Introduction to CUDA

Dorothea vom Bruch

July 2019

1st Real Time Analysis Workshop

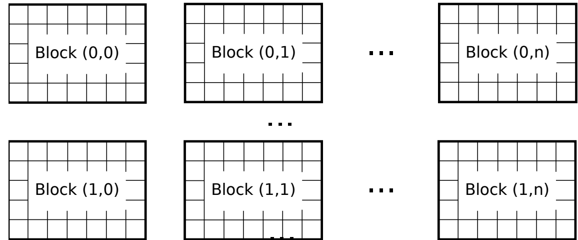
Institut Pascal, Université Paris-Saclay, France

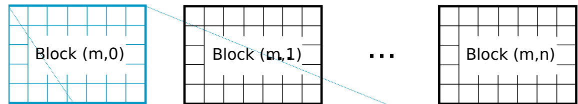
CUDA

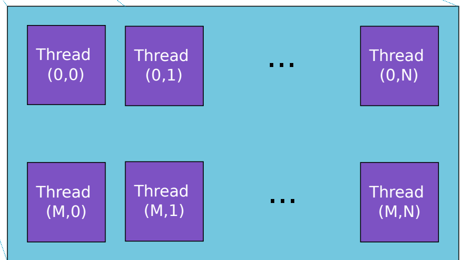
- Nvidia's API extension to C: Compute Unified Device Architecture (CUDA)
- Compiles with nvcc and gcc

  → Runs on host and device
- Very similar to C / C++ code

Thread organization

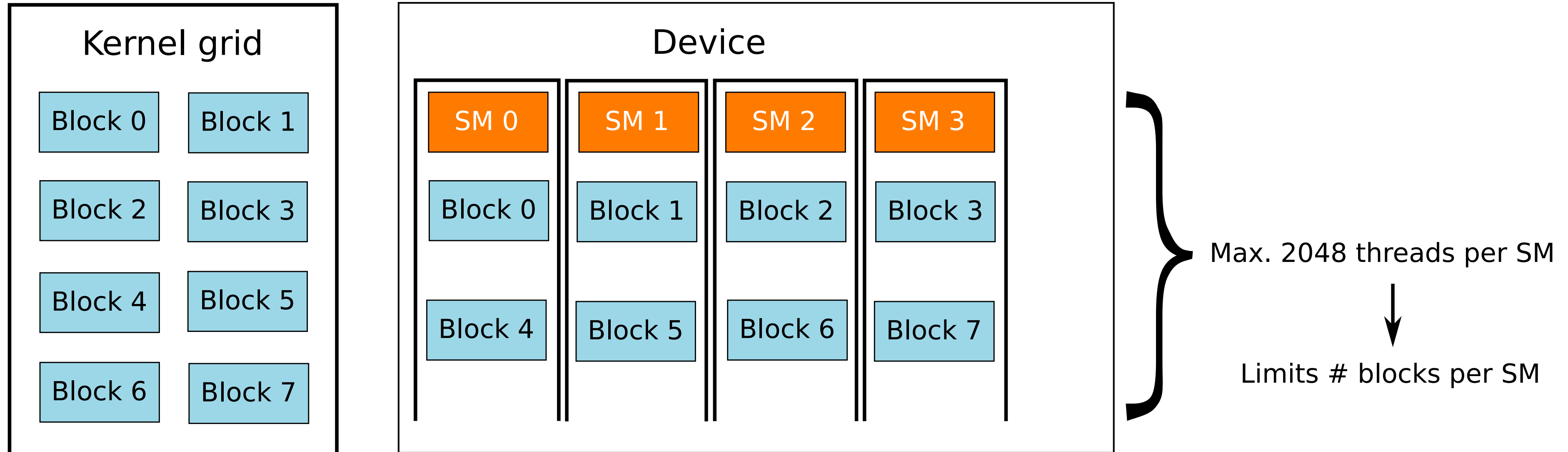






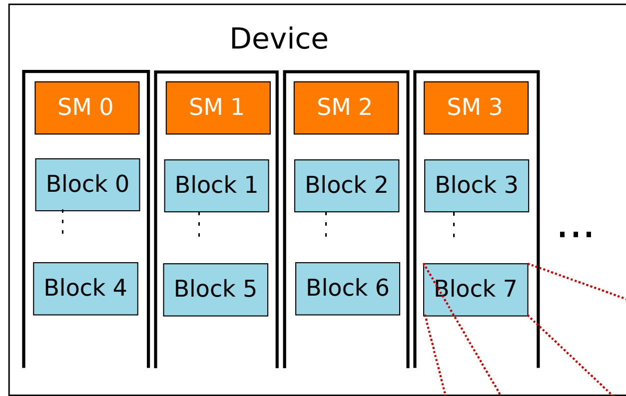
- Each thread processes the same kernel, but on different data
- Up to three dimensions for blocks and threads
- Example: Pascal architecture:
- Max. 1024 threads / block
- Max. grid dimensions:
- 2147483647 x 65536 x 65536 blocks

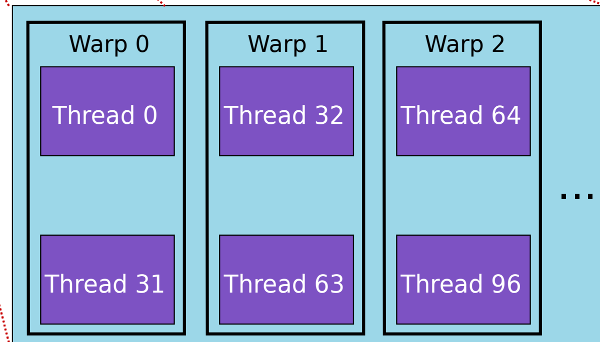
Hardware implementation



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

Hardware implementation: warps



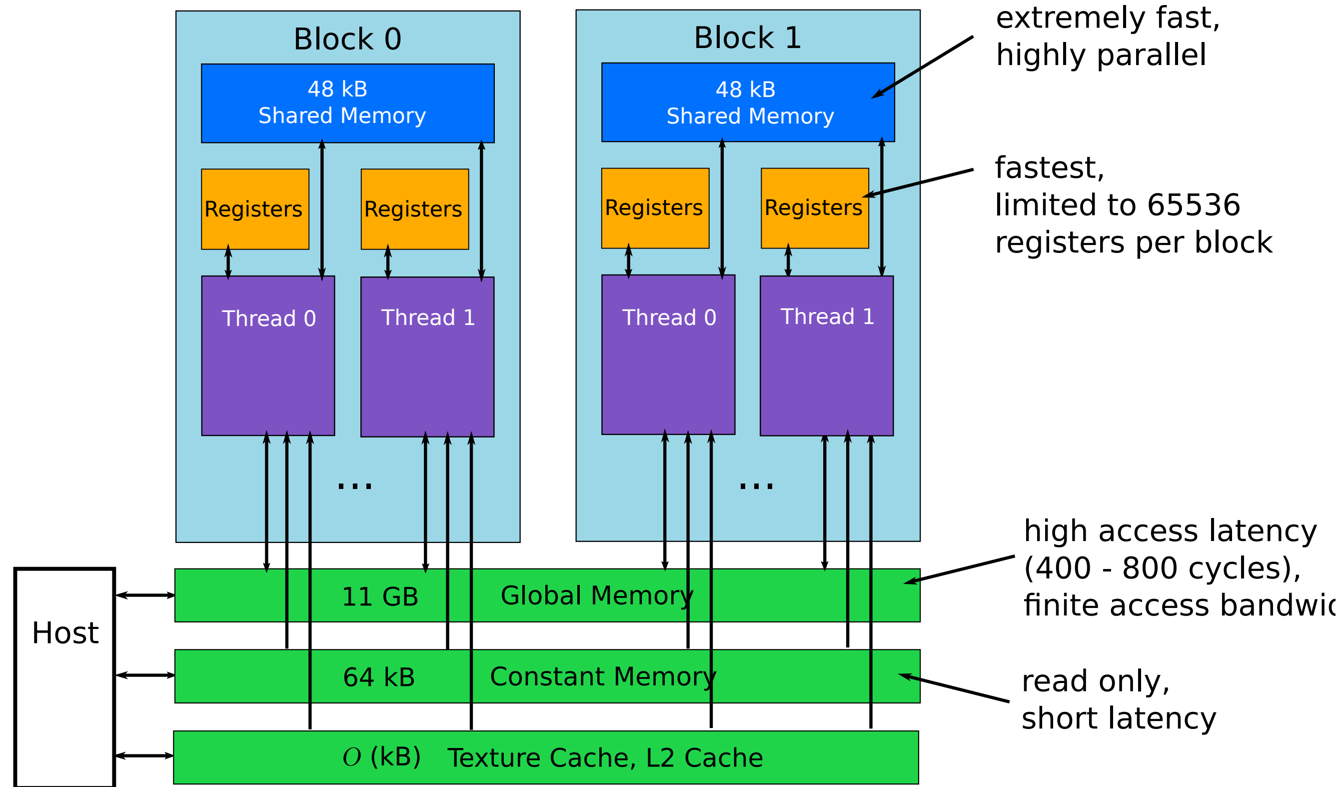


- After block has been assigned to one SM

  → Division into units called warps

  - One warp = 32 threads

Memory layout



Grid 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

First device function

```
__global__ void hello_world( ) {

}
int main( ) {
dim3 n_blocks(1);
dim3 n_threads(1);
hello_wold<<<n_blocks, n_threads>>>(...);
return 0;
}
```

Indicates that function runs on device, is called from host; compiled with nvcc

Standard C / C++ function; compiled with gcc

Structure designed to store size of grid and block

<<< >>>: options for grid launch: # of blocks in grid, # of threads / block
(...): arguments can be passed to kernel function

Thread and block index

```
__global__ void hello_world( ) {

if ( blockIdx.x < 100 && thradIdx.x < 100 )
    printf("Hello World from block %d and thread %d \n",
 blockIdx.x, threadIdx.x);
}
int main( ) {
dim3 n_blocks(1);
dim3 n_threads(1);
hello_world<<<n_blocks, n_threads>>>();
cudaDeviceSynchronize();
return 0;
}
```

Built-in variable to access index of current block

Built-in variable to access index of current thread within block

Printf works within kernels, but should be used with caution:

- Will get output from every thread
- Stored in buffer, up to 1 MB
- Buffer is only copied in certain cases, e.g. device synchronization

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<<<1,1>>>( a_dev, b_dev );
cudaMemcpy( &b_host, b_dev, sizeof(int), cudaMemcpyDeviceToHost );
cudaDeviceSynchronize();
cudaFree( a_dev);
cudaFree( b_dev);
```

Pointer to allocated memory on device is returned

Size of memory to be allocated

Global memory management

Pointer to destination

Pointer to source

Size of memory to be copied (bytes)

Copy direction

```
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);
```

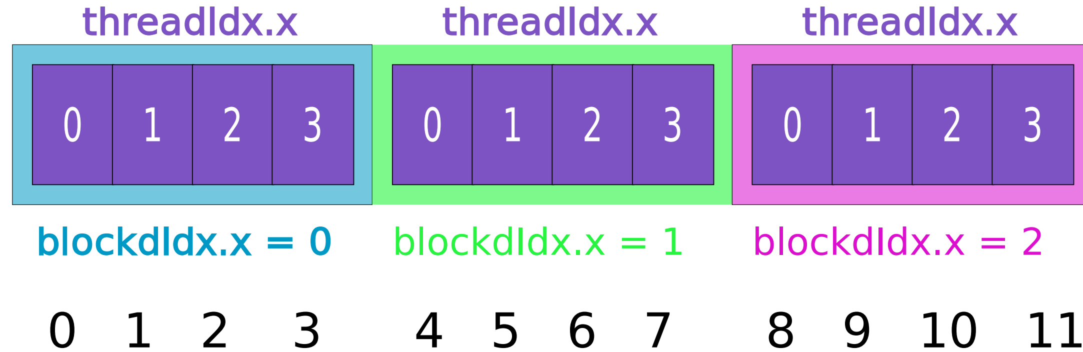
Memory declaration

	Where?	Access?
`__constant__ int a_const;`	Constant memory	Host and device
`__shared__ int a_shared;`	Shared memory	Device, within block

- `__syncthreads():`
- synchronize all threads within ONE BLOCK
- Useful when they all need access to shared memory after filling it in parallel

```
a_shared[threadIdx.x] = a_global[threadIdx.x];
__syncthreads();
int b = 10 * a_shared[threadIdx.x];
```

Index calculation



- Unique index = x + y * size
- `int index = threadIdx.x + blockIdx.x * blockDim.x;`
- `blockDim.x`: number of threads in a block (in x direction), accessible from the kernel
- `gridDim.x`: number of blocks in the grid (in x direction), accessible from the kernel

Compilation

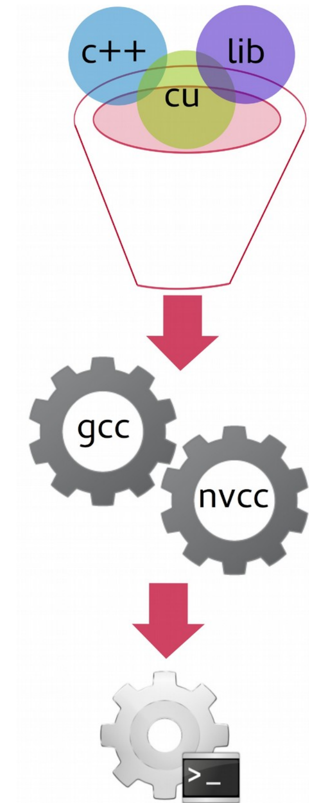
- Use nvcc for compilation:
  - Calls nvcc for CUDA parts
  - Calls gcc for c++ parts

```
nvcc FirstProgram.cu -o executable
```

- Execute:

```
./executable
```

- 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

- 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/

Exercises

- Hello world
- Vector addition
- Matrix multiplication

- To be found here: https://gitlab.cern.ch/dovombru/cuda-introduction-exercises