Just-In-Time Memory Access for Streaming Architectures

Jerry Xu
University of Wisconsin-Madison
Madison, WI
mxu269@wisc.edu

Yu Xia
University of Wisconsin-Madison
Madison, WI
xia73@wisc.edu

Abstract

Tensor Streaming Processors (TSP) have become state of the art architecture for AI applications. TSPs exploit data parallelism and determinism to achieve groundbreaking throughput in tasks such as image processing and natural language processing. In this work, we propose just-in-time (JIT) memory access for streaming architectures. In typical Machine Learning workloads such as convolutions and general matrix multiplications (GEMM), the data dependencies are deterministic. This means we can pipeline the prefetching of data into the SRAM buffer from off-chip DRAM in a JIT manner. We also show there is abundant data reuse in convolutions and GEMM, so off-chip memory access can be minimized. Furthermore, we propose a scheduler, which is aware of the memory behavior of such streaming architectures. This scheduler takes in SRAM size as parameter, and efficiently schedules the workload to maximize data reuse. Finally, we evaluate the relationship between SRAM size and power consumption running ResNet-50.

1 Introduction

In recent years, a variety of novel architectures for AI accelerators have been developed. These architectures generally fall into two primary categories: Chip Multiprocessors (CMPs), including GPUs, and Dataflow architectures like the Tensor Processing Unit (TPU)[4]. Dataflow architectures are particularly noteworthy for their ability to circumvent the Von Neumann bottleneck, an inherent limitation in traditional computing systems where the rate of data transfer between the computation core and memory becomes a critical bottleneck, stalling the computation core. Recently, Groq has introduced its Tensor Streaming Processor (TSP)[1], which innovates with the concept of a "stream register." This streaming architecture completely gets rid of the Von neumann bottleneck by only utilizing 230MB of on-chip SRAM. The lack off-chip memory access makes the operation of large language models costly due to the need for multiple interconnected chips. Our project aims to tackle this limitation by proposing a just-in-time memory access system for off-chip DRAM, potentially reducing costs and improving efficiency in large-scale AI computations.

1.1 Key Contribution

This paper's primary contributions include the introduction of a just-in-time (JIT) memory access model tailored for streaming architectures, the development of a scheduler that optimizes for data reuse, and the creation of a simulator designed to estimate the energy consumption of memory-related operations within the proposed JIT memory access model.

2 Background

2.1 Systolic Array

The systolic array is central to dataflow architectures [3], such as those utilized by TPUs [2]. In this paper, we consider a 2D result-stationary systolic array. This configuration means the result of computations remains stationary in each cell, while the input data flows across the array: from east to west and from north to south. Each cell in the array performs a multiply and accumulate operation — a fundamental computation in many matrix-based algorithms.

2.2 Streaming Architecture

Groq proposed Tensor Streaming Processors that utilize "stream registers." Stream registers operate as a sequence of chained registers, designed to propagate their values unidirectionally. This design can be likened to a factory conveyor belt: as data flows through the stream register, analogous to parts moving along a conveyor belt, various components along the pathway can read and process this data. This mechanism provides a continuous flow of data, which feeds very nicely into a systolic array.

2.2.1 Formalization of Stream Operations. We treat each component as a black box that modifies the state of the stream and formalize this operation as follows:

$$s_{t+\delta} = \sigma(s_t) \quad (1)$$

where s_t is a vector input to the component at time t, with $s_{t+\delta}$ being the vector output at time $t+\delta$. σ is a sequence of operations executed by the component to change the state of the stream.

3 Just-In-Time Memory Access

We introduce the just-in-time memory access architecture: a memory module made from SRAM that buffers data loaded from off-chip DRAM to be accessed in a just-in-time manner.

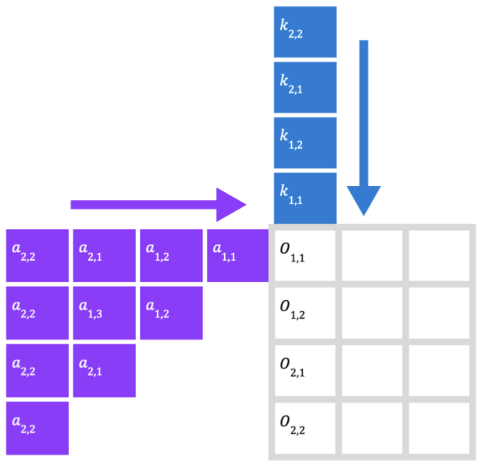

Figure 1. Weight Stationary Systolic Array

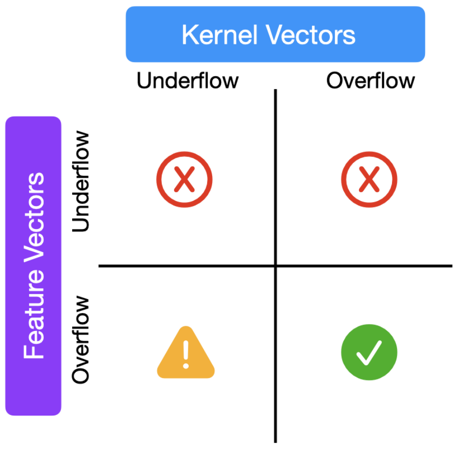

Figure 2. Systolic Array scheduling scenarios

Because Machine Learning workloads are deterministic, one can pre-fetch the data from DRAM and store them in the SRAM buffer. Given enough buffer space, it gives the illusion that data can be fetched from DRAM with low latency. The memory module acts as an interface between off-chip DRAM and the stream.

The memory module has 4 op codes: LD_{stream}, ST_{stream}, LD_{DRAM}, ST_{DRAM}. This allows fine-grained control of the memory behavior.

In a convolution operation, LD_{DRAM} is issued first to load data from DRAM to be filled into the SRAM. Then LD_{stream} instructions are issued to populate the streams with data from SRAM. Finally, the result will be stored through either ST_{stream} to store it to the SRAM, or ST_{DRAM} to write to the DRAM.

4 Scheduling

We consider the task of scheduling a 2D convolution, a key component in the ResNet architecture, onto a 2D weight-stationary systolic array. First, we flatten the features and kernels to form the feature and kernel vectors which both have size $(C*W*H)$. Then the feature vectors are fed into the systolic array from the east side, and the kernel vectors from the north side. The westward-flowing feature vectors will intersect with the southward-flowing kernel vectors, essentially doing dot product where the output features (results) are accumulated in each cell of the systolic array (Figure 1). The output features are then streamed out of the systolic array when it finishes accumulating.

4.1 Keep the Systolic Array Busy

Given a 256×256 result-stationary systolic array, it is fully utilized when each of the 256 feature vectors is doing a multiply accumulate with each of the 256 kernel vectors, ie. doing $256 \times 256 = 65536$ dot products between the feature and kernel vectors. In each direction, a number less than 256 would result in an underflow and a number more than 256 would result in an overflow, as directed in Figure 2 . Let's discuss each situation in detail:

4.1.1 Overflow of features and overflow of kernels.

is when number of feature and kernel vectors both exceed 256. In this case, both the feature vectors and kernel vectors are replayed to ensure a dot product is calculated for each feature and kernel vector pair. This is the ideal case because the systolic array is always fully utilized.

4.1.2 Overflow of features and underflow of kernels.

occurs when the number of feature vectors exceeds 256, while the number of kernel vectors is fewer than 256. To mitigate, overlapping feature vectors—due to stride variations—can merge into a single vector. This combined vector is then processed in the systolic array. Concurrently, a singular kernel vector is distributed across two columns of the systolic array, with the appropriate offset. This setup enables the execution of two dot products with the merged feature vector, each corresponding to one of the original feature vectors. A non-mitigation approach is to just to replay the feature vectors.

4.1.3 Underflow of features and overflow of kernels.

is when you have fewer than 256 feature vectors and more then 256 kernel vectors. This cannot be mitigated, because there is no overlapping values between the kernel vectors, so it cannot be optimized as was in the the previous scenario. The kernel vectors must be replayed.

4.1.4 Underflow of features and underflow of kernels.

is the case when both the number of feature and kernel vectors is less than 256. There is nothing you can do to mitigate.

5 Memory Access

Our design involves two layers of memory, SRAM and DRAM. Every access to the DRAM has to go to SRAM, and then to the stream. Matrix operations requires large data throughput. By storing kernels and features in SRAM, we reduce the overhead of accessing DRAM for frequently reused data, which is especially important for features in our convolution operation, where each feature pixel might be used in multiple feature vectors. For those frequently used feature pixels, we only need to load them from the DRAM for once.

In our design, we split the total SRAM capacity into α and β, where $\alpha + \beta = 1$. α-SRAM is reserved for kernel only, and β-SRAM is reserved for input and output features.

To better understand the write and read operations, we count the number of Load and Store for both SRAM and DRAM in a convolution operation.

Notation used:
F: Number of feature vectors, same as number of output pixels.
K: Number of kernel vectors.
$Kstored$: Number of kernel vectors that can be stored in the αSRAM
$Ksize$: Size of one kernel vector with unit in bytes.
SAh, SAw: The dimension of the systolic array.
N_{OP}: Number of transactions per unit. For kernel loads, it is $\lceil \frac{Ksize}{BUSWIDTH} \rceil$. For feature loads/stores, it is $\lceil \frac{Psize}{BUSWIDTH} \rceil$, where $Psize$ is $DataTypeSize * PixelDepth(Input/Output)$.
Matrix M: Usage frequency of the corresponding input feature pixel in the convolution operation.

5.1 Kernel SRAM Load

$$SRAM_{ld} = \left\lceil \frac{F}{SAh} \right\rceil * K * N_{OP} \tag{2}$$

All kernels have to be loaded from SRAM to the stream. If there are more feature vectors than the systolic array can handle, the whole set of kernels have to be loaded for multiple times (case of replay).

5.2 Kernel DRAM Load

$$DRAM_{ld} = (K + \left\lfloor \frac{F}{SAh} \right\rfloor * (K - KStored)) * N_{OP} \tag{3}$$

All vectors have to be loaded from DRAM for at least once. For the overflowed feature vectors, only the remaining kernels that cannot fit in the α-SRAM need to be fetched from DRAM.

5.3 Feature SRAM Load

$$SRAM_{ld} = \sum M * \left\lceil \frac{K}{SAw} \right\rceil * N_{OP} \tag{4}$$

This is straight forward, where all feature pixels need to be loaded from SRAM.

5.4 Feature SRAM Store

$$SRAM_{st} = F * N_{OP} \tag{5}$$

All outputs have to be stored into SRAM first.

5.5 Feature DRAM Load

$$DRAM_{ld} = (N_{toread} + \left\lceil \frac{K}{SAw} \right\rceil * \sum_{Remaining} M) * N_{OP} \tag{6}$$

Since different pixels have different use frequency, it is preferred to load the most frequently used pixels into the SRAM. We filled the β-SRAM with as many most frequently used features as possible. During this process, N_{toread} number of pixels are fetched from DRAM. Note that some of the most frequently used feature pixels might have already been in SRAM from the previous convolution output, thus don't need to be loaded from DRAM again. The remaining feature pixels have to be loaded from DRAM.

5.6 Feature DRAM Store

$$DRAM_{st} = P * N_{OP} \tag{7}$$

P is the number of output pixels that cannot be stored into the free space within β-SRAM. We will prioritize the stores to β-SRAM of output pixels that is most frequently used in the next convolution operation. The other output pixels will be dumped into the DRAM. Therefore, in the next convolution operation, additional feature vectors might be already present in the SRAM.

6 Energy Consumption

Since our DRAM access has to go through the SRAM, each DRAM Store implicitly incurs an SRAM Load. Similarly, each DRAM Load implicitly incurs an SRAM Store. Therefore, the total enery consumption is

$$\begin{aligned} Energy = E_{DRAM} * (\sum DRAM_{st} + \sum DRAM_{ld}) \\ + E_{SRAMld} * (\sum SRAM_{ld} + \sum DRAM_{st}) \\ + E_{SRAMst} * (\sum SRAM_{st} + \sum DRAM_{ld}) \end{aligned} \tag{8}$$

Where E is the energy consumption per transaction. We use the estimated energy consumption given by Cacti at line size of 64 bytes. DRAM load and store are assumed to have the same energy cost.

7 Results

We developed an energy simulator to evaluate the energy consumption of running ResNet-50 across different sizes of SRAM memory modules. We began with 2KB of SRAM and increased the size in 4KB increments up to 4MB. The results are dipicted in Figure 3 to illustrate how energy consumption varies with SRAM size.

From the simulation, the energy consumption quickly declines as SRAM size grows, reaching a minima of 1.55 mJ when the SRAM size is 678KB. The energy consumption then

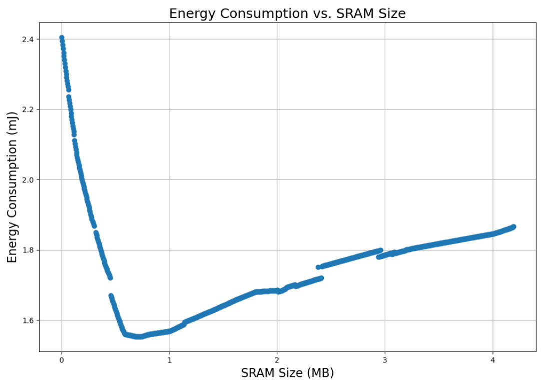

Figure 3. Energy Consumption vs. SRAM Size

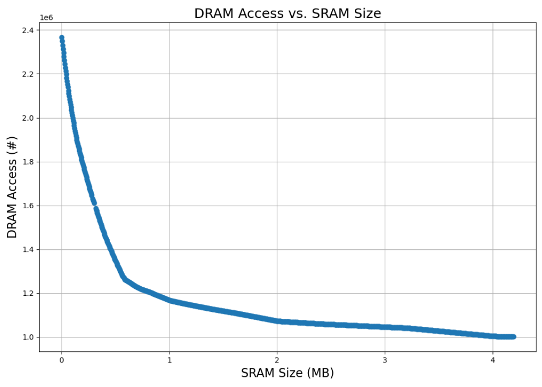

Figure 4. DRAM Access vs. SRAM Size

gradually increase as SRAM size grows bigger. The initial decline on energy consumption is the result of less DRAM access in the case of feature and kernel vector overflow, when they must be replayed. The increase in energy consumption is due bigger SRAM capacity requiring higher access energy.

Figure 4 depicts the declining number of DRAM access (total loads and stores) as SRAM size grows. We observe an inflection point when SRAM size is 678KB, which coincides with the energy consumption minima. This confirms that due to that larger SRAM drastically reduces the DRAM access in case of kernel or feature overflow. Once the SRAM is large enough to hold all kernels and features in a convolution, larger SRAM would start to display a diminishing return.

8 Comments

Since we have multiple lanes feeding data into the systolic array, to be able to fully utilize the systolic array and streams, the SRAM output width must be greater or equal to the total width of all streams. Otherwise, one or more streams might be starved under full load.

We can potentially add more IO ports to the SRAM block. However, this greatly increases the leakage of SRAM. Instead, we divide the SRAM into multiple identical and smaller SRAM banks. Each SRAM bank is directly mapped to a DRAM bank having a certain segment of address. The SRAM bank will be interacting with a smaller set of lanes, so that it would be able to saturate those lanes. Meanwhile, SIMD is enabled by offsetting the addresses to the SRAM banks, according to their corresponding address segments.

References

[1] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, et al. Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 145–158. IEEE, 2020.

[2] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[3] Hsiang-Tsung Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982.

[4] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Ai and ml accelerator survey and trends. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10. IEEE, 2022.