Programming for GPUs

Part 1

Dorothea vom Bruch
Email: dorothea.vom.bruch@cern.ch

Thematic CERN School of Computing, Spring 2021
June 17th 2021
Outline

- From SIMD to SIMT
- Thread and memory organization
- Basic building blocks of a GPU program
- Control flow, synchronization and atomics
Vertex/index buffers: Description of image with vertices and their connection to triangles

Vertex shading
For every vertex: calculate position on screen based on original position and camera view point

Rasterization
Get per-pixel color values

Pixel shading
For every pixel: get color based on texture properties (material, light, ...)

Rendering
Write output to render target

http://fragmentbuffer.com/gpu-performance-for-game-artists/
Vertex/index buffers:
Description of image with vertices and their connection to triangles

Vertex shading
For every vertex: calculate position on screen based on original position and camera view point

Rasterization
Get per-pixel color values

Pixel shading
For every pixel: get color based on texture properties (material, light, ...)

Rendering
Write output to render target

http://fragmentbuffer.com/gpu-performance-for-game-artists/
GPU requirements

- Graphics pipeline: huge amount of arithmetic on independent data:
  - Transforming positions
  - Generating pixel colors
  - Applying material properties and light situation to every pixel

**Hardware needs**

- Access memory simultaneously and contiguously
- Bandwidth more important than latency
- Floating point and fixed-function logic
Mid 2000s

Mid 2000s: unified processors for graphics stages

→ Programmable GPU processors could be used for general purpose computing

Amdahl’s law

\[
\text{Speedup in latency} = \frac{1}{S + \frac{P}{N}}
\]

- **S**: sequential part of program
- **P**: parallel part of program
- **N**: number of processors

- Parallel part: identical, but independent work
- Consider how much of the problem can actually be parallelized to decide whether processing it on a GPU makes sense
## SISD, MIMD & SIMD

<table>
<thead>
<tr>
<th></th>
<th>SISD</th>
<th>MIMD</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single Instruction Single Data</td>
<td>Multiple Instruction Multiple Data</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td></td>
<td>Uniprocessor machines</td>
<td>Multi-core, grid-, cloud-computing</td>
<td>e.g. vector processors</td>
</tr>
</tbody>
</table>

D. vom Bruch
# Single Instruction Multiple Threads (SIMT)

<table>
<thead>
<tr>
<th>SISD</th>
<th>MIMD</th>
<th>SIMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Instruction Single Data</td>
<td>Multiple Instruction Multiple Data</td>
<td>Single Instruction Multiple Threads</td>
</tr>
<tr>
<td>Uniprocessor machines</td>
<td>Multi-core, grid-, cloud-computing</td>
<td>GPUs</td>
</tr>
</tbody>
</table>

**Diagrams:**

- **SISD**: Single Instruction Single Data
- **MIMD**: Multiple Instruction Multiple Data
- **SIMT**: Single Instruction Multiple Threads

**Examples:**

- **PU**: Processing Unit
- **Data Pool**: Source of data for processing

*D. vom Bruch*
SIMD versus SIMT

**SIMD**
- Vectorized instructions executed on modern CPU SIMD cores are executed in lockstep
- No synchronization barrier is needed, as all elements of the vector finish processing at the same time

**SIMT**
- Similar to programming a vector processor
- Use threads instead of vectors
- No need to read data into vector register
- GPUs consist of multiple processing elements, each with multiple SIMT GPU cores → not all threads are processed in lockstep
- A synchronization instruction is required on GPUs
What is a GPU?

- Several processors are grouped into a “multiprocessor”
- Several multiprocessors make up a GPU

(CUDA terminology)
Nvidia Turing architecture

- PCIe interface
- Memory controller
- Streaming Multiprocessor
- NVLink interface
Nvidia Turing: Streaming Multiprocessor

- **Scheduler**
- **64 Single precision cores (FP32)**
- **64 Integer cores (INT32)**
- **Tensor cores**
- **Ray tracing cores (RT)**

Nvidia Turing GPU architecture
GPU Programming Environments

Early days: Problems had to be translated to graphics language via OpenGL
Today: several programming interfaces exist

- Nvidia’s application programming interface: CUDA
  - Only works with Nvidia GPUs
  - Very well documented, many tutorials, low entry level
- AMD ROCm (HIP): Open source platform for GPU computing
  - Supports both AMD and Nvidia GPUs
  - New development → still work in progress, not that many examples / tutorials yet
- OpenCL: Framework for heterogeneous platforms
  - CPUs, GPUs, FPGAs, DSPs, etc.
  - Maintained by the Khronos group, based on C99 and C++11
- SYCL: Single source C++ heterogeneous programming platform, built on OpenCL
  - Will be supported by Intel GPUs
Focus of GPU programming lectures: CUDA

- Widely used in the GPU computing community
- Underlying concepts easily translate to the other programming interfaces
- Lecture on Friday will cover other environments
- Very similar to C/C++ code

- CUDA programming guide
Parallelization

- Any GPU code we write will be executed on many “threads” at once
- These threads are organized in a “grid”, where a fixed set of threads is grouped into one “block”
- Each thread processes the same instructions (kernel), but on different data
- Up to three dimensions for blocks and threads
- Maximum of 1024 threads / block (check specs of GPU)
Example: Parallelization for LHCb’s HLT1

- GPUs provide two levels of parallelization
- Ideally suited for LHCb’s HLT1
- Assign events to blocks
- Intra-event parallelization: threads within one block
- Every thread processes for example
  - Decoding of one detector element
  - 3-hit combination in the pattern recognition step
  - One track candidate
  - One vertex candidate
  - ...

Within one block: intra-event parallelization

Individual events
Assignment to Streaming Multiprocessors

- Execution order of blocks is arbitrary
- Scheduled on Streaming Multiprocessors (SMs) according to resource usage: memory, registers, thread number limit
Threads within a block assigned to one SM are processed in “warps”
- A warp is an entity of 32 threads on Nvidia GPUs
- Recent AMD GPUs use warps of 64 threads
- Warps are the smallest entity on a GPU, i.e. no less than the number of threads in one warp is processed
- The block size should be chosen to be at least 32 (64) threads and ideally a multiple of the warp size
- This ensures that no threads are inherently idle
Memory layout

Specs from the 16 GB Tesla V100

- **Global Memory**: 16 GB
- **Constant Memory**: 64 kB
- **Shared Memory**: 98 kB
- **Texture Cache, L2 Cache**: O (kB)

**Thread 0**
- Registers
- Block 0
- Thread 0

**Thread 1**
- Registers
- Block 1
- Thread 1

**Thread 0**
- Registers
- Block 0
- Thread 0

**Thread 1**
- Registers
- Block 1
- Thread 1

**Host**

- **Extremely fast, highly parallel**
- **Fastest, limited to 65536 registers per block**
- **High access latency (400 - 800 cycles)**
- **Finite access bandwidth**
- **Read only, short latency**

D. vom Bruch
Memory usage

- **Global memory:**
  - Main memory, accessible from everywhere
  - Communication with host

- **Constant memory:**
  - Secondary, can be used to store constants
  - Only writeable to from host

- **Shared memory:**
  - Communication among threads within one block
  - Copy data from global to shared memory for faster access
  - Especially when used by several threads in a block
  - Accessible only from one block on the device

- **Registers:**
  - Accessible only from within a single thread
  - All variables declared inside a kernel are automatically stored in registers
  - Too many registers can result in performance penalty
## Memory overview

<table>
<thead>
<tr>
<th>Name</th>
<th>Host access</th>
<th>Device access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global memory</td>
<td>Dynamic allocation, Read / write</td>
<td>No allocation, Read / write</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Dynamic allocation, Read / write</td>
<td>Static allocation, Read-only</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Dynamic allocation, No access</td>
<td>Static allocation, Read / write access by all elements of a block</td>
</tr>
<tr>
<td>Registers &amp; local memory</td>
<td>No allocation, No access</td>
<td>Static allocation, Read / write access by a single thread</td>
</tr>
</tbody>
</table>
Configuration considerations

- Within one block:
  - Use same shared memory
  - Can synchronize all threads in one block
- Threads in different blocks:
  - Cannot communicate
  - Only through content of global memory
- Grid size:
  - > 2 x number of SMs → hide latencies
- Block size:
  - Consider number of registers used per thread
    → Number of registers / block is limited
  - Optimum: multiple of 32 (warp size) → no inherently idle threads
CUDA has specific variables & functions introduced for
• Identification of GPU code
• Allocation of GPU memory
• Definition of thread grid size
• Options to launch the grid
• …

Kernel: program containing instructions to be executed on the GPU

Host
• Some CPU code
  • Memory allocation (host & device)
    …
  • Launch grid of kernels to run on GPU
    …
  • Some more CPU code
  • Memory deallocation (host & device)

Device
Run kernels
Calling a function in CUDA

```c
/* dim3: CUDA specific variable to declare size of grid in blocks and threads, 
   can take up to three arguments for 3-dimensional grids and blocks */
dim3 blocks(n_blocks);
dim3 threads(n_threads);

/* Syntax to launch a kernel: 
   <<< size of grid in blocks and threads >>> 
   (): any parameters to be passed to the kernel */
hello_world_kernel<<<blocks,threads>>>();

/* Blocks until all requested tasks on device were completed; 
   needed for printf in kernel to work */
cudaDeviceSynchronize();
```

Non blocking function call
Will return to host immediately

Waits for previously launched device work to finish
Simplest CUDA function

Identifier of function executed on the GPU

```
__global__ void hello_world_kernel( void ) {

    /* blockIdx.x: Accesses index of block within grid in x direction
    threadIdx.x: Accesses index of thread within block in x direction
    */
    if ( blockIdx.x < 100 && threadIdx.x < 100 )
        printf("Hello World from block %u, thread %u \n", blockIdx.x, threadIdx.x);
}
```

blockIdx and threadIdx are defined within device code

Can access blockIdx.x, blockIdx.y, blockIdx.z

If only 1-dimensional block is defined, blockIdx.y = 1, blockIdx.z = 1

Only method to pass messages to stdout from device code is printf (std::cout does not work)
What does the parallelization mean?

hello_world_kernel
blockIdx.x = 0
threadIdx.x = 0

hello_world_kernel
blockIdx.x = 0
threadIdx.x = 1

...
Pre-defined variables available in kernel

- blockIdx.x, blockIdx.y, blockIdx.z
- blockDim.x, blockDim.y, blockDim.z
- threadIdx.x, threadIdx.y, threadIdx.z
- blockDim.x, blockDim.y, blockDim.z
- threadIdx.x, threadIdx.y, threadIdx.z
## Function declaration

<table>
<thead>
<tr>
<th><strong>global</strong></th>
<th>Called from</th>
<th>Executed on</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Host</td>
<td>Device</td>
<td>Defines kernel, returns void</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>Device</td>
<td>Device</td>
<td>Like any C(++) function</td>
</tr>
<tr>
<td><strong>host</strong></td>
<td>Host</td>
<td>Host</td>
<td></td>
</tr>
</tbody>
</table>

__device__ __host__ can be combined
useful if same function is executed on host AND device
Global memory management

```c
int a_host = 8, b_host = 0;
int *a_dev, *b_dev;

cudaMalloc((void**)&a_dev, sizeof(int));
cudaMalloc((void**)&b_dev, sizeof(int));
cudaMemcpy(a_dev, &a_host, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(b_dev, &b_host, sizeof(int), cudaMemcpyHostToDevice);
DoStuff<<<1,1>>>(a_dev, b_dev);

cudaMemcpy(&b_host, b_dev, sizeof(int), cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
cudaFree(a_dev);
cudaFree(b_dev);
```
Global memory management (continued)

int a_host = 8, b_host = 0;
int *a_dev, *b_dev;

cudaMalloc((void**)&a_dev, sizeof(int));
cudaMalloc((void**)&b_dev, sizeof(int));
cudaMemcpy(a_dev, &a_host, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(b_dev, &b_host, sizeof(int), cudaMemcpyHostToDevice);

DoStuff<<<1,1>>>(a_dev, b_dev);

cudaMemcpy(&b_host, b_dev, sizeof(int), cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();

cudaFree(a_dev);
cudaFree(b_dev);
Global memory management (continued)

```c
int a_host = 8, b_host = 0;
int *a_dev, *b_dev;

cudaMalloc( (void**)&a_dev, sizeof(int) );
cudaMalloc( (void**)&b_dev, sizeof(int) );

cudaMemcpy( a_dev, &a_host, sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( b_dev, &b_host, sizeof(int), cudaMemcpyHostToDevice );

DoStuff<<<1,1>>>( a_dev, b_dev );

cudaMemcpy( &b_host, b_dev, sizeof(int), cudaMemcpyDeviceToHost );
CudaDeviceSynchronize();

cudaFree( a_dev);
cudaFree( b_dev);
```

Pointers to global memory variables passed to kernel

Pointer to global memory to be freed
Synchronization: Grid level

- Execution order of blocks on SMs is arbitrary
- If we want to ensure that all work has finished, need a synchronization method
- Call to `CudaDeviceSynchronize()` from host waits until all work launched on the device has finished
  - Includes kernel launches (i.e. all instances of `hello_world_kernel` in all blocks have finished)
  - Also includes memory copies

```plaintext
hello_world_kernel
blockIdx.x = 0
threadIdx.x = 0

hello_world_kernel
blockIdx.x = 0
threadIdx.x = 1

... hello_world_kernel
blockIdx.x = 0
threadIdx.x = N
```
Synchronization: Block level

- Execution order of threads within one block is arbitrary
- Only exception: threads in one warp are processed jointly
- To synchronize threads within one block: Call __syncthreads() within the kernel code

```
for (int i = threadIdx.x; i < N; i++) {
    Fill variable[threadIdx.x]
}
__syncthreads();
for (int i = threadIdx.x; i < N; i++) {
    Use variable[threadIdx.x]
}
```
Static shared memory

- Shared memory is allocated within the kernel
- If the size is known at compile time, it is declared with that size directly in the kernel
- Call to `__syncthreads()` is usually needed if results computed with other threads are needed

```c
__global__ void my_kernel(float *my_other_result) {
    __shared__ float var_sh[N];
    for (int i = threadIdx.x; I < N; i++) {
        var_sh[i] = …;
    }
    __syncthreads();
    for (int i = threadIdx.x; I < N; i++) {
        my_other_result[N] = something with var_sh[i]
    }
}
my_kernel<<1024,32>>(my_other_result);
```
Dynamic shared memory

If the size is only known at run time, shared memory can be allocated dynamically.

The size must be known on the host.

It is passed as additional argument to the kernel call.

The amount of shared memory per block is the same for all blocks within one grid.

```c
__global__ void my_kernel(float *my_other_result) {
    extern __shared__ float var_sh[];
    for (int i = threadIdx.x; i < N; i++) {
        var_sh[i] = ...;
    }
    __syncthreads();
    for (int i = threadIdx.x; i < N; i++) {
        my_other_result[N] = something with var_sh[i]
    }
}

my_kernel<<1024,32, N*sizeof(float)>> (my_other_result);
```
Race conditions → atomic operations

- Caution when modifying the same value in memory from different threads:
  - Need to read, modify, write value: three operations
  - Outcome depends on timing of the different threads
  - Thread 1 can modify after thread 2 read a value, but before thread 2 writes a new value!
- Use atomic operations:
  - Read-modify-write cannot be interrupted: appears to be one operation
    - atomicAdd(), atomicSub(), atomicInc(), atomicDec(), …
- Needed for both shared and global memory
Index calculation

- It is often useful to parallelize the processing of one array with both blocks and threads.
- Unique index = \( x + y \times \text{size} \)
- \( \text{int index} = \text{threadIdx.x} + \text{blockIdx.x} \times \text{blockDim.x}; \)
Compilation

- Use `nvcc` for compilation:
  - Calls `nvcc` for CUDA parts
  - Calls `gcc` for C++ parts
- `nvcc FirstProgram.cu -o executableName`
- Also takes C, C++, library, object, shared object... files as input
- Can link libraries, include header files
- Can integrate into larger projects with CMake
Resources

- J. Sanders, E. Kandrot: “CUDA by Example”
- N. Wilt: “The CUDA Handbook”
Summary

- GPU architecture uses SIMT paradigm: threads process same instruction on independent data
- Parallelization occurs on two levels: blocks and threads
- Assignment of blocks to Streaming Multiprocessors based on resource usage
- Memory hierarchy similar to CPU memory, but explicitly chosen by programmer
- Execution order of threads and blocks is random → synchronization required by programmer
- Few special functions in CUDA to express parallelization, memory type and synchronization
- Pipelines (“streams”) allow to hide overhead due to memory copies between host and device
- Pay attention to race conditions when several streams access the same memory location

- Main concept is that of many threads doing work in parallel
- Need to develop algorithm expressing the parallelism
- Coding itself is mainly C / C++
Backup
SoA vs. AoS

Array of Structs (AoS)

Struct of Arrays (SoA)
Control flow

- Hide memory copies between host and device by using several pipelines
- CUDA terminology for pipeline: “stream”
Synchronization with streams

- If no streams are explicitly defined, the “default” stream is used
- To use several streams as pipelines, need to create them specifically

```c
cudaStream_t streams[num_streams];
for (int i = 0; i < num_streams; i++) {
    cudaStreamCreate(&streams[i]);
    cudaMalloc(&data_d[i], N * sizeof(float));
    my_kernel<<1024,32, 0, streams[i]>>(data_d[i],N);
    cudaMemcpyAsync(data_h[i], data_d[i], N * sizeof(float), stream[i]);
}
```

- `cudaDeviceSynchronize()` waits for all streams to have finished
- `cudaStreamSynchronize(stream[i])` waits only for `stream[i]` to finish