Introduction to High Performance Computing

Lecture 8

Jakub Gałecki

GPU programming using CUDA

- GPU hardware model – what's the difference?
- SIMT programming model
- Work organization – kernels, blocks, warps
- Device memory hierarchy
- Examples

Compute **U**nified **D**evice **A**rchitecture

Platform + API for GPU programming

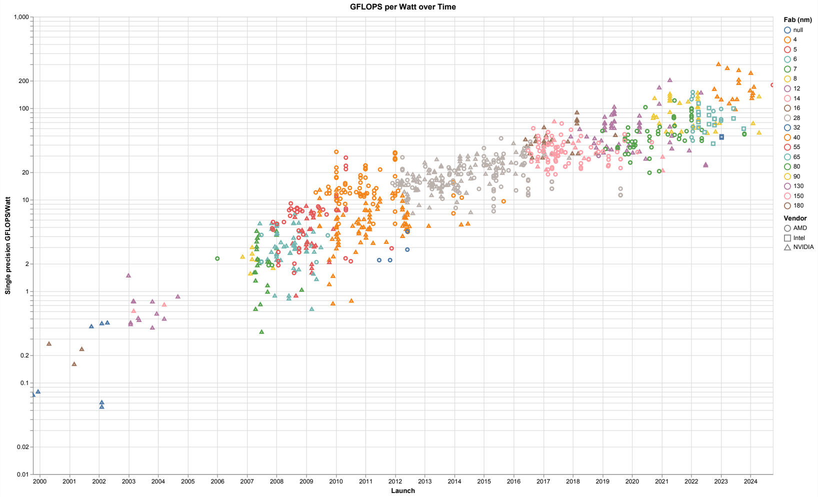
Nvidia – proprietary, closed source

Analogues: ROCm (AMD), OneAPI (Intel)

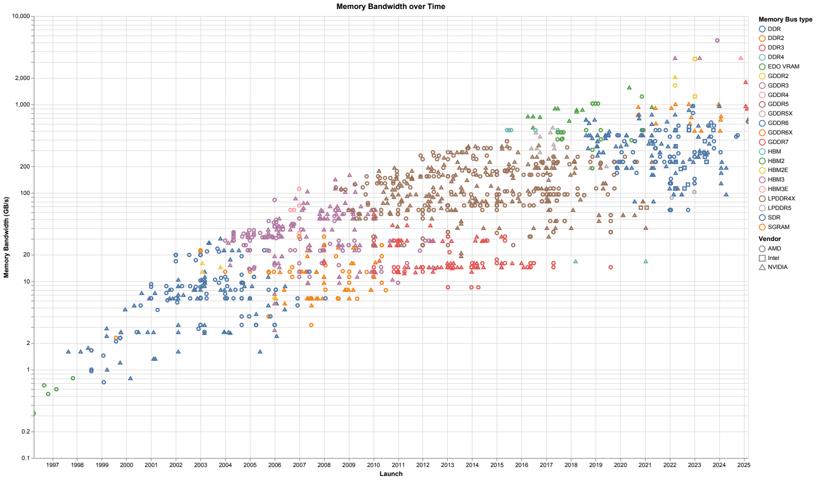
CPU vs GPU

GPUs are constrained by the same transistor/power budget – the difference lies in the approach

GPU Flops/Watt

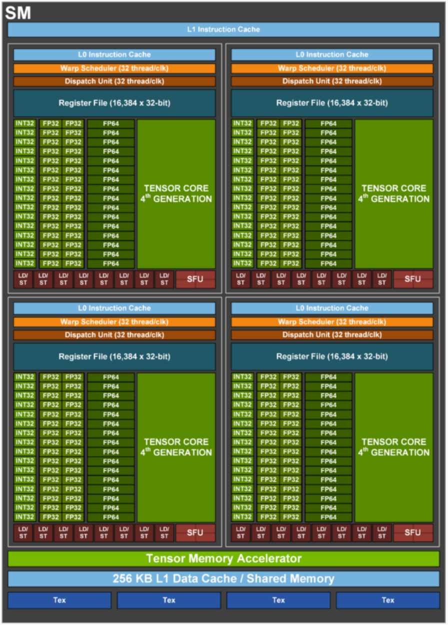


GPU memory bandwidth



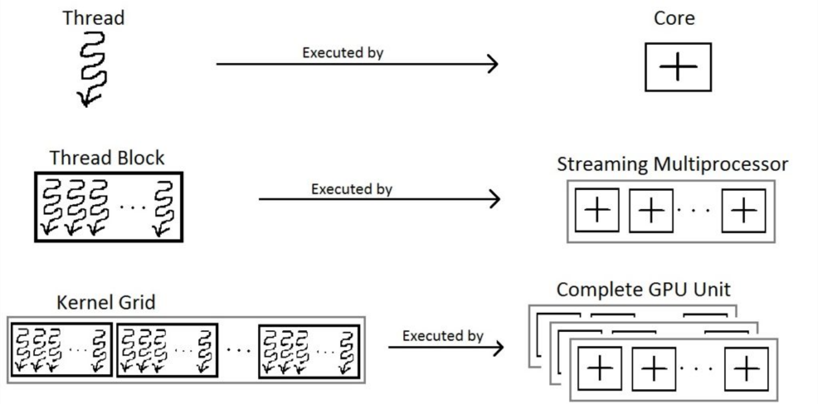
GPU die breakdown (H100)

Streaming Multiprocessor (H100)



Kernel

All threads within the kernel execute the same instructions.
The threads are organized as follows:



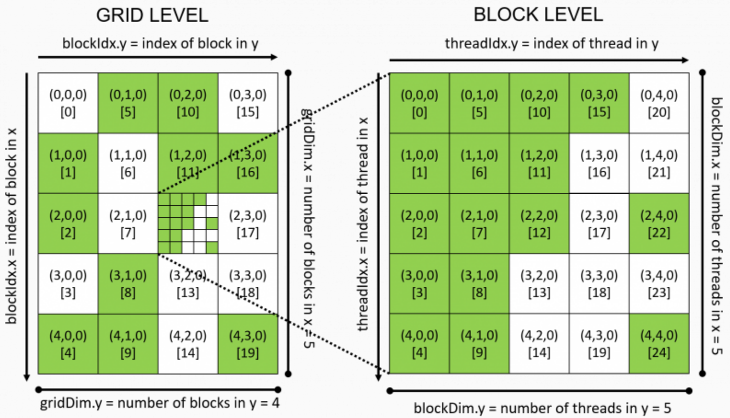
Kernel

- Submitted on host* – special syntax `kernel<<< ... >>>`
- Executed on device
- Function with no return value (`void`)
- Marked as `__global__`
- Grid info (including position) available via special variables
- Kernel arguments shared across threads
- Support for arbitrary arguments, templates

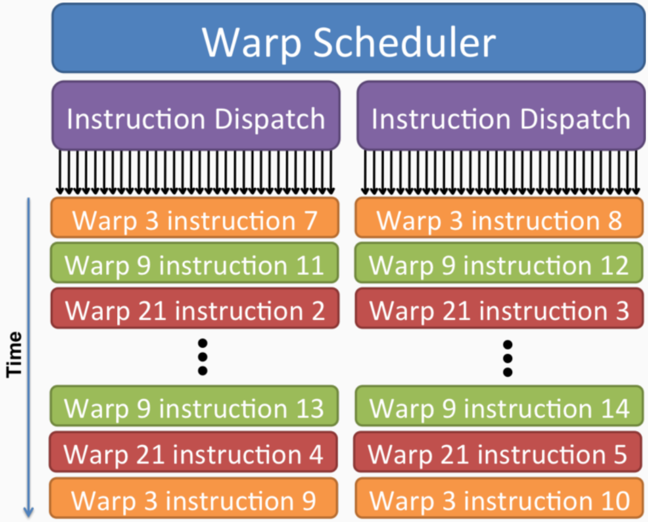
```
__global__ void myAwesomeKernel(float* data, size_t size) {
  unsigned bi = blockIdx.x, bd = blockDim.x, ti = threadIdx.x;
}

// From host code:
myAwesomeKernel<<< blocks, block_dim >>>(data, size);
```

Threads, blocks, and grids

- The grid is organized using a 1D, 2D, or 3D structure
- The type used to convey this is `dim3` (`unsigned x, y, z` fields)
- Application logic has to map this info onto work

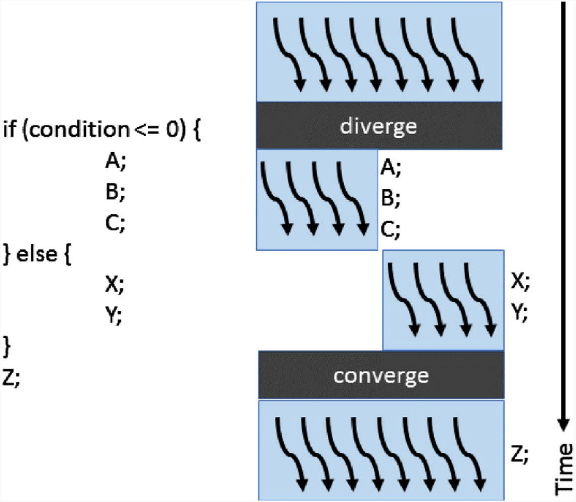


Warp

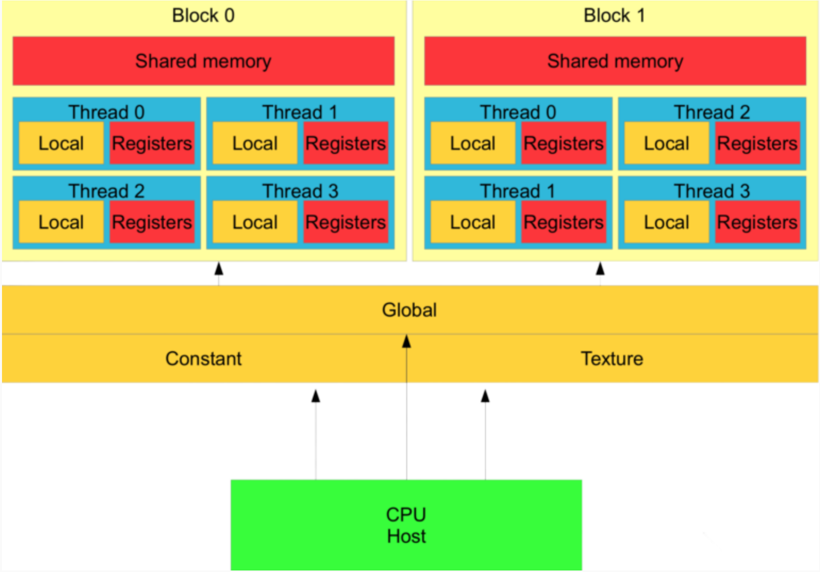


Warp Scheduler
Instruction Dispatch
Instruction Dispatch
Time
Warp 3 instruction 7
Warp 3 instruction 8
Warp 9 instruction 11
Warp 9 instruction 12
Warp 21 instruction 2
Warp 21 instruction 3
Warp 9 instruction 13
Warp 9 instruction 14
Warp 21 instruction 4
Warp 21 instruction 5
Warp 3 instruction 9
Warp 3 instruction 10

Warp divergence



GPU memory hierarchy



GPU memory access

Memory	R/W	Scope	Speed	Size	Access
Registers	RW	Thread	Fast	Small	Compiler
Global	RW	All + CPU	Slow	Large	Dyn. alloc.
Local	RW	Thread	Slow	Large	Register spill
Shared	RW	Block	Fast	Small+	Explicit
Constant	R	All	Fast bcast	Small	Kernel args*

Data is also implicitly cached in hardware caches

Statically sized:

```
__global__ void staticKernel() {
  __static__ float buffer[64];
}

staticKernel<<< blocks, block_dim >>>();
```

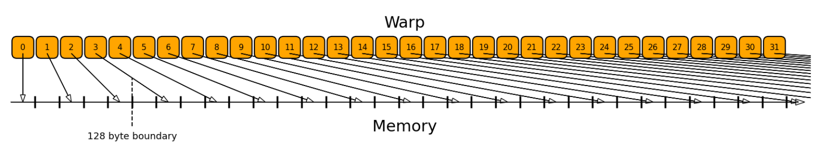
Dynamically sized:

```
__global__ void dynamicKernel() {
  extern __static__ float buffer[];
}

dynamicKernel<<< blocks, block_dim, 64*sizeof(float) >>>();
```

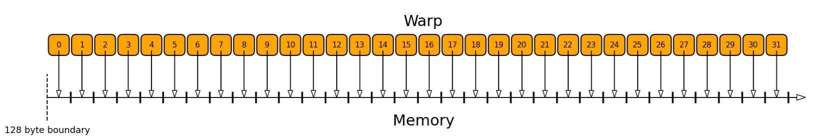
Coalesced memory access

The GPU accesses memory in naturally aligned chunks of size $\in \{32, 64, 128\}$. If all threads access the same chunk, the access is coalesced.

Strided, unaligned access:



Coalesced access:



CUDA streams

Work:

- kernels
- memcpy: host to device
- memcpy: device to host
- user defined calls*

All work is placed into streams – asynchronous sequences of operations

Within a stream, the next item can only be started once the previous one finishes

Work between streams is not synchronized

Streams must be waited upon (from the host)

Thread synchronization & communication

Synchronization:

- Across kernels: streams
- Within kernel: impossible!
- Within block: `__syncthreads()`
- Within warp*: `__syncwarp()`

Communication

- Across/within kernels: global memory
- Within block: shared memory
- Within kernel: special primitives

Managing device memory

All memory operations are initiated by the host!

Memory allocation:

```
float* d_ptr;
cudaMalloc((void**)&d_ptr, N * sizeof(float));
cudaFree(d_ptr);
```

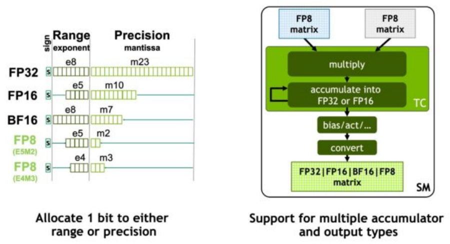
Synchronous copy:

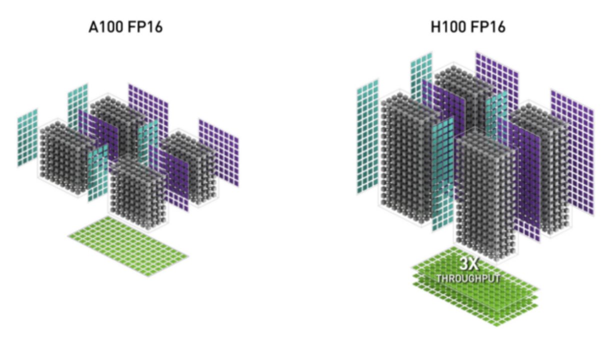
```
cudaMemcpy(d_ptr, h_ptr, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(h_ptr, d_ptr, bytes, cudaMemcpyDeviceToHost);
```

Asynchronous copy:

```
cudaMemcpyAsync(d_ptr, h_ptr, bytes, cudaMemcpyHostToDevice, stream);
cudaMemcpyAsync(h_ptr, d_ptr, bytes, cudaMemcpyDeviceToHost, stream);
```

GPUs for ML & AI





CUDA toolkit

- Compiler: `nvcc`
  - `__host__` code compiled by the (external) host compiler
  - `__device__` code compiled by `nvcc`
- `nvidia-smi` – view status of connected device(s)
- Libraries: cuBLAS, CUB, Thrust, libcu++
- Profiler: `nvprof`
- Debugger: `cuda-gdb`
- ...

Note that this is all user-space functionality, relying on (kernel-space) drivers

Example: vector sum

```
__global__ void add(double* a, double* b, double* c, unsigned N) {
    const auto i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < N) // Why is this `if` needed?
        c[i] = a[i] + b[i];
}

int main(int argc, char* argv[]) {
    const unsigned N = (1 << 14) + 1;
    std::vector<double> ah(N), bh(N), ch(N); // Fill with meaningful values...

    double *ad, *bd, *cd;
    cudaMalloc((void**)&ad, N * sizeof(double));
    cudaMalloc((void**)&bd, N * sizeof(double));
    cudaMalloc((void**)&cd, N * sizeof(double));

    cudaMemcpyAsync(ad, ah.data(), N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpyAsync(bd, bh.data(), N * sizeof(double), cudaMemcpyHostToDevice);

    const unsigned thr_per_blck = 1 << 10;
    add<<< N / thr_per_blck + 1, thr_per_blck >>>(ad, bd, cd, N);

    cudaMemcpyAsync(ch.data(), cd, N * sizeof(double), cudaMemcpyDeviceToHost);

    cudaDeviceSynchronize();

    cudaFree(ad); cudaFree(bd); cudaFree(cd);
}
```

Google colab exercises:
https://github.com/ggruszczynski/gpu_colab

Reduction case study: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Q&A

Thanks for listening!