CS 6023 - GPU Programming

Adv. Memory Tiling, CUDA Events

29/08/2018

Agenda

- Tiling
    - Tile size,
    - Irregular matrices,
    - Control divergence,
    - Memory access patterns
- CUDA events

Acknowledgement: Nvidia teaching kit

Announcement

Quiz on 5th Sept Wed at 4:50pm. Any concerns with scheduling?

Objective questions (25 questions for 40 mins)

Syllabus: Topics till this lecture (Lec 6)

Slides will be up by today

The CUDA kernel for tiled matrix multiplication

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;

  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < Width/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
        Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Choosing the tile size

- We have seen how to choose the block, threads based on number of threads / SM, blocks / SM and threads / warp. Now we have an additional parameter tile size
- For tile of size 16 x 16

- For tile of size 32 x 32

Choosing the tile size

- We have seen how to choose the block, threads based on number of threads / SM, blocks / SM and threads / warp. Now we have an additional parameter tile size
- For tile of size 16 x 16

  Number of loads from global memory in each phase = 2 x 16 x 16 = 512 floats

  Number of addition / multiplication operations in each phase = 256 x 16 x 2 = 8,192 ops

  Computational intensity = 8,192 / 512 = 16 ops/float
- For tile of size 32 x 32

Choosing the tile size

- We have seen how to choose the block, threads based on number of threads / SM, blocks / SM and threads / warp. Now we have an additional parameter tile size
- For tile of size 16 x 16

  Number of loads from global memory in each phase = 2 x 16 x 16 = 512 floats

  Number of addition / multiplication operations in each phase = 256 x 16 x 2 = 8,192 ops

  Computational intensity = 8,192 / 512 = 16 ops/float
- For tile of size 32 x 32

  Number of loads from global memory in each phase = 2 x 32 x 32 = 2,048 floats

  Number of addition / multiplication operations in each phase = 1024 x 32 x 2 = 65,536 ops

  Computational intensity = 65,536 / 2048 = 32 ops/float

Choosing the tile size

- We have seen how to choose the block, threads based on number of threads / SM, blocks / SM and threads / warp. Now we have an additional parameter tile size
- For tile of size 16 x 16

  Number of loads from global memory in each phase = 2 x 16 x 16 = 512 floats

  Number of addition / multiplication operations in each phase = 256 x 16 x 2 = 8,192 ops

  Computational intensity = 8,192 / 512 = 16 ops/float
- For tile of size 32 x 32

  Number of loads from global memory in each phase = 2 x 32 x 32 = 2,048 floats

  Number of addition / multiplication operations in each phase = 1024 x 32 x 2 = 65,536 ops

  Computational intensity = 65,536 / 2048 = 32 ops/float
- Easy to show that for T x T tile, the computational intensity is T ops/float

Choosing the tile size

- But, the full picture is much more complicated

Choosing the tile size

- But, the full picture is much more complicated
- Already seen, that 32x32 tile requires 1,024 threads and if we have 1,536 max. threads per core then we can have only 1 block! For 16x16 tile we can have 6 blocks covering the full 1,536 threads
- Why do we want more threads in an SM?
- But could we want less threads in a block?

Choosing the tile size

- But, the full picture is much more complicated
- Already seen, that 32x32 tile requires 1,024 threads and if we have 1,536 max. threads per core then we can have only 1 block! For 16x16 tile we can have 6 blocks covering the full 1,536 threads
- Why do we want more threads in an SM?
  - In the load stage of any phase, each thread wants to read two floats
  - Thus, for an SM we have 2 * 256 * 6 = 3,072 pending loads => Hide latency
- But could we want less threads in a block?

Choosing the tile size

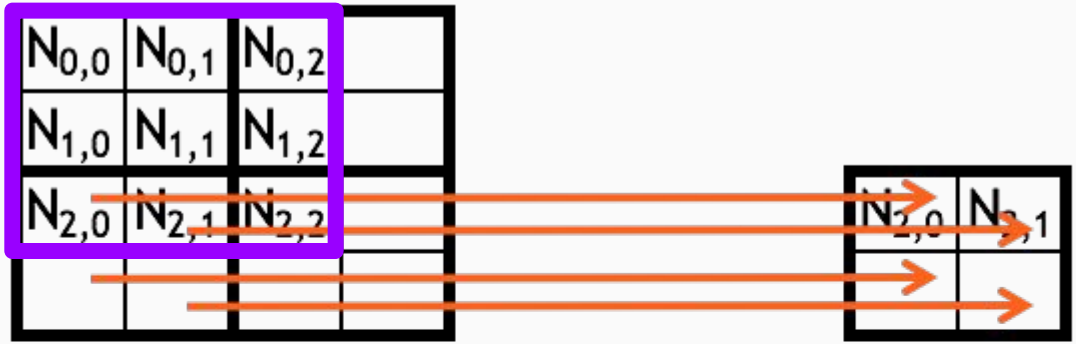
- But, the full picture is much more complicated
- Already seen, that 32x32 tile requires 1,024 threads and if we have 1,536 max. threads per core then we can have only 1 block! For 16x16 tile we can have 6 blocks covering the full 1,536 threads
- Why do we want more threads in an SM?
  - In the load stage of any phase, each thread wants to read two floats
  - Thus, for an SM we have 2 * 256 * 6 = 3,072 pending loads => Hide latency
- But could we want less threads in a block?
  - `__syncthreads()` adds synchronization which suffers from Amdahl's law => Fewer threads means lesser chance of threads waiting

Working with arbitrary matrix sizes

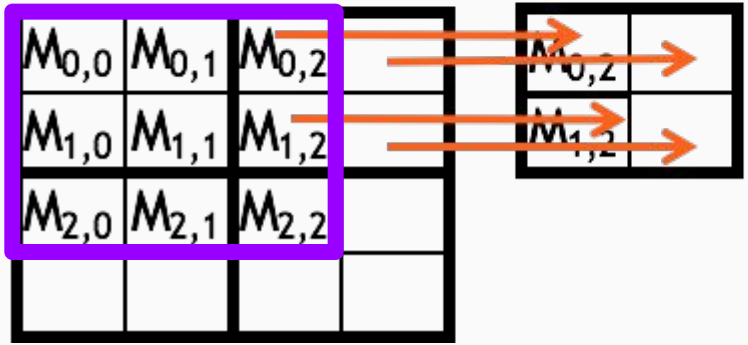
- So far, looked at square matrices, where tile size (= block size) is a divisor of the matrix dimension
- In general, these will not be true
- Easy extension to rectangular matrices (you will work on it in the assignment)
- Let’s look at non-divisor tile size
- One option: Pad matrix with zeros to make its size a multiple of the tiles
  - Actually not a bad option
  - Main downside is memory bandwidth
- We will look at modifying the compute kernels instead

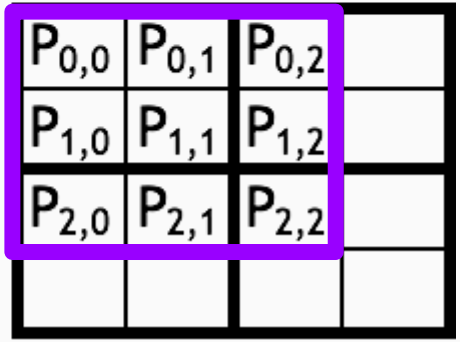
Tiling with arbitrary size

3x3 matrix
Phase 1 loads for block (0,0)



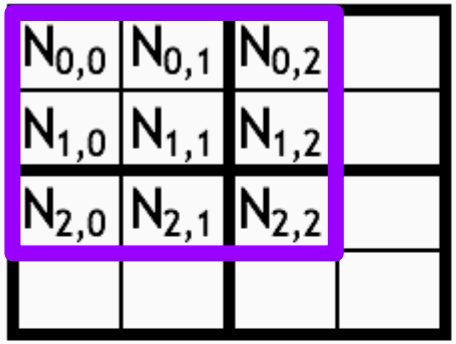
Thread (0,0) and (0,1) need different load instructions

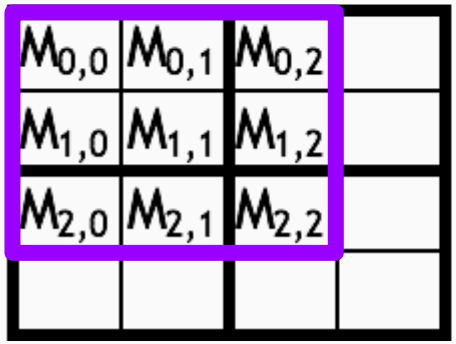


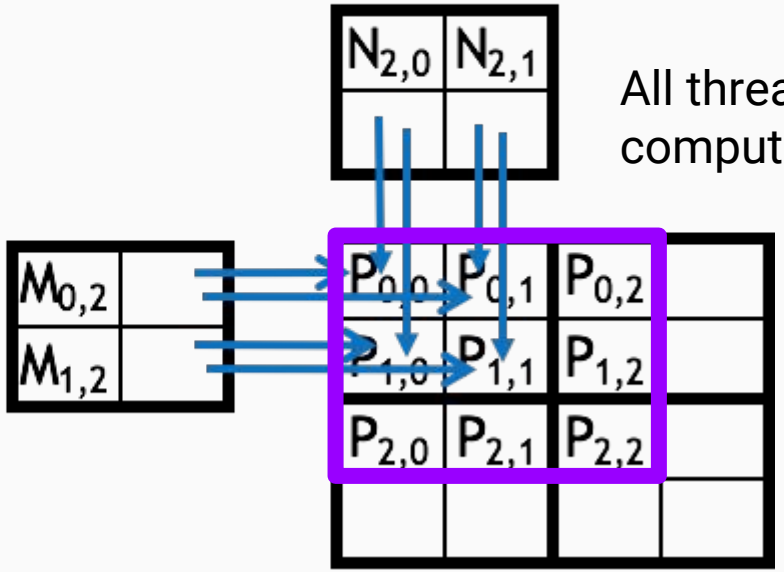


Tiling with arbitrary size

3x3 matrix
Phase 1, compute iteration 1 for block (0,0)







All threads need different compute blocks

Different corner cases

Two classes of threads in corner cases:

- Threads that do not calculate valid P elements but still need to participate in loading the input tiles
  - Phase 0 of Block(1,1), Thread(1,0), assigned to calculate non-existent P[3,2] but needs to participate in loading tile element N[1,2]
- Threads that calculate valid P elements may attempt to load non-existing input elements when loading input tiles
  - Phase 1 of Block(0,0), Thread(1,0), assigned to calculate valid P[1,0] but attempts to load non-existing N[3,0]

An easy solution that works for matrix multiplication

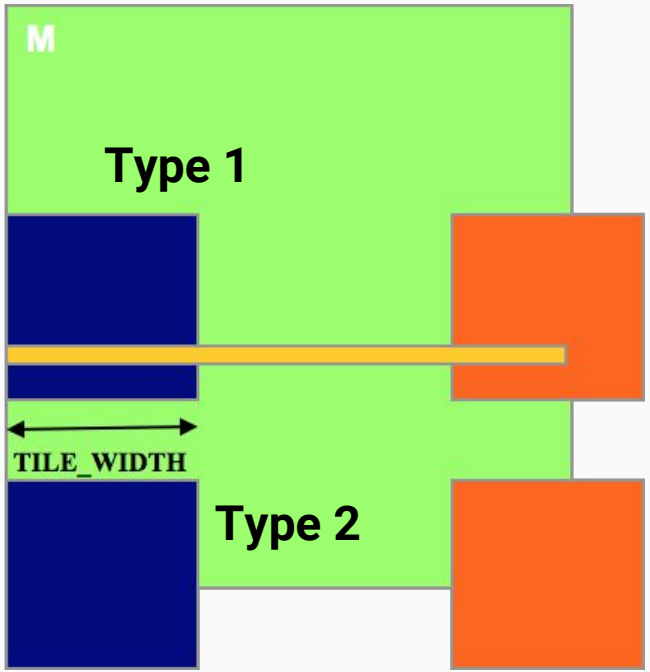
- When loading a value, check if the range is valid
  - If not, simply load 0
  - For reading `M[Row][p*TILE_WIDTH+tx]` need to test:

    `(Row < Width) && (p*TILE_WIDTH+tx < Width)`
  - For reading `N[p*TILE_WIDTH+ty][Col]` need to test:

    `(p*TILE_WIDTH+ty < Width) && (Col < Width)`
- When writing a value to product, check if the range is valid
  - If not, don't write
  - For writing `P[Row][Col]` need to test: `Row < Width && Col < Width`
- Different conditions for loading M, N, and for writing P => **Code in assignment**

Control divergence with irregular sizes

- The if conditions introduced will lead to control divergence of warps
- Useful exercise to compute the maximum cost of such divergence
- Let us just consider the case of loading M
- Define two types of blocks
  - Type 1: Blocks whose tiles have valid range until the last phase
  - Type 2: Blocks whose tiles never have valid range

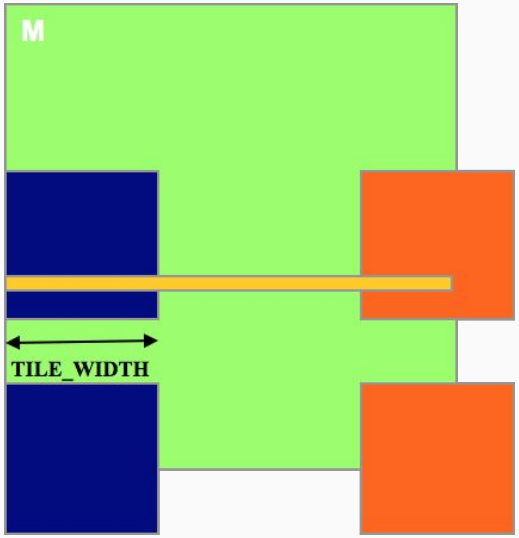


Sample calculation

- Consider a matrix of size 100x100, tiles and blocks of size 16x16

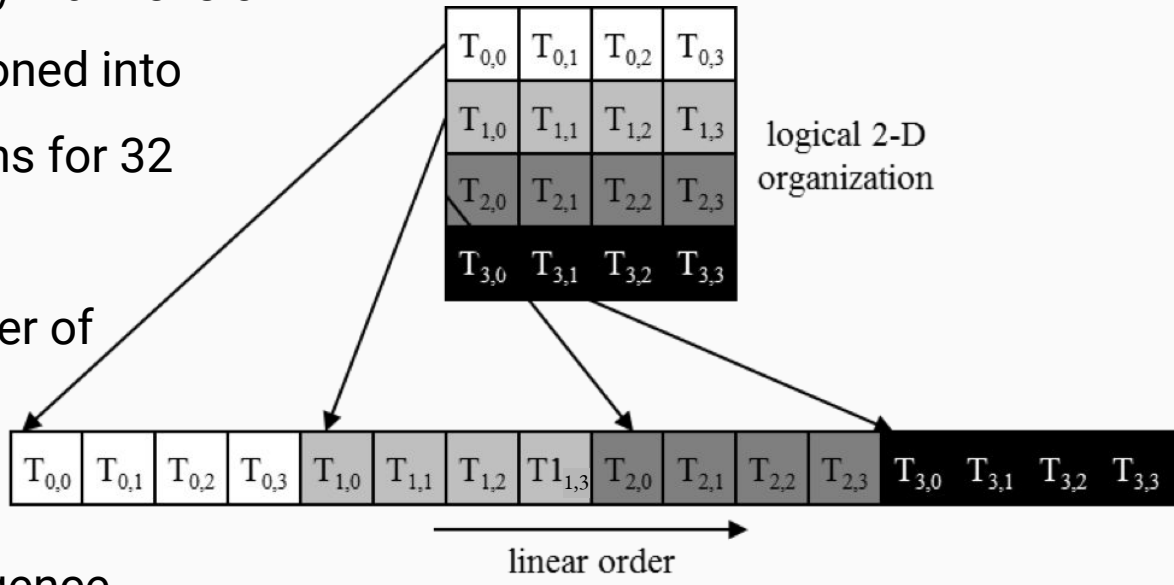
- There are a total of 49 (= square(ceil(100/16))) thread blocks
- Each block has 256 (=16x16) threads and 8 (=256/32) warps
- Each thread goes through 7 (= ceil(100/16)) phases

- How many Type 1 and Type 2 blocks?
- How many of these have warp phases with divergence?



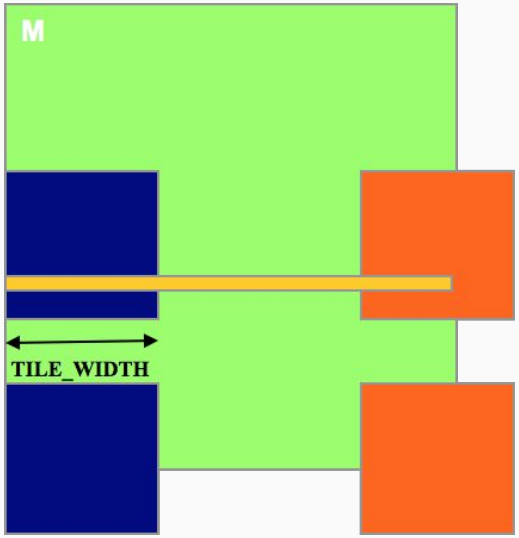
How are warps organized?

- Each block is split in to warps of size 32, but in what order?
- The thread blocks are linearized into one-dimensional order in row-major order
  - First x, then y, and finally z dimension
- Then, the threads are partitioned into warps in consecutive portions for 32
- This partitioning scheme is consistent though the number of threads / warp may change
- Useful in understanding and controlling thread divergence



Sample calculation

- Type 1 blocks are 42 (=6 x 7), with 336 (=42x8) warps, with 2,352 (=336x7) phases
- Control divergence only in the last phase = 336 warp phases

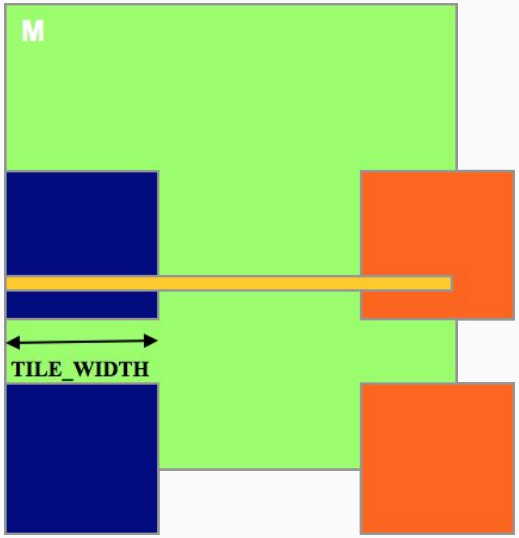


Sample calculation

- Type 1 blocks are 42 (=6 x 7), with 336 (=42x8) warps, with 2,352 (=336x7) phases
- Control divergence only in the last phase = 336 warp phases

- Type 2 blocks are 7 with 56 (=7x8) warps, with 392 (=56x7) phases
- The last 6 warps have no divergence. The first two have divergence only for the last phase
- So, only 2 x 7 = 14 warp phases have divergence

- Total performance impact = (336+14)/(2,352+392) = 12.7 %



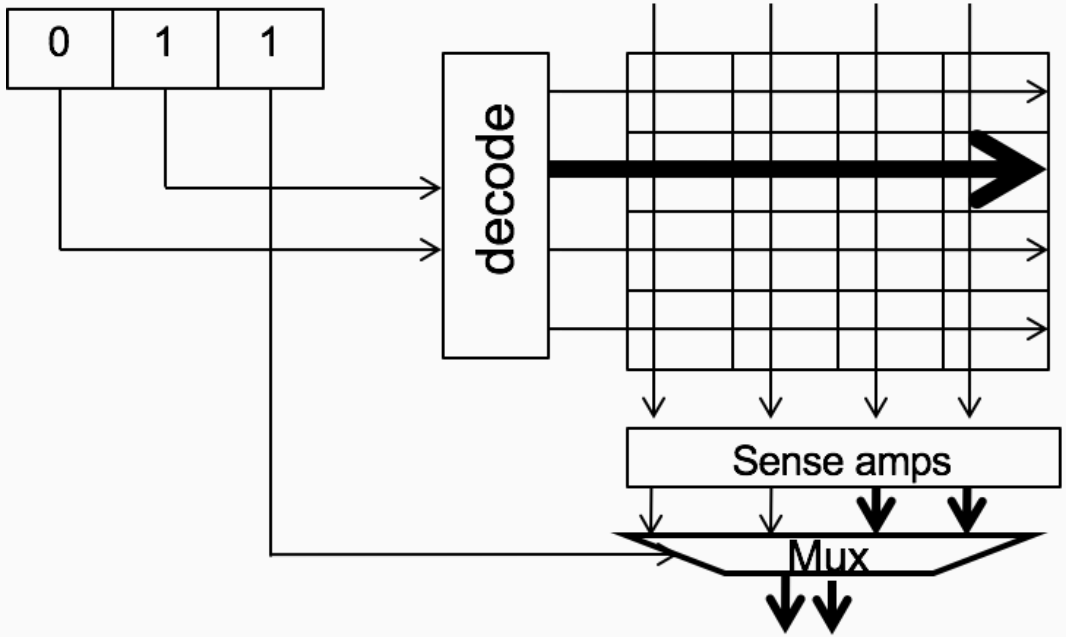
Insight

- Even if there are conditional statements, we may not have performance overhead with control divergence, due to the way the warps are assigned
- Arrange your threads with control divergence in mind

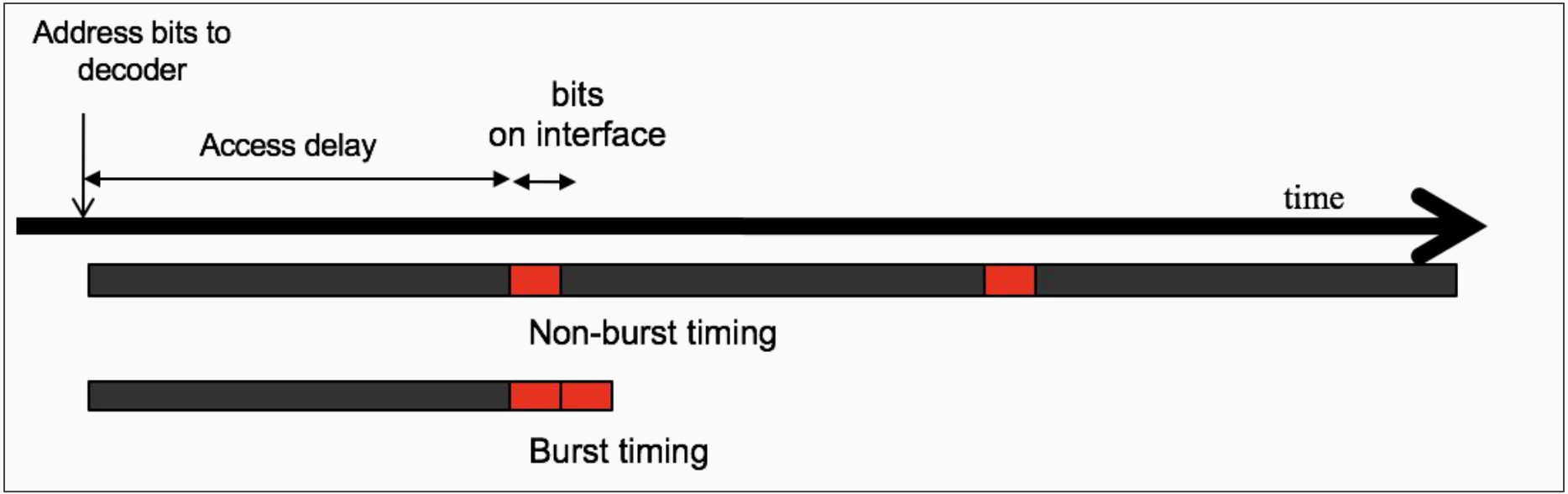
- Control divergence from boundary conditions are typically insignificant (due to large matrix sizes)
- We will see some examples of problems where control divergence is more natural and optimizing that will give significant improvement in performance

DRAM

- DRAM stores bits in tiny capacitors arranged in rows and columns
- They are slow!
- DDR3/GDDR4 has 1/8th clock of the bus => Need to read 8 x bus_width bits from the same row to internal buffer



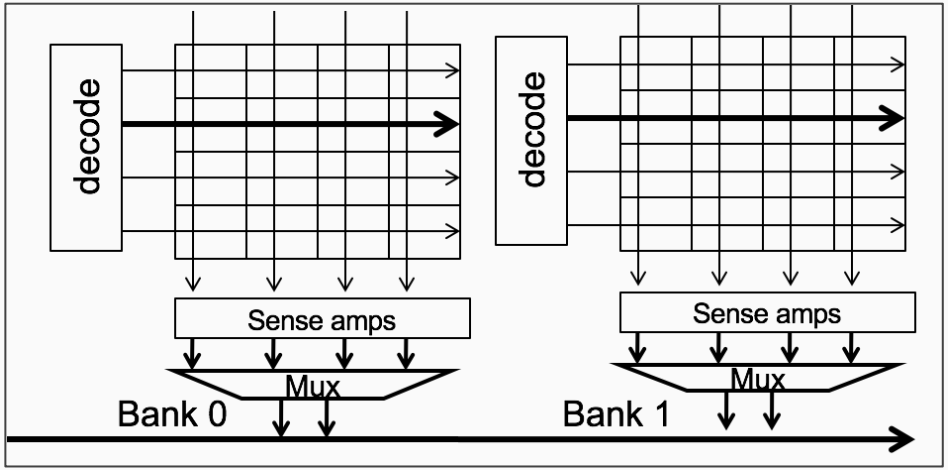
Access timing



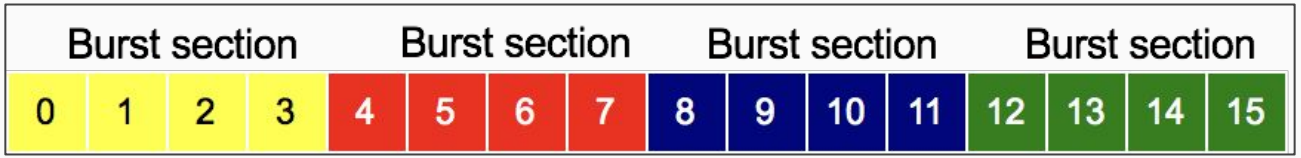
- Need to be able to read multiple values from same row to benefit from burst timing

Multiple DRAM banks

- Second optimization is to have banked memories and access



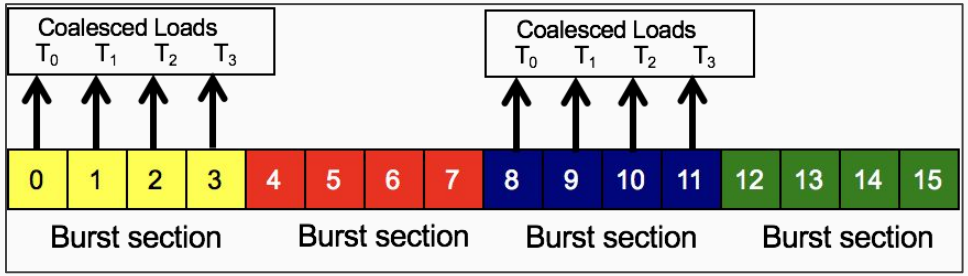
Burst sections



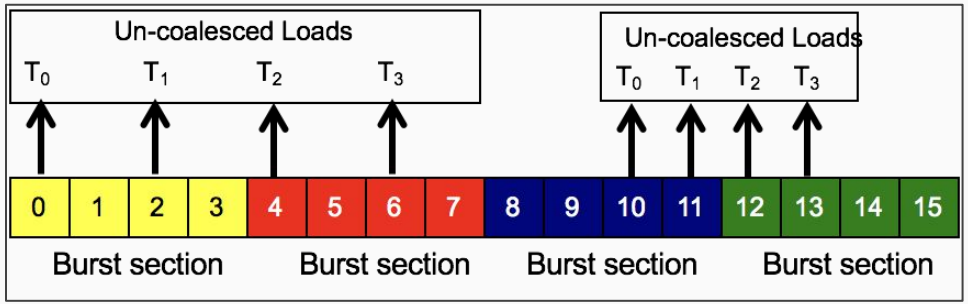
- Address space is partitioned into burst sections
- In these examples we will use a 16 byte address space with burst sections which are 4 bytes
- In real cases, we have address spaces in order of GBs and burst sections in orders of hundreds of KBs

Coalesced loads

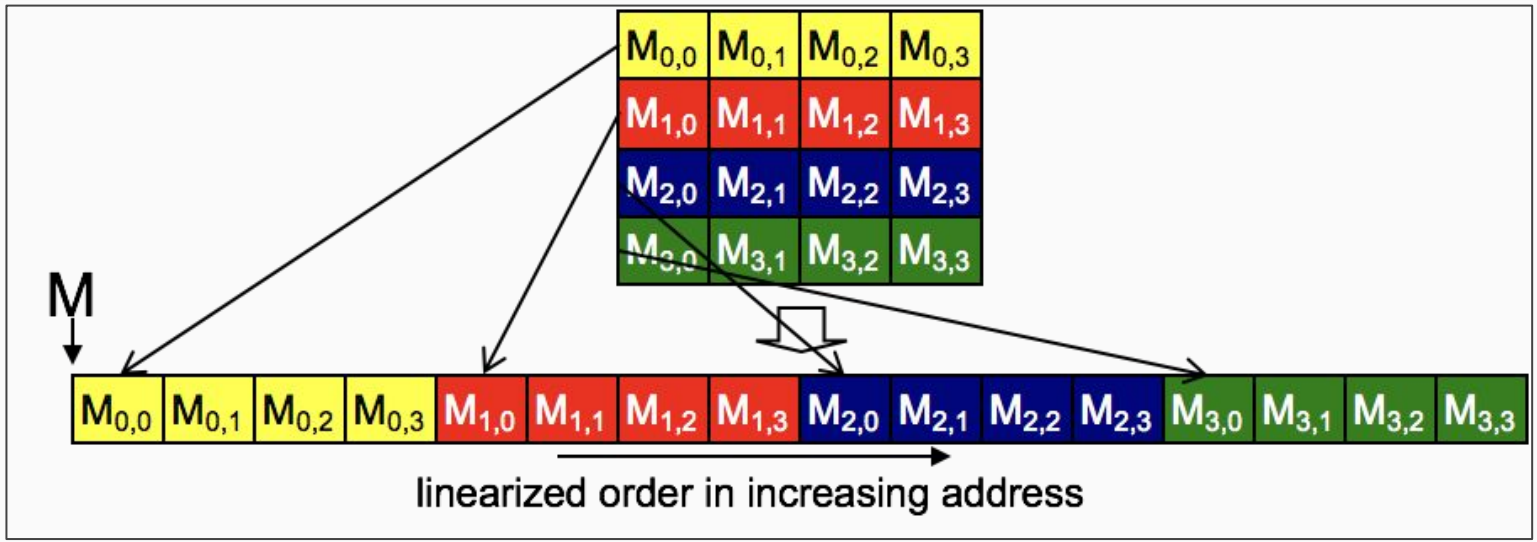
Good



Bad

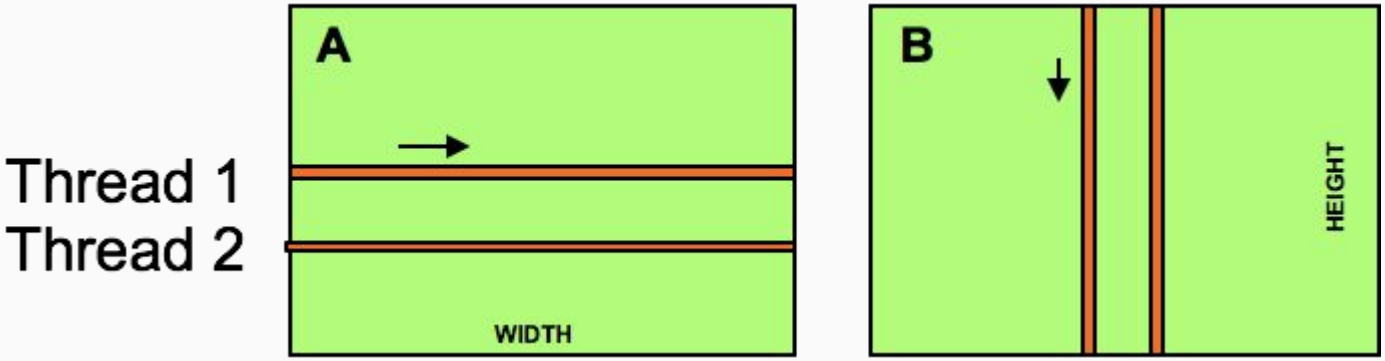


Matrix multiplication



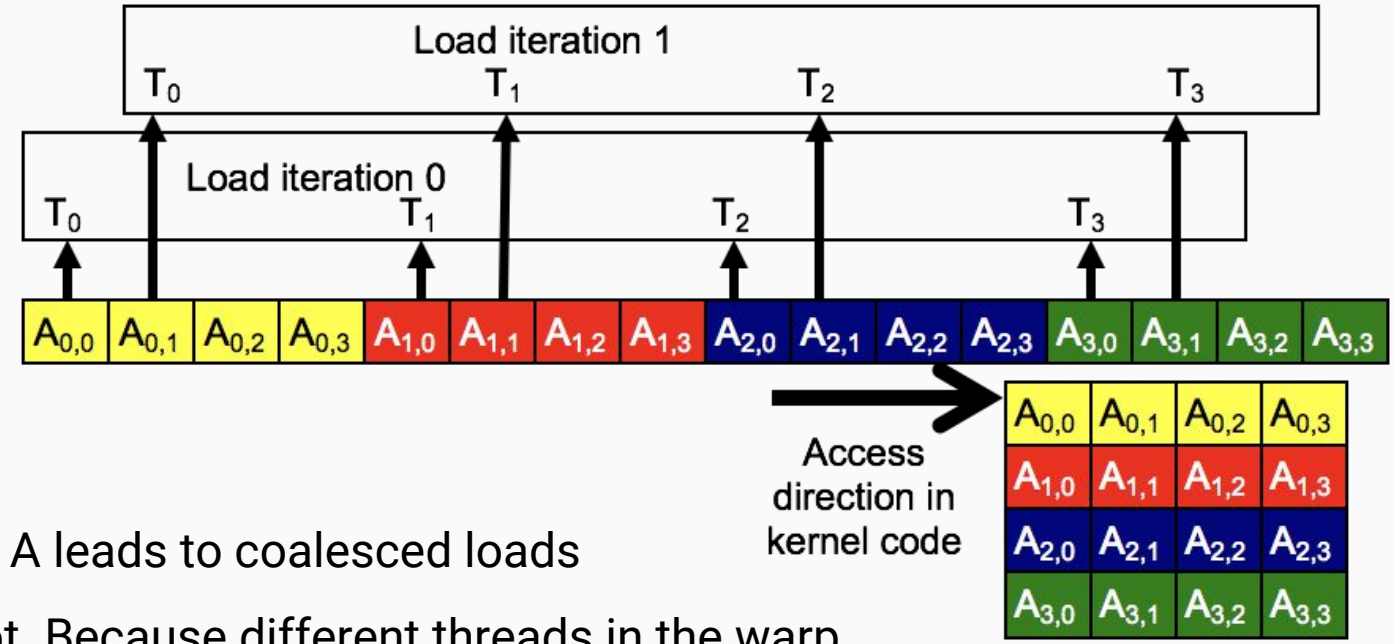
- Toy example of mapping a 4x4 matrix to our address space and burst sections
- Matrix is stored in row major order

Non-tiled matrix multiplication



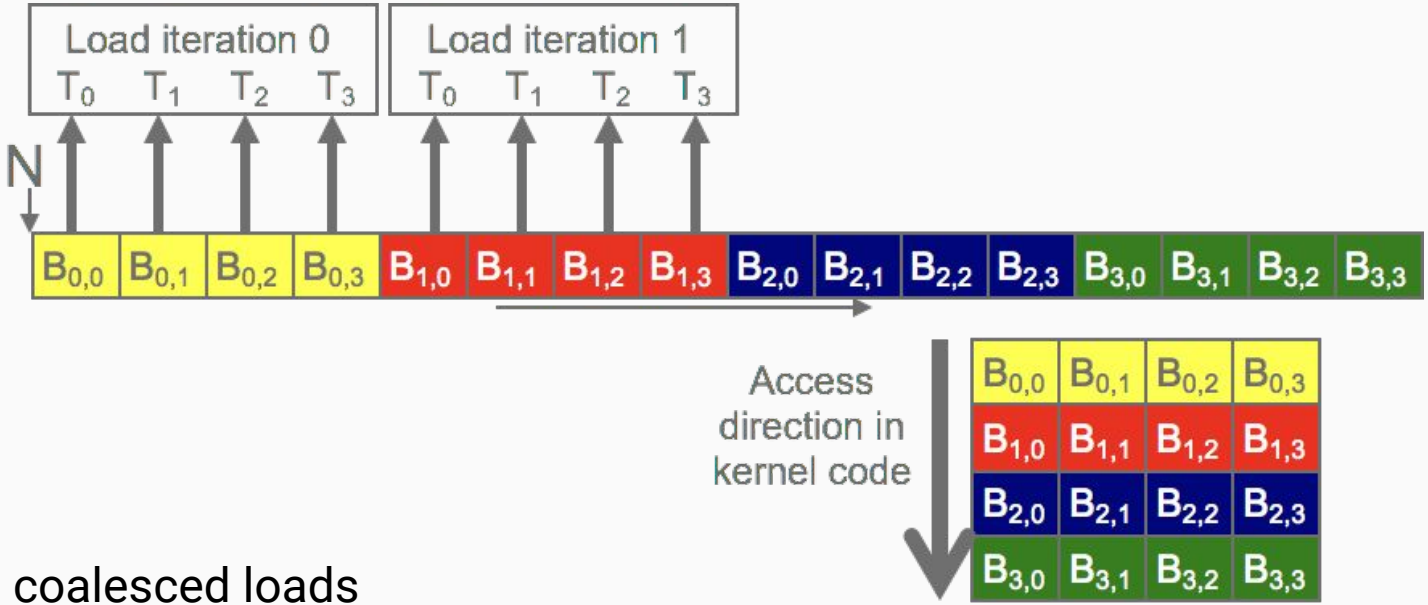
- Let’s say we are computing A x B
- A is accessed row wise and B in column wise
- Which access pattern benefits from memory coalescing?

Access pattern of A



- We may think A leads to coalesced loads
- But it does not. Because different threads in the warp access different burst sections in the same lock-step
- Notice here the difference between locality in serial programs vs a GPU kernel

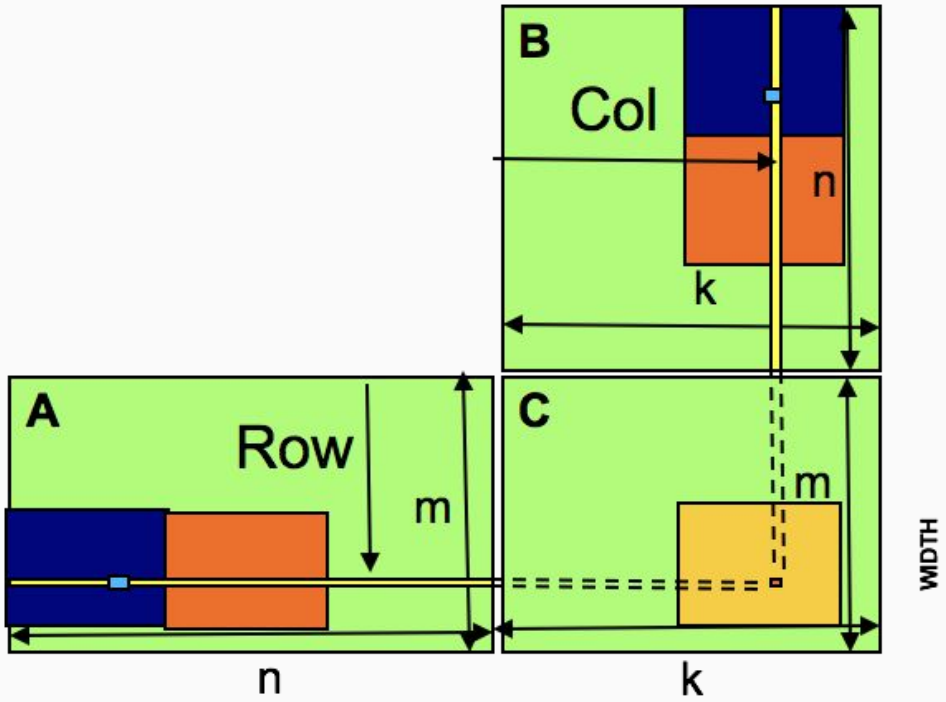
Access pattern of B



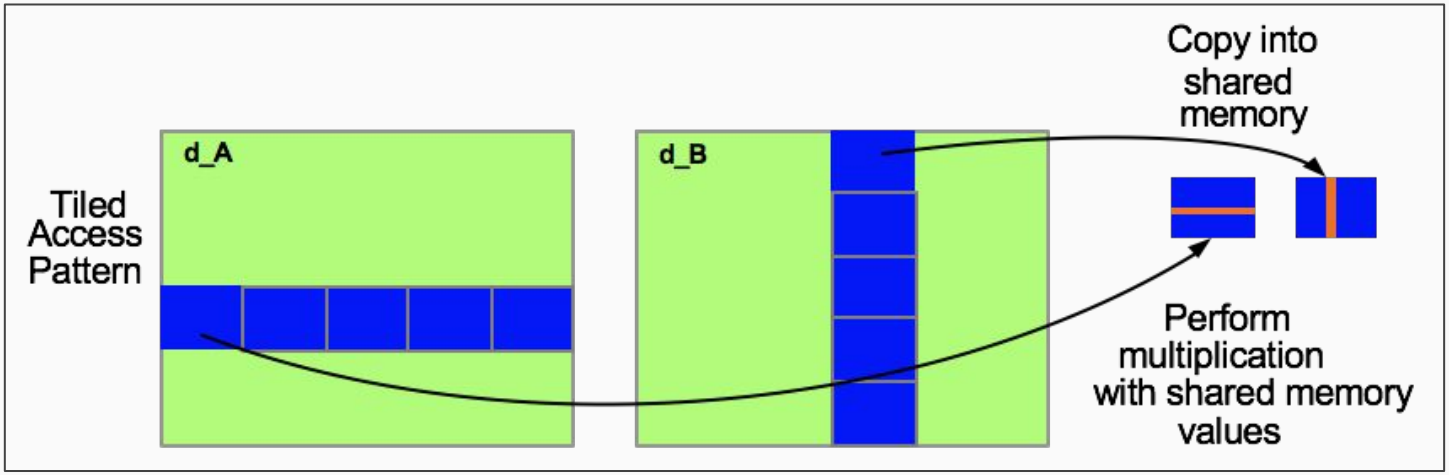
- Accessing B has coalesced loads

What happens with tiling?

- Reading A is uncoalesced and B is coalesced
- How can we improve this?
- Turns out tiling (which had other benefits) also solves this problem for us with no additional change



What happens with tiling



- When we load tiles to the shared memory, threads of the same warp are accessing contiguous memory locations (modulo boundaries)
- Thus we benefit from burst accesses
- Again, this benefit increases with tile size

Tiling as a memory pattern

- In most applications, need to increase computational intensity. One option is to employ shared memory in case of data reuse
- Tiling as a memory pattern works if threads have matching temporal and spatial requirements in their memory accesses
- Share the task of loading data across threads and cooperatively work on loaded data
- Several considerations in choosing the right tile size
- Multiplicative effect on performance due to both increase computational intensity and DRAM coalesced load pattern

CUDA Events

Timing CUDA program with CPU timers

```
cudaMemcpy(d_m, m, N*N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_n, n, N*N*sizeof(float), cudaMemcpyHostToDevice);
... // Region A
matrix_multiply<<<... , ...>>>(..., ...);
... // Region B
cudaMemcpy(p, d_p, N*N*sizeof(float), cudaMemcpyDeviceToHost);
... // Region C
```

- The cudaMemcpy functions are blocking
  - Kernel invocation does not happen until m and n have been sent to device
  - Similarly for copying p to host
- Thus, `region B` gets executed concurrently with the kernel execution, whereas `region C` waits for the memCpy

Timing CUDA program with CPU timers

```
cudaMemcpy(d_m, m, N*N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_n, n, N*N*sizeof(float), cudaMemcpyHostToDevice);
t1 = myCPUTimer();
matrix_multiply<<<... , ...>>>(..., ...);
t2 = myCPUTimer();
cudaMemcpy(p, d_p, N*N*sizeof(float), cudaMemcpyDeviceToHost);
```

- Example cpu timers: `gettimeofday` or `clock_gettime`
- Will this work for measuring the time taken by the kernel?

Timing CUDA program with CPU timers

```
t1 = myCPUTimer();
cudaMemcpy(d_m, m, N*N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_n, n, N*N*sizeof(float), cudaMemcpyHostToDevice);
matrix_multiply<<<... , ...>>>(..., ...);
cudaMemcpy(p, d_p, N*N*sizeof(float), cudaMemcpyDeviceToHost);
t2 = myCPUTimer();
```

- Will this work for measuring the time taken by the kernel and memcopy?

Timing CUDA program with CPU timers

```
cudaMemcpy(d_m, m, N*N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_n, n, N*N*sizeof(float), cudaMemcpyHostToDevice);
t1 = myCPUTimer();
matrix_multiply<<<... , ...>>>(..., ...);
cudaDeviceSynchronize();
t2 = myCPUTimer();
cudaMemcpy(p, d_p, N*N*sizeof(float), cudaMemcpyDeviceToHost);
```

- Explicit synchronization between host and device
- `cudaDeviceSynchronize` blocks all CPU execution until all pending device commands have been executed
- Limitation => stalls GPU execution

CUDA events

- CUDA events API provides a lightweight alternative to CPU timers
- Allows creation, destruction, recording, and computation of elapsed time
- CUDA events are based on a concept called CUDA streams which we will study later in the course (~ mid Sept)
- Provides resolution with errors of about half a micro-second

Timing CUDA program with CUDA events

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(d_m, m, N*N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_n, n, N*N*sizeof(float), cudaMemcpyHostToDevice);

cudaEventRecord(start);
matrix_multiply<<<... , ...>>>(..., ...);
cudaEventRecord(stop);

cudaMemcpy(p, d_p, N*N*sizeof(float), cudaMemcpyDeviceToHost);

float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

Will this work?

Timing CUDA program with CUDA events

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(d_m, m, N*N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_n, n, N*N*sizeof(float), cudaMemcpyHostToDevice);

cudaEventRecord(start);
matrix_multiply<<<... , ...>>>(..., ...);
cudaEventRecord(stop);

cudaMemcpy(p, d_p, N*N*sizeof(float), cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```