Harvard CS 2420: Computing at Scale (Fall 2025) **(Due: 09/22/2025 at 11:59PM EDT)**

Problem Set 1

Instructor: Professor H.T. Kung ***Name:*** Shreshth Rajan shreshthrajan@college.harvard.edu

Please include your full name and email address in your submission. (See Section 6)

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 arithmetic 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 your analysis on 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 for an even integer N:

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 perform N MACs ($2N$ ops). IO: read N elements of a, read N elements of b, and write the final sum once, so IO $= 2N + 1$. Therefore

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

Part 1.2 (10 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) Capacity $\frac{N}{2} + 2$ lets us load half of b at a time and stream the matching half of a, then repeat for the second half. Each element of a and b is read exactly once and the final sum is written once, so IO $= 2N+1$. Ops $= 2N$. Hence

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

Part 1.3 (5 points)

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

(Solution) With capacity $2N + 1$, we can hold all of a and b plus sum. Reads: N of a and N of b; write sum once, so IO $= 2N + 1$. Ops $= 2N$. Thus

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

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 → N do
    for n = 1 → N do
        C[m][n] ⟵ a[m] · 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) Ops: N^2 multiplies. IO (capacity = 3, hold one $a[m]$, one $b[n]$, one $C[m,n]$): read each $a[m]$ once (N), read each $b[n]$ once per row (N^2), write each $C[m,n]$ once (N^2). Thus IO $= N + 2N^2$. Hence

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

Part 2.2 (10 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)

- *Capacity $\frac{N}{2} + 2$:* You can hold exactly one half of b, one $a[m]$, and one $C[m,n]$ at a time.
- *Optimal schedule:* Load the first half of b once and sweep all m; then load the second half once and sweep all m again.
- *b reads:* $\frac{N}{2} + \frac{N}{2} = N$ total (each $b[n]$ read exactly once).
- *a reads:* Each $a[m]$ must pair with *both* halves of b. Since only one half fits at a time, you must reread $a[m]$ in the second sweep $\Rightarrow 2N$ reads.
- *C writes:* Each output is written once; no reads of C per the problem note $\Rightarrow N^2$ writes.

Therefore,

$$\text{IO} = N^2 + 3N, \qquad \alpha = \frac{N^2}{N^2 + 3N} = \frac{N}{N + 3}.$$

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) Capacity $2N+1$: keep all of a and b plus one $C[m,n]$. Reads: N of a and N of b once each; writes: N^2 for C. Hence IO $= N^2 + 2N$. Ops $= N^2$. So

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

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 `pset1.cpp`, as described by the following algorithm:

Algorithm 4: Inner product MMM

for $m = 1 \to M$ **do**
  for $n = 1 \to N$ **do**
    for $k = 1 \to K$ **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 `pset1.cpp`, as described by the following algorithm:

Algorithm 5: Outer product MMM

for $k = 1 \to K$ **do**
  for $m = 1 \to M$ **do**
    for $n = 1 \to N$ **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 runs. 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) **Observed trends.**

- **Outer-product MMM dominates**: the red curve is below the blue for all N and grows more gently.
- **Gap widens with size**: around $N \approx 512$ the blue curve steepens markedly (cache-capacity/working-set transition), while the red curve degrades more gracefully.
- **Large-N speedup**: at $N = 1024$, inner $\approx 1.30 \times 10^9$ ns vs. outer $\approx 2.30 \times 10^8$ ns $\Rightarrow$ outer is $\sim 5.65\times$ faster.
- **Small-N regime**: the curves are close and a bit lumpy. This is expected from timer granularity, loop/setup overhead, and frequency scaling hiding asymptotics.

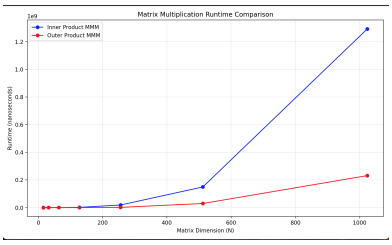

Figure 1: Problem 3.3 illustration

Interpretation. This matches expectations: the outer-product schedule achieves higher arithmetic intensity α and better cache/register reuse (keeping C accumulators hot while streaming A and packed B), so it sustains performance as N grows; the inner-product schedule becomes bandwidth/latency bound once the working set outgrows cache.

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, the filters and other setups 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) **Convolution (stride 1, no padding).** Each feature map is 2×2.

Filter $F1$:

$$F1 = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

$$(I * F1)_{1,1} = 2 \cdot 1 + 0 \cdot 0 + 1 \cdot (-1) + 1 \cdot 0 + 2 \cdot 1 + 0 \cdot 0 + 3 \cdot (-1) + 2 \cdot 0 + 1 \cdot 1 = 1,$$
$$(I * F1)_{1,2} = 0 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1) + 2 \cdot 0 + 0 \cdot 1 + 1 \cdot 0 + 2 \cdot (-1) + 1 \cdot 0 + 0 \cdot 1 = -5,$$
$$(I * F1)_{2,1} = 1 \cdot 1 + 2 \cdot 0 + 0 \cdot (-1) + 3 \cdot 0 + 2 \cdot 1 + 1 \cdot 0 + 0 \cdot (-1) + 1 \cdot 0 + 3 \cdot 1 = 6,$$
$$(I * F1)_{2,2} = 2 \cdot 1 + 1 \cdot 0 + 0 \cdot (-1) + 1 \cdot 0 + 3 \cdot 1 + 2 \cdot 0 + 1 \cdot (-1) + 3 \cdot 0 + 2 \cdot 1 = 3.$$

$$X^{(1)} = \begin{bmatrix} 1 & -5 \\ 6 & 3 \end{bmatrix}$$

Filter F2:

$$F2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$(I * F2)_{1,1} = 2 \cdot 0 + 0 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 2 \cdot (-1) + 0 \cdot 1 + 3 \cdot 0 + 2 \cdot 1 + 1 \cdot 0 = 1,$$
$$(I * F2)_{1,2} = 0 \cdot 0 + 1 \cdot 1 + 3 \cdot 0 + 2 \cdot 1 + 0 \cdot (-1) + 1 \cdot 1 + 2 \cdot 0 + 1 \cdot 1 + 0 \cdot 0 = 5,$$
$$(I * F2)_{2,1} = 1 \cdot 0 + 2 \cdot 1 + 0 \cdot 0 + 3 \cdot 1 + 2 \cdot (-1) + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 + 3 \cdot 0 = 5,$$
$$(I * F2)_{2,2} = 2 \cdot 0 + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 + 3 \cdot (-1) + 2 \cdot 1 + 1 \cdot 0 + 3 \cdot 1 + 2 \cdot 0 = 4.$$

$$X^{(2)} = \begin{bmatrix} 1 & 5 \\ 5 & 4 \end{bmatrix}$$

ReLU (elementwise $\max(0, \cdot)$):

$$A^{(1)} = \begin{bmatrix} 1 & 0 \\ 6 & 3 \end{bmatrix}, \qquad A^{(2)} = \begin{bmatrix} 1 & 5 \\ 5 & 4 \end{bmatrix}.$$

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) **Max-pooling (2×2, stride 2).** Each activated map reduces to a single value:

$$P^{(1)} = 6, \qquad P^{(2)} = 5.$$

So the pooled feature maps are 1×1 with values 6 and 5, respectively.

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) **Flattening.** Concatenate pooled values:

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

Fully connected (before softmax).

$$Z_{fc} = W_{fc} X_{\text{flat}} + b_{fc} = \begin{bmatrix} -1.2 \\ 0.0 \end{bmatrix}.$$

Part 4.4 (20 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)

Setup. With $X_{\text{flat}} = \begin{bmatrix} 6 \\ 5 \end{bmatrix}$, $W_{fc} = \begin{bmatrix} 0.2 & -0.5 \\ -0.3 & 0.4 \end{bmatrix}$, $b_{fc} = \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix}$, we have

$$Z_{fc} = W_{fc} X_{\text{flat}} + b_{fc} = \begin{bmatrix} -1.2 \\ 0.0 \end{bmatrix}, \qquad P = \text{softmax}(Z_{fc}) = \begin{bmatrix} \dfrac{e^{-1.2}}{e^{-1.2}+e^{0}} \\ \dfrac{e^{0}}{e^{-1.2}+e^{0}} \end{bmatrix} = \begin{bmatrix} 0.231475 \\ 0.768525 \end{bmatrix}.$$

Cross-entropy (one-hot $Y = [1,0]^T$):

$$L = -\sum_i y_i \log p_i = -\log(0.231475) = 1.463394 \text{ (rounded 1.463).}$$

Softmax + CE gradient (chain rule). Let $z \in \mathbb{R}^2$, $p = \text{softmax}(z)$, $L = -\sum_i y_i \log p_i$. The softmax Jacobian is

$$\frac{\partial p_i}{\partial z_k} = p_i(\delta_{ik} - p_k).$$

Then

$$\frac{\partial L}{\partial z_k} = \sum_i \frac{\partial L}{\partial p_i}\frac{\partial p_i}{\partial z_k} = \sum_i \Big(-\frac{y_i}{p_i}\Big) p_i(\delta_{ik} - p_k) = -y_k + \Big(\sum_i y_i\Big) p_k = p_k - y_k.$$

Hence

$$dZ_{fc} = \frac{\partial L}{\partial Z_{fc}} = P - Y = \begin{bmatrix} -0.768525 \\ 0.768525 \end{bmatrix}.$$

Gradients for affine layer $z = Wx + b$.

$$\frac{\partial L}{\partial W_{fc}} = dZ_{fc}\, X_{\text{flat}}^{\top}, \qquad \frac{\partial L}{\partial b_{fc}} = dZ_{fc}, \qquad \frac{\partial L}{\partial X_{\text{flat}}} = W_{fc}^{\top} dZ_{fc}.$$

Compute dW_{fc} (outer product):

$$dW_{fc} = \begin{bmatrix} -0.768525 \\ 0.768525 \end{bmatrix} \begin{bmatrix} 6 & 5 \end{bmatrix} = \begin{bmatrix} -4.611149 & -3.842624 \\ 4.611149 & 3.842624 \end{bmatrix} \xrightarrow{\text{to 3 d.p.}} \begin{bmatrix} -4.611 & -3.843 \\ 4.611 & 3.843 \end{bmatrix}.$$

Compute db_{fc} and dX_{flat}:

$$db_{fc} = dZ_{fc} = \begin{bmatrix} -0.768525 \\ 0.768525 \end{bmatrix} \xrightarrow{\text{3 d.p.}} \begin{bmatrix} -0.769 \\ 0.769 \end{bmatrix},$$

$$dX_{\text{flat}} = W_{fc}^{\top} dZ_{fc} = \begin{bmatrix} 0.2 & -0.3 \\ -0.5 & 0.4 \end{bmatrix} \begin{bmatrix} -0.768525 \\ 0.768525 \end{bmatrix} = \begin{bmatrix} -0.384262 \\ 0.691672 \end{bmatrix} \xrightarrow{\text{3 d.p.}} \begin{bmatrix} -0.384 \\ 0.692 \end{bmatrix}.$$

Rounding note (to avoid "rounding errors"). *Do not* round P before backprop. Compute dZ_{fc} and dW_{fc} in full precision, then round final reported values to three decimals as above. If you prematurely round P to $[0.231, 0.769]^T$, you would get

$$dW_{fc} \approx \begin{bmatrix} -0.769 \\ 0.769 \end{bmatrix} \begin{bmatrix} 6 & 5 \end{bmatrix} = \begin{bmatrix} -4.614 & -3.845 \\ 4.614 & 3.845 \end{bmatrix},$$

which differs from the correct result by about 3×10^{-3} per entry.

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 (25 points)

Consider a typical 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. (20 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. (5 points)

(Solution) **(1) SIMD MMM algorithm (maximize reuse).** Assume row-major $A, C \in \mathbb{R}^{1024\times1024}$ and pack B into contiguous *column panels* of 16 columns (to match a 512-bit = 16×fp32 SIMD lane). Use a blocked GEMM with a register microkernel:

- *Blocking:* Choose $M_b \times K_b \times N_b$ so an A panel ($M_b \times K_b$) and a B panel ($K_b \times N_b$) fit in L1/L2. Set $N_b = 16$ to align with the SIMD width; pick M_b so M_r rows of C accumulators fit in vector registers (e.g., M_r around 8).
- *Packing:* Pack each B panel as $(K_b \times 16)$ contiguous in K-major order so each load brings 16 consecutive B values across columns. Keep the A panel row-major for streaming/broadcast.
- *Microkernel ($M_r \times 16$):* For $k = 1, \ldots, K_b$:
  1. Load one vector $b_k = \text{load}(B[k,\ j{:}j+15])$.
  2. For $i = 0, \ldots, M_r{-}1$: broadcast $a_{ik} = \text{broadcast}(A[i_0{+}i,\ k])$ and accumulate $C_v[i] \leftarrow \text{fma}(a_{ik},\ b_k,\ C_v[i])$.

  After the K_b loop, store $C_v[\,]$ to $C[i_0{:}i_0 + M_r - 1,\ j{:}j+15]$.
- *Outer loops & prefetch:* Tile i over M_b (to reuse packed B across many rows of A) and j over N in steps of 16. Prefetch next A rows and B columns; use non-temporal stores for C if reuse is low.

This maximizes reuse: each packed B element is reused across M_b rows from L1/L2; each A scalar is broadcast to 16 lanes and reused across $N_b = 16$ columns; C accumulators remain in registers across the K_b reduction.
From class, this is the CAKE/outer-product schedule: pack B panels (column-major across the SIMD width), stream A rows with broadcasts, keep C accumulators in registers, and choose M_b, K_b at the L1/L2 ridge point to maximize $\alpha = \frac{\text{FLOPs}}{\text{bytes moved}}$. *Stores:* Use non-temporal stores for C when the tile is not reread, avoiding write-allocate traffic.

(2) Theoretical vs. realized speed-up. Ideal vector speed-up from a 16-wide SIMD is

$$S_{\text{ideal}} = 16.$$

In practice $S_{\text{real}} < 16$ due to roofline limits (insufficient arithmetic intensity if blocking is imperfect), memory bandwidth/latency (L2/L3/DRAM stalls), remainder handling (masked tails), front-end/uop throughput, and packing/transpose overheads. Hence

$$S_{\text{real}} = \min\{16,\ \text{compute roof},\ \text{bandwidth roof}\}, \quad \text{and } P = \min\{P_{\text{peak}},\ \alpha \cdot BW\}.$$

Part 5.2 (15 points)

Consider a typical system with 512-bit SIMD registers that can hold 16 single-precision floating-point numbers. You are optimizing the 3D convolution operation in the CNN forward pass (ref. lec-2 p16-22) using SIMD instructions.

1. Propose a SIMD-based approach for the 3D convolution operation that maximizes data reuse. Include details on data layout and access patterns. (Assume the input and filter sizes are much larger than a single SIMD register can hold, and that the stride size matches the filter size.) (10 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 operations in CNN computation. (5 points)

(Solution) **(1) SIMD 3D convolution (stride equals filter size).** Use channel-last layout (NHWC) so channels are contiguous and vectorizable. Let the filter be $(K_z, K_y, K_x, C_{\text{in}}, C_{\text{out}})$ and stride (K_z, K_y, K_x) (non-overlapping windows).

- *Tiling:* Tile the output into (T_z, T_y, T_x) so the needed input subvolume and a block of filters fit in L1/L2. With non-overlapping windows, input reuse is mostly across channels and batches; filter reuse is high across many output positions.
- *Layout & packing:* Keep input as NHWC; pack filter blocks so for each (k_z, k_y, k_x) the C_{in} dimension is stored in contiguous 16-wide chunks. This allows aligned vector loads; use masks when $C_{\text{in}} \bmod 16 \neq 0$.
- *Inner loop (per output voxel and per C_{out} block):*

  1. For each (k_z, k_y, k_x) load $x = \text{load16}(I[z + k_z,\ y + k_y,\ x + k_x,\ c{:}c + 15])$.
  2. Load $w = \text{load16}(W[k_z, k_y, k_x,\ c{:}c + 15,\ c_{\text{out}}])$.
  3. Accumulate $acc \leftarrow \text{fma}(x, w, acc)$.

  Reduce across C_{in} in 16-wide chunks and across (K_z, K_y, K_x); unroll kernel loops for ILP; double-buffer input/filter tiles; prefetch next tiles.

From class, we know that with stride=kernel (no spatial overlap), the efficient choice is *weight-stationary*: keep filter tiles resident while sweeping output voxels; *output-stationary* helps only if C reuse in cache dominates.

(2) Why not 2048-bit SIMD?

- *Utilization/tails:* Wider vectors increase masked-lane waste when C_{in} is not a multiple of the width and for small kernels.
- *Memory-system mismatch:* Typical cache lines are 64 B; 2048-bit (256 B) vectors span multiple lines, increasing alignment penalties, split-line accesses, TLB pressure, and bandwidth waste on gathers/scatters.
- *Power/area/frequency:* Very wide vector files/crossbars raise power and area, often forcing lower sustained clocks (down-binning) under vector load; energy per useful FLOP rises.
- *Poor fit to rest of CNN:* ReLU, pooling, indexing, and softmax have limited data parallelism or control divergence; end-to-end speedup saturates before such widths.

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) **Three reasons a 2×-clock CPU runs slower (same SIMD width) and how to diagnose:**

1. *Bandwidth/latency bound:* Peak DRAM/L3 bandwidth may be unchanged; in cycles, latency is effectively higher at a faster clock. Diagnose with a roofline check (STREAM/triad), and hardware counters for L2/L3 misses and memory-stall cycles.

2. *AVX down-binning/thermal limits:* Wide-vector code can trigger lower sustained core frequency. Measure actual GHz under load (e.g., APERF/MPERF or `turbostat`); compare scalar vs SIMD kernels; correlate IPC and frequency.

3. *Cache/prefetch/tile mismatch:* Prior M_b, K_b, N_b and prefetch distances may thrash new cache geometries or TLBs. Use VTune/Perf/LIKWID to inspect L1D/L2 miss, DTLB miss, split-line events, and retired uops; sweep tile sizes and alignment to re-find the ridge point.

Works Cited for Problem 5 (Non-lecture Sources)

References

[1] K. Goto and R. A. van de Geijn, "Anatomy of High-Performance Matrix Multiplication," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, Article 12, 2008. *Used for:* register-level microkernel design, packing B into column panels, broadcast of A, and outer-product style accumulation in §5.1.

[2] F. G. Van Zee and R. A. van de Geijn, "BLIS: A Framework for Rapidly Instantiating BLAS Functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, Article 14, 2015. *Used for:* blocking hierarchies (L1/L2/L3), microkernel interface, tile sizes (M_b, K_b, N_b), and practical packing/streaming choices referenced in §5.1.

[3] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2023. *Used for:* non-temporal (streaming) stores to avoid write-allocate on C, software prefetching guidance, cache line size (64 B), gather/scatter costs, tail-masking behavior with AVX-512, and general SIMD throughput considerations (§5.1, §5.2, §5.3).

[4] Intel Corporation, "Intel® AVX-512 Frequency Behavior: Guidelines for Mitigating Frequency Reduction," White Paper, 2017/2020. *Used for:* explanation of AVX/AVX-512 frequency down-binning under heavy vector utilization, cited in §5.3 diagnostics.

[5] J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," Technical Report/Benchmark Suite, 1991–present. *Used for:* bandwidth-bound diagnosis and roofline placement (measuring attainable BW) in §5.3.

[6] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," *Proceedings of the ACM International Conference on Multimedia*, 2014. *Used for:* im2col/GEMM mapping of convolution and practical data-layout implications referenced in §5.2.

[7] S. Chetlur *et al.*, "cuDNN: Efficient Primitives for Deep Learning," arXiv:1410.0759, 2014. *Used for:* NHWC vs. NCHW trade-offs, channel-contiguous vectorization across C_{in}, and convolution dataflows referenced in §5.2.

[8] Intel Corporation, *Intel® VTune™ Profiler User Guide*, 2024. *Used for:* cache-miss, op throughput, and memory-stall analysis methodology cited in §5.3.

[9] Linux Kernel Developers, "`perf` (Linux perf_events) User Documentation," 2024. *Used for:* CPU performance counter collection (L1/L2/L3 misses, stalled cycles) referenced in §5.3.

[10] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A Lightweight Performance Tools Suite for x86 Multicore Environments," *Proceedings of PSTI*, 2010. *Used for:* rapid counter-based diagnosis of bandwidth/latency limits and cache/TLB behavior referenced in §5.3.

[11] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Vol. 3 (System Programming Guide), 2024. *Used for:* APERF/MPERF MSRs and related frequency measurement guidance referenced in §5.3 (with `turbostat`).

6 What to Submit

Your submission should be a `.zip` archive with a `CS2420_PSet1_` 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: `CS2420_PSet1_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 3.3.