Introduction to CUDA

Ingemar Ragnemalm
Information Coding, ISY

This lecture:

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?

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.

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

```
#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;
}
```

Kernel

thread identifier

Allocate GPU memory

1 block, 16 threads

Call kernel

Read back data

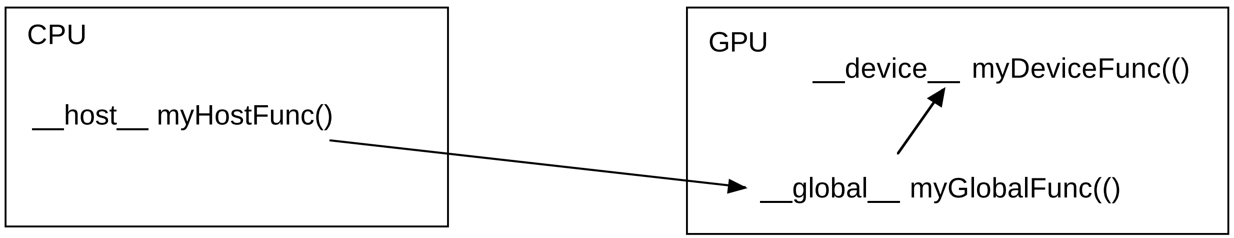
Modifiers for code

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

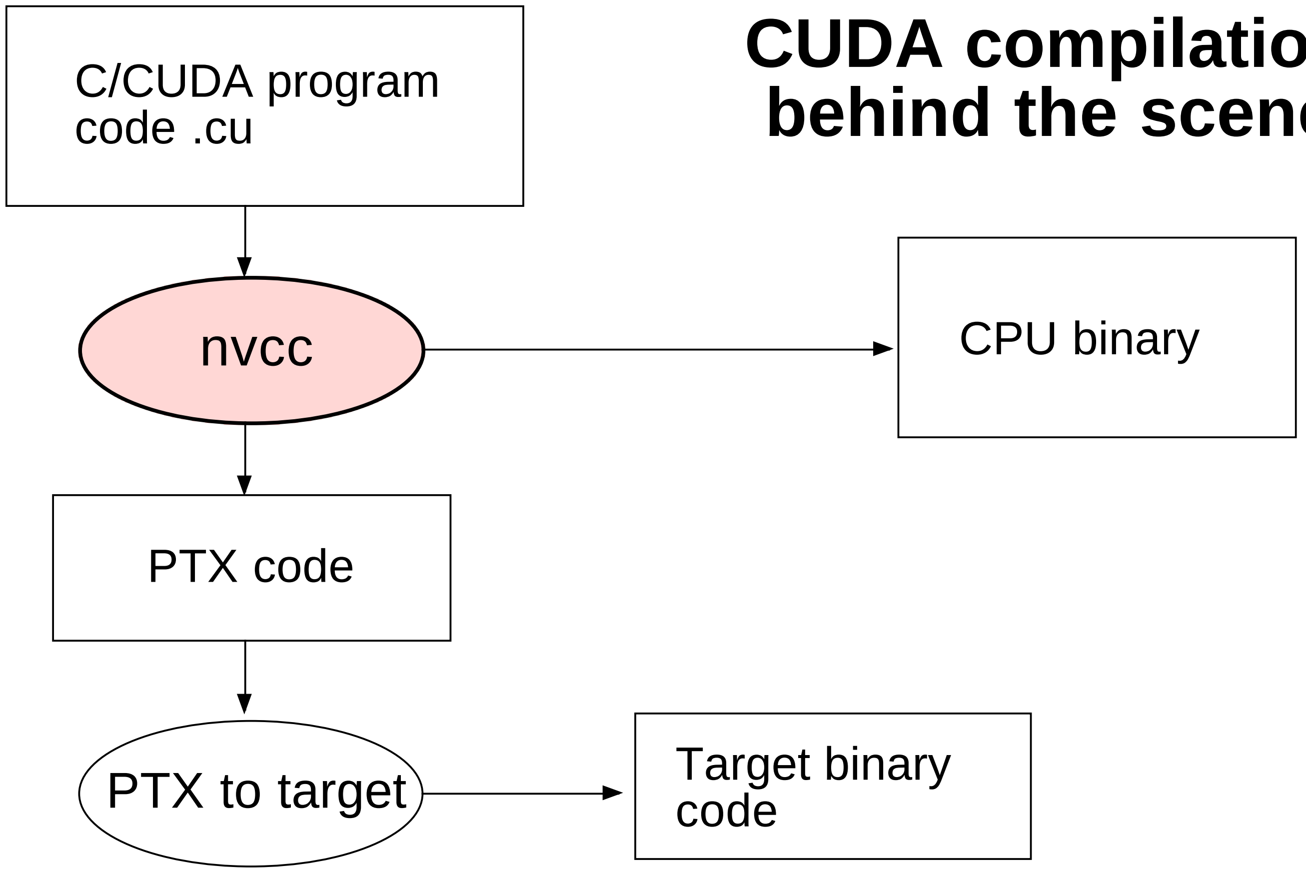
```
nvcc cudademokernel.cu -o cudademokernel.o -c

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

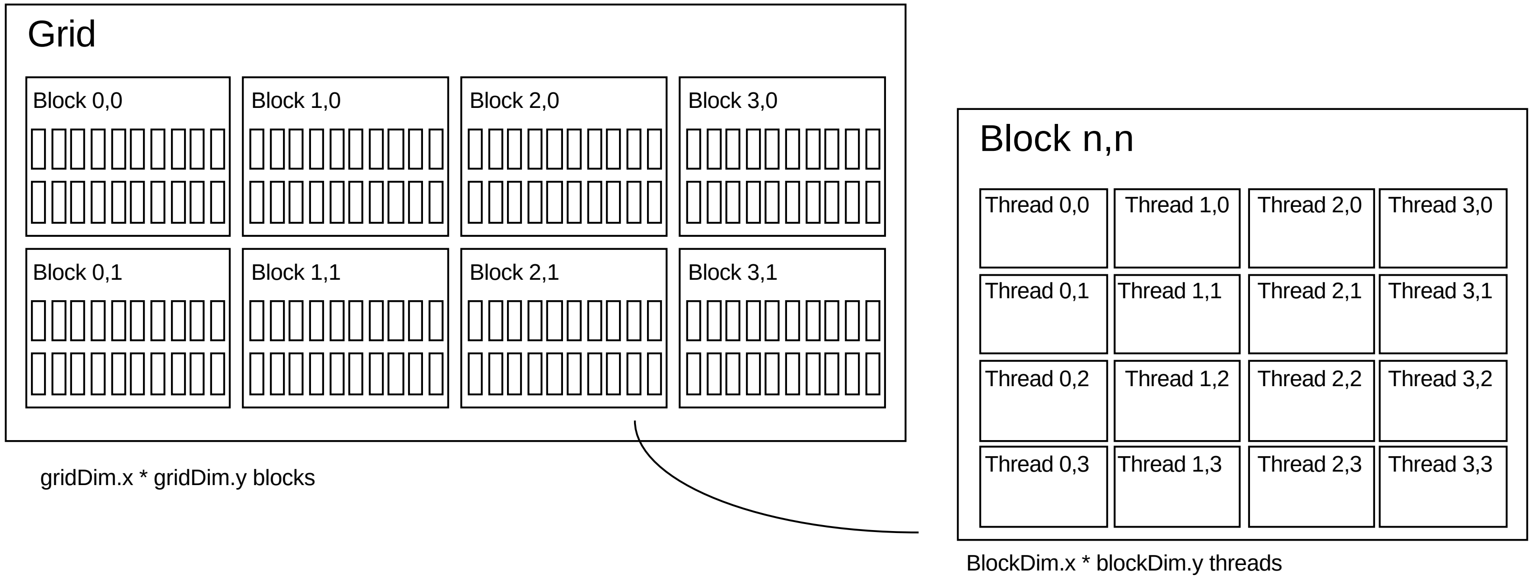
1 block - many threads

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

Thread # multiple of 32

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 square 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);
}
```

Julia example

For this case: Separate for x and y

Actual index which implies memory position

```
__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;
}
```

Every thread computes one single pixel!

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

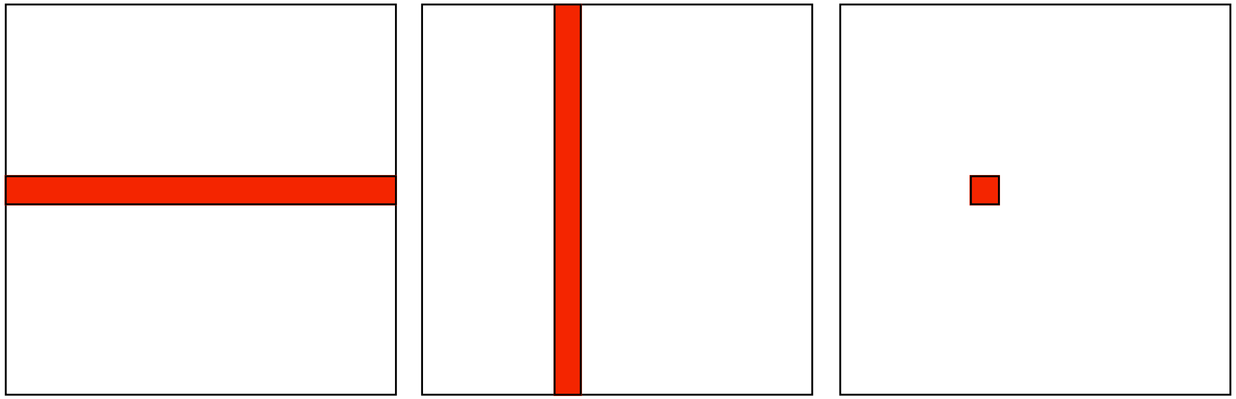
Process

Write back as needed

Shared memory as ”manual cache”

Example: Matrix multiplication

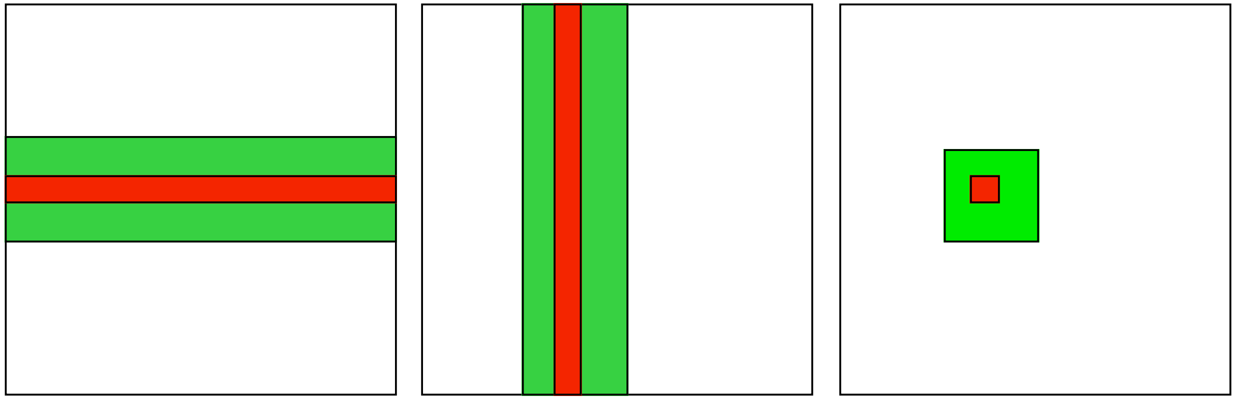
Matrix multiplication



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

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

Matrix multiplication



Let each block handle a part of the output.

Load the parts of the matrix needed for the block into shared memory.

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

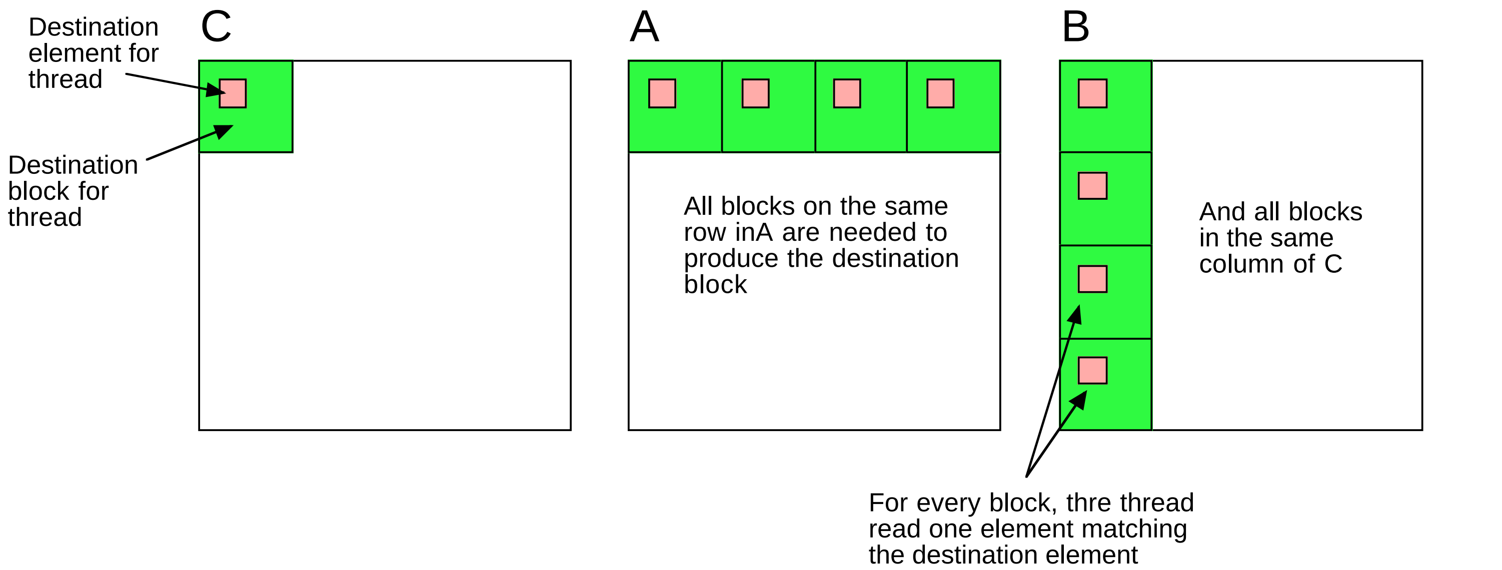
Data split into blocks.

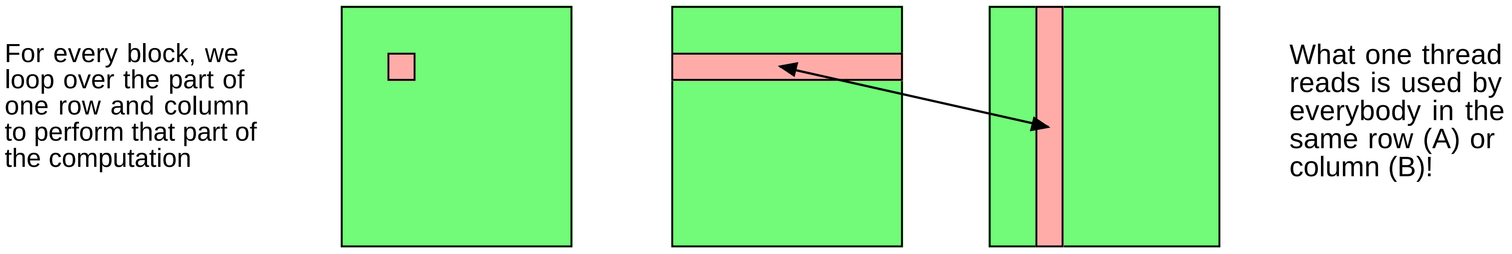
Every element takes part in all the blocks in the same *row* for A, *column* for B

For every such block

Every thread reads *one* element to shared memory

Then loop over the appropriate row and column for the block


Destination element for thread
Destination block for thread
C
A
B
All blocks on the same row inA are needed to produce the destination block
And all blocks in the same column of C
For every block, thre thread read one element matching the destination element


For every block, 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)!

Optimized GPU version

Loop over blocks (1D)

Allocate shared memory

Copy one element to shared memory

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

Write result to global memory

```
__global__ void MatrixMultOptimized( float* A, float* B, float* C, int theSize)
{
    int i, j, k, b, ii, jj;

// Global index for thread
    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;

    float sum = 0.0;
    // for all source blocks
    for (b = 0; b < gridDim.x; b++)
    {
        __shared__ float As[BLOCKSIZE*BLOCKSIZE];
        __shared__ float Bs[BLOCKSIZE*BLOCKSIZE];

        // Index locked to block
        ii = b * blockDim.x + threadIdx.x;
        jj = b * blockDim.y + threadIdx.y;

        As[threadIdx.y*blockDim.x + threadIdx.x] = A[ii*theSize + j];
        Bs[threadIdx.y*blockDim.x + threadIdx.x] = B[i*theSize + jj];

        __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 so nobody starts next pass prematurely
    }

    C[i*theSize + j] = sum;
}
```

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

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

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

That’s all folks!

Next: More about memory management and optimization.