Programming for GPUs

Part 1

Dorothea vom Bruch
Aix Marseille Univ, CNRS/IN2P3, CPPM, Marseille, France
Email: dorothea.vom.bruch@cern.ch

Thematic CERN School of Computing
June 2023
Split, Croatia

Outline

- From SIMD to SIMT
- Thread and memory organization
- Basic building blocks of a GPU program
- Control flow, synchronization and atomics

Graphics Programming Unit

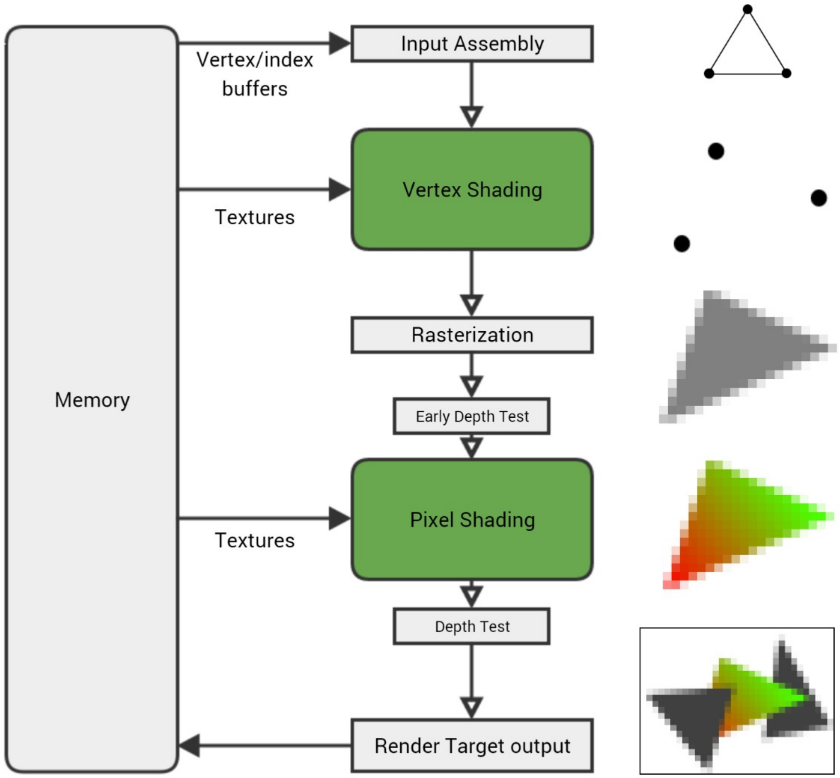
Vertex/index buffers:
Description of image with vertices and their connection to triangles

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/

Graphics Programming Unit

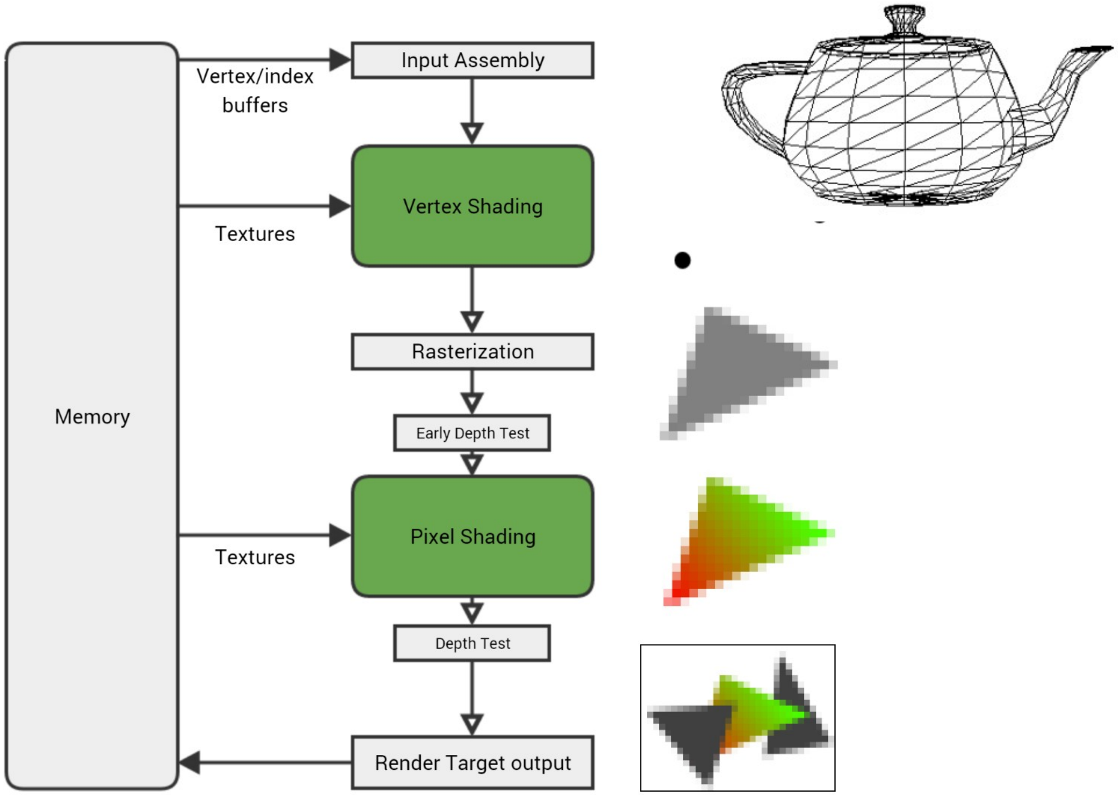
Vertex/index buffers:
Description of image with vertices and their connection to triangles

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



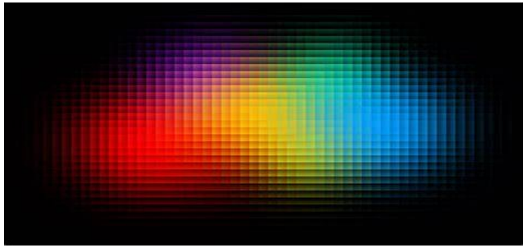
http://fragmentbuffer.com/gpu-performance-for-game-artists/

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

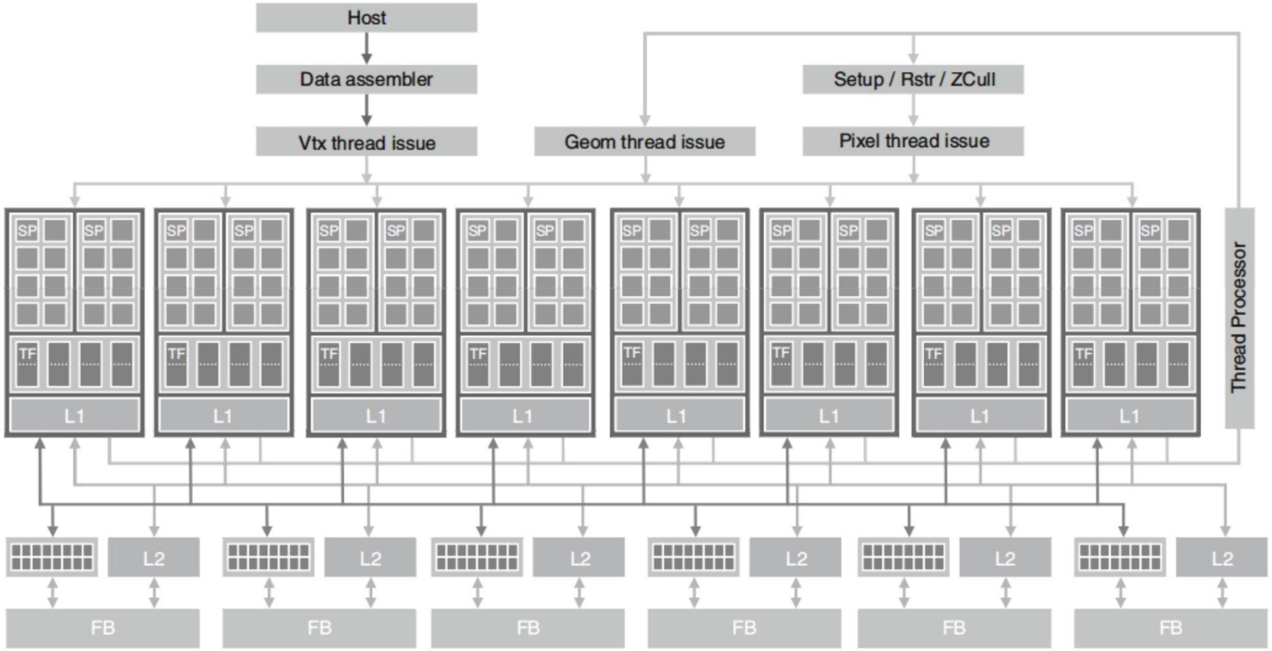
Hardware needs

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



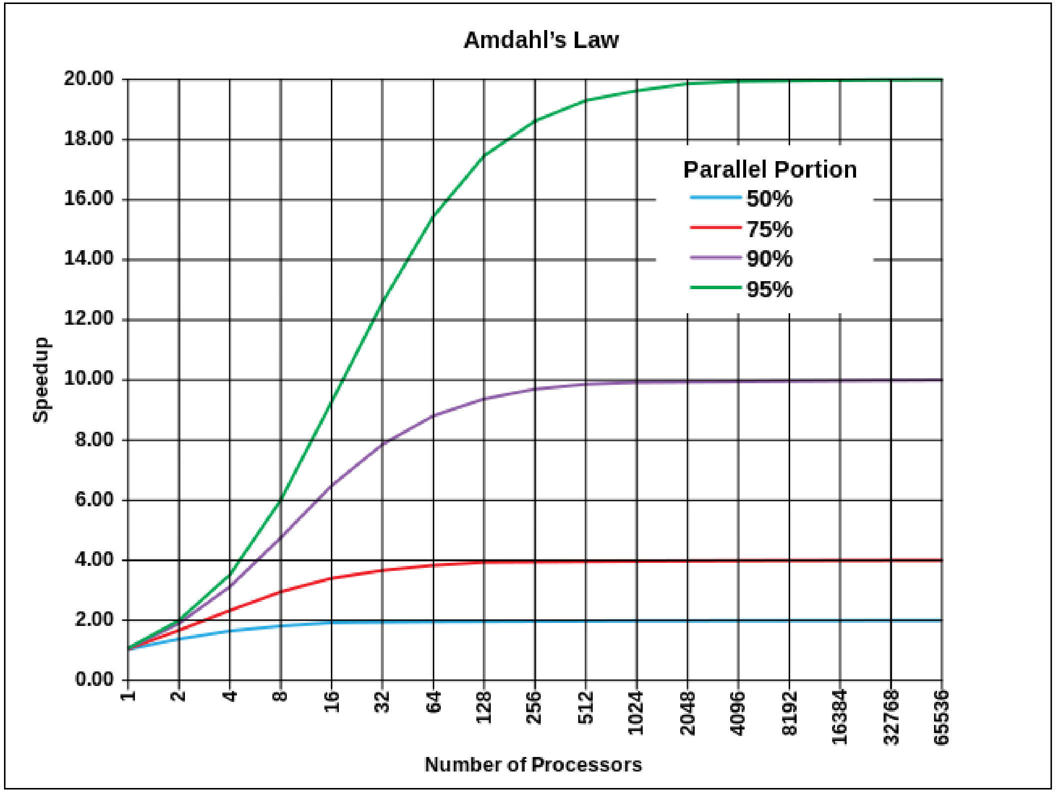
General purpose computing with GPUs

Mid 2000s: unified processors for graphics stages
→ Programmable GPU processors could be used for general purpose computing



From: “Programming Massively Parallel Processors”, D. B. Kirk, W. W. Hwu, 2013, p. 32

Amdahl’s law



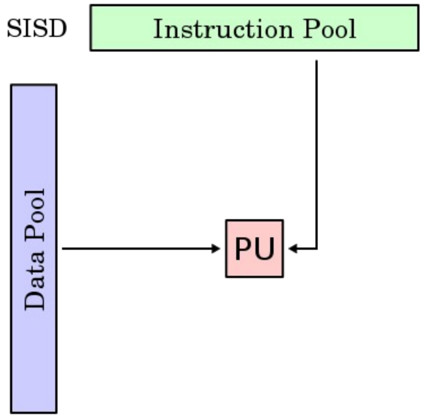
Speedup in latency = 1 / (S + P/N)

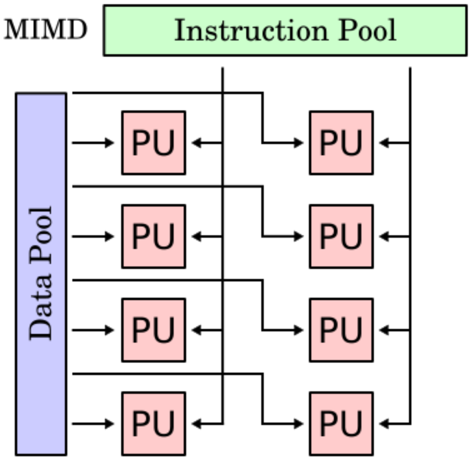
- S: sequential part of program
- P: parallel part of program
- N: number of processors

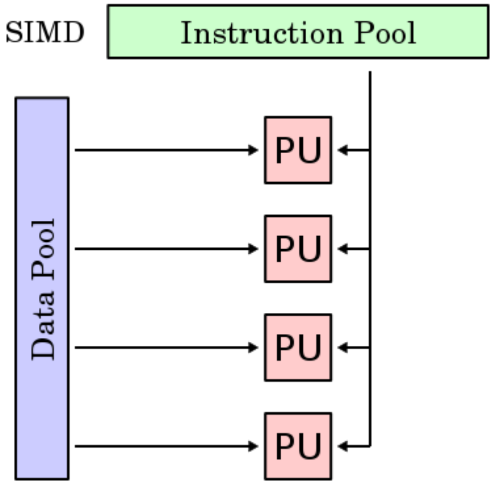
- Parallel part: identical, but independent work
- Consider how much of the problem can actually be parallelized to decide whether processing it on a GPU makes sense

SISD, MIMD & SIMD

SISD	**MIMD**	**SIMD**
Single Instruction Single Data	Multiple Instruction Multiple Data	Single Instruction Multiple Data
Uniprocessor machines	Multi-core, grid-, cloud-computing	e.g. vector processors

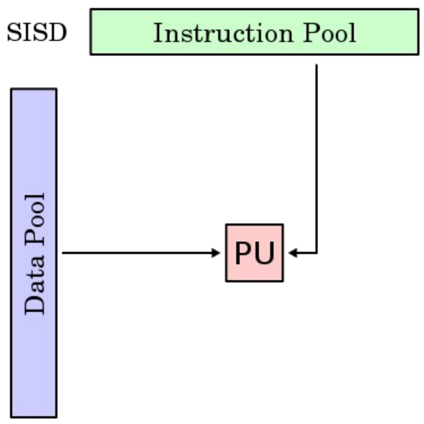


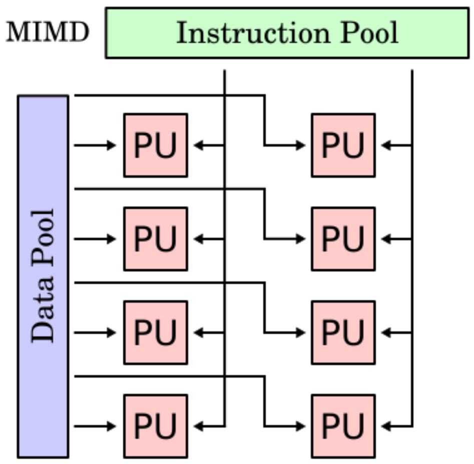


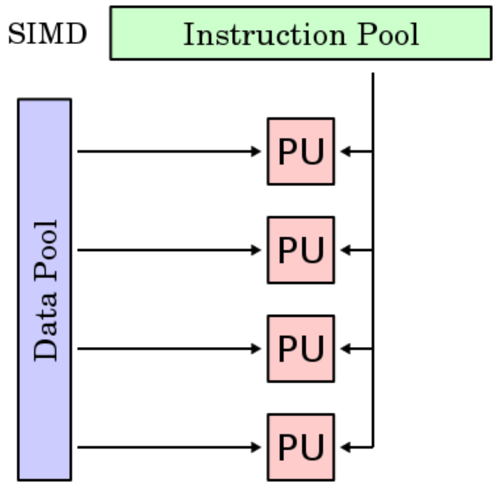


Single Instruction Multiple Threads (SIMT)

SISD	MIMD	SIMT
Single Instruction Single Data	Multiple Instruction Multiple Data	Single Instruction Multiple Threads
Uniprocessor machines	Multi-core, grid-, cloud-computing	GPUs







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

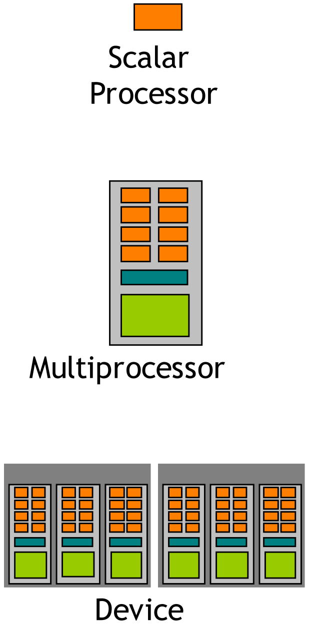
SIMT

- 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?

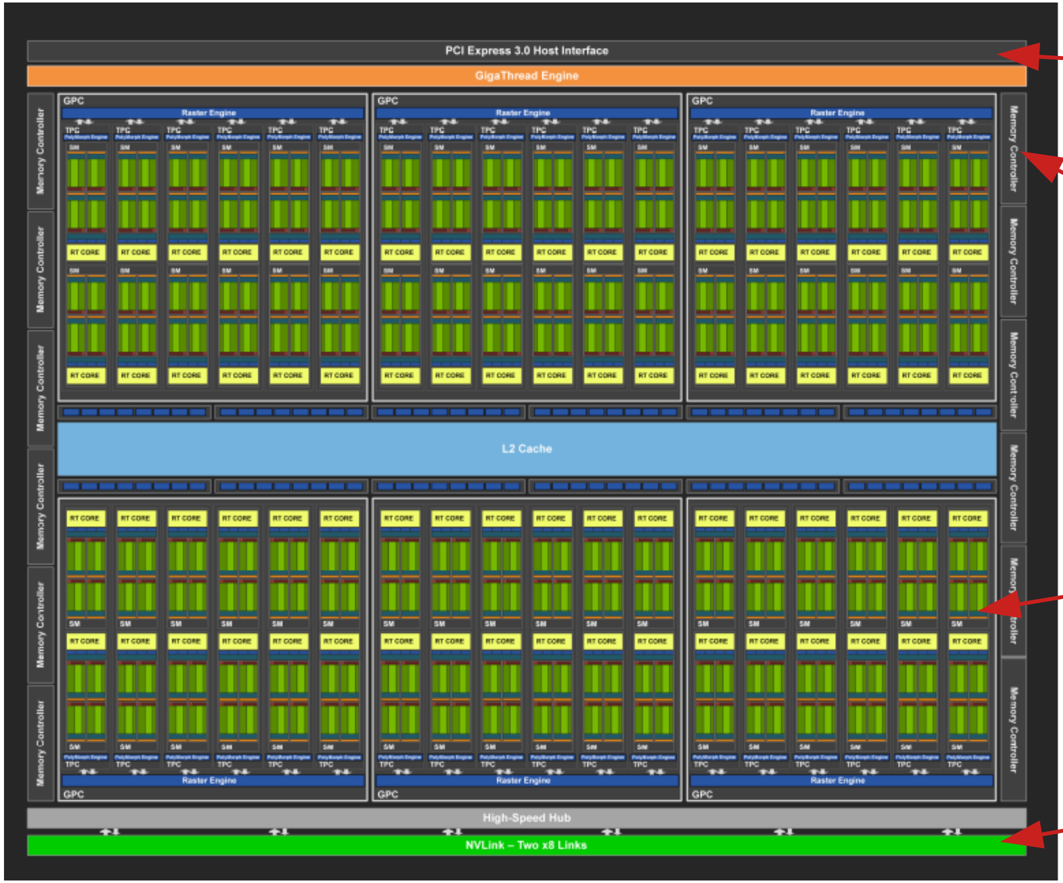
Hardware



(CUDA terminology)

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

Nvidia Turing architecture



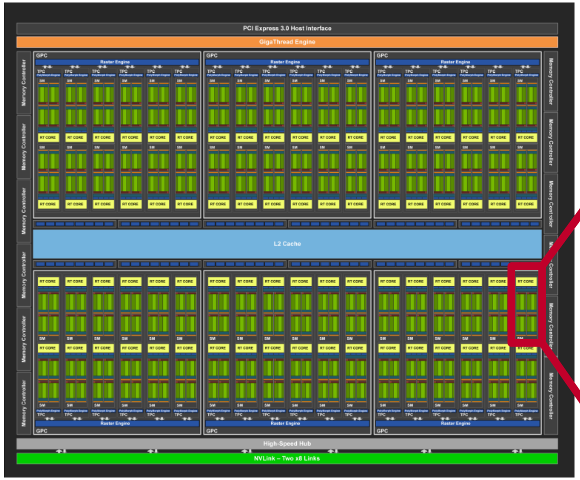
PCIe interface

Memory controller

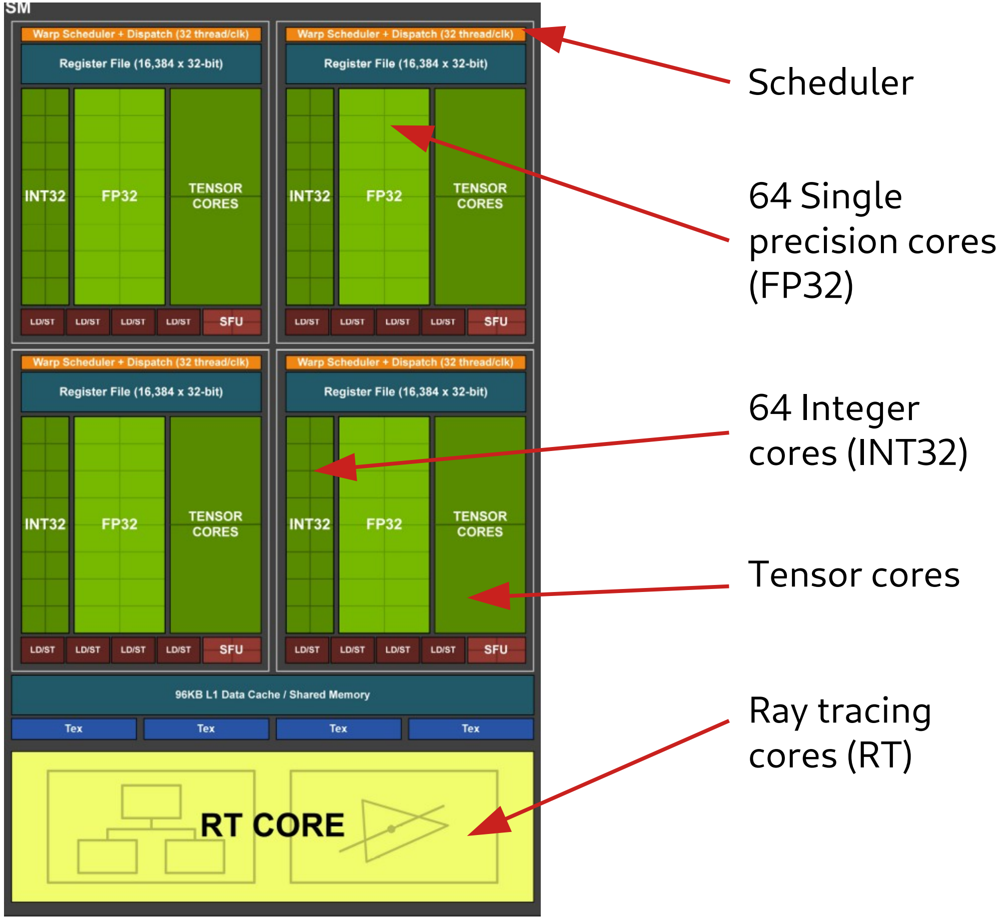
Streaming Multiprocessor

NVLink interface

Nvidia Turing: Streaming Multiprocessor



Nvidia Turing GPU architecture



GPU Programming Environments

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

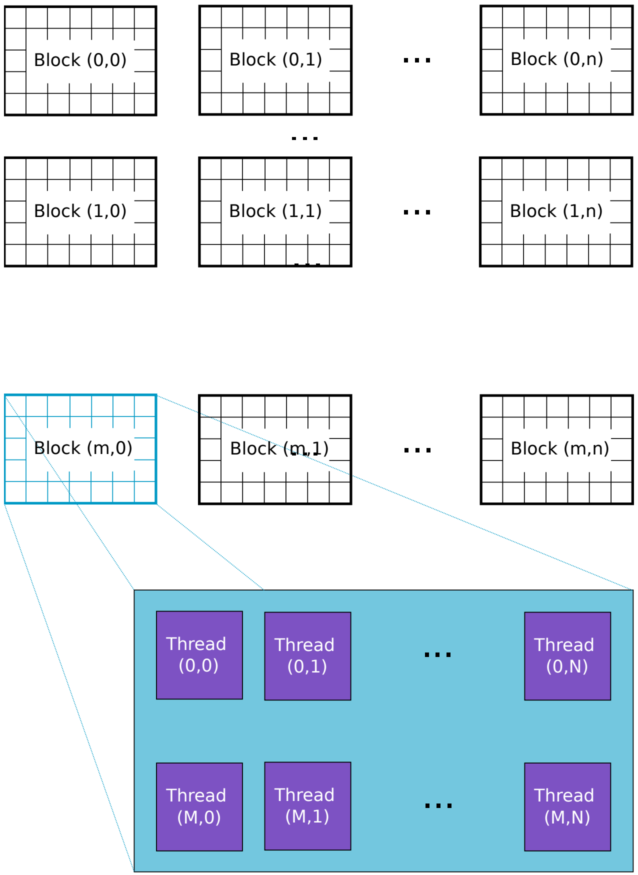
- Nvidia's application programming interface: CUDA
  - Only works with Nvidia GPUs
  - Very well documented, many tutorials, low entry level
- AMD ROCm (HIP): Open source platform for GPU computing
  - Supports both AMD and Nvidia GPUs
  - New development → still work in progress, not that many examples / 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 by D. Campora will cover other environments
- Very similar to C/C++ code

- CUDA programming guide

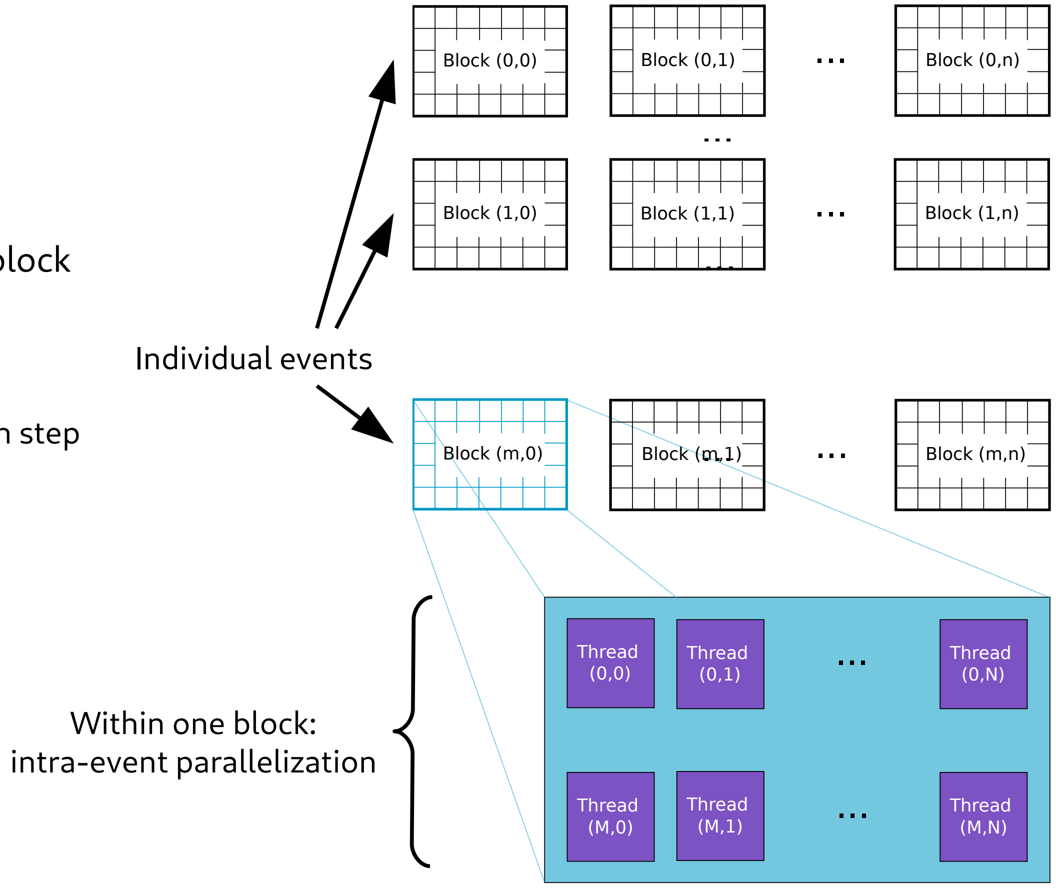
Parallelization



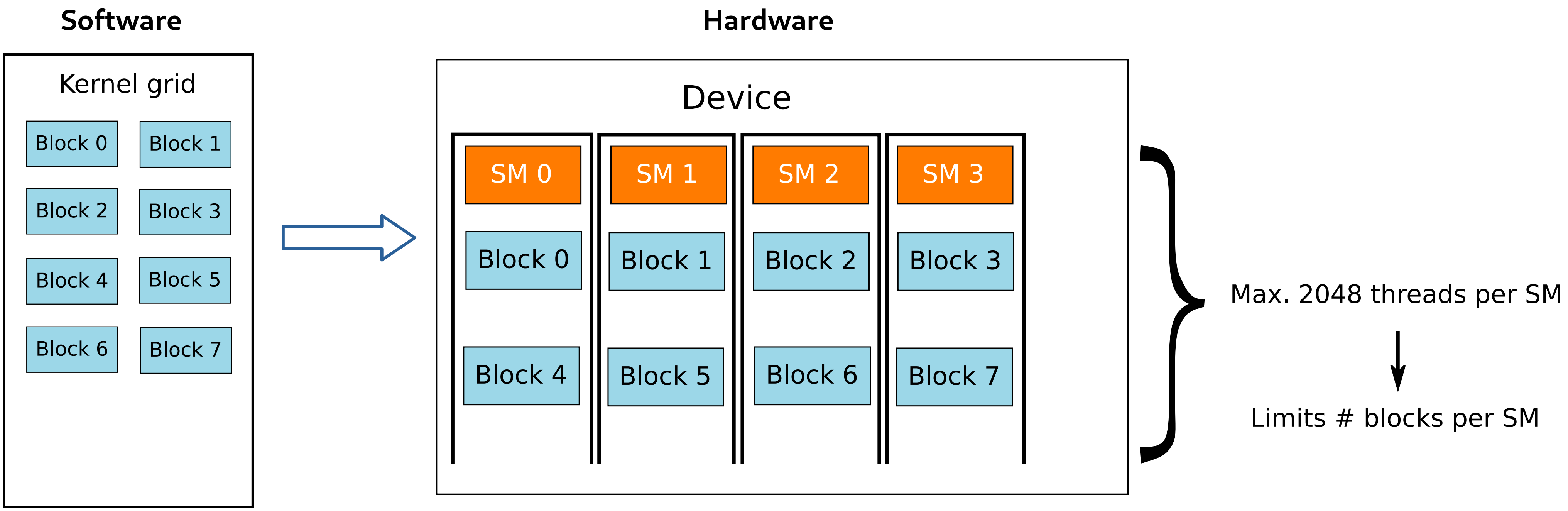
- Any GPU code we write will be executed on many "threads"
- These threads are organized in a "grid", where a fixed set of threads is grouped into one "block"
- Each thread processes the same instructions (kernel), but on different data
- Up to three dimensions for blocks and threads
- Maximum of 1024 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
  - ...

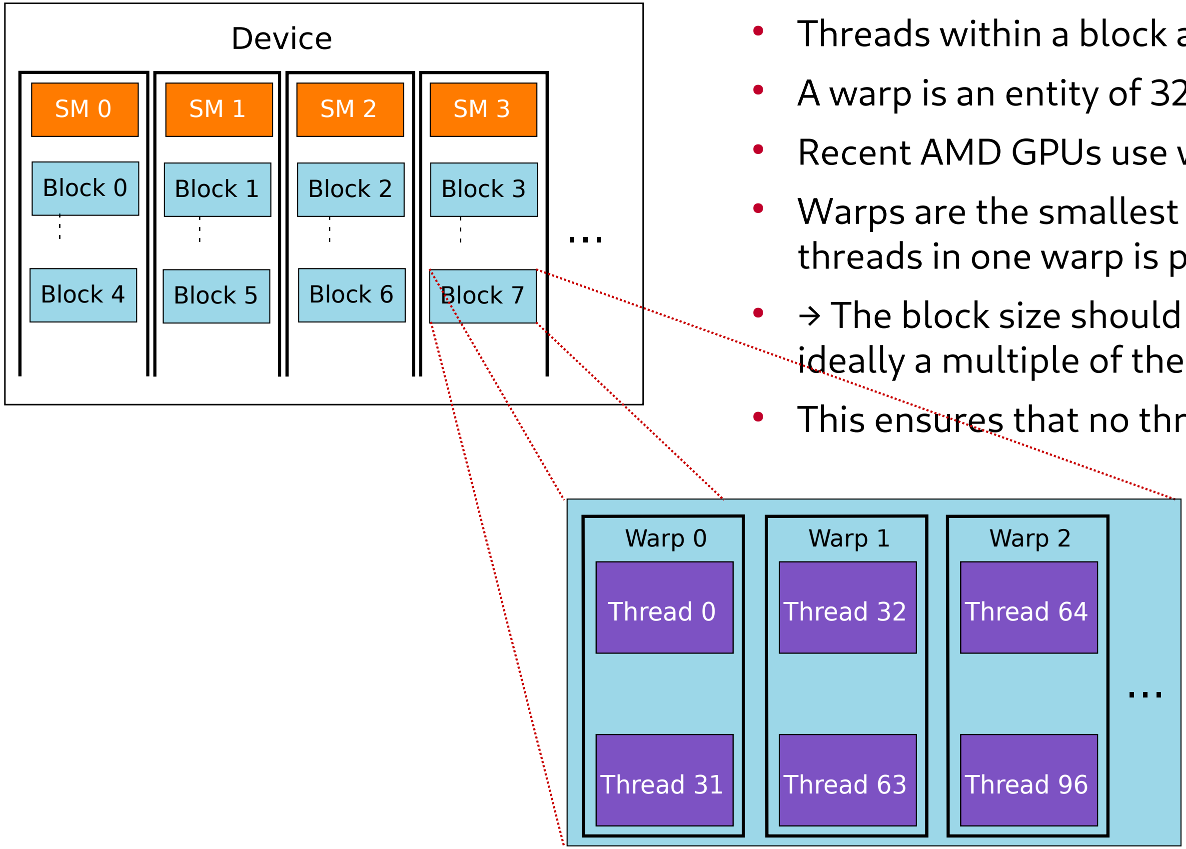


Assignment to Streaming Multiprocessors



- Execution order of blocks is arbitrary
- Scheduled on Streaming Multiprocessors (SMs) according to resource usage: memory, registers, thread number limit

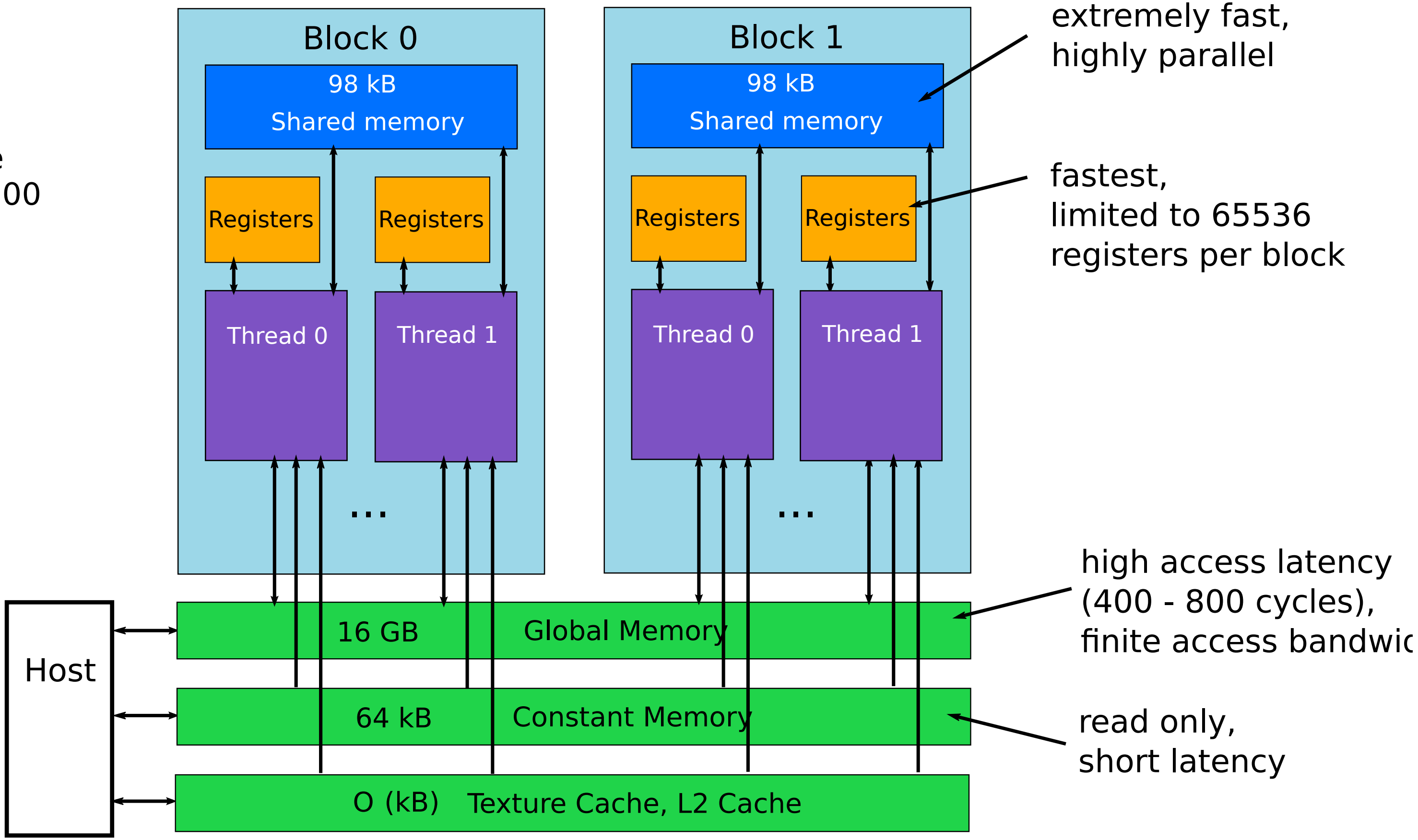
Assignment to warps



- 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

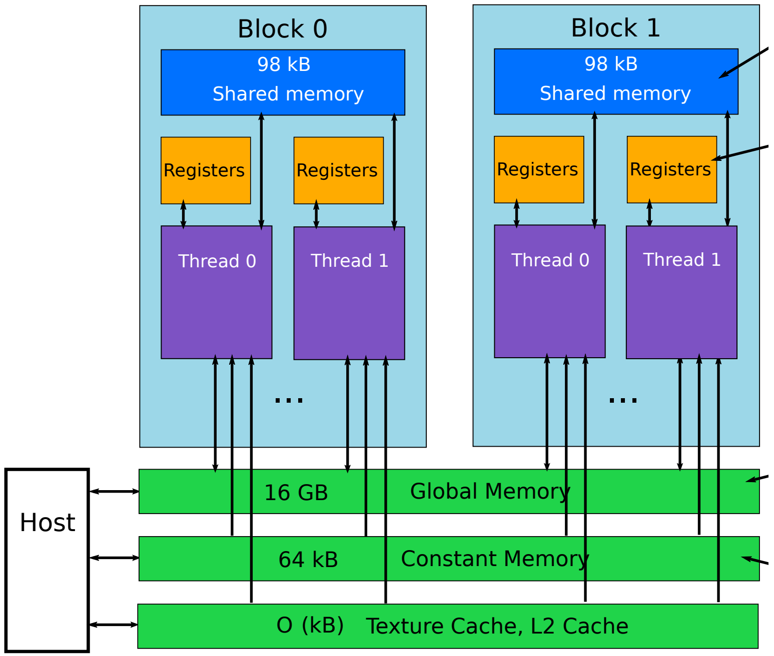
Memory layout

Specs from the
16 GB Tesla V100



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

Name	Host access	Device access
Global memory	Dynamic allocation, Read / write	No allocation, Read / write
Constant memory	Dynamic allocation, Read / write	Static allocation, Read-only
Shared memory	Dynamic allocation, No access	Static allocation, Read / write access by all elements of a block
Registers & local memory	No allocation, No access	Static allocation, Read / write access by a single thread

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

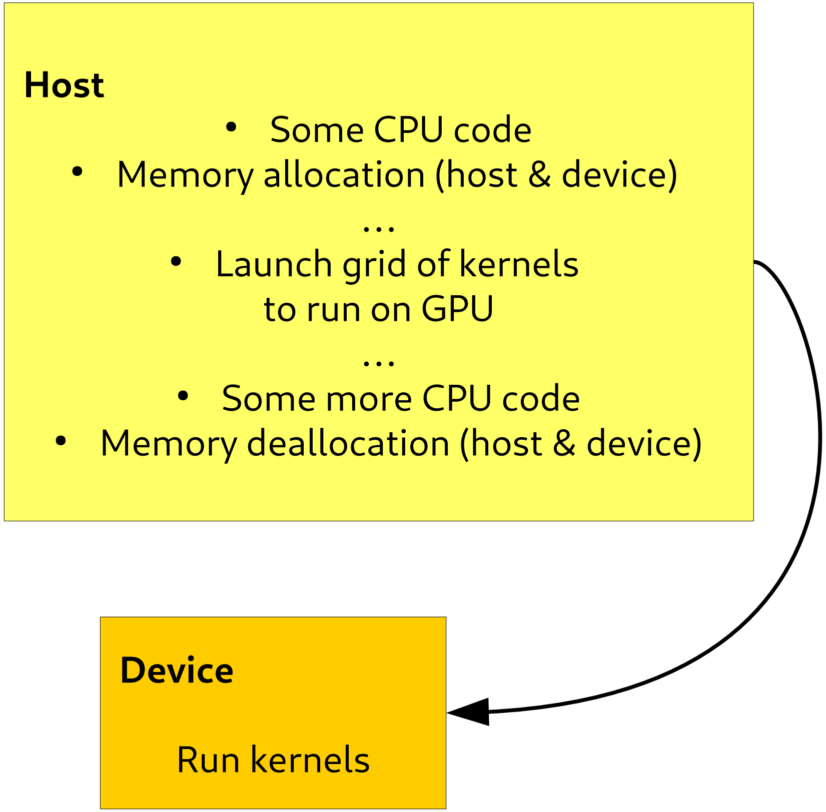
    → Number of registers / block is limited
  - Optimum: multiple of 32 (warp size) → no inherently idle threads

CPU – GPU communication

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



Calling a function in CUDA

```
/* 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

`blockIdx` and `threadIdx` are defined within device code

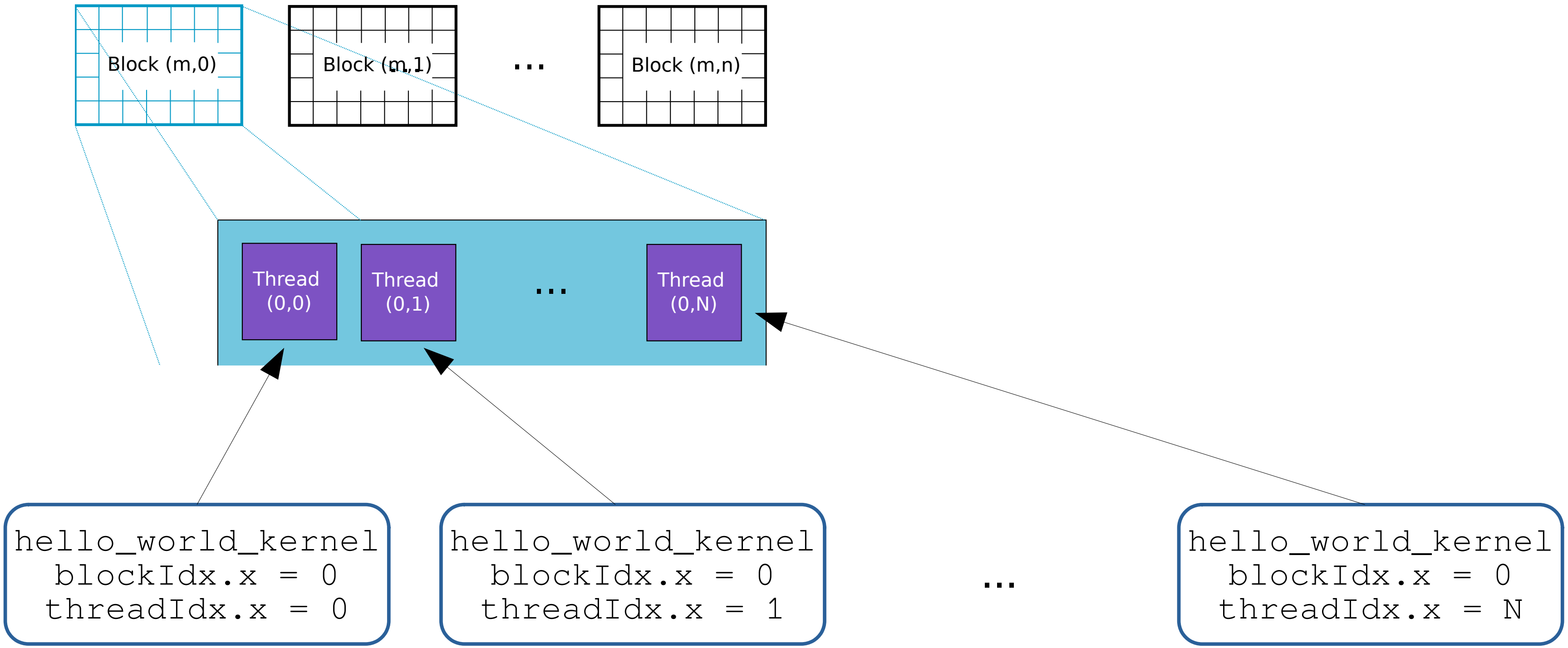
Can access `blockIdx.x`, `blockIdx.y`, `blockIdx.z`

If only 1-dimensional block is defined, `blockIdx.y = 0`, `blockIdx.z = 0`

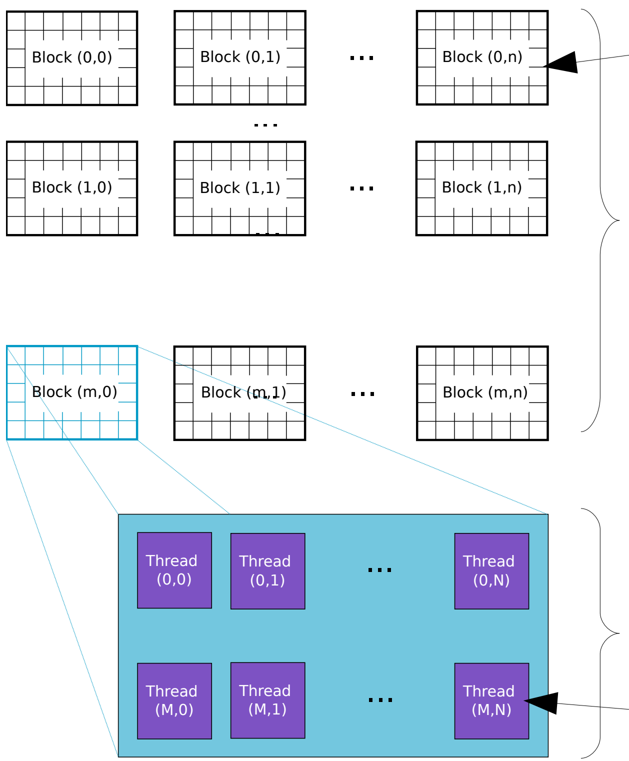
```
__global__ void hello_world_gpu() {

  /* 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 the GPU at block %u, thread %u \n", blockIdx.x, threadIdx.x);

}
```

Only method to pass messages to stdout from device code is `printf` (std::cout does not work)

What does the parallelization mean?



Pre-defined variables available in kernel



`blockIdx.x = n, blockIdx.y = 0, blockIdx.z = 0`

`gridDim.x = n+1, gridDim.y = m+1, gridDim.z = 1`

`blockDim.x = N+1, blockDim.y = N+1, blockDim.z = 1`

`threadIdx.x = N, threadIdx.y = M, threadIdx.z = 0`

Function declaration

	Called from	**Executed on**	**Comment**
`__global__`	Host	Device	Defines kernel, returns void
`__device__`	Device	Device	Like any C(++) function
`__host__`	Host	Host	

`__device__` `__host__` can be combined

useful if same function is executed on host AND device

Global memory management

```
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<<<16,16>>>( a_dev, b_dev );

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

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

Pointer to allocated global memory on device is returned

Size of memory to be allocated

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<<<16,16>>>( a_dev, b_dev );

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

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

Pointer to destination

Pointer to source

Size of memory to be copied (bytes)

Copy direction

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<<<16,16>>>( a_dev, b_dev );
```

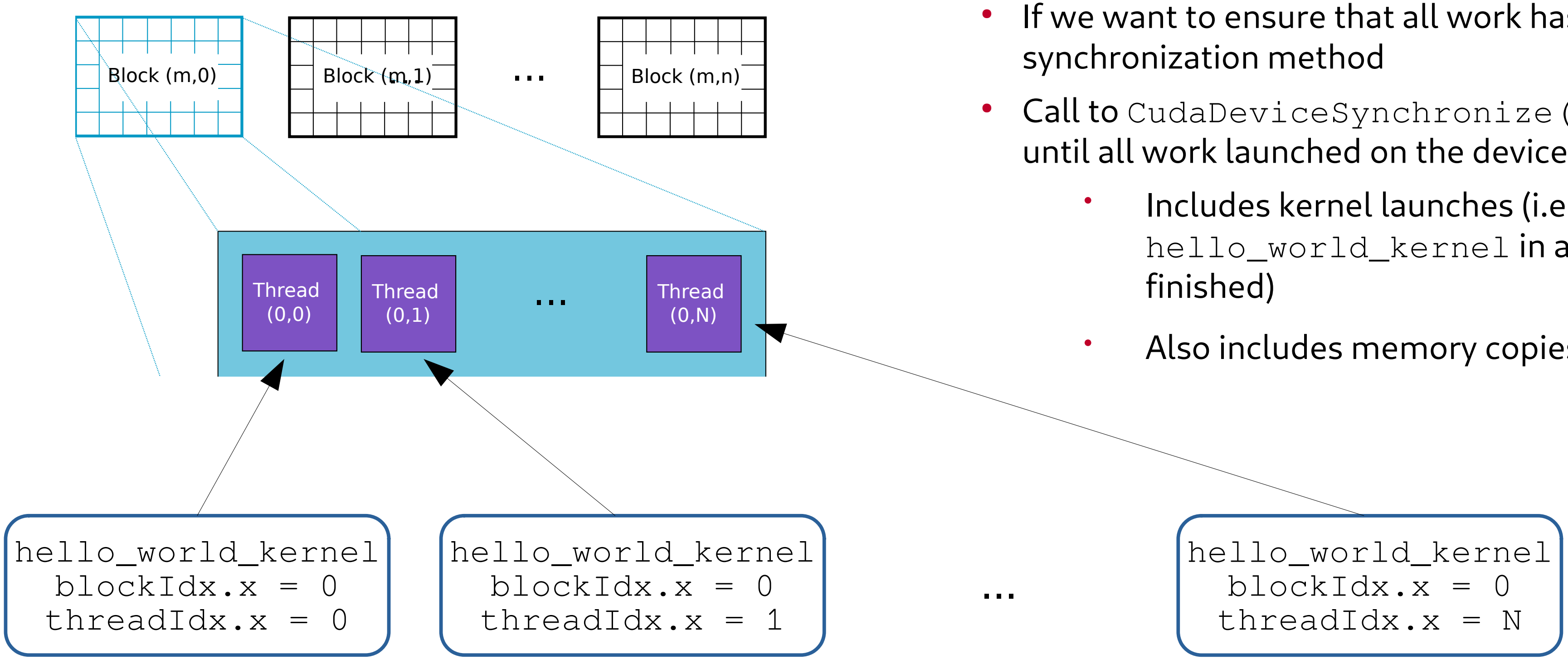
Pointers to global memory variables passed to kernel

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

cudaFree( a_dev);
```

Pointer to global memory to be freed

```
cudaFree( b_dev);
```

Synchronization: Grid level

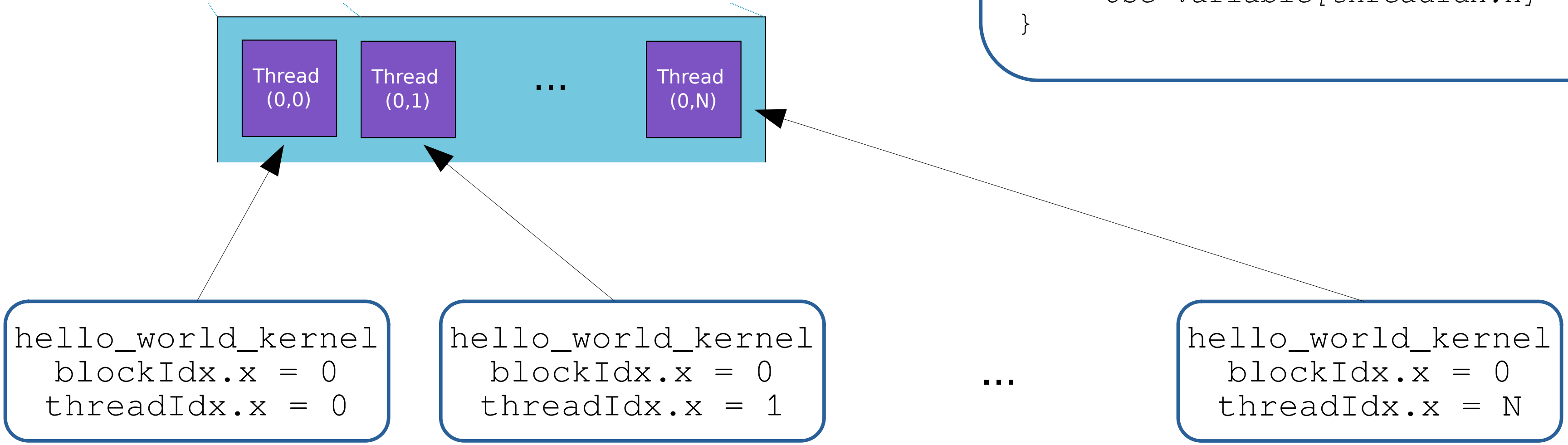


- Execution order of blocks on SMs is arbitrary
- If we want to ensure that all work has finished, need a synchronization method
- Call to `CudaDeviceSynchronize()` from host waits until all work launched on the device has finished
  - Includes kernel launches (i.e. all instances of `hello_world_kernel`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

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+1; i++) {
      variable[threadIdx.x] = ...
}

__syncthreads();

for (int i = threadIdx.x; i < N+1; i++) {
      Use variable[threadIdx.x]
}
```



Static shared memory

```
__global__ void my_kernel(float *my_other_result) {
    __shared__ float var_sh[N+1];

    for (int i = threadIdx.x; i < N+1; i++) {
        var_sh[i] = …;
    }

    __syncthreads();

    for (int i = threadIdx.x; i < N+1; i++) {
        my_other_result[i] = something with var_sh[i]
    }
}

my_kernel<<32,32>>(my_other_result);
```

- 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 needed if entries computed with other threads are used

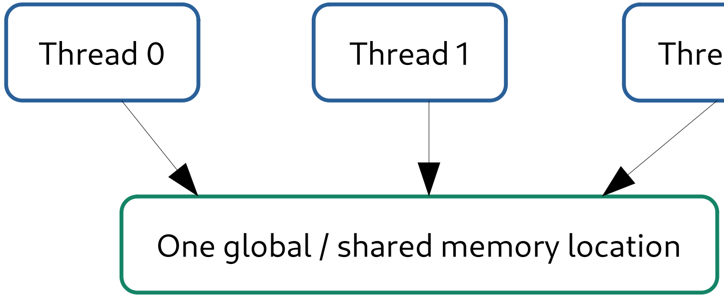
Dynamic shared memory

```
__global__ void my_kernel(float *my_other_result) {
     extern __shared__ float var_sh[];

     for (int i = threadIdx.x; i < N+1; i++) {
          var_sh[i] = …;
     }

     __syncthreads();

     for (int i = threadIdx.x; i < N+1; i++) {
          my_other_result[i] = something with var_sh[i]
     }
}


my_kernel<<32,32, (N+1)*sizeof(float)>>(my_other_result);
```

- 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

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



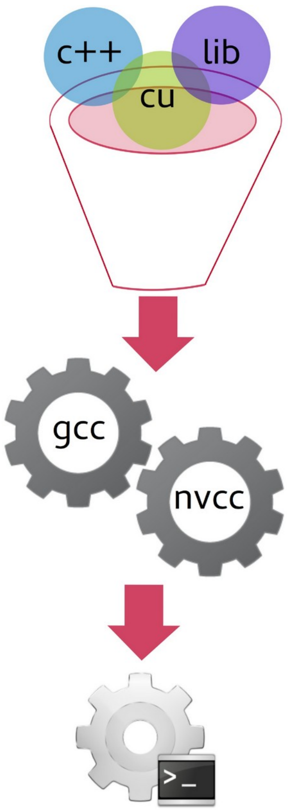
Index calculation

- It is often useful to parallelize the processing of one array with both blocks and threads
- Unique index = x + y * size
- `int index = threadIdx.x + blockIdx.x * 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 ojbect… files as input
- Can link libraries, include header files
- Can integrate into larger projects with CMake



Resources

- D. B. Kirk, W. w. Hwu: “Programming Massively Parallel Processors”
- J. Sanders, E. Kandrot: “CUDA by Example”
- N. Wilt: “The CUDA Handbook”
- http://docs.nvidia.com/cuda/cuda-c-programming-guide/
- http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

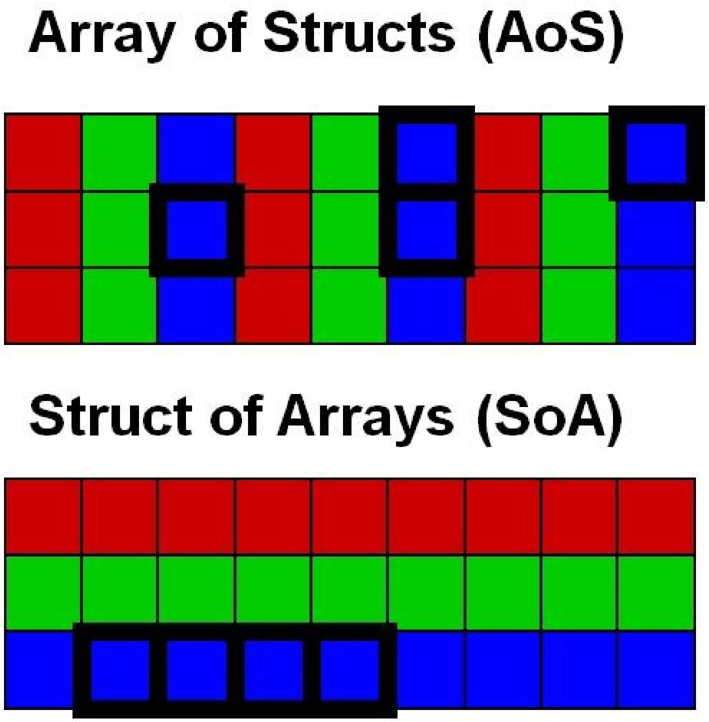
Summary

- 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
- 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



Control flow

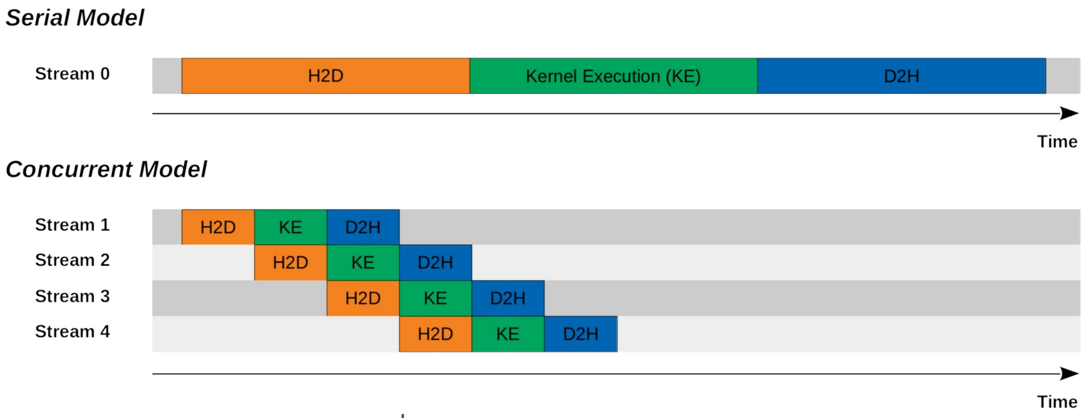
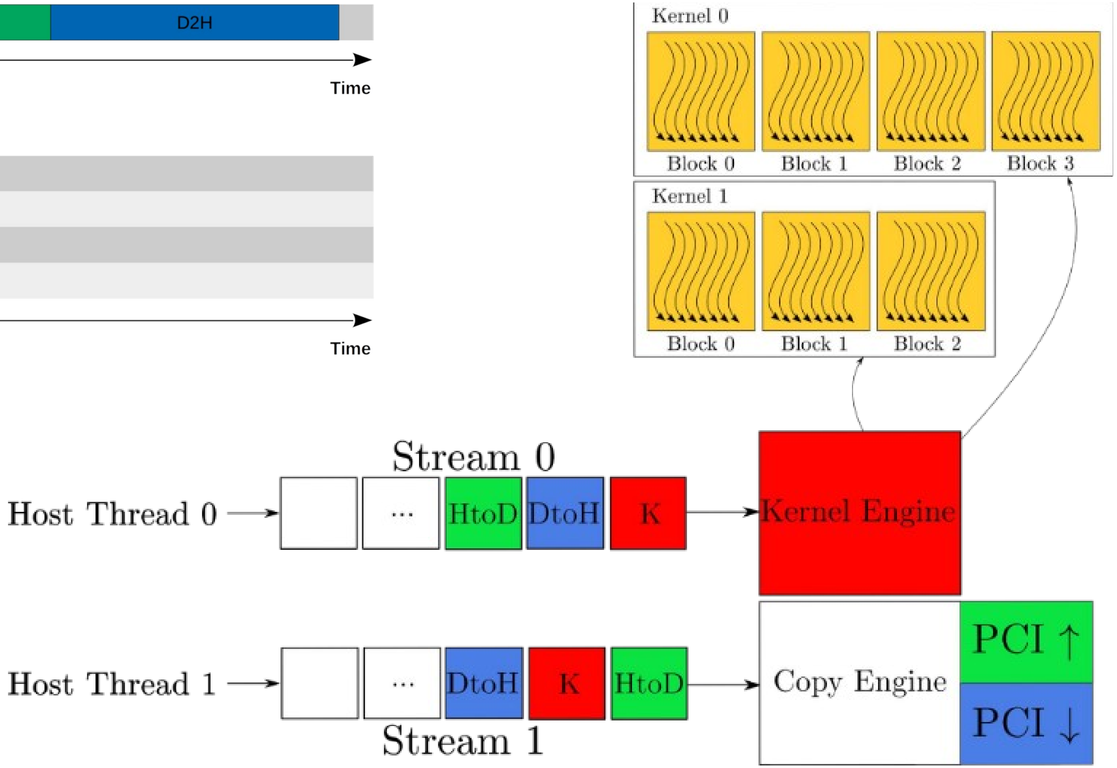


Image source



- 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

```
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