The Past, Present, and Future of AI Engines

Alexander Redding
Dept. of Computer Science and Engineering
University of California San Diego
alredding@ucsd.edu

ABSTRACT

Since the early 2010s, understanding how to accelerate machine learning (ML) workloads has been an active and important area of research as ML becomes increasingly embedded into our daily lives. Since 2018, the size of notable ML models has been growing exponentially due to the introduction of the Transformer architecture. These multi-gigabyte models have exacerbated the existing memory wall issue, which is that compute resources have scaled disproportional to off-chip memory bandwidth. Limited off-chip communication has become the primary bottleneck in the modern compute paradigm, and especially for performance intensive tasks like ML acceleration.

Managing the memory wall has inspired research into novel compute architectures outside of traditional multi-core CPUs and GPUs, with the goal of increasing on-chip data reuse. The reconfigurable architectures of FPGAs and CGRAs are seeing a renewed interest for high-performance computing accelerator exploration and applications. In 2020, AMD/Xilinx released the AI Engine as part of the Versal Adaptive Compute Acceleration Platform. AI Engines are a CGRA consisting of a 2D mesh of general programmable VLIW processors with SIMD vector instructions and distributed memory. Unlike the traditional cache hierarchy of a GPU, AI Engine cores can independently communicate with each other through dedicated streaming interfaces, and at deterministic latency. AI Engines are optimized for dataflow-graph style workloads. This paper surveys the AI Engine architecture; how it compares to prior and contemporary CGRAs and ML accelerators. We also look at existing AI Engine development tools, and discuss open-source alternatives. Finally, we will look at applications deployed on AI Engines.

1 INTRODUCTION

The introduction of AlexNet in 2012 marked a pivotal moment which showcased the effectiveness of hardware acceleration for deep neural network (DNN) training on a graphics processing unit (GPU) [38]. AlexNet achieved a 2D convolution speedup and pipeline parallelism between two NVIDIA GTX 580s, reducing the training time of their convolutional neural network (CNN) on the ImageNet dataset to less than a week. General purpose GPUs (GPGPUs) have since been the preferred hardware choice for machine learning acceleration across datacenters, both for training and inference. Emerging software techniques and hardware additions have enabled increasing speedups through data and tensor parallelism, and optimizations for tensor operations.

While GPGPUs are effective for increasing ML workload throughput, researchers quickly realized the limitations of streaming multiprocessor (SMP) architectures. Parallel efficiency significantly decreases when threads diverge, control logic contributes to a non-negligible amount of die space and power consumption, and non-deterministic cache hierarchies make certain realtime applications impossible [24, 35]. At the same time AlexNet was published, adjacent research was spawned with the focus of designing dedicated compute architectures for accelerating ML operations.

The first dedicated architecture for machine learning acceleration was the original Neural Processing Unit (NPU) in 2012 [17, 24]. This NPU consisted of a network of eight identical processing engines, each with dedicated multiply-add and sigmoid activation hardware, as well as data buffers and a statically configured controller. Applications such as Sobel edge detection and Fast Fourier Transforms were accelerated, with many whole-applications achieving an average 2.3x speedup and an energy savings of 3x. Since the first NPU, "neural processing unit" has become a generic term for ML accelerators.

Shortly after the development of the NPU, Google announced their first version of their tensor processing unit (TPU) which was originally deployed in 2015 for inference workloads in datacenters [35, 56]. The first version of the TPU contained a 256x256 systolic array of 8-bit integer multiply-accumulators (MACs) and a large 24 MiB local buffer. Control logic occupied a minimal amount of die space, as the TPU immediately executed CISC instructions sent from a host machine. Compared to contemporary datacenter CPUs and GPUs, the TPU was many times faster and more energy efficient.

Beyond GPUs, ML accelerator architectures have become increasingly heterogeneous and specialized. As Moore's law declines, there is an increasing motivation to explore novel architectures to overcome the limits of technology scaling. Recently, the rise of large multi-gigabyte machine learning models has exacerbated the memory wall issue for ML accelerators, and has inspired further optimizations for on-chip data management.

1.1 The Memory Wall

Since the introduction of the transformer architecture in 2018 [67], the size of notable SoTA machine learning models has grown exponentially [69]. Large language models (LLMs) are (typically) encoder-decoder transformers trained for natural language processing (NLP) tasks, and the primary contributor to this growth in model parameters. While LLMs have exploded in popularity for both research and industrial applications, their underlying mechanisms are largely a "block box" and an active area of research. However, one property remains clear: LLMs trained with more parameters exhibit emergent abilities and yield increasingly better performance. That is, larger models trained with more parameters acquire unique abilities that aren't present in smaller models [70, 76].

Since at least the mid-90s, researchers have observed that computing resources are not scaling proportionately with off-chip memory bandwidth [45]; and the rise of large multi-gigabyte ML models has exacerbated the memory wall issue for the compute accelerators that execute them [76, 77]. This bottleneck can be reasoned about

by using a workload's arithmetic intensity; or the ratio between the compute operations necessary to execute a unit of work, and the number of bytes that must be accessed [43, 76]. For example, assume we have an NVIDIA A10 GPU with an ideal arithmetic intensity of 208.3 OPs/byte [49]. The standard implementation of the attention layer in Llama 2 7B [63] with float-16 weights and a sequence length of 4096 tokens has an arithmetic intensity of 62 OPs/byte [21, 66], far below our maximum compute capacity (about 30%).

The memory wall is also an issue for models with poor arithmetic intensity, not because they are large, but because of their high throughput inference requirements. Specifically, an increasing number of scientific fields are relying on high data-rate, low-latency neural networks (NNs) to process data recorded by sensors with extremely high sampling rates [72]. For example, take the autoencoder for lossy data compression of calorimeter readings in the CERN Large Hadron Collider [29]. This model has so few parameters that it uses <1% of our compute capacity with an arithmetic intensity of about .98 (assuming the model has float-16 weights for the sake of comparison). The challenge lies in inferencing data within the few nanoseconds between sensor readings. In practice, GPUs are not typically used for similar scientific applications as they usually have strict power constraints.

The memory wall has created a new interest in utilizing reconfigurable computing as the vehicle for ML acceleration [51]. Specifically, coarse-grained reconfigurable arrays (CGRAs) are seeing new applications as compute accelerators for machine learning inferences for the following reasons:

Flexibility. CGRAs offer the flexibility of creating specialized logic for different emerging neural network architectures and operations that application-specific integrated circuits (ASICs) simply cannot, and at a much lower price point. While field-programmable gate arrays (FPGAs) offer more flexibility than CGRAs, CGRAs trade this flexibility for increased clock frequencies and reduced compilation times [51].

Performance. When compared to datacenter GPUs, CGRA architectures tend to exhibit superior performance-per-area, even with inferior transistor node technology. Additionally, CGRAs allow for the creation of application specific optimizations such as customized pipelines for minimal latency [13].

Parallelism. CGRAs offer more opportunities to exploit parallelism as they allows for the creation of custom memory hierarchies and compute units. Additionally, they are able to concurrently process data from multiple external sensors through high-speed I/O directly interfaced with the fabric [13].

In 2020, AMD/Xilinx introduced their AI Engine (AIE), a CGRA architecture which consists of a 2D mesh of general purpose VLIW processors [10]. This paper is a survey of the AIE architecture, specifically looking at:

- How the AIE architecture compares with prior and contemporary CGRA designs, as well as current ML accelerators.
- Existing AIE toolchains and runtimes, both official and third party, across industry and academia.
- Applications deployed on AIEs.

2 BACKGROUND

Although largely dominated by GPGPUs, deep learning (DL) has become the latest focus for reconfigurable computing [41, 51]. While field-programmable gate arrays (FPGAs) and coarse-grained reconfigurable arrays (CGRAs) have traditionally been utilized for applications such as digital signal processing (DSP), both DSP and DL workloads share a common compute-intensive and highly parallel operation: matrix multiplication (MM). As of 2017, Google reported that 90% of their neural network (NN) inference workloads consisted of dense matrix multiplies [36, 77]. Predictably, this has only been exacerbated since the rise of large transformer models in the following years.

The follow section will provide a background on what coarse-grained reconfigurable arrays are, and how they came to be used for accelerating machine learning tasks. We will also look at the history of 2D processor mesh architectures as they lead up to the introduction of the AI Engine.

2.1 Coarse-Grained Reconfigurable Arrays

To understand what coarse-grained reconfigurable arrays are, it is essential to understand the reconfigurable architecture they descended from: field-programmable gate arrays. Originating in the mid-80s, FPGAs were developed to facilitate the simulation and testing of ASICs [51, 64]. Even today, taping out an ASIC comes with a high cost of error and a lengthy turnaround time of weeks to months. FPGAs were designed to simulate any digital logic design with a fine-grained reconfigurable fabric of logic cells known as look-up tables (LUTs). LUTs consist of programmable static random-access memory (SRAM) and a number of input and output wires. The SRAM defines a look-up table which describes a mapping between its input and output signals. FPGAs also contain many programmable interconnects to route connections between LUTs.

While FPGAs were originally developed as a simulation tool, they began to be used as general-purpose compute devices in their own right by the early 90s [51]. FPGAs saw use in telecommunication, military, and automotive applications that required custom, small-scale, high-performance logic that didn't require the efficiency of an ASIC. However, FPGAs have a number of limitations that reduce their effectiveness for general purpose compute. Fine-grained programmability comes at the cost of reduced clock speeds, with many designs running between one to two orders of magnitude slower than an ASIC counterpart [51]. Finding the ideal synthesis, placement, and routing of logic elements means that the electronic design automation (EDA) flow can take hours to days for large designs. About 90% of FPGA fabric consists of interconnects which largely increases the routing problem space, and dedicates a significant amount of power and area away from compute [25]. Additionally, essential arithmetic logic did not map well to the FPGA architecture as blocks such as a simple integer MAC unit could consume a large fraction of an FPGA's resources [51].

Coarse-Grained Reconfigurable Arrays were developed in order to address the limitations of FPGAs by trading some reconfigurability to coarsen certain computing elements in order to make better us its silicon, improve clock speeds, and reduce compilation times. Early CGRAs provided a small amount of coarsening over FPGAs, with architectures like Garp [30] being programmable at a 2-bit

granularity, instead of 1-bit like an FPGA [51]. Following CGRAs like CHESS [44], REMARC [47], and MATRIX [46] added higher bit-width programmability, contained hard compute elements such as programmable ALUs and register files, and introduced new network topologies which enabled communication between neighboring compute elements.

It should be noted that many early CGRAs were tightly-coupled accelerators (TCAs): co-processors which "consist of one or more specialized hardware functional units which can accelerate critical portions of an application kernel" [20]. TCAs are embedded into, or closely with the host processing core, and are typically accessed via instructions extended onto the host instruction set architecture (ISA). The host core shares resources with the TCA, such as register files, memory-management logic, and caches; which means that the host core can stall during TCA execution.

In 2000, MorphoSys [57] and PipeRench [27] added new features now common in modern CGRAs, including the addition of a hardware integer multiplier in the ALU, and new network topologies and virtualization techniques [51]. These were some of the first loosely-coupled accelerators (LCAs): co-processors which are located outside of the host processing core and are capable of direct, independent access to global memory [20]. This allows for more complex accelerator designs than with TCAs, since LCAs are not at risk of degrading the host processor's performance. The hardware additions and network structures introduced by MorphoSys and PipeRench provided the basis for many modern CGRAs [51].

2.2 2D Processor Meshes

The 2D mesh topology is the most commonly used network topology between compute units in CGRAs [51]. Compared to other network topologies, meshes allow for dynamically reconfigurable dataflows which can be scheduled by a compiler, time-sharing or context-switching of resources, and a larger degree of scalability. Mesh topologies are used to network reconfigurable cells of varying complexity, from integer ALUs with a register file [57], to general purpose processors.

This section will focus on 2D meshes of general purpose processors, which can also be considered as distributed memory systems. Distributed memory systems try to optimize access to global resources, such as I/O and global memory, by using multiple processing nodes which are interconnected via a high-speed network [75]. Nodes each have some amount of local memory and communicate with each other via message passing or shared memory.

Figure 1 shows the subsection of a traditional multiprocessor GPGPU and its memory hierarchy (using the NVIDIA naming convention). While this figure depicts a GPU, its memory hierarchy is similar to other general purpose multiprocessor systems. Compared to memory hierarchy shown in Figure 5, which demonstrates the interconnects between AI Engine tiles (as will be explored later in Section 3), distributed memory systems can avoid or reduce contention for global memory. Memory behavior is more deterministic, as traditional cache flushes will not occur. This comes at the cost of added complexity: data movement must be managed manually; kernel programming must adapt new communication models as processor nodes may not share any address space, or may lack

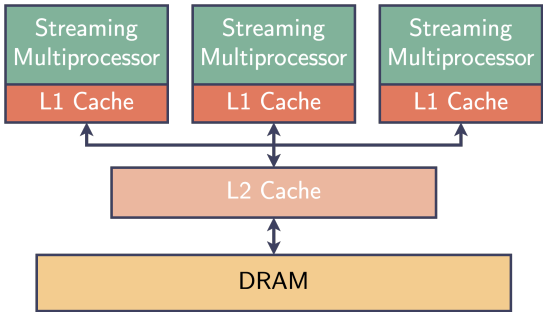

Figure 1: Traditional GPGPU Memory Hierarchy

shared memory all together; and local memories may be smaller than a traditional L1 cache.

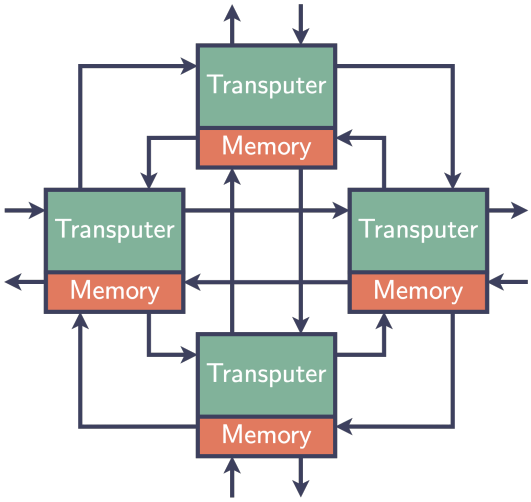

Figure 2: Transputer Network

2.2.1 Transputers. Developed in the early 1980s, Transputers were one of the first processors explicitly designed for parallel, distributed memory computing [18, 39]. A Transputer was a discrete processor which consisted of a stack machine, some local memory, and communication links [73]. Multiple nodes were connected together to form large Transputer networks which could be easily scaled, as seen in Figure 2.

Transputers were designed around the Occam programming language [34, 50], one of the first programming languages built on the concepts of communicating sequential processes [32]. Occam introduced parallel programming concepts for reasoning about and managing concurrency, such as channels [55], which mapped well onto the Transputer hardware. Many of these concepts are now found in modern networking languages, such as Go [14, 28].

Transputers saw use in early supercomputing [18, 31], as well as reconfigurable computing research. By 1995, Transputers were being paired with FPGAs for parallel application acceleration [40]. While Transputers weren't CGRAs, the idea of combining a 2D mesh of hard processor cores, along with some programmable logic

to supplement the processors hardware, is still in use today and can be seen in modern designs such as the AI Engine [10].

2.2.2 CGRAs. One of the first CGRA designs to feature a 2D mesh of general purpose processors was the 167-processor[51, 65], designed in 2009 to accelerate DSP workloads. The 167-processor contains 164 homogeneous processor nodes and introduced new multi-core clocking techniques to reduce power consumption. The processor nodes have relatively simple 16-bit architectures, containing only a fixed-point ALU and multiplier; a 40-bit accumulator; a 128x35-bit instruction memory and 128x16-bit data memory; and two 64x16-bit FIFOs for interprocess communication (IPC) [65]. All processors share three on-chip 16KB global memories. Similar to the Transputer [18], 167-processor nodes each have 4 bidirectional asynchronous communication links between neighbors.

The RHyME/REDEFINE [22, 42, 51] architecture was introduced in the mid-2010s to accelerate MM-based high-performance computing (HPC) applications, and consists of a 6x4 2D mesh of HyperCells. HyperCells are processing nodes with a dataflow architecture [68] designed for executing dataflow graphs. Each HyperCell has an 8x64-bit register file and 16KB of configuration memory [22]. A multi-core HyperCell system has 768 KB of global memory and features a honeycomb network topology which is managed by the RECONNECT network-on-chip (NoC) architecture [26].

Plasticine [51, 52] is an accelerator for general parallel applications which was published in 2017. A Plasticine mesh consists of two kinds of "pattern units": pattern compute units and pattern memory units. A pattern compute unit contains an ALU with floating-point support, as well as single-instruction-multiple-data (SIMD) vector support; control logic; and communication FIFOs. Pattern memory units contain scratchpad memory for the pattern compute units, and address generation units (AGUs) for global memory access. The Delite Hardware Definition Language [59] provides parallel pattern constructs for users to map their workloads to and is used to statically configure Plasticine. Plasticine is one of the first CGRAs to be used for machine learning acceleration, and one of the few to be usable for accelerating training (stochastic gradient descent).

2.3 CGRAs for Machine Learning

Machine learning training typically requires model parameters to be represented with high-precision, floating-point data types. Inference, however, does not require as much precision and can operate on quantized model weights. Quantization is the process of reducing the precision of a floating-point value by projecting it onto a lower bit-width representation. Many models are able to be quantized to integers of 8 bits or less [71]. Integer quantization also has the effect of significantly reducing arithmetic logic (integer ALU instead of floating-point), making weights easier and faster to compute with.

While GPGPUs are typically the dominant hardware choice for ML training, CGRAs are being explored as the vehicle to exploit the efficiency of lower dimensional datatypes for inference tasks. This is a natural extension of previous CGRA work, as CGRA architectures have historically been optimized for integer MAC-heavy workloads, and for matrix multiplication. Additionally, GPGPUs are typically designed to achieve the absolute best compute performance at the cost of high power consumption, whereas CGRA architectures are typically optimized for size and power efficiency [51]. For edge deployments which only require inference tasks, CGRAs may be the preferable choice for acceleration.

3 AI ENGINE ARCHITECTURE

In 2018, AMD/Xilinx announced the first generation of the AI Engine architecture as part of their new Versal Adaptive Compute Acceleration Platform (ACAP) [9]. The Versal ACAP is a line of FPGA system-on-chips (SoCs) which contain an Arm Cortex-A72 application processor, programmable logic (PL), NoC, and an AI Engine array. An AI Engine array contains a 2D mesh of AI Engine tiles. Each tile contains a 6-way VLIW processor with a 32-bit scalar RISC processor, 512-bit fixed-point vector unit; 512-bit floating-point vector unit; two vector load units; and one vector store unit [11]. AMD/Xilinx has not publicly released the instruction set for the AI Engine, but the instruction format can be seen in Figure 4. A diagram of the AI Engine functional unit can be seen in Figure 6. In addition to a VLIW processor, an AI Engine tile also contains a 1024x128-bit (total of 16KB) read-only program memory; data memory interface; cascade stream interface; memory-mapped AXI4 debug interface; and an AXI4-Stream interface consisting of two 32-bit input and two 32-bit output ports.

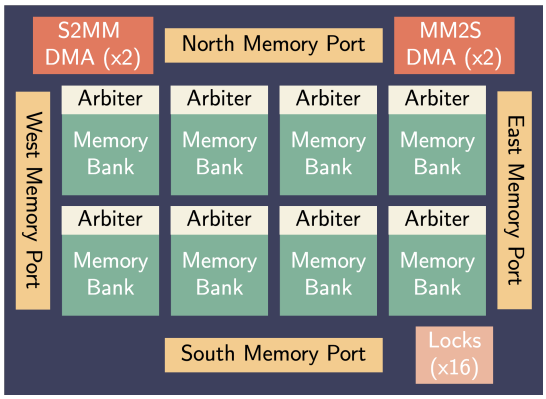

Figure 3: AI Engine Memory Module

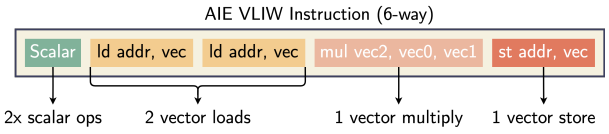

Figure 4: AI Engine Instruction Format

3.1 Motivation

In a 2020 white paper titled "Architecture Apocalypse", AMD/Xilinx discusses the motivation behind their development of the Versal ACAP [1]. The author explains that the Versal AI Engine array is a multi-core architecture that essentially lies somewhere in-between a GPU, and a dedicated ML accelerator ASIC like the Google TPU. By opting for a 2D mesh of VLIW Vector processors, the AIE was

designed to avoid the complex memory hierarchies and cache non-determinism of GPUs and increase communication bandwidth by having direct interconnects between processors. The AIE was designed for around realtime applications, such as DSP tasks for 5G wireless backhaul; and for ML inference.

3.2 Intra-AIE Communication

AI Engine tiles can communicate with each other through three ways: shared memory, cascade stream, and AXI4-Stream. These interconnects can be seen in Figure 5.

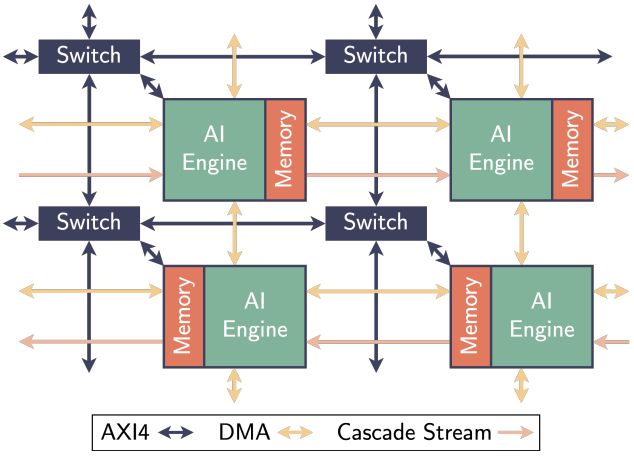

Figure 5: AI Engine Interconnects

3.2.1 Shared memory. In-between all AI Engine tiles are AI Engine memory modules [11], which can be seen in Figure 3. In the original AI Engine architecture, each memory module contains eight banks of 256x128-bit (total of 32KB) single port memory. The first two of these banks have error-correcting code (ECC) protection, while the rest have parity checking. The ECC protection can detect and correct 1-bit errors, and detect 2-bit errors per each 32-bit word. Memory tiles contain two input AXI4-Stream to memory-map DMA interfaces, and two output memory-map DMA to AXI4-Stream interfaces. Additionally, memory modules have synchronization logic including 16 hardware locks and a round-robin request arbitrator to satisfy one new request per cycle.

While AMD/Xilinx literature depicts memory modules as being integrated with AI Engine tiles (as seen in Figure 5), they can be thought of as separate tiles whose memory is shared between neighboring AI Engine tiles on the North, South, East, and West.

Because the memory modules are shared between neighboring AI Engine tiles, they can communicate through shared memory via their data memory interfaces. This means that under ideal scheduling, a maximum of four local AI Engine processors can each store two and load one 256-bit vector(s) into a shared buffer. AI Engine tiles can also access a maximum of four neighboring memory modules as a contiguous block of memory. Shared memory can be used to create pipeline and graph dataflow kernel execution models.

If an AIE tile wishes to access a non-neighboring memory module, it can use a memory-mapped AXI4 to AXI4-Stream interface (located on a local memory module) to do this, at the cost of additional latency since all stream ports have a 32-bit width (memory modules are 128-bits wide). DMA can be used in this way to synchronize ping-pong style memory buffers between AI Engine tiles.

3.2.2 Cascade stream. AI Engine tiles can do a single-cycle, one-way transfer of a 384-bit accumulator register to a horizontal neighbor. That is, a processor can transfer an accumulator to either a left or right neighbor, depending on the row. Each AIE row flip-flops between having the cascade flow left or right.

3.2.3 AXI4-Stream. Each AI Engine tile has two 32-bit input and two 32-bit output AXI4-Stream ports. Stream ports support either circuit-switched or packet-switched communication. Circuit-switching must be configured at compile time, but it allows for direct streaming from a single source to many destinations (through broadcasting). While the maximum number of dedicated circuit-switched streams supported by each interconnect may be limited, it ensures that communication has deterministic latency. Packet-switching effectively allows all AI Engine tiles to share their ports with one-another by using network packets with address headers. This allows for more flexibility in communication routing, at the cost of non-deterministic latency to due potential resource contention and backpressure.

3.3 External Array Communication

In addition to AI Engine tiles, dedicated interface tiles for external communication are located at the bottom of the array. These consist of PL, NoC, and configuration interfaces.

3.3.1 AIE to PL. Programmable-logic interface tiles allow for bidirectional communication between AIE tiles and PL designs via AXI4-Streams. Each PL interface tile has six AIE-to-PL, and eight PL-to-AIE streams per AIE tile column.

3.3.2 AIE to NoC. Network-on-Chip interface tiles allow the AIE to communicate with peripherals on the Versal NoC. Each NoC interface tile has 4 AIE-to-NoC, and 4 NoC-to-AIE AXI4 streams. Through the Global Memory I/O (GMIO) programming model, sections of global DRAM can be memory-mapped into the address space of an AIE tile.

3.3.3 AIE Configuration. AIE arrays only have one configuration interface tile. These tiles handle reset and clock generation for AIE tiles, as well as loading ELF files to the program and data memories of AIE tiles.

3.4 AI Engine Variants

Since the first generation of the AI Engine was released in 2020, AMD/Xilinx has released and announced two new versions.

3.4.1 AIE-ML. The AIE-ML is the second AIE architecture, and was released as part of the Versal Edge series in 2022. While the AIE was marketed as both a DSP and ML tool, the AIE-ML is optimized solely for machine learning inference [12]. The AIE-ML is architecturally similar the the AIE, the the key differences being that the AIE-ML additionally supports the bfloat16 and int4 datatypes, and can achieve double the int8 MACs per cycle (from 128 to 256). A

maximum of 1024 int4 MACs/cyc are possible. The trade-off is that AIE-ML tiles no longer support native 32-bit floats.

Data memory modules have double the capacity, from 32KB to 64KB (eight 8KB banks). Additionally, AIE-ML arrays have new dedicated memory tiles to increasing on-chip memory. Each memory tile contains 512KB of ECC-protected memory (sixteen 32KB banks) and DMA between neighboring memory tiles. Memory tiles can only be accessed vertically by AIE-ML tiles in the same column.

DMA controllers in all tiles support on-the-fly compression and decompression of sparse data to increase communication bandwidth. AIE-ML tiles all have the same cascade stream direction of West to East, and can also stream from North to South.

In additional to Versal Edge SoCs, AIE-ML arrays were added to the AMD Ryzen 7000 series processors [5]

3.4.2 AIE-ML v2. While not yet released, the AIE-ML v2 is a new AI Engine architecture that is part of the second generation of Versal SoCs [4]. AMD claims that AIE-ML v2 tiles will have double the performance of AIE-ML, and can achieve 1028 int8 MACs/cyc (vs 512). AIE-ML v2 will also support new datatypes, including float8; and some of the Open Compute Project's Microscaling Formats (MX), which are an emerging industry-standard implementation of low-precision floating points [53].

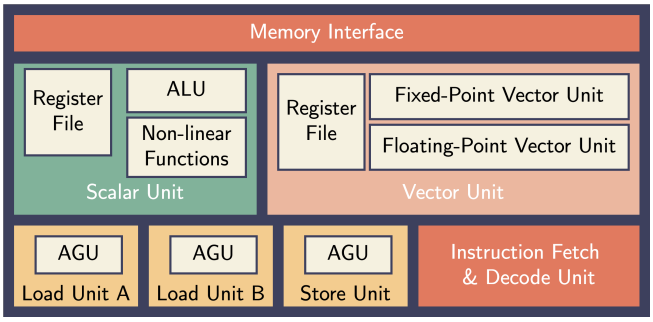

Figure 6: AI Engine

4 OFFICIAL AIE DEVELOPMENT TOOLS

Single kernel programming can be done in C/C++, and the "aiecompiler" tool is included as part of AMD/Xilinx's Vitis Software Platform [7]. To optimize code for the AIE's hardware, there is an AI Engine API which provides a programming abstraction including a way to map data onto vector instructions, and an interface for handling streaming data [3]. For lower-level programming, developers can also use AI Engine intrinsics which map almost directly to AIE instructions. Examples of intrinsics include manual loads and stores, managing data memory locks, and low level streaming control. An AI Engine tile can execute more than one kernel during runtime, the budget for which is dictated by their "run-time ratio". The "run-time ratio" dictates how many cycles can be afforded per kernel invocation, as a ratio of cycles per kernel to the overall cycle budget. Multiple kernels on a single AI Engine tile will execute sequentially.

For managing system-level dataflow between AI Engine tiles/kernels, the "aiecompiler" uses an Adaptive Dataflow graph (ADF) which is a C/C++ abstraction to describe a dataflow graph. Nodes in this graph are AI Engine kernels, and edges are data streams. ADF graphs also allow the programmer to establish connections with programmable logic I/O (PLIO) and global memory I/O (GMIO).

To facilitate the development of higher level compilation flows, AMD/Xilinx has released their own LLVM multi-level intermediate representation (MLIR) dialect for AI Engines called MLIR-AIE [2, 6].

Verification of AI Engine graphs can be done through a variety of simulators. The "x86simulator" can quickly verify functional correctness and independent interactions between kernels in an untimed context. The "aiesimulator" provides a cycle accurate simulation of AIE kernels, and can model external array communication with simulated GMIO and PLIO interfaces. Compared to the "x86simulator", "aiesimulator" offers more accurate performance and power estimations. Both simulators allow the generation of traffic for functional verification. Additionally, the Xilinx Power Estimation (XPE) can be used to estimate design power consumption. XPE takes into account the number of AIE tiles used, vector hardware usage, and memory accesses [8].

5 OPEN-SOURCE AI ENGINE DEVELOPMENT

A significant amount of research literature around AI Engines has primarily focused on facilitating application development. Specifically, these works have investigated how to map certain workloads onto the AIE architecture, and best utilize the hardware. AI Engines and GPUs are both multi-core architectures, but their programming paradigms are vastly different, primarily as a result of their contrasting memory models. As can be seen in Figure 1, GPU cores live in the same multi-GB address space. Although best kernel programming practices necessitate optimizing for data reuse in register files and L1-Cache, having the same address space provides a significant degree of flexibility for data movement.

GPUs are designed to handle general purpose parallel workloads, whereas the AI Engine architecture has been designed around a dataflow model. While AIE tiles can somewhat access memory from non-neighboring tiles, they effectively constitute a distributed memory system in which each system (AIE tile) has a limited amount of memory (see Figure 5. Instead of having cache controllers provide memory coherency, data movement between AIE tiles is deliberate and manual.

Mapping algorithms to distributed memory CGRAs is active area of research

5.1 Simulation

5.1.1 Graphtoy. In early 2024, the Graphtoy simulation tool was created to facilitate graph development for AI Engines [58]. Graphtoy was designed to address limitations of the official Vitis-AIE workflow, specifically the the large upfront cost of having to learn the Vitis flow and spending a disproportionate amount of time optimizing code for the AIE hardware before verification simulation can even begin. Additionally, the official AMD/Xilinx AIE simulation can take a significant amount. The tool aims to allow AIE developers to only focus on higher-level graph development and debugging, before deploying on actual AIE hardware.

To do this, Graphtoy is a simulation framework built on a C++20 coroutine scheduler which accurately models multi-kernel graph

execution. The Graphtoy library provides a stream abstraction to model the dataflow streams of the AI Engine, and allows the developer to write a graph version of their algorithm. Each kernel is essentially an infinite coroutine that receives and emits data from an arbitrary number of I/O streams. For external AIE array communication, there are memory source and sink stream-based DMA abstractions which are a parallel to AIE GMIO. Graphtoy graphs can then be compiled and executed on the developer's own host machine using a standard C++ compiler. By using stackless coroutines to model parallel kernel execution, the overhead of thread context switches can be avoided to further improve simulation time. Additionally, kernels can be debugged with a standard C++ debugger.

To port a Graphtoy application to AIE hardware, the user manually copies their kernel to their AIE project and replaces any Graphtoy-specific library functions with their equivalent AIE intrinsic.

To evaluate the usefulness of Graphtoy, the authors used their tool to port a mini-version of AutoDock [48], an HPC application used to model molecular docking. After creating a graph version of AutoDock, the author's results showed that Graphtoy could simulate the design nearly an order of magnitude faster than the official AI Engine "x86simulator" tool (27.5 seconds vs 299 seconds).

5.2 Compilation

Because of the challenges in optimizing kernels for the AI Engine architecture, there have been a number of works which seek to automate the development of certain critical portions of kernel code, through their own custom compilation flow.

5.2.1 Vyasa. Vyasa is an auto-vectorizing compiler for convolution kernels on AI Engines, published in 2020 [15]. The front-end of the Vyasa compiler is an extension of the Halide domain-specific language [54], which is an image processing language for the description, compilation, and scheduling of high-performance, parallel image processing pipelines. The existing AI Engine compiler handles VLIW scheduling and loop optimizations (pipelining, unrolling, etc), but it has no support for auto-vectorization. While auto-vectorizing code is still an active area of research, Vyasa can generate vectorized code for tensor convolutions from a high-level Halide description. The target of Vyasa's vectorization is the AIE fixed-point vector unit which supports 2D SIMD operations; the authors claim their tool can be generalized to other SIMD instructions on similar hardware platforms. The goal of Vyasa is to facilitate the development of computer vision and deep learning applications which require 2D convolutions.

Vectorized Vyasa code takes advantage of operation chaining, which is when a processor performs a vector operation and reduces the end result to a scalar value, all without writing intermediate values to a register file. The authors claim that operation chaining helps to improve the kernel's energy efficiency. 2D convolutions align well with operation chaining since they are essentially multi-dimensional nested loops whose bodies contain a MAC operation between an input tensor and weight tensor value, and an update (reduction) operation to an output tensor value [15]. To map the Halide language onto operation chained instructions, the authors implemented the Triplet intermediate-representation (IR). A single triplet represents the accesses of two multiplication operands, and an update operand. Vyasa utilizes two compilation passes, one from Halide IR to Triplet IR, and then Triplet IR to AIE intrinsics. The generated code is a single Conv2D kernel which will run on individual AIE Tiles. After compiling Halide IR to AIE intrinsics, Vyasa has an auto-tuning stage in which the compiler tries to find the best graph schedule. This stage explores various schedules as a function of various loop and data-layout choices. To verify optimal configurations, Vyasa uses the cycle-accurate AI Engine simulator.

Vyasa was evaluated on 36 Conv2D workloads in computer vision and deep learning applications. Optimal schedules obtained by the auto-tuning tool were able to achieve a geometric mean of 7.6 and 24.2 MACs-per-cycle for 32-bit and 16-bit vector operands respectively. Both configurations respectively achieved 95.9% and 75.6% peak performance of the AIE vector units.

5.2.2 PyAIE. In mid-2023, PyAIE was created to reduce the complexity of AI Engine development by providing a high-level Python interface for kernel programming [62]. PyAIE is an extension of the PyLog library [33], which itself is a tool to compile Python to high-level synthesis (HLS) code through a custom IR. Like PyLog, PyAIE users add a decorator above the python functions which they wish to run on an AIE. PyAIE then generates host and AIE kernel code using intrinsics, and uses the CHARM [77] matrix multiplication accelerator architecture (which will be discussed later) if it detects a large matrix multiplication. PyLog is used to generate data movement and construction logic in HLS.

5.3 GEMMs

General matrix multiplication (GEMM) lies at the heart of many DSP and deep learning algorithms, and they are often the most compute intensive operation, constituting most of an algorithm's runtime. Because GEMMs are highly parallel, most accelerator architectures have the primary goal of accelerating matrix multiplication. Mapping GEMMs onto heterogeneous architectures is non-trivial, and an active area of research for compiler and hardware designers. The same holds true for the AI Engine, and much of the AIE-related literature is focused on optimizing GEMMs for its 2D processor mesh architecture.

5.3.1 CHARM. Composing Heterogeneous Accelerators for Matrix Multiply on Versal ACAP Architecture (CHARM, not be confused with CHARM [19]) was published in early 2023, and claims to be the first paper to systematically study data movement and computation on the Versal architecture [77]. CHARM is a framework for optimizing and deploying tiled matrix multiplication accelerators onto an AI Engine array. The AI Engine isn't the first CGRA architecture to feature a large number of processing elements with distributed memory, and there have been many previous works exploring how to increase data reuse and computational parallelism on them. The CHARM authors claim that these works are "one-size-fits-all", and they cannot efficiently handle GEMM layers with varying shapes and sizes. Instead, CHARM uses custom design space exploration and composition algorithms to map multiple GEMM accelerators of various sizes onto an AIE array.

CHARM uses a 4-level memory hierarchy for tiling. At bottom level are the actual AIE registers. The second level is an AIE tile's

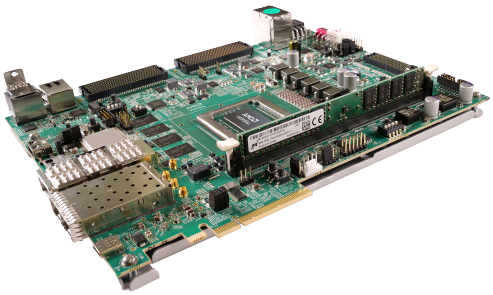

Figure 7: Versal VCK190 Evaluation Kit

local memory, in which an AIE stores and computes its GEMM tile in a fully unrolled loop. The third level is the on-chip memory buffers located in the Versal's programmable logic and accessible to the AIE array through PLIO. These buffers marshal data into the AIEs and stores accumulates partial results. The outermost level is the Versal's off-chip memory. Because there are significantly less PLIOs than AIEs, CHARM utilizes both circuit-switched and packet-switched communication between the two. Packet-switching allows a PLIO to communicate to more AIE tiles than with circuit-switching. Charm uses packet-switching for the PLIO to broadcast left-hand-side matrix row data to multiple AIE tiles for increased memory re-use. The PL logic handles moving memory to and from off-chip memory, and it contains three sets of buffers which are double-buffered.

For design space exploration, CHARM uses a fixed tile size of 32x32x32 to determine that number of AIE and PLIO tiles, and on-chip memory buffers necessary for the given matrix multiplication. The CHARM composer tool finds an optimal workload assignment and resource partitioning scheme between GEMM accelerators. Finally, CHARM generates host, kernel, and PL code.

CHARM was evaluated on a number of different matrix multiplication configurations corresponding to various machine learning models, including BERT [23]. Testing was done with physical hardware on a VCK190 (see Figure 7, which has the largest AI Engine array at 400 AIE tiles.

In mid-2023, CHARM was extended with the AutoMM [78] tool, which provides a Python API for generating matrix multiplication accelerators using the CHARM framework. AutoMM running on a VCK190 was evaluated against an NVIDIA A100 datacenter GPU using the PyTorch runtime on a neural collaborate filtering (NCF) and a multi-layer perceptron (MLP) benchmark. For

On the NCF benchmark using float32 weights, the VCK190 achieved 2.3 TFLOPs with an an energy efficiency ratio of .96x, whereas the A100 achieved 3.5 TFLOPs at an energy efficiency ratio of 1x. The authors believe the VCK190 performance degradation was a result of smaller matrix multiplication sizes. The MLP benchmark also used float32 weights, and the VCK190 achieved 3.5 TFLOPs at an energy efficiency ratio of 1.16x, versus 13.7 TFLOPs and 1x respectively for the A100.

5.3.2 MaxEVA. MaxEVA was published in late 2023, and provides a framework for mapping a single tiled matrix multiplication accelerator onto an AIE array [60]. This work addresses the limitations of CHARM; specifically its memory movement bottlenecks. A custom mapping strategy is used to eliminate inefficient packet-switched communication due to the limited number of PLIOs available.

MaxEVA uses two AIE kernels: MatMul and Add. Each MatMul kernel runs a single tiled matrix multiply on an individual AIE core. Add kernels are used to reduce the output of the MatMul kernels and reduce the output to the PLIOs. Add kernels are organized as adder trees, with multiple Adds running on a single AIE tile. This is possible because an Add kernel's latency is significantly lower than that of MatMul.

Given the sizes of the input matrices, MaxEVA can exhaustively calculate optimal tiling dimensions given that they are powers of 2; this in contrast to CHARM which uses a fixed tile size. Afterwards, an optimal kernel placement strategy can be found which determines the ratio of MatMul tiles per one Add tile (which executes an adder tree reduction on said MatMul tiles). The authors note that they found two optimal patterns for the VCK190, which were 3:1 and 4:1 MatMuls to Adds respectively. These strategies ensure little to no DMA usage between AIE data memories. MaxEVA was evaluated against CHARM and achieved performance gains of 2.19x higher throughput and 20.4% greater energy efficiency.

As of the time of this writing, MaxEVA has the state-of-the-art GEMM implementation for the AI Engine architecture. Although, it should be noted that unlike CHARM, MaxEVA only allows for GEMMs whose data can fit entire on-chip. In early 2024, MaxEVA was extended [61] to support an additional off-chip memory hierarchy to allow for arbitrary sized GEMMs.

6 AI ENGINE APPLICATIONS

As the name "AI Engine" implies, most of the AI Engine application literature revolves around accelerating machine learning inference. AI Engines have also seen interest in finance, weather simulation, and space applications.

6.1 Graph Neural Network Acceleration

6.1.1 H-GCN. In 2022, the AI Engine was first used for Graph Neural Network (GNN) acceleration through the H-GCN framework [74]. GNN acceleration has unique challenges, which include irregular data access patterns and workload imbalances from skewed graph degree distribution. For example, a subgraph me be densely clustered, whereas a neighboring subgraph may be loosely clustered or scattered. These challenges make acceleration on traditional CPU and GPU hardware difficult.

H-GCN is a framework which takes advantage of the heterogeneous hardware of the Versal architecture to perform graph convolutional network (GCN) acceleration. GCNs have two phases: aggregation (message passing), and combination (node embedding) [74]. Previous works have focused on accelerating these phases, whereas H-GCN instead focuses on optimizing for the heterogeneity of the graph itself. To achieve this, H-GCN maps tightly and loosely clustered subgraphs onto AI Engines, and scattered nodes are implemented on the Versal's programmable logic.

For evaluation, H-GCN was compared with an Intel Xeon Gold 6238R and 384 GB of DDR4 memory; an NVIDIA RTX 2060 SUPER with 8 GB of VRAM. H-GCN itself was run on a VCK5000 accelerator card. Both devices (CPU and GPU) were evaluated against across a range of existing GCN acceleration frameworks. H-GCN achieved up to a 155.2x speedup versus the CPU, and up to a 37.8x speedup compared to the GPU.

6.1.2 Chen et al. In 2023, another GNN accelerator framework for Versal SoCs emerged from Chen et al [16]. Like H-GCN, the authors used the AI Engine array for accelerating dense and loosely clustered subgraphs, and used the PL for scattered nodes. However, Chen et al uses the Versal application processor to dynamically partition and reorder computation tasks between the PL and AIE based on sparsity in the data. Because the graph doesn't have to be partitioned and reordered ahead of time, dynamic graph partitioning is much simpler and more flexible.

Chen et al's framework achieved a 9.9x speedup compared to H-GCN on the same VCK5000 when evaluated on the same workloads.

6.2 Finance

FPGAs have historically seen use in high-frequency trading applications, but their learning curve which requires familiarity with hardware description languages creates a high barrier to entry for developers [37]. In early 2024, Klaisoongnoen et al evaluated the AI Engine architecture in a VCK5000 on the STAC-A2 inspired market risk (SIMR) benchmark. The authors found that the limited number of AXI interfaces between the AIE array and the PL created a performance bottleneck, and saw better performance when only using the Versal PL. Because efficient AIE runtimes are just emerging, it is likely that the AIE architecture will be revisited in the future for further evaluation on finance applications.

REFERENCES

[1] Gupta Alok. 2020. Architecture apocalypse dream architecture for deep learning inference and compute-versal ai core. *Embedded World* (2020).

[2] AMD. 2023. T2: Leveraging MLIR to Design for AI Engines. https://www.xilinx.com/content/dam/xilinx/publications/presentations/leveraging-mlir-to-design-for-aie-fpga-2023.pdf.

[3] AMD. 2024. AI Engine Kernel and Graph Programming Guide (UG1079). https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding.

[4] AMD. 2024. AMD Versal™ AI Edge Series Gen 2. https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal/gen2/ai-edge-series.html.

[5] AMD. 2024. The Future of AI PCs Gets Even Better with AMD. https://www.amd.com/en/products/processors/consumer/ryzen-ai.html.

[6] AMD. 2024. mlir-aie. https://github.com/Xilinx/mlir-aie.

[7] AMD. 2024. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393). https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration/Programming-Versal-AI-Engines.

[8] AMD. 2024. Vivado Design Suite User Guide: Power Analysis and Optimization (UG907). https://docs.amd.com/r/en-US/ug907-vivado-power-analysis-optimization/AI-Engine.

[9] AMD/Xilinx. 2018. Xilinx AI Engines and Their Applications (WP506). https://docs.amd.com/v/u/en-US/wp506-ai-engine.

[10] AMD/Xilinx. 2020. Versal: The First Adaptive Compute Acceleration Platform (ACAP). https://docs.amd.com/v/u/en-US/wp505-versal-acap.

[11] AMD/Xilinx. 2023. Versal Adaptive SoC AI Engine Architecture Manual (AM009). https://docs.amd.com/r/en-US/am009-versal-ai-engine.

[12] AMD/Xilinx. 2024. Versal Adaptive SoC AIE-ML Architecture Manual (AM020). https://docs.amd.com/r/en-US/am020-versal-aie-ml/Key-Differences-between-AI-Engine-and-AIE-ML.

[13] Andrew Boutros, Aman Arora, and Vaughn Betz. 2024. Field-Programmable Gate Array Architecture for Deep Learning: Survey & Future Directions. arXiv:2404.10076 [cs.AR]

[14] Matilde Broløs, Carl-Johannes Johnsen, and Kenneth Skovhede. 2021. Occam to Go translator. In *2021 IEEE Concurrent Processes Architectures and Embedded Systems Virtual Conference (COPA)*. 1–8. https://doi.org/10.1109/COPA51043.2021.9541431

[15] Prasanth Chatarasi, Stephen Neuendorffer, Samuel Bayliss, Kees Vissers, and Vivek Sarkar. 2020. Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–10. https://doi.org/10.1109/HPEC43674.2020.9286183

[16] Paul Chen, Pavan Manjunath, Sasindu Wijeratne, Bingyi Zhang, and Viktor Prasanna. 2023. Exploiting On-Chip Heterogeneity of Versal Architecture for GNN Inference Acceleration. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. 219–227.

[17] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. 2020. A Survey of Accelerator Architectures for Deep Neural Networks. *Engineering* 6, 3 (2020), 264–274.

[18] Cambridge Cherry Hinton and Ruth Ivimey. [n. d.]. Legacy of the transputer. *emergence* 80186 ([n. d.]), 19.

[19] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. CHARM: a composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design* (Redondo Beach, California, USA) *(ISLPED '12)*. Association for Computing Machinery, New York, NY, USA, 379–384. https://doi.org/10.1145/2333660.2333747

[20] Emilio G Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. 2015. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the 52Nd Annual Design Automation Conference*. 1–6.

[21] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2024. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA.

[22] Saptarsi Das, Nalesh Sivanandan, Kavitha T. Madhu, Soumitra K. Nandy, and Ranjani Narayan. 2016. RHyMe: REDEFINE Hyper Cell Multicore for Accelerating HPC Kernels. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*. 601–602. https://doi.org/10.1109/VLSID.2016.29

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[24] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 449–460. https://doi.org/10.1109/MICRO.2012.48

[25] Umer Farooq, Zied Marrakchi, and Habib Mehrez. 2012. *FPGA Architectures: An Overview*. Springer New York, New York, NY, 7–48.

[26] Alexander Fell, Prasenjit Biswas, Jugantor Chetia, S.K. Nandy, and Ranjani Narayan. 2009. Generic routing rules and a scalable access enhancement for the Network-on-Chip RECONNECT. In *2009 IEEE International SOC Conference (SOCC)*. 251–254. https://doi.org/10.1109/SOCCON.2009.5398048

[27] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. 2000. PipeRench: A reconfigurable architecture and compiler. *Computer* 33, 4 (2000), 70–77.

[28] Google. [n. d.]. Go. https://go.dev.

[29] Giuseppe Di Guglielmo, Farah Fahim, Christian Herwig, Manuel Blanco Valentin, Javier Duarte, Cristian Gingu, Philip Harris, James Hirschauer, Martin Kwok, Vladimir Loncar, Yingyi Luo, Llovizna Miranda, Jennifer Ngadiuba, Daniel Noonan, Seda Ogrenci-Memik, Maurizio Pierini, Sioni Summers, and Nhan Tran. 2021. A Reconfigurable Neural Network ASIC for Detector Front-End Data Compression at the HL-LHC. *IEEE Transactions on Nuclear Science* 68, 8 (2021), 2179–2186. https://doi.org/10.1109/TNS.2021.3087100

[30] John R Hauser and John Wawrzynek. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)*. IEEE, 12–21.

[31] Anthony J. G. Hey. 1990. Supercomputing with transputers—past, present and future. *SIGARCH Comput. Archit. News* 18, 3b (jun 1990), 479–489. https://doi.org/10.1145/255129.255192

[32] C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (aug 1978), 666–677. https://doi.org/10.1145/359576.359585

[33] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen Mei Hwu. 2021. PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow. *IEEE Trans. Comput.* 70, 12 (1 Dec. 2021), 2015–2028. https://doi.org/10.1109/TC.2021.3123465 Publisher Copyright: © 1968-2012 IEEE..

[34] Christian L Jacobsen and Matthew C Jadud. 2004. The transterpreter: a transputer interpreter. *Communicating Process Architectures 2004* (2004), 99–106.

[35] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. Motivation for and Evaluation of the First Tensor Processing Unit. *IEEE Micro* 38, 3 (2018),

10–19. https://doi.org/10.1109/MM.2018.032271057

[36] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[37] Mark Klaisoongnoen, Nick Brown, Tim Dykes, Jessica R. Jones, and Utz-Uwe Haus. 2024. Evaluating Versal AI Engines for Option Price Discovery in Market Risk Analysis. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 176–182.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[39] Robert Kuhn and David Padua. 2021. *Parallel Hardware*. Springer International Publishing, Cham, 11–54.

[40] Adrian Lawrence, Andrew Kay, Wayne Luk, Toshio Nomura, and Ian Page. 1995. Using reconfigurable hardware to speed up product development and performance. In *Field-Programmable Logic and Applications*, Will Moore and Wayne Luk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–118.

[41] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[42] Kavitha T. Madhu, Saptarsi Das, Nalesh S., S. K. Nandy, and Ranjani Narayan. 2015. Compiling HPC Kernels for the REDEFINE CGRA. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 405–410. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.139

[43] Dan C. Marinescu. 2018. Chapter 4 - Parallel and Distributed Systems. In *Cloud Computing (Second Edition)* (second edition ed.), Dan C. Marinescu (Ed.). Morgan Kaufmann, 113–150.

[44] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. 1999. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. 135–143.

[45] Sally A. McKee and Robert W. Wisniewski. 2011. *Memory Wall*. Springer US, Boston, MA, 1110–1116.

[46] Mirsky and DeHon. 1996. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE, 157–166.

[47] Takashi Miyamori and Kunle Olukotun. 1999. REMARC: Reconfigurable multimedia array coprocessor. *IEICE Transactions on information and systems* 82, 2 (1999), 389–397.

[48] Garrett M. Morris, David S. Goodsell, Ruth Huey, William E. Hart, Scott Halliday, Rik Belew, and Arthur J. Olson. 2001. AutoDock User's Guide Version 3.0.5. https://autodock.scripps.edu/wp-content/uploads/sites/56/2022/04/AutoDock3.0.5_UserGuide.pdf.

[49] NVIDIA. [n. d.]. NVIDIA A10 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/products/a10-gpu/.

[50] Conor O'Neill. [n. d.]. occam-2 language implementation manual. ([n. d.]).

[51] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. 2020. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access* 8 (2020), 146719–146743.

[52] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 389–402.

[53] Open Compute Project. 2023. OCP Microscaling Formats (MX) Specification Version 1.0. https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf.

[54] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[55] Anthony Roscoe. 1998. The theory and practice of concurrency. (1998).

[56] Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, Enrico Calore, Sebastiano Fabio Schifano, Maurizio Palesi, Giuseppe Ascia, Davide Patti, Stefania Perri, Nicola Petra, Davide De Caro, Luciano Lavagno, Teodoro Urso, Valeria Cardellini, Gian Carlo Cardarilli, and Robert Birke. 2023. A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms. arXiv:2306.15552 [cs.AR]

[57] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. 2000. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE transactions on computers* 49, 5 (2000), 465–481.

[58] Jonathan Strobl, Leonardo Solis-Vasquez, Yannick Lavan, and Andreas Koch. 2024. Graphtoy: Fast Software Simulation of Applications for AMD's AI Engines. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, Iouliia Skliarova, Piedad Brox Jiménez, Mário Véstias, and Pedro C. Diniz (Eds.). Springer Nature Switzerland, Cham, 166–180.

[59] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 1–25.

[60] Endri Taka, Aman Arora, Kai-Chiang Wu, and Diana Marculescu. 2023. MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine. In *2023 International Conference on Field Programmable Technology (ICFPT)*. 96–105.

[61] Endri Taka, Dimitrios Gourounas, Andreas Gerstlauer, Diana Marculescu, and Aman Arora. 2024. Efficient Approaches for GEMM Acceleration on Leading AI-Optimized FPGAs. arXiv:2404.11066 [cs.AR]

[62] Hongzheng Tian, Shining Yang, Yoonha Cha, and Sitao Huang. 2023. Late Breaking Results: PyAIE: A Python-based Programming Framework for Versal ACAP Platforms. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–2. https://doi.org/10.1109/DAC56929.2023.10247843

[63] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[64] Stephen M. Trimberger. 2015. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proc. IEEE* 103, 3 (2015), 318–331. https://doi.org/10.1109/JPROC.2015.2392104

[65] Dean N. Truong, Wayne H. Cheng, Tinoosh Mohsenin, Zhiyi Yu, Anthony T. Jacobson, Gouri Landge, Michael J. Meeuwsen, Christine Watnik, Anh T. Tran, Zhibin Xiao, Eric W. Work, Jeremy W. Webb, Paul V. Mejia, and Bevan M. Baas. 2009. A 167-Processor Computational Platform in 65 nm CMOS. *IEEE Journal of Solid-State Circuits* 44, 4 (2009), 1130–1144. https://doi.org/10.1109/JSSC.2009.2013772

[66] Philip Kiely Varun Shenoy. [n. d.]. A guide to LLM inference and performance. https://www.baseten.co/blog/llm-transformer-inference-guide/.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[68] Arthur H Veen. 1986. Dataflow machine architecture. *ACM Computing Surveys (CSUR)* 18, 4 (1986), 365–396.

[69] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. 2022. Machine Learning Model Sizes and the Parameter Gap. arXiv:2207.02852 [cs.LG]

[70] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. arXiv:2206.07682 [cs.CL]

[71] Olivia Weng. 2023. Neural Network Quantization for Efficient Inference: A Survey. arXiv:2112.06126 [cs.LG]

[72] Olivia Weng, Alexander Redding, Nhan Tran, Javier Mauricio Duarte, and Ryan Kastner. 2024. Architectural Implications of Neural Network Inference for High Data-Rate, Low-Latency Scientific Applications. arXiv:2403.08980 [cs.LG]

[73] Colin Whitby-Strevens. 1985. The transputer. *ACM SIGARCH Computer Architecture News* 13, 3 (1985), 292–300.

[74] Chengming Zhang, Tong Geng, Anqi Guo, Jiannan Tian, Martin Herbordt, Ang Li, and Dingwen Tao. 2022. H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 200–208.

[75] Peng Zhang. 2010. CHAPTER 17 - Distributed operating systems. In *Advanced Industrial Control Technology*, Peng Zhang (Ed.). William Andrew Publishing, Oxford, 685–732.

[76] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL]

[77] 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. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23)*. Association for Computing Machinery, New York, NY, USA, 153–164.

[78] Jinming Zhuang, Zhuoping Yang, and Peipei Zhou. 2023. High Performance, Low Power Matrix Multiply Design on ACAP: from Architecture, Design Challenges and DSE Perspectives. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247981