MTIA: First Generation Silicon Targeting Meta's Recommendation Systems

Amin Firoozshahian
Joel Coburn
Roman Levenstein
Rakesh Nattoji
Ashwin Kamath
Olivia Wu
Gurdeepak Grewal
Harish Aepala
Bhasker Jakka
Bob Dreyer
Adam Hutchin
Utku Diril†
Krishnakumar Nair
Ehsan K. Ardestani
Martin Schatz
Yuchen Hao
Rakesh Komuravelli
Kunming Ho
Sameer Abu Asal
Joe Shajrawi
Kevin Quinn
Nagesh Sreedhara
Pankaj Kansal
Willie Wei
Dheepak Jayaraman
Linda Cheng
Pritam Chopda
Eric Wang
Ajay Bikumandla
Arun Karthik Sengottuvel
Krishna Thottempudi
Ashwin Narasimha
Brian Dodds
Cao Gao
Jiyuan Zhang
Mohammad Al-Sanabani
Ana Zehtabioskui
Jordan Fix
Hangchen Yu
Richard Li
Kaustubh Gondkar
Jack Montgomery
Mike Tsai
Saritha Dwarakapuram
Sanjay Desai
Nili Avidan
Poorvaja Ramani
Karthik Narayanan
Ajit Mathews
Sethu Gopal
Maxim Naumov
Vijay Rao
Krishna Noru
Harikrishna Reddy
Prahlad Venkatapuram
Alexis Bjorlin

Meta Platforms Inc.
Menlo Park, CA, USA

ABSTRACT

Meta has traditionally relied on using CPU-based servers for running inference workloads, specifically Deep Learning Recommendation Models (DLRM), but the increasing compute and memory requirements of these models have pushed the company towards using specialized solutions such as GPUs or other hardware accelerators. This paper describes the company's effort in constructing its first silicon specifically designed for recommendation systems; it describes the accelerator architecture and platform design, the software stack for enabling and optimizing PyTorch-based models and provides an initial performance evaluation. With our emerging software stack, we have made significant progress towards reaching the same or higher efficiency as the GPU: We averaged 0.9x perf/W across various DLRMs, and benchmarks show operators such as GEMMs reaching 2x perf/W. Finally, the paper describes the lessons we learned during this journey which can improve the performance and programmability of future generations of architecture.

†Rivos Inc., work done while at Meta Platforms Inc.

ISCA '23, June 17–21, 2023, Orlando, FL, USA.

ACM ISBN 979-8-4007-0095-8/23/06...$15.00.
DOI: https://doi.org/10.1145/3579371.3589348

CCS CONCEPTS

•Computer systems organization~Architectures~Other architectures~Neural networks

KEYWORDS

Accelerators, Machine Learning, Inference, Recommendation Systems, Performance, Programmability

ACM Reference format:

Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, Adam Hutchin, Utku Diril, Krishnakumar Nair, Ehsan K. Ardestani, Martin Schatz, Yuchen Hao, Rakesh Komuravelli, Kunming Ho, Sameer Abu Asal, Joe Shajrawi, Kevin Quinn, Nagesh Sreedhara, Pankaj Kansal, Willie Wei, Dheepak Jayaraman, Linda Cheng, Pritam Chopda, Eric Wang, Ajay Bikumandla, Arun Karthik Sengottuvel, Krishna Thottempudi, Ashwin Narasimha, Brian Dodds, Cao Gao, Jiyuan Zhang, Mohammad Al-Sanabani, Ana Zehtabioskui, Jordan Fix, Hangchen Yu, Richard Li, Kaustubh Gondkar, Jack Montgomery, Mike Tsai, Saritha Dwarakapuram, Sanjay Desai, Nili Avidan, Poorvaja Ramani, Karthik Narayanan, Ajit Mathews, Sethu Gopal, Maxim Naumov, Vijay Rao, Krishna Noru, Harikrishna Reddy, Prahlad Venkatapuram and Alexis Bjorlin. 2023. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. In *Proceedings of 2023 International Symposium on*

Computer Architecture (ISCA'23), June 17-23, 2013, Orlando, FL. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589348

1 Introduction

Machine learning (ML) workloads have become ubiquitous in online activities. In recent years, these models have seen substantial growth in size and complexity, which has contributed towards their increased prediction accuracy and effectiveness. However, at the same time, this growth has presented significant challenges for the hardware platforms that are used for training and inference of these models at very large scales. Total Cost of Ownership (TCO) is one of the major constraining factors in launching models to production in the datacenter, and power is a significant component of TCO for these platforms. Therefore, performance-per-TCO (and performance-per-watt) has become an important metric for any hardware platform targeting these workloads.

Deep Learning Recommendation Models (DLRM) [16] have emerged as one of the most dominant workloads in Meta's datacenters [17][18]. These models combine traditional multilayer perceptron (MLP) operations (referred to as fully connected or FC at times) which are compute intensive, with embedding tables that transform sparse features into a dense representation. These tables contain wide vectors that are indexed randomly and are reduced to a single vector that is then combined with data coming from other layers to produce the final results [16]. While embedding table operations have rather light compute requirements, their memory footprint and bandwidth requirements are rather demanding due to the nature of the data access pattern and size of the tables.

Figure 1 shows the historical and estimated future growth in both complexity and memory footprint of the inference workloads related to recommendation models in Meta's production datacenters. The dashed line shows the estimated growth in the model's compute requirement while the solid lines demonstrate the increase in the memory footprint. The gray solid line captures the footprint of the device memory used to store embedding tables, which is an important component of these models. The level of growth in both compute and memory requirements is certainly an issue that needs to be addressed, especially considering how these workloads are typically run in the datacenter.

2 Motivation

Traditionally CPUs have been used as the primary vehicle to serve inference workloads in Meta's production datacenters, but they are not cost effective in keeping up with the demands of the most recent workloads. To that extent, hardware acceleration has been considered an attractive solution that can address power and performance issues and provide a more efficient way of serving inference requests while at the same time providing enough headroom in compute performance for running future models.

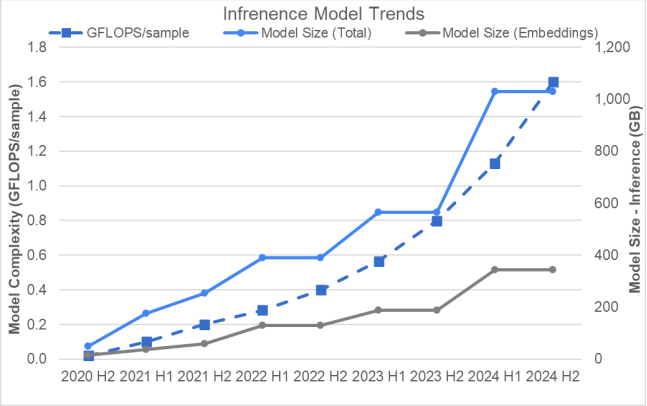

Figure 1: Scaling trends for inference models

Figure 2 shows the estimated number of servers that are deployed for serving inference workloads within the datacenter over the past couple of years. The light solid line shows the number of CPU-based servers, the dashed line shows the number of servers equipped with the first-generation inference accelerator, Intel NNPI [10], and the dark solid line shows the number of GPU-based servers [12]. While the initial demand for increased capacity was temporarily met using the NNPI accelerator, the requirements for the inference models quickly outpaced the NNPI capabilities and provided motivation for using GPUs. This brought the additional advantage of leveraging the existing ecosystem used already for training. Therefore, as can be observed, the increased demand in model complexity is served increasingly with GPUs as accelerators.

While recent generations of GPUs provide a lot of memory bandwidth and compute power, they are not designed with inference in mind, and therefore the efficiency of processing real inference workloads is low. Developers use a myriad of software techniques, such as operator fusion, shape specialization, graph transformations and kernel optimizations to raise the efficiency of GPUs. But despite these efforts, there is still an efficiency gap which makes it challenging and expensive to deploy models in practice.

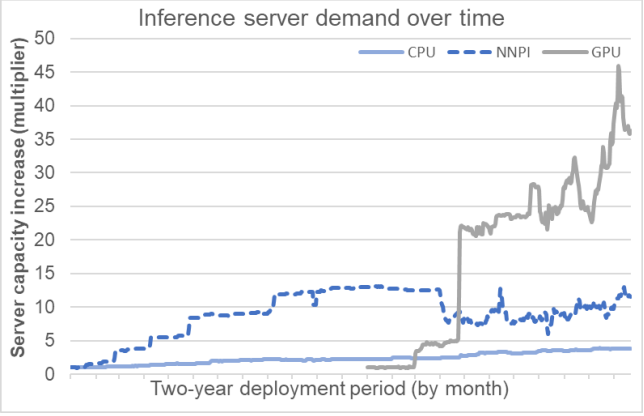

Figure 2: Growth in server demand for inference workloads

Given the experience deploying NNPI and GPUs as accelerators, it was clear that there is room for a more optimized solution for important inference workloads. This optimal solution is based on an in-house accelerator which is architected from the ground up to address the requirements of demanding inference workloads, specifically focused on meeting the performance requirements of DLRM systems. However, while focusing on DLRM workloads (given their ongoing variation and evolution and the fact that the architecture is effectively constructed for forthcoming generations of these workloads) it was also clear that in addition to performance, the architecture should also provide enough generality and programmability, to support future versions of these workloads and potentially other types of neural network models.

While creating a custom silicon solution opens the door for ample innovation and specialization towards the target workloads, creating an accelerator architecture for mass deployment in the datacenter is a monumental task. The focus and strategy when architecting the accelerator therefore has been on adopting and reusing suitable pieces of technology, as well as tools and environments, from vendors and the open-source community. This not only improves the time to market, but it also leverages the support and enhancements that come from the community and vendors and reduces the amount of resources required for building, enabling, and deploying such platforms.

The rest of this paper explains the undertaking of architecting MTIA, Meta's first accelerator chip targeting inference workloads, and the learnings that came with it. The next section details the accelerator's architecture and its various provisioned features and components. Section 4 goes over mapping an example operator to this architecture, demonstrating how various provisioned features are utilized to run the operator efficiently. Section 5 provides an overview of the accelerator's software stack and section 6 describes our evaluation methodology and results. Finally, section 7 discusses a few important lessons learned during this development cycle.

3 Accelerator Architecture

Figure 3 shows the high-level architecture of the accelerator, which is organized as an array of processing elements (PEs) connected on a grid. The grid is connected to a set of on-chip memory blocks and off-chip memory controllers through crossbars on each side. There is a separate control subsystem with dedicated processors and peripherals to run the system's control software. The host interface unit which contains a PCIe interface, associated DMA engines, and secure boot processor also sits alongside this control subsystem.

Figure 4 shows the internal organization of the PE. A PE consists of two RISC-V processor cores and associated peripherals (on the left), as well as several fixed function units specialized in performing specific computations or data movements (on the right). In addition, each PE has 128KB of local storage. A local interconnect establishes the connectivity between processors, their peripherals and custom hardware blocks.

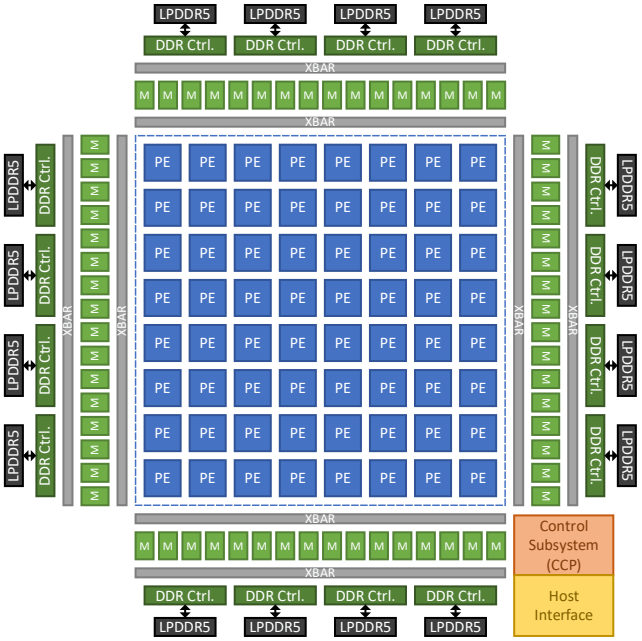

Figure 3: High-level architecture of the accelerator

3.1 Fixed Function Units

Each PE has a total of five fixed function blocks and a Command Processor which orchestrates and coordinates execution of operations on these fixed function blocks. Functional units form a coarse-grained pipeline within the PE, where data can be passed from one unit to the next to perform successive operations. Each functional unit can also access the data directly within the PE's local memory, perform the necessary operations, and write the result back, without passing the data to other functional units.

3.1.1 Memory Layout Unit (MLU)

This block performs operations related to copying and changing the layout of data in the local memory. It can operate on tensors with 4/8/16/32-bit data types. Operations like transpose, concatenation, or reshape are performed using this block. The output data can be sent to the next block directly to be operated on immediately or can be stored in PE's memory. For example, MLU can transpose a matrix and provide the output directly to DPE block for a matrix multiplication operation, or it can format the data properly as part of the depth-wise convolution operation and send it to DPE to perform the actual computation.

3.1.2 Dot-Product Engine (DPE)

This block performs a set of dot-product operations on two input tensors. The first tensor is read and stored within the DPE first, then the second tensor is streamed through the block and a dot product operation is performed with all the rows of the first

tensor. DPE can perform 1024 INT8 multiplications (32×32) or 512 FP16/BF16 multiplications (32×16) per cycle. Operations are fully pipelined; performing multiplication of two maximum size matrices takes 32 clock cycles. In case of INT8 multiplication, the resulting output is stored in INT32 format, while in the case of BF16 or FP16 multiplications, the result is stored in FP32 format. The result is always sent to the next functional unit in the pipeline for storage and accumulation.

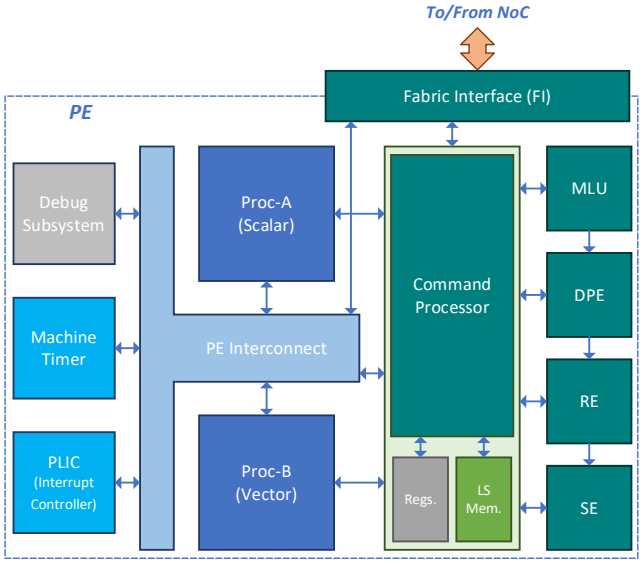

Figure 4: PE's internal organization

3.1.3 Reduction Engine (RE)

The reduction engine hosts the storage elements that keep track of the results of the matrix multiplication operations and accumulates them over multiple operations. There are four separate storage banks that can be independently used to store and accumulate the results coming from DPE. RE can load an initial bias into these accumulators and can also send their contents to neighbor PEs over a dedicated reduction network (discussed later in this section). Upon receiving results over the reduction network, RE accumulates the received values on top of the values in one of the local storage banks. It can then send the result to the next neighbor, to the SE, or store it in the PE's local memory directly.

3.1.4 SIMD Engine (SE)

This block performs operations like quantization/de-quantization and nonlinear functions. Internally the block contains a set of lookup tables and floating-point arithmetic units to calculate linear or cubic approximation of nonlinear functions such as exponentials, sigmoid, tanh, etc. The approximation accepts INT8 or FP16 data types as inputs, producing an INT8 or FP32 result at the output. The unit can receive its inputs directly from the RE block or read them from the local memory. In addition, this block is also capable of using its floating-point ALUs to perform a set of predefined elementwise operations, such as addition, multiplication, accumulation, etc.

3.1.5 Fabric Interface (FI)

This block acts as the gateway in and out of the PE. It connects to and communicates over the accelerator's on-chip network. It formulates and sends memory access requests to on-chip and off-chip memories, as well as system registers, and receives back the data or write completions. It implements a set of DMA-like operations that transfers the data in and out of PE's local memory. It also receives and transmits cache misses and un-cached accesses from processor cores and allows other entities (other PEs or the control subsystem) to access the PE's internal resources.

3.1.6 Command Processor (CP)

In addition to hosting PE's local memory and registers, the CP block acts as the central processing unit that orchestrates execution of various operations on the fixed function blocks concurrently. It receives instructions from the two processor cores in the PE, performs dependency checking, scheduling, and tracking for those instructions, and dispatches them to the fixed function units for execution. It contains two separate schedulers (one for each processor core), a set of command queues, as well as arbitration logic for accessing the local memory and register resources.

The hardware provides a set of basic atomic primitives to allow synchronization between the cores (within the PE or across multiple PEs). These primitives are enacted by processors, which allows atomic update to predefined registers, and can stall the processor until certain conditions are satisfied externally (e.g., a counter reaches a certain value). At the higher level, these mechanisms are used for efficient implementation of software constructs such as locks, ticketing locks, mutexes and barriers. The logic that performs the atomic operations as well as the relevant registers reside within the Command Processor and are tightly integrated with the processor cores through custom interfaces.

3.2 Processor Cores

Each PE contains two RISC-V cores that run the application's code and issue commands to the CP for offloading various computations to fixed function units. The cores are single issue, in-order cores, with a five-stage pipeline (AX25-V100, from Andes Technology), and are heavily customized to suit the functionalities needed. The set of customizations includes custom interfaces, custom registers, custom instructions, and custom exceptions. Custom interfaces connect cores to the CP to issue commands to fixed function units and move data back and forth between cores and local memory. Custom registers store the command information that is sent to the CP upon issuing commands. Custom instructions are added to start the desired operation on each of the fixed function units. And finally custom exceptions ensure correctness of each command issued to the CP and raise an exception in case of illegal values in the command.

One of the processor cores is equipped with the RISC-V vector extension, which adds extra flexibility to the PE and allows implementing operations that do not map well to the existing fixed function units. The vector processing unit contains 32 vector

registers, each 64B wide and has the same width for all vector functional units. It implements version 0.8.1 of the RISC-V vector extension [23].

3.3 Local Memory (LS)

Each PE has total of 128KB of local memory to be used by processors and functional units. The CP implements an arbitration scheme for memory banks and coordinates accesses from cores and fixed function units. Local memories are mapped to the system's address space and can be accessed by cores via regular load/store instructions.

There is an abstraction layer introduced on top of the local memories to simplify usage and dependency checking between operations that use them. This can be considered as further extension of the concept of the buffet [1][2]. Each PE can define circular buffers (CBs) that are mapped to the existing local memory. Each CB is designated with an ID and has a pair of registers that specify its size (depth) and starting address in the local memory. In addition, each CB also implements a set of read and write pointers to implement a hardware FIFO.

In a CB, read operations always read the data starting from the read pointer and write operations always write data starting from the write pointer. Like buffets, read and write operations carry an offset which allows them to access a location other than the current head or tail of the buffer (Figure 5). Fixed function units use the CB IDs as their input/output operands; for example, a matrix multiplication operation uses two CBs as its input operands. Before allowing an operation to start, the Command Processor checks the availability of the data in the input CBs and space in the output CB. It allows the operation to start only if the necessary element and space checks pass. Therefore, an operation is guaranteed to have the necessary resources to complete and will not stall the functional unit in the middle of its execution.

The Command Processor also uses the CB IDs to enforce dependency checks and interlocks between different custom instructions. It ensures that operations that access and modify a particular CB are always executed in program order, while operations that operate on different CBs or different regions of the same CB can execute in parallel. This significantly simplifies the dependency checks as opposed to using absolute local memory addresses for enforcing such interlocks.

CBs also simplify realization of the producer-consumer execution model between different operations. These operations can be initiated by different cores or different fixed function units. For example, a program can issue a series of DMA operations to the hardware (which moves the data from an external memory into a CB), following it up with a set of custom compute operations (e.g., MATMUL) that uses that data, without requiring an explicit synchronization between the two. The MATMUL instruction is automatically stalled by the Command Processor until enough data is brought into the CBs by prior DMA operations, and is started immediately afterwards, relieving the program from explicitly checking the availability of the data.

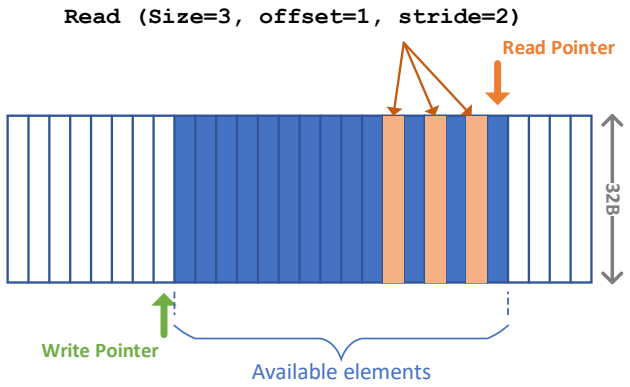

Figure 5: Reading from a Circular Buffer

While some instructions like DMA operations automatically adjust the read and write pointers (as they move the data in and out of the CBs, and hence produce or consume elements), other custom instructions do not move the pointers. This allows data inside the CB to be reused multiple times by different operations before it is explicitly marked as consumed. Hardware provides additional custom instructions that can adjust both read and write pointers in each CB, allowing explicit marking of the data elements as produced or consumed, when necessary.

3.4 Memory Subsystem and Interconnect

In addition to the local memory within the PEs, the accelerator also has 128MB of on-chip SRAM, organized as slices around the grid. This on-chip memory can be used as addressable scratchpad memory, or as a common, shared, memory-side cache. There are four LPDDR5 controllers on each side of the grid, providing a total of 176 GB/s (theoretical) off-chip bandwidth. The accelerator can support a total of 128GB of off-chip memory capacity. Memory addresses are distributed across these controllers, and among the on-chip SRAM slices. When on-chip SRAM is configured as cache, each four cache slices are associated with a single memory controller and cache its addresses.

The on-chip network that connects all the PEs and memories together is based on the AXI interconnect with special enhancements. The interconnects consist of two networks for carrying memory and register accesses separately. The memory access network is equipped with a multicast feature which allows coalescing of requests from multiple PEs into one (if they are made to the same set of addresses). A single request is then sent to the memory blocks to retrieve the data and return it to all requesting PEs. Multicast is only supported for the PEs that are located along the same row or column in the grid however, and cannot be used for an arbitrary group of PEs.

In addition to the main AXI based interconnect, PEs are also connected to each other via a specialized network, called the

reduction network. This is a unidirectional network that travels only from north to south and from west to east. It carries partial sums from the accumulators in the RE block of one PE to another. Using this network, PEs can expediently accumulate the result of their computation without having to save and restore it in memory. The last PE in the row or column can then store the final result in the memory, after all partial values are accumulated.

3.5 Parallelism and Data Reuse

Parallelism, locality, and data reuse play a significant role in efficient utilization of limited hardware resources in any deep learning accelerator. MTIA architecture has provisioned a set of features to allow multiple degrees of parallelism and maximal exploitation of temporal and spatial data reuse in neural network models and operators, as discussed below.

Parallelism: The architecture provides support for multiple levels of parallelism and overlapping of various operations. Data level parallelism (DLP) is exploited by usage of wide vectors in fixed function units as well as the vector processors. Multiple PEs also can operate on the same task in a data parallel manner. Instruction level parallelism is exploited in the Command Processor, by allowing multiple outstanding operations to be handled by different fixed function blocks simultaneously. Memory level parallelism (MLP) is achieved by allowing many outstanding requests to on-chip and off-chip memories from each PE. And finally, thread level parallelism (TLP) can be achieved by utilizing multiple PEs (or groups of PEs) to run parallel threads, as well as by having two independent threads within each PE. Threads within the PE can cooperate in performing a given task, by one thread orchestrating the data movement and the other one orchestrating the computation.

Caching: There are multiple levels of caching in various blocks of the hardware to improve locality and reduce memory bandwidth consumption. This includes instruction and data caches in the processor cores, large on-chip last level cache, and caching for input operands in the DPE block. The caching at the DPE level allows the engine to hold data from both operand A and operand B and save access to local memory upon hit.

Circular buffers / local memories: Circular buffers provide the storage for holding input operands while the PE performs the computations. Flexibility in adjusting pointers as well as offsetting into any location within a circular buffer allows the program to access each line of data multiple times, before deciding to mark it as consumed.

Specialized reduction: Having a dedicated reduction network not only offloads a large part of data transfer from the system's main on-chip network, but also provides a way for grouping PEs together and using their local memories in an aggregate form. This in turn allows storing a larger portion of input operands in the PEs and reducing the bandwidth requirement for loading them from off-chip memory. In addition, the DPE block utilizes reduction trees (spatial sum) to calculate the output of a multiplication operation [1][3][4], which is known to be more energy efficient [5].

Multicasting: As mentioned earlier, the system's NoC allows coalescing requests from multiple PEs when they access the same set of addresses in memory. This reduces memory bandwidth and increases the energy efficiency of data movement by allowing the data to be shared while reading it from memory only once and delivering it to all requesters [1][6][7][8]

Figure 6 shows the die plot with the grid of PEs, surrounded by on-chip SRAMs and off-chip DDR controllers, while Table I lists the summary of the chip features and parameters.

Table I - Summary of MTIA features and parameters.

Parameter	Value
Technology	TSMC 7nm
Frequency	800MHz nominal (1.1 GHz max)
Instances	1.12B gates, 65M flops
Dimensions	19.34 × 19.1mm (373 mm^2)
Package	43 × 43, ~2800 pins
TDP	25 W
Voltage	Dual rail: 0.67V (logic), 0.75V (memories)
Host Connectivity	8× PCIe Gen4 (16 GB/s)
GEMM TOPS (MAC)	102.4 (INT8)
	51.2 (FP16)
SIMD TOPS	Vector: 0.8 (FP32) / 1.6 (FP16) / 3.2 (INT8)
	SE: 1.6 (FP16) / 3.2 (INT8)
Memory Bandwidth	Local memory: 400GB/s per PE
	On-chip SRAM: 800GB/s
	Off-chip DRAM: 176 GB/s
Memory Capacity	Local memory: 128KB per PE
	On-chip SRAM: 128MB
	Off-chip LPDDR5: 64GB (16 channels)

4 Mapping an FC Layer

To demonstrate how all the above-mentioned features work together, let's consider an FC operator that performs a matrix multiplication operation in the form of $CT = A \times B^T$ and see how it maps to a sub-grid of PEs. The reason for performing the operations in a transposed manner is to keep k as the inner dimension for both tensors, to increase the efficiency of memory accesses. Matrix A is assumed to be m×k and matrix B is assumed to be k×n (hence B^T will be n×k), producing output C which will be an m×n matrix (or C^T being an n×m matrix). Inputs are assumed to have row major memory layout. When the inner dimension (k) is not a multiple of 32B, the outer dimension (m or n) stride is aligned to 32B boundaries for efficient data movement. For simplicity, we will assume that all elements are of INT8 data type.

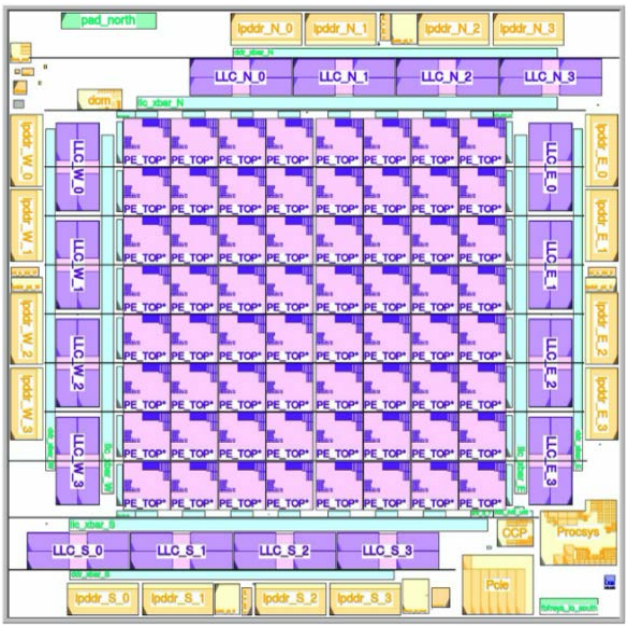

Figure 6: MTIA die plot

As mentioned earlier, DPE works on blocks of 32(m)×32(k)×32(n) inputs, generating 32(n)×32(m) partial results accumulated in the RE. This operation takes 32 clock cycles. In order to feed the DPE's pipeline, 32(m)×32(k) blocks of matrix A and 32(n)×32(k) blocks of matrix B^T must be brought from external memory into PE's local memory in 32 cycles, requiring 64B/cycle of bandwidth. To alleviate this bandwidth pressure, the four accumulators in the RE block are used to accumulate 2×2 blocks of partial results, holding a total of 64(n)×64 (m) elements of the output matrix. By using the accumulators in this manner, we use every 32×32 input block twice, hence reducing the external bandwidth requirement to 32B/cycle.

Tensor dimensions m, n and k are distributed in multiples of 64, 64 and 32 across the PE grid respectively. Each PE hence works on a different sub-block of the larger result matrix in a data parallel fashion. The reduction dimension (k) is distributed over multiple PEs along the row (or column). This facilitates the usage of the reduction network to accumulate partial results after multiplication is completed. PEs pass the calculated partial results to each other to accumulate and pass to the next PE. When two or more PEs along a given row or column use the same block of input data from either input matrix, the multicast feature of the on-chip network is used to coalesce the requests from multiple PEs and send a single request to the memory, further reducing memory bandwidth requirements.

Figure 7 shows an example of distributing an FC operator with dimensions of 512(m), 1024(k) and 256(n) on a 4×4 PE sub-grid. The reduction dimension (k) is distributed across two PEs along the same row and dimension m is distributed across four rows. PEs in columns 0 and 2, and PEs in columns 1 and 3 participate in row multicast-read of matrix A. Similarly, all PEs in each column participate in column multicast-read of matrix B^T.

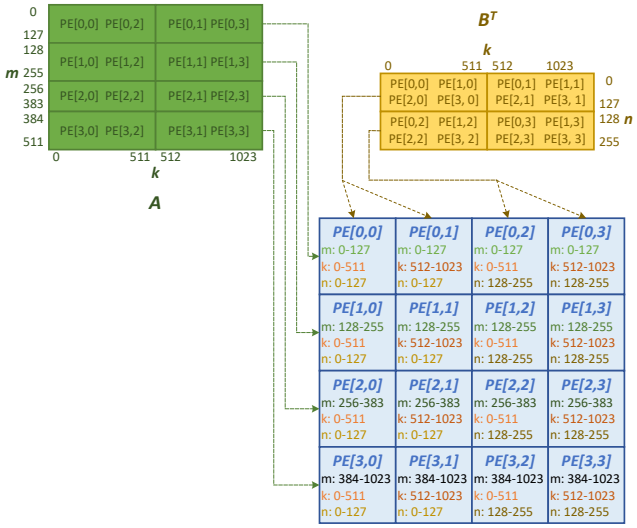

Figure 7: Mapping an FC operator to a sub-grid

Within the PE, the operation is divided between the two cores in a producer-consumer manner. Figure 8 shows the pseudocode corresponding to each of the cores in the PE. Core0 issues a set of DMA operations that move data from main memory into CB_A and CB_B, used to store matrices A and B locally. In a parallel thread, Core1 issues a set of matrix multiply (MML) instructions that reads data from CB_A and CB_B respectively and stores the results in an accumulator register. As can be observed, each block of data is used twice to produce a partial result in each of the accumulator registers. If the operation is the last iteration, the data is marked as consumed in the CB by issuing a POP instruction, otherwise the corresponding CB offsets are incremented to move to the next block of data in the next iteration. At the end, the reduction operation (REDUCE) is called to accumulate all partial sums across PEs. The last PE in the reduction chain sends the data back to main memory using the DMA operation.

The two cores in the PE must synchronize at the start of the operation as only one of them performs the necessary initialization tasks (e.g., setting up the CBs to use). But afterwards, there is no explicit, per iteration synchronization; the producer-consumer synchronization is taken care of by the hardware: If the consumer (the MML operation) attempts to use a CB that does not have enough data, hardware stalls the operation until the producer (DMA operation) places enough data within the CB, at which point it allows the matrix multiplication to proceed. This asynchronicity decouples the producer and consumer threads and allows the producer to move ahead and bring in more data for later iterations.

```
#-------------------------------Core0-----------------------------------
work = GetWorkForMyPE(...)
INIT CB_A, CB_B and CB_C                        # Setup circular buffers
multicast_A, multicast_B = JoinMulticastGroup(...)
Sync(...)                                       # Synchronize with others
read_B = true
for m in range(work.m.begin, work.m.end, 64):  # For every row of "A"...
 read_A = true
 for n in range(work.n.begin, work.n.end, 64): # ...read entire "B"
  for k in range(work.k.begin, work.k.end, 32):
   if read_A:
    DMA GetAddr(A, (m, k)), size=(64,32), CB_A, multicast_A
   if read_B:
    DMA GetAddr(B, (n, k)), size=(64,32), CB_B, multicast_B
  read_A = false
 read_B = false
#-------------------------------Core1-----------------------------------
work = GetWorkForMyPE(...)
Sync(...)                                       # Synchronize with others
for m in range(work.m.begin, work.m.end, 64): # For every two chunks of "A"
 cb_offset_B = 0
 for n in range(work.n.begin, work.n.end, 64):# Multiply two chunks of "B"
  cb_offset_A = 0
  INIT RE acc with 0                            # Initialize accumulators
  for k in range(work.k.begin, work.k.end, 32):
   MML acc=0,size=(32,32,32),CB_B,CB_A,cb_offset_B      ,cb_offset_A
   MML acc=1,size=(32,32,32),CB_B,CB_A,cb_offset_B      ,cb_offset_A+32*32
   MML acc=2,size=(32,32,32),CB_B,CB_A,cb_offset_B+32*32,cb_offset_A
   MML acc=3,size=(32,32,32),CB_B,CB_A,cb_offset_B+32*32,cb_offset_A+32*32
   if ((m + 64) >= work.m.end):                 # If last Iteration...
    POP CB_B, size=2*32*32                      # ...mark "B" data as consumed
   else:                                        # Otherwise...
    cb_offset_B += 2*32*32                      # ...proceed to the next chunk
   if ((n + 64) >= work.n.end):                 # If last Iteration...
    POP CB_A, size=2*32*32                      # ...mark "A" data as consumed
   else:                                        # Otherwise...
    cb_offset_A += 2*32*32                      # ...proceed to the next chunk
  REDUCE destination = neighbor PE or CB_C, size=(64,64))# Send to next PE
  if IsLastPEInReduction(...):                  # If last PE in sequence
   DMA PutAddr(C, (n, m)), size=(64, 64), CB_C # Write result to memory
```

Figure 8: Pseudocode for the FC operator running in PE

5 Software Stack

The software stack for MTIA is designed with two main goals in mind: be efficient for production, meaning achieve higher perf/TCO than other best-in-class solutions, and at the same time, be simple and straightforward to use, even simpler than available alternatives. The software stack for MTIA is designed and built around PyTorch to benefit from its capabilities and to achieve a seamless integration with other components of the ML infrastructure available in a production environment. The rest of this section provides an overview of each component of the software stack as shown in Figure 9.

ML serving platform: At the top of the software stack, we have production-specific ML model serving platforms (Application Layer as illustrated in Figure 9). These serving platforms are operating on top of PyTorch and are mostly hardware agnostic, supporting execution on heterogeneous hardware systems including CPUs, GPUs, and accelerators like MTIA.

PyTorch Runtime: A PyTorch Runtime integration for MTIA was developed which provides necessary functionality and features including MTIA Tensors, a host-side memory allocator, and CUDA-like streaming APIs for scheduling the desired operators to execute on the device. The runtime supports different modes of model execution, including eager mode, as well as full graph compilation and execution to maximize performance. It also supports running models split into partitions spanning multiple cards, providing the necessary synchronization and communication channels between them.

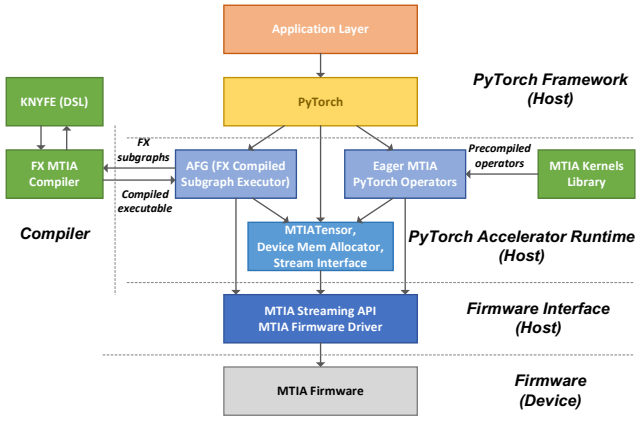

Figure 9: MTIA's software stack

Compilers: The next important component in the software stack is a set of compilers which consists of multiple parts:

• A PyTorch FX-based ML model compiler which applies several transformations and model-level optimizations to the PyTorch graph represented as FX IR [19][20], and gradually converts it into LLVM IR [21][22]. It is responsible for graph optimizations which take advantage of the PE grid and MTIA's memory subsystem. It implements a tensor placement scheme that takes a best-effort approach to keep producer-consumer data in on-chip memory. It can also split a model into sub-graphs intended to run across multiple cards and even across sub-grids within the same chip.

• A DSL-based compiler (codename KNYFE) for ML kernel development, which takes a short high-level description of an ML kernel and produces low-level optimized C++ code. It uses low-level hardware specific APIs to implement the ML operator and is used extensively for developing many of the ML kernels used in MTIA.

• LLVM-based compiler toolchain which converts LLVM IR into an executable for the device. LLVM is used primarily due to the RISC-V support it provides and is responsible for the lowest level of optimizations like register allocation, in-lining and code generation. Most major optimizations like tiling or scheduling of the work and data among PEs are performed by the higher-level compilers mentioned earlier.

Library of ML kernels: Another important component is the library of kernels and ML operators that are used to construct the ML models executing on the device. Many of these kernels are developed using the DSL compiler mentioned earlier, but some of the most performance demanding kernels, e.g., fully connected (FC) layers and embedding bag (EB) layers, are developed by experts directly in low-level C++ using exposed intrinsics to ensure they can achieve the highest levels of performance possible on the hardware.

Host driver and firmware interface: MTIA platform software enables the host to access the accelerator device. It manages the device lifecycle and resources, and it helps initiate and track runtime operations on the device. This part of the stack

is broadly split into two parts: the host software and device firmware. The host software consists of the Linux device driver, a device access library for providing a uniform device interface, and a streaming API to interface with PyTorch, as well as software tools and utilities for managing and monitoring the device.

Device firmware: The device firmware includes a ROM based pre-boot firmware, secure boot firmware running on its own processor, the Control Core Processor firmware running on the control subsystem performing runtime and management operations, and finally the PE monitor that runs on the PEs in the compute grid, which schedules and monitors workloads running on the PEs. The main control firmware is based on the Zephyr Real Time OS [9].

6 Results

We evaluate the performance of the MTIA by comparing it against a baseline accelerator (NNPI) [10] and against more recently deployed GPUs. It should be noted that we report the results collected with an under-development software stack, as we believe this reflects the end-to-end performance and is representative of a production environment. However, this stack is not currently as optimized as the GPU's software stack. Consequently, there are cases where the GPU is more efficient, but we are hoping to close this gap over time and have the MTIA software stack deliver the full gains of the architecture across all the DLRM workload space. We evaluate both operator-based benchmarks as well as full DLRM models varying in complexity, size, and accuracy. Since these accelerators are all based on different hardware platforms, we first compare their system level hardware specification (Table II). These platforms are the following: Yosemite V2 server with six NNPI accelerator cards [11], Zion4S server with eight Nvidia A100 GPUs [12], and Yosemite V3 server [13] with twelve MTIA accelerator cards.

Table II - Inference hardware platforms

Metric		Yosemite V2 (6 NNPI)	Zion4S (8 GPU)	Yosemite V3 (12 MTIA)
Power	System	298 W	4500 W	780 W
	Card	13.5 W	330 W	35 W
	Percentage	27.2 %	58.7 %	53.8 %
Compute	INT8 (TOPS/s)	50 × 6	624 × 8	104 × 12
	FP16 (TF/s)	6.25 × 6	312 × 8	52 × 12
Memory	Type (device)	LPDDR	HBM	LPDDR
	Size (device)	16 GB × 6	40 GB × 8	32 GB × 12
	BW (device)	50 GB/s × 6	1.5 TB/s × 8	150 GB/s × 12
	Size (host)	64 GB	1.5 TB	96 GB
	BW (host)	50 GB/s	400 GB/s	76 GB/s
Comms.	Dev.-to-Dev.	PCIe	NVLink	PCIe
	P2P BW (card)	3.2 GB/s	80 GB/s	12.8 GB/s
	NIC BW	50 Gbps	400 Gbps	100 Gbps

While we can compare the absolute performance of MTIA versus NNPI and GPUs, each device has different capabilities in terms of compute throughput, memory bandwidth, and memory capacity. They also operate under different power budgets. Therefore, in our study we report perf/W (as a proxy for perf/TCO, given the sensitive nature of TCO), because power is an important factor in provisioning for deployment in the datacenter. We use the total platform power divided by the number of accelerator cards to determine power provisioned for each accelerator, as opposed to using the maximum TDP for the card.

6.1 Benchmark Performance

We first evaluate the performance of several important operators and kernels that push the limits of the architecture and are representative of main components in production DLRMs. Table III shows the latency breakdown of a request in a representative DLRM with batch sizes of 64 and 256. The model has approximately 750 layers with nearly 550 consisting of EB operators. For batch size of 64, FC dominates the execution time followed by EB, while for batch size 256, EB dominates FC slightly and the two together account for 62% of the execution time. It should be noted that with larger input shapes, the kernels are able to better amortize the setup costs, and reuse the data more, hence achieving higher utilization of the fixed-function units in the hardware.

Table III - Operator breakdown, medium complexity DLRM

Operator	Batch size 64	Batch size 256
FC (Fully Connected)	42.10 %	32.4%
EB (Embedding Bag)	31.19 %	30.0%
Concat	2.86 %	11.5%
Transpose	8.47 %	5.9%
Quantize	1.55 %	5.3%
Dequantize	2.94 %	3.3%
BatchMatMul	3.30 %	1.7%
Others	7. 59 %	11.0%

Based on the breakdown, we use a set of benchmarks to assess the efficiency of the MTIA's hardware. While not full-fledged workloads, these benchmarks allow exercising various shapes and sizes for important operators (including corner cases) and shed light on the potential deficiencies that might exist in the hardware. GemmBench [14] is used to evaluate dense computation; it creates a model composed of a chain of FC layers. In our benchmark runs we focus on both FP16 and INT8 (quantized) data, which requires additional quantize and dequantize layers. TBEBench [15] is used to evaluate sparse computation, and allows us to configure the batch size, number of tables, number of rows per table, embedding dimension, and pooling factor of TBE operators. BatchGEMMBench [24], ConcatBench [26], and

TransposeBench [26] are used to efficiently cover other significant operators typically seen in recommendation models. We also evaluate several elementwise kernels including quantize, dequantize, and tanh.

Dense computation: We evaluate both INT8 and FP16 Fully Connected (FC) layers (Figure 10 and Figure 11). When accuracy is sufficient, INT8 quantization unlocks a potential 2x improvement in FC throughput. For the set of shapes we evaluate, the trend lines roughly track for MTIA and the GPU across INT8 and FP16, indicating that the software implementations are well optimized across a range of arithmetic intensities. In many cases, MTIA achieves 2x or greater performance per Watt, and is particularly effective for low batch sizes which helps when serving requests under stringent latency requirements. For large batch sizes, the GPU is able to achieve higher utilization with the increased amount of work so the perf/W gains of MTIA are lower. Note that MTIA is most efficient when tensors can be streamed directly from SRAM, which means that graph optimizations and managing data locality are very important for good performance at the model level.

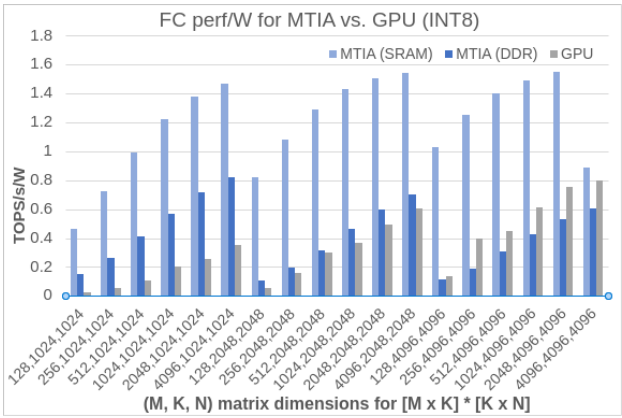

Figure 10: INT8 FC performance

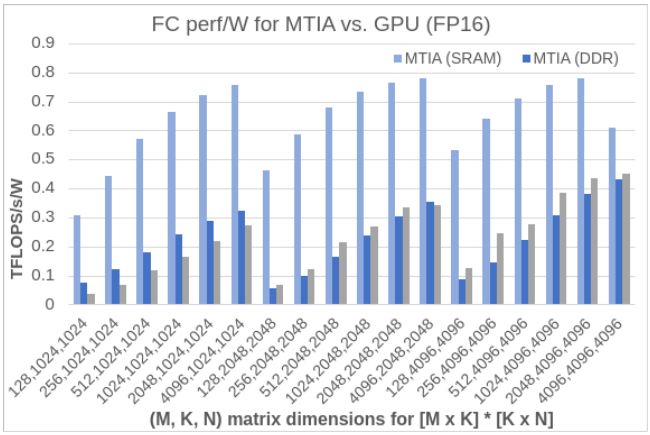

Figure 11: FP16 FC performance

Sparse computation: While a typical recommendation model might include hundreds of EmbeddingBag (EB) operators, they can be merged together into one or more TableBatchedEmbedding (TBE) operators to amortize kernel launch overhead and increase the work that can be parallelized across the device. Figure 12 shows the performance (in GB/s/W) for the TBE benchmark running on MTIA and GPU for a set of representative operator shapes. Note that we report performance in terms of GB/s here because this benchmark is mostly memory bound, and measuring bandwidth as opposed to lookups/sec provides better insight into hardware utilization. Here we utilize the cache configuration of the on-chip SRAM to take advantage of locality across and within batches. In these examples, all table entries use 8-bit quantization and the triplets shown in the graph describe the operator's pooling factor, number of rows in the table, and the embedding dimension (elements per row). MTIA achieves between 0.6x to 1.5x the perf/W of the GPU with the current kernel implementation.

Given the evolving nature of the software stack, we observe that there is significant headroom for improvement: MTIA is reaching just 10-20% of its memory bandwidth whereas the GPU is achieving about 60% of its HBM bandwidth. To ensure that there are no deficiencies in hardware, we used hand-written kernels developed for RTL validation, and could observe performance levels as high as 500 GB/s (more than 60% of roofline) or 6 GB/s/W given sufficient locality in the SRAM. We hope to close this gap by improving the software pipelining and instruction scheduling of the TBE kernels.

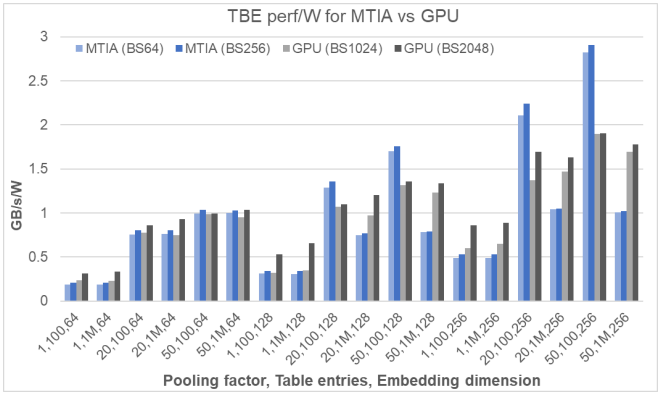

Figure 12: TBE performance

Other operators: While FC and TBE tend to dominate execution time, we found that other operators can be just as important, especially given how much effort is spent optimizing the former. We evaluated BatchMatMul, Concat, Transpose, and several elementwise kernels for M=256, K=128, N=32, with tensor data placed in SRAM and DRAM (Figure 13). These operators tend to be memory bound which is exemplified by BatchMatMul and Tanh, which reach more than 90% and 80% of the SRAM bandwidth, respectively. When data is placed in the DRAM, the efficiency drops down to around 40% on average, because it is more difficult to hide the additional memory latency. We believe implementation of data placement optimizations,

operator fusion, and minimizing data fetch from DRAM could potentially mitigate this issue.

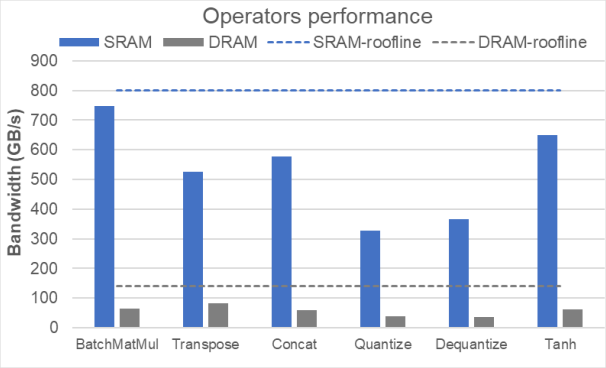

Figure 13: Performance of other operators

6.2 Model Performance

We examine the performance of five different representative DRLMs, described in Table IV, which range from low to high complexity. MTIA can run the same recommendation models that run on NNPI and GPU. With the current level of maturity of the software stack, MTIA achieves near perf/W parity with the GPU and exceeds the perf/W of NNPI, while roofline modeling indicates there is significant room for improvement as the software stack matures further.

Table IV - DLRM models used for evaluation.

DLRM Model	Size (GB)	Complexity (GFLOPS/batch)
Low Complexity 1 (LC1)	53.2	0.032
Low Complexity 2 (LC2)	4.5	0.014
Medium Complexity 1 (MC1)	120	0.140
Medium Complexity 2 (MC2)	200	0.220
High Complexity (HC)	725	0.450

Figure 14 shows the performance (in TFLOPS/s/W) across the above-mentioned set of DLRMs. Compared to NNPI, MTIA achieves 1.6x higher efficiency while compared to GPU, it reaches 0.9x efficiency. There are two important factors to consider in these results: the model characteristics and the level of software optimization in the implementations. For low complexity models, MTIA has a significant advantage over the GPU because these models are dominated by FC layers with smaller input shapes and MTIA handles this quite efficiently, e.g. LC2 shows nearly a 3x improvement. For medium complexity models, MTIA still sees an efficiency gain over the GPU, but it is lower because FCs are less dominant, and the GPU software stack provides more efficient implementations of other operators (with TBE and aggressive operator fusion). For high complexity models, we see that the GPU software stack is better optimized for large shapes, and MTIA needs similar optimizations in order to achieve the same or higher levels of efficiency. These initial results give us insight into areas of the software stack that we should consider focusing on in the future (e.g., large FCs, TBE optimizations, operator fusion, etc.), as well as provide important learnings for next-generation architecture which we discuss next.

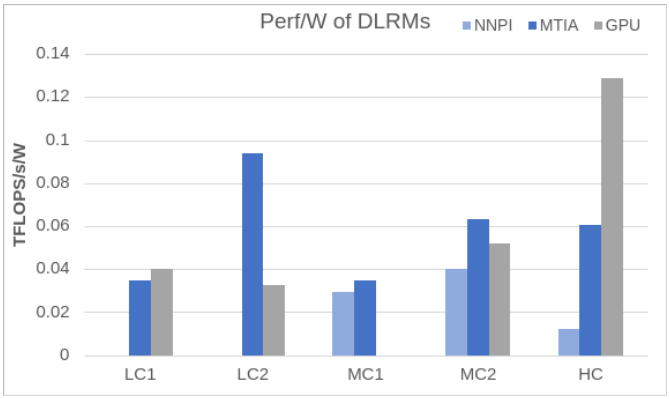

Figure 14: Performance of DLRMs

7 Discussion

Building silicon is always a difficult, lengthy, and time-consuming process, especially when done for the first time. For MTIA, the resulting silicon needed to achieve high performance, handle a wide range of recommendation models, and provide a level of programmability that would allow rapid deployment of models in production. This section highlights our important observations and reflections regarding architectural choices, and how they impacted the software stack, performance, and developer efficiency. These lessons also act as guidance for improving and enhancing future generations of architecture.

Dual-Core PEs: The choice of having two separate processor cores within the PE and allowing both to control the fixed function units provided a great degree of parallelism and flexibility at the thread level, allowing decoupling of compute from data transfer. While this decoupling simplified the programming and alleviated performance issues when a particular operator is instruction bound (by providing twice the overall instruction throughput), using both cores correctly and efficiently in software took some effort. Details such as synchronization between the two cores for initialization and clean up before execution of a job were difficult to get right the first time, but afterwards were leveraged in all workloads through proper integration in the software stack.

General-Purpose Compute: Addition of general-purpose compute in the form of RISC-V vector support proved to be a judicious choice: There were operators which were developed or gained importance after the architecture definition phase, and hence the architecture did not include any offload support for them. Operators like LayerNorm and BatchedReduceAdd were

straightforward to implement with vectors, and these implementations proved superior to versions using scalar cores and fixed function units.

Automated Code Generation: Some of the architectural choices made regarding how the fixed function units are integrated and operated in the PE have made the automatic code generation by compiler difficult. Processor cores must assemble and issue explicit commands to operate any of the fixed-function blocks. While this is done through addition of custom instructions and registers to the processors, it still requires assembling and passing many parameters to each offload engine to specify the details of the operation. Controlling a heterogenous set of fixed-function units from within the program and balancing the data flow between them is a challenging problem for the compiler. Achieving desired levels of utilization on the fixed-function blocks across various input shapes and sizes is also difficult. While our DSL-based KNYFE compiler makes it easier to write kernels and handles many of these issues automatically, it requires learning a new DSL.

Circular Buffers: Addition of the circular buffer abstraction greatly simplified the dependency checking between custom operations that work on the same region of memory, as the circular buffer IDs were used as units of dependency checks (similar to register IDs in the processor cores). They also simplified the implementation of the producer-consumer relationship between fixed function blocks and processors, as the hardware holds off the operation until enough data (or space) is available in the circular buffer without any need for explicit synchronization at the software level. The flexible addressing mechanism also allows arbitrary access to any location within a circular buffer, which simplifies data reuse as different operations can access different segments within the circular buffer multiple times. However, this requires software to explicitly manage the space within the buffer and decide when the data should be marked as consumed, which might create difficult to debug correctness issues if not performed properly.

Memory Latency: Both PE and on-chip SRAM memories turned out to have longer than typical access latencies. Having lots of clients accessing the PE memory complicated the arbitration scheme and added latency cycles. For fixed function blocks, this latency gets rolled into the operation's latency, but when processors try to use the local memory, software must resort to techniques such as unrolling and software pipelining to hide latencies, which increases the register pressure. This becomes exacerbated when developing kernels on the vector processor, as it limits the amount of register grouping that can be performed and hence increases the dynamic instruction count.

Placement of the on-chip SRAMs over the perimeter, while evenly distributing requests over multiple slices, created a large degree of non-uniformity in memory access latencies. Since the requests are always completed after the last piece of data arrives, the overall latency gets impacted when making larger than minimum requests. While in most cases this latency could be hidden by prefetching data into the PE's memory, in cases such as EmbeddingBag operators with small pooling groups the latency gets exposed because the memory access pattern is not known in advance and there are not enough outstanding requests to hide the latency.

Cache Coherence: While the system implements a shared memory paradigm, there is no hardware support for cache coherency. In this shared memory system, inter-PE coherency is not required as different PEs operate on their dedicated part of the dataset in a data parallel manner. However, intra-PE coherency sometimes causes correctness issues: If the same set of memory addresses are touched by the two processor cores, or by the fixed-function units and a core, or addresses are reused by the same core across different operators, the cached copies of these addresses must be explicitly flushed from caches, otherwise the stale data could be used.

Architecture Hierarchy: Recommendation models for the accelerator vary greatly in size and complexity in the layers and operators they employ. While large layers when mapped on the PE grid can extract desired utilization level from available hardware resources and amortize the overhead of job creation and dispatch, the smaller layers or lower batch sizes have to resort to techniques such as exploiting sub-graph parallelism and fusion to get to the same level of utilization. Even though there is plenty of room at the software level to perform such optimizations or reduce the overheads of deploying jobs, we believe some provisioning at the architecture level would have made addressing this problem easier. This is because for smaller jobs the grid must be divided into smaller sub-grids so that each can handle a smaller job, and the task of setting up and tearing down these sub-grids is part of the system's firmware. We believe having another level of hierarchy in the architecture itself, for example clusters of PEs, might have made this problem easier to solve as it provides natural units of isolation and management, compared to a monolithic grid of PEs.

In the first generation of MTIA, we built an architecture that can gain significant levels of efficiency for DLRM workloads compared to GPUs and other accelerators. We hope to continue to increase this efficiency over time as the software stack matures. The experience of writing kernels, building a compiler, and optimizing models for this architecture has given us great insights into what features are more impactful. We are hoping to integrate and leverage all the lessons learned on both hardware and software sides of the project in future generations of architecture.

ACKNOWLEDGMENTS

This project has been the culmination of dedicated and enthusiastic work of many talented teams and individuals. While it is impossible to mention every team and every person here, the authors would like to specially thank the following teams: Infra Silicon, AI Systems Hardware-Software Co-Design, Compiler Backend, Firmware, Emulation, Release to Production (RTP), Sourcing and Operations Engineering (SOE), and Hardware Platforms. We also would like to express our sincere gratitude to Misha Smelyanskiy, Olof Johansson, Kumar Sundararajan, K. Rajesh Jagannath, Xiao He, Jongsoo Park, Changkyu Kim, Mahesh Maddury, Brian Ko, Kaushal Gandhi, Tom Ulrich, Pritesh Modi, Manish Modi, Krishanth Skandakumaran, Teja Kala, Bhargav Alluri, Hao Jin, Adam Bauserman, Sameer

Shripad, Meghana Reddy Swamyreddygari, Sharat Kumar, Dick Tam, Prasad Addagarla, Di Wu, Puneet Anand, Sanjay Kumar, James Hegeman, Nadav Rotem, Fangran Xu, and Shuqing Zhao. We would also like to thank all our vendors for their collaboration and the support that they provided during the course of the project. This work would not have been possible without their close engagement.

REFERENCES

[1] V. Sze, Y. Chen, T. Yang, and J.S. Emer, *Efficient Processing of Deep Neural Networks,* Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2020.

[2] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[3] Y.-H. Chen, J. Emer, and V. Sze, Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks, in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2016.

[4] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks IEEE Journal of Solid-State Circuits (JSSC), 51(1), 2017.

[5] Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, and N. Takagi, "A high-speed multiplier using a redundant binary adder tree," in *IEEE Journal of Solid-State Circuits (JSSC)*, 22(1):28–34, 1987.

[6] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*, 2019.

[7] T. Krishna, H. Kwon, A. Parashar, M. Pellauer, and A. Samajdar, *Data Orchestration in Deep Learning Accelerators*, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2020.

[8] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-L. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2017.

[9] https://zephyrproject.org

[10] M. Anderson, B. Chen, S. Chen, S. Deng, J. Fix, M. Gschwind, A. Kalaiah, C. Kim, J. Lee, J. Liang, H. Liu, Y. Lu, J. Montgomery, A. Moorthy, S. Nadathur, S. Naghshineh, A. Nayak, J. Park, C. Petersen, M. Schatz, N. Sundaram, B. Tang, P. Tang, A. Yang, J. Yu, H. Yuen, Y. Zhang, A. Anbudurai, V. Balan, H. Bojja, J. Boyd, M. Breitbach, C. Caldato, A. Calvo, G. Catron, S. Chandwani, P. Christeas, B. Cottel, B. Coutinho, A. Dalli, A. Dhanotia, O. Duncan, R. Dzhabarov, S. Elmir, C. Fu, W. Fu, M. Fulthorp, A. Gangidi, N. Gibson, S. Gordon, B. Padilla Hernandez, D. Ho, Y. Huang, O. Johansson, S. Juluri, S. Kanaujia, M. Kesarkar, J. Killinger, B. Kim, R. Kulkarni, M. Lele, Huayu Li, Huamin Li, Y. Li, C. Liu, J. Liu, B. Maher, C. Mallipedi, S. Mangla, K.K. Matam, J. Mehta, S. Mehta, C. Mitchell, B. Muthiah, N. Nagarkatte, A. Narasimha, B. Nguyen, T. Ortiz, S. Padmanabha, D. Pan, A. Poojary, Y. Qi, O. Raginel, D. Rajagopal, T. Rice, C. Ross, N. Rotem, S. Russ, K. Shah, B. Shan, H. Shen, P. Shetty, K. Skandakumaran, K. Srinivasan, R. Sumbaly, M. Tauberg, M. Tzur, S. Verma, H. Wang, M. Wang, B. Wei , A. Xia, C. Xu, M. Yang, K. Zhang, R. Zhang, M. Zhao, W. Zhao, R. Zhu, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, V. Rao., "First-Generation Inference Accelerator Deployment at Facebook," in *Arxiv*, 2021. [Online]. Available: https://arxiv.org/abs/2107.04140, unpublished.

[11] J. Ehlen, J. Clow, B. Wei, D. Chong, "Facebook Multi-node Server Platform: Yosemite V2 Design Specification," Open Compute Project, https://www.opencompute.org/documents/facebook-multi-node-server-platform-yosemite-v2-design-specification

[12] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, KR. Kishore, T. Graf, A Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, V. Rao, "Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2022.

[13] M. Haken, J. Clow, Y. Li, B. Wei, D. Chong, T Ky, "Yosemite V3: Facebook Multi-node Server Platform Design Specification", Open Compute Project, https://www.opencompute.org/documents/ocp-yosemite-v3-platform-design-specification-1v16-pdf

[14] GemmBench. [Online]. Available: https://https://github.com/pytorch/glow/blob/master/tests/benchmark/GemmBench.cpp

[15] TableBatchedEmbeddingBagBench (TBEBench). [Online]. Available: https://github.com/pytorch/glow/blob/master/tests/benchmark/TBEBench.cpp

[16] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A.G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, M. Smelyanskiy "Deep Learning Recommendation Model for Personalization and Recommendation Systems," in *Arxiv*, 2021, [Online]. Available: https://arxiv.org/abs/1906.00091, unpublished

[17] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.S.Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, X. Zhang, "The Architectural Implications of Facebook's DNN-based Personalized Recommendation," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, February 2020.

[18] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, D. Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, Mikhail Smelyanskiy "Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications," in *Arxiv*, 2018, [Online]. Available: https://arxiv.org/abs/1811.09886

[19] J. K. Reed, Z. DeVito, H. He, A. Ussery, J. Ansel, "Torch.fx: Practical Program Capture and Transformation of Deep Learning in Python," in *Arxiv*, 2022, [Online]. Available: https://arxiv.org/abs/2112.08429

[20] https://pytorch.org/docs/stable/fx.html

[21] C. Lattner, V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in Proceedings of *International Symposium on Code Generation and Optimization*, 2004.

[22] https://llvm.org/docs/LangRef.html

[23] https://github.com/riscv/riscv-v-spec

[24] BatchGemmBench. [Online]. Available: https://github.com/pytorch/glow/blob/master/tests/benchmark/BatchGemmBench.cpp

[25] ConcatBench. [Online]. Available: https://github.com/pytorch/glow/blob/master/tests/benchmark/ConcatBench.cpp

[26] TransposeBench. [Online]. Available: https://github.com/pytorch/glow/blob/master/tests/benchmark/TransposeBench.cpp