

Advanced Topics in Numerical Analysis: High Performance Computing

Intro to GPGPU

Georg Stadler, Dhairya Malhotra
Courant Institute, NYU

Spring 2019, Monday, 5:10–7:00PM, WWH #1302

April 1, 2019

Outline

Organization issues

Computing on GPUs

Submitting jobs through a scheduler

Organization

Scheduling:

- ▶ Homework assignment #3 due tonight

Topics today:

- ▶ More GPGPU programming
- ▶ Job schedulers (SLURM)

Outline

Organization issues

Computing on GPUs

Submitting jobs through a scheduler

Review of Last Class

CPU

- ▶ fewer cores $\sim O(10)$
- ▶ smaller vector lengths (2-SSE, 4-AVX)
- ▶ large amounts of cache (L1, L2, L3)
- ▶ hide latency by extracting parallelism from sequential code: out-of-order execution, branch-prediction

GPU

- ▶ several streaming multiprocessors $\sim O(100)$
- ▶ wider vector lengths (warp size = 32)
- ▶ smaller cache (L1/programmer managed buffer, L2 on more recent GPUs)
- ▶ hide latency by executing several threads in parallel (pipelined) eg. 64-warps/SM (hyper-threading)

Review of Last Class

Computing on GPU:

- ▶ Device (GPU) is slave to the host (CPU).
- ▶ Workflow for computing on GPU:
 1. copy data from host to device,
 2. launch GPU kernel to compute result on device,
 3. copy result from device to host.
- ▶ Host - device interconnect (PCI Express or NVLink) is slow ($O(10-50)$ GB/s).
- ▶ Computing on GPUs useful only for compute-intensive tasks (when overhead of data transfer is small compared to compute time).

Review of Last Class

GPU Architecture: consists of,

- ▶ main memory (either DRAM or recently HBM with wider bus).
- ▶ PCI Express or NVLink interconnect to host (CPU) DRAM and other GPUs.
- ▶ several Streaming Multiprocessors (SM)
 - ▶ with shared L2 cache.

Each Streaming Multiprocessor,

- ▶ 32 scalar double-precision cores (each executing the same instruction, Single Instruction Multiple Data).
- ▶ can execute up to 1024 scalar threads (or 32-warps) simultaneously (pipelined, hyper-threading to hide instruction latency).
- ▶ L1 cache / shared-memory: shared by all threads in the thread-block.

Review of Last Class

GPU Threads

- ▶ parallelism $\sim O(10^5)$, orders of magnitude more than CPUs
- ▶ design philosophy on CPU vs GPU,
 - ▶ On CPU: #-of-threads \sim #-of-cores
 - ▶ On GPU: #-of-threads $\sim O(N)$ (problem size)
- ▶ thread hierarchy
 - ▶ **threads** partitioned into **blocks**
 - ▶ **blocks** arranged into a **grid**

blockIdx.x	0				1				2			
threadIdx.x	0	1	2	3	0	1	2	3	0	1	2	3
A_i	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}

`idx = threadIdx.x + blockDim.x * blockIdx.x`

```
dim3 GridDim(3), BlockDim(4);  
mykernel<<<GridDim, BlockDim>>>();
```


Review of Last Class

2D Grid:

		0			1		
		0	1	2	0	1	2
0	0	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$
	1	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$
	2	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$
1	0	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$
	1	$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$
	2	$A_{5,0}$	$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$

$A_{i,j}$

i = threadIdx.x + blockDim.x * blockIdx.x

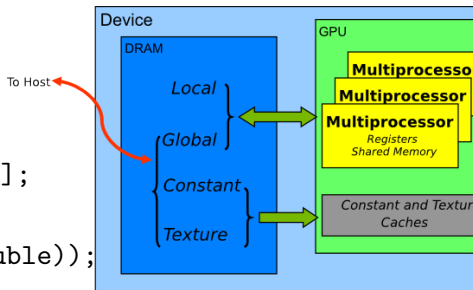
j = threadIdx.y + blockDim.y * blockIdx.y

```
dim3 GridDim(2,2), BlockDim(3,3);  
mykernel<<<GridDim, BlockDim>>>();
```

Review of Last Class

Device Memory Spaces

- ▶ Register: `double x;`
- ▶ Local: `double A[100];`
- ▶ Shared: `__shared__ double B[100];`
- ▶ Global:
`cudaMalloc(&C, 100*sizeof(double));`



Memory	on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	Thread	Thread
Local	Off	Yes	R/W	Thread	Thread
Shared	On	n/a	R/W	Block	Block
Global	Off	Yes	R/W	Global	Host
Constant	Off	Yes	R	Global	Host
Texture	Off	Yes	R	Global	Host

(table from [CUDA best practices guide](#))

Review of Last Class

Global Memory Management:

▶ Synchronous:

- ▶ Device allocation: `cudaMalloc`, `cudaFree`
- ▶ Host allocation: `malloc`, `free`
- ▶ Data transfer: `cudaMemcpy`

▶ Asynchronous:

- ▶ Device allocation: `cudaMalloc`, `cudaFree`
- ▶ Host allocation (page-locked): `cudaMallocHost`, `cudaFreeHost`
- ▶ Asynchronous data transfer: `cudaMemcpyAsync`
- ▶ Wait for data transfer: `cudaDeviceSynchronize`

▶ Driver Managed:

- ▶ Host and Device allocation: `cudaMallocManaged`, `cudaFree`
- ▶ Wait kernel function: `cudaDeviceSynchronize`

Review of Last Class

- ▶ **Best practices for accessing global memory**
 - ▶ minimize number of cache lines accessed.
- ▶ **Best practices for conditional branches**
 - ▶ avoid divergence within warps
- ▶ **Resource limits for different compute capabilities**
- ▶ **Demo:** vector addition.
 - ▶ embarrassingly parallel
 - ▶ thread j independently computes one element:
$$C[j] = A[j] + B[j]$$

Shared Memory

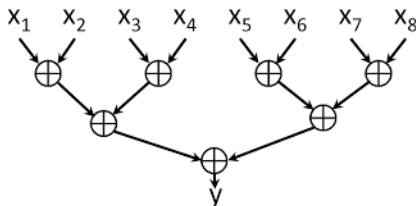
- ▶ **Fast memory** within each streaming multiprocessor
 - ▶ almost as fast as registers.
- ▶ **Configurable:** as L1 cache or programmer managed buffer.
 - ▶ `cudaFuncSetCacheConfig(kernel_fn, cacheConfig)`
- ▶ **Programmer view:** shared between all threads in a thread-block
 - ▶ in device function: `__shared__ double A[128];`
 - ▶ each thread in a block has access, can be used to communicate between threads.
- ▶ **Bank conflicts:**
 - ▶ shared memory partitioned into 32 memory banks, with consecutive (4-byte or 8-byte) data elements in different banks.
 - ▶ concurrent accesses as long as threads within a warp access different memory banks.

Synchronization

- ▶ Need synchronization between threads to avoid data races when using shared memory.
 - ▶ required between any reads and writes of the same shared array element from different threads (read-after-write or write-after-read data races).
- ▶ **__syncthreads()** synchronize all threads within a thread-block.
- ▶ **__syncwarp()** synchronize all threads within a warp (32-threads).

Reduction

Parallel reduction algorithm:



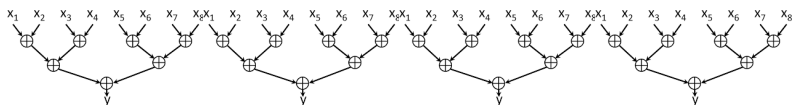
- ▶ $\log_2 N$ stages
- ▶ Requires coordination between threads!
- ▶ In each stage, every thread:
 - ▶ reads two numbers,
 - ▶ computes the sum and
 - ▶ writes its result to memory

Reduction

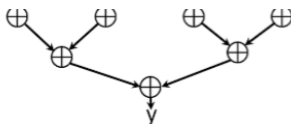
GPU algorithm

- ▶ Compute local reduction within each thread block
- ▶ Multiple kernel invocations for global synchronization

Stage 1: kernel with 4 thread-blocks, 8 threads per thread-block



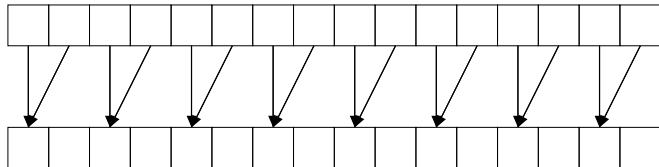
Stage 2: launch kernel with 1 thread-block with 4-threads



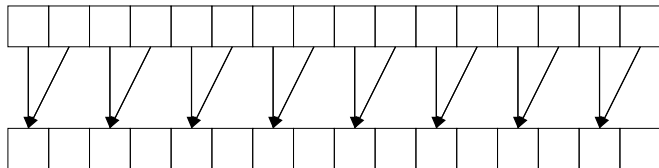
Reduction (within thread block)



Reduction (within thread block)

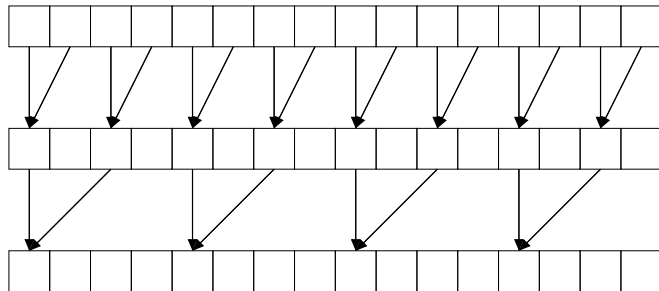


Reduction (within thread block)



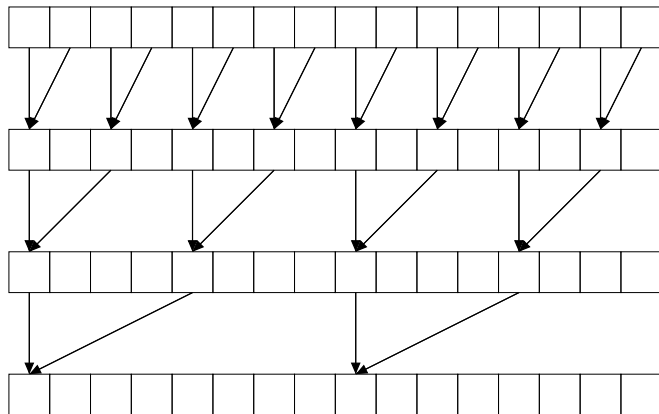
```
--syncthreads();
```

Reduction (within thread block)



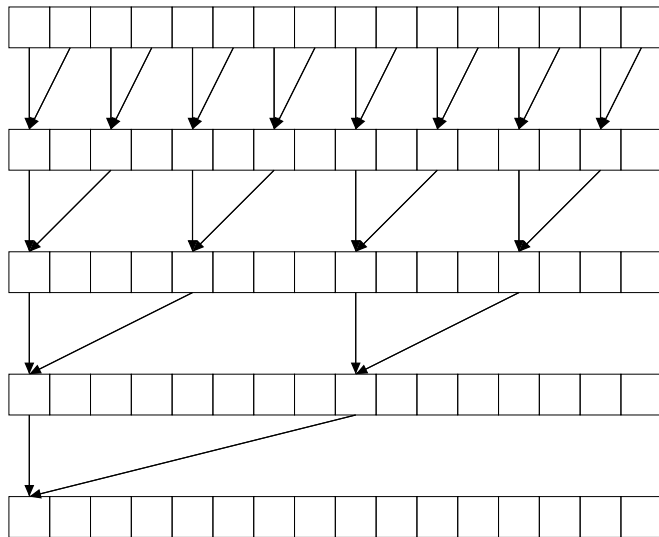
```
--syncthreads();
```

Reduction (within thread block)

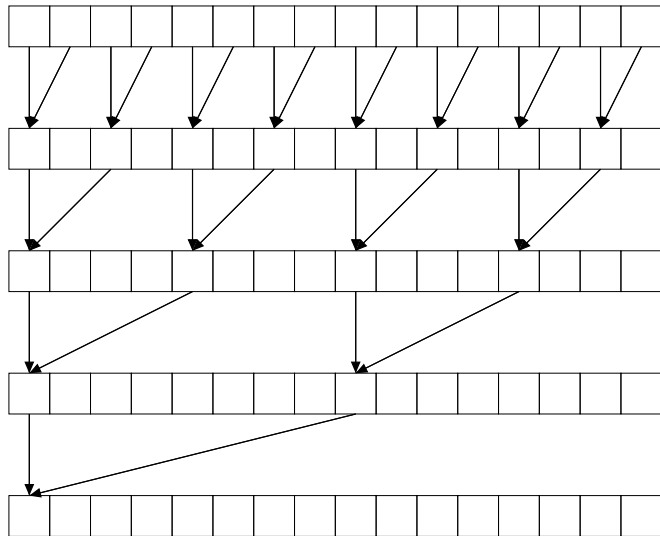


```
__syncthreads();
```

Reduction (within thread block)



Reduction (within thread block)



Problems: warp divergence, shared-memory bank conflicts

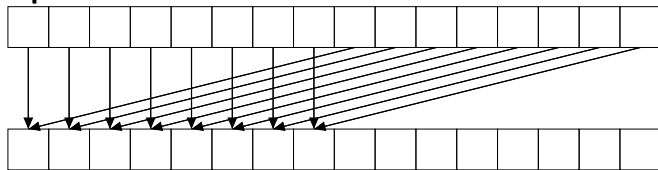
Reduction (within thread block)

Optimized scheme:



Reduction (within thread block)

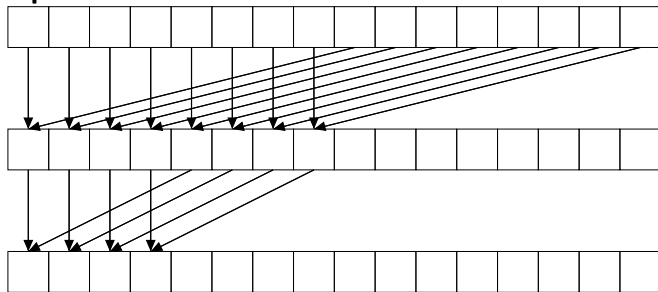
Optimized scheme:



```
__syncthreads();
```

Reduction (within thread block)

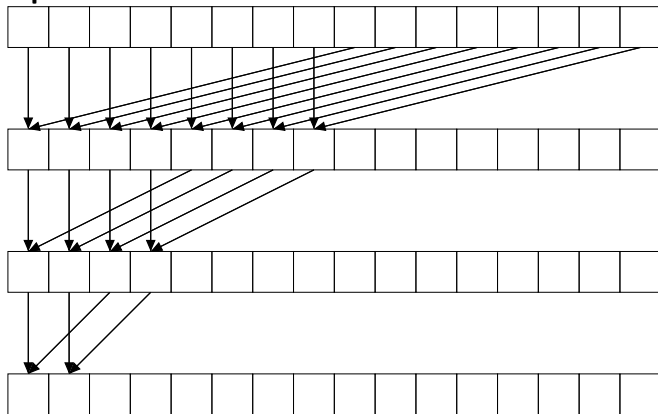
Optimized scheme:



```
--syncwarp();
```

Reduction (within thread block)

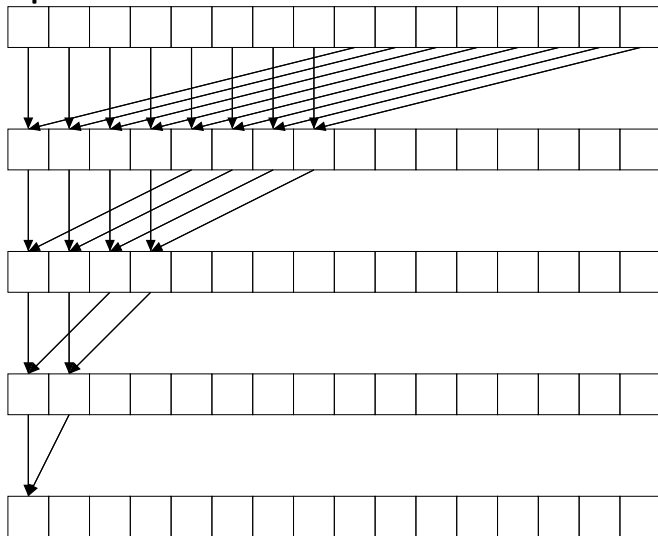
Optimized scheme:



```
__syncwarp();
```

Reduction (within thread block)

Optimized scheme:



Outline

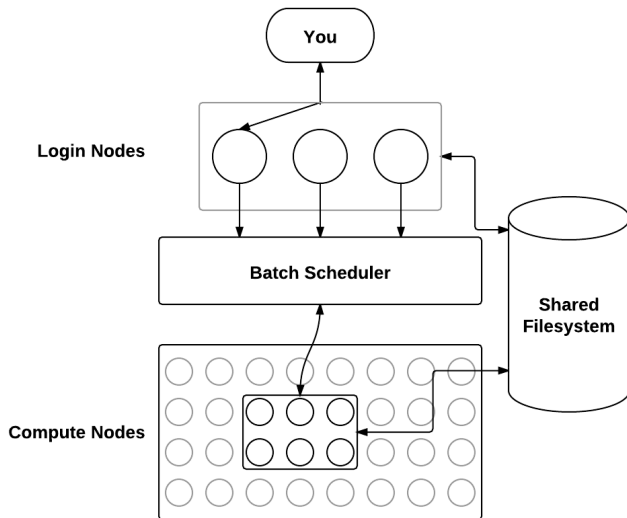
Organization issues

Computing on GPUs

Submitting jobs through a scheduler

Submitting jobs through a scheduler (e.g., on Prince)

Overview of HPC cluster



Submitting jobs on Prince

Stampede user guide: <https://wikis.nyu.edu/display/NYUHPC/Clusters+-+Prince>

Batch facilities: SGE, LSF, SLURM. Prince uses SLURM, and these are some of the basic commands:

- ▶ submit/start a job: `sbatch jobscript`
- ▶ see status of my job: `squeue -u USERNAME`
- ▶ cancel my job: `scancel JOBID`
- ▶ see all jobs on machine: `showq | less`

Submitting jobs on Prince

Some basic rules:

- ▶ Don't run on the login node!
- ▶ Don't abuse the shared file system.

Submitting jobs on Stampede

Example job script (in git repo for lecture5)

```
#!/bin/bash
#SBATCH --nodes=1                \# total number of mpi tasks
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=5:00:00
#SBATCH --mem=2GB
#SBATCH --job-name=myTest
#SBATCH --mail-type=END          \# email me when the job finishes
#SBATCH --mail-user=first.last@nyu.edu
#SBATCH --output=slurm_%j.out

module purge
module load ...
./myexecutable
```