Making AIE Development Easier: Documentation and Examples for Programming on AMD AI Engine

GRAM KOSKI, University of California - San Diego, USA
FRANCISO GUTIERREZ, University of California - San Diego, USA

Recent advances in computer accelerators for machine learning have increased the need for programmers to effectively utilize emerging hardware platforms. AMD AI Engine (AIE) is a novel memory-mapped, network-on-chip architecture for processing custom AI workloads. As a unique architecture, programmers have difficulty implementing optimized designs and fully utilizing its capabilities. We find the existing documentation to be lacking in examples and clear methods of overcoming common programming challenges. We research the space of programming on an AIE and seek to make AIE development easier by creating open-source documentation with examples, guides, and explanations for future AIE programmers.

1 Introduction

Modern machine learning workloads present high compute demands, and have motivated the development of specialty accelerators, mainly GPUs and NPUs, but also FPGAs and ASICs. The AI Engine architecture is optimized for deterministic, low-latency machine learning workloads, especially in embedded and real-time systems. Unlike GPUs, which rely on massive parallelism and dynamic scheduling, the AIE family can issue deterministic instructions for computation and data movement. This fine control in the hardware makes them ideal for edge inference tasks where latency, power efficiency, and tight coupling with pre/post-processing logic, like FPGA fabric, are critical. While AIEs won't replace GPUs for general-purpose deep learning because of the massive parallelism required for such training and the mature development space surrounding ML for GPUs, AIEs can still perform well in embedded ML pipelines requiring tight time constraints and low power.

The AI Engine is built around a scalable, two-dimensional array of highly optimized processor tiles, each integrating a dedicated AI Engine core, local memory, and a programmable interconnect module. This architecture enables significant parallelism at multiple levels, including single-instruction multiple-data (SIMD) operations, very long instruction word (VLIW) execution, and multicore computation across the array. Local memory modules are shared between neighboring tiles, supporting efficient data exchange and minimizing external memory bottlenecks. The programmable AXI4-Stream crossbar switch in each tile facilitates low-latency data streaming, supporting both circuit and packet-switched communication. This tightly coupled network-on-chip design, combined with the deterministic data flow graph programming model, allows the AIE to achieve many different types of parallelism at once. Programmers are given a challenge to effectively exploit this parallelism by navigating around the difficulties and complexity of programming on AIE.

Programming on the AI Engine introduces two central challenges that shape the way developers must approach performance optimization and design.

(1) **Limited Streaming Bandwidth and Tile Placement:** Limited streaming bandwidth between tiles makes the physical placement of kernels a critical factor. Since the architecture is designed around a network of interconnected tiles, data transfers between non-adjacent tiles can quickly become bottlenecked by network congestion and the relatively small AXI4 stream width of 32 bits. As a result, programmers are compelled to carefully co-locate tightly coupled kernels in neighboring tiles and minimize the distance that data must travel often done through *AIE Vitis Constraints*. This often requires thoughtful partitioning of algorithms and a strategic approach to mapping workloads, ensuring that the most communication-intensive operations occur between adjacent tiles to maximize throughput.

Authors' Contact Information: Gram Koski, University of California - San Diego, USA; Franciso Gutierrez, University of California - San Diego, USA.

(2) **Local Memory Constraints:** Second, the severe memory constraints on each local tile further complicate efficient programming. With only a small amount of dedicated memory per tile (32KB), developers must fit code, data, and input/output buffers into a very limited space. Programmers are often forced to break large datasets into smaller blocks, carefully schedule memory usage, or offloading strategies to external memory to avoid stalls and under-utilization of compute resources. These challenges demand an understanding of the AIE's architectural nuances and a deliberate placement of buffers data sharing between neighboring kernels.

In the *Making AIE Development Easier* documentation, we seek to address these two challenges and provide practical examples which satisfy those programmers who are beginners in programming the AIE. Specifically, our documentation includes a Github repo [11] and a ReadtheDocs page [12] with code examples, architectural explanations, and ways to tackle common AIE challenges. We start with the most basic examples for the new AIE programmer. Our code suite and explanations are separated into modules. While we will not discuss in depth every module in our documentation in the following report, because we choose to focus only on interesting attributes of certain modules which address challenges 1 and 2 detailed above. The full module list and specification of our final documentation can be found in *Section 4: Milestones* and *Section 5: Conclusion*. In this report, we discuss the central two challenges of programming on the AIE. In order to do this, we give a technical deep-dive into the following:

- **AI Engine Architecture:** Introduce architectural concepts in-depth from a programmer's perspective.
- **Data Movement**
  - **Using Constraints:** Details how to explicitly map AIE kernels for considering data movement. Since data movement is one of the major challenges, we believe it should be introduced early.
  - **Vector Addition:** Vector addition on a single tile is the first computational example and is used to discuss in more depth the difference between buffers and streams for data movement.
- **Memory Limitations**
  - **Simple Multi-Tile Matrix Multiplication:** Implementation of simple multi-tile matrix multiplication using the built-in API function. This is the first truly multi-tiled program our documentation discusses.
  - **Adder-Tree Matrix Multiplication:** We explain a more efficient multi-tiled matrix multiplication program which serves to show how good parallel algorithms work around the constraints of the AIE.

Our code suite targets the *Versal VCK190* board which has 400 processors in its AIE array. This is on the upper end in terms of AIE compute power, but we believe our examples are sufficiently general to give insight for programmers targeting alternative architectures. We use the AMD Vitis tool flow and compiler, because its the most popular tool flow for programming on the AIE with a larger community and robust software.

2 Related Works

There exists numerous sources which give information about programming on the AI Engine. There is the official guide *Graph and Programming Guides* published by AMD and Xilinx [8][4], and specialty documents which provide information about intrinsic programming [3] and the AIE API [1]. While these documents provide a useful source of raw, accurate information, they do not include many easy examples or provide a useful place for programmers to start. We relied on the Versal architecture documentation [6] to target our specific *Versal VCK190* board, and the AMD Vitis Tool Flow documentation [2] for working with the Vitis compiler and software platform.

Other documents provide examples and educational material for programming on the AI Engine. The *Xilinx University Program* published a sequence of tutorials for programming on the AI Engine [10] (now *AMD University*

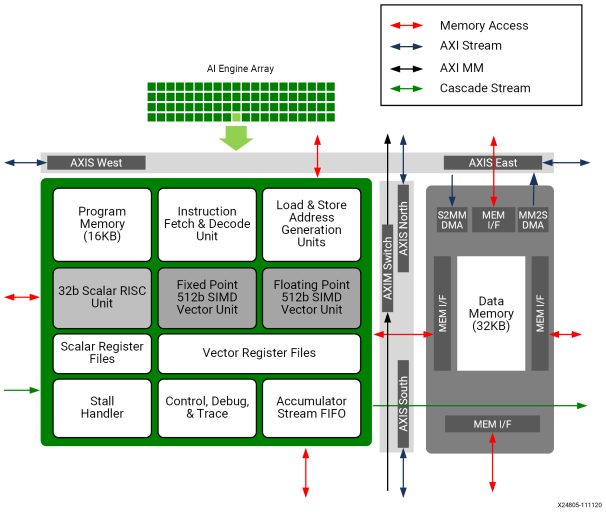


Fig. 1. AIE Tile [7]

Program AI Engine Tutorial), but it uses Vitis 2021 outdated software and outdated programmatic functions/classes. AMD published *Vitis Tutorials: AI Engine Development* [9] which includes many examples but jumps very quickly to complex applications, with limited basic examples.

We also took inspiration from the last section of *Parallel Programming for FPGAs: Projects and Labs* [16] which walks through an implementation of multi-tile matrix multiplication. This tutorial uses the MLIR-AIE tool flow, while we use the Vitis tool flow. We sought to structure our documents somewhat similar to the projects and labs detailed in Prof. Kastner's *CSE160 - Introduction to Parallel Computing* [13]. While the course focused on GPU programming not AIE, we wanted to replicated the educational manner in which the material is presented. The user is introduced to ever more complex examples as they read the document and develop familiarity with the material.

Lastly, for our matrix multiplication programming examples, we take inspiration from *MaxEva* [17] for educational purposes, using an addition tree design to enable better scaling across tiles. Additionally, there exist numerous other documents we encountered which the authors seek to optimize matrix multiplication on the AI Engine, which we use slightly different designs, mostly targeting the AIE-ML architecture. [18][15][14]

3 Technical Material

3.1 AI Engine Architecture

The AI Engine is composed of two primary components: memory and processing cores, both integrated within an AIE tile. Each tile serves as a fundamental unit of the AI Engine, capable of independently executing kernels. What sets the AI Engine apart is its distinctive tiled architecture. Developers have the flexibility to assign multiple kernels to a single tile or to create sequential execution by passing data from one tile to the next. This architectural design supports both parallel execution and efficient data sharing, making it well-suited for high-performance, parallel applications.

To leverage these architectural features, developers utilize the Vitis or AMD AI Engine toolchains, which include compilers and placement tools to determine how kernels and data paths are synthesized onto the hardware. This tool-driven workflow is deterministic, enabling the Vitis IDE to emulate hardware behavior and offer

comprehensive debugging capabilities. While these tools provide helpful abstractions, optimal placement and data movement may require manual intervention. Developers seeking greater performance may need to exert finer control over the synthesized design by adjusting data types, applying explicit constraints, or modifying the graph code to better suit the application's requirements.

3.2 Data Movement Patterns:

3.2.1 Using Constraints. The AI Engine offers multiple layers of programmability, enabling developers to tailor their designs to specific application needs. At the graph level, the Graph API allows programmers to select data types that are best suited for synthesis, while the Streaming API provides control over kernel streaming behavior and data access patterns. One of the most powerful features is the use of Adaptive Data Flow Graph (ADF) constraints, which enable precise specification of spatial relationships and resource allocation within the architecture [7]. With these specialized capabilities, developers can modify aspects such as buffer types, kernel placement, tile assignment, memory bank selection, and memory allocation directly within the graph code. While higher-level APIs offer broad control over the dataflow graph, these options provide granular, low-level control, allowing programmers to directly observe and influence the configuration and layout of the AI Engine's tile array.

In our examples we make use of these feature. We use `single_buffer()` to use less memory, `location<buffer>()=bank()` to specify location of a buffer within a bank,`location<stack>()` to modify the location of the kernel's stack, and `location<kernel>()=tile()` to affix our kernels to specific tiles.

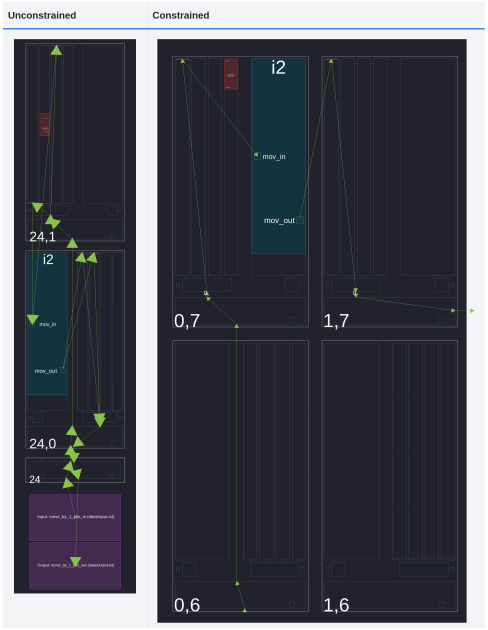


Fig. 2. Constraints Comparison for an adder. Note how the compiler-automated kernel placement (unconstrained) on the left side appears less organized than the constrained version on the right. This can become an issue in highly complex programs. Courtesy of [2]

These constraints are inevitable to achieve the best occupancy and throughput in more complicated multi-tiled programs. For example, the AXI4 streaming network is capable of enabling direct communication between any

tiles in the AIE array. This is a key feature of the AI Engine, but the streaming network is actually better off avoided if programmer a wants to achieve maximum throughput. The stream has a low bit-width of only around 64 bits, and network overcrowding can result in a tangled mess of locks and stalls. Instead, it is ideal to place kernels that frequently exchange data into adjacent tiles so they can directly read and write to neighboring tile's memory. This makes the bandwidth practically infinite between those tiles. The challenge is that it can get rather complicated when there are multiple tiles all communicating with each other. It is often the programmer's job to find an ideal kernel placement.

This can be seen in our *Matrix Multiplication with Adder-Tree* module where the multiplication tiles sit adjacent to the addition tile so that the kernels can write the partial computations directly into the next layer of the tree without having to use the stream. More details of the Addition Tree are discussed in Section 3.3.3.

In the MaxEva [17] implementation, they similarly make use of this feature of intentional tile placement and the assigned kernels for each tile, minimizing the need for streaming and maximizing the throughput of their kernels while giving sequential kernels no starvation.

3.2.2 Buffer vs. Stream.

The *Vector Addition* module includes two different implementations of single-tile vector add. The first uses `aie::stream` and the second example uses `aie::buffer`. Each AI Engine tile has two 64 bit PLIO streams and using both at the same time gives the stream a width of 128 bits. A kernel can therefore receive four `int32` from PLIO at a time. Each instance the kernel receives a new packet of data from PLIO, it performs the computation, adding the four *int32* values in parallel then writing to the output. Therefore, we must run the kernel 256 times to process an entire addition of two int32 vectors of length 1024. Since the output is written every iteration of the kernel, we can expect a lower latency using stream because the output is constantly being written.

When using `aie::buffer`, the kernel is suspended until the entire input buffer is full before initiating its computation. In this case, we are free to use a large SIMD vector iterator of 512 bits to iterate and compute the addition of two vectors, 16 elements at a time. The kernel is run just once and processes the entire input sequence before writing to output. Therefore, buffer gives slightly lower overall runtime in most applications due to kernel function call overhead and more efficient SIMD calculations. At the same time, it has higher latency than stream.

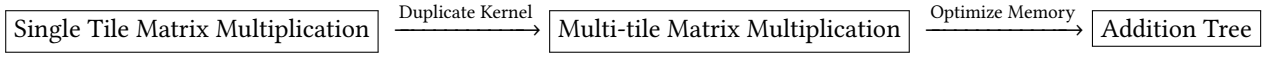
In our *Vector Addition* module, we use Vitis Hardware Emulation to compare the implementations. The stream implementation has very low latency (~200ps), while the buffer's latency is over 50% over the total runtime at ~4430ns of ~7000ns. The buffer was slightly faster by about ~1000ns. The tradeoff is summarized:

Table 1. Comparison of Buffer and Stream

Metric	Buffer	Stream
Latency	Higher (wait for buffer fill)	Lower (immediate processing)
Throughput	Higher (More efficient use of SIMD)	Lower (128 bits per kernel iteration)
Best Use Case	Large datasets, random access	Real-time, low-latency apps

3.3 Matrix Multiplication and Memory Limitations:

The matrix multiplication code suite and accompanying documentation is presented in the following pattern in our ReadtheDocs. Each module is addressed to solve a challenge introduced in the last module. We believe this is a good structure for learning increasingly complex programs.



3.3.1 Single Tile Matrix Multiplication: The AIE API [1] is a C++ library which provides a convenient set of functions and classes for programming kernels. In our documentation, we implement a matrix multiplication kernel to demonstrate a use-case of the API. This program is explained in the *Single-Tile Matrix Multiplication* section of our Readthedocs. We use the int16 datatype and the built-in AIE API class `aie::mmul<M, K, N, int16, int16>` [5]. The class is declared with the size of the multiplication (MxKxN) and a data type. The *mmul::mat()* function writes the result of A*B where A is MxK and B is KxN to an accumulator register. You can optionally declare a desired accumulator precision. The default precision can be found in the docs by TypeAxTypeB multiplication.

This built-in class is convenient, but larger values of M, K, and N are not offered because it is limited by the number of lanes in the accumulator register. With 4x4x4 int16 matrix multiplication, we must store 16 int48 values in the accumulator (if we are using the default int48 precision). This means we are using the entire 768 bit width of the accumulator, so we cannot make the output matrix dimensions (MxN) any larger.

The solution to this problem is to do matrix multiplication in blocks and accumulate partial results. The second kernel included in the *Single Tile Matrix Multiplication* module is the `matmul_4x16x4` kernel. This kernel calculates AB = C where A is 4x16, B = 16x4, and C = 4x4 for the int16 datatype. The program expects the input matrices to be vectorized in a very specific format. The matrices are split into 4x4 blocks so that the `aie::mmul` library may be utilized. Within a 4x4 block, both A and B matrices are stored in row-major format. At the block level, the A matrix is stored in block-wise row major format and the B matrix is stored in block-wise column major format.

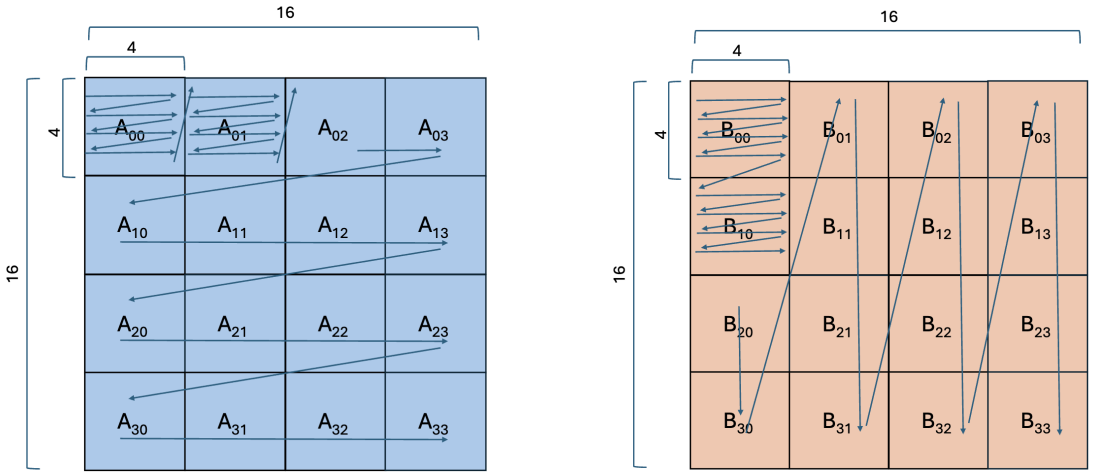
We loop through the sub-blocks of the matrices to compute AB. In the first iteration, we use `aie::mac()` to initialize the accumulator with the correct values of A_0B_0. On subsequent iterations we use `aie::mul()` to compute $acc \rightarrow acc + A_iB_i$ where acc is the running sum of the partial results of the matrix multiplication computations.

3.3.2 Simple Multi-Tile Matrix Multiplication: We use our `mmul_4x16x4` kernel in our *Simple Multi-Tile Matrix Multiplication* module. In this module, we demonstrate the computation of 16x16x16 matrix multiplication for int16 datatype across 16 tiles of the AI Engine array.

Each tile calculates a 4x4 section of the output matrix by working off of one block-wise row of A and one block-wise column of B. The kernel uses an `aie::port<>` scalar parameter to calculate which section of A and B in local memory it should be processing. For example the upper left 4x4 corner of the output matrix C, called C_{00} is calculated by doing `mmul_4x16x4` on the first block-wise row of A ($A_{00} \rightarrow A_{03}$) and one block-wise column of B ($B_{00} \rightarrow B_{30}$)

The *Simple Multi-Tile Matrix Multiplication* module serves as the user's introduction to multi-tiled computation on the AI Engine. As such, the data movement and program complexity is relatively low. The A and B matrices are duplicated 16 times in the local memory of each tile involved, that way the tiles can do each computation separately. The kernels are placed onto a 4x4 grid in the AI Array. This arrangement is ideal for illustrating the simple example, and also because the 4x4 configuration is used in the AIE Ryzen devices so our example program is generalizable to many different AIE hardware versions.

One problem with this approach is clear. Duplicating the data 16 times is a highly inefficient use of the scarce local tile memory. Since there is only 32KB of RAM in each tile, we must be very careful about how we store the matrices in memory when we scale up the size of our computations. Recall that limited local tile memory is the second central challenge of programming on the AI Engine. While, we can technically increase the efficiency of our current implementation, it is still necessary for us to move the entire row of A and column of B into a tile's locally memory. The most efficient way we could configure the memory would still result in 4 times duplication, because we need to examine each row/column 4 times by nature of our 4x4 tiling. Therefore we demonstrate a



(a) A Matrix

(b) B Matrix

Fig. 3. The tiling scheme for the A and B matrices for *Simple Multi-Tile Matrix Multiplication* module.

different approach for multi-tiled matrix multiplication which is not subject to the same memory bottleneck in the *Matrix Multiplication with Adder-Tree* module.

3.3.3 Matrix Multiplication with Adder-Tree: The module demonstrates how to decompose a large summation into a sequence of pairwise additions performed across multiple tiles. Each tile is responsible for summing a subset of the input data, and the partial results are then forwarded to subsequent addition tiles for further accumulation. It enables us to split the columns of A and rows of B onto different tiles. This split is what allows us to not have to duplicate memory.

Our *Making AIE Development Easier* documentation explains in depth a 4x128x128 matrix multiplication using 4 tiles similar to the schema in Figure 4 except the sub-matrices are 4x32 rectangles not perfect squares. In our source code, we also include a larger 4x784x128 example. It can be seen in *Section 3.2: Data Movement and Patterns* that within the addition tree, the placement of kernels and the routing of data streams are carefully orchestrated to maximize throughput. This pattern highlights the importance of thoughtful kernel placement and data movement strategies in order overcome the challenges of limited memory on the AI Engine.

The addition tree is used again in the *MNIST Multi-Layer Perceptron (MLP)* module to perform the dense layer computations. The MLP uses quantized int16 weights with an input dimension of 784, 3 hidden layers, and output dimension of 10. The inference batch size is 4, and the activations are simple RELU. The final activation is *argmax*. The MLP module is the final, most complex example in our documentation and remains unoptimized. Perhaps a direction for future work.

4 Milestones

4.1 Start of Quarter

At the onset of the quarter, we listed milestones for each week. In hindsight, some of the milestones were not ideal, because they included some subjective criteria such as "Think about data flow through NN for the future." We list the more objective milestones in order to illustrate how our project goals and timeline has changed.

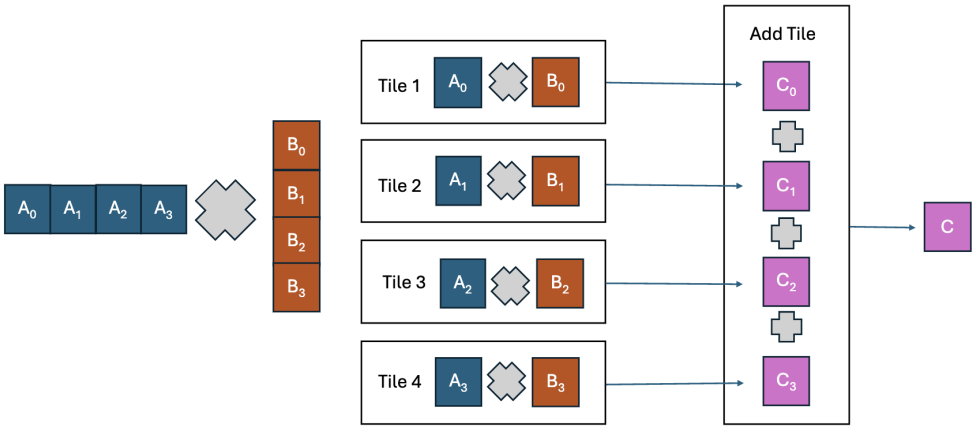


Fig. 4. Illustration of the addition tree structure for parallel summation across AI Engine tiles. Note that our documentation includes a 4x128x128 matrix multiplication which is almost identical to the figure except that the sub matrices A_{xy}, B_{xy} are not perfect squares but rather rectangles.

Milestone	Original Intended Completion	Actual Completion
Basic vector addition kernel written and tested	Week 3	Week 4
Matrix multiplication implemented and benchmarked	Week 4	Week 5
Skinny matrix multiplication and sparse matrix multiplication completed	Week 5	Week 7
Convolution module and initial neural network implementation	Week 6	Week 9
Neural network implementation completed	Week 7	Week 10

Table 2. Comparison of Project Milestone Timeline: Intended vs. Actual Completion

As you can see, we generally lagged behind our intended completion date by two weeks on average due to various challenges and setbacks. Also, note that we removed convolution as a necessary milestone at the mid-quarter update, but we were still able to complete it thanks to Shreeyash Pacharne.

4.2 Mid-Quarter Changes

(1) **New Module: Comparisons to GPU Programming.** One major piece of feedback across the student reviews of our milestone update was some confusion on the specific novelty of the AIE architecture. Some students suggested that we provide a comparison with GPUs to foster understanding by relating AIE to familiar architecture, and others expressed confusion about the motivations of developing on an NPU. In response to this, we added a module comparing AIE programming to GPU programming. This module is titled *AIE vs Other Accelerators* in our Readthedocs.

(2) **De-emphasis on Convolution Operations.** The convolution module is removed from our core reference guide, as AIE does not natively support this operation. Our focus will remain on matrix multiplication and MLP examples. A convolution module may be added if time permits.

(3) **MLP Neural Network.** At the beginning of the quarter we said we'd do a full CNN. Instead of a CNN, decided to implement an MLP for MNIST using our skinny matrix multiplications for dense layers (adder-tree design) and ReLU activations. This decision was made to simplify our code suite and enabled us to use the matrix multiplication design from our addition tree module.

In the mid-quarter update, we decided to give a one-to-one correspondence between our milestones and visible MVP features. These MVP features are code and accompanying documentation that is featured on our Github and ReadTheDocs.

4.3 MVP Feature Status and Milestones

Note that each feature is a code suite (featured on Github) and accompanying documentation (featured on ReadtheDocs). A checkmark means that both of these features are complete.

Feature	Mid-Quarter Status	End of Quarter
Introduction and Architecture	✓	✓
Comparisons to GPU Programming	–	✓
Vector Addition: Stream	✓	✓
Vector Addition: Buffer	✓	✓
Matrix Multiplication: Single-tile	✓	✓
Matrix Multiplication: Multi-tile	✓	✓
Matrix Multiplication: Skinny (adder-tree)	–	✓
MLP Neural Network: PyTorch Model Training	✓	✓
MLP Neural Network: Dense Layers	–	✓
MLP Neural Network: Input/Output Formatting	×	–

4.4 End of Quarter Changes and Challenges

We didn't change much about our milestones or timeline from the mid-quarter update to the end of the quarter. That being said we did make a few changes the precise specifications of the milestones. We renamed the *Comparison between GPU and AIE* module to *AIE vs Other Accelerators* to reflect a more general architectural perspective. In this module, we compare GPUs, AIE, and other NPUs. Also, despite de-prioritizing convolution, we included a simple convolution module although the documentation for it remains sparse.

The largest challenge which set back our timeline in the beginning of the quarter was gaining general aptitude with the Vitis Software Platform. We had trouble compiling our programs and there were a few particular issues that set the team back for a week or two. For example, there was a naming conflict between our kernel name and an internal Vitis API function in our vector addition code that we were not able to catch for longer than it otherwise should have.

The last challenge we've encountered is with formatting the MLP inference program. There is an error in converting the int48 default vector types back to int16 in the case where the int48 values exceed the int16 range. We do not get the behavior we expect. Right now, it rounds these values back to zero, affecting the correctness of the MLP inference. The MLP module remains with sparse documentation due to the ongoing work. The convolution module also includes sparse documentation due to its deprioritization at the mid-quarter update.

5 Conclusion

In Summary, we complete code suites and accompanying documentation for the following:

- **AIE vs Other Accelerators:** We introduce the popular architectures for machine learning workloads and compare them to the AI Engine. This section may be particularly useful for those programmers familiar with GPUs.
- **Setup and Architecture Explanation:** Discussion of the history of the AI Engine, devices with AIE technology, and the important difference between AIE, AIE-ML, and AIE-MLv2.
- **Using Constraints:** Serves as a first example for mapping AIE kernels and considering data movement. Since data movement is one of the major challenges, we believe it should be introduced early.
- **Vector Addition:** The application of vector addition on a single tile is the first computational example and used to talk more in depth about the difference between buffers and streams for data movement.
- **Single Tile Convolution:** Implementation of simple 2D convolution as a programmatic reference.
- **Single Tile Matrix Multiplication:** Implementation of simple matrix multiplication using the built-in API function.
- **Simple Multi-tile Matrix Multiplication:** The first truly multi-tiled program the tutorial discusses. We use the kernel built from the previous example and also discuss its inefficient use of local tile memory.
- **Matrix Multiplication with Addition Tree:** A second, more efficient multi-tiled matrix multiplication program. Serves as a good way of showing the user how good parallel algorithms work around the constraints of the AIE. The addition tree enable the matrices to be split up so that duplication of data is no longer an issue.
- **Simple MLP on MNIST:** Lastly, using the addition tree multiplication, we provide an implementation of a basic neural network. It is used to demonstrate techniques like graph embeddings (graphs within graphs) for building more complex programs.

This work provides open-source documentation and practical examples to address central challenges in programming the AMD AI Engine, mainly managing data movement and local memory constraints, ultimately lowering the barrier for new users and enabling more efficient development of AI workloads on this novel architecture. Our contributions include modular code suites, architectural explanations, and comparative guides that fill gaps left by existing resources. Future work could focus on optimizing the implementation and accuracy of our MLP module, expanding coverage to additional AI Engine hardware variants, or simply cleaning and writing better documentation for our existing code.

Acknowledgments

We thank Zhenghua Ma, Aba Gnaneswaran, Ryan Kastner, and the UC San Diego Kastner Lab for their support throughout the development of this project.

References

[1] AMD. 2024. *AI Engine API User Guide (AIE-API) 2024.2.* Advanced Micro Devices, Inc. Retrieved 2025-06-12 from https://download.amd.com/docnav/aiengine/xilinx2024_2/aiengine_api/aie_api/doc/index.html Version 2024.2.

[2] AMD. 2024. *AI Engine Environment User Guide (UG1076).* Advanced Micro Devices, Inc. Retrieved 2025-06-12 from https://docs.amd.com/r/en-US/ug1076-ai-engine-environment/Overview Document Number: UG1076, Version 2024.2.

[3] AMD. 2024. *AI Engine Intrinsics User Guide (UG1078) v2024.2.* Advanced Micro Devices, Inc. Retrieved 2025-06-12 from https://download.amd.com/docnav/aiengine/xilinx2024_2/aiengine_intrinsics/intrinsics/index.html Version 2024.2.

[4] AMD. 2024. *AI Engine-ML Kernel and Graph Programming Guide (UG1603).* Advanced Micro Devices, Inc. Retrieved 2025-06-12 from https://docs.amd.com/r/en-US/ug1603-ai-engine-ml-kernel-graph/Overview?tocId=U3UXOInouDrMPWWdkItJdg Overview section, Document Number: UG1603, Version 2024.2.

[5] AMD. 2024. *AI Engine ML Kernel mmul Class (UG1603).* Advanced Micro Devices, Inc. Retrieved 2025-06-12 from https://docs.amd.com/r/en-US/ug1603-ai-engine-ml-kernel-graph/Matrix-Multiplications-mmul Document Number: UG1603, Section: Matrix Multiplications (mmul).

[6] AMD. 2024. *Versal Adaptive SoC AI Engine Architecture Manual (AM009).* Advanced Micro Devices, Inc. Retrieved 2025-06-12 from https://docs.amd.com/r/en-US/am009-versal-ai-engine/ Document Number: AM009, Version 2024.2.

[7] AMD. 2025. *AI Engine Kernel Coding (UG1079).* Advanced Micro Devices, Inc. Retrieved 2025-06-9 from https://making-aie-dev-easier-readthedocs.readthedocs.io/en/latest/vs_cpu_gpu.html#ug1079 AI-Engine-Architecture-Overview.

[8] AMD. 2025. *AI Engine Kernel Coding User Guide.* Advanced Micro Devices, Inc. Retrieved 2025-06-12 from https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding/Overview?tocId=_gsmjbSrB9YCl4WkKcFCww UG1079, Version 2025.1.

[9] AMD. 2025. *Vitis Tutorials: AI Engine Development (XD100).* Advanced Micro Devices, Inc. Retrieved 2025-06-12 from https://docs.amd.com/r/en-US/Vitis-Tutorials-AI-Engine-Development/Vitis-Tutorials-AI-Engine-Development-XD100 Version 2025.1.

[10] AMD University Program. 2023. *AUP Vitis-based AI Engine Tutorial.* Advanced Micro Devices, Inc. Retrieved 2025-06-12 from https://xilinx.github.io/xup_aie_training/index.html Version targeting Vitis 2022.2, XRT 2.14.354.

[11] Francsco Gutierrez Gram Koski and Zhenghua Ma. 2025. aie4ml_project1. GitHub repository. https://github.com/zhenghuama/aie4ml_project1/tree/master Accessed: 2025-06-14.

[12] Francisco Gutierrez and Gram Koski. 2025. *Making AI Engine Development Easier.* https://making-aie-dev-easier-readthedocs.readthedocs.io/en/latest/index.html Accessed: 2025-06-14.

[13] Kastner CSE160 Course Staff. 2025. *Programming Assignment 1 - Device Query.* University of California, San Diego. Retrieved 2025-06-12 from https://docs-cse160.readthedocs.io/en/latest/PA1.html CSE160 Parallel Computing, UC San Diego, latest revision.

[14] Tristan Laan and Tiziano De Matteis. 2024. Developing a BLAS library for the AMD AI Engine. arXiv:2410.00825 [cs.DC] https://arxiv.org/abs/2410.00825

[15] Kaustubh Mhatre, Endri Taka, and Aman Arora. 2025. GAMA: High-Performance GEMM Acceleration on AMD Versal ML-Optimized AI Engines. arXiv:2504.09688 [cs.AR] https://arxiv.org/abs/2504.09688

[16] Ryan Kastner and Kastner Lab. 2025. *Project: AMD AI-Engine (AIE) Accelerator.* University of California, San Diego. Retrieved 2025-06-12 from https://pp4fpgas.readthedocs.io/en/latest/project_aie.html pp4fpgas documentation, latest revision.

[17] Endri Taka, Aman Arora, Kai-Chiang Wu, and Diana Marculescu. 2023. MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine. arXiv:2311.04980 [cs.AR] https://arxiv.org/abs/2311.04980

[18] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous Accelerators for Matrix Multiply on Versal ACAP Architecture. arXiv:2301.02359 [cs.AR] https://arxiv.org/abs/2301.02359