Lec6: Performance Considerations

Based on textbook chapter 6

Agenda

- We have discussed in last lecture how to **reduce** global memory access/traffic
- This lecture we learn how to access global memory efficiently:
  - Memory coalescing
  - Latency hiding
- And efficient shared memory access
- And a checklist of performance considerations

Warmup: #0: Control Divergence in Warp

```
// Divergent kernel: Warp splits into two paths based on thread ID
__global__ void divergent_kernel_4way(int *output, int iterations) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int lane_id = threadIdx.x % 32; // Determine lane ID within the warp
    int value = idx;

    for (int i = 0; i < iterations; ++i) {
        // Condition causes divergence within the warp
        if (lane_id % 4 == 0) {
            value = (value * 3 + 5) % 123;
        } else if (lane_id %4 == 1) {
            value = (value * 5 + 4) % 125;
        } else if (lane_id %4 == 2) {
            value = (value * 5 + 89) % 125;
        } else {
            value = (value *2 + 9834) % 434;
        }

    }

    output[idx] = value;
}
```

```
// Divergent kernel: Warp splits into two paths based on thread ID
__global__ void divergent_kernel_2way(int *output, int iterations) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int lane_id = threadIdx.x % 32; // Determine lane ID within the warp
    int value = idx;

    for (int i = 0; i < iterations; ++i) {
        // Condition causes divergence within the warp
        if (lane_id % 2 == 0) {
            value = (value * 3 + 5) % 123;
        } else {
            value = (value * 5 + 3) % 123;
        }
    }

    output[idx] = value;
}
```

```
Divergent kernel time: 15.43 ms
Non-divergent kernel time: 2.91 ms
```

#1: Memory Coalescing

- DRAM is slow:
  - Long latency in accessing the cells
  - Each time one bit is accessed, a range of consecutive bits are provided—for free.
  - Good for spatial data locality
  - Burst access is fast, random access is slow

- Remember a warp of threads execute the same instruction…
- If the instruction is MEMORY LOAD, then we will have have 32 simultaneous memory requests to global memory
- If those 32 mem req are consecutive: e.g. thread 0 reads location X, thread 1 reads location X+1,… then
- The 32 mem req can be combined into 1 mem req, much faster!

Microbenchmarking:

```
// Coalesced kernel: consecutive threads access consecutive memory addresses
__global__ void coalesced_kernel(int *out, const int *in, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n) {
        out[tid] = in[tid];
    }
}

// Non-coalesced kernel: consecutive threads access strided memory addresses (stride = 32 elements)
__global__ void uncoalesced_kernel(int *out, const int *in, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    // Calculate a non-coalesced index: threads in a warp access 32 elements apart
    int index = (tid % 32) * 32  + (tid / 32);


    if (index < n) {
        out[index] = in[index];
    }
}
```

Size ~ 1GB in and out data

Coalesced Kernel Time: 15.48 ms

Non-Coalesced Kernel Time: 97.45 ms

Performance Ratio (Non-Coalesced/Coalesced): 6.29x

2d/3d threadIdx -> warp ID

- In CUDA the warp membership is determined by the thread's linear index within the block. For a 2D thread index, the linear index is computed as:

$$\text{linearID} = \text{threadIdx.x} + \text{threadIdx.y} \times \text{blockDim.x}$$

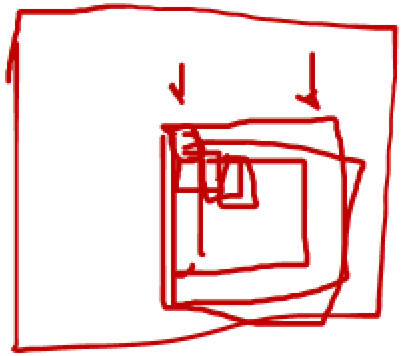
E.g. For a blockDim = (16,16), the following 32 threads belong to warp0: (coordinates are (x,y))

Warp0: (0,0), (1, 0), (2, 0), ..., (15, 0), (0, 1), (1, 1), ..., (15, 1)
Warp1: (0, 2), (1, 2), (2,2), ..., (15, 2), (0, 3), (1, 3), ..., (15, 3)

Memory Coalescing Example: Blur

```
// image: pixel is one float; size m*n stored in row major
__global__ void blur1(float *image, float *out, int m, int n)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    // average of 9 pixels
    if (i>0 && i<m-1 && j>0 && j<n-1)
    {
        float sum = 0;
        for (int ii=-1; ii<=1; ii++)
            for (int jj=-1; jj<=1; jj++)
                sum += image[(i+ii)*n + j+jj];
        out[i*n + j] = sum/9;
    }
}
```

```
// image: pixel is one float; size m*n stored in row major
__global__ void blur2(float *image, float *out, int m, int n)
{
    int i = threadIdx.y + blockIdx.y * blockDim.y;
    int j = threadIdx.x + blockIdx.x * blockDim.x;
    // average of 9 pixels
    if (i>0 && i<m-1 && j>0 && j<n-1)
    {
        float sum = 0;
        for (int ii=-1; ii<=1; ii++)
            for (int jj=-1; jj<=1; jj++)
                sum += image[(i+ii)*n + j+jj];
        out[i*n + j] = sum/9;
    }
}
```



```
dim3 gridDim = dim3(m/16,n/16,1);
dim3 blockDim = dim3(16, 16, 1);
// warmup run
blur1<<<gridDim, blockDim>>>(image_d,image_out_d,  m, n);


cudaEventRecord(start);
blur1<<<gridDim, blockDim>>>(image_d, image_out_d, m, n);
cudaEventRecord(stop);

cudaEventRecord(start2);
blur2<<<gridDim, blockDim>>>(image_d, image_out_d, m, n);
cudaEventRecord(stop2);
```

Effects of Memory Coalescing

```
cudaEventElapsedTime(&milliseconds, start, stop);
printf("BLUR 1 Time: %f, memory bandwidth %f GB/s\n", milliseconds, 2*sizeof(float)*m*n/milliseconds/1e6);
cudaEventElapsedTime(&milliseconds, start2, stop2);
printf("BLUR 2 Time: %f, memory bandwidth %f GB/s\n", milliseconds, 2*sizeof(float)*m*n/milliseconds/1e6);
```

```
BLUR 1 Time: 0.750592, memory bandwidth 178.815824 GB/s
BLUR 2 Time: 0.208896, memory bandwidth 642.509824 GB/s
```

- RTX3080 Global memory bandwidth is 760 GB/s
- Why 2*sizeof(float)*m*n? Is it not reading 9 copies of the input pixel map?
- Think about the scheduling of blocks to SMs, and working set size of a SM.
- How many threads/blocks can be resident on a SM? What's the working set size of a block/thread? The working set size of a SM? Cache size?

Example: Naïve Matrix Multiplication

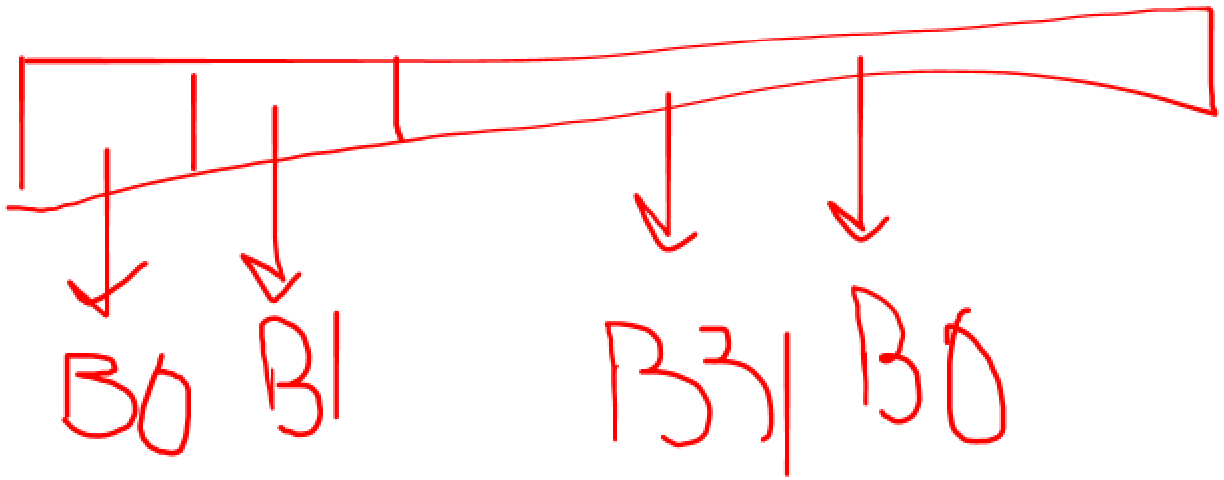
```
// A: m*k, B: k*n, C: m*n; all stored in row major, consecutive in memory
// compute A*B = C
// each thread computes one C[i][j]
__global__ void naive_matmul(float *A,float *B, float *C, int M, int N, int K)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    float c = 0;
    int lda = K, ldb = N, ldc = N;
    for (int k=0; k<K; k++) {
        c += A[i*lda + k] * B[k*ldb + j];
    }
    C[i* ldc + j ] = c;
}
```

- This kernel clearly is bottlenecked by global memory traffic
- Are the accesses coalescing or not?
- If not, how to fix?

#2: Avoid bank conflict in shared memory

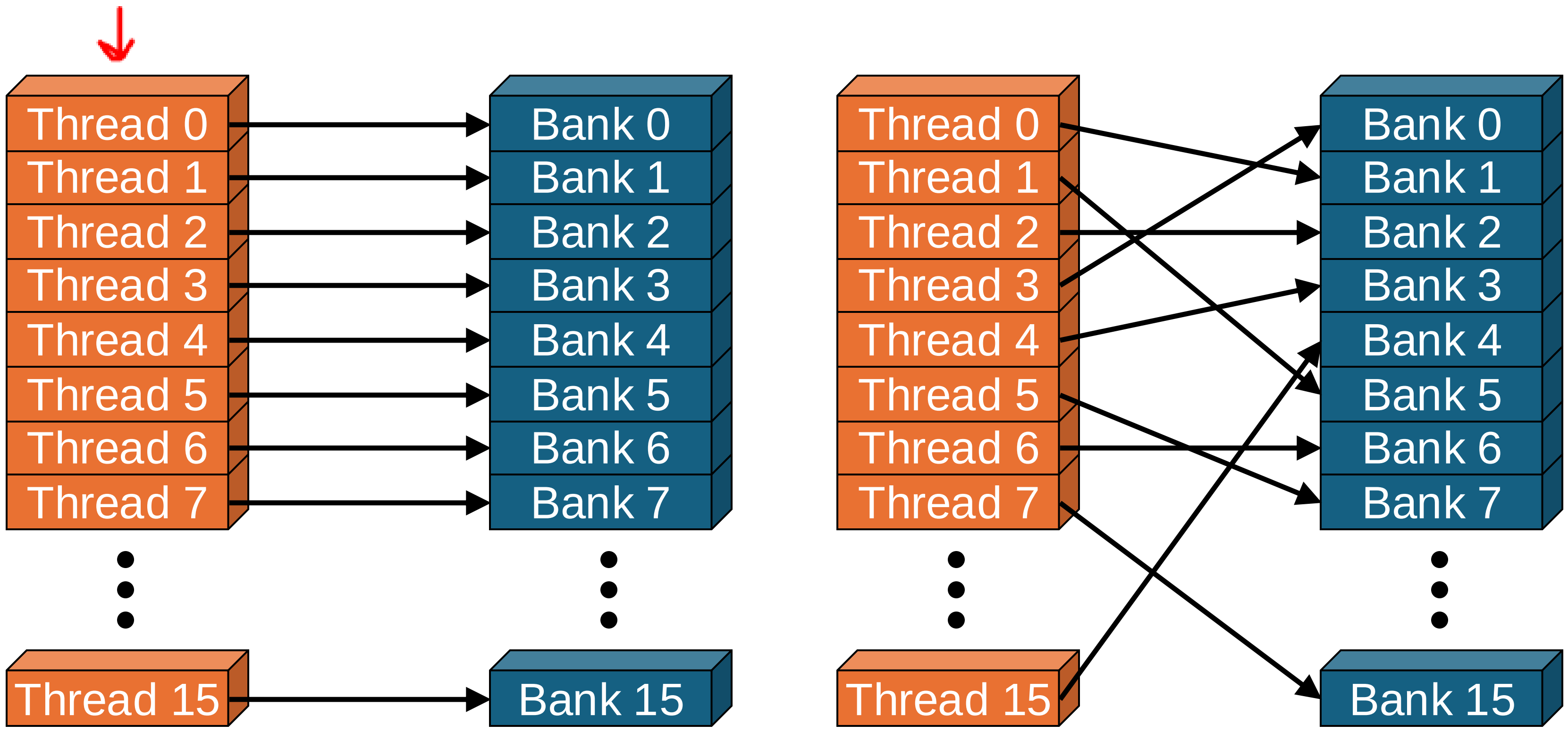
Shared Memory

- Shared memory is an interleaved memory
  - Typically, 32 banks
  - Each bank can service one address per cycle
  - Successive 32-bit words are assigned to successive banks
    - Bank = Address % 32
- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps



Shared Memory

- Bank conflict free



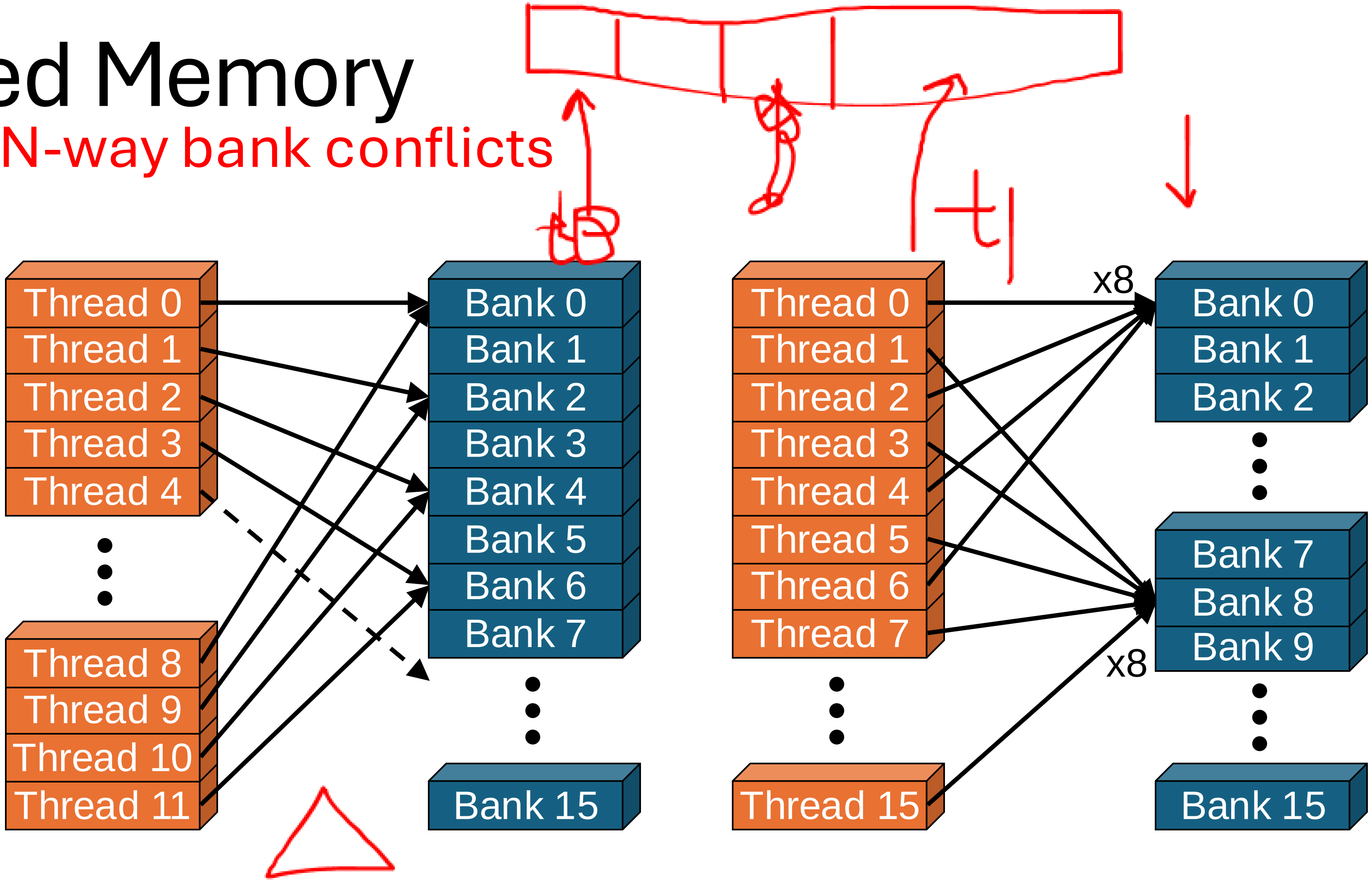
Linear addressing: stride = 1

Random addressing 1:1

Slide credit: Hwu & Kirk

Shared Memory

- N-way bank conflicts



2-way bank conflict: stride = 2

8-way bank conflict: stride = 8

Slide credit: Hwu & Kirk

Example: Microbenchmarking

```
__global__ void kernelNoConflict(int *output, int iterations)
{
    // Declare shared memory as volatile to force a real load each time.
    __shared__ volatile int sdata[BLOCK_SIZE];

    int tid  = threadIdx.x + blockIdx.x * blockDim.x;

    // Initialize shared memory.
    sdata[threadIdx.x] = 1;
    __syncthreads();

    int sum = 0;
    // Repeatedly read from shared memory.
    // Each thread reads its own element (which is in a unique bank within a warp).
    for (int i = 0; i < iterations; i++) {
        sum += sdata[threadIdx.x];
    }

    // Write result to global memory to avoid optimizing away the loop.
    output[tid] = sum;
}
```

```
__global__ void kernelConflict(int *output, int iterations)
{
    // Allocate shared memory large enough so that when using the index below
    // each thread's access lands in the same bank.
    __shared__ volatile int sdata[BLOCK_SIZE * 32];

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    // Compute an index that forces bank conflicts.
    int index = threadIdx.x * 32;

    // Initialize the shared memory element used by this thread.
    sdata[index] = 1;
    __syncthreads();

    int sum = 0;
    // Repeatedly read from the conflict-inducing address.
    for (int i = 0; i < iterations; i++) {
        sum += sdata[index];
    }

    // Write result to global memory.
    output[tid] = sum;
}
```

```
/usr/local/cuda-12.6/bin/nvcc bank_conflict.cu -O2 -arch=sm_86  && ./a.out
Kernel with no bank conflict: 0.943104 ms
Kernel with severe bank conflict: 14.590976 ms
```

Example Scenario: Matrix Multiplication

```
#define TS 32

// block dim 32x32; each thread block computes a 32x32 block matrix in C.
// A: M*K, B: K*N, C: M*N; all row major stored contiguously in memory.
__global__ void MatMulTiled(float *A, float *B, float *C, int M, int N, int K)
{
    __shared__ float As[TS][TS]; // 4KB
    __shared__ float Bs[TS][TS]; // 4KB
    int ldA = K, ldB = N, ldC = N;
    int Bx = blockIdx.x * TS;
    int By = blockIdx.y * TS;

    // perform block inner product of A[rs:re][:] * B[:][cs:ce]
    // ignoring boundry check for now
    for (int k=0; k<(K+TS-1)/TS; k += TS) {
        //step  1: load the A[Bi][Bk] and B[Bk][Bj] into As and Bs
        As[threadIdx.x][threadIdx.y] = A[(Bx+threadIdx.x) * ldA + (k+threadIdx.y)];
        Bs[threadIdx.x][threadIdx.y] = B[(k+threadIdx.x) * ldA + (By+threadIdx.y)];
        __syncthreads();
        //step 2: use As , Bs blocks to accumulate: C += As*Bs
        float Cij = C[Bx+threadIdx.x][By+threadIdx.y];
        for (int kk=0; kk<TS; kk++) {
            Cij += As[threadIdx.x][kk] * Bs[kk][threadIdx.y];
        }
    }

}
```

- This is the tiled MatMul from lec5.
- Now analyze its bank conflicts in shared memory

#3: Thread Coarsening

- In previous examples we mostly decompose the workloads into very fine tasks:
  - Each thread does very little
  - Launch many threads
- Pros:
  - Sufficient (thread) parallelism
  - Easy to start with
- Cons: large overhead, especially fixed cost per thread
  - Registers
  - Redundant loading data/work
  - More synchronization overhead
  - Less data locality
- Solution:
  - Less threads, each doing more
  - A good design approach is to first decompose into fine tasks, and then assign one thread to multiple task

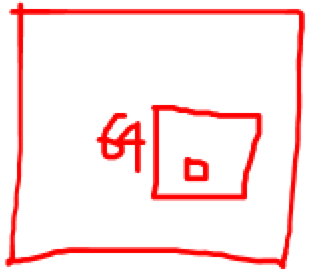
Example: Matrix Multiplication

- We used to assign one thread to one output C[i][j]
- To coarsen threading, we assign one thread to say 4 outputs; they could be contiguous chunk or distributed around.
- So a 16x16 thread block will handle say 32x32 matrix block
- Benefits?
  - Might have huge benefit in data reuse **in registers**
  - Thread level parallelism—pipelining
  - This is analogous to **loop unrolling** in serial programs, this can be the single most effective technique in unblocking compiler/hardware to do a bunch of optimizations

MatMul: Tiled Version (Old)

```
#define TS 32

// block dim 32x32; each thread block computes a 32x32 block matrix in C.
// A: M*K, B: K*N, C: M*N; all row major stored contiguously in memory.
__global__ void MatMulTiled(float *A, float *B, float *C, int M, int N, int K)
{

    __shared__ float As[TS][TS]; // 4KB
    __shared__ float Bs[TS][TS]; // 4KB
    int ldA = K, ldB = N, ldC = N;

    int Bx = blockIdx.x * TS;
    int By = blockIdx.y * TS;

    // perform block inner product of A[Bx:Bx+TS][:] * B[:][By:By+TS]
    // ignoring boundary check for now
    float Cij = 0.0f;
    for (int k = 0; k < (K + TS - 1) / TS; k += TS) {
        // step 1: load the A[Bi][Bk] and B[Bk][Bj] into As and Bs
        As[threadIdx.x][threadIdx.y] = A[(Bx + threadIdx.x) * ldA + (k + threadIdx.y)];
        Bs[threadIdx.x][threadIdx.y] = B[(k + threadIdx.x) * ldA + (By + threadIdx.y)];
        __syncthreads();

        // step 2: use As, Bs blocks to accumulate: C += As*Bs
        for (int kk = 0; kk < TS; kk++)
            Cij += As[threadIdx.x][kk] * Bs[kk][threadIdx.y];
    }
    C[Bx + threadIdx.x][By + threadIdx.y] = Cij;
}
```

MatMul: Tiled Version (thread coarsened)




```
#define TS 32

__global__ void MatMulTiled(float *A, float *B, float *C, int M, int N, int K) {
    __shared__ float As[64][TS]; // 64x32 shared memory for A
    __shared__ float Bs[TS][64]; // 32x64 shared memory for B
    int ldA = K, ldB = N, ldC = N;

    // Block handles 64x64 C tile
    int Bx = blockIdx.x * 64;
    int By = blockIdx.y * 64;

    int tx = threadIdx.x; // 0-31
    int ty = threadIdx.y; // 0-31

    // 2x2 accumulators
    float C00 = 0.0f, C01 = 0.0f, C10 = 0.0f, C11 = 0.0f;

    for (int k = 0; k < K; k += TS) {...}

    // Write 2x2 block to C
    int row = Bx + 2*tx;
    int col = By + 2*ty;
    C[row * ldC + col] = C00;
    C[row * ldC + col + 1] = C01;
    C[(row + 1) * ldC + col] = C10;
    C[(row + 1) * ldC + col + 1] = C11;
}
```

```
for (int k = 0; k < K; k += TS) {
    // Load 64x32 A tile into As
    As[2*tx][ty] = A[(Bx + 2*tx) * ldA + (k + ty)];
    As[2*tx + 1][ty] = A[(Bx + 2*tx + 1) * ldA + (k + ty)];

    // Load 32x64 B tile into Bs
    Bs[tx][2*ty] = B[(k + tx) * ldB + (By + 2*ty)];
    Bs[tx][2*ty + 1] = B[(k + tx) * ldB + (By + 2*ty + 1)];

    __syncthreads();

    // Compute 2x2 block
    for (int kk = 0; kk < TS; kk++) {
        float a0 = As[2*tx][kk];
        float a1 = As[2*tx + 1][kk];
        float b0 = Bs[kk][2*ty];
        float b1 = Bs[kk][2*ty + 1];

        C00 += a0 * b0;
        C01 += a0 * b1;
        C10 += a1 * b0;
        C11 += a1 * b1;
    }

    __syncthreads();
}
```

Discussion: Pros vs Cons

- Reduction to global memory traffic?
- Increased data reuse in shared memory?
- Increased data reuse in register file?

Cons:

- Reduced # threads (occupancy?)
- Increased register pressure?

```cuda
// A: MxK, B: KxN, C: MxN; all stored in row-major order
// Computes C = A * B with thread coarsening: each thread computes a 2x2 block.
__global__ void coarsened_matmul2x2(float *A, float *B, float *C, int M, int N, int K)
{
    // Each thread computes a 2x2 block.
    // Compute the top-left index of the 2x2 block.
    int i_base = (blockIdx.y * blockDim.y + threadIdx.y) * 2;
    int j_base = (blockIdx.x * blockDim.x + threadIdx.x) * 2;

    // Accumulators for the 2x2 block.
    float c00 = 0.0f, c01 = 0.0f, c10 = 0.0f, c11 = 0.0f;
    int lda = K, ldb = N, ldc = N;

    // Loop over the K dimension.
    for (int k = 0; k < K; k++) {
        // Load elements from A if within bounds.
        float a0 = (i_base < M) ? A[i_base * lda + k] : 0.0f;
        float a1 = ((i_base + 1) < M) ? A[(i_base + 1) * lda + k] : 0.0f;

        // Load elements from B if within bounds.
        float b0 = (j_base < N) ? B[k * ldb + j_base] : 0.0f;
        float b1 = ((j_base + 1) < N) ? B[k * ldb + j_base + 1] : 0.0f;

        // Multiply and accumulate for the 2x2 outputs.
        if (i_base < M && j_base < N)
            c00 += a0 * b0;
        if (i_base < M && (j_base + 1) < N)
            c01 += a0 * b1;
        if ((i_base + 1) < M && j_base < N)
            c10 += a1 * b0;
        if ((i_base + 1) < M && (j_base + 1) < N)
            c11 += a1 * b1;
    }

    // Write the results back to C with proper boundary checks.
    if (i_base < M && j_base < N)
        C[i_base * ldc + j_base] = c00;
    if (i_base < M && (j_base + 1) < N)
        C[i_base * ldc + j_base + 1] = c01;
    if ((i_base + 1) < M && j_base < N)
        C[(i_base + 1) * ldc + j_base] = c10;
    if ((i_base + 1) < M && (j_base + 1) < N)
        C[(i_base + 1) * ldc + j_base + 1] = c11;
}
```

```
/usr/local/cuda-12.6/bin/nvcc -O2 -arch=sm_86 matmul.cu  && ./a.out
naive matmul 1 Time: 579.636230, memory bandwidth 0.926220 GB/s, GFLOPS: 1896.899456
coarsened matmul 1 Time: 312.706177, memory bandwidth 1.716854 GB/s, GFLOPS: 3516.117504

Compilation finished at Fri Feb 14 23:45:07
```

Checklist

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/cache lines	Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	–	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data and then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily

Assignment1: Matrix Transpose

- A great exercise to apply the performance considerations discussed so far. Particularly:
  - Coalescing global memory access
  - Use the shared memory (not necessarily to improve data reuse, but rather to achieve Coalescing)
  - Thread coarsening (loop unrolling) to further improve performance
- Two kernels:
  - shmemTransposeKernel: use of shared memory to achieve coalescing
  - optimalTransposeKernel: use thread coarsening
- Goal: approach or (exceed!) the vendor optimized routine memcpy().