CSE-234: Data Systems for Machine Learning, Winter 2025

4: Tensor format, matmul deep dive, accelerators

Lecturer: Hao Zhang

Scribe: Yu-Pao Tu, Zaiyang Zhang, Anay Kulkarni, Mingrui Yin, Peiyun Han, Shyam Nuggehalli, Prabhleen Kaur, Pritika Barshilia, Cheril Shah, Ziqiao Xi, Luting Lei, Spoorthi Kalkunte

To orient ourselves: In Lecture 3, we began the discussion of accelerating operators. We have talked about vectorization and touched briefly on data layout (strides). In Lecture 4, we continue our discussion from there:

1 Operator Acceleration Continued

1.1 Strides Recap

When we talk about how matrices (and, more generally, tensors) are laid out in memory, we're discussing two major things:

1. Contiguity rules — in what order the elements are arranged in the underlying 1D memory space.
2. Strides — how we map an (i, j) or $(i, j, k, \dots)$ index in a multi-dimensional tensor to its address in memory.

1.1.1 Row-major vs. Column-major

Row-major (C-style)

Definition: In row-major order, the rows are laid out *contiguously* in memory. You store all the elements of row 0, then all the elements of row 1, etc. Example: If you have a 3×4 matrix A, the layout in memory looks like:

$$A_{00}, A_{01}, A_{02}, A_{03}, \quad A_{10}, A_{11}, A_{12}, A_{13}, \quad A_{20}, A_{21}, A_{22}, A_{23}$$

Column-major (Fortran-style)

Definition: In column-major order, the columns are stored contiguously in memory. You store all elements of column 0, then column 1, etc. Example: The same 3×4 matrix A in column-major memory would be:

$$A_{00}, A_{10}, A_{20}, \quad A_{01}, A_{11}, A_{21}, \quad A_{02}, A_{12}, A_{22}, \quad A_{03}, A_{13}, A_{23}$$

Strides: A stride tells you how many memory steps (elements, not necessarily bytes) you move to get from one index to the next along a particular dimension. Formally, if you have an n-dimensional tensor $(d_0, d_1, \dots, d_{n-1})$ with strides $(s_0, s_1, \dots, s_{n-1})$, then the memory location of element $(i_0, i_1, \dots, i_{n-1})$ is

given (in pseudo-code) by:

$$\text{memory\_index}(i_0, i_1, \ldots, i_{n-1}) = \text{base\_offset} + \sum_{k=0}^{n-1} (i_k \times s_k)$$

- base_offset: The index (or pointer offset) where the tensor's data starts in memory.
- s_k: The stride for dimension k.

Why are strides used?

1. General Multi-dimensional to 1D Mapping: Memory is ultimately linear (1D). Strides tell us how to jump in that 1D space when we move by 1 step in a given dimension.
2. Slicing or Subviews Without Copy: If you take a subarray (slice) of a bigger array, you can keep the same underlying buffer but just change:
   (a) The starting offset (where the slice begins)
   (b) The shape (the new dimensions)
   (c) The strides (how to jump in memory for each dimension)

   For instance, if you want to skip every other element, you can modify a dimension's stride accordingly, without copying data. Hence, strides are a flexible way to interpret how multi-dimensional indices map to an underlying continuous chunk of memory.

Examples of Strides:

Row-major 2D Shape: [m,n] (m rows, n columns) Typical strides: [n,1]

- The stride in the first dimension (the "row" dimension) is n, meaning if you move from row i to row i+1, you skip n elements in memory.
- The stride in the second dimension (the "column" dimension) is 1, meaning adjacent columns are adjacent in memory.

Column-major 2D Shape: [m, n] Typical strides: [1, m]

- The stride for the row dimension is 1, meaning moving from row i to row i+1 is just 1 step in memory.
- The stride for the column dimension is m, meaning to jump from column j to j+1, you skip m elements in memory.

1.2 Using strides to accelerate operators

1.2.1 Slicing a Tensor

When you "slice" a tensor (or matrix) using the zero-copy method, you're not actually copying any of the data values in memory. Instead, you create a new view by adjusting three things.

1. Offset – Where in the original memory buffer this sub-slice should begin.

2. Shape – The new dimensions (e.g., from 4×5 to 3×2).
3. Strides – How to jump through memory when iterating along each dimension.

Example: Slicing a 4×5 matrix to a 3×2 sub-matrix.

Let's assume we have a 4×5 matrix A, stored in row-major order.

- Shape: [4, 5]
- Strides: [5, 1] in row-major (the first dimension has stride 5, the second has stride 1)
- Base offset: 0 (the start of A in memory)

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

Step 1: Identify the sub-matrix Suppose we want a sub-matrix that starts at row 1, column 2, and has shape 3×2. That means:

Start: $(\text{row}, \text{col}) = (1, 2)$ Size: 3 rows by 2 columns. So it will include rows 1, 2, 3 and columns 2, 3.

Concretely, that sub-matrix is:

$$\begin{pmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$$

Step 2: Compute the new metadata

- New shape = [3, 2].
- New offset:
  - Each row in the original has 5 elements, so row '1' starts at memory index $1 \times 5 = 5$
  - Within that row, we skip 2 columns, so we add $2 \times 1 = 2$ more.
  - Therefore, the sub-matrix starts at memory index $5 + 2 = 7$.
  - So offset = 7.
- Strides: Usually remain the same as the original for a simple "contiguous slice."
  - Row-major strides were [5, 1].
  - The sub-matrix is still laid out row by row, so strides = [5, 1] in this view.

Thus, this sub-matrix is described by:

- Shape = [3,2]
- Offset = 7

- Strides = [5,1]

No actual data was copied. Instead, we just told the system: "Start from memory index 7, and interpret the data with shape [3, 2] and strides [5, 1]."

And that's exactly the sub-matrix we wanted:

$$\begin{pmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$$

1.2.2 Transposing a Tensor

A transpose in the context of multi-dimensional arrays (tensors) can be thought of as rearranging the axes of the tensor. For a 2D matrix (shape $[m, n]$), the simplest transpose is swapping rows and columns to get $[n, m]$. For higher-dimensional tensors, "transpose" typically means a more general permutation of the axes (e.g., turning a tensor with shape $[D_0, D_1, D_2]$ into something like $[D_2, D_0, D_1]$).

Logical (Stride-Based) Transpose (Zero-Copy):

- Instead of physically rearranging data, you can logically transpose by changing how you interpret the buffer in memory.
- In a simple 2D case (row-major):
- Original strides might be $\text{stride\_row}, \text{stride\_col} = [n, 1]$ for an $[m, n]$ array.
- When you transpose, you swap these strides to $[1, m]$, effectively "reading" the data in transposed order.
- This requires no copying, but note that the transposed layout can be less cache-friendly if the data are physically laid out in row-major order.

Physical Transpose (Copy):

- You actually create a new buffer and write elements to their transposed positions so that the transposed data become contiguous in the new layout.
- This gives better memory locality for many operations on the transposed data but requires time and memory to perform the copy.

Most deep-learning or numerical frameworks (NumPy, PyTorch, TensorFlow, etc.) support both ways. `.T` (NumPy) or `.transpose(...)` (PyTorch/TensorFlow) often defaults to returning a view (stride-based transpose) if it can. But if a contiguous buffer is needed for performance, they might materialize (copy) the transposed tensor.

1.2.3 Broadcasting a Tensor

Broadcast is prevalent because many times we want to do operations on tensors with only limited dimensional match. In this lecture, we use the example of adding a and b with a.shape = [4, 3] and b.shape = [1, 3], as visualized by Fig. 1.

How can we do this?

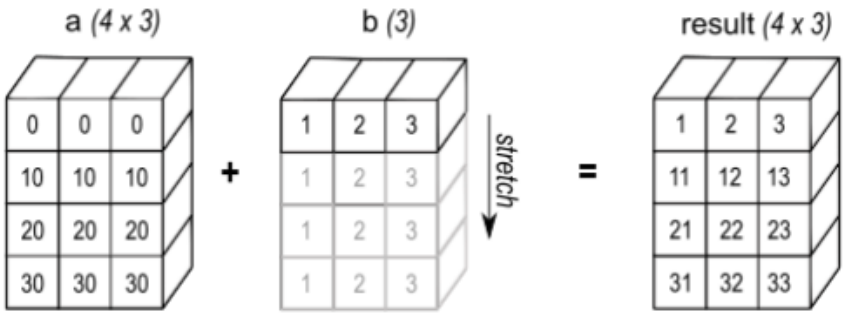

Figure 1: Visualization of a simple broadcast

- Rudimentary way: Copy four times. It is slow and consumes a lot of memory
- How we really implement it: Change b.stride from [1] to [0, 1]; and b.shape from [1, 3] to [4, 3]

Why the second way works? Recall that **PyTorch tensors are backed by contiguous memory arrays**, with the position of each element calculated using .shape and .stride attributes according to the formula:

$$A[i,j] = A.data[\text{offset} + i \times A.\text{strides}[0] + j \times A.\text{strides}[1]]$$

Therefore, with A.strides[0] = 0, we have

$$A[i,j] = A.data[\text{offset} + j \times A.\text{strides}[1]] \quad \text{for any } i.$$

This is essentially "telling" the machine to go back to offset and start over when creating a new line.

In practice, Pytorch does this for us automatically behind the scenes when we input a + b or a @ b

1.2.4 Challenge: swap the tiles with zero-copy

How can you swap the pink part with the blue part of the matrix shown in Fig. 2?

Straight-forward but slow way:

```
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> np.vstack((np.hstack((a[0:2, 0:2], a[2:4, 0:2])),
               np.hstack((a[0:2, 2:4], a[2:4, 2:4]))))
array([[ 0,  1,  8,  9],
       [ 4,  5, 12, 13],
       [ 2,  3, 10, 11],
       [ 6,  7, 14, 15]])
```

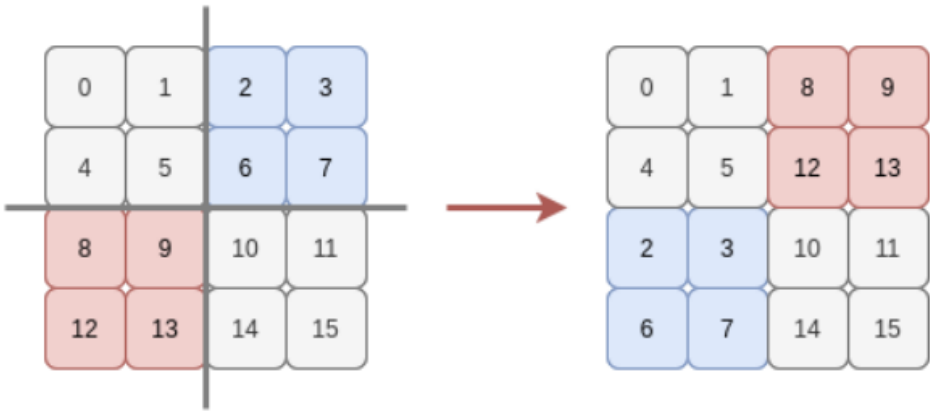

Figure 2: Can you do the permutation with zero-copy?

1.2.5 One-sentence takeaway:

Utilize strides to the best of your interest to achieve zero-copy operations whenever possible!

1.3 Pitfall of stride: discontinuity

Many operations require contiguous memory layout for efficient utilization of hardware memory system.

Using stride can easily violate that requirement. For example, when we create a new tensor by slicing an existing tensor, the storage of this new tensor in the memory is essentially discontinuous as we skip some elements stored in between.

This is why we have contiguous method in Pytorch.

1.4 Parallelize Operators (brief intro for now)

Recall the example we mentioned in Lecture 3, that is, conducting element-wise addition of matrix A and B, then write result into matrix C.

We discussed in Lecture 3 that we could utilize SIMD instructions like load_float4 to vectorize this process. Here, we further point out that, **as this operation is element-wise**, i.e. the addition of each element is **independent** from each other, we could also easily parallelize it using an OpenMP decorator as shown in Fig. 3.

If the operation is not element-wise, i.e., operation on one element is dependent on the one on another, things become tricker. We will cover that situation later in this course.

1.5 Summarizing Operator Acceleration

To make operators faster, we can do:

- Vectorization (discussed in Lecture 3):

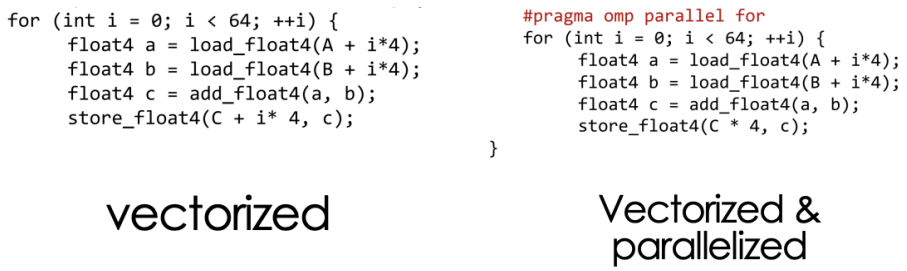

Figure 3: Parallelization is simple with element-wise operation

Leveraging platform-specific vectorized functions

- Data Layout Manipulation (1.1 to 1.3):
  Using strides to enable zero-copy operations
- CPU Parallelization (1.4):
  Straightforward to implement for element-wise operators

2 Matrix Multiplication

2.1 Overview

In this section, we are going to explain some techniques specialized for matrix multiplication

2.1.1 What is Matmul in Code?

The code implements a naive $n \times n$ matrix multiplication: outer loops (i, j) iterate over the output matrix while inner loop (k) computes the dot product of the corresponding row of A and column of B. And the result is stored in $C[i][j]$.

The time complexity of Naive Algorithm is $\mathcal{O}(n^3)$. For every pair of indices (i, j), we perform n multiplications. However the State-of-the-Art algorithm can reach the complexity of $\mathcal{O}(n^{2.371552})$, achieved through advanced techniques like Strassen's algorithm, Coppersmith-Winograd, and later improvements by Le Gall and Williams.

There are some insights into the optimization of matmul: Focus on memory access patterns and arithmetic efficiency. Aim to minimize redundant computations and maximize data reuse.

Significant milestones in reducing the matrix multiplication exponent (ω) starting from Strassen's $\mathcal{O}(n^{2.807})$ in 1969. Some recent progress (Williams et al., 2024) has brought it down to $\mathcal{O}(n^{2.371552})$.

However, further reductions in ω yield diminishing practical benefits for most real-world problems due to:

- **high constant factors in advanced algorithms**;

- **increased complexity in implementation**.

Timeline of matrix multiplication exponent

Year	Bound on omega	Authors
1969	2.8074	Strassen[1]
1978	2.796	Pan[10]
1979	2.780	Bini, Capovani [it], Romani[11]
1981	2.522	Schönhage[12]
1981	2.517	Romani[13]
1981	2.496	Coppersmith, Winograd[14]
1986	2.479	Strassen[15]
1990	2.3755	Coppersmith, Winograd[16]
2010	2.3737	Stothers[17]
2012	2.3729	Williams[18][19]
2014	2.3728639	Le Gall[20]
2020	2.3728596	Alman, Williams[21][22]
2022	2.371866	Duan, Wu, Zhou[23]
2024	2.371552	Williams, Xu, Xu, and Zhou[2]

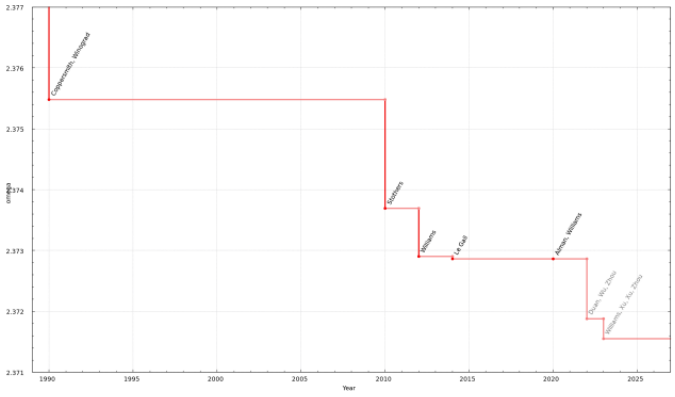

Figure 4: Timeline of Matrix Multiplication Algorithm

2.1.2 How to Make Matmul Fast

Recap of Arithmetic Intensity (AI): Defined as the ratio of arithmetic operations (#ops) to memory traffic(#bytes). Our goal is to maximize AI to utilize compute resources efficiently.

- **Increase #ops:** Fuse operations to combine multiple computations into fewer memory-bound steps; Reuse intermediate results to reduce redundant computations.
- **Decrease #bytes:**Optimize data layout for better cache usage; Use blocking and tiling techniques to keep active data in fast memory (registers/L1 cache).

2.1.3 Recap of Memory Hierarchy

The CPU has access to different levels of memory, with a trade-off between speed and capacity.

- **Registers:** Closest to the CPU, fastest but very limited in size.
- **L1/L2 Cache:** Intermediate layers of memory to reduce latency for frequently used data.
- **DRAM:** Large but significantly slower, used for data that doesn't fit in cache.

Also there exists different optimization techniques:

- **Temporal Locality:** Reuse the same data multiple times while it's in the cache.
- **Spatial Locality:** Load data in contiguous blocks to minimize cache misses.
- **Blocking/Tiling:** Divide computations into smaller blocks that fit into faster memory.

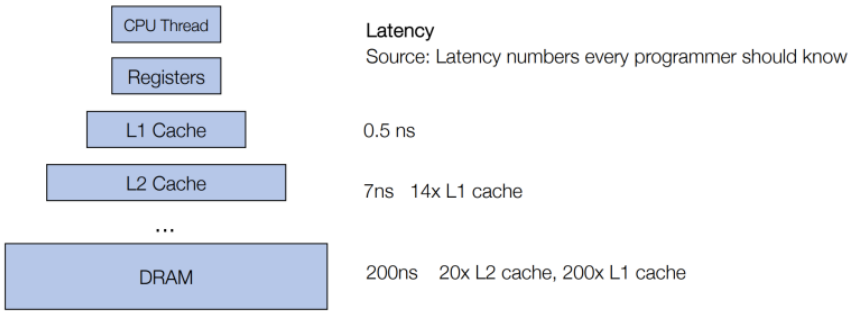

Figure 5: Hierarchy of memory and their speed

Let's make the model simpler.

- **Layers:**
  1. **First Layer (Compute Units):**
     - Registers and CPU ALUs execute operations at high speed.
     - High performance depends on keeping data in registers to avoid accessing slower memory layers.
  2. **Second Layer (DRAM):**
     - Acts as a large but slow memory bank.
     - Accessing DRAM frequently leads to bottlenecks.

To improve performance, we should maximize the use of registers and caches by structuring computations to minimize DRAM accesses.

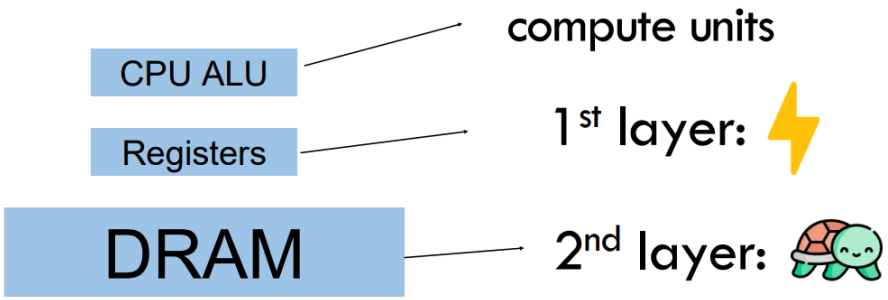

Figure 6: Simplified Memory Hierachy Model

2.1.4 Estimating Arithmetic Intensity

Let's consider an example. We have a piece of code along with its optimized version, and we will calculate the Arithmetic Intensity (AI) for both versions.

Original Code and Calculation

The original code performs the computation $E = D + ((A + B) \times C)$ in multiple steps:

```
float* A, *B, *C, *D, *E, *tmp1, *tmp2;
// assume arrays are allocated here
// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Computation Breakdown:

- Step 1: $tmp1 = A + B$ (1 arithmetic operation per element)
- Step 2: $tmp2 = tmp1 \times C$ (1 arithmetic operation per element)
- Step 3: $E = tmp2 + D$ (1 arithmetic operation per element)

Memory Operations:

- Each step loads two arrays and stores the result in one array.
- Total memory operations = 3 (loads/stores per operation) $\times n$ elements.

Arithmetic Intensity (AI):

- Total arithmetic operations = $3n$ (1 addition + 1 multiplication + 1 addition per element).
- Total memory operations = $9n$ (3 loads/stores per element $\times$ 3 steps).
- AI = $\frac{\text{Arithmetic Operations}}{\text{Memory Operations}} = \frac{3n}{9n} = \frac{1}{3}$.

Optimized Code and Calculation

The optimized code fuses the operations into a single loop:

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i = 0; i < n; i++) {
        E[i] = D[i] + (A[i] + B[i]) * C[i];
    }
}
// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

Computation Breakdown:

- Each loop iteration performs:
  - $A[i] + B[i]$ (1 addition)
  - Multiply the result by $C[i]$ (1 multiplication)

  - Add $D[i]$ to the product (1 addition)
- Total arithmetic operations = $3n$ (3 operations per element).

Memory Operations:

- 4 loads (A, B, C, D) + 1 store (to E) per element.
- Total memory operations = $5n$.

Arithmetic Intensity (AI):

- AI = $\frac{\text{Arithmetic Operations}}{\text{Memory Operations}} = \frac{3n}{5n} = \frac{3}{5}$.

The original implementation has an AI of $\frac{1}{3}$, while the optimized (fused) implementation has an AI of $\frac{3}{5}$. By fusing the operations into a single loop, the arithmetic intensity is increased, reducing memory bandwidth usage and improving compute efficiency. The fused implementation is more performant as it balances the ratio of computation to memory operations more effectively.

2.2 Register Tiled Matrix Multiplication

Matrix multiplication can be computationally inefficient because computing each element in matrix **C** requires repeatedly loading the corresponding row of matrix **A** and column from matrix **B**, leading to poor data reuse. Register tiled matrix multiplication addresses this inefficiency by dividing the matrices into smaller sub-blocks (tiles) that fit within the available registers. This transformation enhances data locality by enabling the loading of entire tiles into registers at once, reducing redundant memory accesses and maximizing data reuse during computation.

```
1  dram  float A[n/v1][n/v3][v1][v3];
2  dram  float B[n/v2][n/v3][v2][v3];
3  dram  float C[n/v1][n/v2][v1][v2];
4
5  for (int i = 0; i < n/v1; ++i) {
6      for (int j = 0; j < n/v2; ++j) {
7          register float c[v1][v2] = 0;
8          for (int k = 0; k < n/v3; ++k) {
9              register float a[0:v1, 0:v3] = A[i][k];
10             register float b[0:v2, 0:v3] = B[j][k];
11             c += dot(a, b.T);
12         }
13         C[i][j] = c;
14     }
15 }
```

Listing 1: Register Tiled Matrix Multiplication

As shown in Listing 1, we introduce three tiling factors v_1, v_2, v_3, which represent the dimensions of the tiles. Specifically, v_1 determines the number of rows in each output tile of matrix **C**, v_2 determines the number of columns, and v_3 corresponds to the shared dimension of the tiles in matrices **A** and $\mathbf{B}^{\mathrm{T}}$. The outer loops (line 5, 6) iterate over the matrix **C** in tiles of size $v_1 \times v_2$, and the algorithm processes these tiles block by block.

Within each tile of matrix **C**, the inner loop (line 8) iterates over the tiles of **A** and $\mathbf{B}^{\mathrm{T}}$. At each step, a tile from matrix **A** of size $v_1 \times v_3$ and a tile from matrix $\mathbf{B}^{\mathrm{T}}$ of size $v_2 \times v_3$ are loaded into registers (lines 9 and

10). These tiles, **a** and **b**, are used to compute a partial dot product, which is accumulated into the tile **c** (line 11).

Once the computation for all v_3-sized chunks is complete, the final result for the tile **c** is written back to memory (line 13), updating the corresponding portion of matrix **C**. By dividing the matrices into smaller tiles and performing computations on these localized blocks, the algorithm minimizes redundant memory accesses and maximizes data reuse within registers.

2.2.1 Computation Analysis

From lines 5, 6, and 8 of the pseudo-code, the memory access for matrix **A** involves loading tiles of size $v_1 \times v_3$ for each computation of a tile in **C**. This results in $\frac{n}{v_1} \times \frac{n}{v_2} \times \frac{n}{v_3}$ iterations, with each iteration loading $v_1 \times v_3$ elements. Therefore, the total read operations for **A** amount to:

$$\frac{n}{v_1} \times \frac{n}{v_2} \times \frac{n}{v_3} \times v_1 \times v_3 = \frac{n^3}{v_2}$$

Similarly, the memory access for matrix **B** involves loading tiles of size $v_2 \times v_3$ with the same number of iterations. The total read operations for **B** are:

$$\frac{n}{v_1} \times \frac{n}{v_2} \times \frac{n}{v_3} \times v_2 \times v_3 = \frac{n^3}{v_1}$$

For matrix **C**, each element is written to memory exactly once, requiring n^2 write operations.

Notably, the tiling factor v_3 does not influence the memory access cost for $\mathbf{A}, \mathbf{B}, \text{or } \mathbf{C}$. This implies that v_3 can be chosen arbitrarily without impacting the memory performance of the algorithm.

The number of registers required corresponds to the size of the tiles $\mathbf{a}, \mathbf{b}, \text{and } \mathbf{c}$, which are used during computation. Specifically:

- **a** requires $v_1 \times v_3$ registers
- **b** requires $v_2 \times v_3$ registers
- **c** requires $v_1 \times v_2$ registers

As a result, the total number of registers requires is:

$$v_1 \times v_3 + v_2 \times v_3 + v_1 \times v_2$$

This indicates that the choice of tiling factors $v_1, v_2, \text{and } v_3$ is constrained by the total number of registers available on each core.

When comparing register-tiled matrix multiplication to the basic implementation of matrix multiplication (discussed in the previous section), a clear trade-off between space and time emerges. The basic implementation requires only three registers, making it minimal in terms of space usage. However, it is significantly slower, as it performs a total of n^3 read operations for matrices **A** and **B**. In contrast, register-tiled matrix multiplication leverages more registers to store tiles of $\mathbf{A}, \mathbf{B}, \text{and } \mathbf{C}$, enabling extensive data reuse. This reduces redundant memory accesses, significantly improving I/O efficiency and, consequently, computational performance.

2.3 Cache-aware Tiled Matrix Multiplication

In real-world computing systems, the memory hierarchy consists of multiple levels designed to optimize data access speeds. These levels include registers, the L1 cache, and DRAM. The L1 cache plays a critical role in bridging the performance gap between the fast registers and the slower DRAM. Its proximity to the processor allows for faster access to frequently used data, reducing latency and improving overall performance.

A key optimization technique for utilizing the L1 cache efficiently is **tiling**, which involves dividing data into smaller blocks, or tiles, that fit into the cache. This enables the reuse of data in the cache, minimizing data transfer from the slower DRAM. Specifically, two tiling strategies are commonly applied:

1. **Cache-Aware Tiling:** This strategy focuses on efficiently transferring data between DRAM and the L1 cache.

2. **Register-Aware Tiling:** This strategy deals with moving data from the L1 cache to the processor registers **(as discussed in the previous section)**

In this context, cache-aware tiling targets improving data locality and reducing cache misses when accessing large datasets, such as matrices involved in matrix multiplication or other computational tasks.

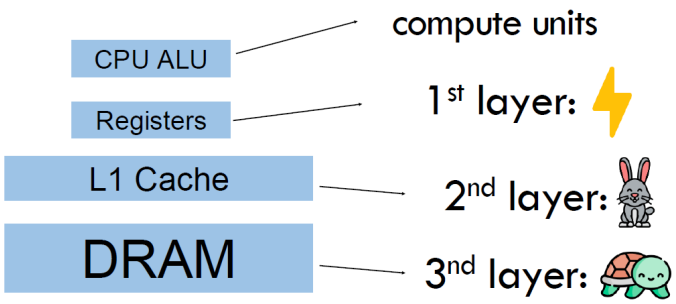

Figure 7: Memory Hierarchy including L1 Cache

2.3.1 Data Movement and Computation

In cache-aware tiling, we introduce two tiling parameters: b_1 and b_2, which define the tile sizes for matrix multiplication. These parameters determine how the data is divided between the L1 cache and the registers. Based on previous analysis, it is noted that the third dimension, b_3, has little impact on performance in this case and is therefore set to $b_3 = 1$.

The workflow for matrix multiplication involves:

1. **DRAM to L1 Cache:** Data is loaded in tiles from DRAM to the L1 cache using cache-aware tiling. This reduces the number of memory accesses to the slower DRAM.

2. **L1 Cache to Registers:** After data is in the L1 cache, it is further divided and transferred to the processor registers for computation. This step is handled by register-aware tiling.

3. **Dot Product Computation:** A dot product is computed between a row of matrix A and a column of matrix B. This is done using data that resides in the registers, but the intermediate steps of transferring data from the L1 cache to the registers are essential to minimize access to slower memory.

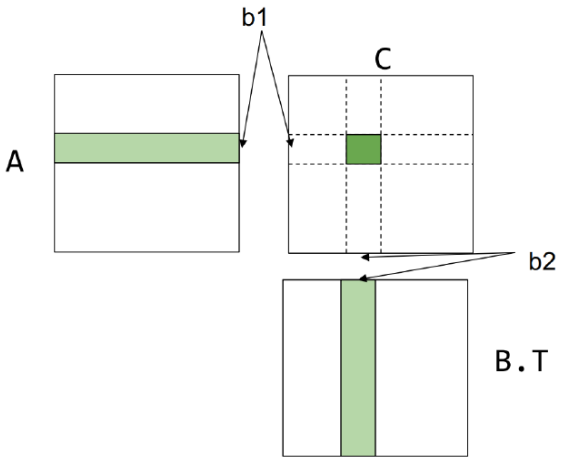

Figure 8: Dot product computation

The code for implementing cache-aware tiling for matrix multiplication is as follows:

```
dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];

for (int i = 0; i < n/b1; ++i) {
    l1cache float a[0:b1, 0:n] = A[i];
    for (int j = 0; j < n/b2; ++j) {
        l1cache float b[0:b2, 0:n] = B[j];
        C[i][j] = dot(a, b.T);
        // Register-aware tiling can be applied here for further optimization
    }
}
```

Listing 2: Cache-Aware Tiling Implementation

2.3.2 Cost Analysis

Understanding the cost of data movement between different memory levels is crucial to optimizing the tiling process. The cost is typically measured in terms of the number of memory accesses required to transfer data from DRAM to the L1 cache and from the L1 cache to registers.

1. Reads Between DRAM and L1 Cache for Matrix A:
   The number of reads required to load matrix A into the L1 cache can be calculated by considering the number of tiles and the size of each tile:
   $$\frac{n}{b_1} \cdot n \cdot b_1 = n^2$$
   This corresponds to the *outer loop* of the tiling process for matrix A, where each tile of A is loaded from DRAM to the L1 cache.

2. Reads Between DRAM and L1 Cache for Matrix B:
   Similarly, the number of reads required for matrix B is:

$$\frac{n}{b_1} \cdot \frac{n}{b_2} \cdot b_2 \cdot n = \frac{n^3}{b_1}$$

   This represents the memory access cost for matrix B during the tiling process.

2.3.3 Total Cost:

The total cost of transferring data from DRAM to the L1 cache is the sum of the costs for matrices A and B:

$$\text{Total Cost} = n^2 + \frac{n^3}{b_1}$$

This cost reflects the data movement overhead involved in the tiling process. Optimizing this cost is critical to achieving better performance in memory-bound operations.

2.3.4 L1 Cache Size Constraint

The effective use of the L1 cache is limited by its size. To ensure that the tiles of data can fit into the cache, the following constraint must be satisfied:

$$b_1 \cdot n + b_2 \cdot n \leq \text{L1 Cache Size}$$

This constraint ensures that the total memory required to hold the tiles of matrices A and B in the L1 cache does not exceed the cache's capacity. If this constraint is violated, cache misses will occur, leading to performance degradation.

2.3.5 Comparison: Cache-Aware Tiling vs. Untiled Matrix Multiplication

In the untiled implementation of matrix multiplication, space usage in the L1 cache is minimal, as data is loaded directly from DRAM into the registers for computation without intermediate buffering. However, this simplicity comes at a significant performance cost, as the process requires n^3 read operations for both matrices A and B, leading to substantial redundant memory accesses. In contrast, cache-aware tiling leverages the L1 cache to store tiles of A, B, and C, enabling extensive data reuse and reducing the number of redundant memory accesses. Specifically, the total cost of cache-aware tiling is $n^2 + \frac{n^3}{b_1}$, which is significantly lower than the untiled version for reasonable tile sizes (b_1 and b_2). While cache-aware tiling requires additional L1 cache space proportional to the tile sizes ($b_1 \cdot n + b_2 \cdot n$), this trade-off results in significantly improved I/O efficiency and computational performance by reducing data transfer overhead between DRAM and the L1 cache.

2.4 Putting Things Together

Finally, we combine register and cache-aware tiling to optimize matrix multiplication across multiple levels of memory hierarchy. Cache tiling partitions the matrices into chunks that fit into the L1 cache, while register tiling further subdivides these chunks to maximize data reuse in registers. This hierarchical approach improves computational efficiency by balancing data movement across different levels.

```
dram float A[n/b1][b1/v1][n][v1];
dram float B[n/b2][b2/v2][n][v2];

for (int i = 0; i < n/b1; ++i) {
    l1cache float a[b1/v1][n][v1] = A[i];
    for (int j = 0; j < n/b2; ++j) {
        l1cache float b[b2/v2][n][v2] = B[j];
        for (int x = 0; x < b1/v1; ++x) {
            for (int y = 0; y < b2/v2; ++y) {
                register float c[v1][v2] = {0};
                for (int k = 0; k < n; ++k) {
                    register float ar[0:v1] = a[x][k][:];
                    register float br[0:v2] = b[y][k][:];
                    C += dot(ar, br.T);
                }
            }
        }
    }
}
```

Listing 3: Combining Register and Cache Tiling

The above code demonstrates how data is moved and processed across different levels of the memory hierarchy.

- **Cache Tiling (Outer Loops):** Parameters b_1 and b_2 are used to divide the computation into tiles that fit into L1 cache, reducing the need to repeatedly fetch data from DRAM.
- **Register Tiling (Inner Loops):** Parameters v_1 and v_2 further divide the tiles into smaller chunks that fit into registers, minimizing L1 cache to register traffic.

2.4.1 Cost Analysis

To evaluate the efficiency of this tiling strategy, we analyze the computational and memory costs at different levels:

DRAM → L1 Cache

$$\text{Cost} = \frac{n}{b_1} \cdot \frac{b_1}{v_1} \cdot n \cdot v_1 = n^2$$

This formula represents the cost of moving data from DRAM to L1 cache. The tiling ensures that we only fetch the required tiles from DRAM, reducing the total memory traffic.

L1 Cache → Registers:

$$\text{Cost} = \frac{n}{b_1} \cdot b_1 \cdot \frac{n}{b_2} \cdot \frac{b_2}{v_2} \cdot n \cdot v_1 = \frac{n^3}{v_2}$$

Register tiling reduces the memory operations between L1 cache and registers. By optimizing v_1 and v_2, we can further lower this cost:

$$\text{Optimized Cost} = \frac{n^3}{v_1}$$

2.4.2 Practical Considerations in CPUs

Modern CPUs follow a hierarchical memory model:

$$\text{Disk} \rightarrow \text{DRAM} \rightarrow \text{L2 Cache} \rightarrow \text{L1 Cache} \rightarrow \text{Registers}$$

Key Questions for Optimization:

- **Parameter Selection:** How do we choose the tiling parameters `v1`, `v2`, `b1`, `b2`, `c1`, `c2`?
  - `v1`, `v2` are register tiling parameters and should fit within the available registers.
  - `b1`, `b2` are cache tiling parameters and must fit within L1 or L2 cache.
- **Concurrent Reads:** Can we overlap memory operations, such as:
  - DRAM → L2 transfers with L2 → L1 transfers?
  - L1 → Registers?

Strategies for Concurrency:

- Non-blocking caches and hardware/software prefetching can enable overlapping data movement at different levels.
- Pipelining memory transfers and computation ensures efficient resource utilization.

2.4.3 Why Tiling Works: Reuse Loading

Tiling improves performance by reusing data across multiple computations, thereby reducing redundant memory accesses.

```
float A[n][n];
float B[n][n];
float C[n][n];

C[i][j] = sum(A[i][k] * B[j][k], axis=k);
```

In matrix multiplication, accessing `A[i][k]` is independent of the j-dimension. By tiling along the j-dimension:

- We can reuse `A[i][k]` for multiple computations of `C[i][j]`.
- This reduces the number of times `A` needs to be fetched from memory.

Benefits of Reuse

- **Reduced Memory Bandwidth:** Fewer fetches from DRAM or cache.
- **Increased Cache Efficiency:** Working sets fit into cache, reducing cache misses.
- **Improved Performance:** More time is spent on computation rather than memory access.

Tiling is a powerful optimization strategy that balances computation and memory reuse by leveraging data locality at multiple levels of the memory hierarchy. By carefully choosing the tiling parameters, we can:

- Minimize redundant memory accesses.
- Maximize data reuse in cache and registers.
- Ensure efficient utilization of modern CPU architectures.

3 The Overall Trend of Hardware Accelerator

3.1 Recap: CPU Parallelization (How to make MatMul faster?)

3.1.1 Parallelization

Modern CPUs often include parallelized instructions . These are designed to operate on multiple data elements in a single instruction (using many concurrent cores). This technique is referred to as SIMD (Single-Instruction Multiple-Data).

3.1.2 Parallel Loops on CPU

A simple C/C++ loop can be parallelized using OpenMP. For instance:

```
# pragma omp parallel for
for (int i = 0; i < 64; ++i) {
    float4 a = load_float4(A + i*4);
    float4 b = load_float4(B + i*4);
    float4 c = add_float4(a, b);
    store_float4(C * 4, c);
}
```

Listing 4: Example of OpenMP parallel for with vector operations

- **Motivation:** By using one instruction that processes multiple pieces of data in parallel,, we can improve throughput significantly for arithmetic-heavy workloads like matrix multiplication, dot products, or element-wise array operations.

3.2 Single-Instruction Multiple-Data (SIMD)

SIMD is a computational paradigm where one instruction is issued, but it operates on multiple elements of data in parallel. In CPU designs, the vector unit typically contains multiple Arithmetic Logic Units (ALUs) that process multiple data elements concurrently.

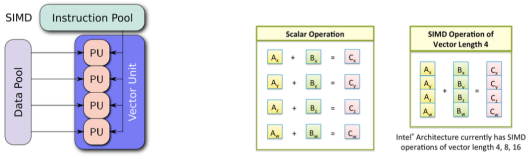

Figure 9: Single Instruction Multiple-Data

CPU is not good at SIMD task and this is related to how CPU is designed.

3.3 Chip Design Trajectory: SIMD

Historically, CPU manufacturers have gradually increased the width of vector instructions. For example: As transistor sizes shrink, more ALUs can be placed on the same chip area (barring power constraints). This has driven the trend of building larger vector units, effectively doubling or quadrupling the parallelism that can be exploited within a single cycle. However, power and heat limitations eventually arise.

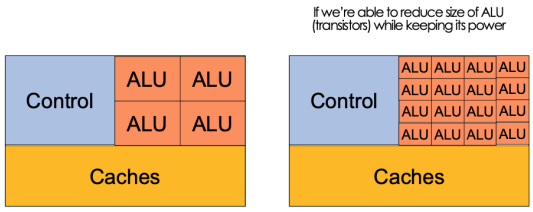

Figure 10: CPU evolved with more and more ALU (cores) units

3.4 Limitations and the Road Ahead

- **Power wall:** Even if we add more ALUs, we risk overheating and surpassing feasible power budgets.
- **Diminishing returns:** Doubling the vector width does not always translate to a 2× speedup in real workloads (due to memory bandwidth, cache efficiency, etc.).

3.5 The End of Moore's Law and the Chip Industry Solution

3.5.1 Moore's Law Slowdown

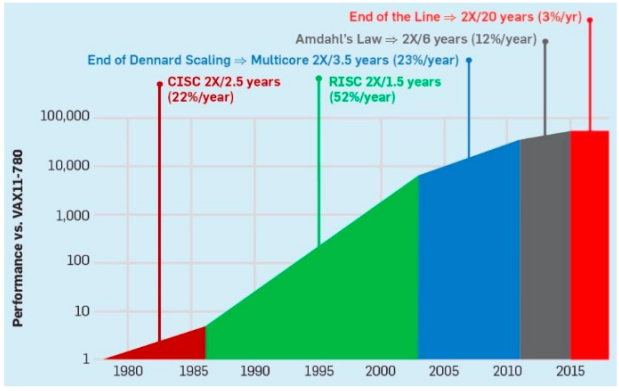

Figure 11: Moore's law comes to an end

Moore's Law (the observation that the number of transistors on a chip doubles approximately every 18–24 months) has fueled the past decades of CPU performance gains. However, transistor scaling has run into physical limitations:

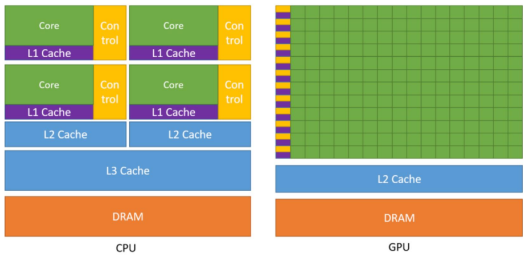

Figure 12: Comparison between GPU and CPU (be aware how many cores GPU possess compares to CPU)

- **Sub-10nm Fabrication:** Manufacturers are moving from 7nm to 5nm, 3nm, and even smaller processes, but the cost of these photolithography steps is immense. Leakage current, heat dissipation, and quantum effects complicate matters.
- **Dennard Scaling's End:** Increasing clock speeds is no longer feasible at the same rate, due to voltage and power density issues.

3.5.2 Industry Direction

The chip industry is exploring two paths:

1. **Quantum Computing:** Potentially a future direction, though currently limited to very specialized, small-scale problems.
2. **Specialized Hardware:** Emphasizing more **domain-specific accelerators** (e.g., GPUs, TPUs).

These specialized solutions often revolve around matrix or tensor operations, particularly relevant for AI/ML workloads that frequently rely on linear algebra.

3.6 Hardware Accelerators: GPUs and Beyond

3.6.1 Graphics Processing Units (GPUs)

- **Idea:** Use a huge number of simpler, specialized ALUs that focus on throughput rather than single-thread performance. This is often described as "SIMD on steroids."
- **Origins:** NVIDIA popularized GPUs for desktop graphics and gaming in the early 2000s.
- **Deep Learning Boom:** GPUs became central to machine learning frameworks once it was discovered that matrix multiplications for neural networks map extremely well to the GPU's parallel architecture.
- **CUDA:** NVIDIA's proprietary parallel computing platform and API (introduced in 2007) made it more accessible for non-graphics tasks. High-level libraries such as cuDNN, cuBLAS, cuSparse further simplified ML integration.

3.6.2 Other Accelerators (ASICs, FPGAs, TPUs)

Beyond GPUs, more specialized accelerators have emerged:

- **Tensor Processing Unit (TPU):**
  - Created by Google as an ASIC specifically for machine learning inference and training.
  - Used in products like AlphaGo and widely available in Google Cloud.
  - Focuses on matrix multiply–accumulate (MAC) operations, often in reduced precision (e.g., `bfloat16`, `fp8`).
- **Future Projections:**
  - *Example:* B200 (projected release in 2025) with `fp4` / `fp8` tensor cores for deep learning.
  - *Example:* Next-gen Apple "M" series chips that tightly integrate specialized matrix hardware alongside traditional CPU cores.

4 Case Studies

4.1 What Does It Mean by "Specialized" In accelerator world

Functionality-specialized

- We can produce chips that are only good at MatMul operations like TPUs, or that are only good at certain computations with sparsity.
- This also includes development of chips which are a mix of CPU and GPU cores. An example of this is Apple's chips, which integrate many different cores, some which are specialized in video processing, and some in deep learning into a single board.

Reduce Precision

- Computations on a shorter floating point representation like fp8 and int8 are faster.
- NVIDIA strongly advocates for reducing precision from fp32 (used most commonly in older systems) to fp4 and fp8.

Tune the distribution of configuration of the board

- This includes making decisions like, how many L1, L2 caches, or memory to include while performing a MatMul tiling computation, and striking a balance between these.
- Some companies in Silicon Valley build chips which do well on certain tasks by giving up memory, i.e a chip with just the L1 cache.

FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core*	989 teraFLOPS
BFLOAT16 Tensor Core*	1,979 teraFLOPS
FP16 Tensor Core*	1,979 teraFLOPS
FP8 Tensor Core*	3,958 teraFLOPS
INT8 Tensor Core*	3,958 TOPS
GPU Memory	80GB
GPU Memory Bandwidth	3.35TB/s
Decoders	7 NVDEC 7 JPEG
Max Thermal Design Power (TDP)	Up to 700W (configurable)
Multi-Instance GPUs	Up to 7 MIGS @ 10GB each
Form Factor	SXM
Interconnect	NVIDIA NVLink™: 900GB/s PCIe Gen5: 128GB/s
Server Options	NVIDIA HGX H100 Partner and NVIDIA-Certified Systems™ with 4 or 8 GPUs NVIDIA DGX H100 with 8 GPUs
NVIDIA AI Enterprise	Add-on

* With sparsity

Figure 13: NVIDIA H100 GPU Product Sheet

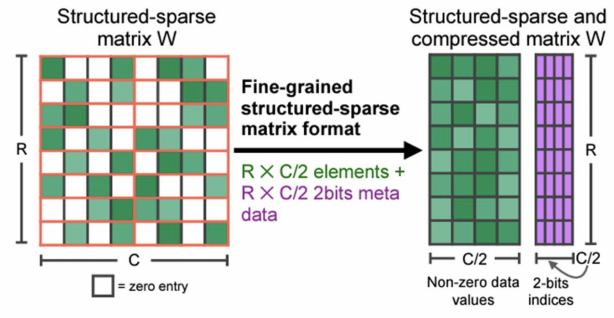

Figure 14: Inducing sparsity in computations

4.2 Case Study 1: NVIDIA GPU Specification

Fig. 13 shows the GPU specification for NVIDIA's H100 GPU. This GPU is specialised for deep learning tasks.

The first few rows of this sheet represent the computing power of this card. Computing power on different cores is different. The second highlighted core is a tensor core, and is good for tensor-related operations. The other highlighted core is a normal core, but which has the same precision as the tensor core, fp32. However, with the Tensor core, we can achieve almost 1000 TeraFLOPS (TeraFLOPS is the unit that describes the computing power, higher the better), while with the normal core, compute is 10 times slower. The other rows of this sheet describe FLOPS for different precision.

Another important specification in this sheet is the Memory Bandwidth, which describes the speed to move things from GPU memory to cache/register. This number determines how fast it is to move a floating point across memory hierarchy.

We can see that some terms come with an asterick above them. This indicates that if we run a sparsified computation, we can achieve the peak FLOPS mentioned. But without sparsity, we can only get half those FLOPS mentioned.

What is sparsity? If we are able to write a kernel/function that always performs computation subject to a constraint that every row will only have half the elements and the rest of them are empty, we will then get the ideal FLOPS, and this is what we refer to as sparsity. However, this is a hard task, to convert our original computation to what is shown in the Fig. 14.

We can observe in the specification that for different precisions in the computation, the peak FLOPS also differ. This is fundamental for quantization, where we attempt to quantize the precision all the way down from fp32 to fp8. If we can quantize our original computation to a lower precision, we can benefit from more powerful computing peak FLOPS, thus leading to faster programs. This brings about the question: In applications which do not involve machine learning, reducing precision leads to more erroneous calculations. So why can this work in ML programs while it does not in other scenarios?

4.3 Case Study 2: Apple Silicon

It seems the apple chip can do a lot of things, including video machine learning and graphics. But how they acheive this? Apple M3 chip is shown in Fig. 15. In the upper-left corner, they place a 16-core CPU. Then, for the rest of the space, they place some control units or caches. In the area below, they place a 40-core

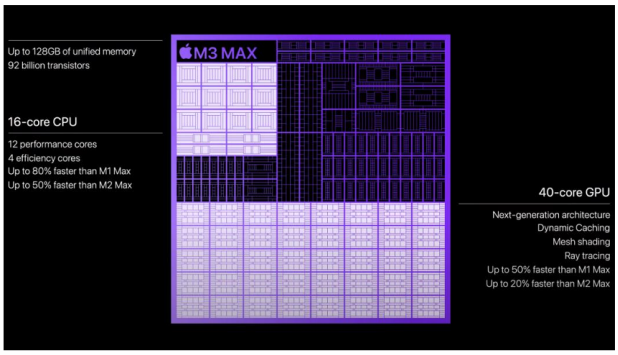

Figure 15: Apple Chip

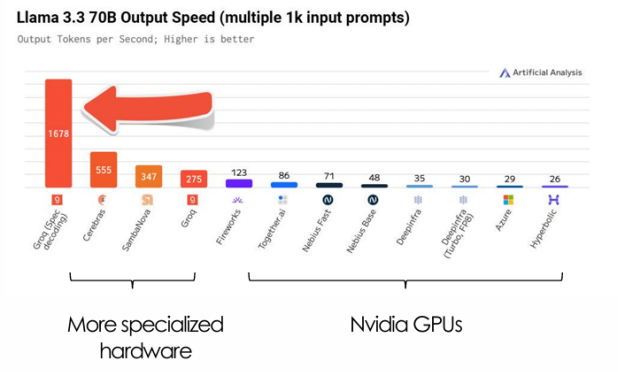

Figure 16: Performance Chart

GPU. So, the M3 chip is mixing some versatile CPU cores with some less versatile GPU cores.Since apple is software hardware company, they can optimize their operating system in a way that if something to be executed is related to machine learning, the program will be dispatched to on GPU cores, while something common like adding a document is executed, it will be dispatched on CPU cores.

4.4 Case Study 3: Leading Chip Startups

The third case is about leading chip startups. Groq, Cerebras, and SambaNova are three companies that are trying to manufacture the next generation of chips optimized for machine learning. They are working on tuning the configuration of cache size, memory, and register size to best suit certain workloads, such as language model inference. The performance chart in Fig. 16 shows a workload of generating tokens from Llama. The Groq hardware can achieve almost 10 times the speed of the other companies using Nvidia GPUs.

The reason it can achieve this can be found in Fig. 17. The key difference between the Groq card and the Nvidia chip is the SRAM. SRAM is similar to L1 cache, which is a layer in the memory hierarchy. It is faster than memory but slower than a register. The Groq card has 230MB of SRAM, whereas the latest Nvidia chip only has 164KB of SRAM. This is beneficial because the matrix can be tiled into a larger size, which will be faster.

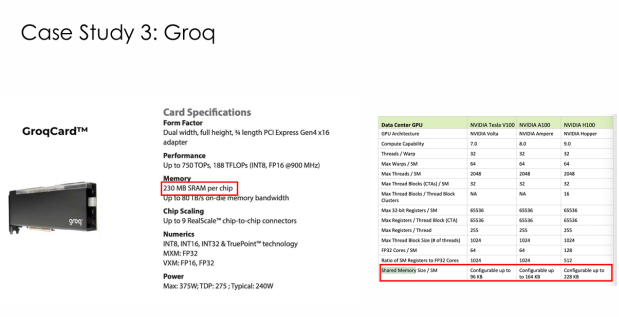

Figure 17: Groq card

5 Contribution

- Zaiyang Zhang: Section 1
- Anay Kulkarni: Section 1
- Mingrui Yin: Section 2
- Peiyun Han: Section 2
- Yu-Pao Tu: Section 2.2, compiling
- Shyam Nuggehalli: Section 2.2
- Prabhleen Kaur: Section 2.3
- Pritika Barshilia: Section 2.3
- Cheril Shah: Section 2.4
- Ziqiao Xi: Section 3
- Luting Lei: Section 4
- Spoorthi Kalkunte: Section 4