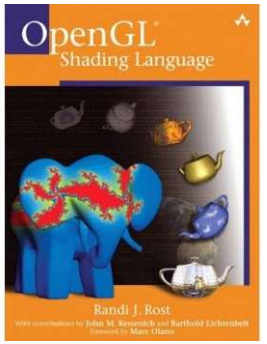
King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 4: GPU Architecture, Pt. 2

Markus Hadwiger, KAUST

Reading Assignment #2 (until Sep 14)



Read (required):

- Orange book (GLSL), Chapter 4
  (*The OpenGL Programmable Pipeline*)
- Nice brief overviews of GLSL and legacy assembly shading language
  `https://en.wikipedia.org/wiki/OpenGL_Shading_Language`
  `https://en.wikipedia.org/wiki/ARB_assembly_language`
- Read:
  `https://en.wikipedia.org/wiki/Instruction_pipelining`
  `https://en.wikipedia.org/wiki/Classic_RISC_pipeline`
- Get an overview of NVIDIA Hopper and Blackwell GPU architectures:
  `https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c`
  `https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/`
  `nvidia-rtx-blackwell-gpu-architecture.pdf`

Read (optional):

- GPU Gems 2 book, Chapter 30
  (*The GeForce 6 Series GPU Architecture*)
  `http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf`

NVIDIA CUDA

- Old acronym: “Compute Unified Device Architecture”
- Extensions to C(++) programming language
- **`__host__`**, **`__global__`**, and **`__device__`** functions
- Heavily multi-threaded
- Synchronize threads with **`__syncthreads`**(), ...
- Atomic functions
  (before compute capability 2.0 only integer, from 2.0 on also float)

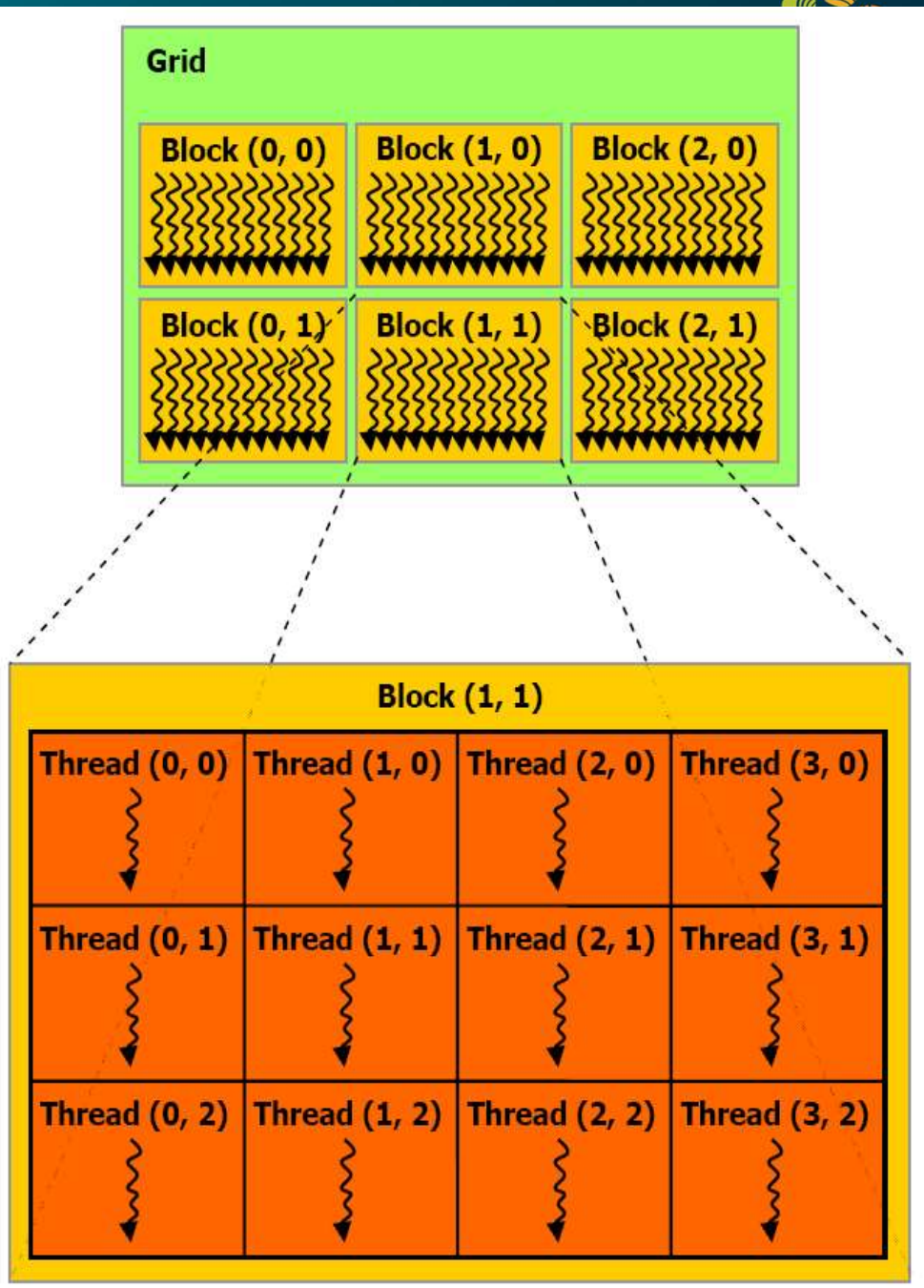
- Compile **`.cu`** files with NVCC
- Uses general C compiler (Visual C, gcc, ...)
- Link with CUDA run-time (**`cudart.lib`**) and cuda core (**`cuda.lib`**)

CUDA Multi-Threading

CUDA model groups threads into **thread blocks**; blocks into **grid**

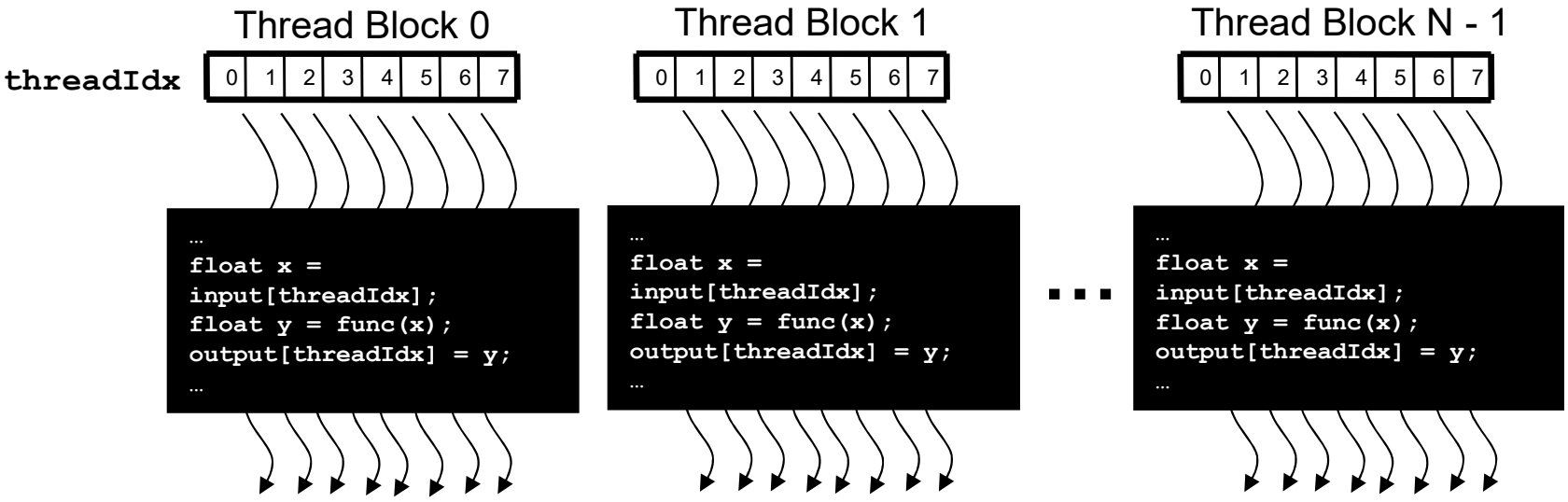
Execution on actual hardware:

- Thread blocks assigned to SM (up to 8, 16, or 32 blocks per SM; depending on compute capability)
- 32 threads grouped into a **warp** (on all compute capabilities)

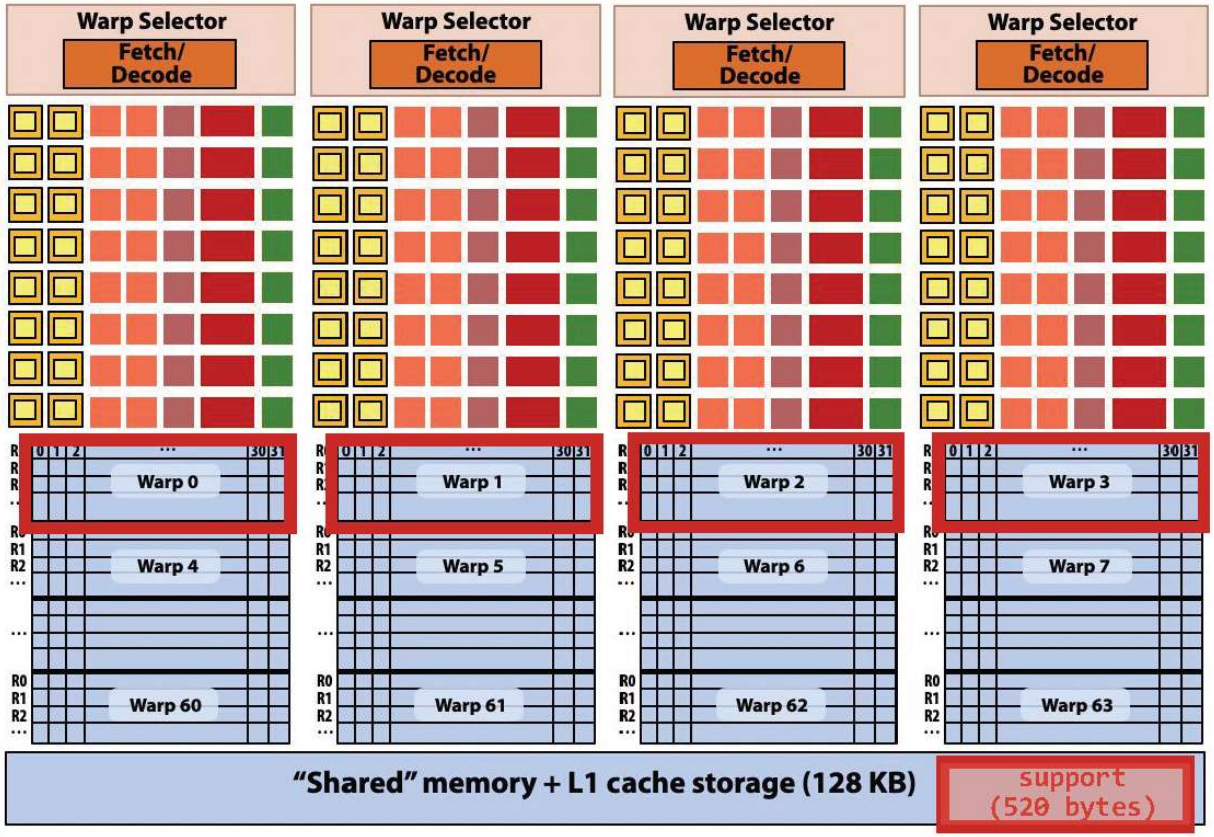


Threads in Block, Blocks in Grid

- Identify work of thread via
  - **`threadIdx`**
  - **`blockIdx`**



Teaser: Typical CUDA Kernel (SM Perspective)




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

A `convolve` thread block is executed by 4 warps
(4 warps x 32 threads/warp = 128 CUDA threads per block)

(sub-core == SM partition)

SM core operation each clock:

- **Each sub-core selects one runnable warp (from the 16 warps in its partition)**
- **Each sub-core runs next instruction for the CUDA threads in the warp (this instruction may apply to all or a subset of the CUDA threads in a warp depending on divergence)**

courtesy Kayvon Fatahalian

Launching Kernels

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(…)
```

- Execution Configuration ("<<< >>>")
  - dG - dimension and size of grid in blocks
    - Two-dimensional: x and y
    - Blocks launched in the grid: dG.x * dG.y
  - dB - dimension and size of blocks in threads:
    - Three-dimensional: x, y, and z
    - Threads per block: dB.x * dB.y * dB.z
  - Unspecified dim3 fields initialize to 1

Shared Memory Allocation

- **2 modes**
- **Static size within kernel**

```
__shared__ float vec[256];
```

- **Dynamic size when calling the kernel**

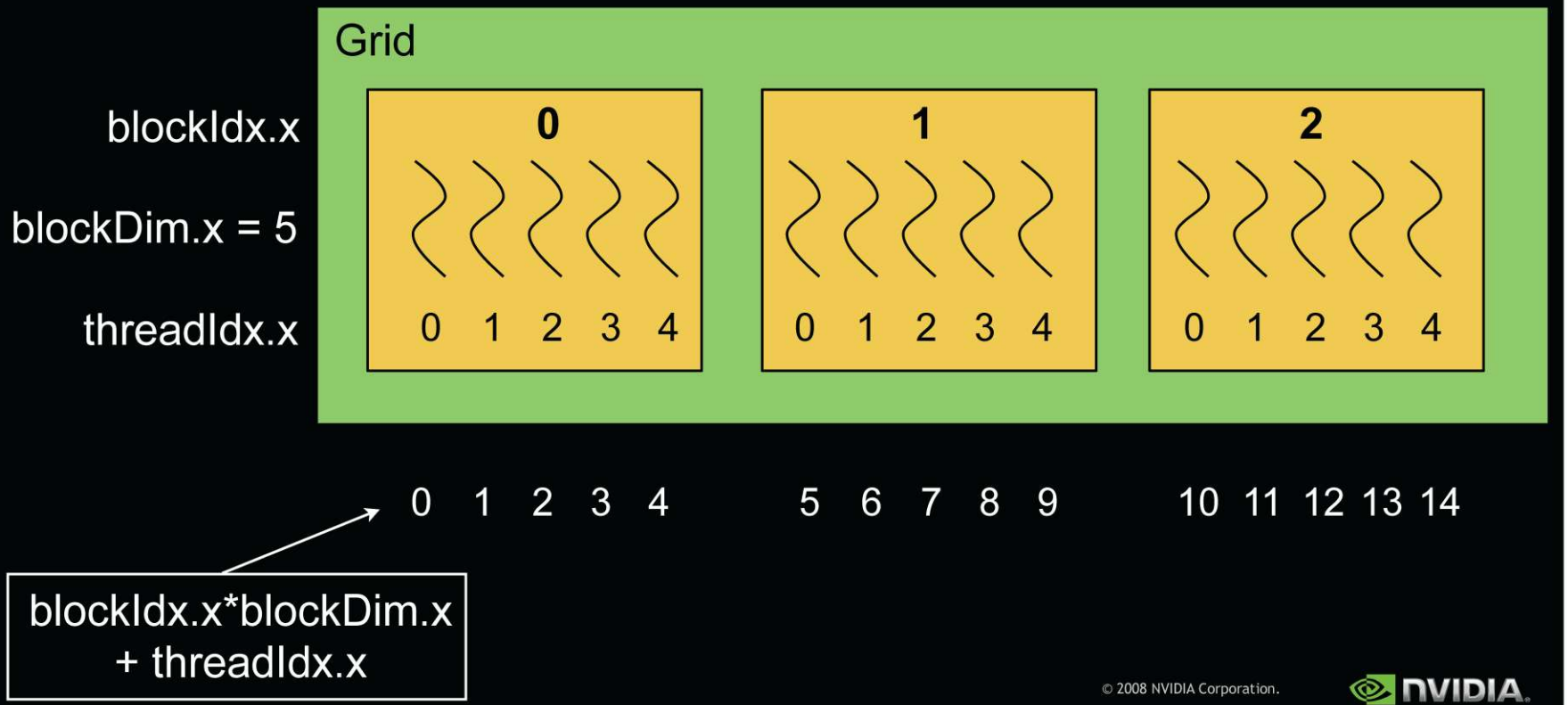
```
// in main
int VecSize = MAX_THREADS * sizeof(float4);
vecMat<<< blockGrid, threadBlock, VecSize >>>( p1, p2, …);

// declare as extern within kernel
extern __shared__ float vec[];
```

CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
  - Map from local thread ID (threadIdx) to a global ID which can be used as array indices



Thank you.