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?


Information Coding / Computer Graphics, ISY, LiTH

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

The source of host and kernel code can be in the same source file, written as one and the same program!

Major difference to shaders and OpenCL, where the kernel source is separate and explicitly compiled by the host.

Kernel code identified by special modifiers.

CUDA

An architecture and C extension (and more!)

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

Just like in graphics! You can't expect all fragments/computations to be executed in parallel. Instead, they are executed a bunch at a time - a *warp*.

But unlike graphics it looks much more like an ordinary C program! No more "data stored as pixels" - they are just arrays!

Information Coding / Computer Graphics, ISY, LiTH

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

Information Coding / Computer Graphics, ISY, LiTH

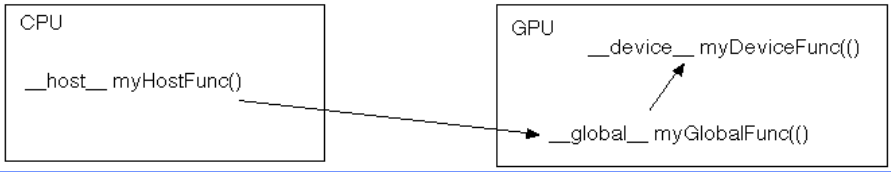
Modifiers for code

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


Information Coding / Computer Graphics, ISY, LiTH

Kernel execution

simple<<<griddim, blockdim>>>(...)

(Weird! Who came up with the syntax...?)

The grid is a grid of thread blocks. Threads have numbers within its block.

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

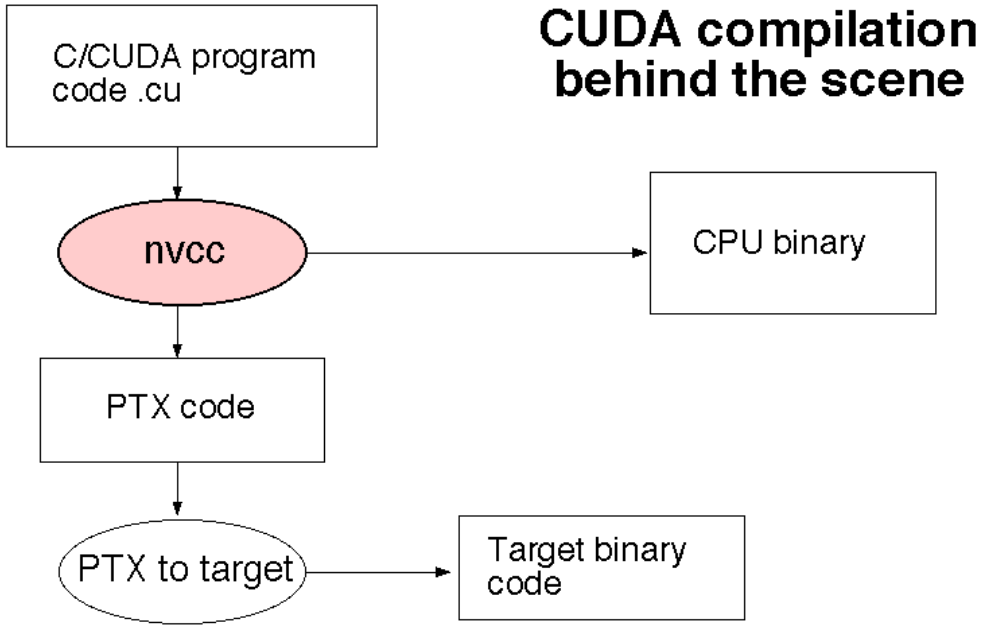
Source files: cudademokernel.cu and cudademo.c

```
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

Information Coding / Computer Graphics, ISY, LiTH

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.

Information Coding / Computer Graphics, ISY, LiTH

Processing organization

1 warp = 32 threads

1 kernel - 1 grid

1 grid - many blocks

1 block - 1 SM

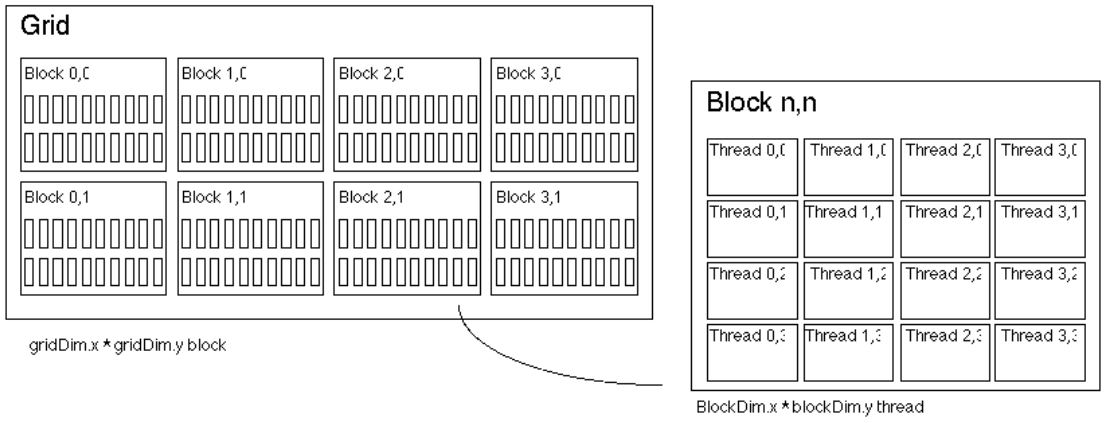
1 block - many threads

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

Thread # multiple of 32

Distributing computing over threads and blocks

Hierarcical model



Information Coding / Computer Graphics, ISY, LiTH

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);
}
```