Introduction to CUDA

Ingemar Ragnemalm
Information Coding, ISY

This lecture:

Programming model and language

Memory spaces and memory access

Shared memory

Examples

Lecture questions:

1. Suggest two significant differences between CUDA and OpenCL.

2. How does matrix transposing 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

Similar to shader-based solutions and OpenCL:

1. Upload data to GPU
2. Execute kernel
3. Download result

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!

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;
}
```

Kernel

thread identifier

```
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;
}
```

Allocate GPU memory

1 block, 16 threads

Call kernel

Read back data

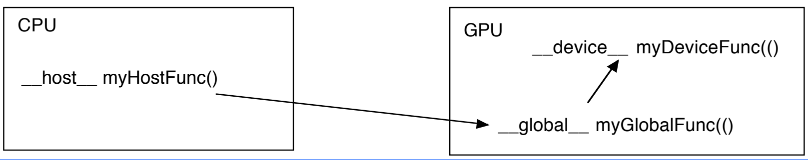
Modifiers for code

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>>>(…)

(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

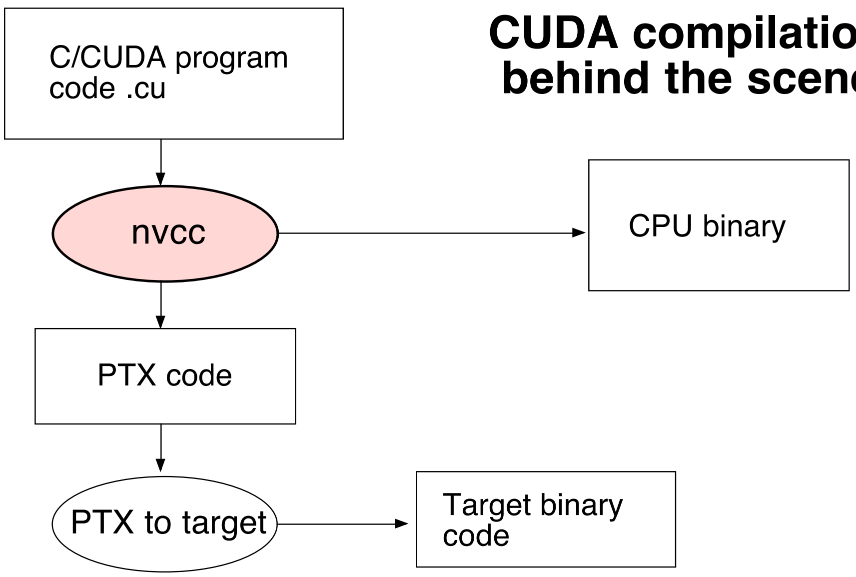
```
nvcc cudademokernel.cu -o cudademokernel.o -c

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

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 is set to 32.

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 thread processor

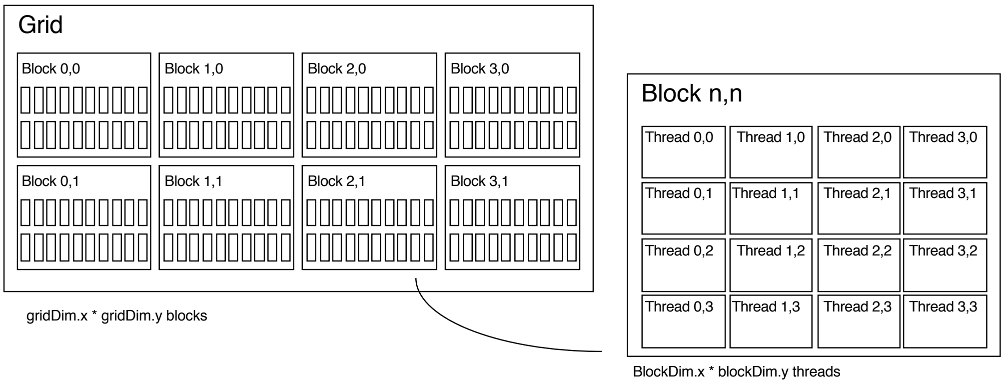
1 block - many threads

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

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);
}
```


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

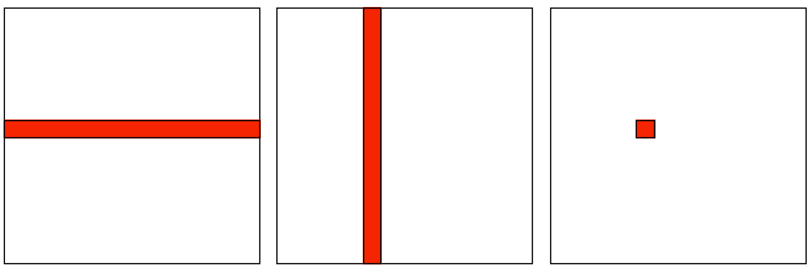
Process

Write back as needed

Shared memory as ”manual cache”

Example: Matrix multiplication

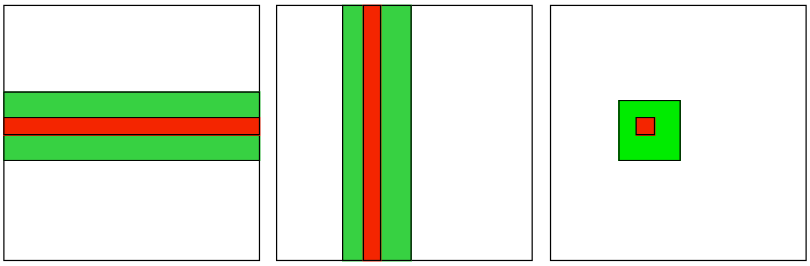
Matrix multiplication



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.

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

Accelerating by coalescing

Pure memory transfers can be 10x faster by taking advantage of memory coalescing!

Example: Matrix transpose

No computations!

Only memory accesses.

Matrix transpose

Naive implementation

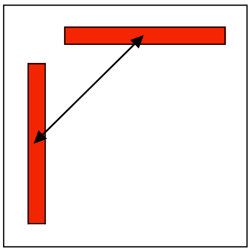
```
__global__ void transpose_naive(float *odata, float* idata, int width, int height)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

   if (xIndex < width && yIndex < height)
   {
       unsigned int index_in  = xIndex + width * yIndex;
       unsigned int index_out = yIndex + height * xIndex;
       odata[index_out] = idata[index_in];
   }
}
```

How can this be bad?

Matrix transpose

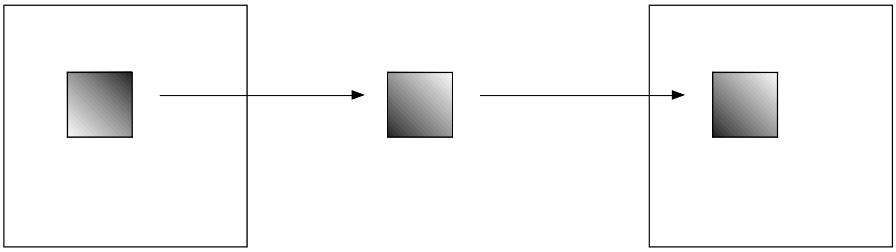
Coalescing problems



Row-by-row and column-by-column.
Column accesses non-coalesced!

Matrix transpose

Coalescing solution



Read from global memory to shared memory

In order from global, any order to shared

Write to global memory

In order write to global, any order from shared

Information Coding / Computer Graphics, ISY, LiTH

Better CUDA matrix transpose kernel

```
__global__ void transpose(float *odata, float *idata, int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

    // read the matrix tile into shared memory
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if((xIndex < width) && (yIndex < height))
    {
        unsigned int index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }

    __syncthreads();

    // write the transposed matrix tile to global memory
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if((xIndex < height) && (yIndex < width))
    {
        unsigned int index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Shared memory for temporary storage

Read data to temporary buffer

Write data to tglobal memory

Coalescing rules of thumb

- The data block should start on a multiple of 64
- It should be accessed in order (by thread number)
- It is allowed to have threads skipping their item
- Data should be in blocks of 4, 8 or 16 bytes

Texture memory

Cached! Can be fast if data access patterns are good.

Texture filtering, linear interpolation.

Especially good for handling 4 floats at a time (float4).

cudaBindTextureToArray() binds data to a texture unit.

Porting to CUDA

1. Parallel-friendly CPU algorithm.
2. Trivial (serial) CUDA implementation.
3. Split to blocks and threads.
4. Take advantage of shared memory.

CUDA emulation mode

CUDA programs can be compiled to CPU only versions.

--device-emulation

Lets you run CUDA (slowly) on non-NVidia hardware

Debugging easier (e.g. printf)

That’s all folks!

Next: Laborations, hands-on experience of all three techniques!