Faster Intersection Units

Vedantha Venkatapathy, Ryan Xiao
Massachusetts Institute of Technology
Cambridge, MA, USA
{vedantha, ryxiao}@mit.edu

***Abstract*—Tensor algebra is at the crux of matrix multiplications in DNNs. Sparsity is an effective solution to reduce skyrocketing computational costs. Output stationary dataflows for sparse matrix multiplication depend on intersection units that find non-zero elements in multiple vectors. Current optimized intersection units do not offer speedups on machine-learning style matrices. Our primary insight is that "skips", or values we can avoid reading in streams, are short in ML workloads. Using this insight, we develop an intersection unit (V2) that performs between 50% to 100% faster than conventional intersection units without increasing memory bandwidth requirements using limited additional buffering.**

I. Introduction

Tensor algebra is at the forefront of modern research in DNN hardware accelerators. Tensors rely on using vectors to complete matrix operations. Oftentimes, these vectors are sparse, so opportunities to improve performance arise from taking advantage of the sparse nature. The densities of these vectors can range from 50% to less than 1%, so the performance of these operations can be significantly improved by only operating on non-zero elements in the vector. The challenge with idea is locating the next pair of elements in the two vectors that are non-zero. We can call these intersections. Therefore, units that calculate the next intersection are called intersection units. This idea of accelerating intersection units has been explored by ExTensor [2], which aims to "skip" over sections of the vector that contain zeros based on a compressed representation of the vector coordinates. We aim to improve upon this by taking advantage of the assumption that sparse vectors will be uniformly distributed, meaning that skips taken will be relatively small, especially for uses in ML. We introduce two novel ideas to accelerate these intersection units. One version uses an increased bandwidth while the other uses a memory bandwidth similar to that of ExTensor.

II. Background

A. Matrix Representation

Consider two matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$

For uncompressed matrices, these can be stored in $O(mk)$ and $O(kn)$ space, and accessing a value (called a payload), given a coordinate (in this case a row and a column) is an $O(1)$ operation. This is because you know location the coordinate points to in memory (called a position).

However, for sparse tensors, this is suboptimal. Such representations take up substantially more space, and more importantly, do not enable efficient usage of sparsity. To see this we can consider the product $C = AB$.

$$C_{ij} = \sum_{r=0}^{k} A_{i,r} B_{r,j}$$

With this, it's clear we only need to consider the elements which are non zero in both A and B, since only those can affect the value C. This brings us to a nice representation for sparse matrices, called fibertrees [1].

We won't introduce fibertrees from scratch for brevity. We use the terminology introduced by this topic heavily.

The main idea is that fibertrees split up the abstraction between a coordinate, a position, and a payload. Depending on the representation used to store a particular level, different operations become cheaper. The particular representation we are interested in is known as a coordinate payload list (CPL), where a node in the fibertree consists of a list of coordinates with non-empty (or non-zero) payloads, and the associated payloads. This means that the operation `getNext`, which returns the next non-zero coordinate and its associated payload is an $O(1)$ operation, or the more generic `advanceStream(W)` and moves the stream forward by W positions, but getting the payload for an arbitrary coordinate is no longer cheap.

B. Output stationary dataflow

Given the above logic, one implementation for matrix multiplication is an output stationary dataflow. That is, our code would look something like this (ignoring tiling, and multiple levels of memory)

```
for i in M:
  for j in N:
    for A_r, B_r in A[i,:] & B[:,j]:
      C_{ij} += A_r * B_r
```

Where the heavy lifting for finding the non-zero values for both of the vectors in the third line is hidden by the `&` operation.

C. The Intersection Unit

We now consider the basic implementation of that `&` operation for a CPL storage system, which is the baseline described in ExTensor [2].

The pseudocode for the baseline is described below, where A and B are the input streams.

```
if A.is_empty() or B.is_empty():
  return DONE
else if A.peek() < B.peek():
  if (A.canAdvanceCpk(B.peek())
    A.advanceToLastCpk(B.peek())
  else
    A.advanceStream(1)
  return None
else if B.peek() < A.peek():
  if (B.canAdvanceCpk(A.peek())
    B.advanceToLastCpk(A.peek())
  else
    B.advanceStream(1)
  return None
else:
  result = A.peek()
  A.advanceStream(1)
  B.advanceStream(1)
  return result
```

(a)

```
if A.is_empty() or B.is_empty():
  return DONE
else if A.peek() < B.peek():
  nextN = A.streamNext(N)
  finalIdx = filter(nextN <= B.peek())
  A.advanceStream(finalIdx + 1)
  return nextN[finalIdx] OR NONE
else if B.peek() < A.peek():
  nextN = B.streamNext(N)
  finalIdx = filter(nextN <= A.peek())
  B.advanceStream(finalIdx + 1)
  return nextN[finalIdx] or NONE
else:
  result = A.peek()
  A.advanceStream(1)
  B.advanceStream(1)
  return result
```

(b)

```
if A_buffer.is_empty() or B_buffer.is_empty()
  return DONE
else if A_buffer.peek() < B_buffer.peek()
  B_buffer.fill()
  A_buffer.advanceTo(B_buffer.peek())
  return None or Relevant Value
else if B_buffer.peek() < A_buffer.peek()
  A_buffer.fill()
  B_buffer.advanceTo(A.peek())
  return None or Relevant Value
else:
  result = A_buffer.peek()
  A_buffer.advanceStream(1)
  B_buffer.advanceStream(1)
  return result
```

(c)

Fig. 1: Pseudocode for Extensor Implementation (left), our improved version with an assumed increase in memory bandwidth (center), and our improved version with a limited memory bandwidth (right).

```
if A.is_empty() or B.is_empty():
  return DONE
else if A.peek() < B.peek():
  A.advanceStream(1)
else if B.peek() < A.peek():
  B.advanceStream(1)
else:
  result = A.peek()
  A.advanceStream(1)
  B.advanceStream(1)
  return result
```

ExTensor improves upon this scheme by storing T coordinates, and their associated positions so a stream can skip to one of these positions when needed. This is beneficial if we consider the streams `A = [1, 2, 3, 4, 100]` and `B = [1, 2, 3, 4, 5, 6, 7, 8, 9, 101]`. Let $T = 5$, so that every other element in B has its coordinate tracked (that is the coordinates `2, 4, 6, 8, 101`, while every coordinate in A also has its coordinate tracked. When the value of `A.peek()` hits `100`, and `B.peek() = 5` we compare the T values tracked for stream B, then we notice that we can skip up to the coordinate `101` in B, avoiding even checking the values `6-9`, saving cycles. For the sake of comparison, we provide the pseudocode for extensor in Fig. 1a.

The issue is that as the stream length S grows, and S/T shrinks, you get coarser coordinates, and thus less opportunity to skip ahead. This condition manifests in the machine learning workloads which naturally have relatively low-levels of sparsity compared to graphs, for example. In addition pruning often leaves us with unstructured sparsity, further limiting the amount of skips. In Figure 14 of their paper, we see that their skipping mechanism gave them no benefit for the AlexNet cases.

Our system improves upon Extensor with the following key observation: since skips tend to be small, we can buffer the next N elements in the stream using a simple wrapper, and get the benefits of fine-grained skipping (large T), while comparing against only N elements. In addition, are able to restrict memory bandwidth, meaning we buffer the next N elements by only reading one element in a given cycle.

Other architectures also try to compare against the next N inputs, for example Sparch [3], which reads a sliding window of size N from either array, and computes, in parallel, all the intersections between the $2N$ elements it reads using N^2 comparators. This is both expensive in terms of requiring N^2 comparators, as well as reading N values from either stream per cycle. Our method only reads one value from each stream in a given cycle, saving both memory bandwidth and chip area. For this reason, we do not compare against SPARCH, as we're solving different problems.

III. Method

We implement two versions of our improved intersection unit.

A. V1: Assume increased memory bandwidth

Our first version of the intersection unit assumes N times the backing memory bandwidth, which lets you perform vector reads of the next N elements in a stream in parallel. Then, we can perform a parallel comparison against these N elements. The pseudocode for this implementation is given in Fig. 1b.

B. V2: Limiting memory bandwidth

We now introduce the following constraint: we cannot read multiple elements for DRAM in a cycle.

To deal with this we introduce a `nextNElements` buffer as a wrapper around the streams we care about. This buffer attempts to store the next N positions and coordinates in the stream.

The buffer will stay up to date with the next N values as long as we only increment either stream by 1 element since we drop one value, and bring a new value in. However, whenever we perform a skip over j elements, then those j elements of the buffer are not useful, and thus the buffer has only $N - j$ useful elements.

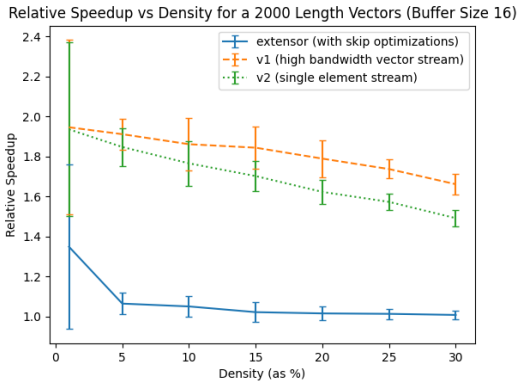

Fig. 2: Relative Speedup vs Vector Density for Vectors of Length 2000 and Buffer of Size 16

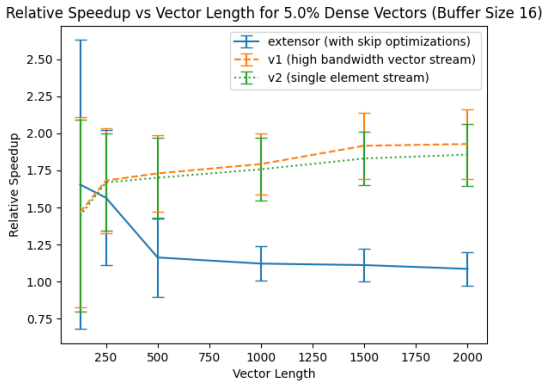

Fig. 3: Relative Speedup vs Array Length for 5.0% Vector Density and Buffer of Size 16

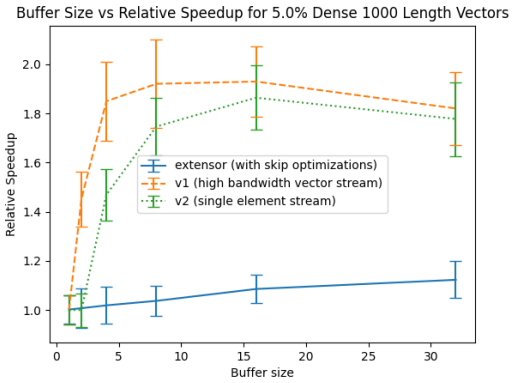

Fig. 4: Relative Speedup vs Buffer Size for Vector Density of 5% and Length of 1000

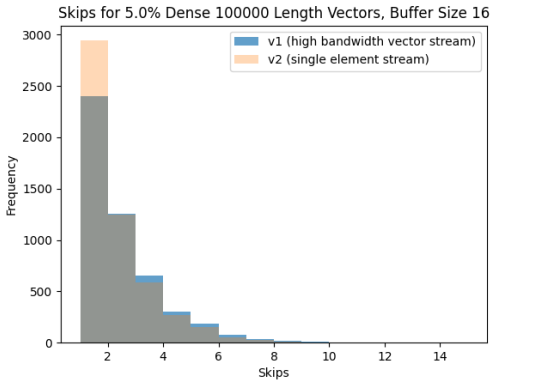

Fig. 5: Frequency of Skip Sizes for Vectors of Density 5.0%, Length 100000, Buffer Size 16

We can "catch up" the buffer, using the following technique: we refill an element (if needed) into the right buffer (from the right stream) in the first `else if`, since that statement nominally only reads from the left stream, and vice versa in the second `else if`. This "gains" us one element in the corresponding buffer.

Still, our buffer may not contain the full next N elements, and have some stale values. This could prevent us from finding the longest possible skip in a given cycle. This loses us throughput. In addition, it means we rely on the "catch-up" mechanism, which means we need enough cases where streams advance by a single element. Luckily, our evaluation suggests that such cases are uncommon, and even with them we still maintain healthy speedups over existing designs. The pseudocode for this version is given in Fig. 1c.

In addition, we assume, like the ExTensor paper, that initially filling up the buffers can be done via double-buffering during the previous input, and thus costs us 0 cycles.

We did not synthesize this, but we expect the hardware to have a similar area as an ExTensor unit with $T = N$, since the designs are similar.

IV. Evaluation

A. Experimental Setup

We first evaluate the robustness of our assumption that most skips taken tend to be smaller. This assumption is based on the probabilistic model that non-zero values in a sparse matrix or vector are uniformly distributed. We evaluate this assumption by counting the frequency at which various skip lengths occur.

We can then measure the speed of each intersection unit by counting the number of cycles it takes to complete. In order to measure the performance of our improved intersection units compared to ExTensor, we measure the speed-up of these units relative to a classic intersection unit. This is calculated by dividing the cycle count for each of the accelerated units by the cycle count for the classic unit. The vector density, vector length, and buffer size can each affect the performance of each unit so in order to exhaustively test, we evaluate the unit on a range of one variable while keeping the other two variables constant. We are only evaluating these units on vector multiplication because this is the only relevant component of matrix multiplication that would benefit from our improvement.

B. Evaluating the small-skips assumption

Our design relies on the fact that the vast majority of skips tend to be small. We evaluate this assumption in Fig. 2.

We notice that the vast majority of skips are rather small, within two or three elements. This validates the premise behind our intersection units only looking for skips within the next N elements.

C. Raw performance comparison

In Fig. 2 and 3, we can see that where ExTensor barely provides improvements over the baseline, our design runs at between $1.5\times$ to $2\times$ the speed.

D. Buffer size comparison

In both ExTensor, and our designs, the buffer lengths are tunable. This manifests as T in ExTensor, and N for us. In Fig. 4, we ran trials comparing the impact of buffer length on speedup. We achieve better speedups with substantially smaller buffer requirements than ExTensor. ExTensor will begin to catch up as the buffer length gets comparable to the stream length, but this requires small tiles. For context, ExTensor uses 32 elements in its implementation in the paper.

E. Evaluating the impact of single-element reads

Our implementation of the limited-memory bandwidth design, has some performance implications. Specifically, we may not capture all of the possible "skip" opportunities as our buffer for the next N elements may lag behind, for reasons mentioned previously. We can explore these impacts via Fig. 5.

From the histogram in Fig. 5, it turns out we do miss a couple opportunities for skips, but not a substantial number. In addition, we see that there are a lot of places where even the ideal design (V1) does not perform a skip (noted as a skip of length 1), validating our assumption that there are a reasonable number of "catch up" elements, which our design uses to refill its buffer.

The combination of these two observations means our realistic design performs marginally worse than the idealized high bandwidth version, and still substantially better than the baseline and ExTensor as can be seen in Fig. 2 and 3.

V. Conclusion

We have shown that we can achieve impressive speedups in the intersection unit by looking at the next N elements in a stream and that this is possible without increasing memory bandwidth.

All code in this project is accessible at https://github.com/v-vedantha/hwdl

References

[1] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2020. Efficient Processing of Deep Neural Networks. Springer.

[2] Hegde Kartik, Asghari-Moghaddam Hadi, Pellauer Michael, Crago Neal, Jaleel Aamer, Solomonik Edgar, Emer Joel, and Fletcher Christopher W.. 2019. ExTensor: An accelerator for sparse tensor algebra. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52). Association for Computing Machinery, New York, NY, USA, 319–333.

[3] Zhang, Zhekai, Hanrui Wang, Song Han, and William J. Dally. "Sparch: Efficient architecture for sparse matrix multiplication." In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 261-274. IEEE, 2020.

[4] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: ineffectual-neuron-free deep neural network computing. SIGARCH Comput. Archit. News 44, 3 (June 2016), 1–13. https://doi.org/10.1145/3007787.3001138