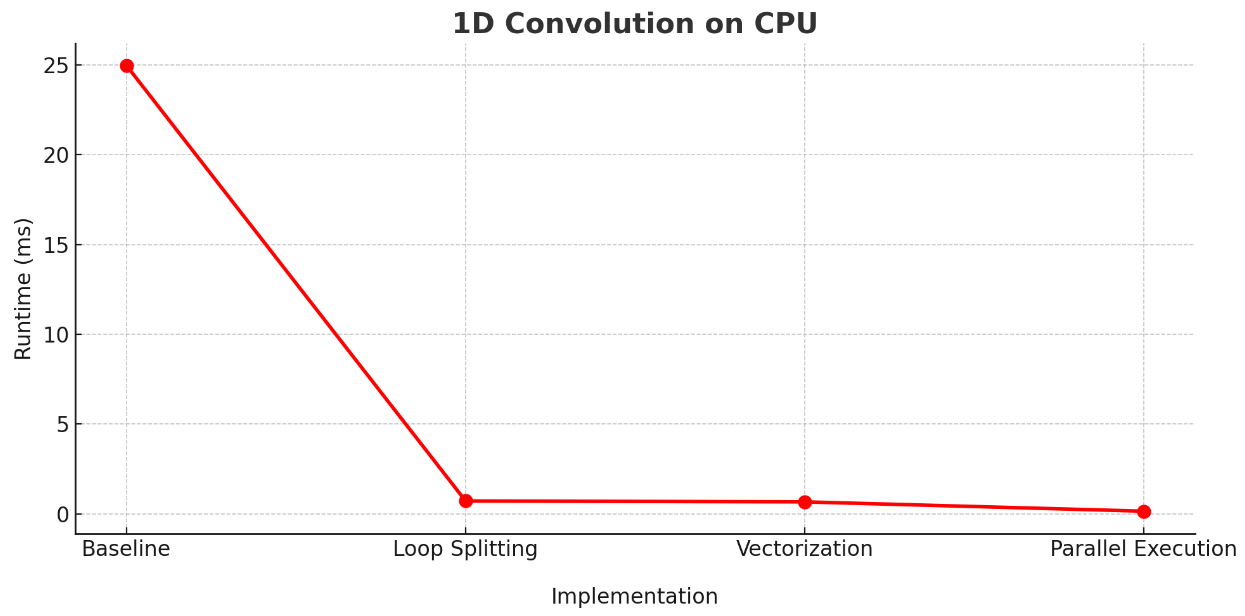
Kevin Pan (kp639)

ECE5545 Assignment 3

** Using 1 late day*

1) 1D convolution on CPU

Implementation	Runtime (ms)	Speedup
Baseline	24.965313	-
Optimization 1: Loop Splitting	0.712652	35.0x
Optimization 2: Vectorization	0.664596	37.6x
Optimization 3: Parallelization	0.144823	172.4x
Numpy Reference	0.151967	-



Baseline
Baseline Runtime: 24.965313 ms
Numpy Reference Runtime: 0.151967 ms

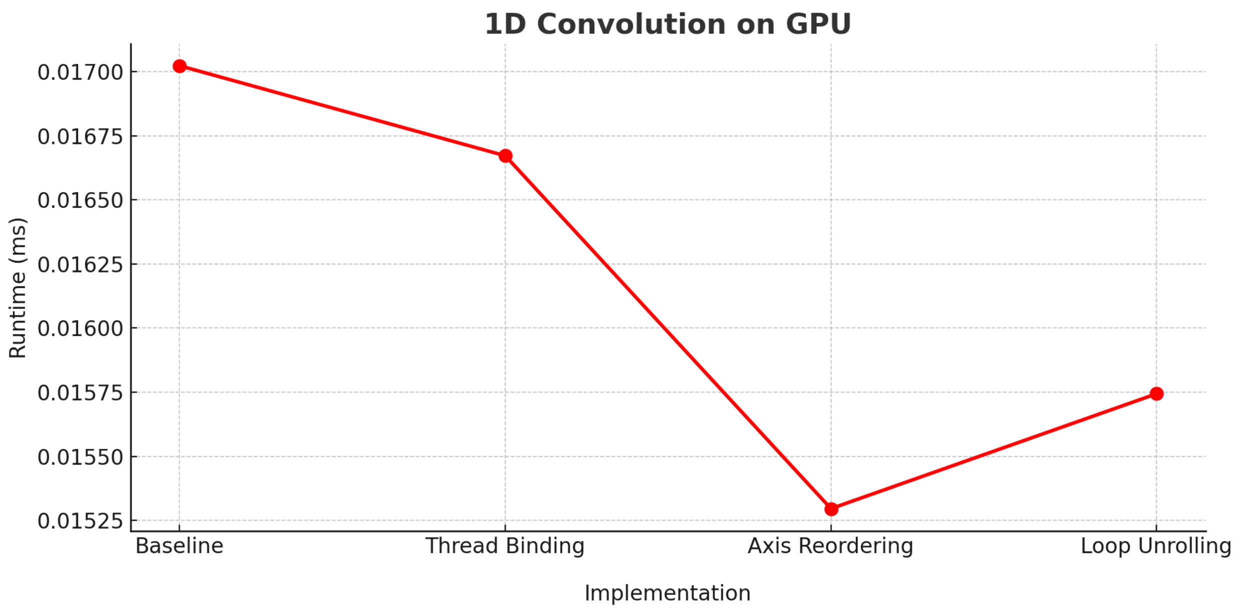
Optimization 1: Loop Splitting
This optimization splits the main computation loop into inner and outer loops with a factor of 32. By dividing the computation into chunks of 32 elements, we improve data locality and prepare for subsequent optimizations. The inner loop processes elements within each chunk, while the outer loop iterates over the chunks themselves. After splitting, the loops are reordered so the inner loop runs to completion before the outer loop advances. This pattern maps well to the CPU's cache hierarchy and enables more efficient memory access. It also creates a structured loop body ideal for applying vectorization and parallelization.
Runtime: 0.712652 ms

Optimization 2: Vectorization
Vectorization is applied to the inner loop introduced by loop splitting. This enables the use of SIMD (Single Instruction Multiple Data) instructions, which allow the CPU to process multiple floating-point elements in parallel with a single instruction. A vector width of 32 is used to match the chunk size, maximizing the number of values processed per cycle. This optimization reduces instruction overhead and increases computational throughput by taking full advantage of the CPU's vector processing units.
Runtime: 0.664596 ms

Optimization 3: Parallelization
This optimization applies parallelization to the outer loop, distributing chunks of computation across multiple CPU cores. Each thread processes a different chunk of the output concurrently, enabling the program to scale with the number of available CPU cores. Combined with vectorization, this yields efficient hierarchical parallelism across both cores and vector units.
Runtime: 0.144823 ms

2) 1D convolution on GPU

Implementation	Runtime (ms)	Speedup
Baseline	0.017023	-
Optimization 1: Thread Binding	0.016672	1.02x
Optimization 2: Axis Reordering	0.015296	1.11x
Optimization 3: Loop Unrolling	0.015744	1.08x
Numpy Reference	0.280423	-



Baseline

Baseline Runtime: 0.017023 ms
Numpy Reference Runtime: 0.280423 ms

Optimization 1: Thread Binding

This optimization splits the output axis into block and thread dimensions with a factor of 128, then binds these dimensions to the GPU's execution hierarchy. Blocks are assigned to streaming multiprocessors, while threads are mapped to individual CUDA cores. This setup enables hundreds of threads to compute output elements in parallel, allowing the GPU to process large workloads efficiently. The result is high throughput driven by the massive parallelism inherent in GPU hardware.
Runtime: 0.016672 ms

Optimization 2: Axis Reordering

Axis reordering changes the loop structure to move the reduction axis (k) inside the thread loop. This ensures each thread performs its full reduction sequentially, improving memory access locality and reducing latency. The adjusted access pattern minimizes thread divergence and improves coherence between adjacent threads, which better utilizes the GPU's memory subsystem and cache hierarchy.
Runtime: 0.015296 ms

Optimization 3: Loop Unrolling

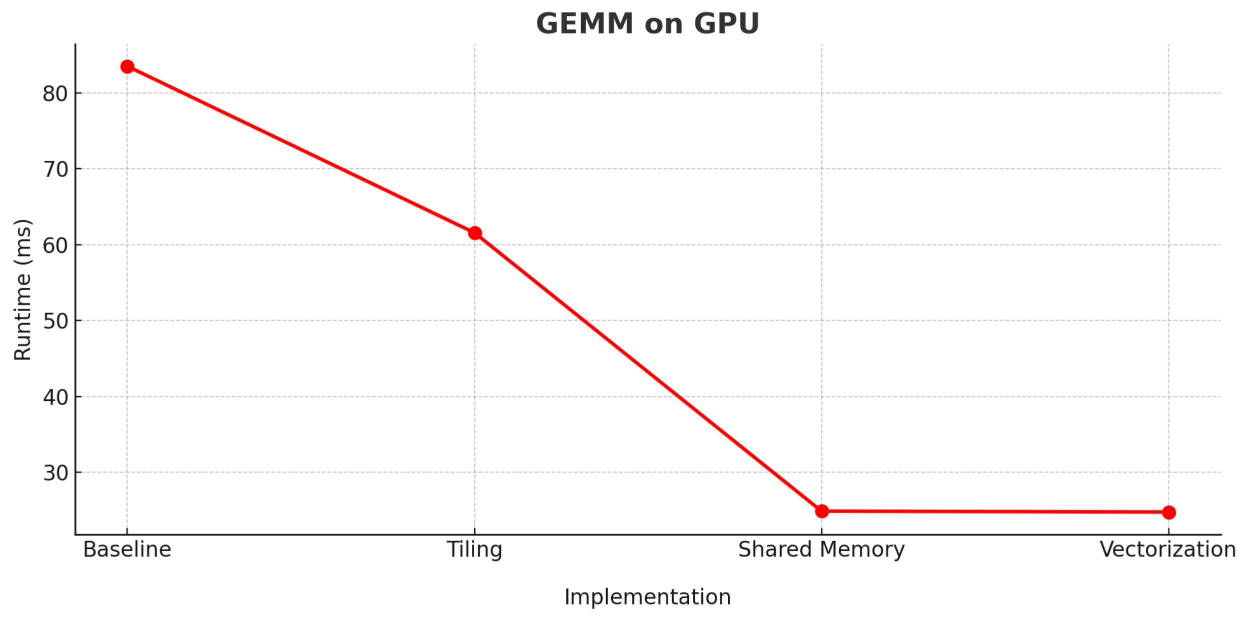
This optimization uses an auto-unroll pragma to unroll reduction loops with up to 8 iterations. Unrolling eliminates loop control overhead and reduces branching, allowing more cycles to be dedicated to computation. It also improves instruction-level parallelism and register usage within threads, leading to more efficient execution for small kernels.
Runtime: 0.015744 ms

Analysis of Optimization Performance:

The minimal performance gains after thread binding indicate that most of the parallelism was already exploited at the initial stage. The 1D convolution is likely memory-bound, where performance is constrained by memory bandwidth rather than compute resources. Once the workload is distributed across threads, additional optimizations such as axis reordering and loop unrolling offer limited benefits due to small kernel sizes and minimal data reuse.

GEMM on GPU

Implementation	Runtime (ms)	Speedup
Baseline	83.515296	-
Optimization 1: Tiling	61.567073	1.4x
Optimization 2: Shared Memory	24.870943	3.4x
Optimization 3: Vectorization	24.762144	3.4x
Numpy Reference	52.949851	-



Baseline

Baseline Runtime: 83.515296 ms
Numpy Reference Runtime: 52.949851 ms

Optimization 1: Tiling

Tiling breaks the matrix multiplication into smaller blocks by splitting the M, N, and K dimensions and mapping them to GPU thread blocks and threads. This allows multiple tiles to be computed in parallel across thousands of GPU cores. The tiling strategy ensures that each thread block operates on a manageable section of data, which can be more efficiently stored in registers or shared memory. This improves parallelism, reduces memory latency, and enables better data reuse within the GPU’s thread hierarchy.
Runtime: 61.567073 ms

Optimization 2: Shared Memory

This optimization caches tiles of the input matrices in fast on-chip shared memory accessible by all threads within a block. Instead of repeatedly accessing high-latency global memory, threads cooperatively load shared data once and reuse it across multiple operations. This reduces memory bandwidth demands and lowers overall latency. The improved locality and reduced contention significantly improve throughput, particularly for large matrix sizes where memory access is a major bottleneck.
Runtime: 24.870943 ms

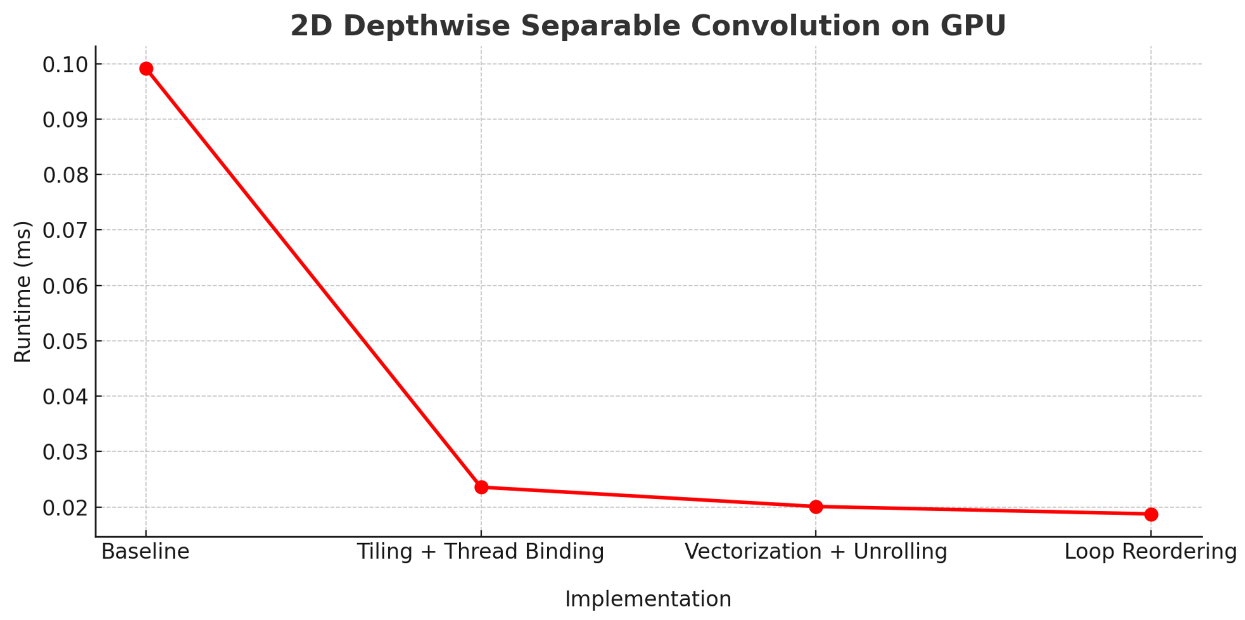
Optimization 3: Vectorization

Vectorization is applied to shared memory operations by splitting fused axes and vectorizing the innermost dimension with a factor of 4. This enables SIMD execution, where multiple data elements are processed simultaneously using a single instruction. Coalesced memory accesses further boost bandwidth efficiency, and the approach complements loop unrolling on the reduction axis to maximize instruction-level parallelism. This optimization improves arithmetic intensity and better utilizes GPU compute pipelines.
Runtime: 24.762144 ms

Performance Analysis:
Most of the performance gain is achieved through tiling and shared memory. Subsequent optimizations such as vectorization yield minimal improvements, likely due to hardware bottlenecks or limited benefit for the specific matrix size used. The matrix dimensions used in testing may have already been optimally mapped to the GPU's architecture by our first two optimizations, and smaller matrices often don't benefit from vectorization as setup costs outweigh potential gains.

2D Depthwise Separable Convolution on GPU

Implementation	Runtime (ms)	Speedup
Baseline	0.099200	-
Optimization 1: Tiling and Thread Binding	0.023584	4.2x
Optimization 2: Vectorization and Unrolling	0.020096	4.9x
Optimization 3: Loop Reordering	0.018751	5.3x



Baseline

Baseline Runtime: 0.099200 ms

Optimization 1: Tiling and Thread Binding

This optimization divides the output tensor into tiles by splitting the batch, channel, height, and width dimensions. The height and width axes are tiled with a factor of 16 to create small patches of output for each thread to compute. These tiles are then mapped to the GPU thread hierarchy: the batch dimension is bound to blockIdx.z, channel to blockIdx.y, and fused height/width outer loops to blockIdx.x. The inner loops are assigned to threadIdx coordinates. This configuration ensures efficient work distribution across threads and streaming multiprocessors, allowing the GPU to execute many tiles in parallel and fully utilize its parallel processing capabilities.
Runtime: 0.023584 ms

Optimization 2: Vectorization and Unrolling

This optimization targets instruction-level parallelism. First, the padding operation is inlined using compute_inline(), eliminating the need for a separate buffer and reducing memory overhead. Padding logic is directly embedded into the main computation, improving cache efficiency. Next, inner loops are unrolled using a pragma with a step size of 512. Unrolling reduces loop control overhead and enables deeper pipelining within each thread, allowing multiple instructions to be executed in parallel. This optimization complements thread-level parallelism with improved per-thread execution efficiency.
Runtime: 0.020096 ms

Optimization 3: Loop Reordering

This optimization reorders the reduction axes (r_h and r_w) so that r_w executes before r_h. Since adjacent elements in the width direction are typically laid out contiguously in memory, this change improves spatial locality and reduces cache misses. With a more cache-friendly traversal pattern, threads access data more efficiently, lowering memory latency during convolution. When combined with tiling and unrolling, this optimization helps maximize memory throughput and ensures that each thread's work is both well-balanced and cache-optimal.
Runtime: 0.018751 ms