CSE 291A/DSC 291: Data Systems for Machine Learning, Spring 2026

4: Tensor Formats, MatMul, Accelerators

Lecturer: Hao Zhang *Scribes: Samar Karanch, Kevin Lin, Owen Park, Shaurya Raswan, Julie Wu, Jaewoong Yun*

1 Arithmetic Intensity (AI)

Arithmetic Intensity is a measure of how compute-heavy an operation is relative to how much data needed to fetch from memory to perform that computation.

Motivating Example:

In standard multi-head attention, to get the Q, K, and V matrices, you need to compute each separately (using matrix multiplication to obtain $Q = XW_Q$, $K = XW_K$, $V = XW_V$).

Instead, you can speed up this process by performing a merged QKV by a single matrix multiplication operation (XW_{QKV} and split into 3 matrices).

1.1 Arithmetic Intensity Formula:

$$\text{AI} = \#\text{ operations } / \#\text{ bytes}$$

This formula represents the ratio of compute work to memory work. A high AI value means that many computations are performed for each byte of memory read, while a low AI value means the opposite.

Transformer operations usually have low AI as entire matrices must be loaded in from memory while performing low computation, like dot products. As a result, GPU compute time is wasted on idling while waiting for data to be loaded.

The practical goal is to raise AI enough so that memory is not a bottleneck in computing speed.

1.2 Low AI Example 1:

```
void add(int n, float *A, float *B, float *C) {
    for (int i = 0; i < n; i++) {
        C[i] = A[i] + B[i];
    }
}
```

Simple add() kernel

The code computes the $A[i] + B[i]$ operation n times via the for loop.

It also loads $A[i], B[i]$ from memory n times each and stores $C[i]$ to memory n times each, resulting in $3n$ computations total.

According to the AI formula, we have # operations as n, and # memory loads as $3n$. Therefore, $AI = \frac{n}{3n} = \frac{1}{3}$.

1.3 Low AI Example 2:

```
void add(int n, float* A, float* B, float* C){
    for (int i=0; i<n; i++) {
        C[i] = A[i] + B[i];
    }
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++) {
        C[i] = A[i] * B[i];
    }
}

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);
```

Simple add() and mul() kernel

Both the add and multiply operations are $\frac{1}{3}$, as explained in Example 1. If we call the add operation twice and multiply operation once, we have $3n$ total computations and $9n$ total loads and stores, so $\frac{2n}{6n} = \frac{1}{3}$.

Therefore, AI is still $\frac{1}{3}$.

This is wasteful, as we see *tmp*1 being written to memory during add(n, A, B, tmp1), and loaded from memory during mul(n, tmp1, C, tmp2). The same process applies to *tmp*2. Why not keep *tmp*1 or *tmp*2 in memory for quicker processing?

1.4 More Efficient Example (Kernel Fusion):

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

Fused kernel

We can instead combine all compute operations in one kernel as shown above.

Now we have the following:

$\# \, Operations = 3n$ (2 additions, 1 multiplication for each loop iteration)

$\# \, Bytes \; Loaded = 5n$ (For each iteration, load $A[i], B[i], C[i], D[i]$, and store once to $E[i]$)

AI = 3/5. Higher AI! Since we lowered the amount of memory accesses, we have sped up our kernel processing time.

2 Layers of Optimization Overview

2.1 Graph Optimization

In order to improve performance (e.g., increase arithmetic intensity and reduce memory access), we can optimize a data flow graph by rewriting it, often by fusing operations together. At a high level, this can be done through:

- Writing Rules/Template: Engineers manually define optimization rules. For example, if the graph has $Conv \rightarrow ReLU$, we can fuse them into a single operation.
- Auto Discovery: The system automatically generates different valid versions of the computation graph (e.g., with different fusion or execution strategies), evaluates their performance using a cost model or by profiling, and selects the most efficient one.

2.2 Parallelization

Another way to improve performance is to parallelize the graph compute over multiple devices. In order to do this, we have to consider the following:

- How to partition? How to split the computation graph across multiple devices (e.g., by data, model, or operators).
- How to communicate? How devices exchange intermediate results (e.g., gradients, activations) efficiently with minimal overhead.
- How to schedule? How to order and overlap computation and communication to maximize hardware utilization and reduce idle time.
- Consistency: How to ensure all devices maintain a correct and synchronized view of the model/parameters during training.
- How to auto-parallelize? How to automatically decide partitioning, communication, and scheduling without manual tuning.

2.3 Runtime: Schedule/Memory

Another way to improve performance is through runtime scheduling, i.e., deciding when to execute computation, communication, and memory operations. The goal is to:

- Schedule compute, communication, and memory operations in an order that minimizes total execution time.
- Overlap communication with computation whenever possible to avoid idle hardware.
- Respect memory constraints, making sure that intermediate results fit within available memory.

2.4 Operator Implementation

Lastly, we can improve performance by optimizing the implementations of individual operations such as such as matrix multiplication (MatMul) and convolution (Conv2d). This involves:

- Adapting implementations to different hardware platforms (e.g., V100, A100, H100, mobile devices, TPUs), each with different constraints.
- Supporting different numerical precisions (e.g., fp32, fp16, fp8, fp4; fp = floating point).
- Handling different input shapes and variants (e.g., conv2d_3x3, conv2d_5x5, 2D/3D matmul, attention), which may require specialized implementations.

3 How to Make Operators Fast in General?

3.1 Vectorization

```
float A[256], B[256], C[256]
for (int i = 0; i < 256; ++i) {
    C[i] = A[i] + B[i]
}
```

Unvectorized

```
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 + i* 4, c);
```

Vectorized

Vectorization speeds up performance by allowing each instruction to process multiple data elements in parallel using SIMD hardware. It also reduces instruction overhead, since there are fewer instructions overall.

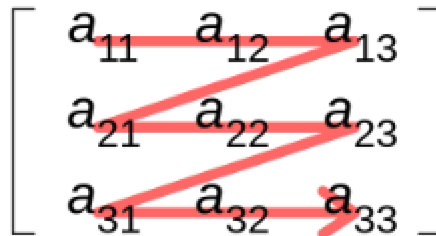
3.2 Data Layout

Data layout describes how multi-dimensional data (like matrices or tensors) is arranged in linear memory. Although we index data with multiple dimensions, it is stored as a 1D array, and the layout determines how indices map to memory locations.

3.2.1 Row-Major and Column-Major

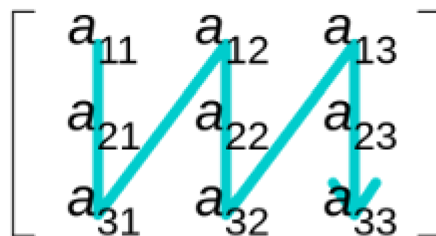
Row-major means elements in each row are stored contiguously in memory, and rows are placed one after another. The indexing formula is as follows:

$$A[i, j] = A.data[i \cdot A.shape[1] + j]$$



Column-major means elements in each column are stored contiguously in memory, and columns are placed one after another. The indexing formula is as follows:

$$A[i, j] = A.data[j \cdot A.shape[0] + i]$$



In order to make a program efficient, it is important to consider how the matrix is stored. For example, consider the following program, assuming row-major layout:

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}
```

Summing a 2D array

In this program, the loop iterates over columns first (index j) and then rows (index i), accessing elements in the order $a[0][j]$, $a[1][j]$, $a[2][j]$, and so on.

For row-major layout, where elements in each row are stored contiguously, this results in a strided (non-contiguous) memory access pattern, leading to poor cache utilization and inefficient use of memory bandwidth.

A more efficient approach is to iterate over rows first and columns second, so that elements are accessed sequentially within each row. This contiguous access pattern improves cache locality and overall performance.

3.2.2 Strides

Most modern ML systems represent tensors using the stride format, which separates the underlying storage and the view of the tensor. In this format, each dimension has a stride that specifies how many elements to skip in memory when that index increases, along with an offset indicating the starting position. This representation is especially powerful for tensor operations, as many transformations (e.g., slicing or transposing) can be implemented as zero-copy operations by simply adjusting the strides and offset, without moving data. The indexing formula is:

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

Operator: Slice Used to get particular rows and columns of a tensor. The naive way is to copy the tensor and get the values you want. However, with strides you can reuse the same existing memory where the tensor already exists. Thus, a slice operation can be implemented as a zero-copy operation by adjusting the offset and shape of the tensor. Specifically, for a slice that starts at index s along dimension i:

- Update the offset: $A.\text{offset} += s \cdot A.\text{strides}[i]$
- Update the shape: $A.\text{shape}[i] = \text{new_size}$ (where new_size is the size of the slice along dimension i)
- Strides remain unchanged since the memory layout is the same, just with a different starting point and size.

However, for a step slice (e.g., $A[::2]$), we also need to update the stride:

- Update the offset: $A.\text{offset} += s \cdot A.\text{strides}[i]$
- Update the shape: $A.\text{shape}[i] = \text{new_size}$ (where new_size is the size of the slice along dimension i, accounting for the step size)
- Update the stride: $A.\text{strides}[i] *= \text{step}$ (where step is the stride of the slice, e.g., 2 for $A[::2]$)

Operator: Transpose A transpose operation can be implemented as a zero-copy operation by modifying the tensor's metadata without changing the underlying data. Specifically, transpose swaps the corresponding dimensions in both the shape and strides.

For a transpose between dimensions i and j:

- Swap $A.\text{shape}[i]$ and $A.\text{shape}[j]$
- Swap $A.\text{strides}[i]$ and $A.\text{strides}[j]$

Operator: Broadcast To broadcast a tensor, we modify the shape and strides while keeping the offset the same. The general rule for broadcasting along dimension k is:

- If the original size is 1 and expanded to size N, set $A.\text{shape}[k] = N$
- Set $A.\text{strides}[k] = 0$

Operator: Swap Tiles To swap tiles of a tensor, we reinterpret the tensor as a grid of tiles and then change how tiles are traversed via shape and strides. The general rule is:

- First reshape the tensor to separate tile dimensions (i.e., split each dimension into number of tiles and tile size)
- Swap the tile-related dimensions in A.shape
- Swap the corresponding entries in A.strides

Example: Consider a 4×4 tensor stored in row-major order:

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

Divide the tensor into 2×2 tiles. Then:

- Original shape: $[4, 4]$, strides: $[4, 1]$
- Reshape to: $[2, 2, 2, 2]$ (tile_row, tile_col, inner_row, inner_col)
- New strides: $[8, 2, 4, 1]$

To swap tiles, swap the tile dimensions (first two axes):

- New shape: $[2, 2, 2, 2]$
- New strides: $[2, 8, 4, 1]$

Resulting layout:

$$\begin{bmatrix} 0 & 1 & 8 & 9 \\ 4 & 5 & 12 & 13 \\ 2 & 3 & 10 & 11 \\ 6 & 7 & 14 & 15 \end{bmatrix}$$

Problems of Strides While the stride format is flexible and enables zero-copy operations, it can introduce inefficiencies. In particular:

- Memory access may become non-contiguous, leading to poor cache utilization
- Many vectorized operations require contiguous memory, so non-contiguous tensors may need to be copied

3.3 Parallelization

Parallelization speeds up computation by running independent operations concurrently across multiple processing units. In many numerical workloads, loop iterations are independent and can be executed in parallel. For example:

```
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 + i* 4, c);
}
```

Vectorized

Here, vectorization processes multiple elements at once using SIMD instructions (e.g., `float4`), providing data-level parallelism. However, the loop still executes sequentially and does not utilize thread-level parallelism.

```
#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 + i* 4, c);
}
```

Vectorized & Parallelized

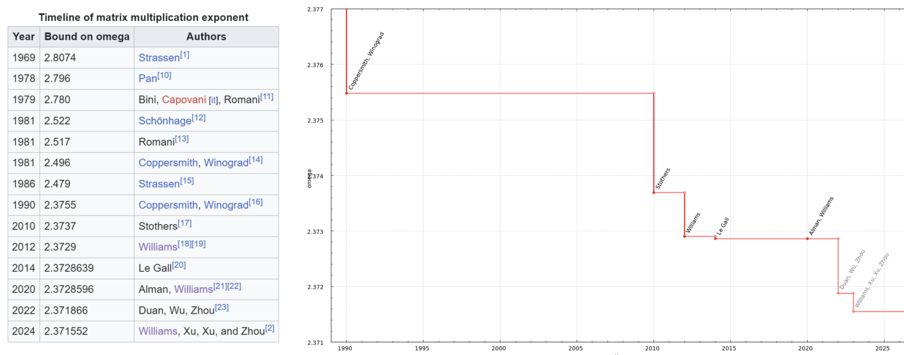
By adding `#pragma omp parallel for`, the loop iterations are distributed across multiple threads, enabling thread-level parallelism. Since each iteration is independent, this is safe and can significantly improve performance.

4 MatMul-Specific Optimization

This section goes over how we can speed up an important operation in neural networks: Matrix Multiplication.

4.1 MatMul Complexity

MatMul normally takes $O(N^3)$ complexity, but advanced algorithms can reduce computation time to $O(N^{2.371552})$.



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]

To make matmul fast, we increase the number of operations while decreasing the number of bytes loaded from memory. Ideally, we want everything to be local to processors (in registers), but registers are expensive and small, so we have memory hierarchy.

Consequently, we have the following memory hierarchy, from fastest to slowest: CPU Thread < Registers < L1 Cache (0.5 ns) < L2 Cache (7 ns) < DRAM (200 ns)

4.2 Slow Operation Example of MatMul:

```
dram float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        register float c = 0;
        for (int k = 0; k < n; ++k) {
            register float a = A[i][k];
            register float b = B[j][k];
            c += a * b;
        }
        C[i][j] = c;
    }
}
```

This is a standard matrix multiplication that uses 3 registers. The read cost is high since $A[i][k]$ and $B[j][k]$ both read from DRAM. Since there is a triple nested for-loop, there are $2n^3$ read operations when reading from matrices A and B. Additionally, there are n^2 write operations to C.

4.2.1 How to Make MatMul Fast:

In order to make matrix multiplication fast, we need to maximize Arithmetic Intensity (i.e. keep the number of operations high while decreasing the number of bytes).

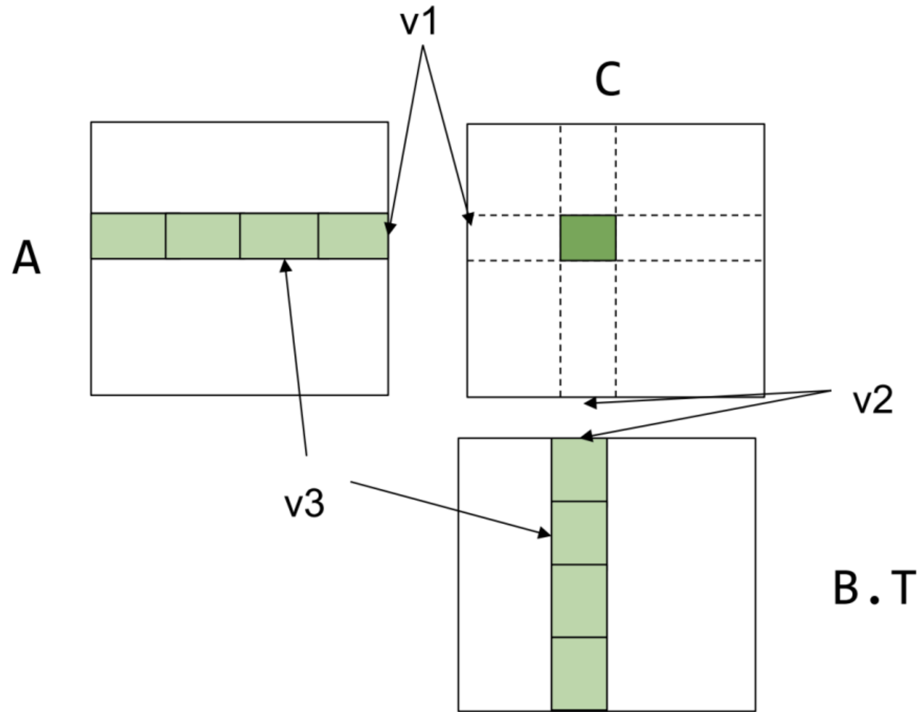
How do we do this? We can do this by reusing the data loaded into the **registers** and reduce the number of **DRAM** reads. Since registers are local to processors, registers extremely fast when reading/writing data (essentially zero latency). On the other hand, DRAM is much slower. However, registers are expensive and small in memory size, so we need to find a way to use these efficiently.

To reuse data in registers, we can use approaches like tiling, which reuses data from registers instead of accessing DRAM memory, as shown in the next section.

4.3 Register Tiled MatMul

```
dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
    for (int j = 0; j < n/v2; ++j) {
        register float c[v1][v2] = 0;
        for (int k = 0; k < n/v3; ++k) {
            register float a[v1][v3] = A[i][k];    // index 4d array to get block shape [v1,
  v3]
            register float b[v2][v3] = B[j][k];    // like b[0:v2, 0:v3] = B[j][k]
            c += dot(a, b.T);
        }
        C[i][j] = c;
    }
}
```



Main Idea: We update our for loop to iterate by block/tile instead of one single element. This reduces read costs since we read each tile once and re-use it across multiple iterations.

$v1$, $v2$, and $v3$ represent how big the tiles are. For example, if $v1 = 2$, $v2 = 2$, $v3 = 2$, then the tile of A is $2 \cdot 2$, tile of B is $2 \cdot 2$, and output tile of C is $2 \cdot 2$. This is what we load and write into each register for computation.

Recall that in our slower standard MatMul, each iteration loads one element at a time (i.e., register $a = A[i][k]$, where $A[i][k]$ is one element). For our tiled version, **register "a"** loads a $(v1 \cdot v3)$ tile at a time ($a[v1][v3] = A[i][k]$, where $A[i][k]$ is an entire $v1 \cdot v3$ block).

Instead of computing one by one, we are now batching a many elements and performing dot products for each tiled batch.

Now, let's re-compute the cost:

Our total length for the triple-nested for loop is now of size $\frac{n}{v1} \cdot \frac{n}{v2} \cdot \frac{n}{v3}$, which simplifies to $\frac{n^3}{v1 \cdot v2 \cdot v3}$.

Breakdown per loop iteration (i, j, k):

- Load one tile from A of size $v1 \cdot v3$
- Load one tile from B of size $v2 \cdot v3$

Thus, the number of total reads from A is:

$$\frac{n^3}{v1 \cdot v2 \cdot v3} \cdot (v1 \cdot v3) = \frac{n^2}{v2}$$

Total number of reads from B is:

$$\frac{n^3}{v1 \cdot v2 \cdot v3} \cdot (v2 \cdot v3) = \frac{n^2}{v1}$$

Now for writes to C: for each (i, j) tile we write a $v1 \cdot v2$ output tile once (after accumulating over k). The number of (i, j) tiles is $\frac{n}{v1} \cdot \frac{n}{v2}$, so total writes are:

$$\frac{n}{v1} \cdot \frac{n}{v2} \cdot (v1 \cdot v2) = n^2$$

The total DRAM element traffic is:

$$\frac{n^2}{v2} + \frac{n^2}{v1} + n^2$$

For the write cost, c is still n^2, since we still need to store every element regardless of how we tile. As a result, only the read time is improved.

We also need to increase the number of registers, since each register can only hold one element. Consequently, the number of registers is the sum of our size of our tiles for A $(v1 \cdot v3)$, B $(v2 \cdot v3)$, and C $(v1 \cdot v2)$.

This increase in registers for faster computation is the tradeoff for tiling. We want to set $v1$ and $v2$ to be as large as possible, as long as we have the amount of registers to support it.

4.3.1 Cache-aware Tiling

```
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[b1][n] = A[i];
    for (int j = 0; j < n/b2; ++j) {
        l1cache b[b2][n] = B[j];

        C[i][j] = dot(a, b.T);
    }
}
```

Main Idea: Instead of loading tiles into registers, this loads tiles into the L1 cache, which is slower than registers but faster than DRAM.

The data movement is from DRAM, to dram → L1 cache (cache tiling), to L1 cache → registers (register tiling).

You can fit larger tiles in this L1 cache compared to the registers. As shown in the code, we are now loading the entire row of the matrix A and B since the L1 cache is larger than registers.

We load **L1cache a** $\frac{n}{b1}$ times via for loop, where each load is size $b1 \cdot n$. So, we have n^2 total reads.

We load **L1cache b** $\frac{n}{b1} \cdot \frac{n}{b2}$ times via for loop where each load is size $b2 \cdot n$. So, we have $\frac{n^3}{b1}$ total reads.

Choose $b1$ and $b2$ to be small enough so that the tiles can fit in this L1 cache, similar to how $v1$ and $v2$ is small enough to fit in registers.

4.3.2 Combining both Register Tiling and Cache Tiling

```
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 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[v1] = a[x][k][:];
                    register float br[v2] = b[y][k][:];
                    C += dot(ar, br.T)
                }
            }
        }
    }
}
```

Main Idea: We combine both register and cache tiling to minimize data movement among the memory hierarchy. Cache tiling reduces DRAM reads, and register tiling reduces L1 cache reads.

For the 2 outer loops i and j, we implement a cache tiling level by loading large tiles of A and B into the L1 cache. The cost of reading **L1 cache a** is n^2, **L1 cache b** is $\frac{n^3}{b1}$.

Our 3 inner loops x, y, k take small subsets from this L1 cache and put the values into registers for the dot products to be computed.

By loading values into the L1 cache, we can avoid accessing the DRAM for every multiply operation.

4.3.3 Data Reuse:

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

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

Note that the read cost for A depends on $v2$ and not $v1$, because $A[i][k]$ is independent of j. This means tiling the j dimension by $v2$ allows the same tile of A to be reused $v2$ times, which is exactly where the cost reduction comes from.

Scribe Responsibilities

- Samar Karanch: Section 2, Images
- Kevin Lin: Outline + Code Blocks, Section 3
- Owen Park: Section 2, 3
- Shaurya Raswan: Outline, Proofreading, Images
- Julie Wu: Proofreading
- Jaewoong Yun: Section 1, 4

All work can be tracked through the GitHub repository: https://github.com/TangyKiwi/CSE-291A-Scribe