Harvard CS 242: Computing at Scale (Fall 2024) **(Due: 09/18/2024 at 11:59PM EST)**

Programming Assignment 1

Instructor: Professor H.T. Kung ***Name:*** Jaray Liu

Please include your full name in your submission.

Introduction

Assignment Objectives

The objectives of this assignment are to:

- Introduce the concept of data reuse strategies when performing computations from within a memory hierarchy.
- Illustrate how outer products can be more efficient for matrix-matrix multiplication (MMM) than inner products.
- Demonstrate how arithmetic intensity can affect runtime.

Virtual Machine Setup

- If using a device with Apple Silicon (M1 and M2 laptops) please see this Google Doc for virtual machine setup instructions.
- If using a device with Intel or AMD processors (including older Macbooks from 2020 or earlier), please see this Google Doc for virtual machine setup instructions.

Refresher

Matrix-Matrix Multiplication

We define matrix-matrix multiplication (MMM) as $C = C + A \times B$, where A and B are input matrices of size $M \times K$ and $K \times N$, respectively, and C is the $M \times N$ result matrix. Keep in mind that C being both an input and output means elements of C must be **read from the external memory** into the local memory first before being used and then written back. For simplicity, we assume square matrices throughout this assignment (i.e., $M = N = K$). However, concepts and methods learned from this assignment generalize to non-square matrices. As discussed in lecture, MMM is a fundamental operation underlying many machine learning computations.

Arithmetic Intensity

Arithmetic intensity (α) is the ratio of arithmetic operations performed (or # ops) to number of memory accesses (IO):

$$\alpha = \frac{\#\text{ ops}}{\text{IO}}$$

Note: we will consider IO in units of 4-byte elements (i.e., 32-bit floating point numbers). Arithmetic intensity is an important metric because it captures how much computation may be done for each memory access. We prefer algorithms with a high arithmetic intensity in order to minimize the IO for the same number of operations performed.

Memory Hierarchy and Computation from Local Memory

For simplicity, we will assume a simple memory hierarchy with 2 levels: a **fast** local SRAM and a **slow** external DRAM. We will also assume the local memory (SRAM) is entirely user-managed (i.e., the memory does not have a built-in eviction policy). That is, we can specify what data is held, for whatever length of time, so long as there is sufficient memory capacity.

Computations may only be performed directly on data stored in local memory, so operands (inputs and outputs to be used or written back) must be located in the local SRAM (and not the external DRAM). For any algorithm, final results must be written back to the slower external memory (DRAM), but intermediate values can be either held in faster memory or written back to slower memory.

Analysis Details and Hints

For our analysis, we will follow these guidelines for ease of analysis:

- Individual reads and writes to external memory are considered **separate IO operations**.
- **Initializing a variable in local memory** does not require an IO operation.
- At the start of each question, we assume **fast memory is entirely cleared/zeroed** (all zeroes).
- All final computed values of C should be **written back to external memory** and not just kept in fast memory. (Hint: this means your IO should *always* factor in at least reading and writing C!).
- Multiply-accumulates (MACs) count as **two arithmetic operations** (i.e., one multiplication followed by one addition):

$$z \leftarrow z + x \cdot y \text{ means } \texttt{z += x * y (C code)}$$

Example: Arithmetic Intensity for Incrementing All Elements of a Vector by 1

In the next section, we will be asking you to derive the arithmetic intensity for certain algorithms. For each question, clearly state **where data is being moved** to/from as well as **when the data is being moved** (i.e., the order of data movement). Describe any additional assumptions you make. To simplify analysis, you may express the arithmetic intensity in terms of N.

Here is an example of how to calculate the arithmetic intensity for the following algorithm when local memory can hold only $\frac{N}{2}$ elements:

Algorithm 1: Vector increment

```
// a is a vector with N elements
for n = 1 → N do
| a[n] ⟵ a[n] + 1;
```

We start by reading in the first $\frac{N}{2}$ elements of a from slow external memory into local memory. Next, we add 1 to the value of each element in local memory, and then write them back to external memory. We repeat this for the last $\frac{N}{2}$ elements of a. In total, we read N elements from external memory and write back N elements to external memory, and perform N additions. That is, we have $2N$ accesses to slow memory and N operations. As a result, our arithmetic intensity is:

$$\alpha = \frac{N}{2N} = \frac{1}{2}$$

1 Arithmetic Intensity for an Individual Inner Product

An inner product (also called a dot product) between a and b, two N-dimension vectors with a being a row vector and b a column vector, will result in a single element, as computed by the following algorithm:

Algorithm 2: Inner Product of Two Vectors

$sum \longleftarrow 0$; **for** $n = 1 \rightarrow N$ **do**
  $sum \longleftarrow sum + a[n] \cdot b[n]$;
return sum

Part 1.1 (5 points)

Calculate arithmetic intensity for the inner product computation (Algorithm 2) when local memory can hold 3 elements for the scheme where we keep *sum*, a single element of a, and a single element of b in memory. Reminder: initializing *sum* to zero does not require any reads from external memory, but the final result must be written back to external memory!

(Solution) We start by reading in the first element of a and the first element of b from external memory. Note that the third spot in local memory is reserved for *sum*. Then, we perform a multiplication between the element of a and element of b and add the product to *sum*. We repeat this N times where on the N-th iteration, we also must write *sum* back into external memory. Thus, we read $2N$ elements from external memory, write 1 element into external memory, and perform $2N$ operations. Therefore, our arithmetic intensity is:

$$\alpha = \frac{2N}{2N+1} = \frac{2N}{2N+1}$$

Part 1.2 (5 points)

Calculate arithmetic intensity for the inner product when local memory can hold $\frac{N}{2} + 2$ elements. (Hint: start by bringing in the first half of b.)

(Solution) We start by reading in the first $\frac{N}{2}$ elements of b from external memory to the local memory. The $\frac{N}{2} + 1$-th slot in local memory is reserved for *sum*. Then, we read the first element of a from external memory and multiply it with the first element of b stored in local memory, and finally adding the product to *sum*. Then, we repeat for the next element of a. Note that we only read the first $\frac{N}{2}$ elements from a during this process since the first $\frac{N}{2}$ elements of b are already in local memory. Thus, we read $\frac{N}{2}$ from external memory to get the first $\frac{N}{2}$ elements of b from external memory. Then, the $\frac{N}{2}$ repetitions gives us $\frac{N}{2}$ reads and N arithmetic operations. Thus, we have a total of N IO operations and N arithmetic operations.

We repeat this exact process for the last half of b, with the exception that we must write the final *sum* back into external memory at the end, adding an extra access to the slow external memory. This gives us an arithmetic complexity of:

$$\alpha = \frac{2N}{2N+1} = \frac{2N}{2N+1}$$

Part 1.3 (5 points)

Calculate arithmetic intensity for the inner product when local memory can hold $2N + 1$ elements.

(Solution) We start by reading all N elements of a and all N elements of b from external memory. The last slot in local memory is reserved for *sum*. Then, we multiply the first elements of a and b together and add their product to *sum*. We repeat this process N times for all N elements of a and b. Then, we write *sum* back into external memory. In total, we read $2N$ elements from external memory and write 1 element into external memory and perform $2N$ operations. Thus, we have:

$$\alpha = \frac{2N}{2N+1} = \frac{2N}{2N+1}$$

2 Arithmetic Intensity for an Individual Outer Product

An outer product between two N-dimension vectors a and b, where a is a column vector and b is a row vector, will result in an $N \times N$ matrix C. $C = a \times b$ is computed by the following algorithm:

Algorithm 3: Outer Product on a Pair of Vectors

for $m = 1 \rightarrow N$ **do**
  for $n = 1 \rightarrow N$ **do**
    $C[m][n] \longleftarrow a[m] \cdot b[n]$;

Note: in this example, C does not need to be fetched from external memory—we are only writing out values, **not** updating pre-existing ones.

Part 2.1 (5 points)

Calculate the arithmetic intensity for the outer product computation (Algorithm 3) when local memory can hold 3 elements for the scheme where we hold a single element of A, B, and C at a time.

(Solution) For the sake of simplicity in explanation, we will deal with the inner loop of the algorithm first. We assume that we have already read an element of A into local memory in the outer loop. We start by reading the first element of B into local memory, with the third memory slot is reserved for the corresponding element of C (which does not need to be read from external memory). Then, we perform one multiplication between the element of A and the element of B and assign the product to the element of C. Finally, we write the new value of this element of C back to external memory. Since inner loop of Algorithm 3 runs N times for all elements in B, we repeat this N times. Thus, for every full run of the inner loop, we read N elements from external memory, write N elements to external memory, and perform N operations.

Then, we must consider the outer loop of the algorithm. The outer loop iterates N times and each time we read an element of A into local memory. In addition, each iteration of the outer loop contains a complete run of the inner loop. Therefore, in total, we read $N^2 + N$ elements, write N^2 elements, and perform N^2 multiplications. In other words, we have $N^2 + N$ accesses to slow memory and N^2 operations. Thus, we have:

$$\alpha = \frac{N^2}{2N^2 + N} = \frac{N}{2N + 1}$$

Part 2.2 (5 points)

Calculate the arithmetic intensity for the outer product when local memory can hold $\frac{N}{2} + 2$ elements, using the scheme where we hold half of B, and a single element of A and C in memory.

(Solution) This solution operates under the assumption that we are following the exact pseudocode for Outer Product MM. That is, the outer loop of the function is the loop for a and the inner loop is the loop for b.For the sake of simplicity in explanation, we will deal with the inner loop of the algorithm first. We assume that we have already read an element of A into local memory in the outer loop. We start by reading the first $\frac{N}{2}$ elements of B into local memory. The last slot of local memory is reserved for the element of C. Then, for each of the $\frac{N}{2}$ elements of B in local memory, we multiply it by the element of A in local memory and write the product element of C to external memory. We repeat this for the last $\frac{N}{2}$ elements of B. In total, we read N elements from external memory, write back N elements to external memory, and perform N operations.

Now, we must consider the outer loop of the algorithm, The outer loop iterates N times and each time we read an element of A into local memory. In addition, each iteration of the outer loop contains a complete run of the inner loop. Therefore, in total, we read $N^2 + N$ elements of B, write N^2 elements to external memory, and perform N operations. That is, we have $2N^2 + N$ accesses to slow memory and N^2 operations. Thus, we have:

$$\alpha = \frac{N^2}{2N^2 + N} = \frac{N}{2N + 1}$$

Note that we can change the arithmetic complexity if we assume that we can have b in the outer loop and a in the inner loop. Thus, we first read $\frac{N}{2}$ elements of b into local memory, and then iterate through the elements of

a by reading them in one at a time. Once again, the last slot in local memory is reserved for the initialization of the element of C. Then, as we read in element $a[n]$, we multiply it with all of the values in the first half of b and write their values to external memory in the corresponding elements of C. Per element of a, we perform 1 read, $\frac{N}{2}$ write, and $\frac{N}{2}$ multiplications. Since a contains N elements, we have a total of $\frac{3N}{2}$ reads, $\frac{N^2}{2}$ writes, and $\frac{N^2}{2}$ operations. We repeat this process for the last half of b, giving us $3N$ reads, N^2 writes, and N^2 operations. Thus, we have $N^2 + 3N$ accesses to slow memory and N^2 operations, giving us an arithmetic complexity of:

$$\alpha = \frac{N^2}{N^2 + 3N} = \frac{N}{N + 3}$$

Part 2.3 (5 points)

Calculate the arithmetic intensity for the outer product when local memory can hold $2N + 1$ elements, using the scheme where we hold A and B in local memory and write the computed values of C to external memory.

(Solution) We first read all N elements of A and all elements N elements of B into local memory. The last slot in local memory is reserved for an element of C (such as C[1][1]). Then, we multiply the first element of A with the first element of B and store it in C[1][1], writing C[1][1] to external memory after the operation. We repeat this process until we have multiplied every element in A with every element in B and written the corresponding element in C to external memory. Thus, we have N^2 repetitions, and in each repetition we write 1 element to external memory and perform 1 multiplication. In total, we perform N^2 operations and $2N + N^2$ accesses to slow memory. Thus, we have:

$$\alpha = \frac{N^2}{2N + N^2} = \frac{N}{N + 2}$$

3 MMM Runtimes

In this section you will implement MMM with both inner and outer products, execute the code on your computer or VM and report their runtimes.

Implementing Inner Product MMM

Part 3.1 (10 points)

Implement inner product MMM in the function `inner_product_mmm()` contained in `pa1.cpp`, as described by the following algorithm:

Algorithm 4: Inner product MMM

for $m = 1 \rightarrow M$ **do**
  for $n = 1 \rightarrow N$ **do**
    for $k = 1 \rightarrow K$ **do**
      $C[m][n] \longleftarrow C[m][n] + A[m][k] \cdot B[k][n]$;

Implementing Outer Product MMM

Part 3.2 (10 points)

Implement outer product MMM in the function `outer_product_mmm()` contained in `pa1.cpp`, as described by the following algorithm:

Algorithm 5: Outer product MMM

for $k = 1 \rightarrow K$ **do**
  for $m = 1 \rightarrow M$ **do**
    for $n = 1 \rightarrow N$ **do**
      $C[m][n] \longleftarrow C[m][n] + A[m][k] \cdot B[k][n]$;

Timing

Part 3.3 (15 points)

With your implementation of the MMM functions, run the provided code. Plot the run time results, taking the average over 5 iterations. As mentioned earlier, use only square matrices (i.e., $M = N = K$) to simplify timing comparisons. The Y-axis should be the run time (in nanoseconds) and X-axis should be the matrix dimension N. Be sure to appropriately label your generated plot with axes, title, and legend. Your plot must include data for following values of N: 16, 32, 64, 128, 256, 512, 1024 (every power of 2 between 2^4 and 2^{10}, inclusive). For plotting, use Python's `matplotlib`. What trends do you observe? Does anything stand out or seem unusual?

(Solution) **(Plots are below)** After implementing both the Outer Product and Inner Product MMM, I have a few observations. As the matrix size N increases, the two methods of MMM diverge exponentially, with the Inner Product taking far longer to compute on average. In addition, for all values of N, the Outer Product MM is faster than the Inner Product MM, with the average times of both methods growing at an exponential rate. However, for Outer Product MM, the exponential curve is much flatter compared to the Inner Product MM, signaling a much slower growth as N grows large. Thus, the Outer Product method is much more efficient at handling large matrix multiplications.

Something intriguing is the stark difference in runtime between the Outer Product MM and Inner Product MM, despite their implementations being so similar. In my C++ code, the *only* difference in the two methods is a switch in the order of the nested for loops. Such a small change leading to drastically different runtimes, especially for large N, is something that stands out.

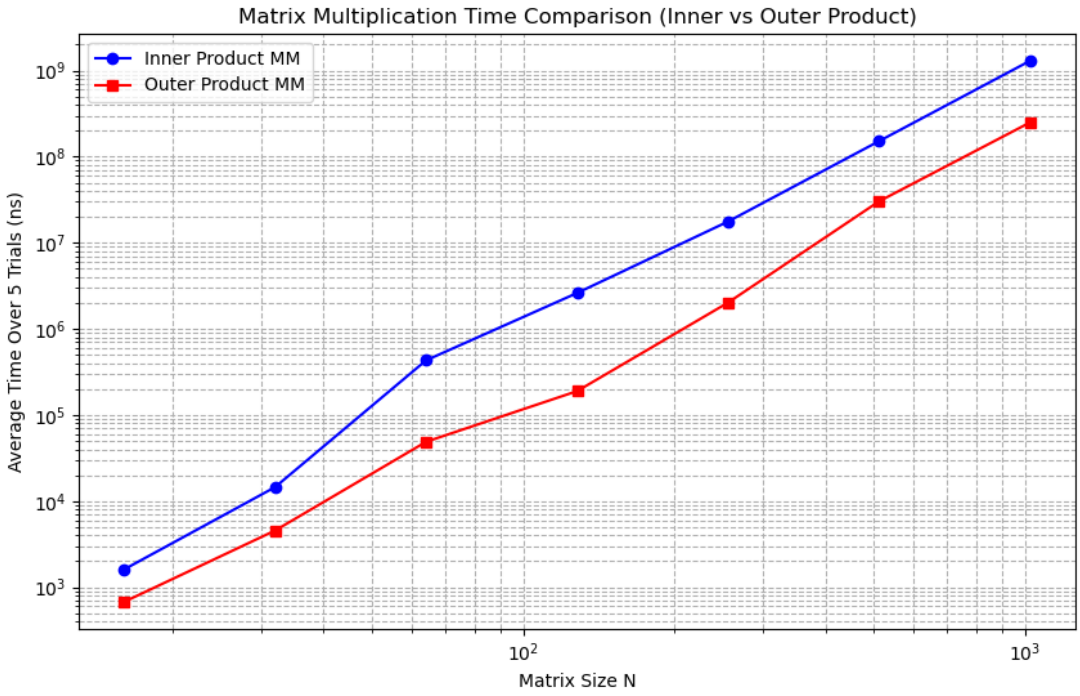


Figure 1: Log Scale

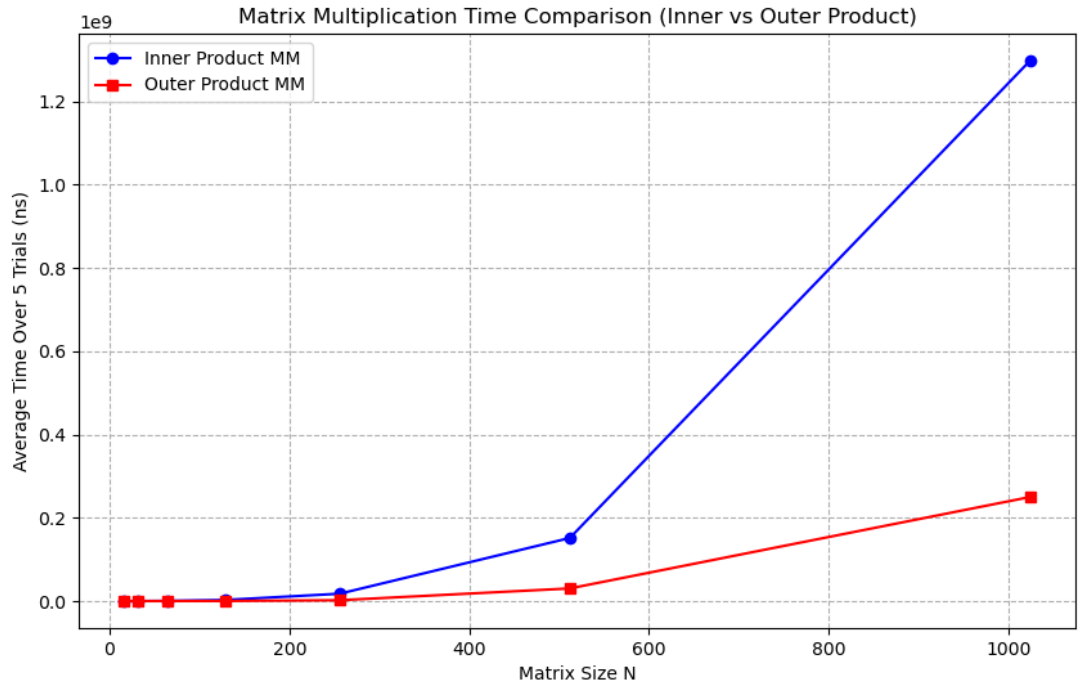


Figure 2: Linear Scale

One possible explanation for the difference in runtimes is how efficiently each method uses the CPU cache. The Inner Product method accesses matrix B in a column-wise fashion, which leads to poor spatial locality and more frequent cache misses. This inefficient memory access forces the program to fetch new data from external slow memory often, increasing latency. On the other hand, the Outer Product method accesses matrices A and B in a way that benefits from better data reuse and cache locality, leading to fewer cache misses and faster execution times.

This could also explain the vertical shift between the two methods in the runtime graph. This can be attributed to the number of cache misses each method experiences. The Inner Product method incurs significantly more cache misses, particularly as matrix size increases, due to inefficient memory access patterns, which inflates its runtime.

4 CNN Forward Pass and Backpropagation

In this section, you will perform a complete forward pass of a convolutional neural network (CNN) and execute backpropagation for the fully connected layer at the end.

Background

The CNN architecture is as follows:

- **Input Image**: A 4×4 grayscale image with a single channel.
- **Convolution Layer**: 2 filters (3×3) with stride 1 and no padding. No bias is considered in the convolution layer.
- **Activation Function**: ReLU.
- **Pooling Layer**: 2×2 max-pooling with stride 2.
- **Fully Connected Layer**: The flattened output of the pooling layer is connected to a fully connected layer with 2 output units.

Below is the input image and the filters provided for the CNN:

- **Input Image (I)**, **Filter 1 (F1)**, and **Filter 2 (F2)**:

$$I = \begin{bmatrix} 2 & 0 & 1 & 3 \\ 1 & 2 & 0 & 1 \\ 3 & 2 & 1 & 0 \\ 0 & 1 & 3 & 2 \end{bmatrix} \qquad F1 = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \qquad F2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

- **Fully Connected Weights (W_fc)** and **Fully Connected Bias (b_fc)**:

$$W_{fc} = \begin{bmatrix} 0.2 & -0.5 \\ -0.3 & 0.4 \end{bmatrix} \qquad b_{fc} = \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix}$$

- **Target Labels (Y)**:

$$Y = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Part 4.1 (5 points)

1. **Convolution**: Perform the convolution operation between the input image I and both filters $F1$ and $F2$, using stride 1 and no padding. Calculate the output feature maps for each filter.

- Show step-by-step convolution calculation for both feature maps.

2. **Activation (ReLU)**: Apply the ReLU activation function to the convolution results. Replace any negative values in the feature maps with zero.

- Write the resulting activated feature maps after applying ReLU.

(Solution) We start by placing $F1$ over the top left corner of I and performing the element-wise product of the filter and the input:

$$I[0:2][0:2] * F1 = \begin{bmatrix} 2 & 0 & 1 \\ 1 & 2 & 0 \\ 3 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} = \sum_{m=0}^{2} \sum_{n=0}^{2} I(0+m, 0+n) \cdot F(m,n) = 2+0-1+0+2+0-3+0+1 = 1$$

Sliding filter 1 to the right with stride 1, we again perform the element-wise product of the filter and the input at this new position:

$$I[0:2][1:3] * F1 = \begin{bmatrix} 0 & 1 & 3 \\ 2 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} = \sum_{m=0}^{2} \sum_{n=0}^{2} I(0+m, 1+n) \cdot F(m,n) = 0+0-3+0+0+0-2+0+0 = -5$$

Since we have hit the end of the input image matrix and we have no padding, we slide filter 1 to the bottom left corner:

$$I[1:3][0:2]{*}F1 = \begin{bmatrix}1 & 2 & 0\\ 3 & 2 & 1\\ 0 & 1 & 3\end{bmatrix} * \begin{bmatrix}1 & 0 & -1\\ 0 & 1 & 0\\ -1 & 0 & 1\end{bmatrix} = \sum_{m=0}^{2}\sum_{n=0}^{2} I(1{+}m, 0{+}n)\cdot F(m,n) = 1{+}0{+}0{+}0{+}2{+}0{+}0{+}0{+}3 = 6$$

Finally, we slide filter 1 to the right with stride 1:

$$I[1:3][1:3]{*}F1 = \begin{bmatrix}2 & 0 & 1\\ 2 & 1 & 0\\ 1 & 3 & 2\end{bmatrix} * \begin{bmatrix}1 & 0 & -1\\ 0 & 1 & 0\\ -1 & 0 & 1\end{bmatrix} = \sum_{m=0}^{2}\sum_{n=0}^{2} I(1{+}m, 1{+}n)\cdot F(m,n) = 2{+}0{-}1{+}0{+}1{+}0{-}1{+}0{+}2 = 3$$

Thus, we have the feature map for $F1$ as:

$$\textbf{Feature Map F1} = \begin{bmatrix}1 & -5\\ 6 & 3\end{bmatrix}$$

Now, we repeat this entire process with $F2$. We start by placing $F2$ over the top left corner of I:

$$I[0:2][0:2]{*}F2 = \begin{bmatrix}2 & 0 & 1\\ 1 & 2 & 0\\ 3 & 2 & 1\end{bmatrix} * \begin{bmatrix}0 & 1 & 0\\ 1 & -1 & 1\\ 0 & 1 & 0\end{bmatrix} = \sum_{m=0}^{2}\sum_{n=0}^{2} I(0{+}m, 0{+}n)\cdot F(m,n) = 0{+}0{+}0{+}1{-}2{+}0{+}0{+}2{+}0 = 1$$

Sliding filter 2 to the right with stride 1, we again perform the element-wise product of the filter and the input at this new position:

$$I[0:2][1:3]{*}F2 = \begin{bmatrix}0 & 1 & 3\\ 2 & 0 & 1\\ 2 & 1 & 0\end{bmatrix} * \begin{bmatrix}0 & 1 & 0\\ 1 & -1 & 1\\ 0 & 1 & 0\end{bmatrix} = \sum_{m=0}^{2}\sum_{n=0}^{2} I(0{+}m, 1{+}n)\cdot F(m,n) = 0{+}1{+}0{+}2{+}0{+}1{+}0{+}1{+}0 = 5$$

We then slide filter 2 to the bottom left corner:

$$I[1:3][0:2]{*}F2 = \begin{bmatrix}1 & 2 & 0\\ 3 & 2 & 1\\ 0 & 1 & 3\end{bmatrix} * \begin{bmatrix}0 & 1 & 0\\ 1 & -1 & 1\\ 0 & 1 & 0\end{bmatrix} = \sum_{m=0}^{2}\sum_{n=0}^{2} I(1{+}m, 0{+}n)\cdot F(m,n) = 0{+}2{+}0{+}3{-}2{+}1{+}0{+}1{+}0 = 5$$

Finally, we slide filter 2 to the right with stride 1:

$$I[1:3][1:3]{*}F2 = \begin{bmatrix}2 & 0 & 1\\ 2 & 1 & 0\\ 1 & 3 & 2\end{bmatrix} * \begin{bmatrix}0 & 1 & 0\\ 1 & -1 & 1\\ 0 & 1 & 0\end{bmatrix} = \sum_{m=0}^{2}\sum_{n=0}^{2} I(1{+}m, 1{+}n)\cdot F(m,n) = 0{+}0{+}0{+}2{-}1{+}0{+}0{+}3{+}0 = 4$$

Thus, we have the feature map for $F2$ as:

$$\textbf{Feature Map F2} = \begin{bmatrix}1 & 5\\ 5 & 4\end{bmatrix}$$

Then, we must apply the ReLU activation function to each feature map. Since the ReLU function sets all negative values to 0, we have:

$$\textbf{Activated Feature Map F1} = \begin{bmatrix}1 & 0\\ 6 & 3\end{bmatrix} \quad \textbf{Activated Feature Map F2} = \begin{bmatrix}1 & 5\\ 5 & 4\end{bmatrix}$$

Part 4.2 (5 points)

1. **Max-Pooling**: Apply a 2×2 max-pooling operation with stride 2 to the activated feature maps from both filters. This step reduces the size of the feature maps. No padding is applied for the max-pooling layer.

- Compute the max-pooled feature maps for both activated feature maps
- Write down the resulting pooled feature maps.

(Solution) Recall from 4.1 that we have:

$$\textbf{Activated Feature Map F1} = \begin{bmatrix} 1 & 0 \\ 6 & 3 \end{bmatrix} \quad \textbf{Activated Feature Map F2} = \begin{bmatrix} 1 & 5 \\ 5 & 4 \end{bmatrix}$$

Since our feature maps are of dimensions 2×2, we apply the max-pooling operation to the entire feature map. Thus, we have:

$$\textbf{Max-Pooling(F1)} = \max(1, 0, 6, 3) = [6] \quad \textbf{Max-Pooling(F2)} = \max(1, 5, 5, 4) = [5]$$

Part 4.3 (10 points)

1. **Flattening**: Flatten the output of the pooling layer into a single vector to be fed into the fully connected layer.

- Write the flattened vector.

2. **Fully Connected Layer**: Using the weights W_{fc} and bias b_{fc}, compute the output of the fully connected layer by applying the following equation:

$$Z_{fc} = W_{fc} \cdot X_{flat} + b_{fc}$$

where X_{flat} is the flattened vector from the pooling layer.

- Compute the output of the fully connected layer (before applying softmax).

(Solution) The output of the max-pooling layer was

$$\textbf{Max-Pooling(F1)} = \max(1, 0, 6, 3) = 6 \quad \textbf{Max-Pooling(F2)} = \max(1, 5, 5, 4) = 5$$

Thus, flattening this output into a single vector, we have:

$$X_{flat} = \begin{bmatrix} 6 \\ 5 \end{bmatrix}$$

Then, we must use the weights W_{fc} and bias b_{fc} to compute the output of the fully connected layer.

$$Z_{fc} = W_{fc} \cdot X_{flat} + b_{fc} = \begin{bmatrix} 0.2 & -0.5 \\ -0.3 & 0.4 \end{bmatrix} * \begin{bmatrix} 6 \\ 5 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix} = \begin{bmatrix} -1.2 \\ 0 \end{bmatrix}$$

Part 4.4 (15 points)

1. **Softmax Activation**: Apply the softmax activation function to the output of the fully connected layer to get the predicted probabilities P. Write your answer to the nearest three decimal places. Rounding or truncating is fine. Please keep this in mind for all future parts of this problem too.

- Compute the softmax function for the output vector Z_{fc}.

2. **Loss Calculation (Cross-Entropy)**: Using the provided target labels Y, compute the cross-entropy loss L between the predicted probabilities P and the true labels.

- Write down the cross-entropy loss.

3. **Backpropagation**: Perform one round of backpropagation on the fully connected layer using the cross-entropy loss. Compute the gradients of the loss with respect to the weights W_{fc}, the bias b_{fc}, and the input vector X_{flat}.

- (a) Compute the gradient of the loss with respect to the output of the fully connected layer (dZ_{fc}).

- (b) Compute the gradient with respect to the weights W_{fc}, the bias b_{fc}, and the input X_{flat}.

Solution Guidelines

For the convolution, use the discrete convolution formula:

$$(I * F)(i,j) = \sum_{m=0}^{2} \sum_{n=0}^{2} I(i+m, j+n) \cdot F(m,n)$$

Apply ReLU by replacing negative values with 0.
For max-pooling, take the maximum value in each 2×2 window of the feature maps.

The softmax function is given by:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

The cross-entropy loss is:

$$L = -\sum_i y_i \log(p_i)$$

where y_i is the true label and p_i is the predicted probability from softmax.

(Solution) We have $Z_{fc} = \begin{bmatrix} -1.2 \\ 0 \end{bmatrix}$ and the formula for the softmax function as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

We can calculate $\sum_j e^{z_j} = e^{-1.2} + e^0 = 0.301 + 1 = 1.301$. Therefore, we have:

$$\text{softmax}(z_1) = \frac{e^{-1.2}}{1.301} = \frac{0.301}{1.301} = 0.232$$

$$\text{softmax}(z_2) = \frac{e^{0}}{1.301} = \frac{1}{1.301} = 0.768$$

$$\text{softmax}(Z_{fc}) = \begin{bmatrix} 0.232 \\ 0.768 \end{bmatrix}$$

Next, we calculate the cross entropy loss using this softmax output. Let L be the cross entropy loss:

$$L = -\sum_i y_i \log(p_i) = -(1 * \log(0.232) + 0 * \log(0.768)) = 1.461$$

This is under the assumption that log is the natural logarithm.

Finally, we must perform one round of backpropagation on the FC layer using the cross entropy loss. We start by calculating the gradient of the lsos with respect to the output of the FC layer ($\frac{\partial L}{\partial Z_{fc}}$). Let P be defined as the output of the softmax function. Since we are given the softmax and cross entropy loss functions, we employ the chain rule to get:

$$\frac{\partial L}{\partial Z_{fc_i}} = \frac{\partial L}{\partial P_i} \frac{\partial P_i}{\partial Z_{fc_i}} = P_i - Y_i$$

$$\frac{\partial L}{\partial Z_{fc_1}} = 0.232 - 1 = -0.768$$

$$\frac{\partial L}{\partial Z_{fc_2}} = 0.232 - 1 = -0.768$$

$$\frac{\partial L}{\partial Z_{fc}} = \begin{bmatrix} -0.768 \\ 0.768 \end{bmatrix}$$

Next, we will calculate the gradient of the loss with respect to the weights W_{fc}, the bias b_{fc}, and the input X_{flat}. We are given the equation:

$$Z_{fc} = W_{fc} \cdot X_{flat} + b_{fc}$$

Therefore, $\frac{\partial Z_{fc}}{\partial W_{fc}} = X_{flat}$ and we have the gradient with respect to weights as:

$$\frac{\partial L}{\partial W_{fc}} = \frac{\partial L}{\partial Z_{fc}} \frac{\partial Z_{fc}}{\partial W_{fc}} = \begin{bmatrix} -0.768 \\ 0.768 \end{bmatrix} \cdot X_{flat}^T = \begin{bmatrix} -0.768 \\ 0.768 \end{bmatrix} \cdot \begin{bmatrix} 6 & 5 \end{bmatrix} = \begin{bmatrix} -4.608 & -3.840 \\ 4.608 & 3.840 \end{bmatrix}$$

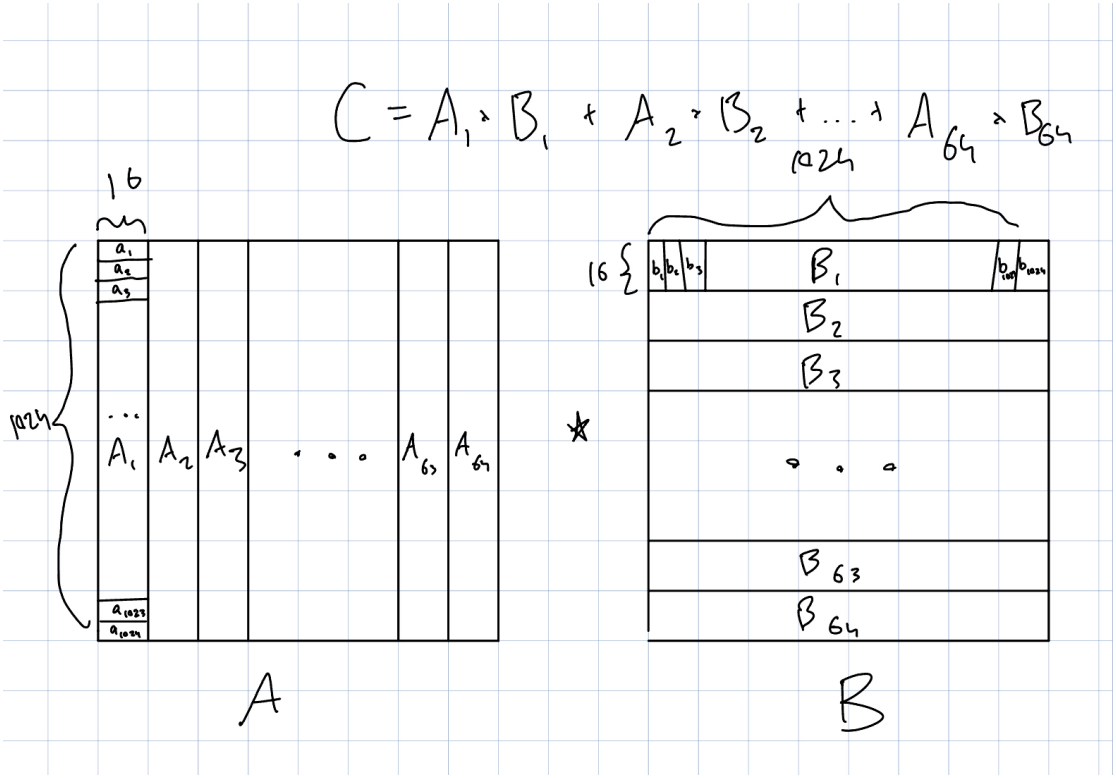
Note that we must transpose X_{flat} during the calculation of the partial derivative of matrix multiplication. We continue by calculating the gradient of the loss with respect to the bias. Since we have $\frac{\partial Z_{fc}}{\partial b} = 1$, then:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial Z_{fc}} \cdot 1 = \begin{bmatrix} -0.768 \\ 0.768 \end{bmatrix}$$

Finally, we will calculate the gradient of the loss with respect to the input. We know that $\frac{\partial Z_{fc}}{\partial X_{flat}} = W_{fc}$. Thus,

$$\frac{\partial L}{\partial X_{fc}} = \frac{\partial L}{\partial Z_{fc}} \frac{Z_{fc}}{X_{flat}} = W_{fc}^T \cdot \begin{bmatrix} -0.768 \\ 0.768 \end{bmatrix} = \begin{bmatrix} 0.2 & -0.3 \\ -0.5 & 0.4 \end{bmatrix} \begin{bmatrix} -0.768 \\ 0.768 \end{bmatrix} = \begin{bmatrix} -0.384 \\ 0.691 \end{bmatrix}$$

Where again we must transpose the weights during the calculation of the partial derivative.



5 SIMD

SIMD (Single Instruction, Multiple Data) is a technique used to improve the performance of computationally intensive tasks by performing the same operation on multiple data points simultaneously. In this question, you will explore how SIMD can be applied to matrix multiplication, a core operation you've worked with earlier in this assignment.

Part 5.1 (10 points)

Consider a system with 512-bit SIMD registers that can hold 16 single-precision floating-point numbers. You are implementing matrix multiplication C = A * B, where A, B, and C are 1024 x 1024 matrices.

1. Describe a SIMD-based algorithm for this matrix multiplication that maximizes data reuse. Include details on data layout and access patterns. (5 points)
2. Calculate the theoretical speed-up of your SIMD implementation compared to a scalar version. Then, explain why the actual speed-up is likely to be lower, considering memory bandwidth limitations and cache behavior.

(Solution)

2: We will assume that the matrix multiplication we are implementing is the inner product of two matrices. We also assume that we have 3 512-bit registers available to use.

I propose the following SIMD based algorithm that utilizes our systems 512-bit registers. We first partition A into tiles of 1024×16 and partition B into tiles of 16×1024 and store them in the cache (under the assumption that we can fit these tiles on-chip). The layout of the tiles are shown above in the figure. We wish to compute $A_1 \times B_1, A_2 \times B_2, ..., A_64 \times B_64$ so that we can compute C. To do this, we again partition A_i and B_i into subvectors $a_1, ..., a_{1024}$ and $b_1, ..., b_{1024}$. Here, we can utilize SIMD to accelerate the computations of our algorithm. We load in a_i into a 512-bit register and b_i into another 512-bit register (assuming we can load/store 16 elements per clock).

Then, we can apply the same multiplication operation to the elements in the registers and add this partial sum (the result) back to the appropriate position in matrix C. By loading 16 consecutive elements of matrix A into the SIMD registers and reusing them for multiple iterations through matrix B, we significantly reduce the number of times elements from matrix A need to be loaded. After all the results from the SIMD multiplication are accumulated, the updated partial sum value for $C[m][n]$ is written back to memory. This approach ensures both efficient data reuse (by reusing a_i in the register as we iterate through the b_i's).

Algorithm 6: SIMD-based Inner Product MMM for Tiles $A_i \times B_i$

```
for m = 1 → 1024 do
    for n = 1 → 1024 do
        C[m][n] ⟵ 0;
        for k = 1 → 16 step 16 do
            Load a_{i=k/16} ⟵ A[m][k : k + 15] into SIMD register;
            Load b_{i=k/16} ⟵ B[k : k + 15][n] into SIMD register;
            Accumulate the sums of the SIMD products in the partial sum C[m][n];
```

Since we use 512-bit registers which can hold and operate on 16 floating point numbers concurrently, we can perform 16 multiplications in the time that a scalar algorithm can perform 1. Thus, if we consider the calculations required for computing a single value of C, by definition of the inner product there must be 1024 multiplications occur between the corresponding row and column vectors of size 1024. However, since we can operate and perform 16 of these multiplications at once and store it in a partial sum, we can calculate this value of C in 64 operations. Therefore, we have a theoretical speedup of $16\times$.

2: However, we will likely observe a slower speed-up empirically. This is because even though we can speed up the CPU performance by performing operations in parallel, we must contend with memory bandwidth limitations. That is, our algorithm works faster by moving more data for the CPU to operate on at a time. However, if we cannot move this increased amount of data back and forth from the registers as fast as the CPU performs the SIMD operations (for example, if our assumption that we can move 16 elements in one clock cycle is wrong), then the CPU will end up waiting for the data to arrive. Thus, the memory bandwidth could serve as a bottleneck for performance as our CPU would still have to sit idle as it waits for the data to be held in the registers.

Another possible limitation is the cache. Although we made the assumption that we could fit our tiles in the cache, this could be false, in which case we would have to constantly read from slow external memory to populate our registers which would drastically increase our runtime. Moreover, since we can't fit the large matrix C (1024×1024), we constantly have to write back into slow external memory which would increase our runtime and cause our speed-up to be slower than theoretically expected.

Part 5.2 (10 points)

You are optimizing the convolution operation in the CNN forward pass (from Part 4) using SIMD instructions.

1. Propose two different SIMD-based approaches for the convolution operation. Compare their efficiency in terms of SIMD register utilization and potential for instruction-level parallelism. (5 points)
2. One of your colleagues suggests using a SIMD width of 2048 bits to further improve performance. Explain the potential drawbacks of this approach, considering aspects such as power consumption, chip area, and applicability to other parts of the CNN computation. (5 points)

(Solution)

1: One SIMD-based approach for the convolution operation would be to load in the values of a filter matrix into a register in row-major ordering (ie. for a 4x4 filter we would store row 1, then row 2, etc). Then, we would stream in the image patches and store them in row-major ordering in the register. After streaming in a patch, we would use SIMD multiplication to perform all the element wise multiplication in one operation, and

accumulating the sum with an adder tree for the feature map. Finally, we would stream in the next image patch taking stride into account. We repeat this process for all the filters, loading new values in the filter register only when we are done with the previous filter.

This method allows us to reuse data as we do not need to write new values in the filter register (since we pass the same filter over all the image patches). Thus, if the filter size matches the size of the register, we efficiently utilize the register space and SIMD operations. However, if we had many empty values after the filter in the register, or if the filter itself had too many elements to fit into the register, the efficiency of this method would decrease drastically. In addition, if our stride is less than the size of the image patches, we would redundantly have to write and load in elements as the image patches would have overlapping sections. Since we take advantage of SIMD to accelerate the repeated operations inherent to the convolution forward pass, this method has high potential for instruction-level parallelism.

Another SIMD-based approach for the convolution operation would be to flatten each image patch into a vector by row-majoring order. We would perform this operation on every image patch and stacking the vector-patches on top of each other to form a matrix. Thus, each image patch is a row in this new matrix. Similarly, we could flatten each filter into a vector by row-major order and line the vectors up as columns in another matrix. Then, we could simply perform the inner product of these two matrices using the SIMD-based MM algorithm we explained in part 1 to calculate the feature maps in one go. To extract the feature maps, we would just need to get each column of the resulting matrix (since each element in a column represents the element-wise product of a filter with a patch by the properties of inner product matrix multiplication).

This method relies on the SIMD inner product MMM algorithm. As such, we can leverage the ILP efficiency of the SIMD MMM algorithm. Moreover, this method still suffers from the redundant register utilization of method 1: when the stride is less than the patch size, we end up writing out and loading in many of the same elements. This is the same, as our transformed patch matrix would still have overlap in patches and consequently our register utilization would not be as advantageous. In addition, when the filters are again smaller than the register size, then our MMM algorithm will still have leftover space in the register when performing the matrix multiplications (see 5.1 for why) leading to inefficiencies. However, when the filters are larger than the register size, this method performs better since the MMM multiplication simply chunks the filter into sizes of the register and processes them the same way a normal matrix multiplication would work.

2: One potential drawback of this approach would be the increased power consumption by the processing units in the CPU. With 2048 bit registers, the processor would need to handle 64 concurrent floating point operations at a time, which would be computationally much more expensive due to the increased need in processing units. Moreover, utilizing 2048 bit registers would entail moving much more data to and from these registers, leading to increased memory bandwidth usage and more power consumption. Another potential drawback is the increased chip size. Larger registers would mean that the chip would need much more processors to handle the SIMD operations. This would mean either the chip size would increase, increasing the complexity of the chip and heat, or taking away from other important parts of the chip like L1, L2, L3 cache memory. A final drawback would be the applicability of this increase in register size. Although SIMD operations accelerate CNN computation, there are other parts of the CNN computation that would not necessitate an increased register size such as non-vectorized operations. Operations like softmax, ReLU, pooling, and normalization would not benefit from this. Finally, just because the register size would increase does not mean the overall speed of computation would increase. For example, if the memory bandwidth decreased or the cache size decreased due to the increased processors taking up space, then we would face a memory bottleneck and the CPU would end up waiting for the data.

Part 5.3 (5 points)

Your SIMD-optimized matrix multiplication code from Part 5.1 runs slower on a new CPU architecture with the same SIMD width but double the clock speed. Identify three possible reasons for this performance degradation and briefly explain how you would diagnose each issue.

(Solution)
Memory Bandwidth Bottleneck
This new architecture could have much lower memory bandwidth than the previous CPU architecture. Thus,

even if the clock speed was twice as fast as the old CPU, the new CPU wouldn't be able to move the data into the cache and internal memory as fast. This would result in more time spent on moving the data into the actual CPU itself, which could very likely outweigh the benefit of the faster clock speed and cause slower performance. I would diagnose this issue by measuring the memory bandwidth utilization on both the old and new CPU architectures by using performance monitoring tools.

Latency in SIMD Operations

Just because the new CPU architecture has the same SIMD width and double the clock speed as our old architecture, does not mean that all operations have scaled equally with its clock speed. That is, this new CPU architecture may have longer latencies for SIMD operations which cause our algorithm to take longer in spite of the clock speed. I would diagnose this issue by using a performance counter to compare the number of clock cycles it takes to execute a SIMD operation on the new architecture with the number of clock cycles it takes to execute a normal operation. I would also measure the latency of SIMD operations on this new CPU.

Cache/Internal Memory Differences

This new CPU architecture could have much worse internal memory capabilities such as a reduced cache size. In this case, it would be likely that the tiles or matrices which we previously stored in our cache would be forced to be kept in slow external memory. Thus, the increase in cache misses would outweigh the benefit of the faster CPU clock speed with much much slower data movement. I would diagnose this by monitoring the number of cache misses in one run of the program on the new CPU and compare it to the number of cache misses in one run of the program on the old CPU. I would also inspect the specifications of the new CPU architecture to see if the cache size is smaller.

6 What to Submit

Your submission should be a `.zip` archive with a `CS242_PA1_` prefix followed by your full name. The archive should contain:

- PDF write-up
- Assignment code
- Text files or PDFs containing the complete outputs (e.g., ChatGPT logs) of all generative AI tools used.

Example filename: `CS242_PA1_FirstName_LastName.zip`

Write-up

Written responses should be contained within a single PDF document. (LaTeX is highly recommended!) Each response or figure should clearly indicate which problem is being answered.

Code

You should include **all** files that were provided, but with the changes you made. Additionally, you must include your graphing code and timing data for Part 4.3.