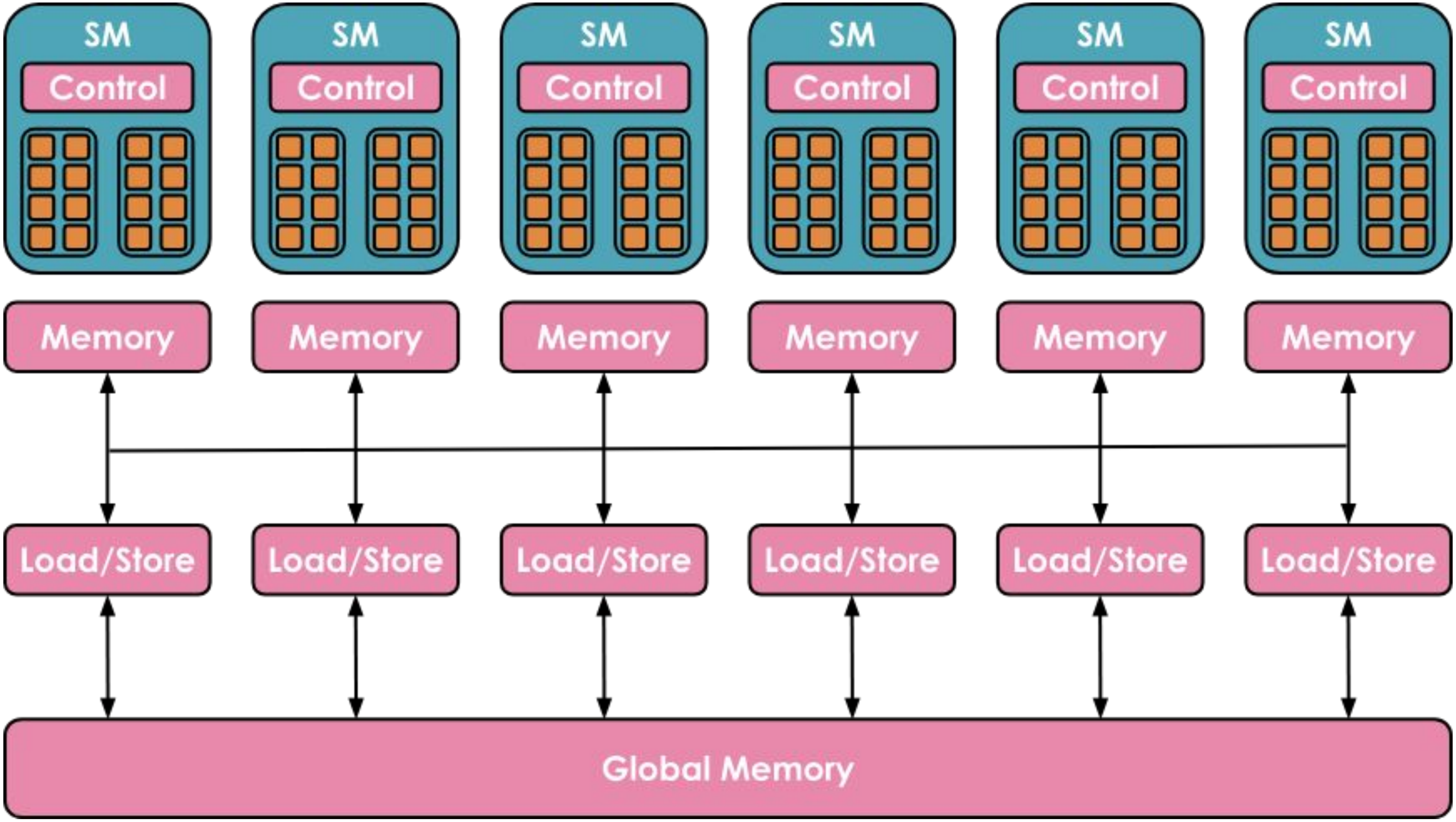
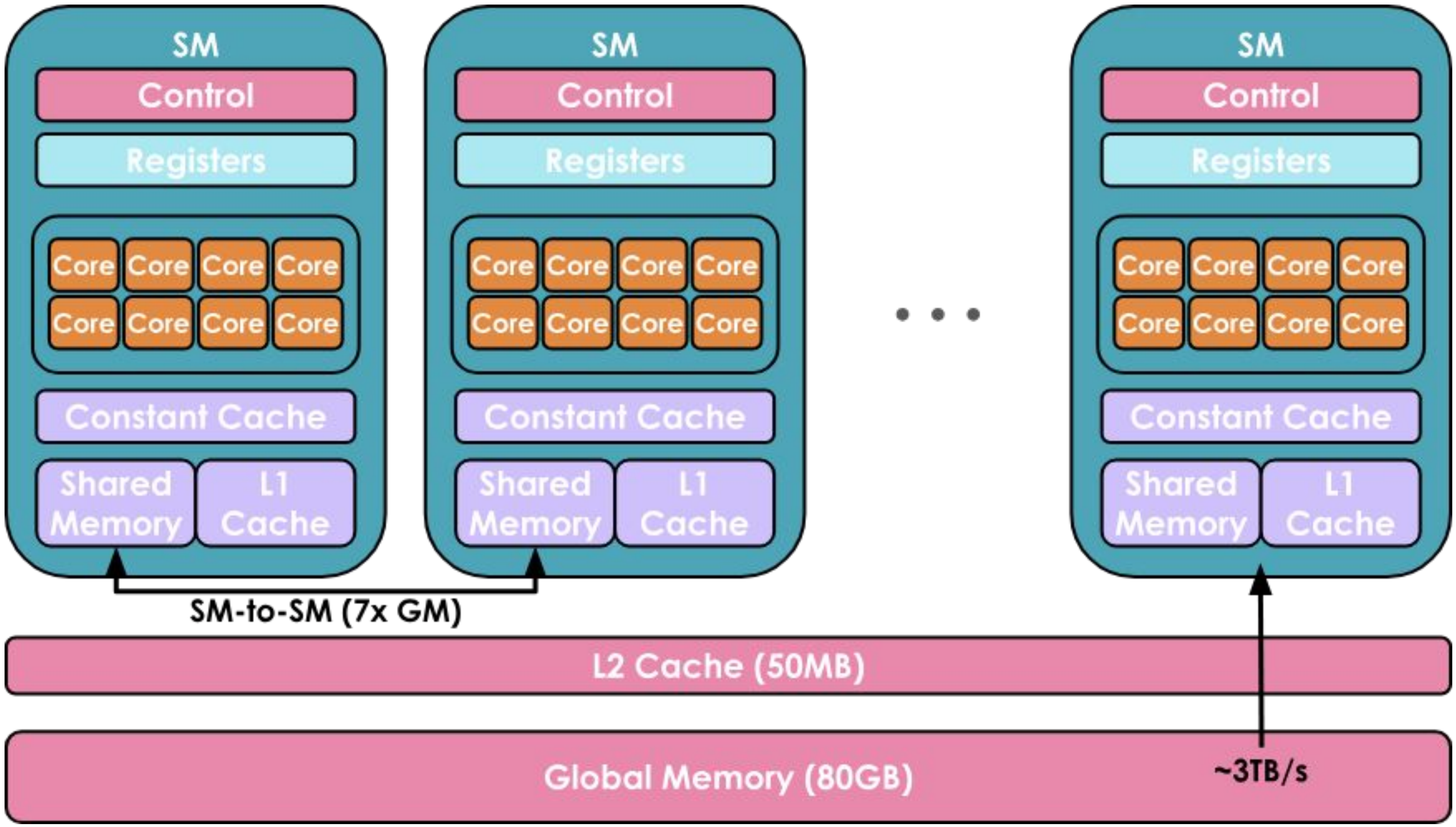
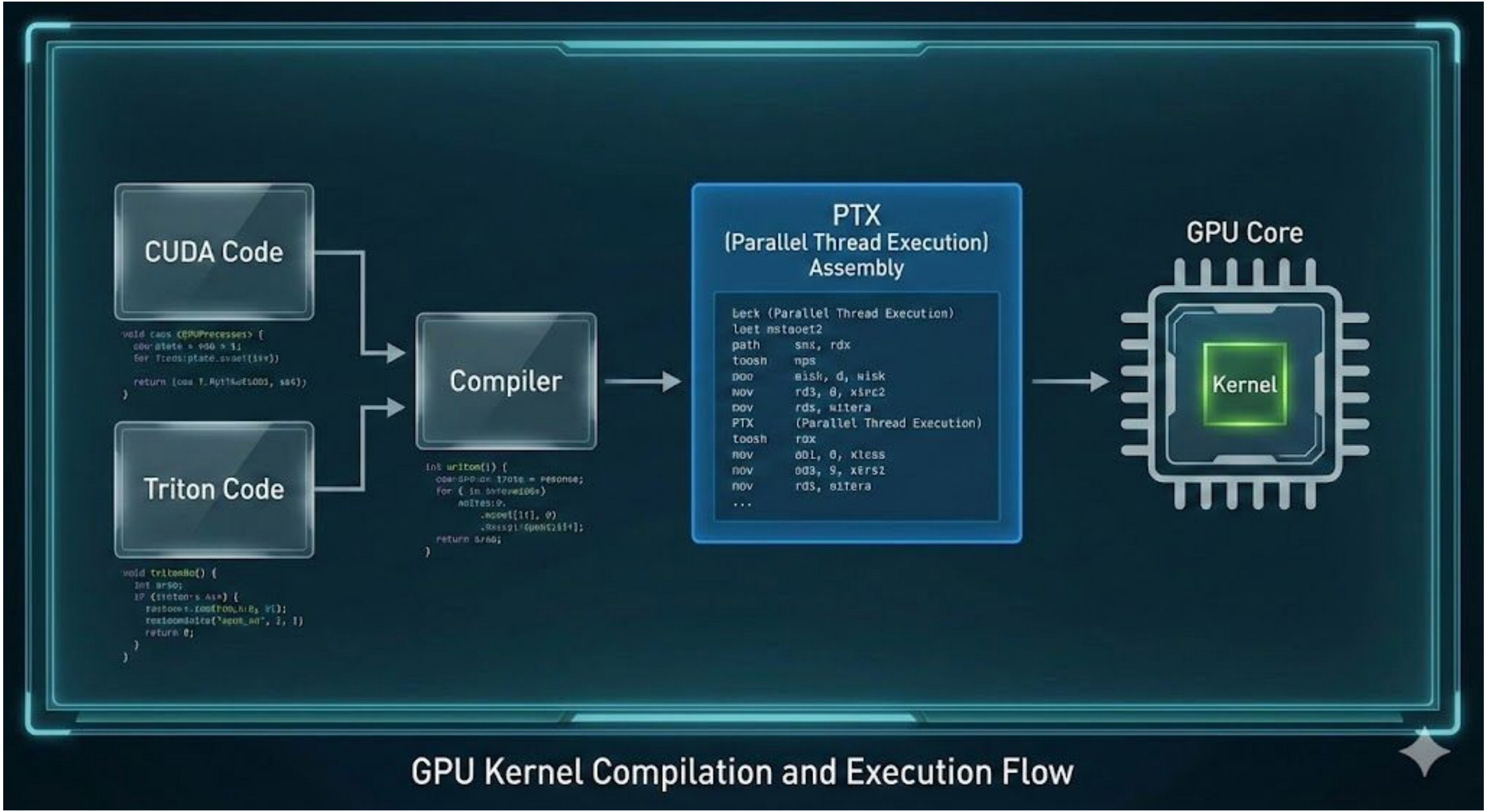
Diving into the GPUs [:fused kernels]

made with ❤️ for “Little ML book club”


SM
Control
SM
Control
SM
Control
SM
Control
SM
Control
SM
Control
Memory
Memory
Memory
Memory
Memory
Memory
Load/Store
Load/Store
Load/Store
Load/Store
Load/Store
Load/Store
Global Memory

GPU	Architecture	Year	SMs	Cores/SM	Total CUDA Cores	Tensor Cores	HBM	Memory BW
P100	Pascal	2016	56	64	3,584	—	16 GB HBM2	732 GB/s
V100	Volta	2017	80	64	5,120	640	32 GB HBM2	900 GB/s
A100 (80GB)	Ampere	2020	108	64	6,912	432	80 GB HBM2e	2,039 GB/s
H100 SXM	Hopper	2022	132	128	16,896	528	80 GB HBM3	3,350 GB/s
H200	Hopper	2024	132	128	16,896	528	141 GB HBM3e	4,800 GB/s
B200	Blackwell	2024	192	128	24,576	768*	192 GB HBM3e	8,000 GB/s

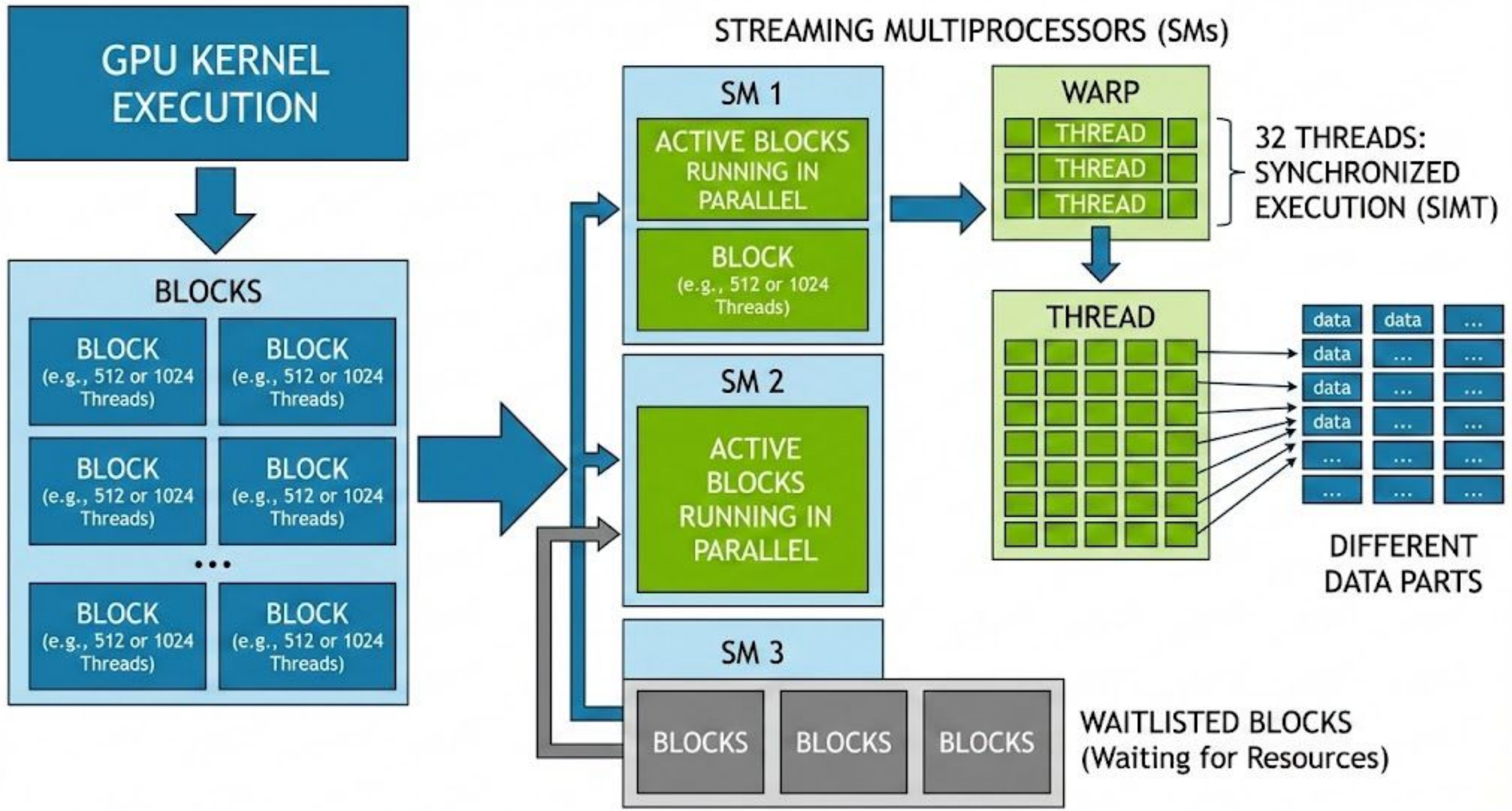

SM
Control
Registers
Core Core Core Core
Core Core Core Core
Constant Cache
Shared Memory
L1 Cache
SM
Control
Registers
Core Core Core Core
Core Core Core Core
Constant Cache
Shared Memory
L1 Cache
SM
Control
Registers
Core Core Core Core
Core Core Core Core
Constant Cache
Shared Memory
L1 Cache
SM-to-SM (7x GM)
L2 Cache (50MB)
~3TB/s
Global Memory (80GB)


CUDA Code
Triton Code
Compiler
PTX
(Parallel Thread Execution)
Assembly
GPU Core
Kernel
GPU Kernel Compilation and Execution Flow

```
// Host code
void vecAdd(float* h_A, float *h_B, float *h_c, int n) {
    // Allocate vectors in device memory
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

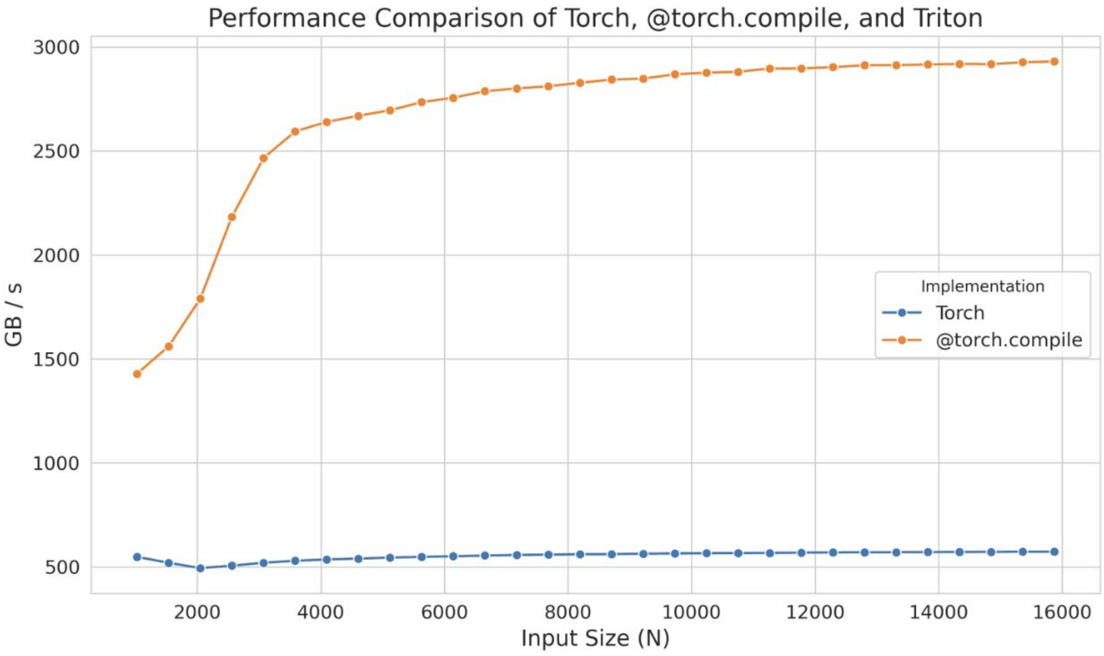
```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Device code containing the definition of the vector addition kernel, adapted from https://docs.nvidia.com/cuda/cuda-c-programming-guide/ and https://blog.codingconfessions.com/p/gpu-computing


GPU KERNEL EXECUTION
BLOCKS
BLOCK (e.g., 512 or 1024 Threads)
BLOCK (e.g., 512 or 1024 Threads)
BLOCK (e.g., 512 or 1024 Threads)
BLOCK (e.g., 512 or 1024 Threads)
...
BLOCK (e.g., 512 or 1024 Threads)
BLOCK (e.g., 512 or 1024 Threads)
STREAMING MULTIPROCESSORS (SMs)
SM 1
ACTIVE BLOCKS RUNNING IN PARALLEL
BLOCK (e.g., 512 or 1024 Threads)
SM 2
ACTIVE BLOCKS RUNNING IN PARALLEL
SM 3
BLOCKS
BLOCKS
BLOCKS
WAITLISTED BLOCKS (Waiting for Resources)
WARP
THREAD
THREAD
THREAD
32 THREADS: SYNCHRONIZED EXECUTION (SIMT)
THREAD
data
data
data
data
...
DIFFERENT DATA PARTS

$$\mathrm{ELU}(x) = \begin{cases} e^x - 1 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

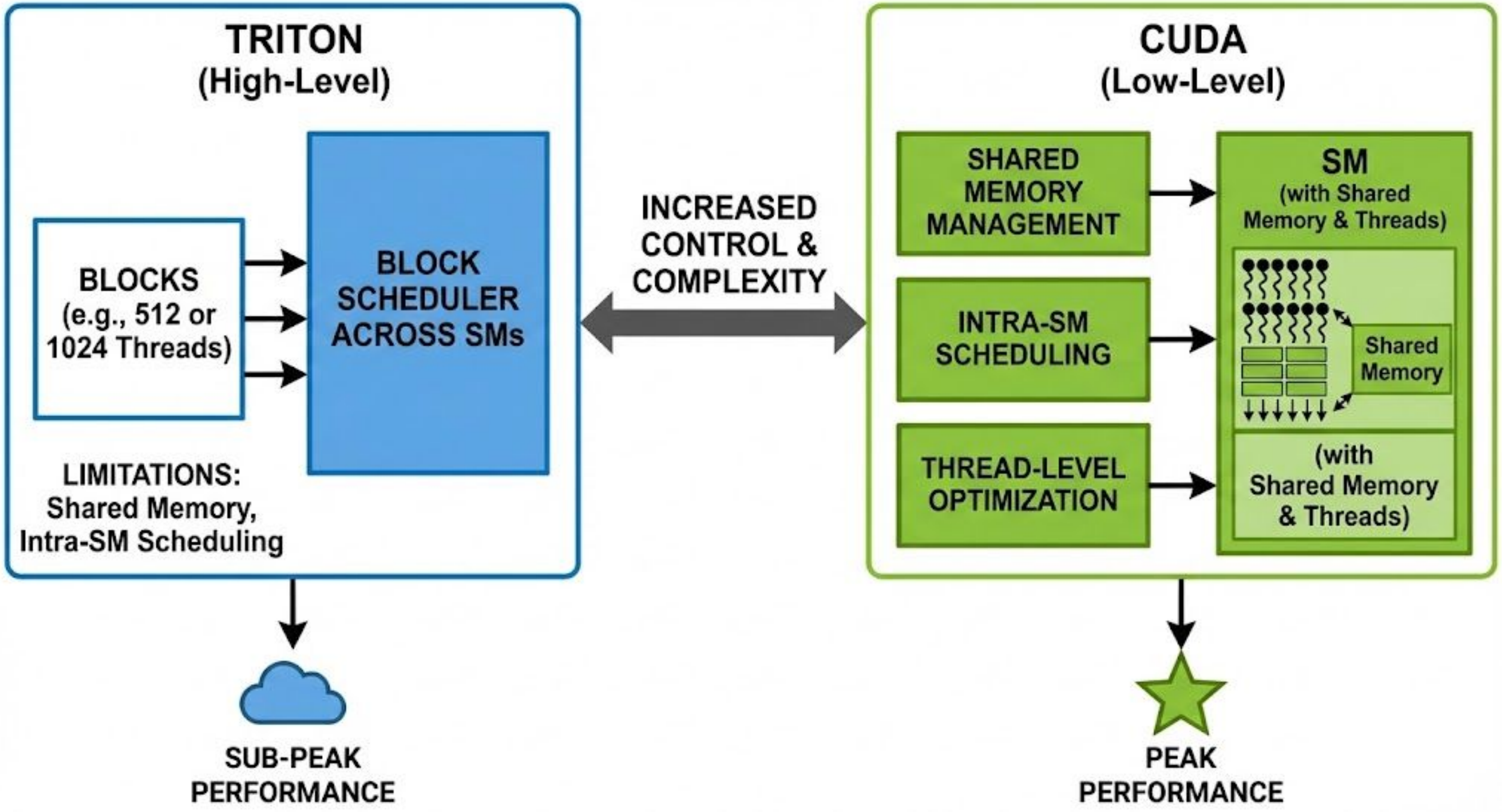
```
@torch.compile
def elu(x, alpha=1.0):
    return torch.where(x < 0, alpha * (torch.exp(x) - 1), x)
```



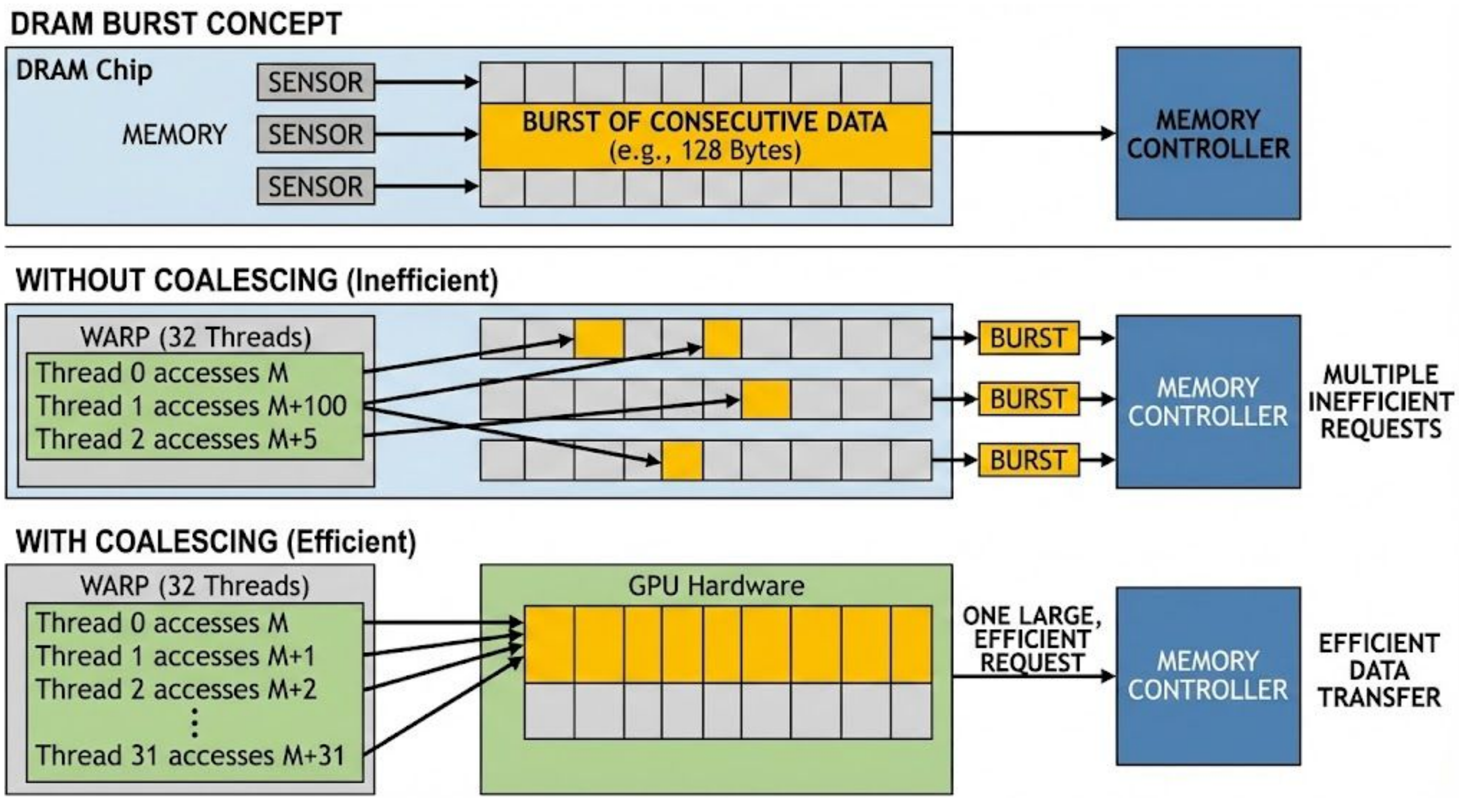
```
export TORCH_LOGS="output_code"
```

```python
@triton.jit
def triton_(in_ptr0, out_ptr0, xnumel, XBLOCK : tl.constexpr):
    xnumel = 100000000
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + (x0), xmask)
    tmp1 = 0.0
    tmp2 = tmp0 < tmp1
    tmp3 = tl_math.exp(tmp0)
    tmp4 = 1.0
    tmp5 = tmp3 - tmp4
    tmp6 = tl.where(tmp2, tmp5, tmp0)
    tl.store(out_ptr0 + (x0), tmp6, xmask)
```

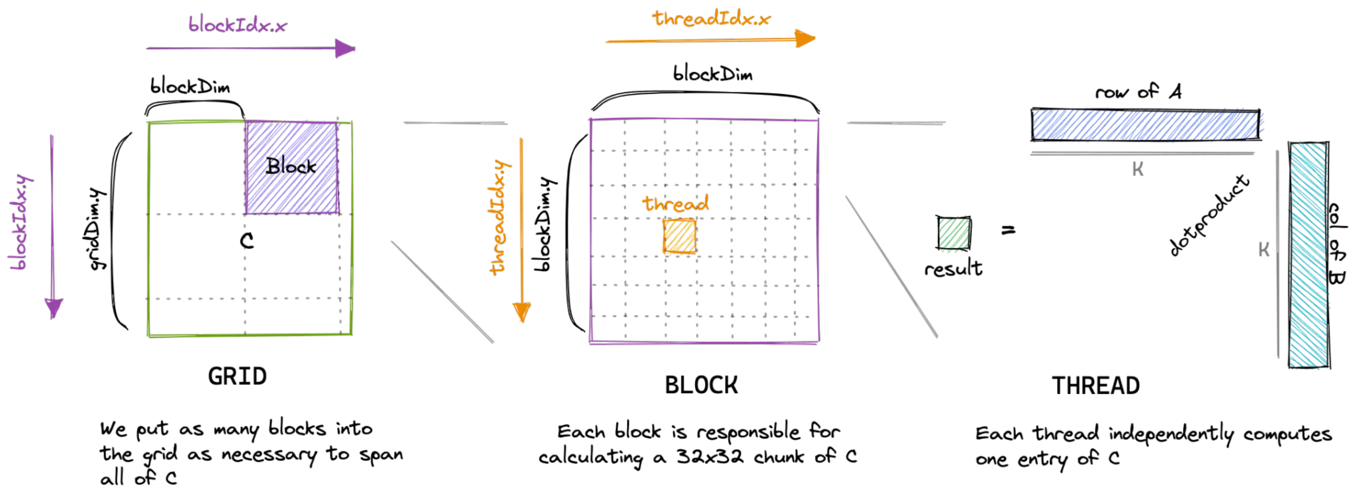
```python
@triton.jit
def elu_kernel(input_ptr, output_ptr, num_elements, BLOCK_SIZE: tl.constexpr):
    # Calculate the starting index for this block
    block_start = tl.program_id(0) * BLOCK_SIZE
    # Create an array of indices for this block
    block_indices = block_start + tl.arange(0, BLOCK_SIZE)[:]
    # Create a mask to ensure only valid indices are processed
    valid_mask = block_indices < num_elements
    # Load input values from the input pointer based on valid indices
    input_values = tl.load(input_ptr + block_indices, valid_mask)
    # Define the ELU parameters
    zero_value = 0.0  # Threshold for ELU activation
    negative_mask = input_values < zero_value
    exp_values = tl.math.exp(input_values)
    # Define the ELU output shift
    one_value = 1.0
    shifted_exp_values = exp_values - one_value

    output_values = tl.where(negative_mask, shifted_exp_values, input_values)

    # Store the computed output values back to the output pointer
    tl.store(output_ptr + block_indices, output_values, valid_mask)
```

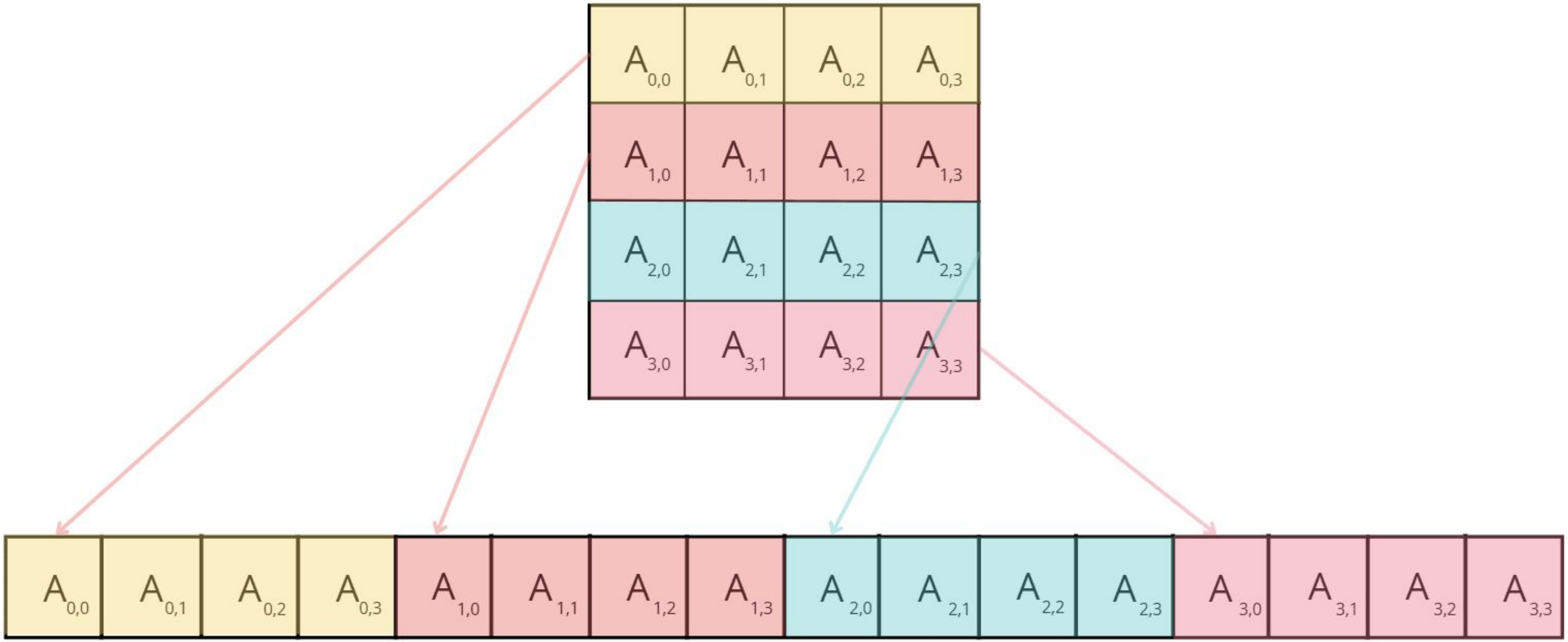

TRITON
(High-Level)
BLOCKS
(e.g., 512 or
1024 Threads)
BLOCK
SCHEDULER
ACROSS SMs
LIMITATIONS:
Shared Memory,
Intra-SM Scheduling
INCREASED
CONTROL &
COMPLEXITY
CUDA
(Low-Level)
SHARED
MEMORY
MANAGEMENT
INTRA-SM
SCHEDULING
THREAD-LEVEL
OPTIMIZATION
SM
(with Shared
Memory & Threads)
Shared
Memory
(with
Shared Memory
& Threads)
SUB-PEAK
PERFORMANCE
PEAK
PERFORMANCE

Memory coalescing

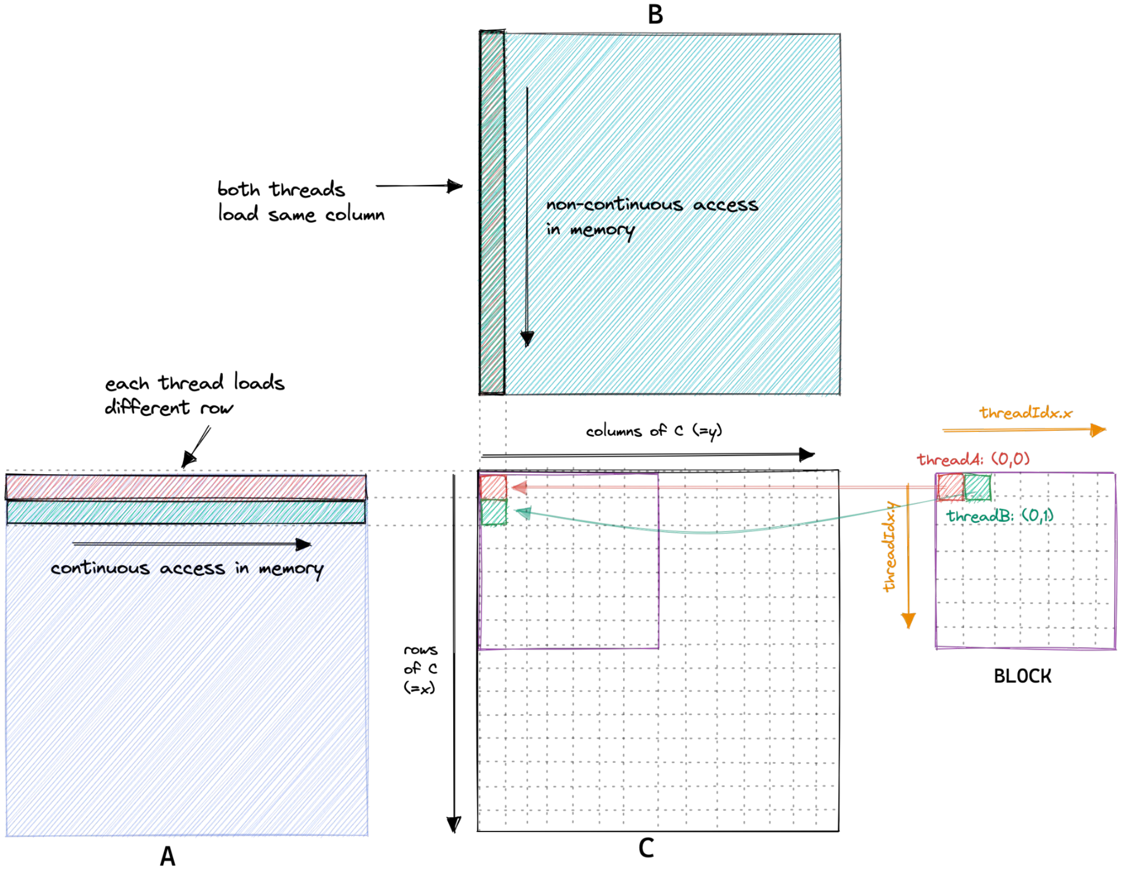


```
__global__ void matmul_naive(int M, int N, int K, const float *A, const float *B, float *C) {
    const uint x = blockIdx.x * blockDim.x + threadIdx.x;
    const uint y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < M && y < N) {
        float tmp = 0.0;
        for (int i = 0; i < K; ++i) {
            tmp += A[x * K + i] * B[i * N + y];
        }
        C[x * N + y] = tmp;
    }
}
```



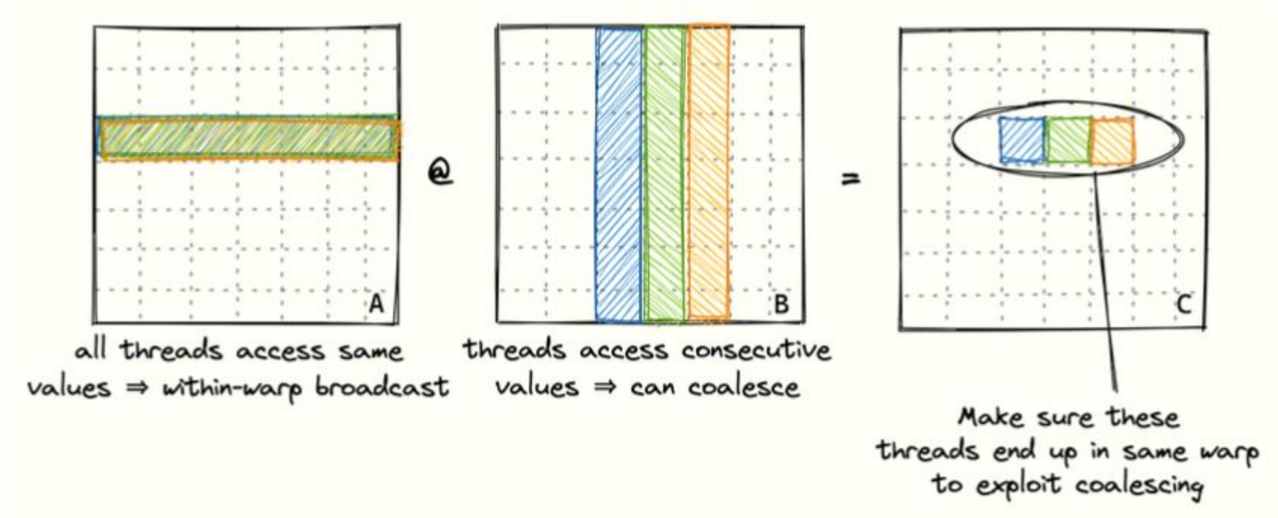


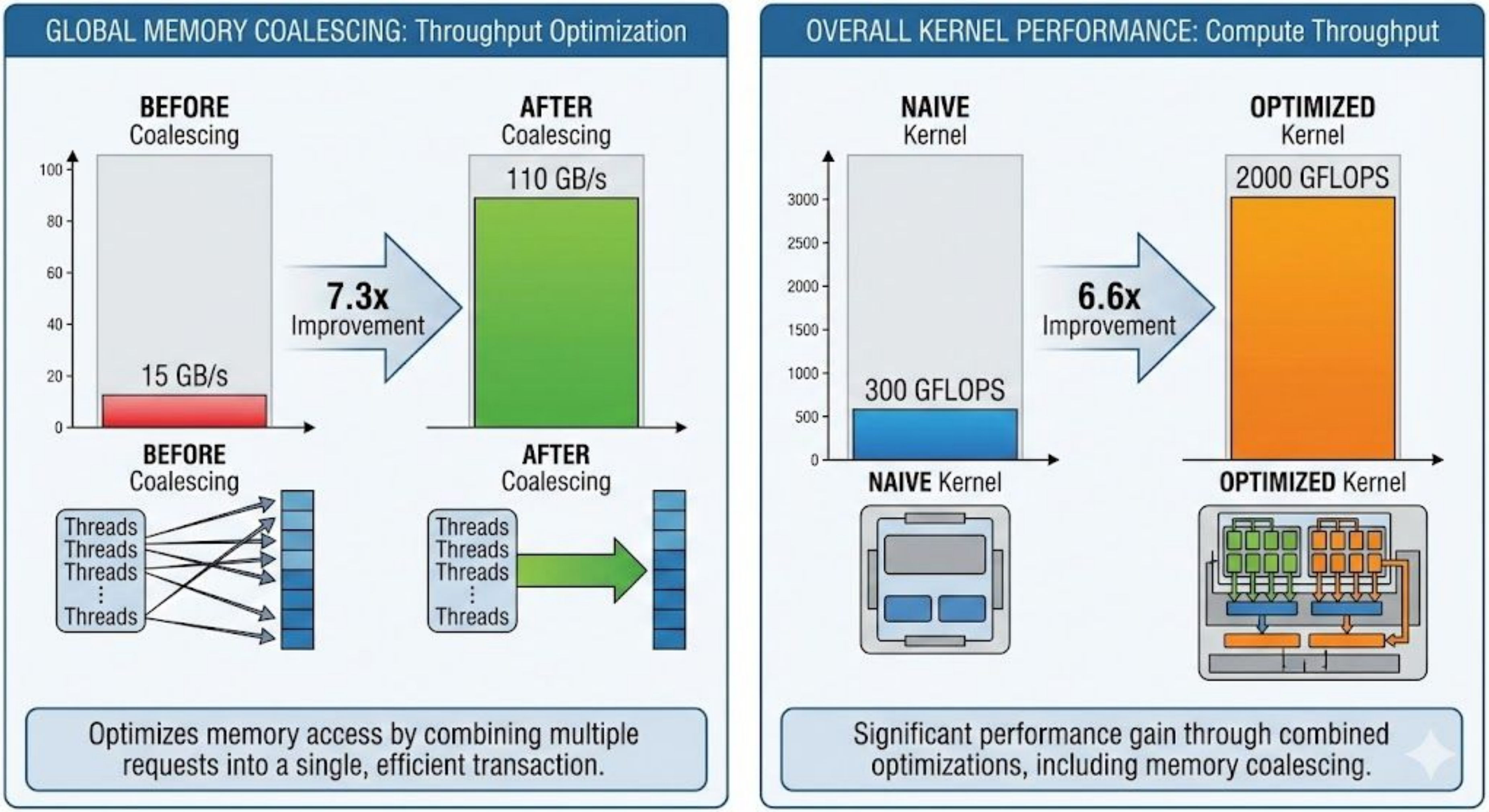
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$



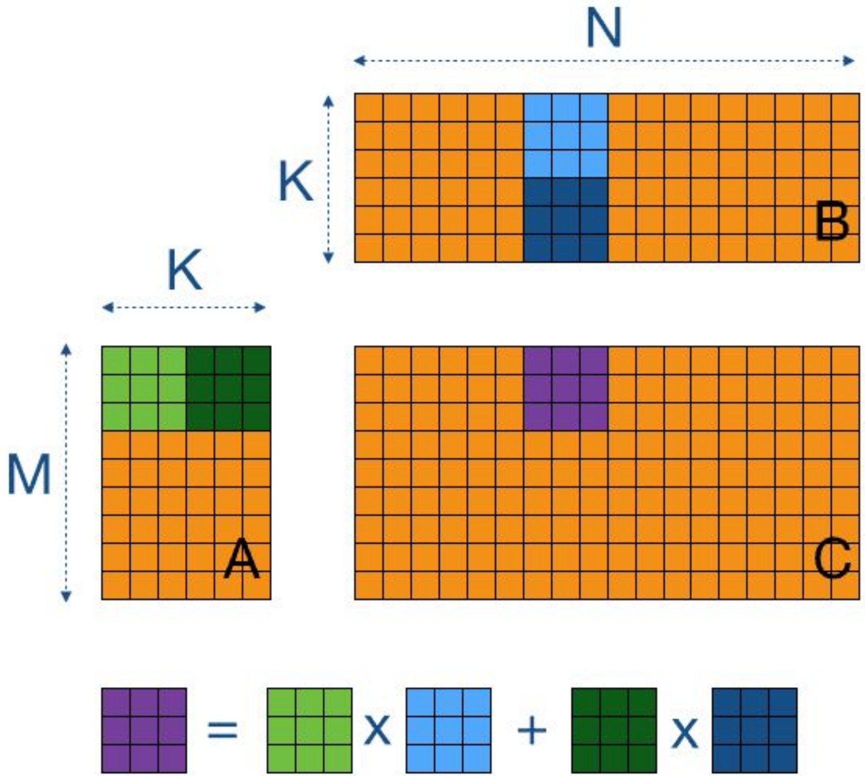
A, B, C are stored in row-major order.
This means that the last index (here y)
is the one that iterates continuous through
memory (=has stride 1).

```
const int x = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
const int y = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);

if (x < M && y < N) {
float tmp = 0.0;
for (int i = 0; i < K; ++i) {
    tmp += A[x * K + i] * B[i * N + y];
}
C[x * N + y] = tmp;
}
```




GLOBAL MEMORY COALESCING: Throughput Optimization
BEFORE Coalescing
AFTER Coalescing
100
80
60
40
20
0
15 GB/s
110 GB/s
7.3x Improvement
BEFORE Coalescing
AFTER Coalescing
Threads
Threads
Threads
Threads
Threads
Threads
Threads
Threads
Optimizes memory access by combining multiple requests into a single, efficient transaction.
OVERALL KERNEL PERFORMANCE: Compute Throughput
NAIVE Kernel
OPTIMIZED Kernel
3000
2500
2000
1500
1000
500
0
300 GFLOPS
2000 GFLOPS
6.6x Improvement
NAIVE Kernel
OPTIMIZED Kernel
Significant performance gain through combined optimizations, including memory coalescing.

Tiling



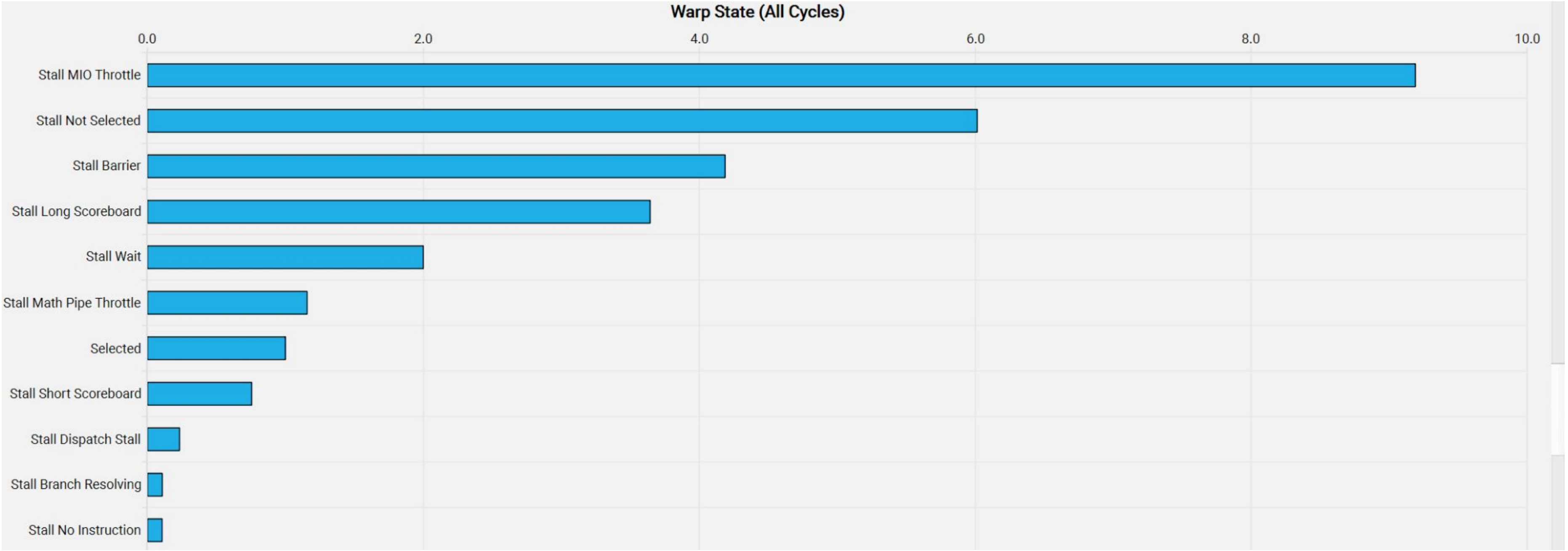
For simplicity, we consider a square-shaped tile.

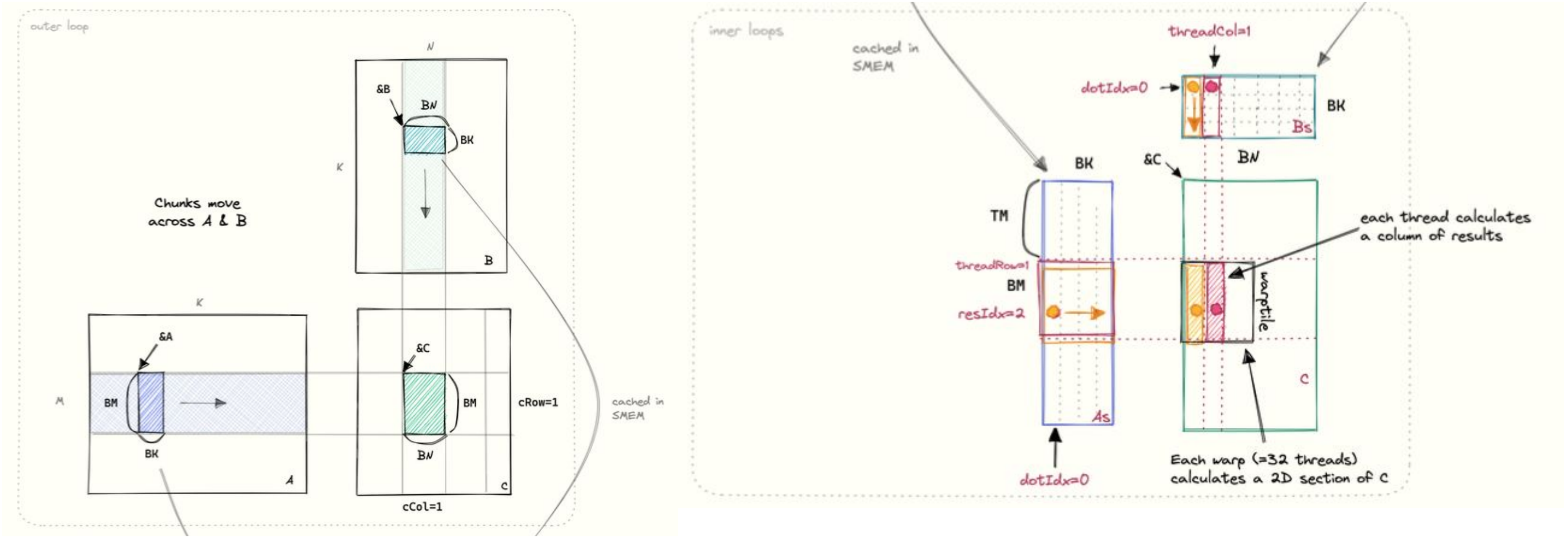
```
// Set pointers to the starting elements
A += blockRow * TILE_SIZE * K; // Start at row = blockRow, column = 0
B += blockCol * TILE_SIZE; // Start at row = 0, column = blockCol
C += blockRow * TILE_SIZE * N + blockCol * TILE_SIZE; // Start at row = blockRow, column = blockCol
float sum = 0.0;
// The outer loop moves through tiles of A (across columns) and B (down rows)
for (int tileIdx = 0; tileIdx < K; tileIdx += TILE_SIZE) {
sharedA[localRow * TILE_SIZE + localCol] = A[localRow * K + localCol];
sharedB[localRow * TILE_SIZE + localCol] = B[localRow * N + localCol];

// Ensure all threads in the block have completed data loading
__syncthreads();

// Shift pointers to the next tile
A += TILE_SIZE;
B += TILE_SIZE * N;

// Compute the partial dot product for this tile
for (int i = 0; i < TILE_SIZE; ++i) {
    sum += sharedA[localRow * TILE_SIZE + i] * sharedB[i * TILE_SIZE + localCol];
}
// Synchronize again to prevent any thread from loading new data
// into shared memory before others have completed their calculations
__syncthreads();
}
C[localRow * N + localCol] = sum;
```

```
// Set pointers to the starting elements
A += blockRow * TILE_SIZE * K; // Start at row = blockRow, column = 0
B += blockCol * TILE_SIZE; // Start at row = 0, column = blockCol
C += blockRow * TILE_SIZE * N + blockCol * TILE_SIZE; // Start at row = blockRow, column = blockCol
float sum = 0.0;
// The outer loop moves through tiles of A (across columns) and B (down rows)
for (int tileIdx = 0; tileIdx < K; tileIdx += TILE_SIZE) {
    sharedA[localRow * TILE_SIZE + localCol] = A[localRow * K + localCol];
    sharedB[localRow * TILE_SIZE + localCol] = B[localRow * N + localCol];

    // Ensure all threads in the block have completed data loading
    __syncthreads();

    // Shift pointers to the next tile
    A += TILE_SIZE;
    B += TILE_SIZE * N;

    // Compute the partial dot product for this tile
    for (int i = 0; i < TILE_SIZE; ++i) {
        sum += sharedA[localRow * TILE_SIZE + i] * sharedB[i * TILE_SIZE + localCol];
    }
    // Synchronize again to prevent any thread from loading new data
    // into shared memory before others have completed their calculations
    __syncthreads();
}
C[localRow * N + localCol] = sum;
```

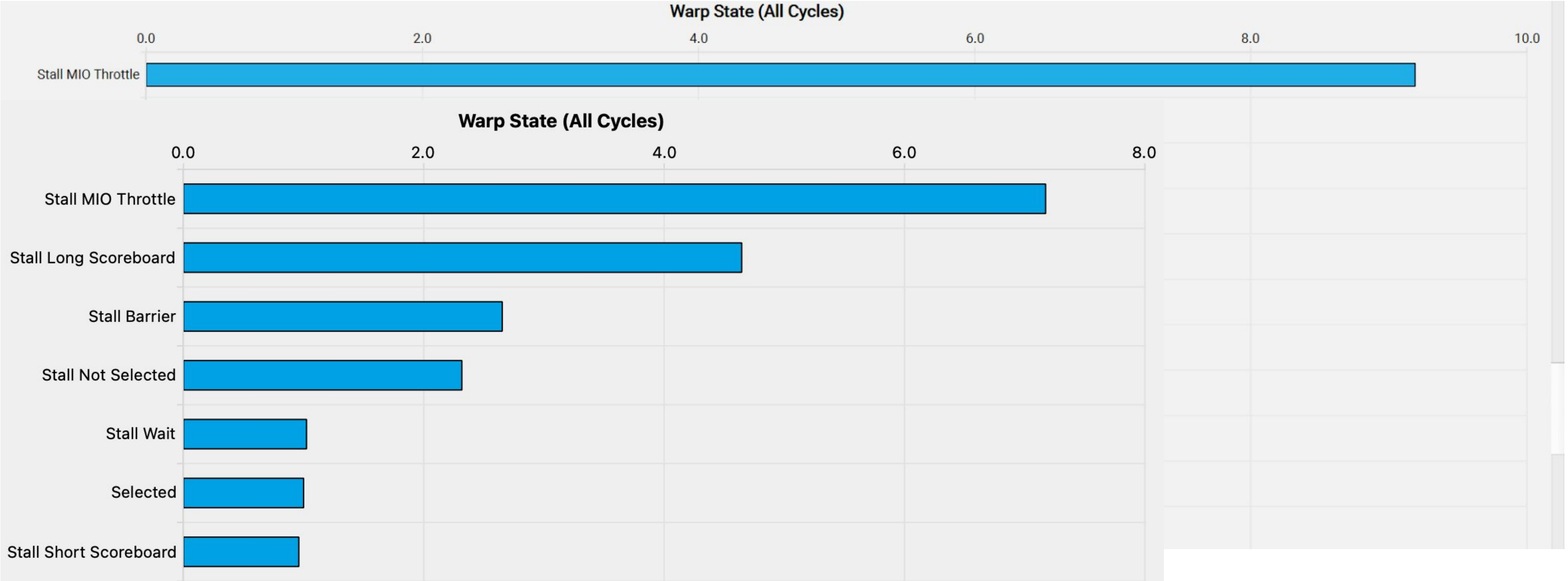
~2200 GFLOPS, a 50% improvement over the previous version

Thread coarsening



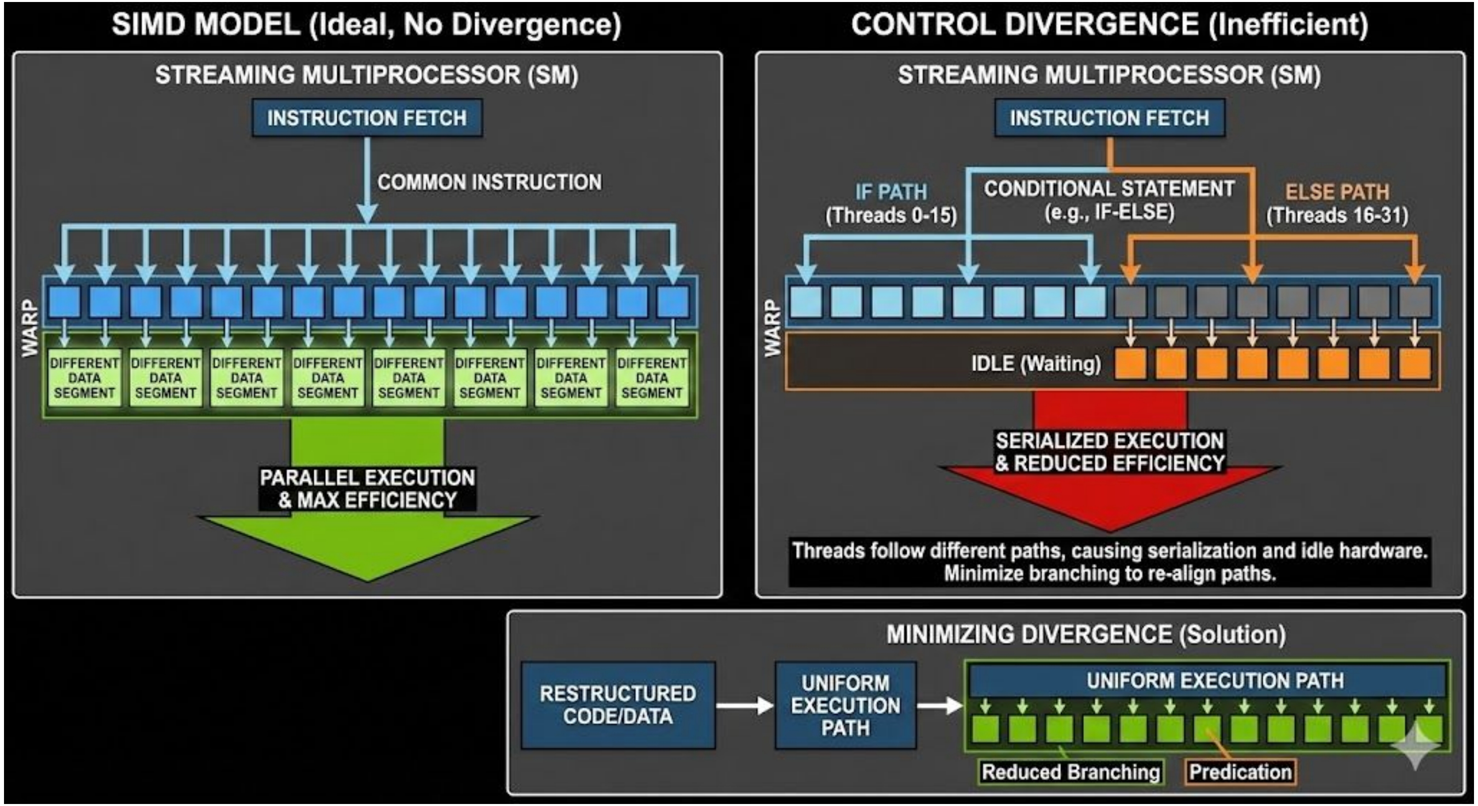


outer loop
N
&B
BN
BK
K
Chunks move
across A & B
B
K
&A
&C
M
BM
BK
A
BM
BN
C
cRow=1
cCol=1
cached in
SMEM
inner loops
cached in
SMEM
threadCol=1
dotIdx=0
BK
Bs
BK
&C
BN
TM
each thread calculates
a column of results
threadRow=1
BM
resIdx=2
warptile
C
As
dotIdx=0
Each warp (=32 threads)
calculates a 2D section of C



Warp State (All Cycles)
0.0
2.0
4.0
6.0
8.0
10.0
Stall MIO Throttle
Warp State (All Cycles)
0.0
2.0
4.0
6.0
8.0
Stall MIO Throttle
Stall Long Scoreboard
Stall Barrier
Stall Not Selected
Stall Wait
Selected
Stall Short Scoreboard

Minimizing control divergence



see you next time