Memory access

Vital for performance!

Memory types

Coalescing

Example of using shared memory

Memory types

Global

Shared

Constant (read only)

Texture cache (read only)

Local

Registers

Care about these when optimizing - not to begin with

Global memory

400-600 cycles latency!

Very large, gigabytes

Main storage

Coalesce memory access!

Continuous
Aligned on power of 2 boundary
Addressing follows thread numbering

Shared memory

Fast temporary storage

Accessible by all threads in a block

Use shared memory for reorganizing data for faster access and coalescing!

Important!

Constant memory

Part of global

Some possibilities for speedups by broadcasting.

Texture memory

Part of global

Special access features:

Interpolation

Edge tests

Registers

Very fast

Private storage for each thread

Local memory

Actually not local!

Part of global memory when you run out of registers!

Slow!

Using shared memory to reduce number of global memory accesses

Read blocks of data to shared memory

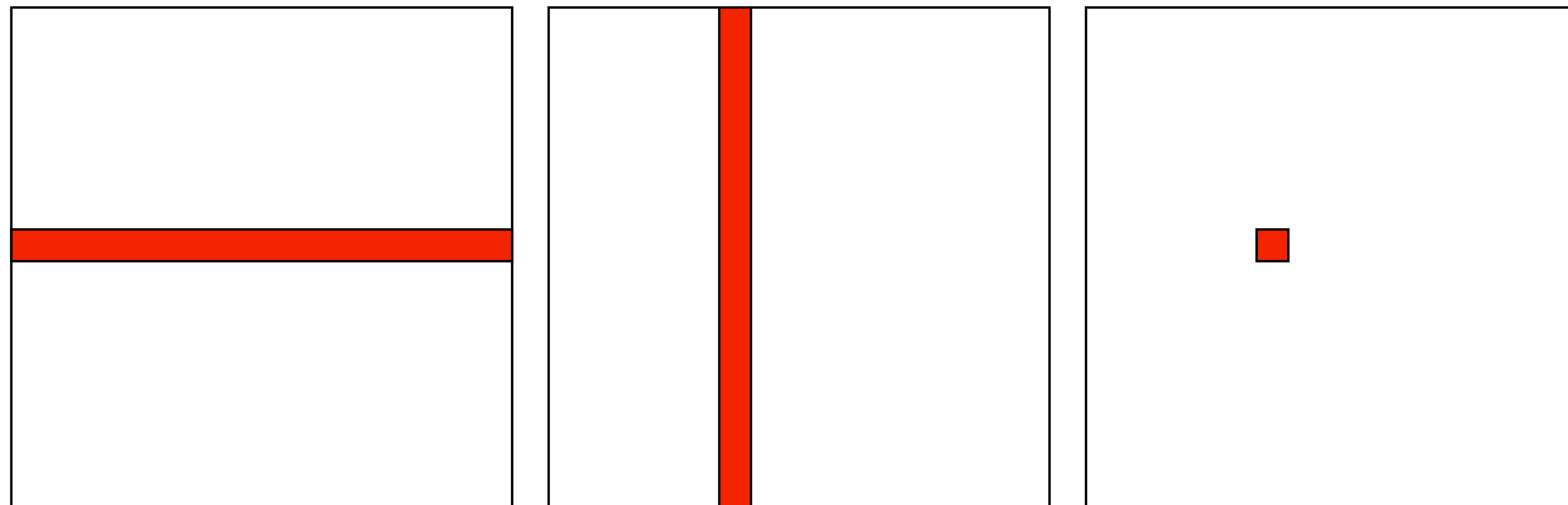
Process

Write back as needed

Shared memory as ”manual cache”

Example: Matrix multiplication

Matrix multiplication



To multiply two N*N matrices, every item will have to be accessed N times!

Naive implementation: $2N^3$ global memory accesses!

Matrix multiplication on CPU

Simple triple ”for” loop

```
void MatrixMultCPU(float *a, float *b, float *c, int theSize)
{
  int sum, i, j, k;

  // For every destination element
  for(i = 0; i < theSize; i++)
    for(j = 0; j < theSize; j++)
    {
      sum = 0;
      // Sum along a row in a and a column in b
      for(k = 0; k < theSize; k++)
        sum = sum + (a[i*theSize + k]*b[k*theSize + j]);
      c[i*theSize + j] = sum;
    }
}
```

Naive GPU version

Replace outer loops by thread indices

```
__global__ void MatrixMultNaive(float *a, float *b, float *c,
int theSize)
{
  int sum, i, j, k;

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;

  // For every destination element
  sum = 0;
  // Sum along a row in a and a column in b
  for(k = 0; k < theSize; k++)
    sum = sum + (a[i*theSize + k]*b[k*theSize + j]);
  c[i*theSize + j] = sum;
}
```

Naive GPU version inefficient

Every thread makes 2N global memory accesses!

Can be significantly reduced using shared memory

Optimized GPU version

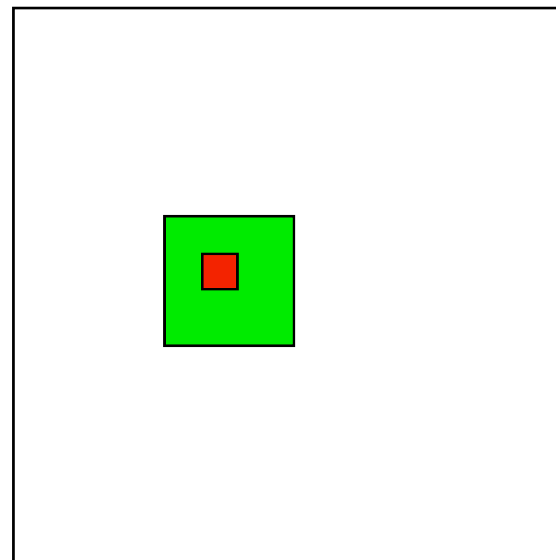
Data is split into patches.

Every element accesses data in all the patches in the same row for A, column for B

Each output patch is mapped to one block.

For every such block:
Every thread reads one element to shared memory
Then loop over the appropriate row and column for the block

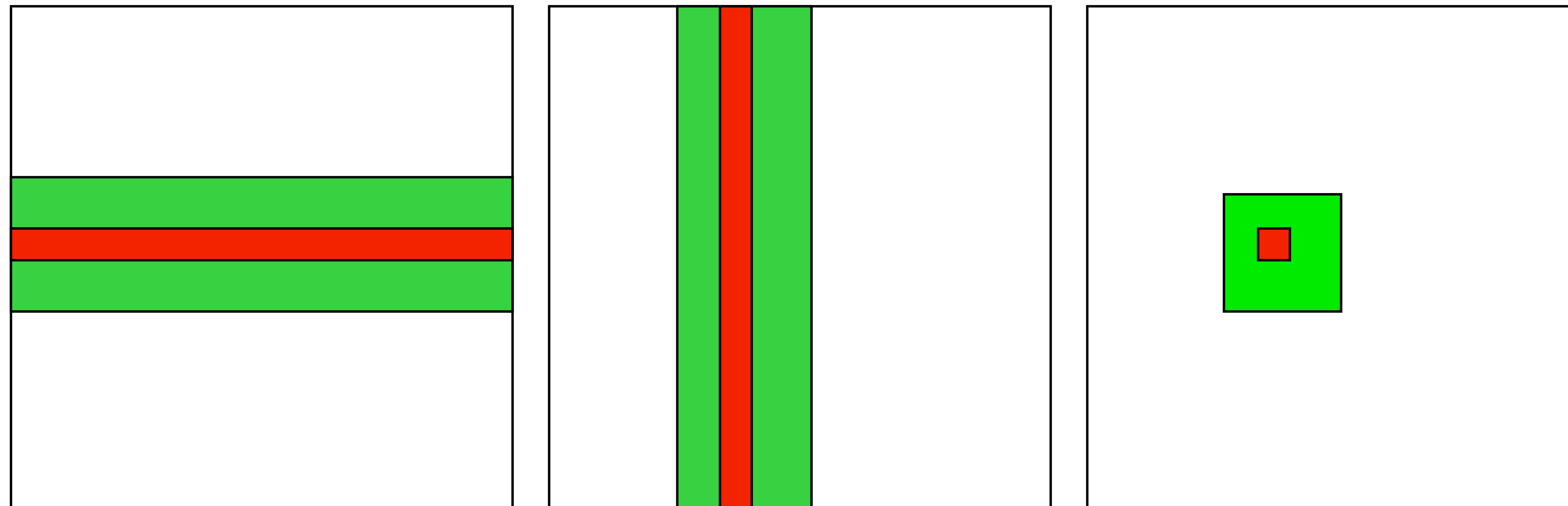
Split problem to parts, separate patches per block



Let each block (green) handle a part of the output.

Every thread handles one output (red)!

Contributing areas for patch



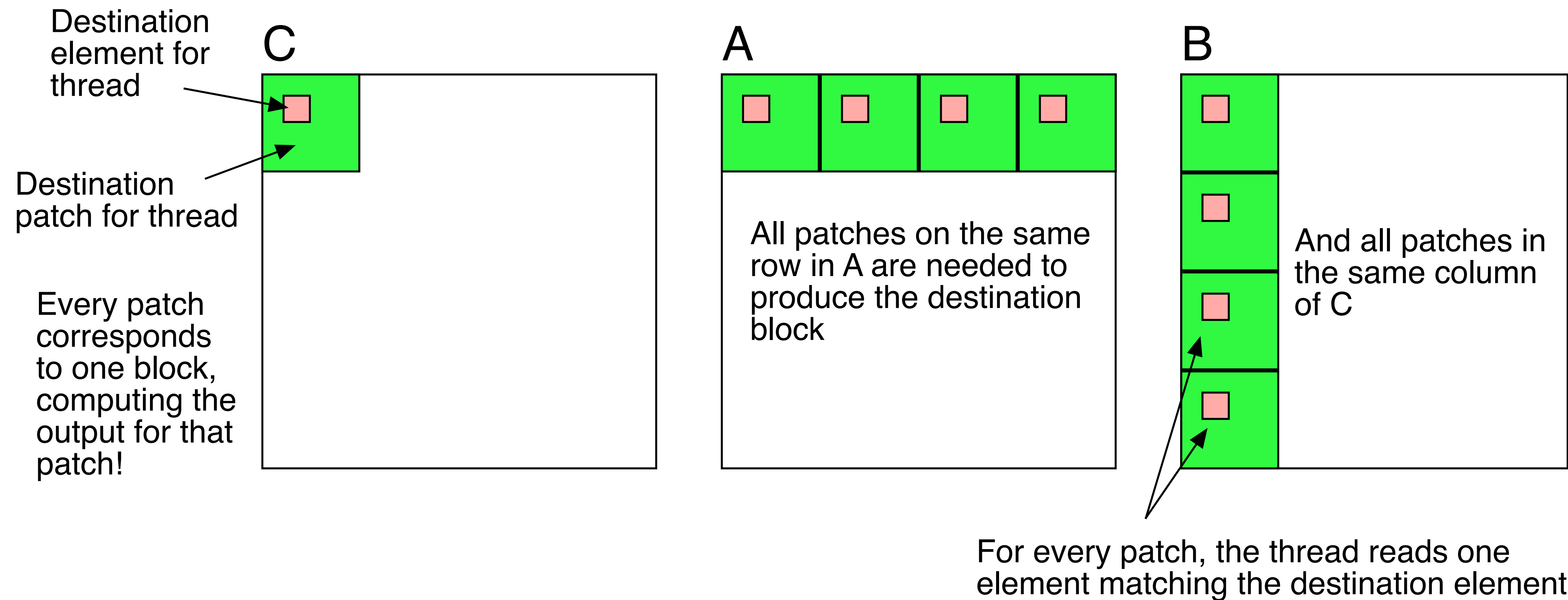
Load the contributing areas into shared memory.

The thread (red left) needs to access the items marked as red bars.

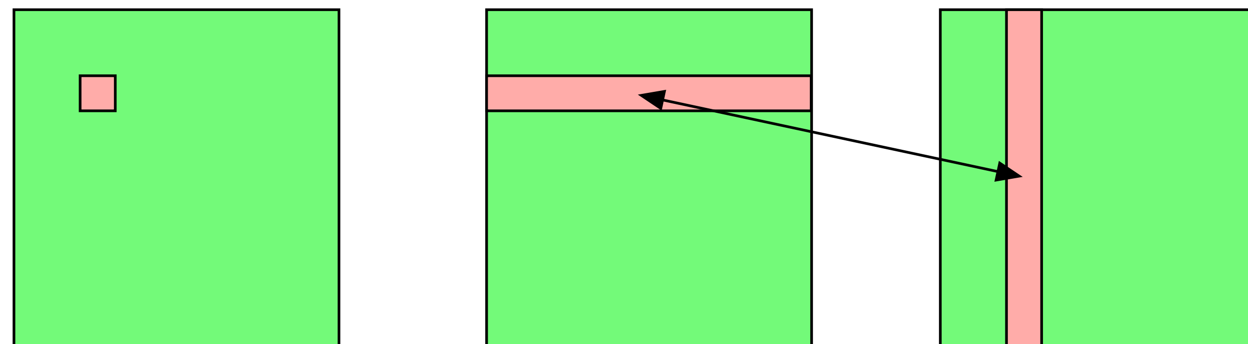
This means that the block needs the total green/red!

Example: 16 blocks

16 blocks picture

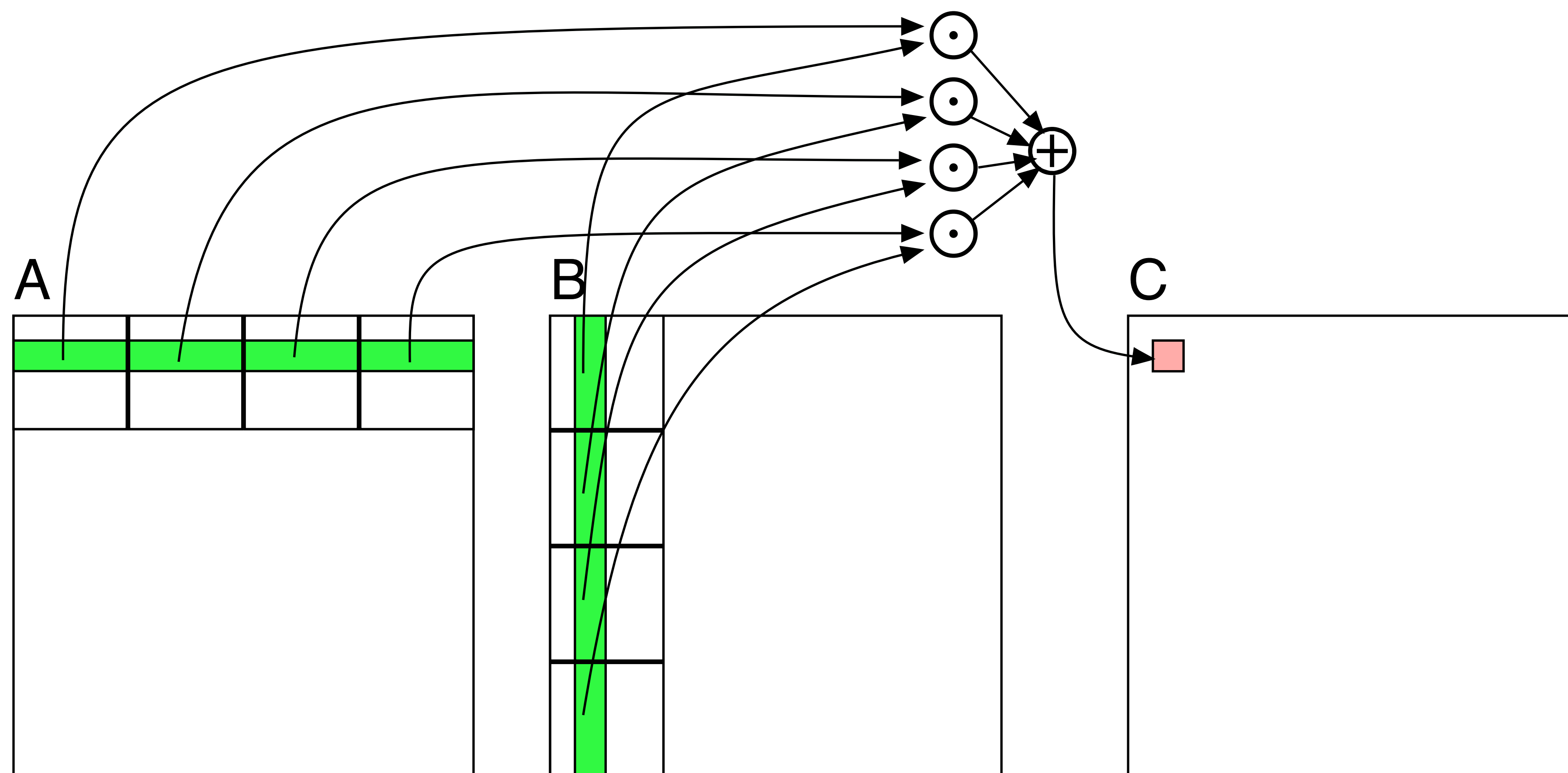


For every patch, we loop over the part of one row and column to perform that part of the computation



What one thread reads is used by everybody in the same row (A) or column (B)!

Piece by piece, patch by patch



Optimized GPU version

Loop over patches (1D)

Allocate shared memory

Copy *one* element to shared memory

Loop over row/column in patch, compute, accumulate result for one element

Write result to global memory

```
__global__ void MatrixMultOptimized( float* A, float* B, float* C, int theSize)
{
    int k, b, gx, gy, gi, bx, by, gia, gib, li;

// Global index for thread
    gx = blockIdx.x * blockDim.x + threadIdx.x;
    gy = blockIdx.y * blockDim.y + threadIdx.y;
    gi = gy*theSize + gx;

// Local index for thread
    li = threadIdx.y*blockDim.y + threadIdx.x;

    float sum = 0.0;
    // for all source blocks
    for (b = 0; b < gridDim.x; b++) // We assume that gridDimx and y are equal
    {
        __shared__ float As[BLOCKSIZE*BLOCKSIZE];
        __shared__ float Bs[BLOCKSIZE*BLOCKSIZE];

        bx = blockDim.x*b + threadIdx.x; // modified x for A
        by = blockDim.y*b + threadIdx.y; // modified y for B
        gia = gy*theSize+bx; // resulting global index into A
        gib = by*theSize+gx; // resulting global index into B

        As[li] = A[gia];
        Bs[li] = B[gib];

                __syncthreads(); // Synchronize to make sure all data is loaded

        // Loop in block
        for (k = 0; k < blockDim.x; k++)
            sum += As[threadIdx.y*blockDim.x + k] * Bs[k*blockDim.x + threadIdx.x];

        __syncthreads(); // Synch again so nobody starts loading data before all finish
    }

    C[gi] = sum;
}
```

5-10 times faster? So what did I do?

- Decent number of threads and blocks
- Use shared memory for temporary storage
- All threads read ONE item, but use many!
- Synchronize
- Even more for CPU - compared to single-thread CPU :)

Modified computing model:

Upload data to global GPU memory

For a number of parts, do:

Upload partial data to shared memory

Process partial data

Write partial data to global memory

Download result to host

More code ≠ slower!

More optimization often needs more code!

Compare to loop unrolling.

In our case: Program steps are cheap, global memory access is expensive!

Thread synchronization

As soon as you do something where one part of a computation depends on a result from another thread, you must synchronize!

__syncthreads()

Typical implementation:

- Read to shared memory
- __syncthreads()
- Process shared memory
- __synchthreads()
- Write result to global memory

Thread synchronization

Really wonderfully simple - everybody are doing the same thing anyway!

Synchronization simply means "wait until everybody are done with this part"

Deadlocks can still occur!

Limitation of synchronization

Thread synchronization can only be done within a block! No synchronization between blocks!

Why is this a necessary limitation?

Limitation of synchronization

Thread synchronization can only be done within a block! No synchronization between blocks!

Why is this a necessary limitation?

Because all blocks are not active at the same time! Blocks are queued until an SM is free!

Limitation of synchronization

Thread synchronization can only be done within a block! No synchronization between blocks!

Why is this a necessary limitation?

Because all blocks are not active at the same time! Blocks are queued until an SM is free!

But I must synchronize globally!

Answer: Run multiple kernel runs! More on this later.

Global synchronization

Another synchronization is global synchronization.

Called by the host.

Wait until all blocks are finished!

cudaDeviceSynchronize()

There is also

cudaStreamSynchronize()

Lecture questions revisited:

1. What concept in CUDA corresponds to a SM (streaming multiprocessor) in the architecture?

2. How does matrix multiplication benefit from using shared memory?

3. When do you typically need to synchronize threads?

One more time:

More code ≠ slower

The fast version requires more code!

Instructions are cheap, memory access is expensiove!

Summary:

- Make threads and blocks to make the hardware occupied
- Access data depending on thread/block number
- Memory accesses are expensive!
- Shared memory is fast
- Make threads within a block cooperate
- Synchronize

Thread/block balancing

How many threads?

How many threads per block?

If many, try 256 threads/block

If smaller problem, how do you utilize the hardware the best?

What comes next?

- More CUDA
- Even more CUDA

but then

- OpenCL and compute shaders - the alternatives

Most that I say about CUDA translate easily to other platforms!

That’s all folks!

Next: More about memory management and optimization.