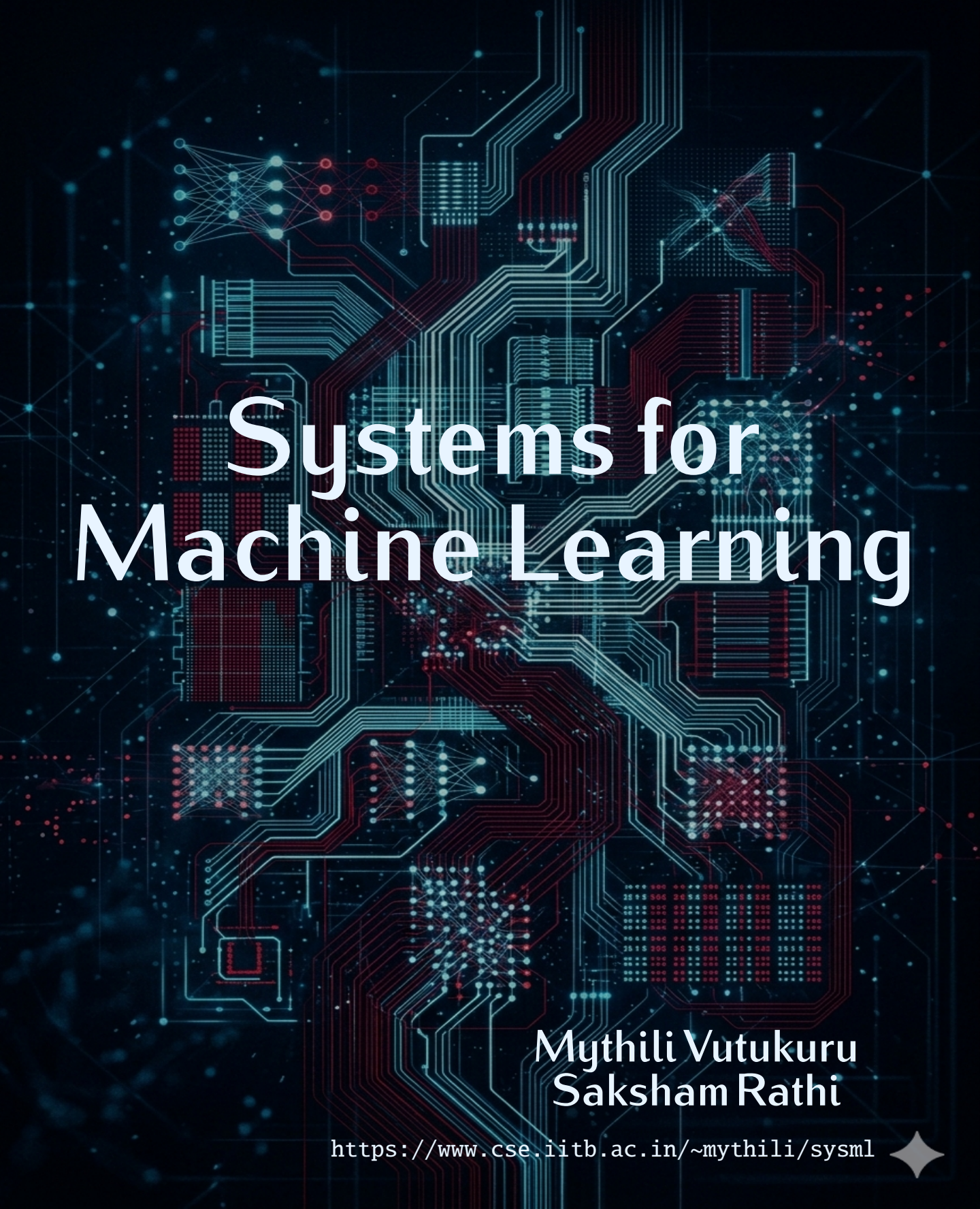

Systems for Machine Learning
Mythili Vutukuru
Saksham Rathi
https://www.cse.iitb.ac.in/~mythili/sysml

DRAFT CHAPTER

August 2026

4

Hardware-Aware Performance Optimizations

In the previous chapter, we studied the architectures of various AI accelerators such as GPUs and TPUs, and understood how their compute and memory subsystems work. We also learnt about CUDA programming and how to write GPU kernels. A natural question one must ask next is: how do we write code that uses the underlying hardware efficiently? This chapter describes a set of ideas that are used across the design of software ML frameworks and compilers, to write code which delivers high performance on hardware accelerators. The hardware-aware performance optimizations discussed in this chapter deal with three closely related questions.

The first question is about compute placement. An ML algorithm is a collection of tensor operations, e.g., matrix multiplications, convolutions, element-wise functions, which must be mapped onto the actual processing elements of the chip, e.g., CUDA cores, tensor cores, systolic arrays on a TPU. Writing performant code involves thinking carefully about how one must distribute these computations across the various processing elements, in a way that utilizes the compute optimally.

The second question is about memory management. ML algorithms need to store and process various data elements: inputs, weights, activations, and gradients during training. To achieve good performance, one needs to plan the memory layout and placement of these data elements across the memory hierarchy of the accelerators, from the fast on-chip caches to the slower off-chip HBM.

The final, and perhaps the most important question pertains to the movement of data between the memory hierarchy and the compute elements. Modern ML workloads involve computations on large tensors, while accelerators have limited memory access bandwidth between the compute elements and various levels of memory, especially off-chip memory. Therefore, data movement imposes a significant overhead, in terms of time and energy, and often ends up being the real performance bottleneck. Well-designed code not only places compute and data optimally, but also orchestrates data movement in a way that ensures compute units rarely stall for data.

This chapter describes several hardware-aware performance optimization techniques that help us fully utilize compute and memory resources, as well as manage data movement across them efficiently. We begin by discussing how to reason about performance and bottlenecks. We then discuss a series of practical and widely used optimization techniques to improve performance, e.g., batching, overlapping compute and data movement using parallel streams, choosing optimal memory

layouts, and so on. Next, we look more closely at the idea of tiling, and understand how it is applied to attention computation in the popular Flash Attention algorithm. Finally, we briefly touch upon optimizations that operate not just at the model implementation level, but at the model design itself, e.g., pruning and sparsity-aware attention kernels.

4.1 Understanding Performance

In this section, we will first understand what limits the performance of a GPU program. A GPU kernel can be "slow" either because the processing elements are busy doing arithmetic and everything else is waiting on them, or the processing elements are stalled, waiting for data to arrive from memory. These two regimes are called compute-bound and memory-bound respectively. This understanding is the first step to applying suitable performance optimization techniques that we will study later.

4.1.1 Compute vs. Memory

Every operation that runs on a GPU needs two things to happen: the data it operates on has to be fetched from memory, and the actual arithmetic has to be carried out on that data. The time taken by an operation therefore depends on two quantities: the time taken to move the required data from memory to the compute unit, which we call the communication time, and the time taken to perform the arithmetic itself, which we call the compute time. If the GPU were only able to do one of compute or communication at a time, the total time taken for an operation would simply be the sum of the compute and communication times.

$$T_{\text{total}} = T_{\text{compute}} + T_{\text{memory}}$$

In practice, however, modern accelerators are designed so that computation and data movement can happen at the same time on different parts of the hardware. While one set of data is being processed inside the compute units, another set of data can already be on its way from memory. Assuming this overlap is perfect, the total time taken by an operation is no longer the sum of the two times, but simply the larger of the two:

$$T_{\text{total}} = \max(T_{\text{compute}}, T_{\text{memory}})$$

If T_{compute} is the larger of the two terms, then no matter how fast we make the data transfer, the operation cannot finish any sooner than the time it takes to do the arithmetic. In this case, we say the operation is compute-bound. On the other hand, if T_{memory} is the larger term, then the arithmetic units will likely be idle and stalled for data most of the time, and the operation is memory-bound.

This discussion leads to the following conclusion. The performance achieved by an ML model does not depend only on the number of operations it requires, and fewer operations does not always mean faster code. On real hardware, an ML algorithm that does fewer operations per byte of data fetched can be slower than one that does more operations, simply because the compute units could be stalled for data in the first case.

Let us consider a small example. Suppose we have a GPU that can perform 1000 operations per second, and whose memory access bandwidth can deliver 100 bytes per second to the compute units. The ratio of these two numbers, 1000/100 = 10 operations per byte, is a property of the hardware itself, and it tells us how many operations the hardware can perform in the time it takes to fetch

one byte of data. Consider two ML algorithms, one that performs 20 operations for every byte of data it fetches, and one that performs 5 operations for every byte (Figure 4.1). In the case of the first algorithm, since the number of operations required per byte of data fetched is more than the hardware's capacity of 10 operations per byte, the performance is limited by the compute units, which will always have enough data available to process. Therefore, this algorithm is compute-bound, and will run at the GPU's full throughput of 1000 operations per second.

In contrast, for the second algorithm, the arithmetic units finish their work for a given chunk of data well before the next chunk has arrived, since the ML algorithm can only perform 5 operations per byte $\times$ 100 bytes per second = 500 operations per second, which is well below the GPU's peak of 1000 operations per second. This algorithm is now limited by how fast data can be supplied to the compute units, which in turn depends on the memory access bandwidth. This algorithm is therefore memory-bound.

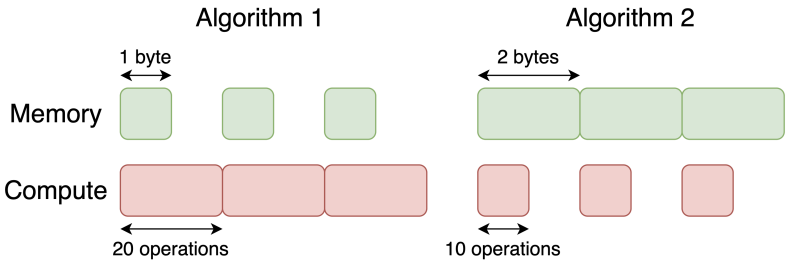


Figure 4.1: Example memory and compute of ML algorithms

In general, for this particular GPU, any algorithm that performs fewer than 10 operations per byte will be memory-bound, and any algorithm that performs more than 10 operations per byte will be compute bound. This threshold of 10 operations per byte is entirely a property of the hardware, and every accelerator has its own such threshold, determined by the ratio of its peak compute throughput to its peak memory bandwidth.

4.1.2 Arithmetic Intensity

We define arithmetic intensity (or operational intensity) as the ratio of the number of arithmetic operations required by an ML algorithm to the number of bytes of data accessed.

$$\text{Arithmetic Intensity} = \frac{\#\text{ops}}{\#\text{bytes accessed}}$$

Note that arithmetic intensity is a property of the algorithm and its implementation, not of the hardware it runs on. Whether a given ML algorithm is compute-bound or memory-bound on a given hardware depends on how the algorithm's arithmetic intensity compares to the ratio of peak throughput to peak memory bandwidth of the accelerator itself, as we describe below formally.

If an ML algorithm is compute-bound, it means that $T_{\text{compute}} > T_{\text{memory}}$, which further implies:

$$\frac{\text{Computation FLOPs}}{\text{Accelerator FLOPs/s}} > \frac{\text{Communication Bytes}}{\text{Bandwidth Bytes/s}}$$

Rearranging this expression:

$$\frac{\text{Computation FLOPs}}{\text{Communication Bytes}} > \frac{\text{Accelerator FLOPs/s}}{\text{Bandwidth Bytes/s}}$$

which implies

$$\text{Intensity(Operation)} > \text{Intensity(Accelerator)}$$

So an algorithm is compute-bound when its arithmetic intensity exceeds the intensity of the hardware it is running on, and it is memory-bound otherwise. This is an important insight when optimizing the performance of an ML model. For example, if we find that an algorithm's implementation is memory-bound, we must find a away to do more compute for every byte of data fetched, so that its intensity rises above the hardware's threshold. On the other hand, if it's arithmetic intensity is higher than that of the hardware already, we must focus on reducing the amount of compute involved itself, to efficiently use the processing elements.

Let us now compute the arithmetic intensity of some common ML operations. Consider multiplying an $M \times N$ matrix with an $N \times K$ matrix. The number of arithmetic operations required is roughly $2 \times M \times N \times K$ (factor of 2 because we count multiply and add separately), while the number of bytes that need to be read is proportional to $M \times N + N \times K + M \times K$, the sizes of the two input matrices and the output matrix. If all three matrices are square, with side length of N, the number of operations grow as N^3 while the number of bytes accessed grows as N^2, so the arithmetic intensity grows as $\frac{N^3}{N^2} = O(N)$. This tells us something important: matrix multiplication becomes more compute-bound as the matrices get larger. A linear layer applied to a small batch of data, where N is small, may well be memory-bound, while the same linear layer applied to a large batch, where N is large, is much more likely to be compute-bound. This is one of the reasons why batching, which we will discuss shortly, helps so much with GPU utilization.

In contrast to matrix multiplication, several simple operations in neural networks have very low arithmetic intensity, because they do very little arithmetic for every byte of data they touch. Consider the ReLU activation function, which simply replaces every negative value with zero. For every element, ReLU performs a single comparison-and-select operation, while it has to read the element (typically 4 bytes, if stored as a 32-bit float) and write it back out. If we only count the bytes read, the arithmetic intensity of ReLU is $1/4 = 0.25$, which is quite low. Similarly, element-wise addition of vectors, say, adding two vectors of length N each, requires N addition operations, but reads $2N$ input elements and writes N output elements back, resulting in an arithmetic intensity less than 1 operation per element accessed. Such element-wise operations are almost always memory-bound on modern hardware because of the low arithmetic intensity.

This observation has an important consequence. Computations in an ML models often consist of matrix multiplications interleaved with element-wise operations like activation functions, normalization layers, residual additions, and so on. Even if the matrix multiplications themselves are compute-bound, the element-wise operations are often memory-bound, and without careful optimization, the time spent on these low-intensity operations can dominate the total runtime of the model. This is the motivation behind techniques like kernel fusion, which we will study in the next section.

4.1.3 Roofline Plot

A convenient way to visualize the relationship between an ML algorithm's arithmetic intensity and its performance is the roofline plot (Figure 4.2). The horizontal axis of the plot is the arithmetic intensity of an algorithm, measured in FLOPs per byte, and the vertical axis is the maximum possible performance that can be achieved by the algorithm, measured in FLOPs per second. Both axes are typically drawn on a logarithmic scale. The plot gets its name from the shape it traces out, which looks like the roof of a house.

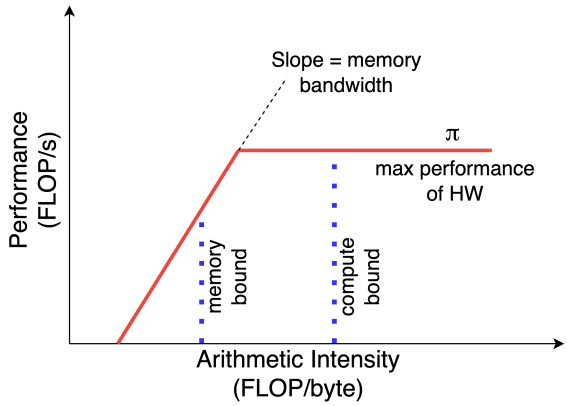
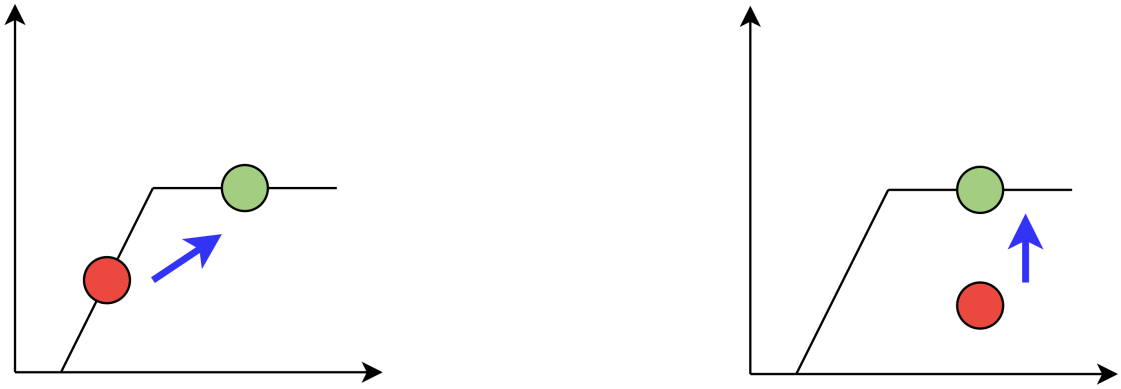


Figure 4.2: Roofline plot

For low values of arithmetic intensity, the performance of an ML algorithm is memory bound, and is limited by how quickly data can be supplied to the compute units. That is, the maximum achievable FLOPs/sec is simply the product of the arithmetic intensity (FLOPs/byte) and the memory access bandwidth (bytes/sec). The roofline plot in this region is a straight diagonal line whose slope is equal to the memory bandwidth of the hardware. As arithmetic intensity increases, however, the diagonal line eventually meets a horizontal ceiling, set by the peak compute throughput of the hardware, often denoted by π. No matter how high the arithmetic intensity of an algorithm is, its performance can never exceed this ceiling, because the compute units themselves cannot run any faster. An ML algorithm whose arithmetic intensity falls in this region is compute-bound. The point at which the diagonal line meets the horizontal ceiling is exactly the hardware's own arithmetic



(a) Memory-bound

(b) Compute-bound

Figure 4.3: Memory-bound and compute-bound operations

intensity: the ratio of peak compute throughput to peak memory bandwidth. Note that the roofline plot only gives the maximum achievable performance estimate, and inefficient implementations can often fail to achieve this upper bound.

The roofline plot is a useful diagnostic tool in practice. Given the specifications of an accelerator, which tell us its peak FLOPs per second and its peak memory bandwidth, and given the arithmetic intensity of a particular algorithm, we can usually estimate whether the algorithm is compute-bound or memory-bound, and what type of optimization strategies are likely to help. If an algorithm is memory-bound (Figure 4.3a), the only way to improve its performance is to move it to the right along the diagonal, that is, to increase its arithmetic intensity. This can be achieved by techniques like tiling, kernel fusion, and caching, which increase the amount of compute performed per byte accessed. If, on the other hand, an algorithm is already compute-bound (Figure 4.3b), improving its arithmetic intensity further is unlikely to help increase performance. In such cases, techniques that make optimal use of compute resources, say, via batching, overlapping compute with data transfers, avoiding stalls for memory with optimized data layouts, all help ensure that a compute-bound algorithm can achieve performance close to the hardware's peak throughput. The next section examines several such performance optimization techniques.

Practice Problem 4.1

Given an input image G of size $N \times N$, and a convolution weights filter W of size $k \times k$, a simplified CNN layer computes the output image H of size $N \times N$ as: $H_{i,j} = \sum_{di} \sum_{dj} W_{di,dj} G_{i+di,j+dj}$

(i) How many FLOPs are required to compute the forward pass through this simple CNN layer for one image? You may ignore edge effects for the image. Count multiply and add as two separate operations. Assume

$k = 3$ throughout this question.

Solution: For each output pixel, we must multiply each of the 9 filter weights ($k \times k = 9$ for $k = 3$) with the corresponding input pixel and accumulate the sum, resulting in 9 multiply-add operations. Since we do this for every one of the $N \times N$ output pixels, and counting multiply and add separately, we get $18N^2$ FLOPs.

Note: $17N^2$ can also be accepted as a correct answer: summing 9 numbers only requires 8 addition operations (not 9), since the first product doesn't need to be "added" to anything.

(ii) How many bytes need to be fetched from memory to the compute elements for one forward pass of the CNN layer for one image, assuming we can reuse data fetched from off-chip memory optimally? Count only the input image and filter weights that need to be read, and not the output that needs to be written. Assume 2 bytes per element in FP16 representation.

Solution: With optimal reuse, each input pixel $G_{i,j}$ only needs to be fetched from memory once and can then be reused (e.g., held in registers or fast on-chip memory) across all the output computations that need it, rather than being re-fetched from memory every time it's used in a different sliding-window position. This means we only need to fetch the N^2 elements of the input image once. Similarly, the 9 filter weights are shared across every single output pixel computation, so they too only need to be fetched once from memory and reused N^2 times. Adding these together gives $N^2 + 9$ elements, and multiplying by 2 bytes per FP16 element gives a total of $2(N^2 + 9)$ bytes.

(iii) Suppose the ops/bytes ratio of the GPU is $R < 9$. What is the minimum value of N for which the CNN computation becomes compute-bound?

Solution: A computation is compute-bound (rather than memory-bound) when the arithmetic intensity of the workload meets or exceeds the GPU's ops/byte ratio R. Using our results from parts (i) and (ii), the arithmetic intensity of this CNN layer is $\frac{18N^2}{2(N^2+9)}$, so we require:

$$\frac{18N^2}{2(N^2+9)} \geq R$$

Solving this inequality for N, we get the minimum value:

$$N \geq \sqrt{\frac{9R}{9-R}}$$

Note that this requires $R < 9$ for the expression under the square root to remain positive, which matches the constraint given in the problem (since the maximum possible arithmetic intensity, achieved as $N \to \infty$, approaches 9).

4.2 Performance Optimization Techniques

The architectures of AI accelerators bestow them with certain strengths and weaknesses. For example, GPUs can efficiently execute massively parallel computations over large tensors, but do not do as well with simple element-wise operations with low arithmetic intensity. In this section, we discuss several optimization techniques that leverage such knowledge of underlying accelerator hardware architecture to improve the performance of ML algorithm implementations. Later in this chapter, we will study hardware-aware optimization techniques applied to the model design itself, which trade off model accuracy for better performance on modern accelerators.

4.2.1 Batching

Consider a single feed-forward layer of a neural network. Suppose the input of this layer has size N, and the hidden layer has size M. The input can be represented as a $1 \times N$ row vector, and the weights of the layer can be represented as an $N \times M$ matrix. The output of the layer is then obtained by a simple matrix-vector multiplication, producing a $1 \times M$ vector, as shown below:

$$\underbrace{[1 \times N]}_{\text{input}} \times \underbrace{[N \times M]}_{\text{weights}} = \underbrace{[1 \times M]}_{\text{output}}$$

While this operation is mathematically simple, it is a poor fit for the architecture of a GPU. A single matrix-vector multiplication does not expose very much parallel work, and worse, every weight in the $N \times M$ matrix is used only once before it is discarded. The amount of computation performed per byte of data moved from memory is quite low, and such computations with low arithmetic intensity tend to be memory-bound. Finally, matrix-vector multiplications cannot leverage accelerator cores that are specifically designed for matrix-matrix multiplications, e.g., tensor cores.

Suppose we process a batch of B inputs at once instead, stacked into a matrix of shape $B \times N$. The computation now becomes a matrix-matrix multiplication (Figure 4.4).

$$\underbrace{[B \times N]}_{\text{batched input}} \times \underbrace{[N \times M]}_{\text{weights}} = \underbrace{[B \times M]}_{\text{batched output}}$$

This mechanism of batching has several desirable consequences. First, the same weight matrix fetched from memory can potentially be reused multiple times across all B inputs in the batch, which improves the arithmetic intensity of the operation considerably. Second, because the B rows of the input are independent of one another, the GPU now has B times as much independent work to distribute across its cores, which allows it to keep many more threads, warps, and streaming multiprocessors busy simultaneously. Third, the fixed overhead of transferring data from the host to the device, and the fixed overhead of launching a kernel on the GPU, are now amortized over a much larger amount of useful computation, so the relative cost of these overheads shrinks as the batch size grows. Finally, once the operation becomes a matrix-matrix product rather than a matrix-vector product, it becomes possible to make use of the specialized tensor cores available on modern GPUs.

For these reasons, batching is a very commonly employed optimization techniques when running ML models, and allows more efficient use of the accelerator's hardware resources. Of course, very large batch sizes consume large amounts of memory to store intermediate layer activations, and may not be feasible due to GPU memory constraints.

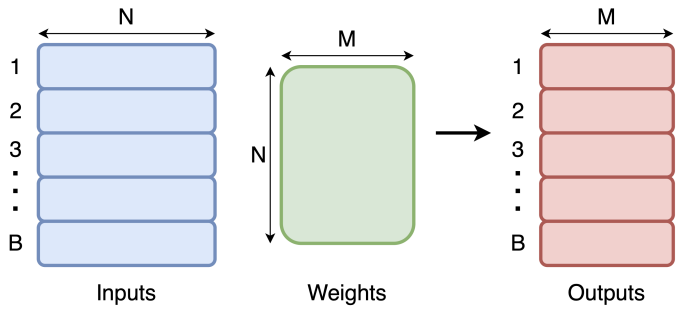


Figure 4.4: Batching

4.2.2 Data Reuse

Consider the multiplication of an input matrix and weight matrix to produce an output matrix. To compute a single output element at a compute core of the GPU, one entire row of the input matrix and one entire column of the weight matrix must be fetched from memory, multiplied element by element, and accumulated into a partial sum, which itself must be stored somewhere until the accumulation is complete. While the Single Instruction Multiple Thread (SIMT) execution model of a GPU provides a great deal of parallelism, and allows the computation of multiple output elements simultaneously by different threads, it alone does not guarantee that data fetched from memory is reused efficiently. If every thread independently re-fetches the same row or the same column from slow off-chip memory, the computation will waste precious memory bandwidth.

Now, some amount of data reuse may happen naturally when rows and columns of the input matrices are stored in various levels of GPU caches. However, such reuse cannot always be guaranteed, especially for large matrix dimensions. A better way to address this problem is to explicitly keep one of the input matrices stationary in a small, fast, on-chip memory (which is called shared memory for GPUs), while the other input is streamed in and out of this fast memory as needed (assuming both inputs do not fit in the on-chip memory). Because the stationary matrix remains close to the arithmetic units for an extended period of time, it can be reused many times without being re-fetched from slow memory, and only the streaming matrix needs to be repeatedly loaded. Such explicit data reuse via programmer-managed caches is widely used across different ML algorithm implementations today. For example, the smaller filter weights of a CNN are typically pinned in shared memory, and reused multiple times, while the much larger activation maps are streamed in and out. Similarly, the keys and values in a transformer can be pinned and reused across attention computation across

multiple tokens, especially in the decode phase.

To understand data reuse using shared memory, consider the following two examples of a CNN convolution kernel.

Code 4.1: Naive CNN Implementation

```
__global__ void naive_conv2d(const float* input, const float* weight, float*
    output, int H, int W, int K) {
    int out_row = blockIdx.y * blockDim.y + threadIdx.y;
    int out_col = blockIdx.x * blockDim.x + threadIdx.x;
    int H_out = H - K + 1;
    int W_out = W - K + 1;

    if (out_row < H_out && out_col < W_out) {
        float acc = 0.0f;
        for (int ki = 0; ki < K; ++ki) {
            for (int kj = 0; kj < K; ++kj) {
                // both weight and input re-fetched from HBM
                // for every single output element
                float w = weight[ki * K + kj];
                float x = input[(out_row + ki) * W + (out_col + kj)];
                acc += w * x;
            }
        }
        output[out_row * W_out + out_col] = acc;
    }
}

```

In the above naive formulation, every thread computes one output element and independently walks the entire $K \times K$ filter, issuing a global memory load for `weight[ki][kj]` on every iteration. Since every one of the $H_{out} \times W_{out}$ output elements requires its own pass over all $K \times K$ filter weights, the same weight values are fetched from HBM $H_{out} \times W_{out}$ times over the course of the computation, even though there are only $K \times K$ distinct weight values in total. The number of HBM accesses for weights is $H_{out} \cdot W_{out} \cdot K^2$, and the number of HBM accesses for the input is $H_{out} \cdot W_{out} \cdot K^2$, since each input pixel participating in multiple overlapping patches is likewise re-read from HBM once per output element that touches it.

Code 4.2: CNN Implementation with Weights Pinned in Shared Memory

```
__global__ void shared_weight_conv2d(const float* input, const float* weight,
    float* output, int H, int W, int K) {
    extern __shared__ float s_weight[]; // K*K filter weights, pinned on-chip

    int tid = threadIdx.y * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * blockDim.y;

    // cooperatively load the filter into shared memory once per block
    for (int idx = tid; idx < K * K; idx += num_threads) {
        s_weight[idx] = weight[idx]; // only fetch from HBM here
    }
    __syncthreads();

    int out_row = blockIdx.y * blockDim.y + threadIdx.y;
    int out_col = blockIdx.x * blockDim.x + threadIdx.x;
    int H_out = H - K + 1;
    int W_out = W - K + 1;

    if (out_row < H_out && out_col < W_out) {
        float acc = 0.0f;
        for (int ki = 0; ki < K; ++ki) {
            for (int kj = 0; kj < K; ++kj) {
                // weight reused from shared memory, no HBM access
                float w = s_weight[ki * K + kj];
                float x = input[(out_row + ki) * W + (out_col + kj)];
                acc += w * x;
            }
        }
        output[out_row * W_out + out_col] = acc;
    }
}

```

In this optimized version shown above, each thread block loads the $K \times K$ filter into shared memory exactly once, using $K \times K$ total HBM accesses per block, and every thread in that block then reuses the pinned weights directly from shared memory for the rest of its computation. Assuming the kernel is launched with a single block covering the whole output, the number of HBM accesses for weights drops to K^2, independent of the output size, compared to $H_{out} \cdot W_{out} \cdot K^2$ in the naive case — a reduction by a factor of $H_{out} \cdot W_{out}$. The input, which is not pinned in this example, is still

streamed from HBM on every access, as in the naive case. This illustrates the core idea of data reuse: by pinning the small, frequently-reused filter weights on-chip and only streaming the much larger input activations, the kernel eliminates redundant HBM traffic for the weights.

Beyond the simple idea of using shared memory caches, data reuse can also be improved by a technique called tiling, which we will study in more detail in § 4.3.

4.2.3 Kernel Fusion

An obvious way to implement a sequence of operations in a neural network model is to use one GPU kernel for each operation. While this is simple to implement, it can be very inefficient, because every kernel must read its input from, and write its output to, the GPU's main memory, which is slow to access. Consider a simple example of a linear layer followed by a ReLU activation, expressed as $Z = XW, \quad A = \text{ReLU}(Z)$. If this computation is implemented using two separate kernels, one for the matrix multiplication, and one for the activation function, then the following sequence of memory operations takes place. The matrix multiplication kernel loads X and W from HBM, computes Z, and writes Z back to HBM. The activation kernel then loads Z back from HBM, computes $A = \text{ReLU}(Z)$, and writes A back to HBM. The intermediate result Z is therefore written to slow memory and then immediately read back from it, even though it is never needed for anything other than computing A.

Kernel fusion avoids this wasted traffic by combining both operations into a single kernel. In a fused kernel, each thread computes its portion of Z, immediately applies the ReLU activation to that value while it is still present in a fast, on-chip register, and then writes only the final result A to HBM. The intermediate value Z never needs to be written to, or read back from, slow memory at all, which significantly reduces the total amount of memory traffic for the combined operation. The price paid for this benefit is that the fused kernel must hold more intermediate values in registers, or in shared memory, at once, which increases register pressure and can, if taken too far, reduce the number of threads that can be scheduled concurrently on a streaming multiprocessor. Kernel fusion is therefore most beneficial for sequences of simple, element-wise operations, where the savings in memory traffic far outweigh the modest increase in register usage.

The following CUDA codes illustrate the difference between an unfused and a fused implementation of this example. The unfused version uses two separate kernels – one to multiply a batch of inputs with a weight matrix to produce the hidden layer outputs, and one to perform the ReLU activation — with an intermediate buffer in global memory holding the output from the first kernel that is fed as input to the second kernel. On the other hand, the fused version computes the activation immediately after the matrix multiplication, without ever writing the intermediate result to global memory.

Code 4.3: Unfused Matmul and ReLU Kernels

```
// Kernel 1: computes Z = X * W and writes Z to global memory
__global__ void matmul_kernel(const float* X, const float* W,
                              float* Z, int Bsz, int N, int M) {
    int row = blockIdx.y * blockDim.y + threadIdx.y; // batch index
    int col = blockIdx.x * blockDim.x + threadIdx.x; // hidden unit index

    if (row < Bsz && col < M) {
        float acc = 0.0f;
        for (int k = 0; k < N; ++k) {
            acc += X[row * N + k] * W[k * M + col];
        }
        Z[row * M + col] = acc; // intermediate result written to HBM
    }
}

// Kernel 2: reads Z back from global memory and applies ReLU
__global__ void relu_kernel(const float* Z, float* A, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        float z = Z[idx];          // read intermediate result from HBM
        A[idx] = z > 0.0f ? z : 0.0f;
    }
}

// Host-side launch sequence
matmul_kernel<<<gridDim1, blockDim1>>>(X, W, Z, Bsz, N, M);
relu_kernel<<<gridDim2, blockDim2>>>(Z, A, Bsz * M);
```

Code 4.4: Fused Matmul + ReLU Kernel

```
// Fused kernel: computes Z = X * W and applies ReLU in registers,
// writing only the final activated output A to global memory
__global__ void fused_matmul_relu_kernel(const float* X, const float* W,
                                         float* A, int Bsz, int N, int M) {
    int row = blockIdx.y * blockDim.y + threadIdx.y; // batch index
    int col = blockIdx.x * blockDim.x + threadIdx.x; // hidden unit index

    if (row < Bsz && col < M) {
        float acc = 0.0f;
        for (int k = 0; k < N; ++k) {
            acc += X[row * N + k] * W[k * M + col];
        }
        // ReLU applied directly on the register-resident value acc;
        // the unactivated intermediate value is never stored to HBM
        A[row * M + col] = acc > 0.0f ? acc : 0.0f;
    }
}

// Host-side launch sequence: a single kernel launch instead of two
fused_matmul_relu_kernel<<<gridDim, blockDim>>>(X, W, A, Bsz, N, M);
```

Notice that in the fused kernel, the variable `acc` plays the role of Z, but it lives only in a register for the lifetime of the thread, and is never written to or read from global memory.

4.2.4 Improving GPU Compute Utilization

While improving memory reuse and increasing arithmetic intensity can improve the performance of memory-bound operations, compute-bound operations can be optimized mainly by increasing the efficiency and utilization of the GPU's processing cores. This can be achieved in several ways. GPU programs must ensure that there is enough parallel work, in the form of threads and thread blocks, submitted to the GPU scheduler to dispatch to the compute cores. This often means choosing kernel launch configurations carefully, considering the number of streaming multiprocessors on the target device. Further, one must optimize the resource usage (e.g., shared memory and registers) of the threads in a kernel, so that enough warps can be resident concurrently on each SM, improving occupancy. The memory access pattern of threads in a warp should be optimized for coalesced memory accesses, and caches should be utilized where possible to reduce memory stalls. One must also be wary of unintended effects like warp divergence and wave quantization that leave SMs under-utilized for some periods of time. The previous chapter on GPU architecture has explored these topics in depth.

Where communication with the host device or other GPUs is involved, we must ensure that the GPU compute units are not stalled excessively for communication, by overlapping compute with communication where possible. For example, in a typical training or inference pipeline, data is transferred from host memory to device memory, the GPU performs its computation on that data, and results are then transferred back to the host. If these activities (host-to-device transfer and device computation) are performed one after another, the GPU sits idle during data transfer, and the data transfer engine sits idle during computation (Figure 4.5). By issuing data transfers asynchronously within parallel CUDA streams, it becomes possible to overlap the data transfer of one CUDA stream with the computation on the other, improving utilization of GPU compute. Similarly, when a model running on the GPU needs to send or receive data from other GPUs over the network, one can overlap the network communication with compute as far as possible, e.g., by pre-fetching data for a future computation over the network, while the current computation is underway on the GPU.

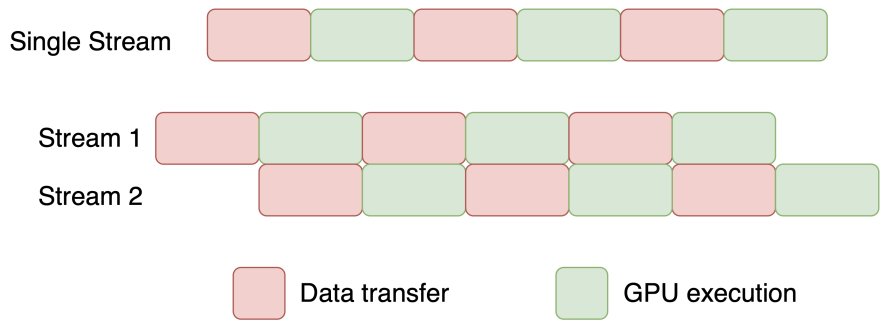


Figure 4.5: Overlapping data transfer with GPU execution with multiple streams

4.2.5 Memory Layout

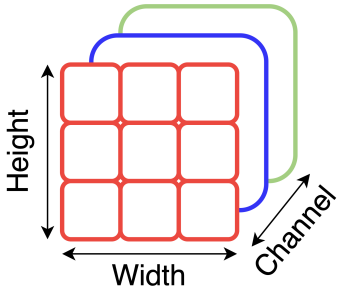


Figure 4.6: Memory layout

The technique we discuss next concerns not the computation itself, but the way data is arranged in memory before any computation takes place. The same logical tensor can be laid out in memory in more than one way, e.g., a matrix can be stored in row-major order or column-major order, and this choice of the layout can have a substantial effect on performance.

Consider an image of height H, width W, with each pixel having values corresponding to the C channels (Figure 4.6). This image can be represented as a tensor of shape (H, W, C). One common way to store this image in memory is in row-major order, where the image is laid out row by row, and for each pixel of a row, all channels are stored together contiguously. This layout is usually called NHWC, standing for batch, height, width, channel, and it places the channel dimension as the fastest-varying, or innermost dimension in memory. As an example, for a small image with $H = 2$, $W = 2$, and $C = 3$, the memory layout of the image values looks like below (indices written as $I(H, W, C)$):

$$I(0,0,0), I(0,0,1), I(0,0,2), I(0,1,0), I(0,1,1), I(0,1,2),$$
$$I(1,0,0), I(1,0,1), I(1,0,2), I(1,1,0), I(1,1,1), I(1,1,2)$$

An alternative way to store the same image is in a channel-major layout, where all of the values belonging to a single channel, across every row and every column of the image, are stored together, before moving on to the next channel. This layout is usually called NCHW, standing for batch, channel, height, width, and it places the channel dimension as one of the slower-varying, or outer dimensions in memory. The example image layout in this case will look like:

$$I(0,0,0), I(0,1,0), I(1,0,0), I(1,1,0),$$
$$I(0,0,1), I(0,1,1), I(1,0,1), I(1,1,1),$$
$$I(0,0,2), I(0,1,2), I(1,0,2), I(1,1,2)$$

The choice between these two layouts, and more generally the choice between row-major and column-major storage for any multidimensional array, matters because of how memory accesses are performed. Memory systems on both CPUs and GPUs are organized around cache lines, which are blocks of contiguous bytes that are fetched from memory together as a single unit. When a program accesses memory in a pattern that matches the contiguous layout of the underlying array, an access to one element tends to bring nearby elements into caches as well, to be accessed almost for free when they are needed. Such spatial locality also leads to effective hardware-level prefetching when the hardware enables such optimizations. On a GPU, this contiguity in memory accesses has an additional and very important consequence. As we saw in the previous chapter, when consecutive threads in a warp access consecutive memory addresses, the hardware can coalesce these many individual memory requests into a single, wide memory transaction known as a DRAM burst. On the other hand, if the addresses accessed by consecutive threads are scattered far apart in memory,

the hardware must issue many separate, narrow transactions, and a large fraction of the available memory bandwidth is wasted. Therefore, contiguous memory accesses in a warp are even more important in GPUs.

Note that there is no single layout that is best for every situation; the right layout depends on how the data is processed during computation. If an operation needs to look at all of the channels of a given pixel together, e.g., when applying a point-wise operation that mixes information across channels, then the row-major (NHWC) layout is preferable, because all of the channel values for a single pixel are already contiguous in memory and can be loaded together very efficiently. If, on the other hand, an operation applies a filter independently to each channel, then the channel-major (NCHW) layout is preferable, because all the values belonging to a single channel are stored contiguously, and can be processed via coalesced memory accesses. Choosing a layout that matches the access pattern of the dominant operation in a pipeline, and converting between layouts only when absolutely necessary, is therefore essential for high performance.

4.2.6 Shared Memory Bank Conflicts

The technique of keeping data stationary in shared memory, described above, is only beneficial if the accesses to shared memory are themselves efficient. Shared memory is physically divided into a fixed number of equally sized modules, called banks, and on most current GPU architectures there are 32 such banks, matching the number of threads in a warp. Successive four-byte words in shared memory are assigned to successive banks in a round-robin fashion, so that the first word belongs to bank zero, the second word belongs to bank one, and so on, wrapping around to bank zero after every thirty-second word. Each bank is capable of servicing exactly one memory request per clock cycle. As long as 32 threads of a warp access 32 different banks, all of these accesses can be serviced simultaneously, in a single transaction, and shared memory delivers its full, high bandwidth.

A bank conflict occurs when two or more threads within the same warp attempt to access different addresses that happen to fall within the same bank, during the same instruction. Because each bank can only service one request per cycle, the hardware cannot satisfy these accesses at the same time. Instead, it serializes them, servicing one thread's request, then the next, and so on, until every conflicting request has been handled, before the warp is allowed to execute. It is worth emphasizing that a bank conflict occurs only when threads access different addresses within the same bank; if multiple threads in a warp read the exact same address, the hardware is able to service it without any conflict, by broadcasting the data to every requesting thread simultaneously.

As an example to illustrate bank conflicts, consider a 2D array stored in shared memory, laid out in row-major order, where each row occupies exactly 32 words, matching the number of banks. If threads in a warp all access the elements of one row, then each thread accesses a different bank, and there is no bank conflict. On the other hand, if a kernel is written so that threads of the same warp access the same column, moving down successive rows of the array, then every thread is separated from its neighbour by a stride of 32 words and lands on exactly the same bank (Figure 4.7a). Hence, the entire warp's access is serialized into thirty-two separate transactions. Consider the example

code shown below to perform a matrix transpose, where threads in a warp read a tile of the matrix into shared memory along the row, and write it back along the column. This code is an example of a kernel that has a complete bank conflict during the writing of the transpose matrix.

Code 4.5: Shared Memory Access with Bank Conflicts

```
#define TILE_DIM 32

__global__ void transpose_conflicted(const float* in, float* out, int N) {
    __shared__ float tile[TILE_DIM][TILE_DIM]; // 32 columns: stride = 32 words

    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;

    // Threads in a warp write consecutive columns of the same row:
    // consecutive addresses, consecutive banks, no conflict.
    tile[threadIdx.y][threadIdx.x] = in[y * N + x];
    __syncthreads();

    // Threads in a warp now read down a column: each thread's address
    // is threadIdx.x * TILE_DIM apart from its neighbor's, i.e. a
    // multiple of 32 words, so every thread lands on the same bank.
    out[x * N + y] = tile[threadIdx.x][threadIdx.y];
}
```

The most common technique for eliminating bank conflicts is called padding (Figure 4.7b). Rather than declaring the shared memory array with exactly as many columns as there are banks, one additional column is added to the declaration, so that each row occupies 33 words instead of 32, as shown in the code below. This single extra column has no effect on the amount of useful data stored, since the additional element in each row is simply never used, but it changes the stride between corresponding elements of successive rows from 32 words to 33 words. Because 33 is not a multiple of the number of banks, successive rows are now offset from one another by one bank position, and threads that previously all landed on the same bank when accessing a column are now spread across 32 different banks, one per row.

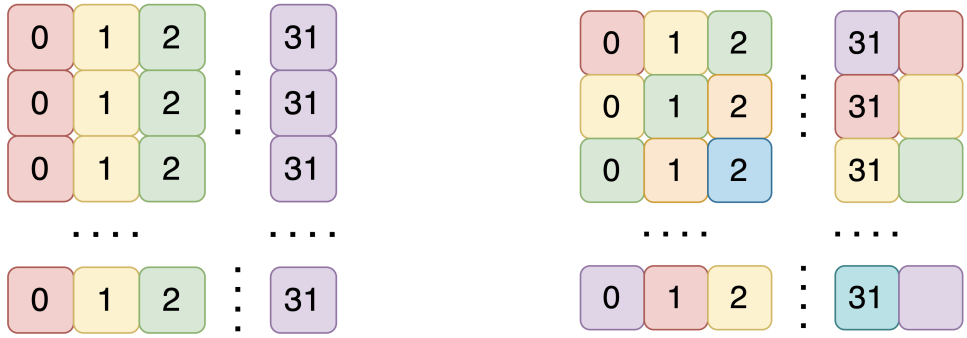
Code 4.6: Eliminating Bank Conflicts with Padding

```
#define TILE_DIM 32

__global__ void transpose_padded(const float* in, float* out, int N) {
    // One extra column per row shifts the stride from 32 to 33 words,
    // so that column accesses no longer collide on a single bank.
    __shared__ float tile[TILE_DIM][TILE_DIM + 1];

    // rest code remains the same as before
}
```

Padding is not the only way to avoid bank conflicts, and in some kernels it is preferable to restructure the access pattern itself, e.g., by having threads cooperate to load data in a row-major order and only rearrange it locally within registers, or by choosing tile dimensions that are not exact multiples of the number of banks in the first place.



(a) Without padding

(b) With padding

Figure 4.7: Layout of the array on shared memory, different colours show different banks, and the numbers inside boxes denote warp numbers accessing the array element. Without padding, threads of the same warp access the same coloured bank, while with padding, threads of the same warp land on different banks.

Practice Problem 4.2

Consider a toy GPU with a peak compute capacity 300 FLOP/s and memory bandwidth 30 bytes/sec.

(i) Draw the roofline plot for this toy GPU. Clearly write the x-axis and y-axis labels (along with units), and also mark the x and y coordinates of the ridge/transition point.

Solution: The roofline plot has arithmetic intensity (FLOP/byte) on the x-axis and achievable performance (FLOP/s) on the y-axis, both typically shown on log-log scales. The plot consists of a sloped line, given by Performance = Bandwidth × Arithmetic Intensity = 30 × AI, which represents the maximum performance achievable when the workload is memory-bound, and a horizontal line at $y = 300$ FLOP/s, representing the GPU's peak compute capacity, which caps performance regardless of how high the arithmetic intensity is. The ridge point, where these two lines intersect, marks the transition between memory-bound and compute-bound regimes: setting $30 \times \text{AI} = 300$ gives an x-coordinate of AI = 10 FLOP/byte and a y-coordinate of 300 FLOP/s, so the ridge point is at $(10, 300)$.

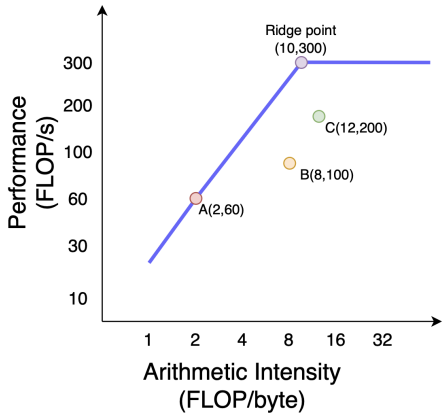


Figure 4.8: Roofline Plot

Given below are several ML operations, their arithmetic intensity, and the maximum throughput (FLOP/s) achieved by the GPU while running the operation. In each case, mention a possible performance optimization which you can use to improve the performance of the operation, and also provide a justification as to why the optimization is suitable.

(ii) Operation with arithmetic intensity 2, and maximum throughput 60 FLOP/s.
Solution: Since the ridge point occurs at an arithmetic intensity of 10, an arithmetic intensity of 2 places this operation well to the left of the ridge of the roofline plot, making it memory-bound. A useful optimization is kernel fusion; by merging multiple memory-bound operations into one kernel, we avoid writing intermediate results back to global memory and reading them again in the next kernel, reducing the round trips to global memory and effectively increasing the arithmetic intensity, which pushes the operating point up and to the right toward the ridge.

(iii) Operation with arithmetic intensity 8, and maximum throughput 100 FLOP/s.
Solution: An arithmetic intensity of 8 is still less than the ridge-point value of 10. Further, the performance is also not on the roofline curve itself, since $30 \times 8 = 240 \neq 100$, meaning the GPU is achieving less than the theoretical memory-bound peak performance. So this is an example of a memory-bound operation, and additionally suffering from inefficient memory accesses. One possible optimization is memory coalescing, which ensures that threads accessing memory do so in contiguous, aligned patterns so that the memory

bandwidth is better utilized, bringing actual throughput closer to the theoretical roofline limit.

(iv) Operation with arithmetic intensity 12, and maximum throughput 200 FLOP/s.
Solution: Since the arithmetic intensity is 12, which is greater than the ridge-point value of 10, the operation lies to the right of the ridge and could be compute-bound. However, the achieved throughput of 200 FLOP/s is well below the peak compute throughput of 300 FLOP/s, so we cannot conclude that compute is the only bottleneck. One possibility is that the GPU is not fully utilizing its compute resources due to insufficient parallelism. In that case, increasing the amount of independent work (e.g., launching enough threads or better overlapping computation and communication) can improve performance. Another possibility is that the operation is still limited by memory bandwidth in practice because of inefficient memory accesses or cache behaviour, preventing the GPU from supplying data fast enough to sustain peak compute throughput. In that case, memory optimizations such as improving memory coalescing or cache locality may also increase performance.

Practice Problem 4.3

Consider a shared memory array in a GPU, of 32×32 elements (`smem[32][32]`), which is accessed by a thread block of 32×32 threads. The array is laid out in shared memory in a row-major order, and every set of 32 consecutive elements (starting from the first element) in the array are served from separate banks in parallel. Threads in a warp can access the shared memory array either row-wise or column-wise.

(i) Will one of the above two methods (threads in a warp reading the shared memory elements along rows vs along columns) result in significantly lower shared-memory bank conflicts as compared to the other? If yes, which one, and why?
Solution: Since the array is stored in row-major order and there are exactly 32 banks (one for each of the 32 consecutive elements in memory), each element within a single row maps to a distinct, consecutive bank, i.e., `smem[i][0]` is in bank 0, `smem[i][1]` is in bank 1, and so on up to `smem[i][31]` in bank 31. Therefore, threads in a warp accessing row-wise (i.e., 32 threads reading `smem[i][0]` through `smem[i][31]` for a fixed row i) will result in much lower bank conflicts, because each thread in the warp reads from a different bank, allowing all 32 accesses to be serviced in parallel in a single memory transaction. In contrast, when accessing the array column-wise (i.e., 32 threads reading `smem[0][j]` through `smem[31][j]` for a fixed column j), every element accessed lies exactly 32 elements apart in memory, and hence maps to the very same bank. As a result, threads of a warp will read from the same bank when accessing the array column-wise, causing a 32-way bank conflict, where the accesses must be serialized instead of serviced in parallel, significantly degrading performance.

(ii) Now suppose we pad the shared memory array to have an extra row (`smem[33][32]`). Because we have more array elements than threads, assume that the threads in the block access only the original 32x32 array, and the extra padded row is not assigned to any thread. What is your answer to part (a) now?
Solution: Padding with an extra row only adds 32 unused elements at the very end of the array, without changing the number of columns. Since the stride between consecutive elements within a row, and between corresponding elements in consecutive rows, is unaffected by an extra row tacked on at the end, the extra padded row does not change the bank mappings of the original 32x32 shared memory array accessed by threads. Row-wise access still spreads a warp across 32 distinct banks, and column-wise access still

conflicts on a single bank, so the answer remains the same as above.

(iii) Now suppose we pad the shared memory array to have an extra column (`smem[32][33]`). Because we have more array elements than threads, assume that the threads in the block access only the original 32x32 array, and the extra padded column is not assigned to any thread. What is your answer to part (a) now?
Solution: With the extra padded column, each row has 33 elements, so the stride between `smem[i][j]` and `smem[i+1][j]` is 33 instead of 32. Since 33 mod 32 = 1, moving down one row shifts the bank index by one, so column-wise accesses are distributed across all 32 banks instead of mapping to the same bank. Row-wise accesses are unchanged, as consecutive elements in a row already map to different banks. Therefore, neither row-wise nor column-wise accesses incur bank conflicts, and both achieve similar performance.

4.3 Tiling

Many of the workloads we care about in machine learning systems operate on data that is far too large to sit comfortably in the small, fast memories close to the compute units. As we saw in the previous chapter, a GPU streaming multiprocessor (SM) has a tiny register file and a modestly sized shared memory or L1 cache. Yet the matrices it is asked to multiply can easily contain millions of elements stored in high bandwidth memory (HBM) or, worse, all the way back in host memory. If we naively stream the full operands through the compute units, we may move the same data back and forth many times, and the system quickly becomes memory bound rather than compute bound.

Tiling is the general technique of breaking a large computation over large data into many smaller sub-computations over small blocks of data, called tiles. Instead of operating on an entire array at once, we carve both the data and the computation into tiles that are small enough to fit into a local or cached memory. We bring one tile (or a small working set of tiles) into fast memory, perform every computation that can be done using only that data, and only then move on to the next tile. Because each tile, once resident in fast memory, is reused for many arithmetic operations before being evicted, tiling directly increases data reuse, reduces the total number of slow memory accesses, and pushes the computation toward better compute efficiency. This idea is not specific to any one algorithm; it shows up in matrix multiplication, convolution, attention, reductions, and essentially any computation where the same piece of data participates in many arithmetic operations. We will use matrix multiplication as our running example to illustrate tiling, because it is simple to reason about, and because it is the single most important computational primitive in deep learning.

4.3.1 Matrix Multiplication with Tiling

Consider multiplying two $N \times N$ matrices A and B to produce the product matrix $C = A \times B$. Every element of C is a dot product between a row of A and a column of B,

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$$

There are N^2 elements in C, and computing each one takes N multiply-add operations, so the total amount of arithmetic work is:

$$\#\text{ops} = N^2 \text{ elements} \times N \text{ multiply-add ops} = N^3$$

Now consider how much data movement this computation requires. If we are only worried about the one-time cost of bringing A and B from CPU memory onto the accelerator, then in principle we only need to move each matrix once, giving

$$\#\text{memory accesses to fetch } A \text{ and } B \text{ only once} = 2 \times N^2$$

This looks fine, but it hides a much more serious problem that occurs once the data is already on the device. The accelerator's cache or scratchpad is usually far too small to hold all of A and B at the same time, especially for the large matrices that show up in practice. If A and B do not fit in the cache, then computing a single element of C requires fetching a full row of A and a full column of B from the slow memory (HBM) into the cache, and as soon as we move on to the next element of C, the corresponding rows and columns may already have been evicted, forcing us to fetch them all over again. In the worst case, every one of the N^2 elements of C requires its own fresh row-and-column fetch, so

$$\text{\#memory accesses to fetch row of } A \text{ and col of } B \text{ for all elements of } C = N^2 \times 2 \times N = 2N^3$$

Now, the number of memory accesses scales as N^3, exactly the same order as the number of arithmetic operations. This means the ratio of operations to bytes fetched stays roughly constant as N grows, and matrix multiplication may end up remaining memory-bound. This is exactly what tiling is designed to fix.

With tiling, we think of A, B, and C as collections of smaller blocks, or tiles. Suppose we divide each matrix into tiles of size $S \times S$, so that there are $K = \frac{N}{S}$ tiles along each dimension. The product matrix C is then naturally divided into a $K \times K$ grid of tiles, and each tile of C is itself the sum of several smaller matrix products between tiles of A and tiles of B.

Let us take a small example, as shown in Figure 4.9. Let $N = 4$ and let tile size $S = 2$, so $K = 2$. Label the sixteen entries of A and B from 0 to 15 in row-major order. The matrix A is split into a first set of tiles, containing entries $\{0, 1, 4, 5\}$, and a second set of tiles, containing entries $\{2, 3, 6, 7\}$, and similarly for B. The element of C in the top-left 2×2 block is not computed directly from full rows and columns of A and B; instead it is accumulated by multiplying the first tile of A with the first tile of B, then multiplying the second tile of A with the corresponding tile of B, and adding the two partial results together. That is, the four entries of the top-left tile of C are

$$\begin{aligned} C_0 &= a_0 b_0 + a_1 b_4 + a_2 b_8 + a_3 b_{12}, \\ C_1 &= a_0 b_1 + a_1 b_5 + a_2 b_9 + a_3 b_{13}, \\ C_4 &= a_4 b_0 + a_5 b_4 + a_6 b_8 + a_7 b_{12}, \\ C_5 &= a_4 b_1 + a_5 b_5 + a_6 b_9 + a_7 b_{13}. \end{aligned}$$

The point of writing it out this way is to notice that each of these four sums naturally splits into two halves of two terms each, and each half comes from multiplying a 2×2 tile of A by a 2×2 tile of B. The first half of every sum above (the terms involving b_0, b_1, b_4, b_5) comes from multiplying the first tile of A by the first tile of B; the second half (the terms involving b_8, b_9, b_{12}, b_{13}) comes from multiplying the second tile of A by the second tile of B. In other words, one tile of C is accumulated from a sum over several tile-by-tile products of A and B, as shown in Figure 4.9. This is the key

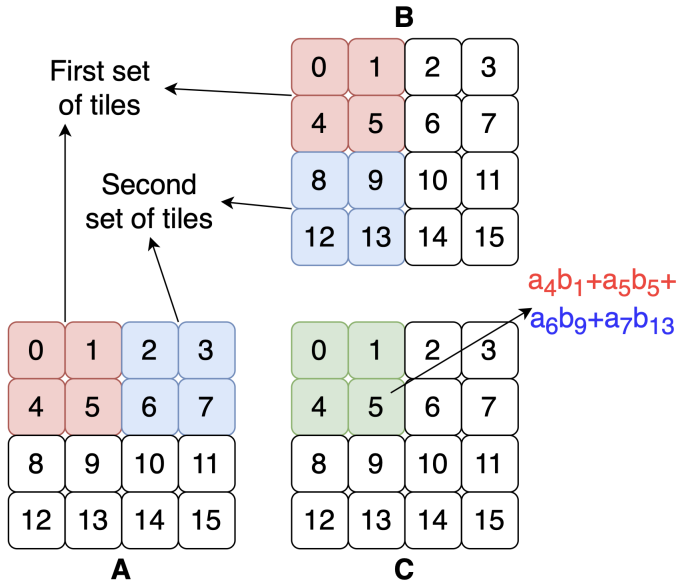


Figure 4.9: A 4×4 matrix multiply tiled into 2×2 blocks. A tile of C is accumulated from a few tile-by-tile products of A and B, rather than from full rows and columns.

structural insight that tiling exploits: once a tile of A and a tile of B are fetched to fast local memory, they can be reused for many elements in a tile of C, before either of them is evicted.

To generalize, for each of the $K \times K$ tiles of C, we fetch the K tiles of A along the corresponding row of tiles and the K tiles of B along the corresponding column of tiles, one pair at a time, into local scratchpad memory. We multiply the $S \times S$ tile of A by the $S \times S$ tile of B and accumulate the result into the tile of C we are building (Figure 4.10). This is repeated K times, once for each tile along the shared dimension, after which the tile of C is complete and can be written back out.

Let us redo the operation and memory access counts from before, now accounting for tiling, and see precisely why this restructuring helps. As before, let S be the tile size and $K = N/S$ the number of tiles along each dimension. There are $K \times K$ tiles in C, and producing each tile of C requires K separate multiplications of two $S \times S$ tiles. Each such tile multiplication costs S^3 multiply-add operations, so the total number of operations is

$$\#\text{ops} = K^2 \times S^3 \times K = N^3$$

which is exactly the same as before, as it must be: tiling changes how the computation is scheduled, not how much arithmetic work it fundamentally requires.

What does change is the number of memory accesses. For each of the $K \times K$ tiles of C, we fetch K tiles of A and K tiles of B, and each tile contains S^2 elements. Crucially, because the tile is small

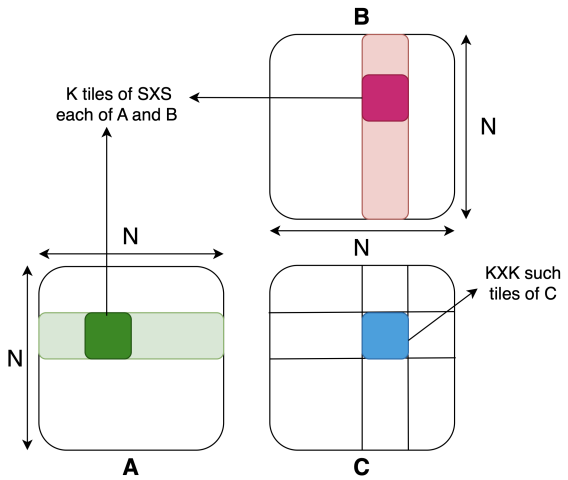


Figure 4.10: Matrix multiplication restructured using tiles

enough to fit in the local scratchpad, once it is fetched it is reused for every multiply-add that needs it, rather than being fetched again and again as in the case without tiling. The total number of memory accesses is therefore

$$\#\text{mem accesses} = K^2 \times 2 \times S^2 \times K = \frac{2N^3}{S}$$

Comparing this to the $2N^3$ memory accesses required without tiling, we see that tiling has reduced the number of memory accesses by exactly a factor of S, the tile size. Equivalently, the ratio of operations to bytes moved, has improved by a factor of S. This is the central quantitative result of tiling: every element of A and every element of B, once loaded into fast memory, is reused across S different output elements of C instead of being used once and discarded.

4.3.2 Choosing the Optimal Tile Size

The analysis above might suggest we should just make the tile size S as large as possible, since bigger tiles mean more reuse. In practice, S is constrained by hardware in two related but distinct ways, as described below.

Constraint 1: Cache Capacity

A tile of A, a tile of B, and the partial tile of C must all fit simultaneously in fast memory for reuse to work. Too large tiles no longer fit alongside everything else that needs to be resident — this causes thrashing, where tiles are repeatedly evicted and reloaded, destroying the reuse benefits that tiling was meant to provide. Too small tiles fit easily, but now there are many more of them: fixed

overhead per transfer grows relative to transfer size, and the reuse factor S in our formula shrinks. At the extreme $S = 1$, the tiled formula collapses to $2N^3$ memory accesses — exactly as in the case without tiling.

Constraint 2: SM Occupancy

As we saw in the previous chapter, shared memory isn't privately owned by one thread block — it is a fixed pool on each streaming multiprocessor (SM), typically a few hundred KB, that must be split among however many blocks run concurrently. A small tile uses less shared memory and helps increase SM occupancy, but gives less reuse per byte, sacrificing the point of tiling. A large tile can consume most of an SM's shared memory by itself, leaving room for only a small number of resident blocks, and too few concurrent warps to hide memory latency, resulting in low SM occupancy.

So, choosing a tile size depends on many factors: data reuse pushes for larger S, while cache capacity and SM occupancy constraints push back towards smaller tile sizes. The optimal tile size is large enough to provide good data reuse, but also small enough to ensure it fits in shared memory and doesn't hurt SM occupancy by much. In practice, auto-tuning systems search over a small set of candidate tile sizes, measure achieved throughput on the actual hardware, and pick the one that yields the best performance, since the exact crossover point depends on cache associativity and other low-level hardware details that are difficult to predict analytically. This is also why real-world high performance libraries such as cuBLAS, cuDNN, expose or internally search over multiple tile size configurations rather than committing to a single fixed value: there is no globally optimal tile size, only a tile size that is well matched to a particular combination of matrix shape, data type, and hardware memory hierarchy.

4.3.3 Tiling in CUDA Example 1: One Thread per Output Element Tiling

The discussion so far has described tiling only at a conceptual level. We will now understand how tiling is actually implemented in a CUDA kernel, and study examples of how the compute can be distributed across the many threads running on a GPU. We first discuss one of the simplest ways to implement tiled matrix multiplication in CUDA, where every single thread is responsible for computing exactly one element of the output matrix C, and shared memory is used as the fast local memory that holds the current tiles of A and B.

Suppose we launch a grid of thread blocks laid out to match the tile structure of C: if C is divided into a $K \times K$ grid of tiles, we launch a $K \times K$ grid of thread blocks, and each block is responsible for computing exactly one tile of C. Within a block, we launch an $S \times S$ arrangement of threads, one thread for every element of the tile. Every block loads and processes shared memory tiles of matrices A and B, to compute a tile of C. As shown in Figure 4.11, consider an example with $N = 12$ and a tile size of $S = 4$, so that $K = 3$. The grid then has $3 \times 3 = 9$ blocks, and each block has $4 \times 4 = 16$ threads. Block $(1, 1)$ owns the middle tile of C, covering rows 4 through 7 and columns 4 through 7, and within that block, thread $(1, 1)$ owns the single element $C[5][5]$.

Code 4.7: One Thread per Output Element Tiling

```
__global__ void matmul_tiled_element_kernel(const float* A, const float* B,
    float* C, int N, int TILE) {
    extern __shared__ float s[]; // first TILE*TILE for A, next for B
    float* sA = s;
    float* sB = s + TILE * TILE;
    int globalRow = blockIdx.y * blockDim.y + threadIdx.y;
    int globalCol = blockIdx.x * blockDim.x + threadIdx.x;
    int localRow = threadIdx.y;
    int localCol = threadIdx.x;
    float acc = 0.0f;
    int numTiles = (N + TILE - 1) / TILE;
    for (int t = 0; t < numTiles; ++t) {
        int aRow = globalRow;
        int aCol = t * TILE + localCol;
        int bRow = t * TILE + localRow;
        int bCol = globalCol;

        // Load A tile
        if (aRow < N && aCol < N)
            sA[localRow * TILE + localCol] = A[aRow * N + aCol];
            // sA[localRow][localCol] = A[aRow][aCol]
        else
            sA[localRow * TILE + localCol] = 0.0f;
            // sA[localRow][localCol] = 0.0f

        // Load B tile
        if (bRow < N && bCol < N)
            sB[localRow * TILE + localCol] = B[bRow * N + bCol];
            // sB[localRow][localCol] = A[bRow][bCol]
        else
            sB[localRow * TILE + localCol] = 0.0f;
            // sB[localRow][localCol] = 0.0f
        __syncthreads();

        int limit = TILE;
        for (int k = 0; k < limit; ++k) {
            acc += sA[localRow * TILE + k] * sB[k * TILE + localCol];
            // acc += sA[localRow][k] * sB[k][localCol]
        }
```

```
39
40          __syncthreads();
41      }
42
43      if (globalRow < N && globalCol < N)
44          C[globalRow * N + globalCol] = acc;
45          // C[globalRow][globalCol] = acc
46  }
```

In lines 3-4, two pointers, `sA` and `sB`, are carved out of a single block of dynamically allocated shared memory, one holding the current $S \times S$ tile of A and the other holding the current $S \times S$ tile of B. Every thread in the block shares these two arrays; they are the fast local memory that the whole tiling scheme is built around. As shown in line numbers 5-8 of the code snippet below, the kernel begins by computing four indices for every thread. These are the local row/column indices of the element the thread processes (i.e., where the thread sits inside its own block, and which elements of the shared memory tiles it will touch), and the corresponding global row/column indices (i.e., where that position falls in the global matrix). The variable `acc` in line 9 is a private, per-thread register that will accumulate the dot product for this thread's one output element as we sweep across tiles, and `numTiles`, equal to K in line 10, tells us how many such tiles we need to visit before the output element is complete. Notice that `acc` lives in a register and is never written to shared or global memory until the very end, so this partial sum is kept as close to the compute unit as possible throughout the entire computation.

With the indices established, the kernel loops over the K tiles that lie along the shared dimension in lines 11-41. On each iteration, the block as a whole needs to bring one new tile of A and one new tile of B into shared memory, and this loading step is itself done cooperatively. Rather than any one thread loading the whole tile, every thread loads exactly one element of the A tile and one element of the B tile, so that the sixteen threads in our example each fetch one entry, and together they fill the tile in a single coordinated step.

The indexing here is worth studying closely, because it is exactly what makes each thread load a different, non-overlapping element. A thread's row within the A tile it loads is always its own global row, but the column it loads shifts with the current tile index t, since as t advances we are sliding across to a new block of columns of A each time. The situation is mirrored for B: a thread's column within the tile it loads is always its own global column, while the row it loads shifts with t, since we are sliding down through successive blocks of rows of B. Concretely, `aRow` is pinned to `globalRow` while `aCol` advances with the tile index via `aCol = t * TILE + localCol`; symmetrically, `bCol` is pinned to `globalCol` while `bRow` advances via `bRow = t * TILE + localRow`. It is worth walking through this concretely for the thread we identified earlier. Thread $(1,1)$ in block $(1,1)$ has `globalRow` $= 1 \cdot 4 + 1 = 5$, `globalCol` $= 1 \cdot 4 + 1 = 5$, `localRow` $= 1$, and `localCol` $= 1$, matching its ownership

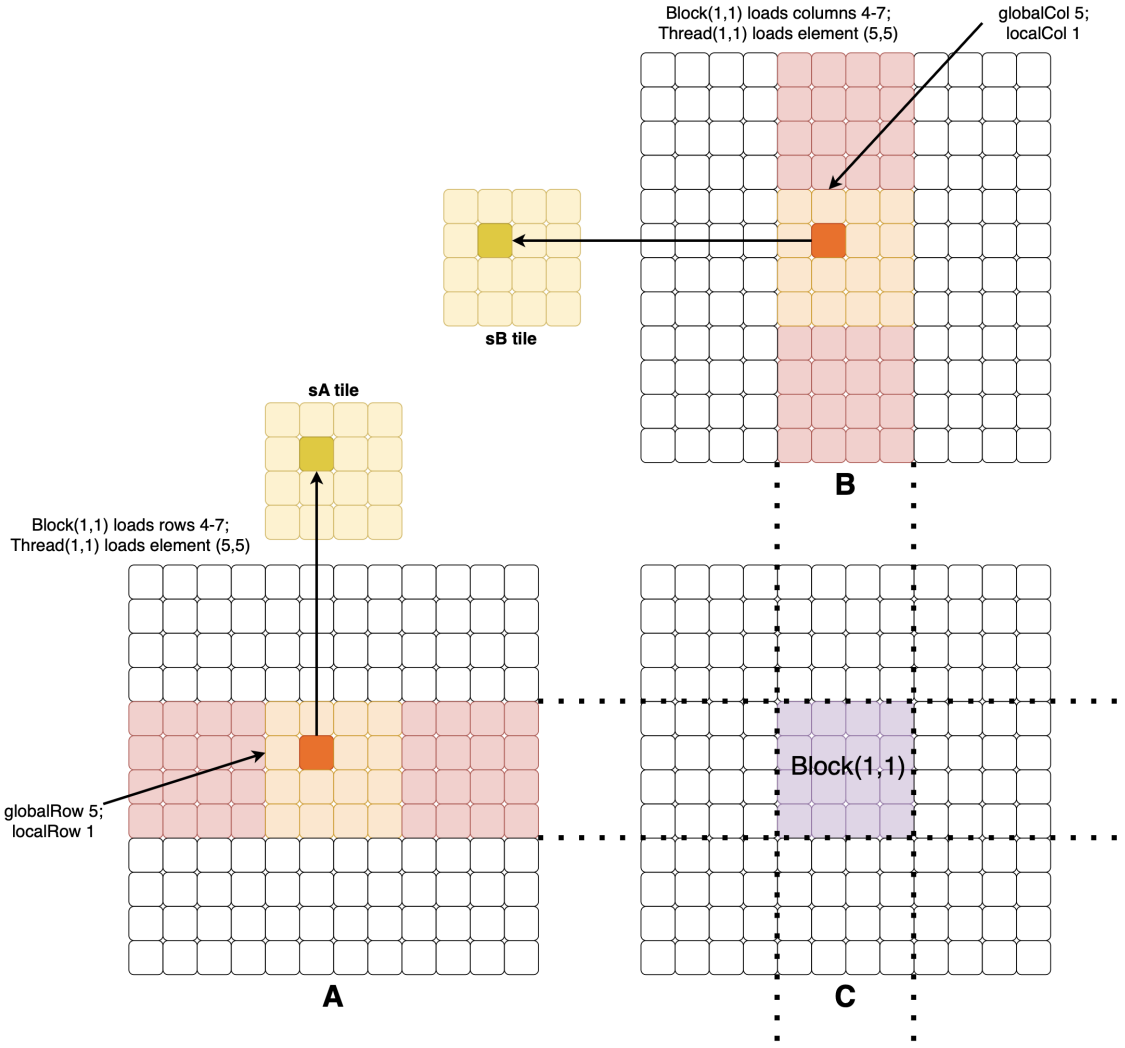


Figure 4.11: Tile and block structure for one element per thread tiling example

of C[5][5]. As t runs from 0 to 2, this thread loads A[5][1], then A[5][5], then A[5][9] into `sA`, while it loads B[1][5], then B[5][5], then B[9][5] into `sB`; notice the row of A and the column of B never change, only the sliding index does.

The conditional checks in lines 17-26, guarding each load, exist to handle the case where N is not an exact multiple of the tile size S. When this happens, the very last tile along a dimension is only partially filled with real data, and the remaining positions do not correspond to any actual element of A or B. Rather than reading out of bounds, which would be both incorrect and unsafe, the kernel simply writes a zero into that position of the shared tile.

Immediately after both tiles are loaded, every thread in the block calls `__syncthreads()`. This barrier is essential for correctness. A thread that finishes loading its own single element of `sA` and `sB` early has no guarantee that the other fifteen threads in the block have finished loading theirs, since threads within a block do not all execute at exactly the same rate and the hardware provides no ordering guarantee between them. If a fast thread were allowed to race ahead into the dot product step, it might read a tile entry that another, slower thread has not written yet, and the computation would silently use stale or garbage data. The barrier forces every thread to wait until the entire tile, contributed to by all sixteen threads together, is fully and correctly populated before any thread is allowed to read from it. Once the barrier has passed, every thread now has access to a complete, correct $S \times S$ tile of A and $S \times S$ tile of B sitting in fast shared memory, and it uses them to advance its own running dot product.

Thread `(localRow, localCol)` walks along row `localRow` of the shared A tile and down column `localCol` of the shared B tile, multiplying corresponding entries and adding them into `acc`. Every one of these S reads from `sA` and `sB` comes from shared memory rather than from HBM, so none of them pays the cost of a trip out to slow memory. Just as importantly, the same tile entries are being read by many different threads at once: the entry `sA[localRow][k]` is used by every thread in row `localRow` of the block, one for each value of `localCol`, and the entry `sB[k][localCol]` is used by every thread in column `localCol` of the block, one for each value of `localRow`. A single element fetched once from HBM into shared memory during the load phase is therefore reused S times during the compute phase, which aligns with what was derived earlier.

A second `__syncthreads()` in line 31, appears at the end of the loop body, after the dot product step and before the loop advances to the next tile. This barrier protects against the opposite race condition from the first one: on the next iteration, the block is about to overwrite `sA` and `sB` with the next pair of tiles, and some threads may still be in the middle of reading the current tiles during their dot product accumulation. Without this second barrier, a fast thread could begin overwriting shared memory with new tile data while a slower thread is still reading the old tile data it needs to finish its current partial sum, corrupting the computation. Together, the two barriers enforce a strict alternating pattern for the whole block: first, all loads complete before any compute begins; then, all compute completes before any new loads begin. This allows a single, small piece of shared memory to be safely reused K times over the course of the kernel, once for each tile along the shared dimension, without ever mixing data from two different tiles. After the loop

over all K tiles has finished, `acc` holds the complete dot product for this thread's output element. Every thread then performs exactly one write to global memory, placing its finished value into `C[globalRow][globalCol]`. Continuing the example, on each of the three tile iterations, thread $(1,1)$ in block $(1,1)$ computes `acc += sA[1][k] * sB[k][1]` for `k = 0..3`, so across all three tiles it accumulates exactly the 12-term dot product of row 5 of A with column 5 of B. It then performs its single write, `C[5][5] = acc`, which is the one output element this thread was assigned back at the start.

4.3.4 Tiling in CUDA Example 2: Column-Contiguous Tiling

The one-element-per-thread kernel gives every thread exactly one output value to compute, which is a very small amount of arithmetic to perform for the price of reading two operands out of shared memory, and it leaves the arithmetic units under-fed relative to what they are capable of. A natural next step is to give each thread more than one output element to compute, so that a single pair of values read from shared memory can be reused across several accumulations before the thread moves on. We now examine a kernel where each thread computes several contiguous elements of C running down a single column, rather than just one.

The key structural change to the code seen earlier is that a thread is no longer identified with a single output position; it is identified with a short contiguous run of output positions stacked vertically. If we let `E` denote how many such elements each thread is responsible for, then thread `(localRow, localCol)` inside a block no longer owns the single element at row `localRow`; it owns `E` consecutive rows, and `localRow` now indexes which group of rows the thread is responsible for rather than a single row directly.

Because each thread now covers `E` rows instead of one, a block of `TILE`×`TILE` threads collectively covers `TILE` $*$ `E` rows of C, while still covering only `TILE` columns, since the column assignment is unchanged. Note that the number of threads per block, `TILE`×`TILE`, has not changed at all; only the amount of C each of those threads is responsible for has grown. This asymmetry is important and is why the grid and shared memory tiles for A and B are no longer the same shape. As an example (Figure 4.12), take `N` = 12, `TILE` = 4, and `E` = 3. A block of $4 \times 4 = 16$ threads now covers $4 \times 3 = 12$ rows and 4 columns of C, so a single block along the row direction is enough to cover all of C's rows, and the grid needs only `blocks_y` = `N`/(`TILE` $*$ `E`) = 1 block in the row direction, while it still needs `blocks_x` = `N`/`TILE` = 3 blocks in the column direction.

Because the shape of the work has changed, the shared memory tiles must change shape to match. The tile of B is unaffected: it is still a plain `TILE` × `TILE` block, since B only participates through its columns, and the column assignment per thread has not changed. The tile of A, however, must now hold enough rows to supply every thread with all the rows it is responsible for, so it grows to (`TILE` $*$ `E`) × `TILE`: it still has `TILE` columns, since A is still sliced along the shared dimension in blocks of width `TILE`, but it now has `TILE` $*$ `E` rows to cover the taller row range that the block as a whole owns. This is the price of giving each thread more work: the amount of A that must be resident in shared memory at once grows in proportion to `E`, and this limits how large `E` gets.

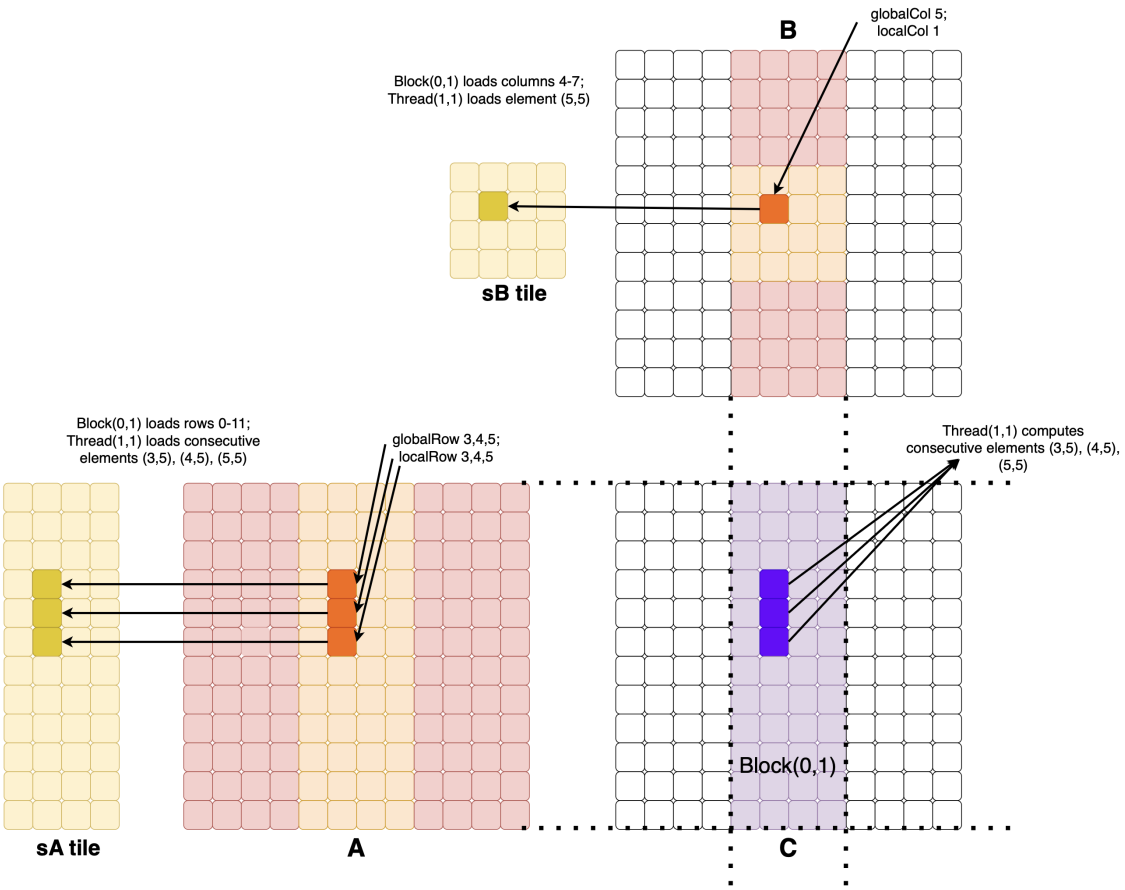


Figure 4.12: Tile and block structure for column-contiguous tiling example

Code 4.8: Column-Contiguous Tiling

```
__global__ void matmul_tiled_col_contiguous_kernel(const float* A, const float*
    B, float* C, int N, int TILE, int E) {
    extern __shared__ float s[];
    float* sA = s; // (TILE*E) x TILE
    float* sB = s + (size_t)TILE * (TILE * E); // TILE x TILE

    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;
    int localRow = threadIdx.y;  // 0 .. TILE-1
    int localCol = threadIdx.x;  // 0 .. TILE-1
    int baseGlobalRow = blockRow * (TILE * E);
    int globalCol = blockCol * TILE + localCol;
    int threadRowStart = baseGlobalRow + localRow * E;

    float acc[MAX_E];
    for (int e = 0; e < E; ++e) {
        acc[e] = 0.0f;
    }
    int numTiles = (N + TILE - 1) / TILE;
    int sA_row_stride = TILE;
    int sB_row_stride = TILE;

    for (int t = 0; t < numTiles; ++t) {
        // --- Load A tile ---
        int aColBase = t * TILE + localCol;
        for (int e = 0; e < E; ++e) {
            int gRow = threadRowStart + e;
            float aval = 0.0f;
            if (gRow < N && aColBase < N) {
                aval = A[gRow * N + aColBase];
                // aval = A[gRow][aColBase]
            }
            sA[(localRow * E + e) * sA_row_stride + localCol] = aval;
            // sA[localRow * E + e][localCol] = aval
        }

        // --- Load B tile ---
        int bRow = t * TILE + localRow;
        int bColBase = blockCol * TILE;
```

```
39          float bval = 0.0f;
40          if (bRow < N && (bColBase + localCol) < N) {
41              bval = B[bRow * N + (bColBase + localCol)];
42              // bval = B[bRow][bColBase + localCol]
43          }
44          sB[localRow * sB_row_stride + localCol] = bval;
45          // sB[localRow][localCol] = bval
46
47          __syncthreads();
48
49          // --- Compute partial sums ---
50          for (int k = 0; k < TILE; ++k) {
51              float b_k = sB[k * sB_row_stride + localCol];
52              // float b_k = sB[k][localCol]
53              for (int e = 0; e < E; ++e) {
54                  int sA_row = localRow * E + e;
55                  float a_k = sA[sA_row * sA_row_stride + k];
56                  // float a_k = sA[sA_row][k]
57                  acc[e] += a_k * b_k
58              }
59          }
60          __syncthreads();
61      }
62
63      // --- Write results back (contiguous rows) ---
64      for (int e = 0; e < E; ++e) {
65          int grow = threadRowStart + e;
66          if (grow < N && globalCol < N) {
67              C[grow * N + globalCol] = acc[e];
68              // C[grow][globalCol] = acc[e]
69          }
70      }
71  }
```

Lines 3 and 4 initialize the individual shared memory start addresses. In line 10, `baseGlobalRow` is the row at which the entire block's row range begins, playing the same role that `blockRow` × `TILE` played in the one-element kernel, except scaled up by `E` to reflect the taller region the block now covers. From this, `threadRowStart` in line 12 is the global row index of the first of this thread's `E` output elements. Every one of this thread's other elements is found simply by adding an offset e, running from 0 to $E - 1$, to `threadRowStart`. This is what "contiguous" means in the name of the

kernel: a thread's outputs are not scattered around the tile but form one unbroken run of rows in the same column. Just as in the one-element kernel, an accumulator is needed to hold the running dot product for each output element while we sweep across tiles along the shared dimension. The difference is that a single scalar register is no longer enough; the thread needs one accumulator per output element it owns, resulting in a small private array. It is worth noting that `sA_row_stride` and `sB_row_stride` in lines 19 and 20 are both equal to `TILE`, even though `sA` has $\texttt{TILE} \times \texttt{E}$ rows while `sB` has only `TILE` rows, because both tiles have `TILE` columns. Concretely, for thread $(1,1)$ of block $(0,1)$ from the earlier example ($\texttt{N} = 12$, $\texttt{TILE} = 4$, $\texttt{E} = 3$):

$$\begin{aligned} \texttt{blockRow} = 0, \quad \texttt{blockCol} = 1, \quad \texttt{localRow} = 1, \quad \texttt{localCol} &= 1 \\ \texttt{baseGlobalRow} = \texttt{blockRow} \times (\texttt{TILE} \times \texttt{E}) &= 0 \times 12 = 0 \\ \texttt{globalCol} = \texttt{blockCol} \times \texttt{TILE} + \texttt{localCol} &= 1 \times 4 + 1 = 5 \\ \texttt{threadRowStart} = \texttt{baseGlobalRow} + \texttt{localRow} \times \texttt{E} &= 0 + 1 \times 3 = 3 \end{aligned}$$

so this thread owns global column 5 and the three output elements $C[3][5]$, $C[4][5]$, $C[5][5]$, matching `acc[0]`, `acc[1]`, `acc[2]` respectively.

The array `acc` in line 14 is sized to a fixed compile-time maximum, `MAX_E`, and only the first `E` entries are actually used at run time. This is a common pattern in CUDA kernels where the true amount of per-thread work is a runtime parameter: since the compiler generally needs to know the size of a local array to place it in registers rather than spilling it to slow local memory, a small array with a small fixed maximum is declared, and the kernel only uses however much of it the runtime configuration calls for.

The loop over tiles along the shared dimension in lines 22-61 has exactly the same overall structure as before: for each tile index `t`, the block cooperatively loads the current tile of A, cooperatively loads the current tile of B, synchronizes, does some multiply-adds, and synchronizes again before moving to the next tile. What changes inside this loop is how much work is done at each of these steps, since a thread now handles `E` rows rather than one.

Loading the A tile now requires a small loop inside the main tile loop (lines 24-34), because each thread must fetch `E` elements of A rather than one. The column being loaded, `aColBase` in line 24, is the same for all of these elements and shifts with the tile index `t` exactly as before, since we are still sliding across the shared dimension one tile-width at a time. What differs is the row: for offset e, the thread loads global row `threadRowStart + e` in line 26, which is exactly the global row of the `e`-th output element this thread owns, and it stores that value into shared memory at local row $\texttt{localRow} \times \texttt{E} + \texttt{e}$. Since thread `localRow` owns rows $\texttt{localRow} \times \texttt{E}$ through $\texttt{localRow} \times \texttt{E} + \texttt{E} - 1$ within the block, its loads land at exactly those rows of `sA`, so that once every thread has finished this loop, the entire $(\texttt{TILE} * \texttt{E}) \times \texttt{TILE}$ tile of A has been filled in, one small contiguous vertical strip per thread. Loading the B tile needs no such inner loop, because a thread's column assignment did not change, and B's tile is still just $\texttt{TILE} \times \texttt{TILE}$.

To make this concrete, consider tile $t = 1$ for thread $(1,1)$ of block $(0,1)$. Here, `aColBase` =

$t \times$ `TILE` + `localCol` $= 4 + 1 = 5$, so the thread loads $A[3][5]$, $A[4][5]$, $A[5][5]$ for $e = 0, 1, 2$ (using `threadRowStart+e` for the row, which does not depend on `t`), and stores them into `sA` at local rows `localRow`$\times$`E`+`e` $= 3, 4, 5$, all at local column 1. For the B tile, `bRow` $= t \times$ `TILE`+`localRow` $= 4+1 = 5$ and `bColBase` $=$ `blockCol` $\times$ `TILE` $= 4$, so the thread loads the single value $B[5][5]$ and stores it into `sB` at local row 1, local column 1. The barrier after both loads at line 47 plays exactly the same role as before: it guarantees that every thread has finished writing its portion of `sA` and `sB` before any thread is allowed to read from either array during the compute phase that follows. The compute step is where the benefit of handling multiple contiguous rows per thread actually shows up.

Now in the compute step in lines 50-59, we need to take a dot product between a column of B tile with multiple rows of A tile. By reading `b_k` once at line 51, outside the inner loop over `e`, and reusing it `E` times inside that loop, the kernel amortizes the cost of that one shared memory read across several multiply-add operations rather than paying for it once per output element. The read of A inside the inner loop in lines 53-58, by contrast, does depend on `e`, since each output row needs its own corresponding row of A; `sA_row` recovers the correct local row of `sA` for output offset `e` using the same `localRow` $\times$ `E` + `e` indexing that was used when the tile was loaded, and `a_k` is read fresh for each `e`.

Each of the `E` accumulators is then updated with its own multiply-add, using the shared `b_k` and its own private `a_k`. Over the full inner loop, this single iteration of the outer k loop performs one shared memory read of B, `E` shared memory reads of A, and `E` multiply-adds, compared to a bare minimum of `E` reads of both A and B if each output element were handled by a fully independent one-element thread. The saving on B reads specifically is what motivates organizing the work this way rather than simply launching more independent threads.

The second `__syncthreads()` after the compute step at line 60 guards against the same hazard described in the one-element kernel: it prevents any thread from beginning to overwrite `sA` or `sB` with the next tile's data while another thread in the block is still reading the current tile's data during its own accumulation. Once all K tiles have been processed, every entry of `acc` holds the finished value for one of this thread's output elements, and the kernel writes all of them back to global memory in a final loop in lines 64-70. Each of these writes lands at global row `threadRowStart + e` and the thread's fixed global column, so the `E` writes performed by a single thread go to `E` consecutive rows of the same column of C.

The two kernels developed above, one thread per output element and one thread per contiguous run of elements down a column, are only two points along a much larger space of possible designs. A thread does not need to be restricted to a single row or a single column either: it is equally possible to have each thread compute a small two-dimensional block of output elements at once, which extracts reuse from both operands rather than from B alone as in the column-contiguous kernel above. Beyond changing how many elements a thread owns, there is a further layer of optimization that changes where operands live once they are needed for computation: rather than reading every operand directly out of shared memory on every use, a thread can first stage the handful of values it will reuse most into its own private registers, since a register read is cheaper than a shared memory

read in roughly the same way a shared memory read is cheaper than a trip to HBM. Production matrix multiplication kernels in libraries such as cuBLAS combine several of these ideas at once, along with additional hardware-specific tricks.

Practice Problem 4.4

Consider a multiplication of matrix A (of size $M \times K$) with B ($K \times N$) to produce C ($M \times N$). We will consider two ways of multiplying the matrices, one without tiling, and the other using tiling. For the case with tiling, we will consider tiles of sizes $m \times k$, $k \times n$, and $m \times n$ for A, B, C respectively. Note that the dimensions of the original matrices are represented by upper-case letters (M, N, K) and the dimensions of the corresponding tiles are lower case (m, n, k). Also note that the ratio of dimension of matrix to dimension of tile is a constant T. That is, $M/m = N/n = K/k = T$.

(i) How many multiply-add operations are required to compute C? Note that the answer will be the same with and without tiling. Count multiply-add as one operation, not two.

(A) $(M + N)K$

(B) $M \times N \times K$

(C) $M \times K + K \times N + M \times N$

(D) $2 \times M \times N \times K$

Solution: (B)
Each of the $M \times N$ elements of C requires a dot product over K terms, i.e., K multiply-add operations, giving $M \times N \times K$ total.

(ii) Consider matrix multiplication without tiling, where we fetch one row of A and one column of B into the GPU cache to compute an element of C. Assume cache is large enough, and matrices small enough, that all matrices A, B, and C fit into cache, and the rows/columns can be reused to compute many elements of C, once fetched. How many elements of A and B must be fetched into cache from HBM in total, to compute all elements of C? There is no need to count accesses of C to write C into HBM.

(A) $(M + N)K$

(B) $M \times N \times K$

(C) $M \times K + K \times N + M \times N$

(D) $2 \times M \times N \times K$

Solution: (A)
Since everything fits in cache and is fully reused, each of the $M \times K$ elements of A and each of the $K \times N$ elements of B is fetched from HBM exactly once, giving $MK + KN = (M + N)K$.

(iii) Consider matrix multiplication without tiling, where we fetch one row of A and one column of B into cache to compute an element of C. Assume cache is small enough to fit only one row and column of A/B. So a row/column fetched to compute one element of C cannot be reused to compute another element. How many elements of A and B must be fetched into cache from HBM in total, to compute all elements of C? You must count multiple fetches of an element multiple times, in case the fetched element cannot be reused again. There is no need to count accesses of C to write C into HBM.

(A) $(M + N)K$

(B) $M \times N \times K$

(C) $M \times K + K \times N + M \times N$

(D) $2 \times M \times N \times K$

Solution: (D)
For each of the $M \times N$ elements of C, we must re-fetch an entire row of A (K elements) and an entire column of B (K elements) since nothing is reused, giving $M \times N \times 2K = 2 \times M \times N \times K$ total fetches.

(iv) Suppose we multiply the matrices using tiling. How many tiles of A (of size $m \times k$ each) must be fetched into cache to compute one tile of C (of size $m \times n$). Recall that $M/m = N/n = K/k = T$.

(A) $m \times n$

(B) $m \times n \times k$

(C) T

(D) $T \times T$

Solution: (C)
Computing one tile of C requires accumulating over the full K dimension, which is covered by $K/k = T$ tiles of A (one for each position along K).

(v) Suppose we multiply the matrices using tiling. How many tiles of C (of size $m \times n$) must be computed to complete the entire matrix multiplication operation. Recall that $M/m = N/n = K/k = T$.

(A) $M \times N/T$

(B) $M \times N \times K/T$

(C) T

(D) $T \times T$

Solution: (D)
C is tiled into $M/m = T$ tiles along rows and $N/n = T$ tiles along columns, giving $T \times T$ total tiles to compute.

Programming Assignment 4.1: One Element per Thread Tiling in Matrix Multiplication

In this assignment, you will implement a shared-memory tiled CUDA kernel for matrix multiplication in which each thread computes a single output element as shown in §4.3.3. Rather than repeatedly re-reading rows and columns of the input matrices directly from global memory, the kernel has each thread block cooperatively stage small tiles of A and B into fast on-chip shared memory. You're given a `student.cu` file with the testing infrastructure and kernel launch wrapper `matmul_gpu()` already written — your job is to fill in the single kernel function `matmul_tiled_element_kernel`. The kernel launches one `TILExTILE` thread block per output tile, with each thread owning exactly one output element: it loops over tiles along the shared K dimension, cooperatively loads one element of A and one element of B per

thread into shared memory, synchronizes, accumulates its partial dot-product contribution into a private register, synchronizes again before moving to the next K-tile, and finally writes its accumulated result back to the output matrix in global memory once all tiles are processed. The assignment, along with a detailed `README`, is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/one_element_per_thread_tiling

Programming Assignment 4.2: Row-wise Tiling in Matrix Multiplication

In this assignment, you will implement a tiled CUDA kernel for matrix multiplication in which each thread computes several output elements along a single row, rather than just one, amortizing the cost of loading A over multiple output columns and improving arithmetic intensity. You're given a `student.cu` file with the testing infrastructure and kernel launch wrapper `matmul_gpu_row()` already written — your job is to fill in the single kernel function `matmul_tiled_row_kernel`. The kernel launches one thread block per $\texttt{TILE} \times (\texttt{TILE} \cdot \texttt{E})$ output tile, with each thread owning one row of that tile and `E` output columns spaced `TILE` apart: it loops over tiles along the shared K dimension, cooperatively loads a sub-block of A and `E` sub-blocks of B into shared memory, synchronizes, accumulates partial products into a private register array by reusing each loaded value of A across all of its output columns, synchronizes again before moving to the next K-tile, and finally writes its accumulated register values back to the output matrix in global memory once all tiles are processed. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/row_wise_tiling

Programming Assignment 4.3: Column-Contiguous Tiling in Matrix Multiplication

In this assignment, you will implement a CUDA kernel in which each thread computes several contiguous output elements down a single column, rather than a single output element, as shown in §4.3.4. You're given a `student.cu` file with the testing infrastructure and kernel launch wrapper `matmul_gpu_col_contiguous()` already written — your job is to fill in the kernel function `matmul_tiled_col_contiguous_kernel`. The kernel launches one $\texttt{TILE} \times \texttt{TILE}$ thread block per output region of `TILE` columns by $\texttt{TILE} * \texttt{E}$ rows, with each thread owning `E` contiguous rows within a single column: it loops over tiles along the shared K dimension, cooperatively loads a $(\texttt{TILE} * \texttt{E}) \times \texttt{TILE}$ sub-block of A and a $\texttt{TILE} \times \texttt{TILE}$ sub-block of B into shared memory, synchronizes, accumulates partial products into a private register array by reusing each loaded B value across all of its owned rows, synchronizes again before moving to the next K-tile, and finally writes its accumulated register values back to the output matrix in global memory once all tiles are processed. The assignment, along with a detailed `README`, is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/column_contiguous_tiling

Programming Assignment 4.4: 2D Tiling in Matrix Multiplication

In this assignment, you will implement a CUDA kernel, where rather than having each thread compute a single output element, it has each thread compute an entire $R \times R$ sub-block of the output matrix, maximizing data reuse from shared memory and improving arithmetic intensity. You're given a `student.cu` file with the testing infrastructure and kernel launch wrapper `matmul_gpu_block2d()` already written — your job is to fill in the single kernel function `matmul_tiled_block2d_kernel`. The kernel launches one thread block per `TILE`×`TILE` output tile, with each thread owning an $R \times R$ patch of that tile: it loops over tiles along the shared K dimension, cooperatively loads sub-blocks of A and B into shared memory, synchronizes, accumulates partial products into a private $R \times R$ register tile, synchronizes again before moving to the next K-tile, and finally writes its accumulated register block back to the output matrix in global memory once all tiles are processed. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/2d_tiling_matrix_multiplication

Programming Assignment 4.5: Register Tiling in Matrix Multiplication

In this assignment, you will implement a CUDA kernel, where rather than having each thread compute a single output element, the kernel has each thread compute several output elements along a single row, holding them in private registers across the entire K-loop and maximizing reuse of values loaded from shared memory. You're given a `student.cu` file with the testing infrastructure and kernel launch wrapper `matmul_gpu_registers()` already written — your job is to fill in the single kernel function `matmul_tiled_registers_kernel`. The kernel launches one thread block per `TILE` × (`TILE` · `E`) output tile, with each thread owning one output row and `E` columns of that row, strided `TILE` apart: it loops over tiles along the shared K dimension, cooperatively loads a sub-block of B into shared memory (while reading each needed value of A directly from global memory and reusing it across all of its assigned columns), synchronizes, accumulates partial products into a private register array, synchronizes again before moving to the next K-tile, and finally writes its accumulated register values back to the output matrix in global memory once all tiles are processed. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/register_tiling

4.4 Flash Attention

In the previous section, we saw that tiling lets us multiply two large matrices without paying the full cost of moving data between slow main memory and fast on-chip cache. The same idea of "fetch a small block into cache, do as much work as possible on it, and only then move to the next block" can also be used to optimize more complicated operations, like the attention mechanism that lies at the heart of the Transformer architecture.

As we saw in Chapter 2, attention is not a single matrix multiplication. It is a chain of three operations. First we compute a similarity score matrix between queries and keys, then we normalize each row of that matrix using the softmax function, and finally we use the normalized scores to take a weighted combination of value vectors. That is, given a query matrix Q, a key matrix K, and a value matrix V, attention computes:

$$S = QK^T, \qquad P = \text{softmax}(S), \qquad O = PV.$$

Here Q, K, V, and the output O all have shape $N \times d$, where N is the sequence length (the number of tokens) and d is the head dimension. The intermediate matrices S and P, however, have shape $N \times N$, because every query attends to every key. This quadratic scaling of attention values, and the resulting high memory access demands, is the central problem that FlashAttention [Dao et al., 2022] is designed to solve.

To see why this matters, consider an example. Suppose $N = 1024$ and $d = 128$, and each element of these matrices is stored using 2 bytes (float16). Then Q, K, V, and O each occupy $1024 \times 128 \times 2$ bytes, which is 256 kilobytes. This is small enough to plausibly fit in on-chip shared memory. But S and P each occupy $1024 \times 1024 \times 2$ bytes, which is 2 megabytes per attention head, per layer. As we saw before, shared memory on a modern GPU is typically only a few tens of megabytes shared across many concurrent thread blocks, so materializing S and P in full, for every layer, and every head, forces enormous traffic to and from HBM. This is a situation similar to what tiling is meant to address: the operands Q, K, V are individually small, but their pairwise interaction S is too large to keep resident, so we must be clever about how we compute with it. For simplicity, throughout this discussion we will ignore the causal mask used in autoregressive models; the tiling and I/O-reduction ideas we develop apply equally well once masking is added back in.

4.4.1 Standard Attention

To understand why FlashAttention is needed, we first look closely at how attention is normally implemented, and count how many times data must move between HBM and on-chip memory. A standard, unoptimized implementation of the forward pass, shown below, proceeds in three sequential steps, each of which reads its input from HBM and writes its output back to HBM. Notice that the full $N \times N$ matrix S is written out to HBM after step 1, then read back in step 2, and the full $N \times N$ matrix P is written out after step 2 and read back in step 3. This is wasteful because computing

softmax correctly requires an entire row of S to be available at once, since softmax normalizes by the sum over an entire row; this operation cannot easily be done on a subset of the matrix. This forces the algorithm to materialize the entire matrix in each step. This is a common pattern in machine learning workloads even beyond the specific case of attention: operations like ReLU or softmax are individually cheap to compute, but if their inputs and outputs must each be written to and read from HBM, the operation becomes memory bound, meaning that the time spent moving data dominates the time spent doing arithmetic.

Algorithm 4.1: Standard Attention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^T$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: **return** $\mathbf{O}$.

Memory Accesses in Standard Attention

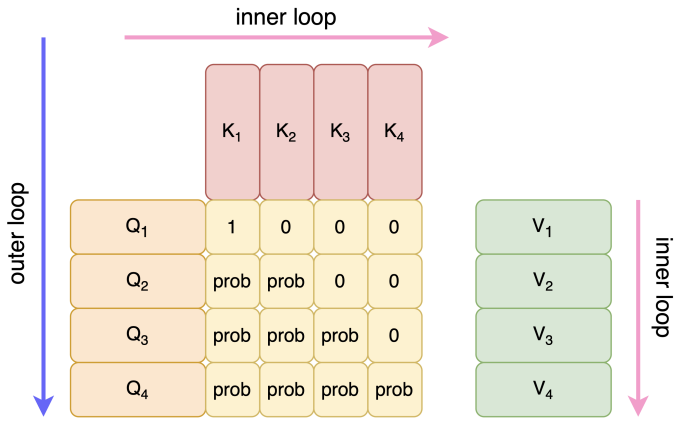


Figure 4.13: Standard attention implementation

Before counting memory accesses, note that the three-step algorithm above cannot be executed efficiently on real hardware once N is large: computing $S = QK^T$, $P = \text{softmax}(S)$, and $O = PV$ all at once would require holding the full $N \times N$ matrices S and P in on-chip memory, which grows quadratically in N and quickly exceeds the capacity of SRAM. In practice, the computation is therefore carried out incrementally, one query vector Q_i at a time, with each row of S, P, and O computed, written to HBM, and discarded before moving to the next row. The algorithm below

makes this row-by-row execution explicit, which lets us tally exactly how many times data must travel between HBM and on-chip memory.

As shown in the Figure 4.13 above and algorithm below, the outer loop iterates over the N rows of Q, one query vector Q_i at a time, where each Q_i has size d. For each Q_i, the inner loop walks over N vectors of K to compute the score row S_i and the softmax row P_i, and a second inner loop walks over all N vectors of V to accumulate the output row O_i. Each pass over K or V inside the inner loop costs $O(Nd)$ memory accesses, because there are N vectors each of size d. Since this inner loop is repeated once for every one of the N rows of Q in the outer loop, the total number of memory accesses is $N \times O(Nd) = O(N^2 d)$. This quadratic dependence on the sequence length N is the fundamental bottleneck in scaling attention. As N grows, for example when processing longer documents or longer contexts, the memory traffic required by standard attention grows quadratically, even though the useful information contained in Q, K, and V only grows linearly with N.

Algorithm 4.2: Memory Accesses in Standard Attention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

for $i = 1$ to N **do** ▷ outer loop over rows of Q
  Load $Q_i \in \mathbb{R}^d$ from HBM ▷ $O(d)$ access
  for $j = 1$ to N **do** ▷ inner loop over rows of K
    Load $K_j \in \mathbb{R}^d$ from HBM ▷ $O(d)$ access
    Compute $S_{ij} = Q_i \cdot K_j^T$
  end for ▷ total for this pass: $O(Nd)$ memory accesses
  Write row $S_i \in \mathbb{R}^N$ to HBM
  Read row S_i from HBM
  Compute $P_i = \text{softmax}(S_i)$
  Write row $P_i \in \mathbb{R}^N$ to HBM
  Read row P_i from HBM
  $O_i \leftarrow 0 \in \mathbb{R}^d$
  for $j = 1$ to N **do** ▷ inner loop over rows of V
    Load $V_j \in \mathbb{R}^d$ from HBM ▷ $O(d)$ access
    $O_i \leftarrow O_i + P_{ij} V_j$
  end for ▷ total for this pass: $O(Nd)$ memory accesses
  Write row O_i to HBM
end for
return O

4.4.2 The Softmax Obstacle

The previous section showed that tiling lets us compute a large matrix multiplication in blocks, loading only small tiles of the operands into on-chip memory at a time, and this idea applies just as well to the two matrix multiplications inside attention: $S = QK^T$ and $O = PV$ can each be broken into blocks and computed incrementally without ever materializing the full $N \times N$ matrix. If attention were nothing more than these two matrix multiplications, tiling alone is enough to reduce the number of global memory accesses. The obstacle is the step sandwiched between them: the softmax that turns S into P. Unlike a matrix multiplication, where each output tile can be computed independently from the corresponding input tiles, softmax normalizes by the sum over an entire row, so computing even one entry of P_{ij} correctly requires access to every entry in row S_i, not just the tile currently in on-chip memory. So, how can we process a row of $\mathbf{S}$ one block at a time if softmax needs the whole row at once? The resolution is a technique called online softmax, which restructures the softmax computation so that it can be updated incrementally as each new block of columns arrives, producing running estimates that are corrected on the fly and converge to the exact softmax once the entire row has been seen.

To keep the idea simple for a moment, ignore the exponential that a real softmax applies, and suppose we simply want to normalize three raw values X, Y, Z that belong to the same row. The normalized value coresponding to Y would then be:

$$P(Y) = \frac{Y}{X + Y + Z}.$$

Suppose at some earlier point in a streaming computation we have only seen X and Y, but not yet Z. At that point, the best estimate we could compute is

$$P(Y) = \frac{Y}{X + Y}$$

which is wrong because it is missing the contribution of Z in the denominator. Once Z becomes available, we cannot simply patch the future values; we must also correct the past softmax estimates, replacing the old partial sum $X + Y$ with the new, complete sum $X + Y + Z$. In other words, every previously computed softmax value is only provisional, and must be rescaled once more of the row becomes visible. In our example, once Z becomes available, we can scale the old value of $P(Y)$ as:

$$P(Y) = P^{\text{old}}(Y)\frac{X + Y}{X + Y + Z}$$

This matters directly for attention, because in a tiled computation we see the columns of K (equivalently, the entries of a row of S) in blocks, not all at once. If row Q_i has already been compared against columns K_1 through K_j, the entries $S_{i1}, \ldots, S_{ij}$ are known, but the entries corresponding to future columns $K_{j+1}, \ldots, K_N$ are not. How do we correctly compute softmax in such cases of tiled computation?

Formally, define the row-wise normalization constant

$$L_i = \sum_j e^{Q_i K_j^T}$$

and the corresponding attention output:

$$O_i = \sum_j P_{ij} V_j = \frac{\sum_j e^{Q_i K_j^T} V_j}{L_i}$$

Both L_i and O_i are sums over the entire row, so at the moment we only have a partial block of columns, neither quantity is final. The trick, known as online softmax, is to maintain running estimates of L_i and O_i and to update them incrementally as each new block of columns arrives. For a new block of key and value vectors K_j, V_j, the running normalization constant is updated as:

$$L_i^{\text{new}} = L_i^{\text{old}} + e^{Q_i K_j^T},$$

and the running output is updated as

$$O_i^{\text{new}} = \frac{O_i^{\text{old}} L_i^{\text{old}} + e^{Q_i K_j^T} V_j}{L_i^{\text{new}}}.$$

The term $O_i^{\text{old}} L_i^{\text{old}}$ undoes the previous normalization so that the old accumulated numerator is exposed again, the new block's contribution $e^{Q_i K_j^T} V_j$ is added to it, and the whole sum is divided by the updated, larger normalization constant L_i^{new}. In this way, O_i is correct after every block, in the sense that it always equals the softmax-weighted average of all value vectors seen so far, even though the full row has not yet been seen.

Tracking a Running Maximum

There is one more complication that we cannot skip over. In practice, softmax is never computed by directly exponentiating raw scores, because the scores $Q_i K_j^T$ can be large, and $e^{Q_i K_j^T}$ can overflow floating point representation. The standard fix is to subtract the maximum value in the row before exponentiating, which does not change the mathematical result because it cancels out in the ratio, but keeps every exponentiated value bounded between 0 and 1. With this correction, softmax is written as

$$P_{ij} = \frac{e^{Q_i K_j^T - M_i}}{L_i}, \qquad \text{where} \qquad M_i = \max_j \left(Q_i K_j^T \right), \qquad L_i = \sum_j e^{Q_i K_j^T - M_i}.$$

To compute online softmax with this change, for a new block K_j, the running maximum is updated as

$$M_i^{\text{new}} = \max \left(M_i^{\text{old}}, Q_i K_j^T \right).$$

Whenever the running maximum changes, every quantity computed relative to the old maximum must be rescaled by the factor $e^{M_i^{\text{old}} - M_i^{\text{new}}}$ to convert an exponential taken relative to the old maximum into one taken relative to the new maximum. The normalization constant and the output are therefore updated together as

$$L_i^{\text{new}} = L_i^{\text{old}}\, e^{M_i^{\text{old}} - M_i^{\text{new}}} + e^{Q_i K_j^T - M_i^{\text{new}}},$$

$$O_i^{\text{new}} = \frac{O_i^{\text{old}} L_i^{\text{old}}\, e^{M_i^{\text{old}} - M_i^{\text{new}}} + e^{Q_i K_j^T - M_i^{\text{new}}} V_j}{L_i^{\text{new}}}.$$

The only extra cost of this numerically stable version is that we must keep two small running vectors, L and M, one scalar per query row, alongside the accumulating output O. Since L and M each have only N entries, compared to the $N \times N$ size of the full attention matrix, this extra storage is negligible.

4.4.3 The FlashAttention Algorithm

We now have all the ingredients needed to describe FlashAttention itself: tiling, to keep working blocks small enough to fit in on-chip cache, and online softmax with a running maximum, to correctly and incrementally normalize attention weights one tile at a time. FlashAttention combines these two ideas into a single kernel that never writes the full $N \times N$ matrices S or P back to HBM (Figure 4.14).

The computation is organized as a nested loop over tiles, an outer loop over blocks of columns of K and rows of V, and an inner loop over blocks of rows of Q. Intuitively, the outer loop picks a chunk of keys and values and copies it into on-chip cache once. Then, holding that chunk of K and V fixed in cache, the inner loop sweeps through all the query rows, computes the partial attention contribution of this chunk of keys to every query row, and immediately merges that partial contribution into the running output for each query, using the online softmax update we saw above. Once the inner loop has visited every query row, the outer loop advances to the next chunk of K and V, and the whole process repeats until every column has been visited by every row.

Written out formally, with block sizes B_r for rows of Q and B_c for columns of K and V, the algorithm is as follows. We will walk through this algorithm line by line, since every step plays a specific role. Line 1 chooses the tile sizes. The column block size B_c and row block size B_r are chosen so that a tile of K or V, and a tile of Q, together with the working space needed to compute a $B_r \times B_c$ score block, all fit comfortably inside the on-chip SRAM of size M. Because B_r is capped at d, tiles of Q end up being at most $d \times d$ in size, which is far smaller than the full $N \times d$ matrix.

Algorithm 4.3: FlashAttention Forward Pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \left\lceil \frac{M}{4d} \right\rceil$, $B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.
2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ into $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_1, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_1, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} \left(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j\right)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: **return** $\mathbf{O}$.

Line 2 allocates the output accumulator O, initialized to all zeros, together with two auxiliary vectors: ℓ, which will hold the running normalization constant L_i for each row, initialized to zero since nothing has been summed yet, and m, which will hold the running row maximum M_i, initialized to $-\infty$ so that the very first score seen for any row immediately becomes the current maximum.

Lines 3 and 4 partition every matrix into tiles along the sequence dimension: Q is split into T_r row blocks, and K and V are each split into T_c row blocks. The output O and the auxiliary vectors ℓ and m are split into matching row blocks so that each block of Q has a corresponding block of accumulator state.

Line 5 begins the outer loop, iterating over the T_c blocks of K and V, as shown in Figure 4.14. This is the loop that determines which chunk of keys and values is currently resident in fast memory. Line 6 performs the key data movement of the outer loop: it copies the current block K_j and V_j from HBM into on-chip SRAM exactly once per outer iteration. Because this copy happens outside the inner loop, it is reused by every row block of Q during the inner loop, rather than being re-fetched for every query row. Line 7 begins the inner loop, iterating over the T_r row blocks of Q.

Line 8 loads the current block Q_i, along with the corresponding slices of the accumulator state

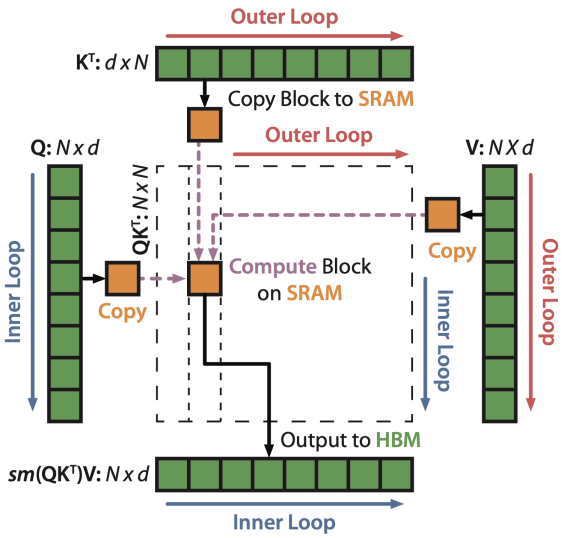


Figure 4.14: FlashAttention (Image credit: [Dao et al., 2022])

O_i, ℓ_i, and m_i, from HBM into on-chip SRAM (Figure 4.14). This is the state that will be updated based on how the current key and value tile interacts with this block of queries. Line 9 performs the local score computation: it multiplies the small tile Q_i by the small tile K_j^T to produce a small $B_r \times B_c$ block of the score matrix, entirely on chip. This block, called S_{ij}, is only a tiny fraction of the full $N \times N$ matrix, and it is never written to HBM.

Line 10 computes, still entirely on chip, the row maximum $\tilde{m}_{ij}$ of this local score block, the locally exponentiated and shifted probabilities $\tilde{P}_{ij}$, and the local row sums $\tilde{\ell}_{ij}$ of those probabilities. These are exactly the quantities $\tilde{m}$, $\tilde{P}$, and $\tilde{\ell}$ that appear in the online softmax update derived earlier, except restricted to this one tile's worth of columns rather than the full row. Line 11 merges the newly computed local statistics with the running statistics carried over from previous outer loop iterations, using the online softmax updated: the running maximum m_i^{new} becomes the larger of the old running maximum and the new local maximum, and the running normalization constant ℓ_i^{new} is updated by rescaling the old sum to the new maximum and adding in the rescaled local sum.

Line 12 updates the output accumulator. The old accumulated output, un-normalized by multiplying back through ℓ_i and rescaled to the new maximum, is added to the new tile's contribution $\tilde{P}_{ij}V_j$, also rescaled to the new maximum, and the sum is divided through by the new normalization constant ℓ_i^{new}. The result is written back to HBM. Line 13 writes the updated running statistics ℓ_i and m_i back to HBM, so that they are available when the next outer-loop iteration (the next block of keys and values) revisits this same block of queries. Line 16 returns the completed output O, which, after all T_c outer iterations have contributed their tiles, is exactly equal to the softmax attention output

that standard attention would have computed, but without ever materializing the $N \times N$ matrices S or P in HBM.

Memory Accesses in FlashAttention

Having described the algorithm, we now redo the memory access counting exercise, in order to see quantify the savings. In FlashAttention, tiles of K and V are sized so that a tile occupies roughly M elements, where M is the size of the on-chip SRAM. Tiles of Q, because B_r is capped at d, are of size roughly $d \times d$. The outer loop, which iterates over blocks of K and V, therefore runs approximately Nd/M times, since the total size of K (or V) is $N \times d$ and each outer iteration consumes a tile of size M. Each outer iteration requires the algorithm to sweep through all T_r blocks of Q in the inner loop, and every pass through the full set of Q blocks costs $O(Nd)$ memory accesses, since all of Q has size $N \times d$ and must be read once per outer iteration. Multiplying the cost of one outer iteration by the number of outer iterations gives the total memory access cost of FlashAttention,

$$\frac{Nd}{M} \times O(Nd) = O\left(\frac{N^2 d^2}{M}\right).$$

Compare this to the $O(N^2 d)$ cost of standard attention derived earlier. The ratio of the two costs is $O\left(\frac{d}{M}\right)$, and since the on-chip SRAM size M is, in practice, much larger than the head dimension d (that is, $d^2 \ll M$, or equivalently $d \ll M$), this ratio is much smaller than one. In other words, FlashAttention reduces the number of expensive HBM accesses by a substantial factor compared to standard attention, even though both algorithms perform exactly the same number of floating point operations and produce numerically identical results.

4.4.4 The Backward Pass, and Later Versions of FlashAttention

We now describe other ideas in FlashAttention. Training an attention-based model requires a backward pass as well: given the gradient of the loss with respect to the output, dO, we must produce dQ, dK, and dV, using the intermediate activation values S and P, which are not materialized into HBM by FlashAttention. The fix is recomputation — rather than storing S and P from the forward pass, the backward pass simply recomputes each small tile of them on chip, from the same Q, K, V tiles that must be loaded into SRAM anyway to perform the backward matrix multiplications. This costs extra floating point operations but no extra HBM traffic, since those tiles were going to be loaded regardless.

To make this recomputation exact, FlashAttention saves a small amount of extra information per query row from the forward pass, rather than the full S and P matrices. This is enough to reconstruct what's needed for the backward pass without redoing the full softmax computation from scratch, and without materializing S or P in HBM. The extra information is $O(N)$ in size, negligible compared to the $O(N^2)$ cost of storing S or P. The backward pass is then tiled with the same outer loop over K and V, and inner loop over Q, as in the forward pass. The gradients dK and dV are accumulated in SRAM across the inner loop and written once per outer iteration, while dQ is accumulated in

HBM across outer iterations. Because standard attention's backward pass is memory bound rather than compute bound, this recompute-instead-of-store strategy ends up being faster in practice than the naive backward pass, despite doing more arithmetic, for the same reason the forward pass was faster: it eliminates the dominant cost, which was never the FLOPs but the HBM traffic.

After the success of the initial version of FlashAttention, the authors made further improvements to the FlashAttention kernels. FlashAttention-2 [Dao, 2023] leaves the tiling and online-softmax ideas untouched and instead fixes how the original kernel's work is scheduled on the GPU, which limited the first version in reaching only 25 to 40 percent of peak throughput despite already solving the HBM traffic problem. The FlashAttention-2 algorithm defers the division by the running normalization constant ℓ_i, which the original version performed at every inner-loop iteration, to a single division after the loop completes, cutting down on element-wise arithmetic that competes with matrix multiplication for the GPU's compute units. The kernel also swaps the loop order, so the outer loop now runs over query blocks and the inner loop over key/value blocks, whereas the original kernel ran the outer loop over K/V and the inner loop over Q. Because each query block's output can now be computed independently of every other, this restructuring lets the kernel parallelize across the sequence-length dimension as well, not just across batch and attention heads, launching separate thread blocks for separate blocks of queries. This parallelization keeps the GPU occupied even when processing a single long sequence with a small batch size, a regime where the original kernel left most streaming multiprocessors idle. Within a thread block, work is split across warps by query block rather than by key and value block, so each warp owns an independent slice of the output and no longer needs to exchange partial sums with other warps through shared memory. Together, these changes bring FlashAttention-2 to 50 to 73 percent of peak throughput on an A100, with an unchanged numerical result.

FlashAttention-3 [Shah et al., 2024] targets the Hopper GPU generation, whose asynchronous hardware features are not fully exploited by earlier versions of FlashAttention, resulting in the algorithm reaching only about 35 percent of Hopper's peak throughput. As we saw in the previous chapter, Hopper allows data movement and matrix multiplication to be issued asynchronously, so FlashAttention-3 assigns separate warps permanently to either fetching tiles from HBM or computing on tiles already in SRAM, letting the two proceed simultaneously rather than in lockstep. It further overlaps the matrix multiplications within a tile with the softmax computation of a neighboring tile, since softmax's exponentials and divisions would otherwise leave the tensor cores idle while they run. Finally, it exploits Hopper's low-precision FP8 tensor cores, using a separate quantization scale per tile and a random rotation of the query and key tiles before quantizing, which spreads out outlier values so a single large entry does not degrade the precision of an entire tile. Together these changes give FlashAttention-3 roughly 1.5 to 2 times the throughput of FlashAttention-2 on H100 GPUs.

Practice Problem 4.5

Consider the attention operation of a single head and a single layer in a transformer model. The input sequence has length N and each attention head has dimension d. The query, key, value and output matrices

Q, K, V, O are each stored with one byte per element. The unnormalized score matrix is S, and the row-wise softmax of S is P (again one byte per element). In each of these parts, you are required to calculate the minimum cache size and also the memory traffic to and from the DRAM in bytes. There is just a single level of user-programmable cache available above DRAM, which is initially empty. All matrices/rows/tiles needed for the computation must be loaded into this cache, and they need not be reloaded from DRAM as long as they remain resident in the cache; only the first load (and final write if modified) counts towards DRAM traffic. Ignore causal window throughout this question.

(i) Suppose N and d are small enough that the cache can simultaneously hold all six matrices Q, K, V, S, P, and O needed for the attention computation. Write an expression for the minimum capacity C (in bytes) required to achieve this.

Solution: The six matrices and their sizes (in bytes) are: $Q, K, V, O \in \mathbb{R}^{N \times d}, S, P \in \mathbb{R}^{N \times N}$, and since each element takes one byte, the size in bytes equals the number of elements. To hold all six simultaneously, we simply sum their sizes: $C = 4Nd + 2N^2$.

(ii) If the cache is sized according to the previous part, determine the minimum number of bytes read from and written to the DRAM for the attention computation. Assume that the intermediate scores S and softmax outputs P are not required for any computation later, and hence need not be materialized into DRAM.

Solution: Since the entire cache from part (i) is large enough to hold everything, S and P are computed and kept entirely in cache, and are never written out to or read back from DRAM. Therefore, only Q, K, V need to be read from DRAM once each, and O is written once, giving a total DRAM traffic of $4Nd$.

(iii) Now suppose N is large enough that Q, K, V, S, P, and O do not simultaneously fit in cache. Consider the standard attention algorithm that iterates over each row $i = 1, \ldots, N$ of Q in the outer loop. In each iteration it loads row $Q[i, :]$, loads all of K, computes the score row $S[i, :] = Q[i, :]K^\top$, computes the softmax row $P[i, :] = \text{softmax}(S[i, :])$, loads all of V, accumulates the output row $O[i, :] = P[i, :]V$, and writes $O[i, :]$ back to DRAM. State the minimum cache capacity required per iteration of the outer loop, to keep all of these matrices and rows in the cache.

Solution: Within a single iteration i, the cache must simultaneously hold $Q[i, :], K, S[i, :], P[i, :], V, O[i, :]$; note that $Q[i, :]$, $S[i, :]$, $P[i, :]$, and $O[i, :]$ are each single rows of length d or N, while K and V are the full $N \times d$ matrices. Summing the sizes of these six terms, the minimum cache capacity per iteration is $C = 2d + 2N + 2Nd$ bytes.

(iv) If the cache is sized according to the previous part, derive the total DRAM I/O over all N iterations of the outer loop. Assume that in the user programmable cache, K and V are pinned and can be reused across iterations. Also, assume that attention scores and softmax outputs don't need to be materialized back into DRAM.

Solution: Since K and V are pinned in cache, they can be loaded once (each of size Nd) and reused across all N iterations, rather than being re-fetched every iteration. The row $Q[i, :]$, however, is different in each iteration and so must be read fresh from DRAM in iteration i, contributing a total of Nd bytes summed over all N rows. $S[i, :]$ and $P[i, :]$ are computed entirely in cache each iteration and never written to DRAM, so they contribute nothing. $O[i, :]$ is written to DRAM at the end of each iteration, again contributing Nd bytes in total. Therefore, the total DRAM traffic is Nd for each of K, V, Q, O, totalling to $4Nd$.

(v) Now, let us say we use the Flash attention algorithm with tiling of Q, K, V, O into blocks of B rows each, giving $T = N/B$ blocks per matrix. We have an outer loop which iterates over the T blocks of K and V,

while the inner loop iterates over the T blocks of Q and O. Each block of K and V is loaded once per outer iteration (i.e. T times total). The score tile $S_{\text{tile}} \in \mathbb{R}^{B\times B}$ is computed and consumed entirely within each inner iteration and is never written to DRAM (softmax updates happen in place on S_{tile}). Although m and ℓ are of size $\mathbb{R}^N$, only the values for the current B rows are kept in cache during the inner loop, and must be read and written to DRAM during each inner iteration. Derive the cache capacity required to hold all the data needed per each inner iteration.

Solution: Within one inner iteration, the cache must hold the four $B\times d$ blocks Q-block, K-block, V-block, O-block (each of size Bd), the length-B running statistics m and ℓ, and the $B \times B$ score tile S_{tile}. Summing these sizes, we get $C = 4Bd + B^2 + 2B$ bytes.

(vi) If the cache is sized according to the previous part, derive the total DRAM I/O (including both reads and writes) for the attention implementation described earlier, over the entire computation. *Hint: The vectors O, l, m must be read and then written back after each update in every inner iteration. For simplicity, assume they are also read during the first iteration.*

Solution: Each outer iteration fixes one block of K and V (each of size Bd) and loads them once per outer iteration; since there are T outer iterations, this amounts to $2 \times T \times Bd = 2Nd$ bytes of reads in total. Each Q block, however, must be reloaded fresh in every inner iteration nested within every outer iteration, giving $T \times T \times Bd = N^2d/B$ bytes of reads. Similarly, O must be read and updated in every inner iteration, contributing N^2d/B bytes of reads and N^2d/B bytes of writes. The running statistics m and ℓ (each of size B) are loaded from and written to DRAM for every one of the $T \times T$ inner iterations, which leads to $4NT$ bytes total of reads and writes (counting over all the outer iterations). (S_{tile} is never materialized to DRAM, so it contributes nothing.) Summing all these contributions, the total DRAM I/O is: $2Nd + \frac{3N^2d}{B} + \frac{4N^2}{B}$ bytes.

(vii) Consider the second version of Flash attention, which reverses the loop order relative to the previous case: the outer loop now iterates over the $T = N/B$ blocks of Q (and O), while the inner loop iterates over all T blocks of K and V. The score tile $S_{\text{tile}} \in \mathbb{R}^{B\times B}$ is computed and consumed entirely within each inner iteration and is never written to DRAM (softmax updates happen in place on S_{tile}). The running statistics m and ℓ are now maintained entirely in the cache across the full outer loop for a fixed Q block, so they are never written to or read from the DRAM. Derive the cache capacity required to keep all these matrices for each inner iteration.

Solution: As in the previous version, within one inner iteration the cache must simultaneously hold the four $B\times d$ blocks Q-block, K-block, V-block, O-block, along with m, ℓ, and S_{tile}, even though the reuse pattern across iterations differs from part (v). Hence, the capacity expression is unchanged: $C = 4Bd + B^2 + 2B$ bytes.

(viii) If the cache is sized according to the previous part, derive the total DRAM I/O for the previous part for the entire attention computation.

Solution: Each outer iteration fixes one Q block and one O block, which are read and written (respectively) exactly once per outer iteration rather than once per inner iteration as before; summed over all T outer iterations, this gives a total of $2Nd$ bytes (Q is read and O is written). Each inner iteration, however, loads a fresh K block and V block since these change every inner step; over all T outer and T inner iterations, this amounts to $2N^2d/B$ bytes of reads. Since m, ℓ, and S_{tile} are all kept resident in cache across the inner loop for a fixed Q block (as stated in the problem), they are never read from or written to DRAM, contributing nothing. Total DRAM I/O is $2Nd + \frac{2N^2d}{B}$ bytes. Note that the improvement over part (vi) comes from two separate sources: reusing Q and O across the inner loop instead of re-fetching them every inner

iteration, and additionally avoiding the $4NT$ bytes of traffic on m and ℓ entirely, since the loop reordering lets these running statistics stay resident in cache for the whole inner loop rather than being written and re-read at every step.

Programming Assignment 4.6: Flash Attention

In this assignment, you will implement the CUDA kernel for FlashAttention. Rather than materializing the full N×N attention matrix, the kernel processes Q, K, and V in tiles loaded into shared memory, computing attention scores tile-by-tile and maintaining running softmax statistics (a per-row running maximum m and normalization factor ℓ). You're given a mostly-complete `student.cu` file with the launcher, CPU reference implementation, and test harness already written — your job is to fill in the single kernel function `flash_forward_kernel`. The kernel launches one thread block per (batch, head) pair, with each thread owning one query row: it loops over key/value tiles, loads them into shared memory, computes the tile's attention scores, updates the running max/normalization using the online softmax update rule, accumulates a rescaled partial output, and writes the final max, normalization factor, and output back to global memory once all tiles are processed. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/flash_attention

4.5 Model Optimizations

In the preceding sections, we looked at how a fixed neural network can be made to run faster and more efficiently on a given piece of hardware, through techniques such as batching, kernel fusion, and tiling. In every one of those techniques, the model itself was left completely untouched: the same weights, the same number of parameters, and the same mathematical functions were being computed, just with a smarter execution strategy. This section turns to a different, and in some ways more powerful lever: changing the model itself so that it needs less memory, less computation, and less data movement in the first place.

A model with hundreds of billions of parameters, stored even in a compact numerical format, can occupy hundreds of gigabytes of memory, and evaluating it for a single input can require trillions of floating point operations. Hardware-level optimizations can only take us so far against numbers of this size. If we instead ask whether the model can be made structurally smaller, or represented more compactly, without giving up too much of its accuracy, we open a second, complementary axis of efficiency, which we discuss in this section.

4.5.1 Pruning

The starting observation behind pruning is that not every parameter in a trained neural network contributes equally to its output. If we inspect the weight matrices of a trained model, we will typically find that many of the individual weights have very small magnitude. A weight that is extremely close to zero contributes almost nothing when it is multiplied against an activation and summed into the next layer, so removing it, that is setting it to exactly zero, is unlikely to change the model's predictions by very much. Pruning is the general name for techniques that identify such low-importance weights and eliminate them, with the goal of reducing the total number of parameters that need to be stored and computed.

The simplest and most direct method of doing this is called magnitude pruning, in which one looks at the absolute value of each weight and replaces those whose magnitude falls below some threshold with a zero value. But because most modern hardware cannot get a speedup from a "zero" value unless it is told explicitly not to compute it, realizing performance gains from pruning is not straightforward — one either needs access to hardware optimized for "sparse" computations, or pruning must lead to a significant reduction in the dimensions of the tensors themselves. Further, pruning weights can lead to a loss in model accuracy, which is why it is typically followed by a fine-tuning phase, in which the remaining weights are updated to compensate for the loss of the pruned ones, thereby restoring accuracy. Typically, we iterate over pruning and fine-tuning multiple times, gradually increasing the fraction of weights removed while monitoring accuracy. This iterative process can also eventually lead to smaller tensor dimensions, and hence, better performance even on accelerator hardware optimized for dense operations. Pruning can be organized along a few different axes, depending on which parts of the model are removed and how the decision of what to remove is made.

Structured versus Unstructured Pruning

Structured pruning removes entire structural units of the model, such as individual neurons, convolutional filters, channels, attention heads, or even whole layers, rather than removing individual weights in isolation. Because entire rows, columns, or blocks disappear, the resulting weight matrices are smaller in a way that ordinary dense matrix multiplication libraries can immediately take advantage of. In other words, structured pruning preserves the overall dense structure of the computation, just at a reduced size, which makes it comparatively easy to realize a real speedup on existing hardware that has been optimized for dense linear algebra.

Unstructured pruning, in contrast, removes individual weights wherever they happen to have small magnitude, without regard to the row, column, or block they belong to (Figure 4.15). This tends to preserve accuracy better for a given amount of compression, since the pruning decision is made at the finest possible granularity, but it leaves behind a weight matrix that is sparse in an irregular pattern. Realizing an actual speedup from such an irregular sparsity pattern requires specialized sparse matrix representations and sparse matrix multiplication kernels, or even hardware that is optimized for sparse matrix operations (e.g., sparse Tensor Cores on NVIDIA's A100 GPU). On software and hardware that is not optimized to exploit sparsity, an unstructured pruned model may not run any faster at all, even though it has fewer non-zero parameters, because all matrix positions are used in every computation.

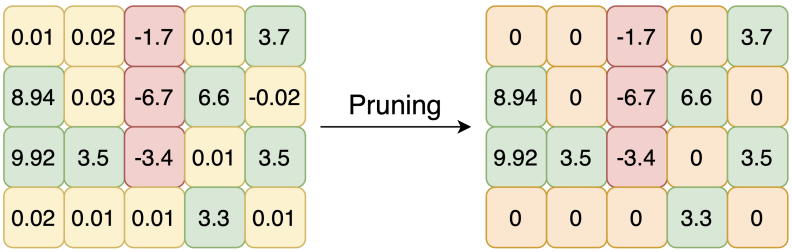


Figure 4.15: Unstructured pruning

Dynamic Pruning

A further refinement is dynamic pruning, in which the decision about which weights or structures to skip is not fixed after training, but is instead made at inference time based on the actual input being processed. For example, one might look at the magnitude of the activations produced by a given input and decide that certain weights connected to very small activations can be skipped for this particular input, even though these same weights might matter for a different input. Dynamic pruning trades a fixed, input-independent sparsity pattern for one that adapts to the data, which can improve accuracy for a given compression ratio, at the cost of additional runtime logic to make and act on these decisions.

4.5.2 Knowledge Distillation

Knowledge distillation approaches the compression problem from an entirely different angle. Rather than removing parameters from an existing large model, it trains a brand new, smaller model, called the student, to reproduce the behaviour of a larger, already trained model, called the teacher. The intuition is that a large model, once trained, has learned a rich internal representation of the problem, and that this knowledge can be transferred into a much smaller network if the smaller network is taught to mimic the larger one's outputs, rather than being trained from scratch purely on the original training data.

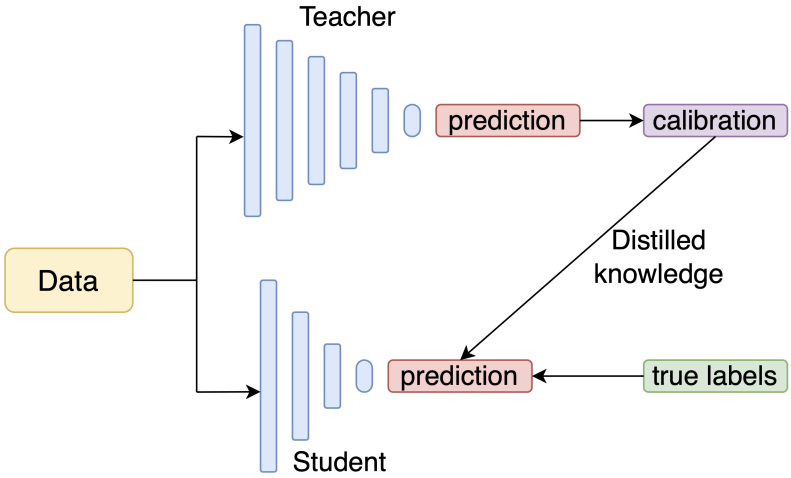


Figure 4.16: Knowledge distillation

They key idea that makes distillation effective is the use of soft labels rather than hard labels. In an ordinary classification setup, a model is trained against hard labels, meaning a one-hot vector indicating the single correct class is used to compute the loss function. A trained teacher model, however, produces a full probability distribution over all possible classes, and the relative probabilities it assigns to the incorrect classes carry useful information. For instance, an image of a cat might be classified correctly, but the teacher's output distribution might assign a small but non-negligible probability to "dog" and a vanishingly small probability to "airplane". This pattern reflects a notion of similarity between classes that a one-hot label simply cannot express. To make this soft information easier for the student to learn from, the teacher's output logits are passed through a softmax function with an elevated temperature parameter, denoted by T, which flattens the probability distribution and makes the relative magnitudes of the smaller probabilities more pronounced. Using a temperature of $T = 1$ recovers ordinary, sharply peaked softmax probabilities, sometimes called the hard predictions, whereas a temperature $T > 1$ produces the softer distribution used for distillation.

During training, the student model is optimized against two objectives at once (Figure 4.16). The first is a distillation loss, which measures the difference, typically using something like a KL-divergence or cross-entropy, between the softened output distribution of the teacher and the softened output distribution produced by the student for the same input. The second is a conventional student loss, which measures the difference between the student's own hard predictions and the actual ground-truth label. The final training objective is a weighted combination of these two losses, allowing the student to benefit both from the teacher's rich, generalization-aware knowledge and from the ground truth itself. Because the student network can have far fewer layers and far fewer parameters than the teacher, this process can produce a model that is significantly smaller and faster to run, while retaining much of the accuracy of the original large model.

4.5.3 Low-Rank Matrix Factorization

A third way to shrink a model is to change how its weight matrices are represented mathematically, rather than removing individual weights outright. Many of the weight matrices inside a neural network, especially the large, dense matrices found in fully connected layers, are not "full rank" in a meaningful sense: much of the information they contain can be captured using a lower-dimensional approximation. Low-rank matrix factorization exploits this property by decomposing a large matrix into a product of two, much smaller matrices.

Formally, suppose we have a weight matrix W of dimension $M \times N$. Ordinarily, storing and multiplying by this matrix costs $O(MN)$ in both memory and computation. Low-rank factorization instead approximates W as the product of two smaller matrices,

$$W \approx L_K \times R_K^T,$$

where L_K has dimension $M \times K$ and R_K^T has dimension $K \times N$, for some rank K that is much smaller than either M or N (Figure 4.17). Storing and multiplying these two smaller matrices costs $O(MK + KN)$ instead of $O(MN)$, which can be a substantial saving when K is small relative to M and N.

The standard mathematical tool for finding the best possible low-rank approximation, in a least-squares sense, is the singular value decomposition, or SVD. Given a matrix A, its SVD expresses A as a product of three matrices ($A = U\Sigma V^T$) capturing an orthogonal basis for its rows (U), a set of scaling factors called singular values (Σ), and an orthogonal basis for its columns (V). The singular values are ordered from largest to smallest, and they indicate how much of the matrix's structure each corresponding basis direction is responsible for. By keeping only the top K singular values and their associated basis vectors, and discarding the rest, one obtains the best possible rank-K approximation of the original matrix. This retained top-K decomposition is exactly the L_K and R_K^T pair described above.

The main practical challenge in applying low-rank factorization is choosing an appropriate value of K: too small a value and the approximation loses too much information, degrading accuracy; too large a value and little compression is achieved. In practice, K is often chosen empirically, by

measuring the accuracy of the resulting model on a validation set for several candidate values and picking the smallest k that keeps accuracy within an acceptable tolerance. Low-rank factorization is particularly well suited to the dense weight matrices found in fully connected and projection layers, and the same underlying idea generalizes beyond two-dimensional matrices to multi-dimensional tensors, using tensor decomposition methods that are the higher-order analogue of the matrix SVD. As we will see later in this section, this idea of representing a large matrix as a product of two much smaller ones, reappears as the foundation of low-rank adaptation for efficient fine-tuning.

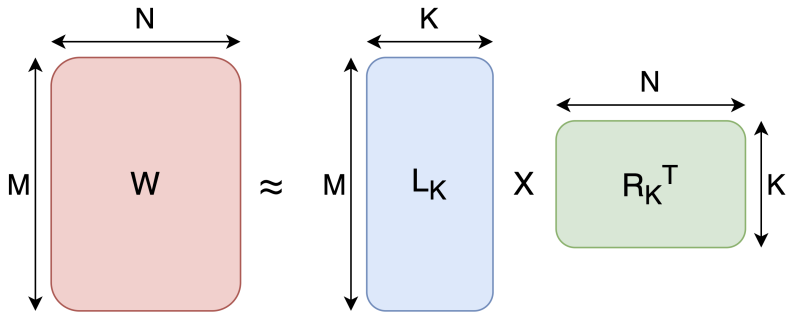


Figure 4.17: Low-Rank matrix factorization

4.5.4 Quantization

Every number stored and manipulated inside a neural network, whether it is a weight, an activation, or a gradient, has to be represented using some numerical format, and the choice of this format is itself a significant lever to control memory footprint and computational cost. Quantization is the family of techniques that reduces the numerical precision used to represent these values, trading off some numerical accuracy in exchange for substantially lower memory usage, lower power consumption, and faster arithmetic.

Numerical Formats

Neural network training has traditionally used 32-bit floating point, commonly called FP32 (Table 4.1) or single precision, which allocates one bit for the sign, eight bits for the exponent, and twenty-three bits for the mantissa, or fractional part. This format offers a wide dynamic range and fine-grained precision, but at the cost of four bytes of storage per value and comparatively expensive arithmetic. A range of lower-precision alternatives have been developed to reduce this cost. Sixteen-bit floating point, or FP16, halves the storage requirement by using a five-bit exponent and a ten-bit mantissa, which gives it a narrower dynamic range than FP32 but is often sufficient for inference and, with care, for training. An alternative sixteen-bit format called BFloat16, or BF16 (developed by Google), instead keeps the same eight-bit exponent as FP32, sacrificing mantissa bits, seven of them, to

preserve dynamic range at the expense of fine-grained precision; this makes BF16 more robust to the very large or very small values that can occur during training, even though it represents nearby values less precisely than FP16. Even more aggressive formats such as eight-bit floating point, FP8, an intermediate format called TF32 that is used internally by some hardware accelerators (e.g., TPUs), and eight-bit integer, INT8, push the precision reduction further still, at a corresponding increase in the risk of numerical error. As a general rule, lower precision formats are favoured for inference and for deployment on resource-constrained edge devices, where memory and power are at a premium, while higher precision formats continue to be preferred, at least for parts of the computation, during training, where the accumulation of small numerical errors over many iterations can otherwise destabilize learning.

Format	Total Bits	Sign	Exponent	Mantissa	Typical Use
FP32 (Single Precision)	32	1	8	23	Training (high precision)
FP16 (Half Precision)	16	1	5	10	Inference / mixed-precision training
BF16 (BFloat16)	16	1	8	7	Training (robust to large/small values)
TF32	19	1	8	10	Internal accelerator computation
FP8	8	1	4/5*	3/2*	Aggressive training/inference
INT8	8	—	—	—	Edge / resource-constrained inference

* FP8 is commonly implemented in two variants: E4M3 (4 exponent, 3 mantissa bits) and E5M2 (5 exponent, 2 mantissa bits), trading off precision versus dynamic range.

Table 4.1: Common numerical formats

Quantization Strategies

There are two broad strategies for introducing quantization into a model's lifecycle. The first, post-training quantization, trains the model as usual using full FP32 precision, and only afterward converts the trained weights, and often the activations as well, down to a lower precision format such as FP16 or INT8 for inference. This is the simpler of the two approaches, since it requires no changes to the training procedure, but because the model was never exposed to the effects of reduced precision during training, it can sometimes suffer a noticeable drop in accuracy, particularly at very aggressive precisions such as INT8 or lower. The second strategy, quantization-aware training, instead simulates the effect of reduced precision throughout the training process itself, so that the model's parameters are learned in a way that is already robust to the rounding and range limitations that quantization introduces. Quantization-aware training generally preserves accuracy better than post-training quantization for a given target precision, at the cost of a more involved training pipeline.

A related technique is mixed-precision training, in which different parts of the training process use different numeric precisions rather than quantizing the entire model to a single low-precision format. Most computations in the forward and backward passes, such as matrix multiplications, are performed using lower-precision formats like FP16 or BF16 to improve throughput and reduce

memory usage, while quantities that are particularly sensitive to numerical error, such as the master copy of the weights, gradient accumulation, and optimizer state, are maintained in FP32. This allows training to achieve nearly the same accuracy as full-precision training while substantially reducing computational and memory costs.

Irrespective of the quantization strategy, converting a set of floating point values into a lower-precision format requires a calibration step: because the lower-precision format has a much smaller range of representable values than FP32, the original values must be scaled to fit within that smaller range before they are rounded. This calibration is typically done by examining the actual distribution of values, whether weights or activations, and choosing a suitable clipping range: values outside this range are clamped, and values inside it are mapped, through a scale factor, onto the discrete set of levels that the target format can represent. This clipping range can be computed once for the entire model, referred to as global quantization, or it can be computed separately for each layer, or even for each individual channel within a layer, referred to as layer-wise or channel-wise quantization; finer-grained calibration generally yields better accuracy because it accounts for the fact that different layers, or different channels, can have very different natural ranges of values.

Quantization can also be applied selectively: to just the weights, to just the activations, or to both simultaneously. Interestingly, activation statistics can themselves be used as a signal for deciding which weights are safe to quantize aggressively and which are not, since a weight that is consistently multiplied by a large activation on the output may need to be preserved with higher precision. Finally, quantization can be performed statically, using a fixed scale and clipping range determined ahead of time from a representative calibration dataset, or dynamically, where the scale and range are recomputed at runtime based on the actual data seen during inference; dynamic quantization can adapt better to inputs whose statistics differ from the calibration set, at the cost of some additional runtime computation to determine the scale factors on the fly.

4.5.5 Mixture-of-Experts

The techniques discussed so far — pruning, distillation, low-rank factorization, and quantization — all aim to make a model's parameters occupy less space or require less computation, while the model still, in principle, uses all of its parameters for every input. Mixture-of-experts takes a fundamentally different approach: it increases the total number of parameters in the model, often very substantially, while ensuring that only a small fraction of those parameters is actually activated for any single input. Therefore, the goal is to achieve a large effective model capacity at a fraction of the inference cost of a dense model of the same total size.

In a standard transformer layer, the dense feed-forward network (FFN) that follows the self-attention block is applied identically to every token. In a mixture-of-experts layer (Figure 4.18), this single dense FFN is replaced by a collection of several smaller FFNs, referred to as experts, for example FFN1, FFN2, FFN3, and FFN4. A small auxiliary neural network, called the router, or gating network, examines each token's representation and decides which expert, or which small subset of experts, that particular token should be sent to. The router typically outputs a probability

distribution over the available experts, and the token is routed to the expert, or group of experts, that has received the highest probability. This probability also used to weight the expert's output before it is combined back into the model. Different tokens in a batch can be routed to different experts as they pass through the same layer, thereby activating a completely different set of parameters.

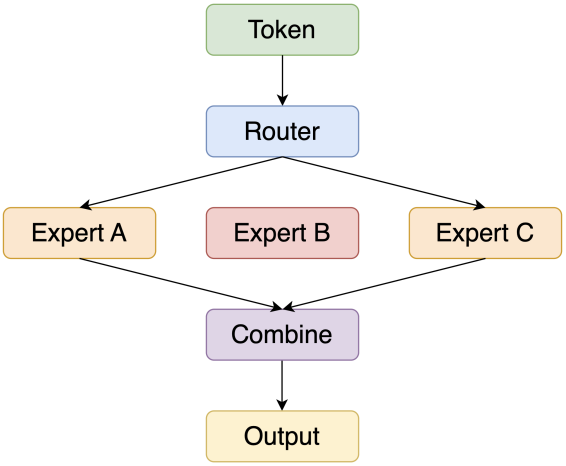


Figure 4.18: Mixture of experts

Because only a small number of experts are evaluated for each token, rather than the full set, the amount of computation performed per token stays comparable to that of a much smaller dense model, even though the total number of parameters stored across all experts can be much larger. This lets the model use a much larger total capacity, since there are simply more parameters available across the collection of experts, without inflating the actual inference cost per token, which depends only on the parameters actually used in computation. The trade-off is a set of new systems challenges: the routing decision introduces a data-dependent, and therefore harder to statically schedule, computation pattern. Different experts may end up handling very different numbers of tokens, leading to load imbalance across the hardware devices that host them. Moreover, because different experts are often distributed across multiple accelerators, routing tokens to their assigned experts can require significant communication between devices. A substantial amount of systems research is devoted to balancing the load across experts and minimizing this communication overhead.

4.5.6 Efficient Attention and Efficient Transformers

The self-attention mechanism at the heart of the transformer architecture computes, for a sequence of length N, an $N \times N$ matrix of pairwise attention scores between every pair of tokens. This gives the transformer its expressive power, but it also means that both the compute and the memory required by attention grow quadratically with sequence length. As models are asked to process ever

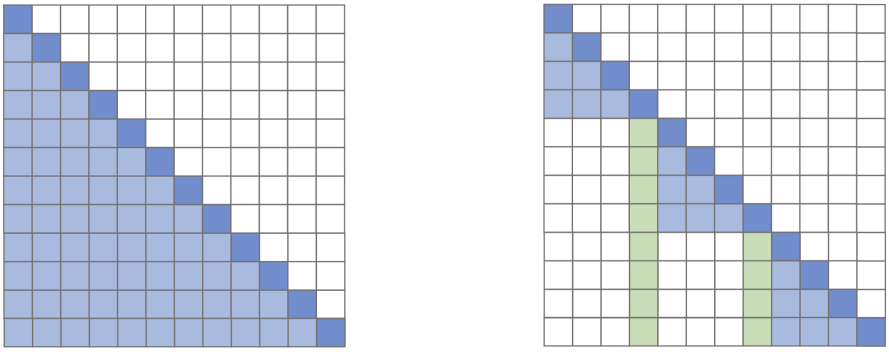
longer sequences, whether long documents, long conversations, or high-resolution images broken into many patches, this quadratic cost becomes a serious bottleneck. Therefore, current research has also proposed changing the attention algorithm itself to avoid this quadratic blow-up, in a way that is complementary to the purely hardware-level attention optimizations, such as FlashAttention, discussed in the previous section. Those techniques make the exact quadratic computation faster and more memory-efficient on given hardware, whereas the techniques in this section change what is being computed so that the underlying algorithmic complexity itself is reduced.

These approaches can be organized into a small number of design patters, which are often combined with one another in a model. One pattern is recurrence, in which the sequence is broken into segments and information is carried forward from one segment to the next through a compact recurrent state, avoiding the need to attend across the full, unsegmented sequence at once; Transformer-XL [Dai et al., 2019] is a well known example of this approach. A second pattern is memory and downsampling, in which the model first compresses the input sequence into a much smaller set of learned memory or summary representations and then performs attention over this compressed representation instead of the full sequence. By reducing the effective sequence length, both the computational and memory cost of attention are lowered. Compressive Transformer [Rae et al., 2019] extends this idea by storing compressed representations of past activations, while Nyströmformer [Xiong et al., 2021] approximates the full attention matrix using a small set of landmark tokens. A third pattern relies on structured sparse attention patterns, in which each token is only allowed to attend to a carefully chosen subset of other tokens, for example nearby tokens within a local window, or tokens at fixed strided intervals, rather than to every other token in the sequence; Longformer [Beltagy et al., 2020] and BigBird [Zaheer et al., 2021] are examples that combine local windows with a small number of globally attended tokens. A fourth pattern uses learnable attention patterns, where the model itself learns, often through a clustering or routing mechanism, which tokens should attend to which, rather than fixing the sparsity pattern in advance. Routing Transformer [Roy et al., 2021] is a representative example, clustering similar tokens so that attention is computed primarily within each cluster instead of across the entire sequence. A fifth pattern, broadly called low-rank and kernel-based methods, avoids materializing the full $N \times N$ attention matrix altogether: low-rank methods project the key and value matrices onto a lower-dimensional subspace before attending, while kernel-based methods reformulate the similarity function so that the query-key comparison can be computed implicitly, without ever forming the full attention matrix. Both reduce complexity from quadratic to linear in sequence length. Linformer [Wang et al., 2020] and Performer [Choromanski et al., 2022] are representative examples.

An illustrative example of the sparse attention family is the Sparse Transformer [Child et al., 2019]. In an ordinary transformer, every attention head attends to all preceding tokens, producing a dense, lower-triangular pattern of non-zero attention weights when a causal mask is applied, since each token can only see tokens that come before it. The Sparse Transformer instead splits the attention heads into two groups: half of the heads perform local attention, meaning each token only attends to a fixed-size window of nearby preceding tokens, while the other half perform strided attention,

meaning each token attends only to tokens at fixed periodic intervals further back in the sequence (Figure 4.19). Combining these two patterns across the two groups of heads allows information to still propagate across the full length of the sequence, through a combination of short-range local hops and longer-range strided hops, while the attention matrix that any individual head has to compute is now sparse rather than fully dense. Because the resulting attention matrix has a fixed, known sparsity pattern, it can be computed efficiently using sparsity-aware attention algorithms, or using hardware tensor cores that have native support for structured sparsity.

It is also worth mentioning that alongside these attention-pattern changes, a simpler and now widely adopted family of efficiency techniques reduce the size of the key and value projections themselves rather than the attention pattern. One example is multi-query attention (MQA), in which all attention heads share a single set of key and value projections while retaining separate query projections. Its variant is grouped-query attention (GQA), in which heads are divided into a small number of groups that each share a key and value projection. Both approaches substantially shrink the memory needed to cache keys and values during autoregressive generation, which as we saw previously, is often the dominant cost when generating long sequences.



(a) Transformer (b) Sparse Transformer

Figure 4.19: Illustration of patterns of the attention matrix for dense self-attention in Transformers and sparse fixed attention in Sparse Transformers. Image credit: [Tay et al., 2022]

4.5.7 Parameter-Efficient Fine-Tuning

The final technique in this chapter addresses a somewhat different problem from the ones discussed so far. Rather than compressing a model that will be deployed as-is, parameter-efficient fine-tuning addresses the cost of adapting an already large, pretrained model to a new, specific task or domain. Once a large language model has been pretrained, organizations frequently want to customize it, for example to specialize it for legal documents, medical records, or a particular company's internal data. The obvious way to do this is to continue training all of the model's parameters on the new,

task-specific data, but for a model with tens or hundreds of billions of parameters, this is extremely expensive: it requires storing gradients and optimizer state for every single parameter, and it produces an entirely new, equally large copy of the model for every task the model is adapted to.

Low-Rank Adaptation, commonly abbreviated LoRA [Hu et al., 2021] offers a much cheaper alternative. The insight behind LoRA is that the update to a weight matrix needed to adapt a pretrained model to a new task tends to have a low "intrinsic rank", meaning it can be well approximated by a low-rank matrix, even though the original pretrained weight matrix itself is full rank. Concretely, consider a pretrained weight matrix W_0 that would ordinarily be updated during fine-tuning to some new matrix $W = W_0 + \Delta W$. Instead of directly updating every entry of $W_0 \in \mathbb{R}^{d\times k}$, LoRA freezes W_0 entirely, leaving it completely untouched, and introduces a separate, much smaller pair of trainable matrices, $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times k}$, where the rank r is chosen to be much smaller than either dimension of the original weight matrix ($r \ll d, k$). The update is then represented as the low-rank product of these two new matrices, scaled by a constant α/r,

$$W = W_0 + \frac{\alpha}{r}AB$$

and only A and B are trained during fine-tuning, while W_0 remains fixed at its pretrained value. Because r is small, the number of trainable parameters introduced by A and B together is a tiny fraction of the number of parameters in W_0 itself, often under 1% of the original model's size. This means that fine-tuning with LoRA requires only a small amount of additional memory for gradients and optimizer state, since those only need to be tracked for A and B, and it produces a compact set of "adaptor" matrices for each new task, rather than an entirely new full-size copy of the model. Multiple LoRA adaptors, one per task, can even be swapped in and out on top of the same frozen base model.

QLoRA [Dettmers et al., 2023] extends this idea one step further by combining it with the quantization techniques discussed earlier in this section. In QLoRA, the frozen pretrained weight matrix W_0 is stored and used in a heavily quantized, low-precision format, significantly reducing the memory needed just to hold the base model in memory during fine-tuning, while the small trainable LoRA matrices A and B are still updated in higher precision to preserve the accuracy of the learned adaptation. This combination makes it possible to fine-tune very large models on hardware with comparatively smaller global memory size, since the dominant memory cost of storing the base model's parameters, is shrunk through quantization, while the actual learning signal is carried entirely by the small, full-precision low-rank adaptors.

4.6 Summary

In this chapter, we shifted our focus from the architecture of AI accelerators to the practical question of how to actually use that hardware efficiently when running ML workloads. We began by learning to reason about performance itself: any operation on a GPU is either compute-bound, limited by the speed of its arithmetic units, or memory-bound, limited by how quickly data can be fetched from memory. Whether an operation falls into one regime or the other depends on how its arithmetic intensity — the ratio of operations performed to bytes of data accessed — compares against the hardware's own compute-to-bandwidth ratio. We visualized this relationship using the roofline plot, and saw that operations with low arithmetic intensity, such as element-wise activations or normalization layers, are usually memory-bound, while large matrix multiplications tend to be compute-bound, and that this distinction dictates which optimization strategies are worth applying in a given problem.

We then surveyed several widely used optimization techniques that improve performance on real hardware. Batching converts inefficient matrix-vector operations into matrix-matrix operations, exposing more parallel work and reusing weights across many inputs at once. Data reuse, through techniques such as pinning frequently accessed values in shared memory, cuts down on redundant trips to slow off-chip memory. Kernel fusion combines multiple operations into a single kernel, avoiding the wasteful round trip of writing an intermediate result out to memory only to immediately read it back in. We also examined how the layout of data in memory, such as the choice between NHWC and NCHW for image tensors, and how shared memory bank conflicts, and their mitigation through padding, can affect how efficiently the GPU's memory hierarchy is used.

We then looked more closely at tiling, one of the most important of these techniques. By decomposing a large matrix multiplication into smaller tile-by-tile products that fit into fast on-chip memory, tiling allows each fetched element to be reused across several output computations, reducing the number of expensive memory accesses, without changing the underlying arithmetic. We saw that choosing a tile size is a balancing act between maximizing data reuse and respecting the hardware's cache capacity and occupancy constraints.

Building on this idea of tiling, we studied FlashAttention, an algorithm that applies the same principle to the attention mechanism. Standard attention materializes the full $N \times N$ score and softmax matrices in HBM, resulting in memory traffic that grows quadratically with sequence length. FlashAttention avoids this by tiling the query, key, and value matrices, and using an online softmax technique to incrementally and correctly compute normalized attention outputs one tile at a time, all without ever writing the full attention matrix back to memory.

Finally, we looked beyond hardware-level optimizations to techniques that change the model itself to reduce its resource demands. Pruning removes low-importance weights or entire structural units from a trained network. Knowledge distillation trains a smaller student model to mimic a larger teacher's soft output distributions. Low-rank matrix factorization approximates large weight matrices as the product of two much smaller ones. Quantization reduces the numerical precision used to represent weights and activations, trading some accuracy for large savings in memory and

compute. Mixture-of-experts grows a model's total parameter count while keeping the computation performed per input small, by routing each input to only a handful of expert sub-networks. We also surveyed efficient attention variants that reduce the quadratic cost of self-attention itself, and closed with parameter-efficient fine-tuning methods such as LoRA and QLoRA, which adapt large pretrained models to new tasks by training only a small number of additional low-rank parameters, leaving the bulk of the original model untouched.

Having understood both the hardware that runs ML workloads and the techniques used to make model execution efficient on the hardware, we now turn to the software layer that ties all of these ideas together. The next chapter studies high-level ML programming frameworks like PyTorch and TensorFlow, and ML compilation frameworks that convert high-level software to low-level code, both of which incorporate several such hardware-aware performance optimization techniques, allowing users to easily write high-level performant code.

References

Beltagy, Iz, Matthew E. Peters, and Arman Cohan (2020). *Longformer: The Long-Document Transformer*.

Child, Rewon, Scott Gray, Alec Radford, and Ilya Sutskever (2019). *Generating Long Sequences with Sparse Transformers*.

Choromanski, Krzysztof, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller (2022). *Rethinking Attention with Performers*.

Dai, Zihang, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov (2019). *Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context*.

Dao, Tri (2023). *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*.

Dao, Tri, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré (2022). "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Dettmers, Tim, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer (2023). *QLoRA: Efficient Finetuning of Quantized LLMs*.

Hu, Edward J., Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen (2021). *LoRA: Low-Rank Adaptation of Large Language Models*.

Rae, Jack W., Anna Potapenko, Siddhant M. Jayakumar, and Timothy P. Lillicrap (2019). *Compressive Transformers for Long-Range Sequence Modelling*. URL: https://arxiv.org/abs/1911.05507.

Roy, Aurko, Mohammad Saffar, Ashish Vaswani, and David Grangier (2021). "Efficient Content-Based Sparse Attention with Routing Transformers". In: *Transactions of the Association for Computational Linguistics* 9, pp. 53–68. DOI: 10.1162/tacl_a_00353. URL: https://aclanthology.org/2021.tacl-1.4/.

Shah, Jay, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao (2024). *FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision*.

Tay, Yi, Mostafa Dehghani, Dara Bahri, and Donald Metzler (2022). *Efficient Transformers: A Survey*.

Wang, Sinong, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma (2020). *Linformer: Self-Attention with Linear Complexity*.

Xiong, Yunyang, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh (2021). *Nyströmformer: A Nyström-Based Algorithm for Approximating Self-Attention*. URL: https://arxiv.org/abs/2102.03902.

Zaheer, Manzil, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed (2021). *Big Bird: Transformers for Longer Sequences*.