclusion that AIE is more resource efficient if there are more parallel computations in the model.

8 Discussion

8.1 Initiation Interval

The initiation interval of the AI Engine implementation is worse than the VNN in our 1-D and 2-D designs. The major reasons are the bottleneck of stream data transfer and thus the nature of reuse

of AIEs. As mentioned in section 4.1, 4.3 and 4.5, other data transfer methods do not meet our need, leaving streaming as the only option (cascade data transfer cannot merge data because each AIE has only 1 cascade port, window data transfer adds at least 12ns per inference because of program loading overhead). There is a maximum of two 32-bit stream ports for read or write accordingly, giving the AIE a total capability of 64 bits per cycle for data transfer.

To achieve a small initiation interval, a highly pipelined design is desirable. This means each AIE performs a small amount of computation (e.g. 128-MAC8) quickly and passes the data to another AIE. However, this idea faces the multi-join problem mentioned in section 4.5 if we try to break down a specific layer and thus leads to a high latency overhead.

Another consideration is resource utilization. Take as an example our 2-D model. If we force each AIE to complete its computation in one cycle, then we would need 24 AIEs for the Conv layer, at least 32*4 AIEs for Relu layer (unavoidable pack-unpack), 8 AIEs for Dense layer and 10 AIEs for Softmax layer. This would require 170 out of 400 AIEs, resulting in a utilization percentage of 42.5% compared to 0.4% for the VNN.

Additionally, consider the minimum II, some operations naturally require more than one cycle in the AIE, as shown in the table below:

operation	load vector of x bits	data stream of x bits	data type conversion	loop
cycles	x/256	x/64	6	6

Table 6. multi-cycle operations

The load operation requires the use of the 256-bit read port of the AIE, loading any vector longer than 256-bit would require more than one cycle. The streaming data transfer is also limited by the port width as we only have two 32-bit ports per AIE. The data type conversion from accumulator to vector requires going through the MAC or shift-round-saturate data path, and they are six-stage

pipelined. The looping in the AIE requires at least 6 cycles because from the assembly observation we found it has a branch delay slot of 5. Any loop body that compiles to less than 6 lines of assembly would still require 6 cycles for one iteration. Those set a narrow limitation to AIEs if we still want to achieve minimum II.

8.2 Latency

The latency of the AIE is also not competitive among the 1-D and 2-D models. This is primarily because data transfer across AIEs introduces extra overhead, whereas the FPGA transfers the data with programmable interconnection that completes the transfer within one cycle and can be configured to any arbitrary bit width.

The second factor is the synchronization across AIE cores. In our streaming approach, if an AIE does not receive the stream input, it will stall and wait. This stall then propagates to the next stage and continues to the last stage. If a particular AIE performs slow computations, the entire design incurs a cumulative latency penalty.

To resolve the slow AIE core issue, we use multiple AIEs to perform the original work. This can result in either a reduction tree that adds latency, or a sequential round-robin pattern that adds latency. The VNN does not have this problem because it has a versatile interconnect network and versatile logic to perform the reduction or accumulation of the partial result.

8.3 Resource Utilization

The resource use percentage of AIE is surprisingly higher than expected, and here are several possible reasons: First, the vector data path bit width is set to power of two. If our data falls between two values, we have to take the upper bound. This includes our data element bit width. For example if we had a 9-bit number then we will have to use the 16-bit data type for AIE. Our data element

count is also included in this scenario, as we use a 10-element 8-bit input for our 1-D model, then when we do the vector programming it is converted to a 16-element 8-bit vector.

Second, sometimes the mapping of arbitrary computation to the vector MAC operation is inefficient. Our 2-D conv has 2 kernels, it cannot be vectorized channel-wise because the channel count is too small. The remaining row-based mapping is only doing 24 MAC operations using a 128-MAC unit, wasting 81% of the computation unit.

Finally, most of the AIE in our design are only partly used. An AIE consists of a scalar unit, vector unit, floating point unit, data transfer interconnection and memory. Below is a table showing our kernels' detailed usage of these resources. Though there is possibility to merge the usage of standalone units onto single AIE with the help of VLIW, it is not as flexible as the FPGA fabric, as we have to make sure the data I/O throughput is still sufficient, there are no resource conflicts, and the compiler can still schedule the workload in the correct way.

Kernel	Scalar unit	Vector unit	Floating point unit	Interconnection	Memory
Dense		✓		✓	✓
Conv		✓		✓	✓
ReLU		✓		✓	✓
Sigmoid	✓			✓	✓
Reduction tree		✓		✓	

Table7. Kernel AIE resource usage

The resource analysis also helped to explain the high usage of die area, high power usage and large cost of the AIE implementation, since they are all proportional to the AIE amounts used.

8.4 Limitations

The result of our evaluation is valid for our 1-D and 2-D model, and the limitation is introduced by some special characteristics of them. First, our Conv layer and the Dense layer have specific shapes and are lightweight, the two-kernel conv prevents us from channel-wise vectorization, and the 32-element narrow vector-vector MAC cannot be fed to the MAC data path directly. This leads to a low utilization of the MAC unit resource. If there are at least 8-kernel conv, or a 8*8 shape dense input, then we can use the entire MAC unit and the AIE's performance could be better.

Second, the AIE's performance could also be worse if we choose to quantize the workload to a non-power of 2 bit width. For bit widths in the range of 1~8 we use int8, and for bit widths in range of 9~16 we use int16. The FPGA can use the exact resources required by a particular bit width. Thus, in this scenario, the FPGA resource utilization and power efficiency can be better.

8.5 Adapting MLPerf Tiny Metrics

The MLPerf Tiny Benchmark [Banbury21] is a benchmark suite for low-power tiny machine learning systems. It serves as a unified comparison of various machine learning systems' efficiency. The performance metrics previously presented in this thesis include the data from three benchmarks and are adapted to align with the MLPerf Tiny Benchmark. While this adaption facilitates comparison and supports researchers interested in this standard, the measurement procedures in this work do not adhere strictly to the MLPerf Tiny methodology. This section aims to provide a complementary perspective, offering results in a familiar framework for easier interpretation.

	1-D		2-D		2-D stride	
Data type	8b	16b	8b	16b	8b	16b
Latency (us)	0.173	0.125	0.476	0.545	0.283	0.217
Energy (uJ)	0.060	0.048	0.321	0.471	0.265	0.386

Table 8. AIE MLPerf Tiny Metrics

8.6 Possible Enhancement

Currently, the next generation of AIE is available, called the AIE-ML. There are some architectural enhancements that could possibly make the AIE-ML more competitive than the fabric. One direct upgrade is the MAC operation capability gets doubled, for example 256 of int8 MAC per cycle is now 512-MAC.

To help with the reduction scenario, AIE-ML introduced a deterministic merge. If the II/Latency of the merge is better than the current reduction tree solution, the entire II, latency and utilization of the model could be better. Furthermore, even if the deterministic merge is unavailable, the AIE-ML also introduced one more cascade port (in contrast to only one port per AIE). This will help the original reduction tree have better bandwidth (32-bit stream port to 512-bit cascade port).

Specifically with the non-linear layer, we use a lookup table (LUT) to avoid complex computation, and we have to iterate sequentially for 10 times to complete one softmax layer. The parallel lookup introduced by the AIE-ML supports getting data from the LUT in parallel, by storing multiple copies of the LUT in different banks of the AIE-ML local memory. This will directly boost the performance of our 2-D softmax layer.

9 Conclusion

In this thesis, the efficiency of Versal AI Engine with respect to machine learning workload is evaluated. With the design space exploration performed in the first benchmark, the AI Engine characteristics are summarized to help consider model performance. We concluded the window data transfer and cascade data transfer cannot meet certain requirements for mapping machine

learning models and the AIE has a limitation of data arrangement internally that prevents the user from implementing high-performance convolution layers in certain shapes.

Furthermore, with the comparison of AIE and FPGA implementation of our three benchmarks, we concluded that AI Engine is less desirable in terms of II and latency for small (<5% resource utilization) neural networks with ultra-fast requirement (<10 FPGA cycles). The AIE silicon area efficiency for higher bit width and parallel computations is also demonstrated. Finally, the limitation of AIE and possibility of having better efficiency with next-generation AIE (AIE-ML) is discussed.

Bibliography

[Banbury21] Colby Banbury, Vijay-Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau et al. Mlperf tiny benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[Brown23] Nick Brown. Exploring the versal ai engines for accelerating stencil-based atmospheric advection simulation. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 91-97. 2023.

[Chatarasi20] Prasanth Chatarasi, Stephen Neuendorffer, Samuel Bayliss, Kees Vissers, and Vivek Sarkar. Vyasa: a high-performance vectorizing compiler for tensor convolutions on the xilinx ai engine. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1-10. IEEE, 2020.

[Chen23] Paul Chen, Pavan Manjunath, Sasindu Wijeratne, Bingyi Zhang, and Viktor Prasanna. Exploiting on-chip heterogeneity of versal architecture for gnn inference acceleration. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 219-227. IEEE, 2023.

[CERN17] CMS collaboration et al. The phase-2 upgrade of the cms endcap calorimeter. 2017.

[Deng12] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141-142, 2012.

[Gaide19] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx adaptive compute acceleration platform: versal architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 84-93. 2019.

[Han15] Song Han,Huizi Mao, and William-J Dally. Deep compression: compressing deep neural networks with pruning, trained quantization and human coding. *ArXiv preprint arXiv:1510.00149*, 2015.

[Heinz24] Carsten Heinz, Torben Kalkhof, Yannick Lavan, and Andreas Koch. Tapas co-aie: an open-source framework for streaming-based heterogeneous acceleration using amd ai engines. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 155-161. IEEE, 2024.

[Ho23] Rui-En Ho. Onnx2versal. https://github.com/rehohoho/onnx2versal, 2023.

[Jia24] Xijie Jia, Yu Zhang, Guangdong Liu, Xinlin Yang, Tianyu Zhang, Jia Zheng, Dongdong Xu, Zhuohuan Liu, Mengke Liu, Xiaoyang Yan et al. Xvdpu: a high-performance cnn accelerator on the versal platform powered by the ai engine. *ACM Transactions on Reconfigurable Technology and Systems*, 17(2):1-24, 2024.

[Johnson23] Caroline Johnson. Evaluating the quality of hls4ml's basic neural network implementations on fpgas. Master's thesis, University of Washington, 2023.

[Lei24] Jie Lei and Enrique-S Quintana-Ort. Mapping parallel matrix multiplication in gotoblas2 to the amd versal acap for deep learning. *ArXiv preprint arXiv:2404.15043*, 2024.

[Reyes-Ortiz13] Anguita, Davide, Ghio, Alessandro, Oneto, Luca, Reyes-Ortiz, Jorge and Xavier Parra. Human Activity Recognition Using Smartphones. UCI Machine Learning Repository, 2013. DOI: https://doi.org/10.24432/C54S4K.

[Singh23] Gagandeep Singh, Alireza Khodamoradi, Kristof Denolf, Jack Lo, Juan Gómez-Luna, Joseph Melber, Andra Bisca, Henk Corporaal, and Onur Mutlu. Sparta: spatial acceleration for efficient and scalable horizontal diffusion weather stencil computation. In *Proceedings of the 37th International Conference on Supercomputing*, pages 463-476. 2023.

[Taka23] Endri Taka, Aman Arora, Kai-Chiang Wu, and Diana Marculescu. Maxeva: maximizing the efficiency of matrix multiplication on versal ai engine. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 96-105. IEEE, 2023.

[Weng23] Olivia Weng. Ecoder. https://github.com/oliviaweng/fastml-science/tree/quantizedautoencoder/sensor-data-compression, 2023.

[Xilinx22] Xilinx. Ai engine intrinsics user guide. https://www.xilinx.com/htmldocs/xilinx2022_2/aiengine_intrinsics/intrinsics, 2022.

[Xilinx22] Xilinx. Ai engines and their applications (wp506). https://docs.amd.com/v/u/en-US/wp506-aiengine, 2022.

[Zhuang23] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu et al. Charm: composing heterogeneous accelerators for matrix multiply on versal acap architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 153-164. 2023.