Lecture 10

Introduction to CUDA

Ingemar Ragnemalm

Information Coding, ISY

Laborations

Lab 4-6 are ready, no changes planned *but* last minute changes may occur.

The ”lab questions” are vital! Answers *must* be written down before we can examine you!

Thus - no lab reports needed.

Lecture material

Lecture material available on the web, new and last year's.

The old local course page is obsolete but is linked to

http://computer-graphics.se/TDDD56

The lecture material is linked from the "Lectures" page.

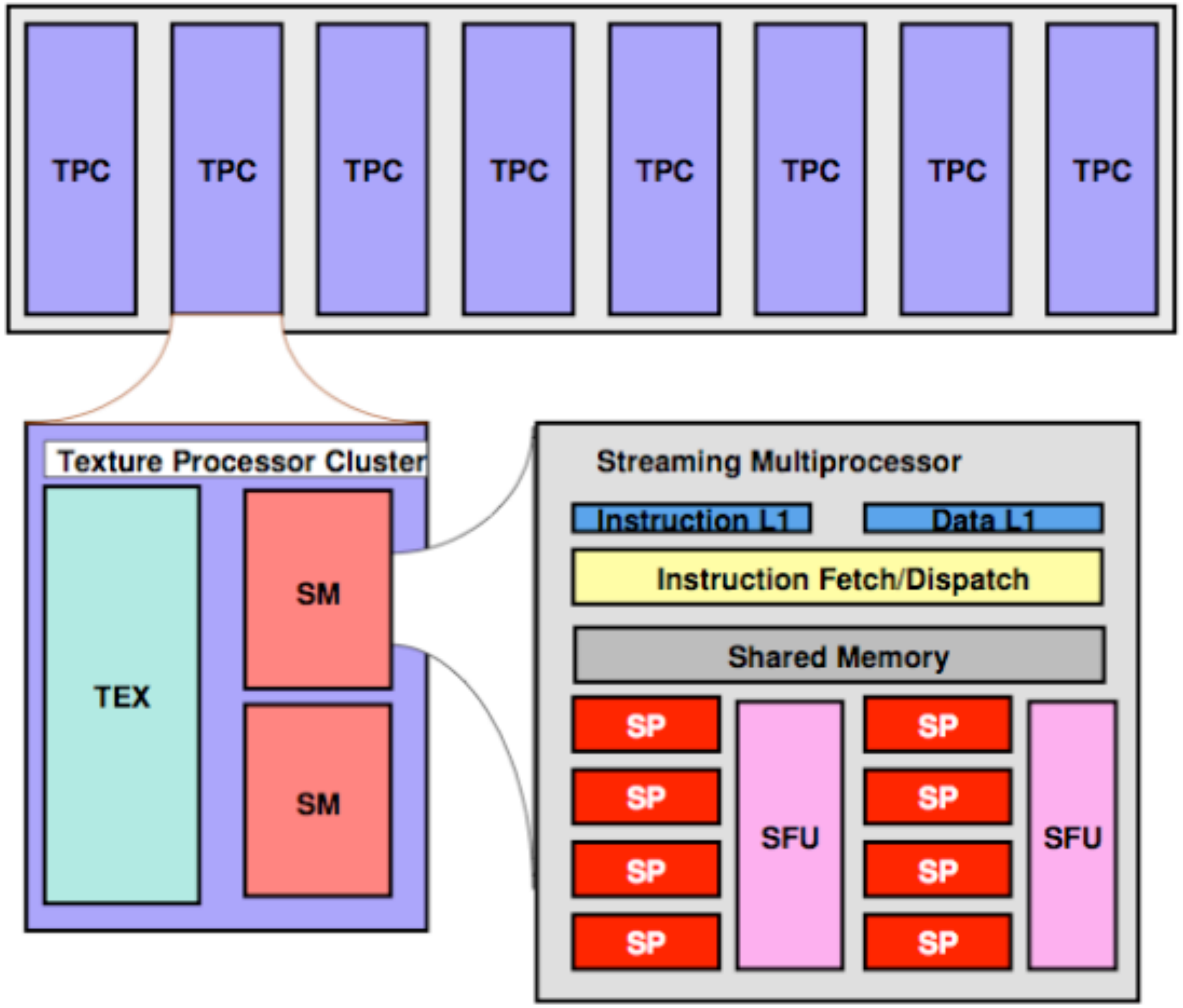
Previous lecture:

GPU development - why did it become a general purpose parallel architecture

GPU architecture

A quick look at GPU coding (Hello World!)

G80 processor hierarchy



8 top-level groups of TPCs

SM (Streaming Multiprocessor) is a group of 8 SIMD cores

This lecture:

CUDA

Programming model and language

Introduction to memory spaces and memory access

Shared memory

Matrix multiplication example

Lecture questions:

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?

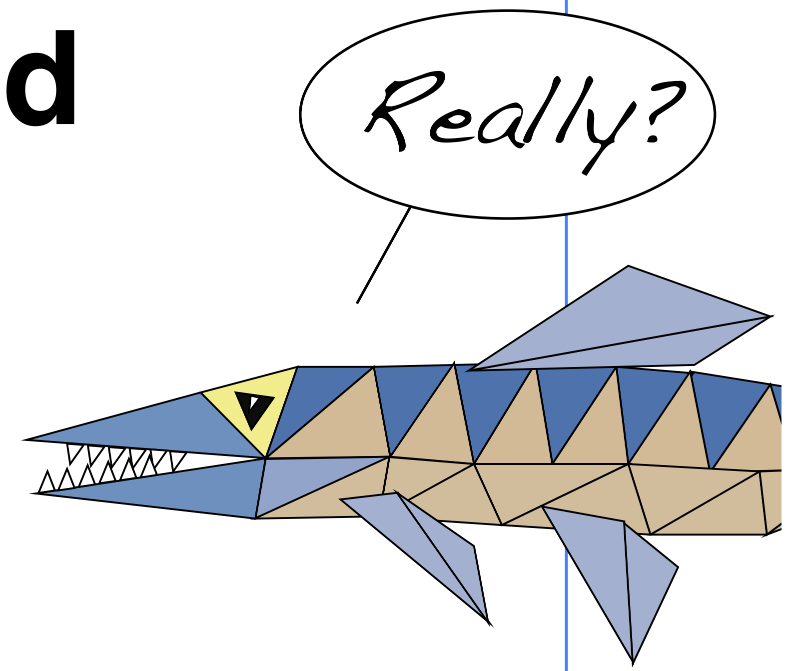
Why do we focus on CUDA?

Easiest start! Compact and comfortable code.

Drawback: NVidia only.

We do not forget the alternatives! We return to them later.

CUDA = Compute Unified Device Architecture



Developed by NVidia

Only available on NVidia boards, G80 or better GPU architecture

Designed to hide the graphics heritage and add control and flexibility

Computing model:

1. Upload data to GPU

2. Execute kernel

3. Download result

Similar to shader-based solutions and OpenCL

Integrated source

Source of host and kernel code in the same source file!

Major difference to shaders and OpenCL.

Kernel code identified by special modifiers.

Threads and warps

CUDA = Architecture and C extension

Spawn a large number of threads, to be ran virtually in parallel

Just like in graphics! Fragments/computations not *quite* executed in parallel.

A bunch at a time - a *warp*.

Looks much more like an ordinary C program! No more ”data stored as pixels” - just arrays!

Simple CUDA example

A working, compilable example

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void simple(float *c)
{
 c[threadIdx.x] = threadIdx.x;
}

int main()
{
 int i;
 float *c = new float[N];
 float *cd;
 const int size = N*sizeof(float);

 cudaMalloc( (void**)&cd, size );
 dim3 dimBlock( blocksize, 1 );
 dim3 dimGrid( 1, 1 );
 simple<<<dimGrid, dimBlock>>>(cd);
 cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
 cudaFree( cd );

 for (i = 0; i < N; i++)
  printf("%f ", c[i]);
 printf("\n");
 delete[] c;
 printf("done\n");
 return EXIT_SUCCESS;
}
```

Simple CUDA example

A working, compilable example

Kernel

thread identifier

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void simple(float *c)
{
 c[threadIdx.x] = threadIdx.x;
}

int main()
{
 int i;
 float *c = new float[N];
 float *cd;
 const int size = N*sizeof(float);
```

Allocate GPU memory

1 block, 16 threads

Call kernel

Read back data

```
 cudaMalloc( (void**)&cd, size );
 dim3 dimBlock( blocksize, 1 );
 dim3 dimGrid( 1, 1 );
 simple<<<dimGrid, dimBlock>>>(cd);
 cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
 cudaFree( cd );

 for (i = 0; i < N; i++)
  printf("%f ", c[i]);
 printf("\n");
 delete[] c;
 printf("done\n");
 return EXIT_SUCCESS;
}
```

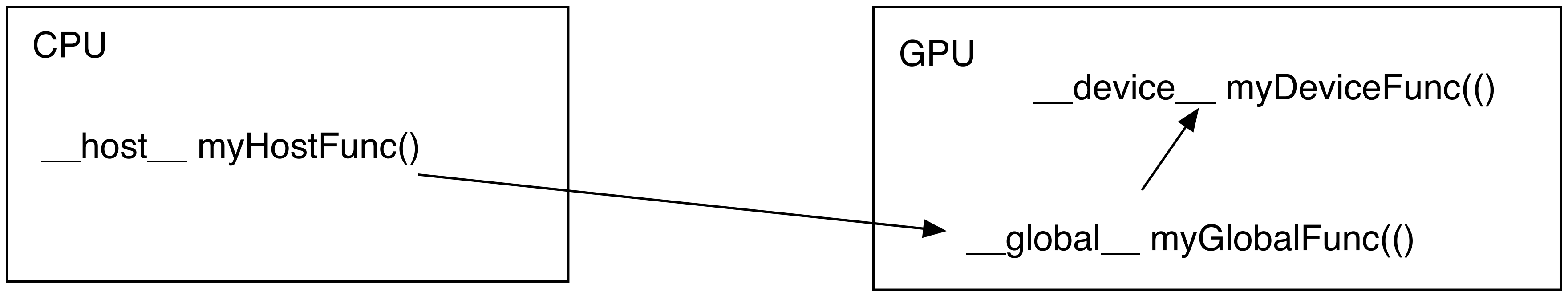
Modifiers for code

Three modifiers are provided to specify how code should be used:

__global__ executes on the GPU, invoked from the CPU. This is the entry point of the kernel.

__device__ is local to the GPU

__host__ is CPU code (superfluous).



Memory management

cudaMalloc(ptr, datasize)
cudaFree(ptr)

Similar to CPU memory management, but done by the CPU to allocate on the GPU

cudaMemCpy(dest, src, datasize, arg)

arg = cudaMemcpyDeviceToHost or cudaMemcpyHostToDevice

Kernel execution

simple<<<griddim, blockdim>>>(…)

grid = blocks, block = threads

Built-in variables for kernel:

threadIdx* and *blockIdx
blockDim* and *gridDim

(Note that no prefix is used, like GLSL does.)

Compiling Cuda

nvcc

nvcc is nvidia's tool, /usr/local/cuda/bin/nvcc

Source files suffixed .cu

Command-line for the simple example:

```
nvcc simple.cu -o simple
```

(Command-line options exist for libraries etc)

Compiling Cuda for larger applications

nvcc and gcc in co-operation

nvcc for .cu files

gcc for .c/.cpp etc

Mixing languages possible.

Final linking must include C++ runtime libs.

Example: One C file, one CU file

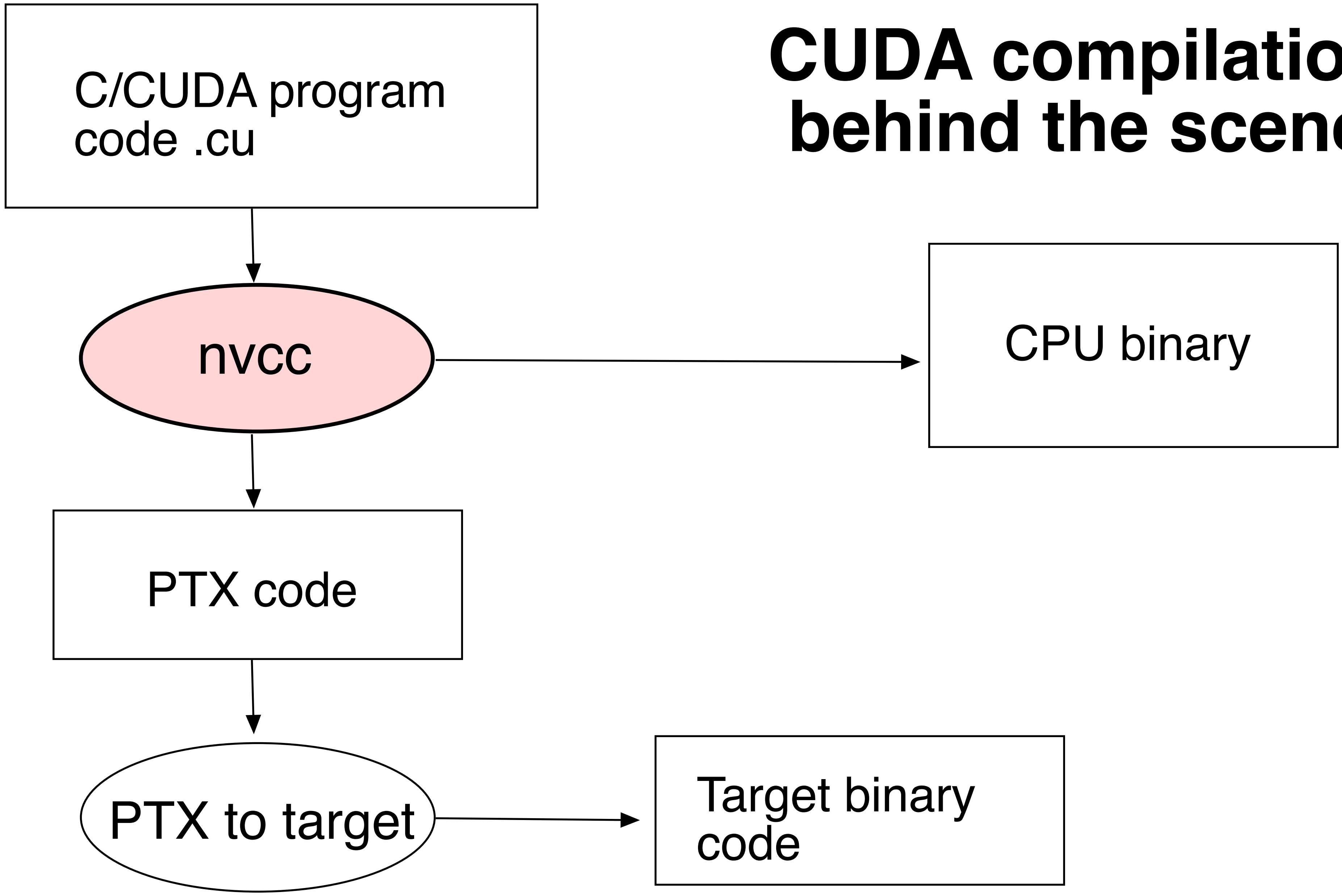
Example of multi-unit compilation

Source files: cudademokernel.cu and cudademo.c

```
nvcc cudademokernel.cu -o cudademokernel.o -c

gcc -c cudademo.c -o cudademo.o -I/usr/local/cuda/include

g++ cudademo.o cudademokernel.o -o cudademo -L/usr/local/
    cuda/lib -lcuda -lcudart -lm
```

Link with g++ to include C++ runtime

CUDA compilation behind the scene



Executing a Cuda program

Must set environment variable to find Cuda runtime.

```
export DYLD_LIBRARY_PATH=/usr/local/cuda/lib:$DYLD_LIBRARY_PATH
```

Then run as usual:

./simple

A problem when executing without a shell!

Launch with execve()

Computing with CUDA

Organization and access

Blocks, threads...

Warps

A warp is the minimum number of data items/threads that will actually be processed in parallel by a CUDA capable device. This number varies with different GPUs.

We usually don't care about warps but rather discuss threads and blocks.

Processing organization

1 warp = 32 threads

1 kernel - 1 grid

1 grid - many blocks

1 block - 1 SM

1 block - many threads

Use many threads and many blocks! > 200 blocks recommended.

Thread # multiple of 32

...almost right

1 block -> 1 SM

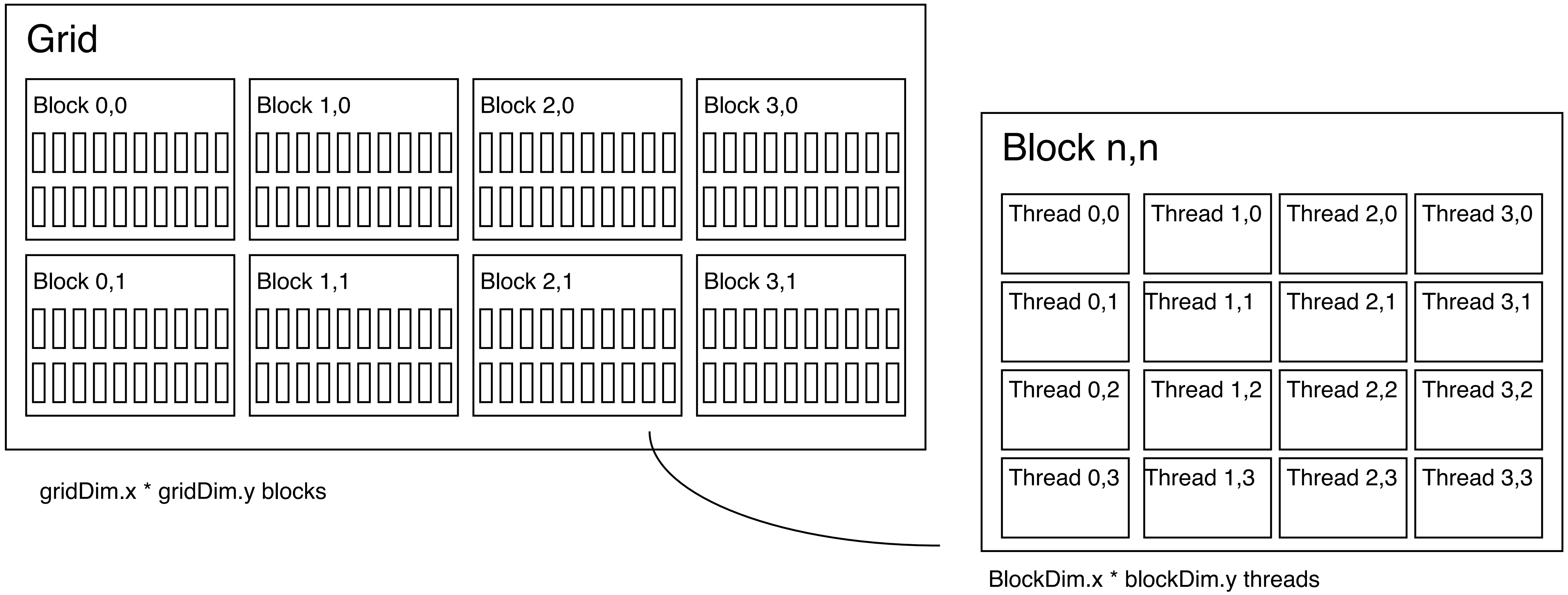
but not

1 SM -> 1 block

A block is always assigned to one SM, but one SM may run more than one block at a time.

Distributing computing over threads and blocks

Hierarcical model



Indexing data with thread/block IDs

Calculate index by blockIdx, blockDim, threadIdx

Another simple example, calculate square of every element, device part:

```
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
 int idx = blockIdx.x * blockDim.x + threadIdx.x;
 if (idx<N) a[idx] = a[idx] * a[idx];
}
```

Host part of "cudademo" example

Set block size and grid size

```
// main routine that executes on the host
int main(int argc, char *argv[])
{
 float *a_h, *a_d; // Pointer to host and device arrays
 const int N = 10; // Number of elements in arrays
 size_t size = N * sizeof(float);
 a_h = (float *)malloc(size);
 cudaMalloc((void **) &a_d, size);   // Allocate array on device
// Initialize host array and copy it to CUDA device
 for (int i=0; i<N; i++) a_h[i] = (float)i;
 cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
// Do calculation on device:
 int block_size = 4;
 int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
 square_array <<< n_blocks, block_size >>> (a_d, N);
// Retrieve result from device and store it in host array
 cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
// Print results and cleanup
 for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
 free(a_h); cudaFree(a_d);
}
```

Some new calls in cudademo example

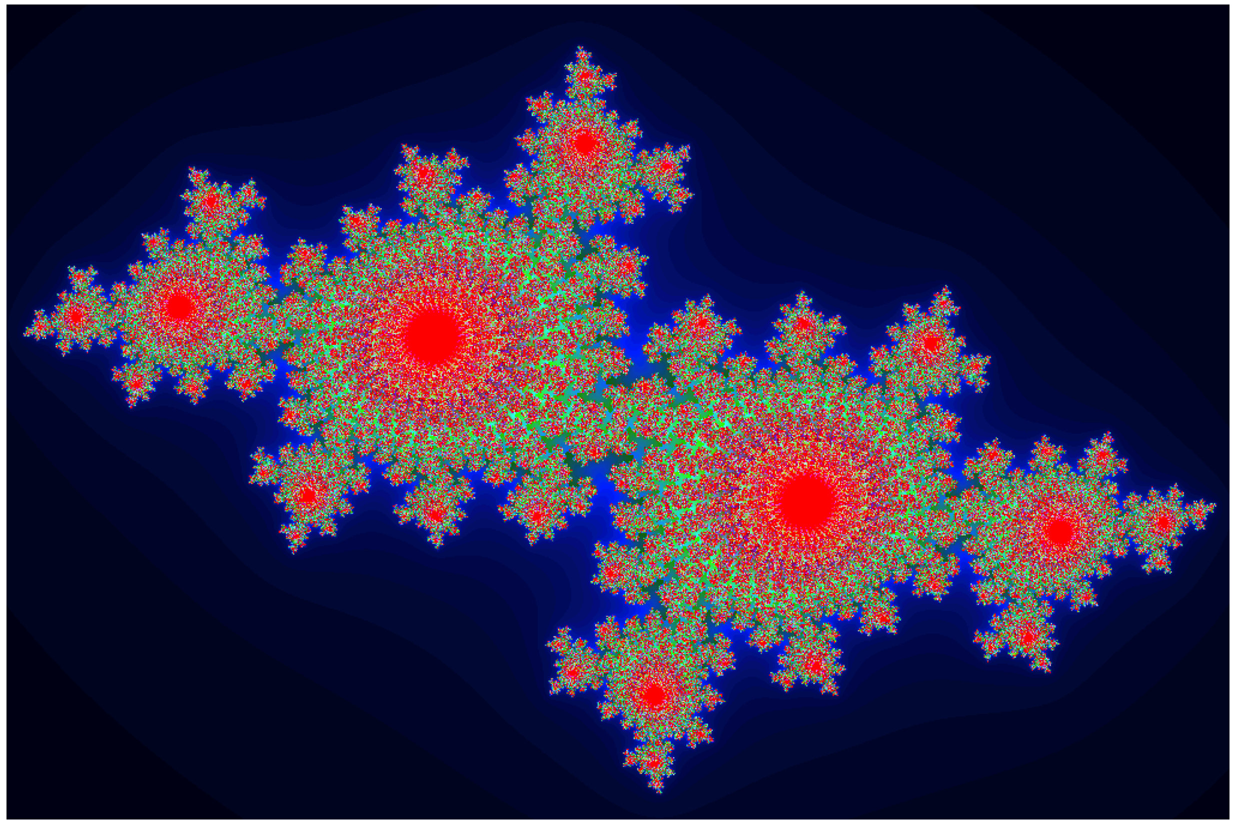
```
/* find number of device in current "context" */
cudaGetDevice(&devid);

/* find how many devices are available */
cudaGetDeviceCount(&devcount);
```

and there is also

```
cudaSetDevice(devId)
```

Julia example

- Bigger problem, addressing calculation must be 2D
- Simple OpenGL output (similar to the labs)



Julia example

For this case: Separate for x and y

```
__global__ void kernel( unsigned char *ptr, float r, float im)
{
    // map from blockIdx to pixel position
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int offset = x + y * DIM;

    // now calculate the value at that position
    int juliaValue = julia( x, y, r, im );
--- calculate colors ---
    ptr[offset*4 + 0] = red;
    ptr[offset*4 + 0] = green;
    ptr[offset*4 + 0] = blue;
    ptr[offset*4 + 3] = 255;
}
```

Actual index which implies memory position

Every thread computes one single pixel!

Julia conclusions

Many blocks, many treads in each block. Make sure everything is in use.

Index by thread and block.

Exceptional speedup - trivially parallellizable problem!

Load balancing? No problem. Why?

Conclusion about indexing

Every thread does its own calculation for indexing memory!

blockIdx, blockDim, threadIdx

1, 2 or 3 dimensions

Usually 2 dimensions

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!

Shared memory fast temporary storage

Coalesce memory access!

Continuous
Aligned on power of 2 boundary
Addressing follows thread numbering

Use shared memory for reorganizing data for coalescing!

Using shared memory to reduce number of global memory accesses

Read blocks of data to shared memory

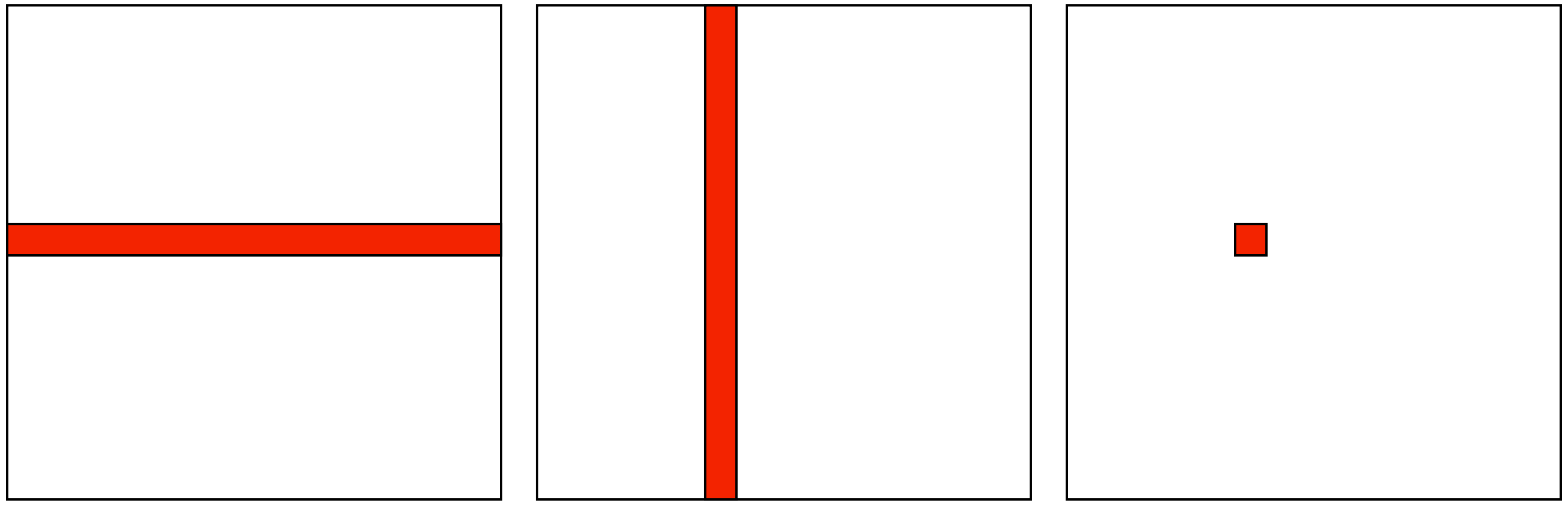
Process

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

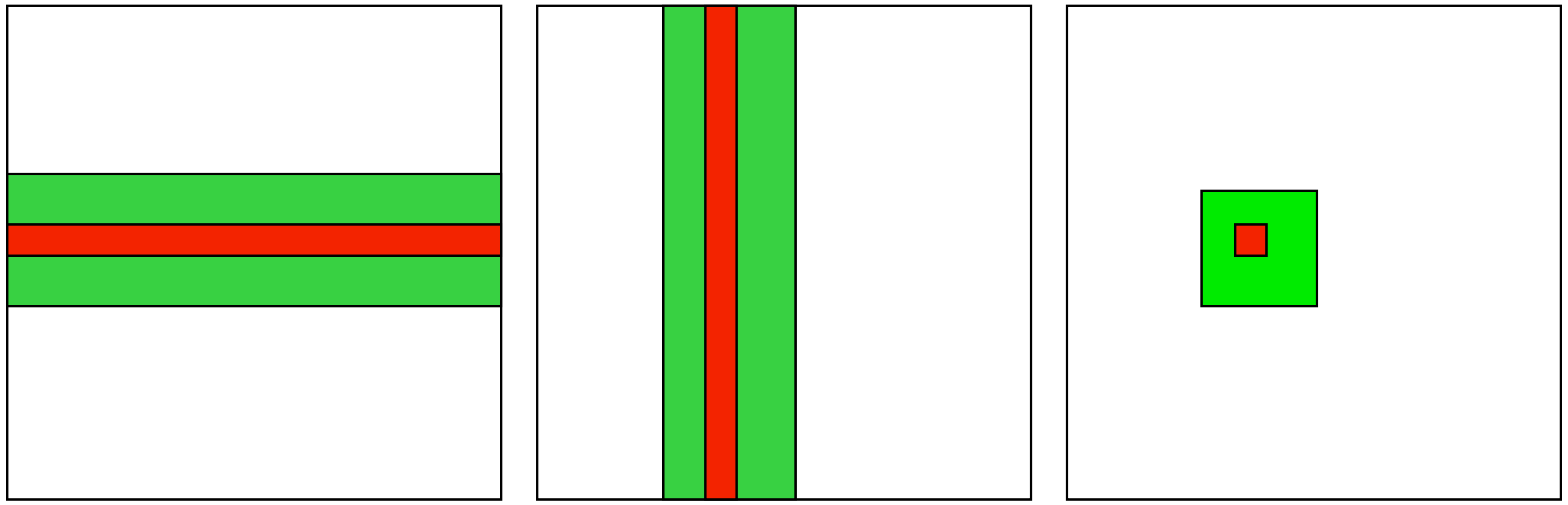
Data is split into patches.

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

Contributing areas for patch

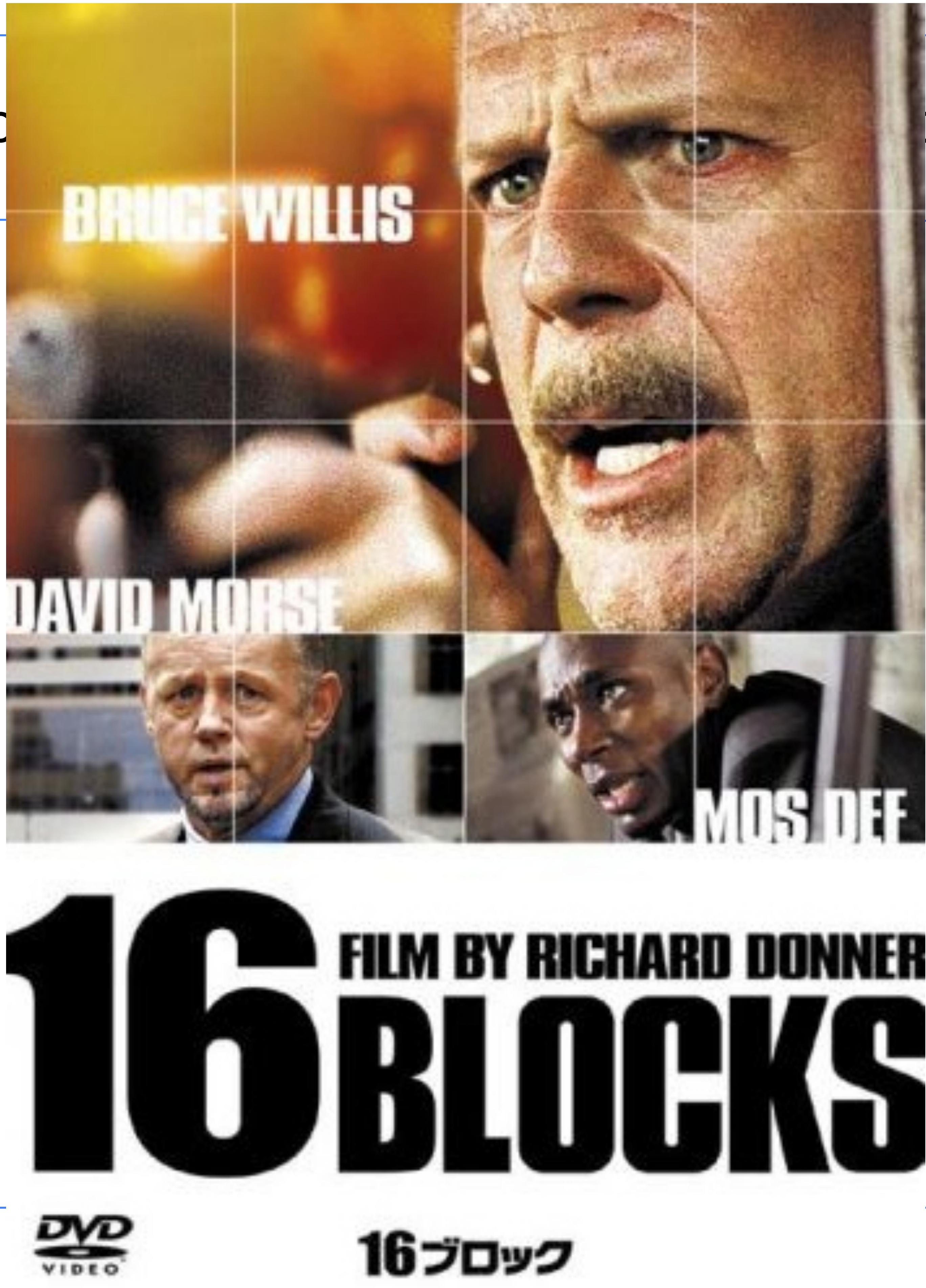


Let each block handle a part of the output (green right).

Green areas middle and left contibute to output.

Load the contributing areas into shared memory.

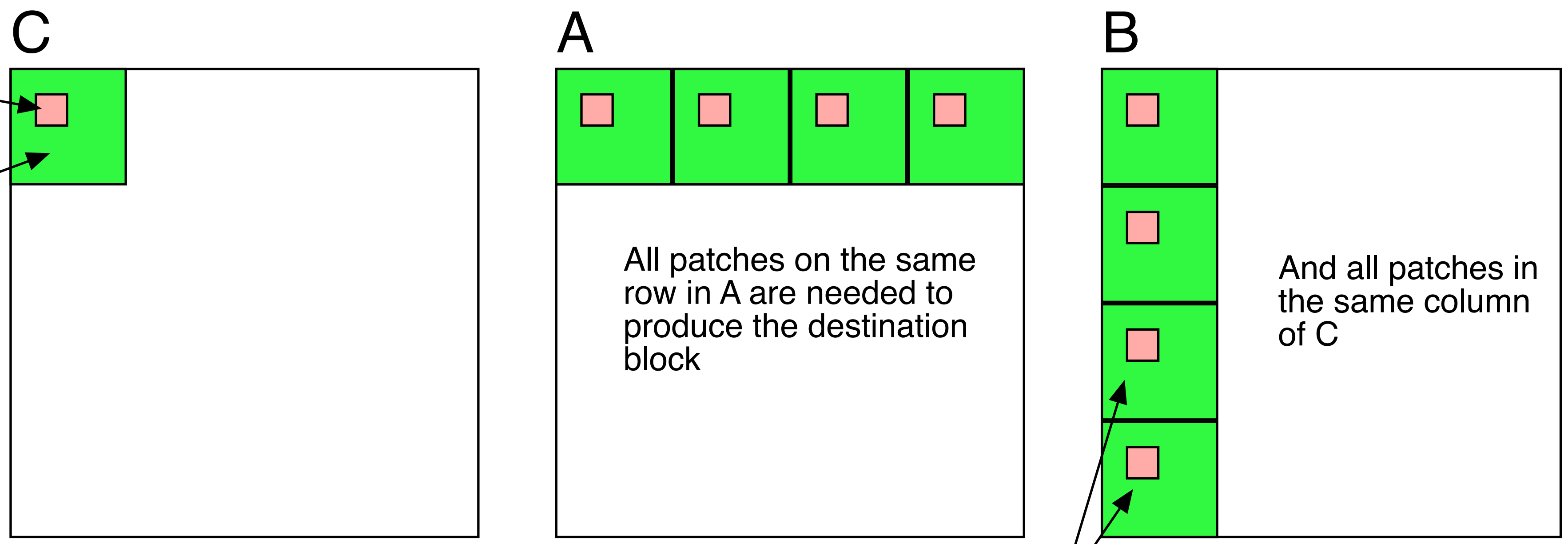
Example: 16 blocks


BRUCE WILLIS
DAVID MORSE
MOS DEF
16 BLOCKS
FILM BY RICHARD DONNER
DVD VIDEO
16ブロック

Destination element for thread

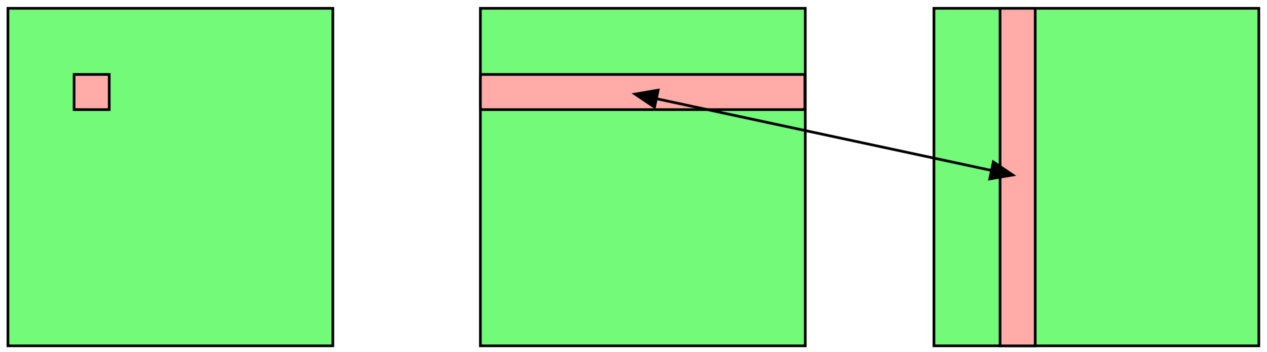
Destination patch for thread

Every patch corresponds to one block, computing the output for that patch!



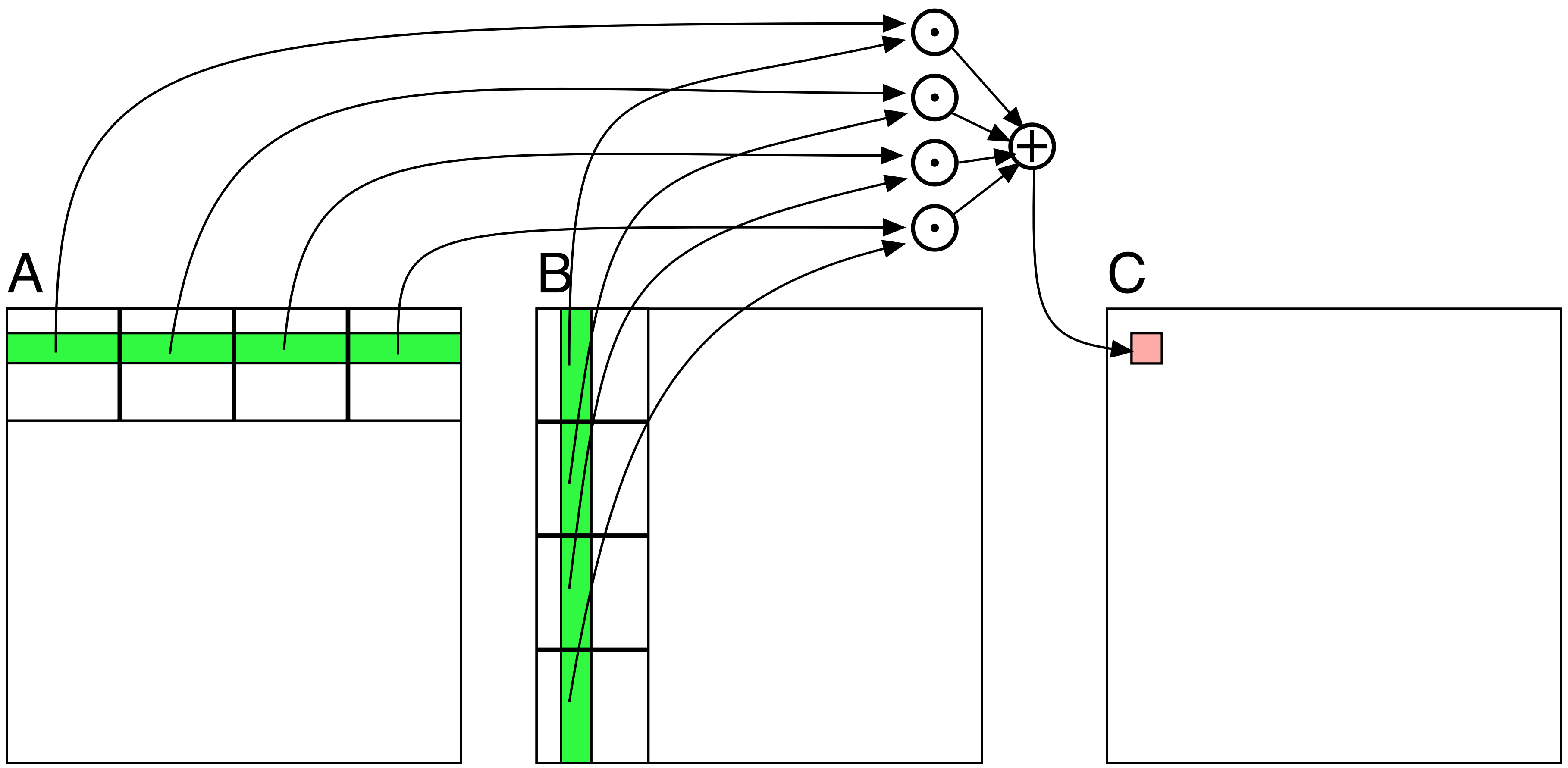
For every patch, the thread reads one element matching the destination element

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

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?

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

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.