Programming Parallel Computers

Jukka Suomela · Aalto University · ppc.cs.aalto.fi

Part 5B: How does the GPU execute code?

What happens inside the GPU?

- ***The same general principles hold for a wide range of different GPUs***
- However, when we need some concrete numbers to illustrate these ideas, we will use the following GPU:
  - NVIDIA Quadro K2200
  - "**Maxwell**" microarchitecture
  - **5 × streaming multiprocessors** (SM)

Key concepts that we need

- Kernel ≈ some instructions that we want to execute
- Blocks that consist of warps
- Warps that consist of 32 threads
- Shared memory
- ***Registers***

GPU registers

Lots of data, needs to be stored somewhere!

- ***At most 255 registers*** per thread
  - scalar registers, can hold 32-bit numbers
- When your kernel is compiled,
  the compiler will decide how many registers are used
  - for each kernel, the compiler stores some metadata, e.g.:

> “To run this kernel,
> we will need 96 registers per thread,
> and 2 KB of shared memory per block”

Chapter 4, part V2

```
__global__ void mykernel(...) {
    ...
    float v[8][8];
    ...
    for (int k = 0; k < n; ++k) {
        float x[8];
        float y[8];
        ...
        x[ib] = ···;
        y[jb] = ···;
        ...
        v[ib][jb] = min(v[ib][jb], x[ib] + y[jb]);
    }
    ...
}
```

64 registers?

8 + 8 registers?

cuobjdump --dump-sass

```
...
FADD R79, R86.reuse, R79 ;
FADD R85, R86.reuse, R85 ;
FADD R89, R86, R89 ;
FMNMX R69, R88, R69, PT ;
FMNMX R67, R90, R67, PT ;
FMNMX R56, R75, R56, PT ;
FMNMX R53, R95, R53, PT ;
FMNMX R34, R87, R34, PT ;
FMNMX R26, R83, R26, PT ;
...
```

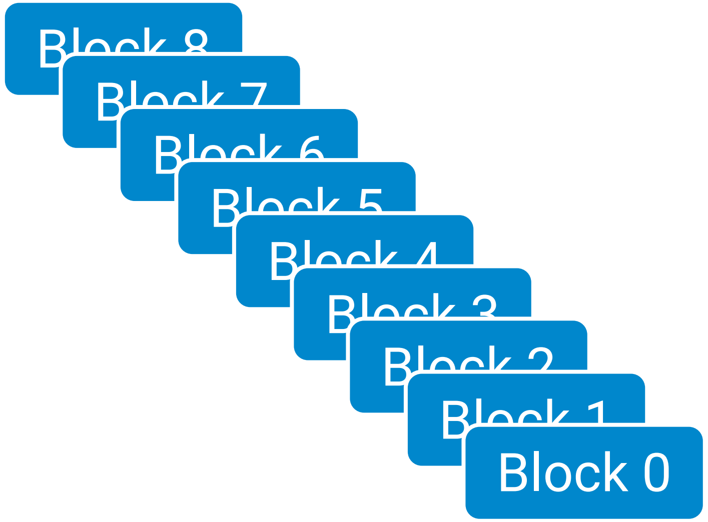
Uses 96 registers (R0 ... R95)

Key choices

- **Fixed:** 32 ***threads per warp***
- **We choose:** how many ***threads per block***
  - at most 1024
- **We choose:** how much ***shared memory per block***
  - at most 48 KB
- **Compiler chooses:** how many ***registers per thread***
  - depends on our kernel code
  - at most 255

What happens when we launch a kernel?

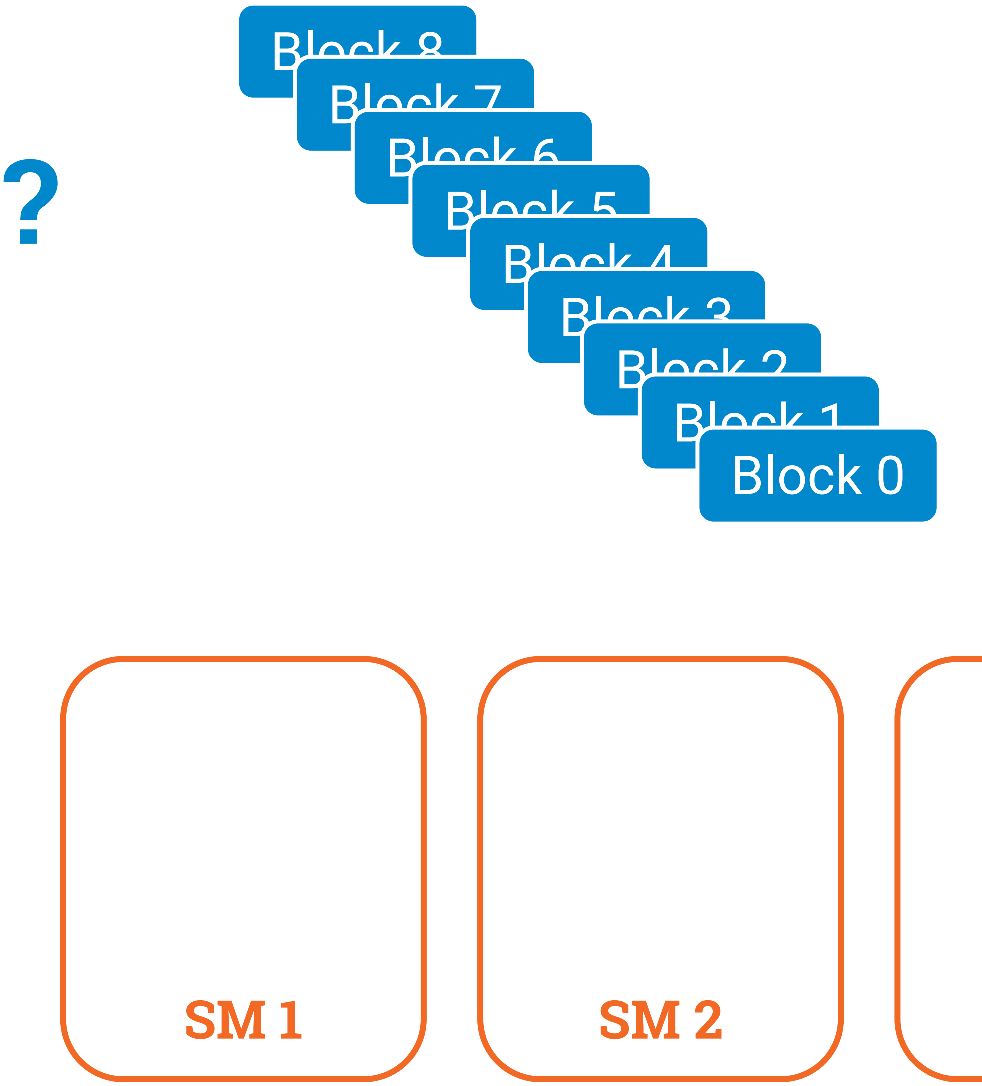
- All **blocks** are put in a GPU-wide queue
  - cheap, no resources allocated yet



Block 8
Block 7
Block 6
Block 5
Block 4
Block 3
Block 2
Block 1
Block 0

What happens when we launch a kernel?

- All **blocks** are put in a GPU-wide queue
    - cheap, no resources allocated yet
- ***5 "streaming multiprocessors" (SM)***



Block 8

Block 7

Block 6

Block 5

Block 4

Block 3

Block 2

Block 1

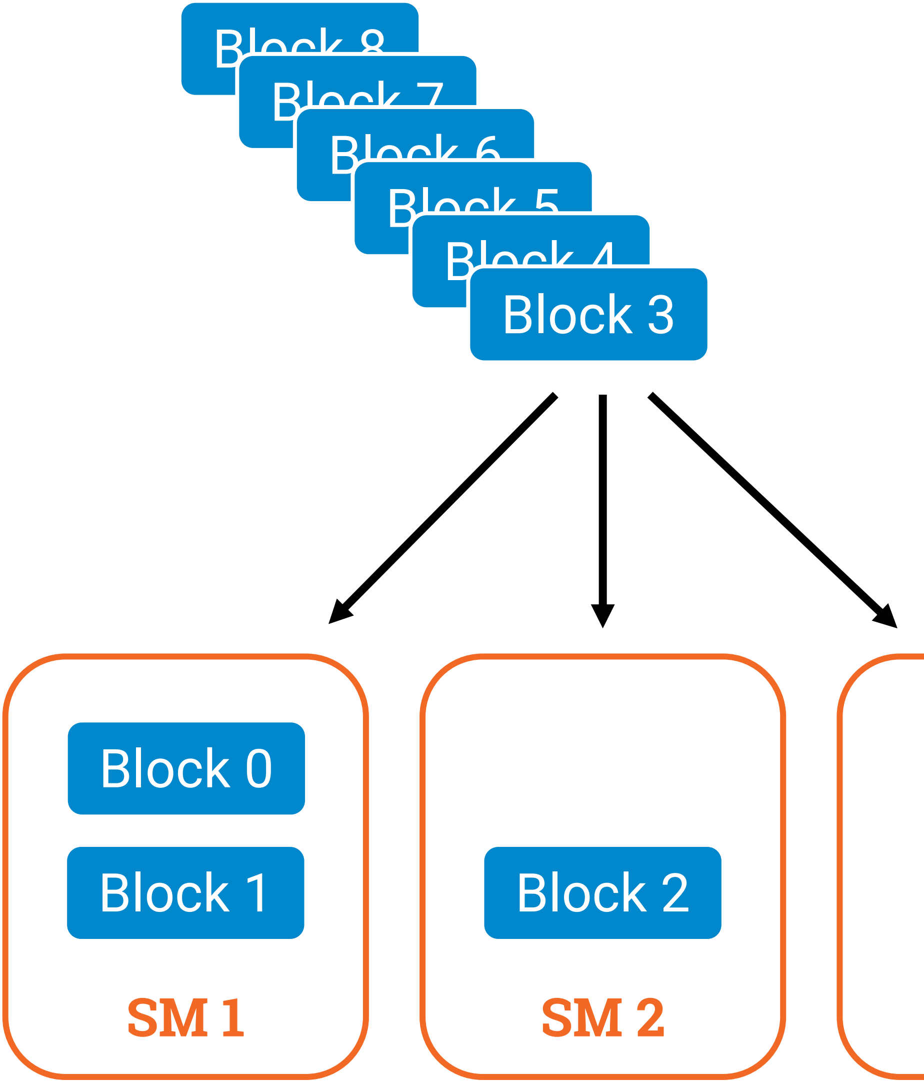
Block 0

SM 1

SM 2

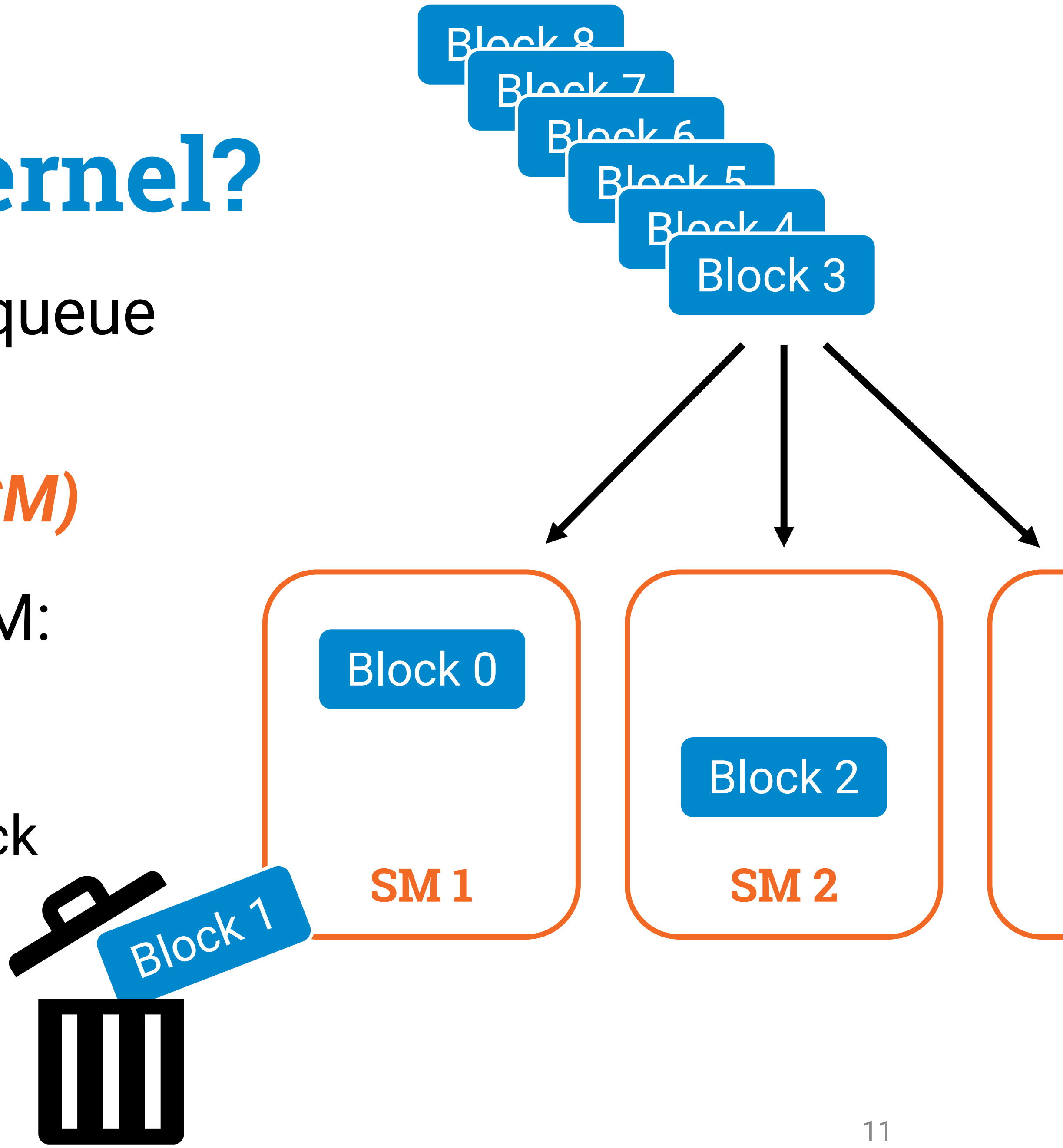
What happens when we launch a kernel?

- All **blocks** are put in a GPU-wide queue
  - cheap, no resources allocated yet
- ***5 “streaming multiprocessors” (SM)***
- Whenever there is room in one SM:
  - SM takes a block from the queue
  - the block becomes active
  - resources are allocated for the block



What happens when we launch a kernel?

- All **blocks** are put in a GPU-wide queue
  - cheap, no resources allocated yet
- ***5 “streaming multiprocessors” (SM)***
- Whenever there is room in one SM:
  - SM takes a block from the queue
  - the block becomes active
  - resources are allocated for the block
  - the block is there until all threads in the block finish running, then resources are freed



What happens when SM starts to process a block?

- Block becomes active
  - room for **32 active blocks** per SM
- All warps of the block become active
  - room for **64 active warps** per SM
- Shared memory allocated for the block
  - **64 KB shared memory** available per SM
- Physical registers allocated for each thread
  - **65536 physical 32-bit registers** per SM

Blocks will have to wait in the queue until all these resources are available!

What happens when SM starts to process a block?

- Block becomes active
  - room for **32 active blocks** per SM
- All warps of the block become active
  - room for **64 active warps** per SM
- Shared memory allocated for the block
  - **64 KB shared memory** available per SM
- Physical registers allocated for each thread
  - **65536 physical 32-bit registers** per SM

64 active warps
× 32 threads/warp
× 5 SMs
= 10240 active threads

What happens when SM starts to process a block?

- Block becomes active
  - room for **32 active blocks** per SM
- All warps of the block become active
  - room for **64 active warps** per SM
- Shared memory allocated for the block
  - **64 KB shared memory** available per SM
- Physical registers allocated for each thread
  - **65536 physical 32-bit registers** per SM

Limits parallelism if blocks too small

What happens when SM starts to process a block?

- Block becomes active
  - room for **32 active blocks** per SM
- All warps of the block become active
  - room for **64 active warps** per SM
- Shared memory allocated for the block
  - **64 KB shared memory** available per SM
- Physical registers allocated for each thread
  - **65536 physical 32-bit registers** per SM

64 warps
× 32 threads/warp
× 32 registers/thread
= 65536 registers

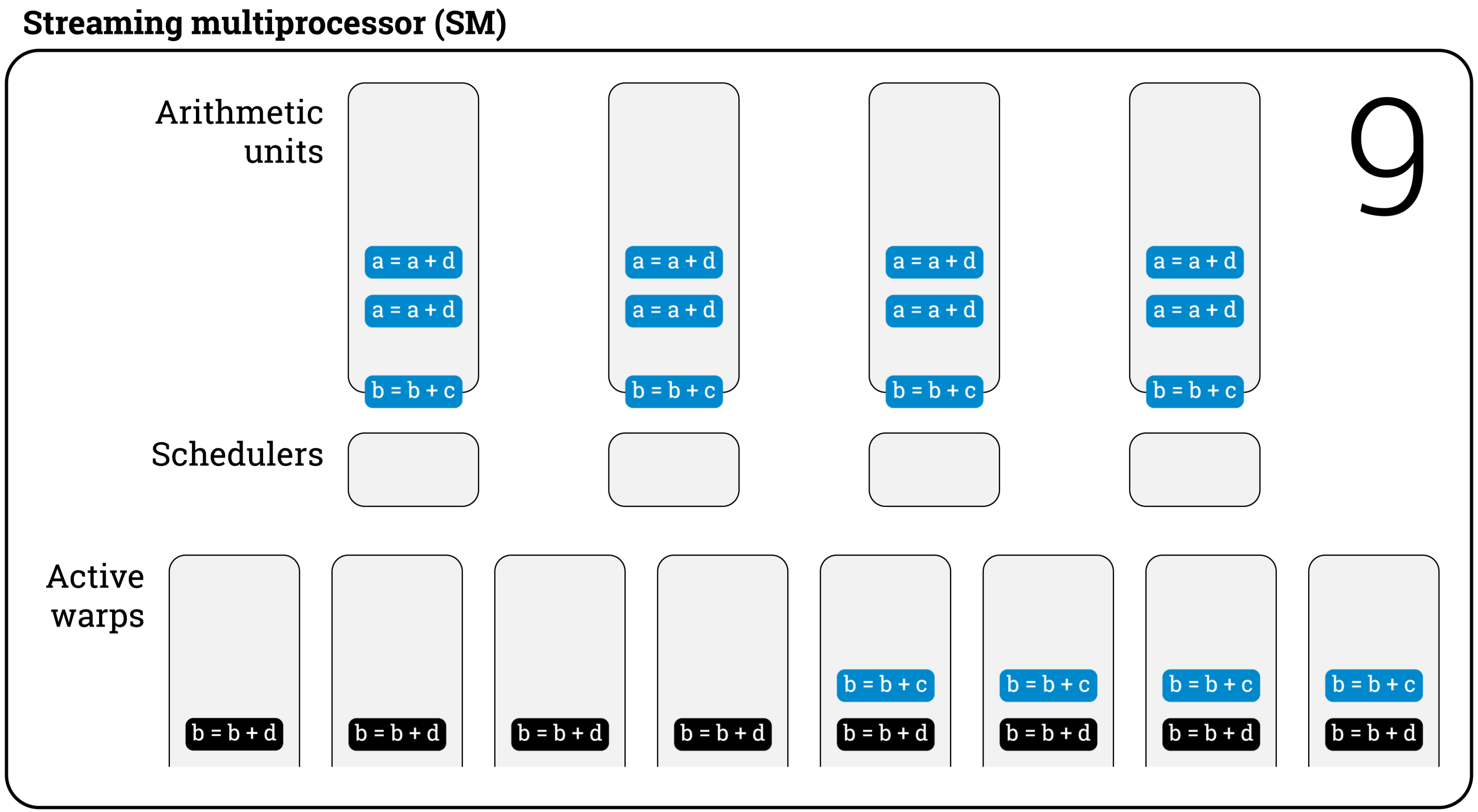
What happens when SM starts to process a block?

- Block becomes active
  - room for **32 active blocks** per SM
- All warps of the block become active
  - room for **64 active warps** per SM
- Shared memory allocated for the block
  - **64 KB shared memory** available per SM
- Physical registers allocated for each thread
  - **65536 physical 32-bit registers** per SM

32 active blocks: 2 KB shared memory per block

How does SM execute code from active warps?

See the videos for an animation…



Keeping arithmetic units busy (in theory)

- **Lots of independent instructions:**
  - e.g. floating-point additions: throughput **4 warps per clock cycle**
  - **4 active warps** per SM enough to keep all arithmetic units busy
  - in each clock cycle there is something to do in each warp
- **All instructions depend on previous instruction:**
  - e.g. floating-point addition: latency **6 clock cycles**
  - **$6 \cdot 4 = 24$ active warps** per SM enough to keep arithmetic units busy
  - in each clock cycle there is a warp that is ready

Keeping arithmetic units busy (in theory)

- **Lots of independent instructions:**
  - e.g. floating-point additions: throughp **rps per clock cycle**
  - **4 active warps** per SM enough metic units busy
  - in each clock cycle th each warp
- **All instruc ous instruction:**
  - e.g. float on: latency **6 clock cycles**
  - **6 · 4 = 24 warps** per SM enough to keep arithmetic units busy
  - in each clock cycle there is a warp that is ready

The real hardware is a bit more complicated…

Keeping arithmetic units busy (in practice)

- **Lots of independent "+" instructions:**
  - **4 active warps** per SM enough to keep arithmetic units ≥ **82%** busy
  - **8 active warps** per SM enough to keep arithmetic units ≥ **96%** busy
- **Pairs of independent "+" instructions:**
  - **12 active warps** per SM enough to keep arithmetic units ≥ **87%** busy
  - **16 active warps** per SM enough to keep arithmetic units ≥ **97%** busy
- **All "+" instructions depend on previous instruction:**
  - **16 active warps** per SM enough to keep arithmetic units ≥ **65%** busy
  - ***additional warps do not help to get beyond 65%***