Designing an AI Chip with assistance from Met

Machine Learning (ML) has become ubiquitous in online activities. In recent years, the scale and complexity of these models have grown significantly.

HONG KONG, CHINA, August 11, 2023 /EINPresswire.com/ -- I.Introduction

Machine Learning (ML) has become ubiquitous in online activities. Jak Electronics reports that the size and complexity of these models has grown significantly in recent years, which has helped improve the accuracy and effectiveness of predictions.. However, this growth brings substantial challenges for the hardware platforms used for large-scale training and inference of these models. The Total Cost of Ownership (TCO) is one of the main limiting factors for deploying models into production in data centers, with power being a significant component of these platform's TCO. Therefore, the performance per TCO (and per watt) has become a crucial benchmark for all hardware platforms targeting machine learning.

Deep Learning Recommendation Models (DLRM) have become one of the primary workloads in Meta's data centers. These models combine computation-intensive traditional Multi-Layer Perceptron (MLP) operations (sometimes referred to as fully connected or FC) with embedding tables that translate sparse features into dense representations. These tables contain wide vectors indexed randomly and reduce them to a single vector, which is then combined with data from other layers to produce the final result. While the computational demands of embedding table operations are relatively low, their memory footprint and bandwidth demands are relatively high due to the nature of the data access patterns and the size of the tables.
II. Motivation

Traditionally, CPUs have been the primary tool for serving inference workloads in Meta's production data centers, but they are not cost-effective in meeting the demands of the latest workloads. To some extent, hardware acceleration is considered an attractive solution that can address power and performance issues and provide a more efficient way to serve inference requests while leaving enough computational headroom to run future models.

While recent generations of GPUs offer a tremendous amount of memory bandwidth and computational power, they were not designed with inference in mind and therefore run actual inference workloads inefficiently. Developers have used countless software techniques such as operator fusion, graph transformations, and kernel optimizations to improve the efficiency of GPUs. However, despite these efforts, there is still an efficiency gap that makes deploying models in practice challenging and costly.

Based on the experience of deploying NNPI and GPUs as accelerators, it's clear that there is room for a more optimized solution for critical inference workloads. This optimal solution is based on an in-house accelerator, built from scratch to meet the stringent demands of inference workloads, with a particular emphasis on meeting the performance requirements of DLRM systems. However, while focusing on DLRM workloads (considering their ongoing evolution and the case that this architecture is effectively built for the next generation of these workloads), it is evident that, in addition to performance, the architecture should also offer enough generality and programmability to support future versions of these workloads and potentially other types of neural network models.

While creating custom silicon solutions opens the door for ample innovation and specialization for target workloads, creating an accelerator architecture for large-scale deployment in data centers is a daunting task. Therefore, the focus and strategy when building accelerators have always been to adopt and reuse technologies, tools, and environments provided by vendors and the open-source community. This not only shortens the time to market but also leverages support and enhancements from the community and vendors, reducing the amount of resources needed to build, enable, and deploy such platforms.

III. Accelerator Architecture

1.Fixed Function Units

Each PE has a total of five fixed function blocks and a command processor that coordinates operation execution on these fixed function blocks. The function 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 function unit can also directly access data within the PE's local memory, perform necessary operations, and write back the results without having to pass the data to other function units.

2.Memory Layout Unit (MLU)

This functional block performs operations related to modifying and copying the data layout in local memory. It can operate on tensors of 4/8/16/32-bit data types. Operations such as transpose, concatenation, or data reshaping are performed using this block. The output data can be directly sent to the next block for immediate operation or stored in the PE's memory. For instance, the MLU can transpose a matrix and feed the output directly to the DPE block for matrix multiplication operations, or it can format the data correctly as part of a depth convolution operation and send it to the DPE to perform the actual computations.

3.Dot Product Engine (DPE)

This functional block performs a set of dot product operations on two input tensors. It first reads

the first tensor and stores it within the DPE, then streams the second tensor and performs a dot product operation on all rows of the first tensor. The DPE can perform 1024 INT8 multiplications (32x32) or 512 FP16/BF16 multiplications (32x16) per cycle. The operations are fully pipelined; performing two maximum matrix multiplications requires 32 clock cycles. In the case of INT8 multiplications, the result 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.

4.Reduction Engine (RE)

The RE hosts storage elements that track the results of matrix multiplication operations and accumulate them over multiple operations. There are four independent storage banks that can be used independently for storing and accumulating results from the DPE. The RE can load initial biases into these accumulators and can also send their contents to adjacent PEs through a dedicated reduction network (discussed later in this section). When receiving results through the reduction network, the RE accumulates the received values on top of one of the values in local storage. It can then send the result to the adjacent functional block or SE, or store it directly in the PE's local memory.

5.SIMD Engine (SE)

This functional block performs operations such as quantization/dequantization and nonlinear functions. Internally, this block contains a set of lookup tables and floating-point arithmetic units for calculating linear or cubic approximations of nonlinear functions like exponential, sigmoid, tanh, etc. The approximations take INT8 or FP16 data types as input and produce INT8 or FP32 results at the output. The unit can directly receive its inputs from the RE block, or read them from local memory. In addition, this block is also capable of performing a set of predefined element-wise operations like addition, multiplication, accumulation, etc., using its floating-point ALU.

6.Fabric Interface (FI)

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

IV. Discussion

Building a chip is always a difficult, lengthy, and expensive process, especially when it is the first attempt. For MTIA, the resulting chip needed to deliver high performance, handle a wide range

of recommendation models, and offer a degree of programmability to allow rapid deployment of models in production.

Dual-Core PE: The choice to have two independent processor cores within the PE, and to allow both to control fixed function units, provides a great deal of parallelism and flexibility at the thread level, allowing computation to be decoupled from data movement. While this decoupling simplifies programming and alleviates performance issues from instruction constraints for specific operators (by providing twice the total instruction throughput), using the two cores effectively and correctly in software requires some effort. Details like synchronization between the two cores for initialization and cleanup were difficult to get right before the first run, but have been well-utilized across all workloads through proper integration in the software stack afterwards.

General-Purpose Computation: Adding general-purpose computation in the form of RISC-V vector support was a good choice: some operators were developed or found to be important after the architecture definition phase, so the architecture does not include any offloading support for them. Operators like LayerNorm and BatchedReduceAdd are easily vectorizable, and these implementations have proved to be superior to versions using scalar cores and fixed function units.

Automatic Code Generation: Some architectural choices about how to integrate and operate fixed function units within the PE made automatic code generation by the compiler difficult. The processor has to install and issue explicit commands to operate any fixed function block. While this is done by adding custom instructions and registers to the processor, it still requires assembling many arguments and passing them to each target engine to specify the details of the operation. Controlling a set of heterogeneous fixed function units from a program and balancing the data flow between them is a challenging task for the compiler. Achieving desired utilization levels on the fixed function units across a variety of input shapes and sizes is also difficult. While our DSL-based KNYFE compiler made writing kernels easier and automatically handled many of these issues, it still required learning a new DSL.

Buffers: Adding the circular buffer abstraction greatly simplified dependency checks between custom operations working on the same memory area, as the circular buffer IDs were used as a unit for dependency checks (similar to register IDs in processor cores). They also simplified the implementation of producer-consumer relationships between fixed function units and the processor, as the hardware would stall operations until enough data (or space) was available in the circular buffer, without any explicit synchronization needed at the software level. The flexible addressing mechanism also allowed arbitrary accesses to any position within the circular buffer, simplifying data reuse as different operations could access different segments of the circular buffer multiple times. However, this required explicit management of space within the buffer by software and deciding when to mark data as consumed, which could lead to hard-to-debug issues if not done correctly.

JAK Electronics
JAK Electronics
+852 9140 9162
it@jakelectronics.com
