Cornell University
Center for Advanced Computing

Introduction to CUDA Programming

Philip Nee
Cornell Center for Advanced Computing

June 2013

Based on materials developed by CAC and TACC

Outline

- Heterogeneous Parallel Computing
- Stampede and NVIDIA K20 GPU
- Programming Structure
- Thread Hierarchy
- Memory Model
- Performance Topics

Terminology

GPU	– Graphics Processing Unit
CUDA	– Compute Unified Device Architecture
Manycore	
Multicore	
SM	– Stream Multiprocessor
SIMD	– Single Instruction Multiple Data
SIMT	– Single Instruction Multiple Threads

Overview

What is CUDA?

- Compute Unified Device Architecture
  - Manycore and shared-memory programming model
  - An Application Programming Interface (API)
  - General-purpose computing on GPU (GPGPU)

- Multicore vs Manycore
  - Multicore – Small number of sophisticated cores
  - Manycore – Large number of weaker cores

Overview

- Why CUDA?
  - High level
    - C/C++/Fortran language extensions
  - Scalability
  - Thread-level abstraction
  - Runtime library
  - *Thrust* parallel algorithm library

- Limitations
  - Not vendor neutral: NVIDIA CUDA-enabled GPUs only
    - Alternative: *OpenCL*

This course will be in C

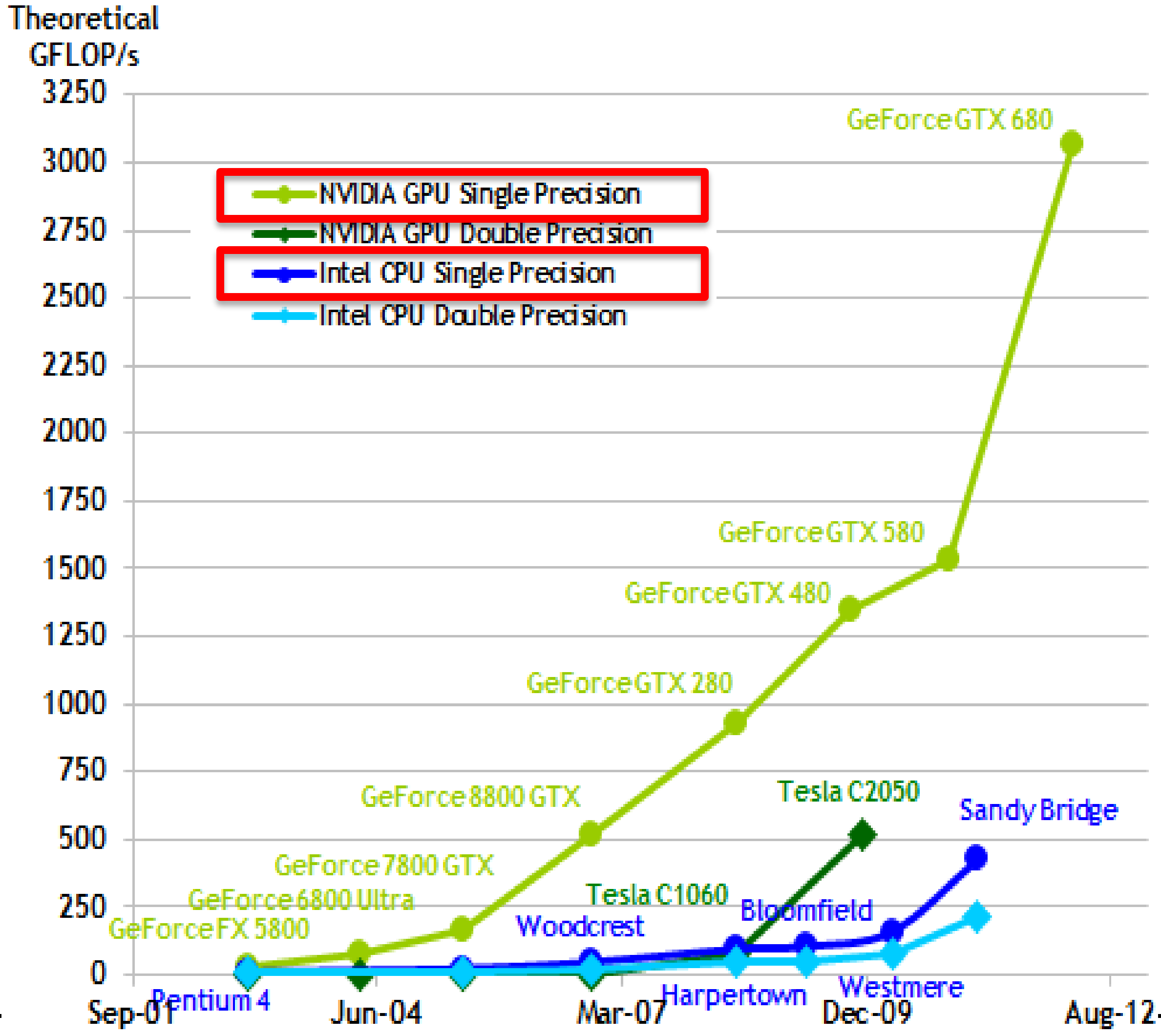
Overview

Why are we using GPU?

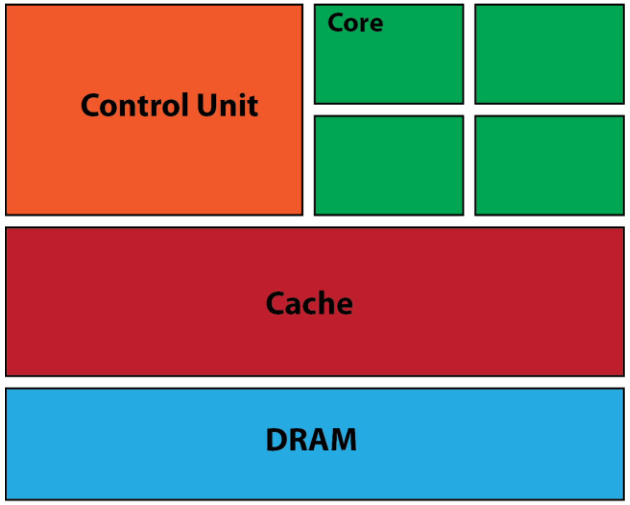
- Parallel and multithread hardware design
- Floating point computation
    - Graphic rendering
    - General-purpose computing
- Energy Efficiency
    - More FLOPS per Watt than CPU

- MIC vs GPU
    - Comparable performance
    - Different programming models

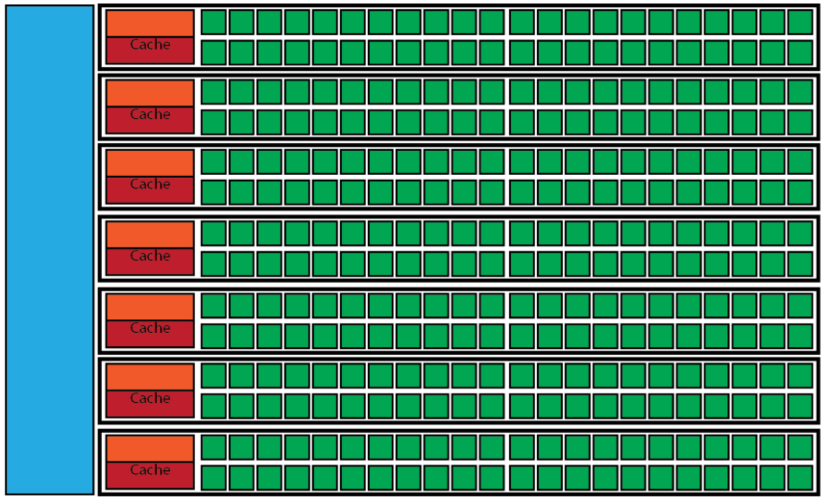
Overview


Theoretical GFLOP/s
3250
3000
2750
2500
2250
2000
1750
1500
1250
1000
750
500
250
0
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
GeForce GTX 680
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla C2050
Tesla C1060
Sandy Bridge
Bloomfield
Woodcrest
Harpertown
Westmere
Pentium 4
Sep-01
Jun-04
Mar-07
Dec-09
Aug-12

Overview | Heterogeneous Parallel Computing



CPU Architecture



GPU Architecture

Different designs for different purposes

- CPU: Fast serial processing
  - Large on-chip cache, to minimize read/write latency
  - Sophisticated logic control
- GPU: High computational throughputs
  - Large number of cores
  - High memory bandwidth

Overview Heterogeneous Parallel Computing

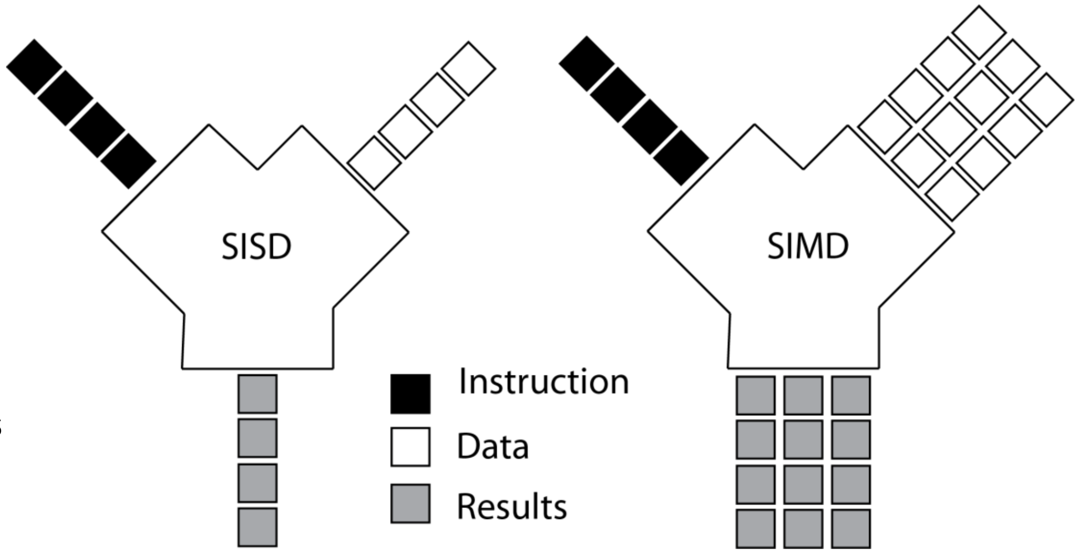
	Sandy Bridge E5 - 2680	NVIDIA Tesla K20
Processing Units	8	13 SMs, each with 192 cores, 2496 cores total
Clock Speed (GHz)	2.7	0.706
Maximum Threads	8 cores, 2 threads each = 16 threads	13 SMs, each with 192 cores, 32 way SIMD = 79872 threads
Memory Bandwidth	51.6 GB/s	205 GB/s
L1 Cache Size	64 KB/core	64 KB/SMs
L2 Cache Size	256 KB/core	768 KB, shared
L3 Cache Size	20MB	N/A

SM = Stream Multiprocessor

SIMD

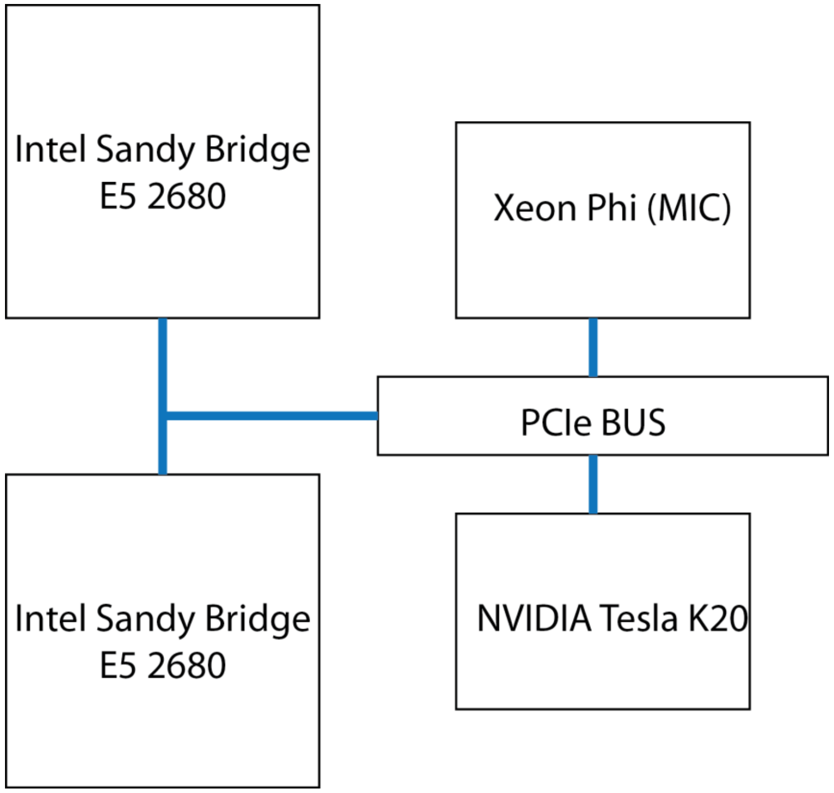
- SISD: Single Instruction Single Data
- SIMD: Single Instruction Multiple Data
  - A vector instruction that perform the same operation on multiple data simultaneously

- SIMD Instruction Sets:
  - MMX
    - Multimedia eXtension
  - SSE
    - Streaming SIMD Extensions
  - AVX
    - Advanced Vector Extensions



Overview | CUDA on Stampede

- 6400+ compute nodes, each has:
  - 2 Sandy Bridge processors (E5-2680)
  - 1 Xeon Phi Coprocessor (MIC)
- There are 128 GPU nodes, each is augmented with1 NVIDIA K20 GPU
- Login nodes do not have GPU cards installed!



CUDA on Stampede

Running your GPU application on Stampede:

- Load CUDA software using the *module* utility
- Compile your code using the NVIDIA *nvcc* compiler
  - Acts like a wrapper, hiding the intrinsic compilation details for GPU code
- Submit your job to a *GPU queue*

Lab 1: Querying Device

1. Extract the lab files to the home directory

```
$ cd $HOME
$ tar xvf  ~tg459572/LABS/Intro_CUDA.tar
```

2. Load the CUDA software

```
$ module load cuda
```

Lab 1: Querying Device

3. Go to lab 1 directory, *devicequery*

```
$ cd Intro_CUDA/devicequery
```

- There are 2 files:
  - Source code: ***devicequery.cu***
  - Batch script: ***batch.sh***

4. Use NVIDIA *nvcc* compiler, to compile the source code

```
$ nvcc -arch=sm_30 devicequery.cu  -o devicequery
```

Lab 1: Querying Device

5. Job submission:
   – Running 1 task on 1 node: *#SBATCH -n 1*
   – GPU development queue: *#SBATCH -p gpudev*

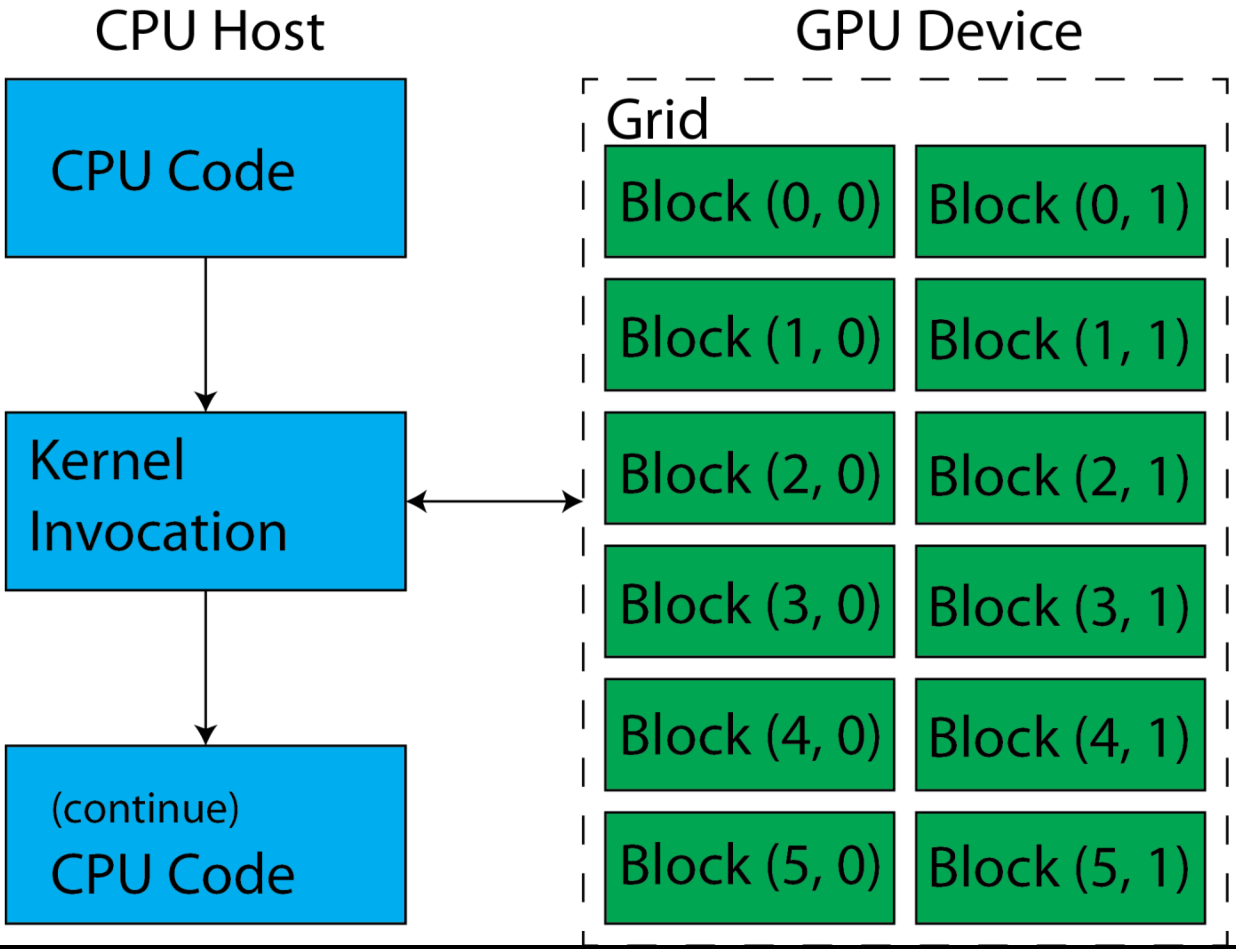
```
$ sbatch batch.sh
$ more gpu_query.o[job ID]
```

Queue Name	Time Limit	Description
gpu	24 hrs	GPU queue
gpudev	4 hrs	GPU development node
vis	8 hrs	GPU nodes + VNC service

Lab 1: Querying Device

```
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:          3
Minor revision number:          5
Name:                           Tesla K20m
Total global memory:            5032706048
Total shared memory per block:  49152
Total registers per block:      65536
Warp size:                      32
Maximum memory pitch:           2147483647
Maximum threads per block:      1024
Maximum dimension 0 of block:   1024
Maximum dimension 1 of block:   1024
Maximum dimension 2 of block:   64
Maximum dimension 0 of grid:    2147483647
Maximum dimension 1 of grid:    65535
Maximum dimension 2 of grid:    65535
Clock rate:                     705500
Total constant memory:          65536
Texture alignment:              512
Concurrent copy and execution:  Yes
Number of multiprocessors:      13
Kernel execution timeout:       No
```

Programming Structure



Basics | Programming Structure

- Host Code
  - Your CPU codes
  - Takes care of:
    - Device memory
    - Kernel invocation

```
int main() {
        …
        //CPU code
        [Invoke GPU functions]
        …
}
```

- Kernel Code
  - Your GPU code
  - Executed on the device
  - *__global__* qualifier
    - Must have return type *void*

```
__global___
void gpufunc(arg1, arg2, …){
        …
        //GPU code
        …
}
```

Basics | Programming Structure

- Function Type Qualifiers in CUDA
  - **__global__**
    - Callable from the host only
    - Executed on the device
    - void return type
  - **__device__**
    - Executed on the device only
    - Callable from the device only
  - **__host__**
    - Executed on the host only
    - Callable from the host only
    - Equivalent to declare a function without any qualifier
- There are **variable type qualifiers** available
- Visit the NVIDIA documentation for detail information

Basics Programming Structure

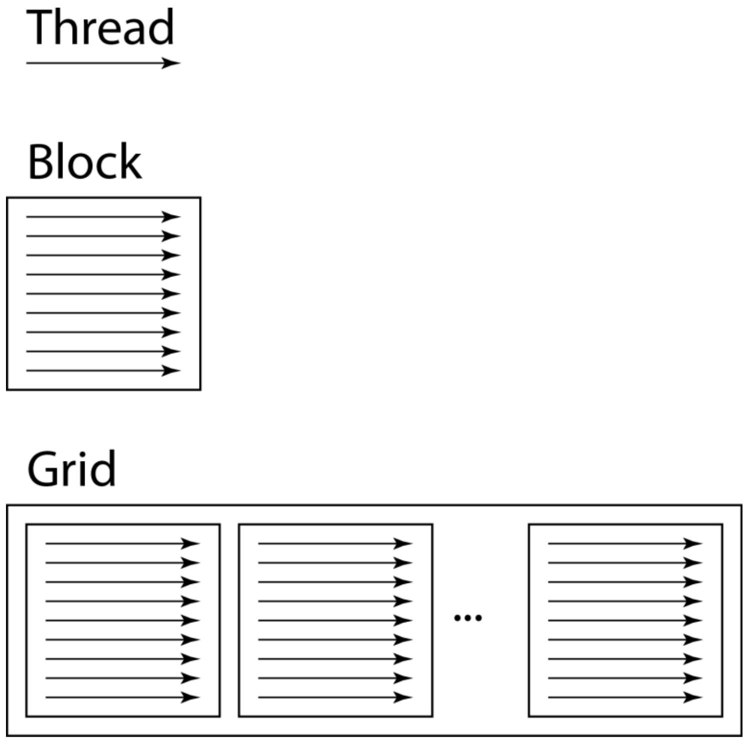
- Kernel is invoked from the host

```
int main() {
        …
        //Kernel Invocation
        gpufunc<<<gridConfig,
blkConfig>>>(arguments…)
        …
}
```

- Calling a kernel uses familiar syntax (function/subroutine call) augmented by Chevron syntax
- The Chevron syntax (<<<…>>>) configures the kernel
  - First argument: How many blocks in a grid
  - Second argument: How many threads in a block

Thread | Thread Hierarchies

- Thread
- Block
  - Assigned to a SM
  - Independent
  - Threads within a block can:
    - Synchronize
    - Share data
    - communicate
  - On K20: 1024 threads per block (max)
- Grid
  - Invoked kernel



Thread Keywords

- Threads and blocks have its unique ID
  - Thread: ***threadIdx***
  - Block: ***blockIdx***
- *threadIdx* can have maximum 3 dimensions
  - ***threadIdx.x***, ***threadIdx.y***, and ***threadIdx.z***
- *blockIdx* can have maximum 2 dimensions
  - ***blockIdx.x*** and ***blockIdx.y***
- Why multiple dimensions?
  - Programmer's convenience
  - Think about working with a 2D array

Thread Parallelism

Types of parallelism:

- Thread-level Task Parallelism
  - Every thread or group of threads, executes a different instruction
  - Not ideal because of thread divergence
- Task Parallelism
  - Different blocks perform different tasks
  - Invoke multiple kernels to perform different tasks
- Data Parallelism
  - Shared memory across threads and blocks

Thread Warp

Threads in a block are bundled into small groups of *warps*

- 1 warp = 32 Threads with consecutive ***threadIdx*** values
  - Example: [0..31] from the first warp, [32…63] from the second warp

- A full warp is mapped to the SIMD unit (Single Instruction Multiple Threads, *SIMT*)

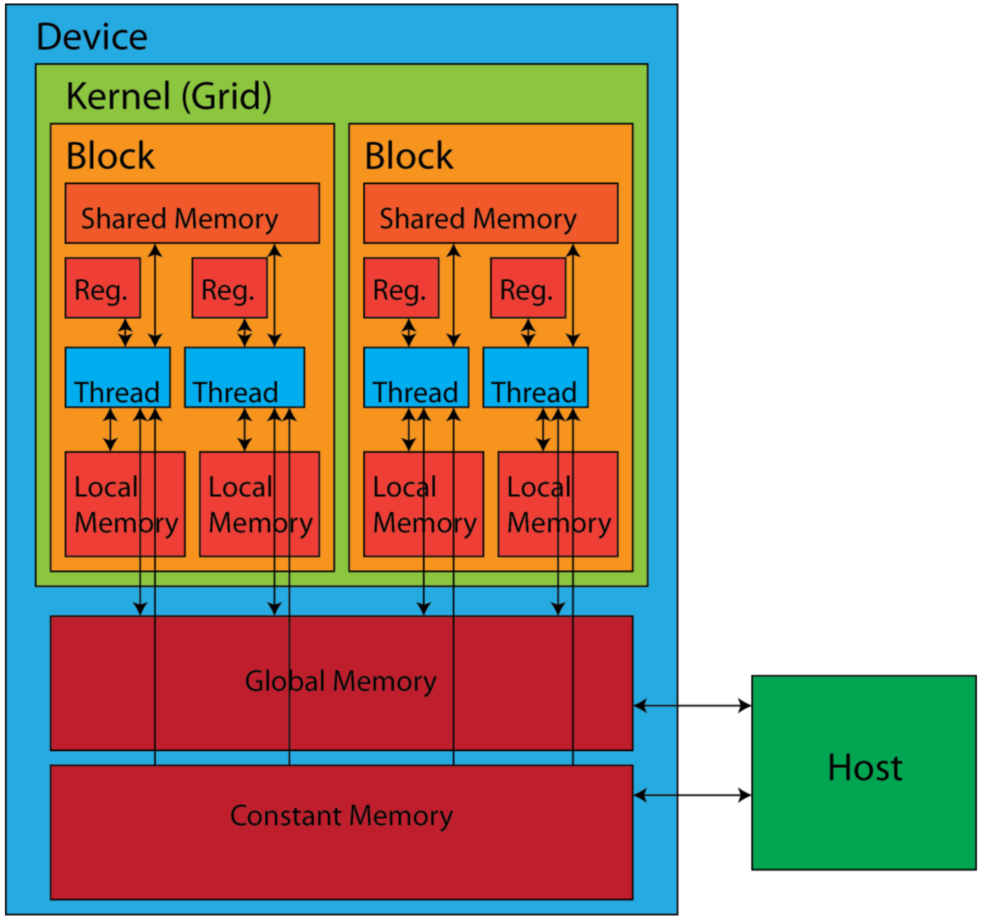
- Threads in a warp cannot diverge
  - Divergence serializes the execution

- Example: In an *if-then-else* construct:
  1. All threads will execute *then*
  2. then execute *else.*

Memory Model

- Kernel
  - per-device Global Memory
- Block
  - per-block shared memory
- Thread
  - per-thread register

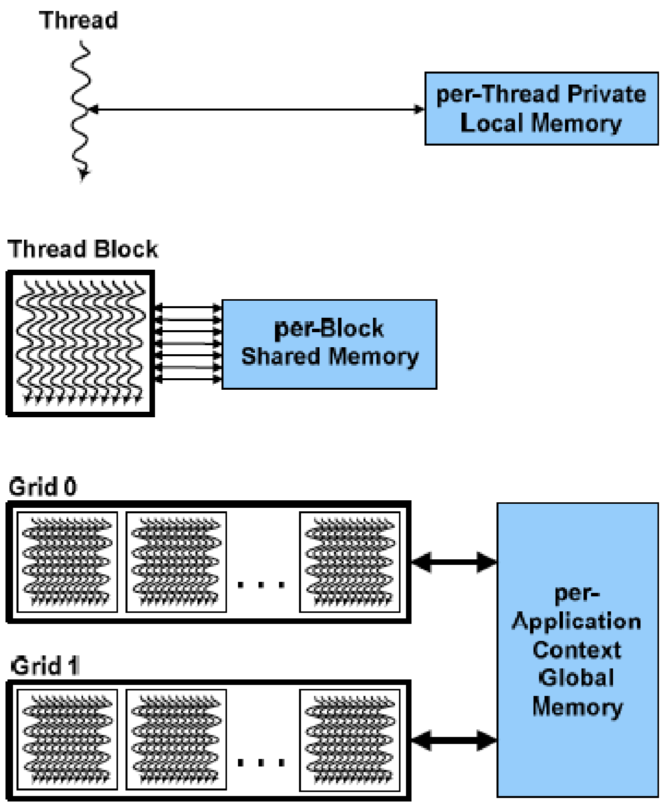
- CPU and GPU do not share memory



Two-way arrows indicate read/write capability

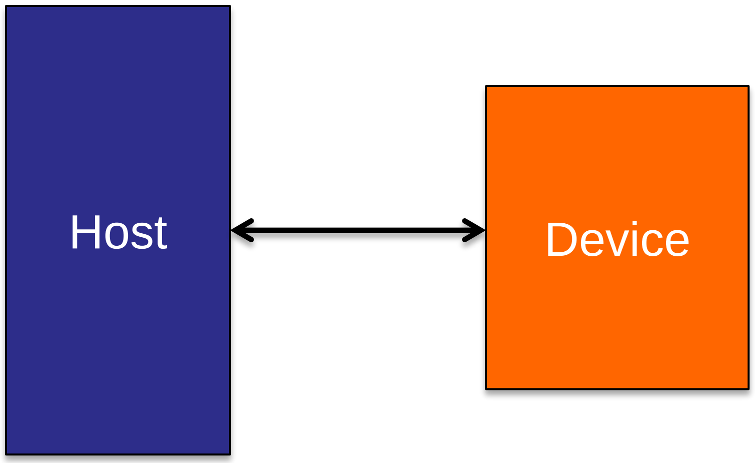
Memory Model

- per-thread register
  - Private, storage for local variables
  - Fastest
  - Life time: thread
- per-block shared memory
  - Shared within a block
  - 48k, fast
  - Life time: Kernel
  - __**shared**__ qualifier
- per-device global memory
  - Shared
  - 5G, Slowest
  - Life time: Application


Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory

Memory Transfer

- Allocate device memory
  - **cudaMalloc()**
- Memory transfer between host and device
  - **cudaMemcpy()**
- Deallocate memory
  - **cudaFree()**



Memory

Lab 2: Vector Add

h_varname : host memory
d_varname : device memory

```
int main()
{
        //Host memory allocation
        //Host Memory Allocation
        h_A=(float *)malloc(size);
        h_B=(float *)malloc(size);
        h_C=(float *)malloc(size);

        //Device memory allocation
        cudaMalloc((void **)&d_A, size);
        cudaMalloc((void **)&d_B, size);
        cudaMalloc((void **)&d_C, size);

        //Memory transfer, kernel invocation
        cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_C, h_C, size, cudaMemcpyHostToDevice);

        vec_add<<<N/512, 512>>>(d_A, d_B, d_C);

        cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);
        free(h_A);
        free(h_B);
        free(h_C);
}
```

Allocate host memory

Allocate device memory

Transfer memory from host to device

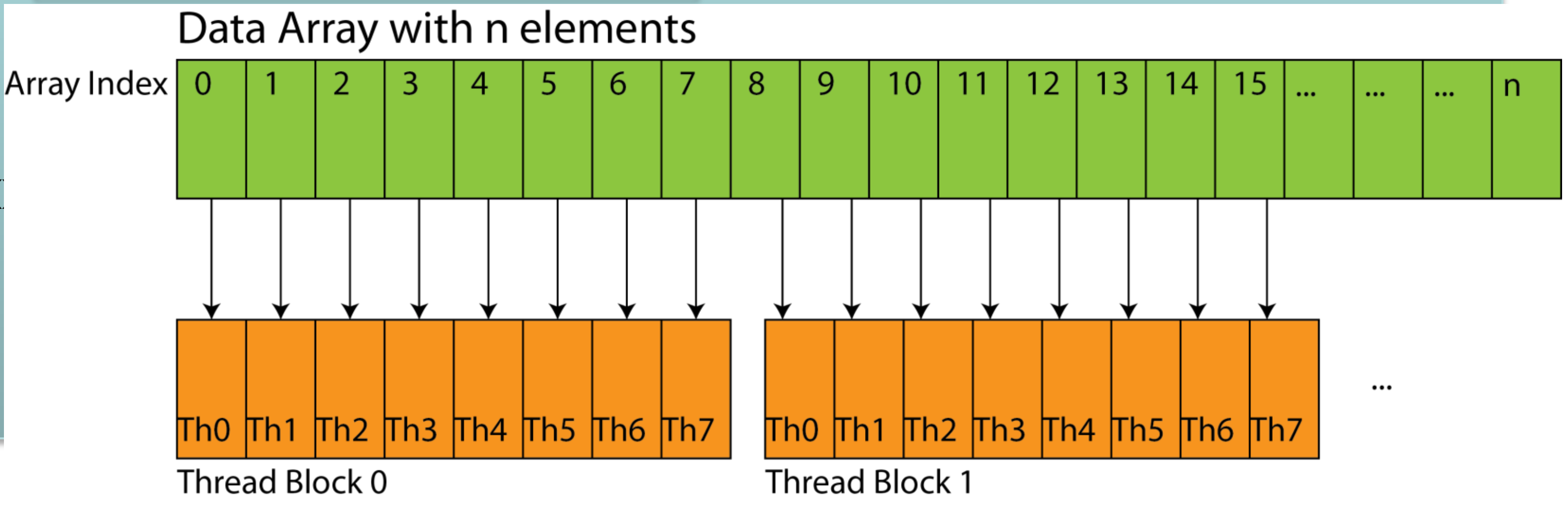
Invoke the kernel

Transfer memory from device to host

Deallocate the memory

Lab 2: Vector Add

```
//Vector Size
#define N 5120000

//Kernel function
__global__
void vec_add(float *d_A, float *d_B, float *d_C)
{
            //Define Index
            int i=blockDim.x * blockIdx.x + threadIdx.x;



}

int main(
{



}
```



Lab 2: Vector Add

```
$ cd $HOME/Intro_CUDA/vectoradd
$ nvcc -arch=sm_30 vectoradd.cu -o vectoradd
$ sbatch batch.sh
```

- Things to try on your own *(After the talk)*:
  - Time the performance using different vector length
  - Time the performance using different block size
- Timing tool:
  - ***/usr/bin/time –p <executable>***
  - CUDA also provides a better timing tool, see NVIDIA Documentation

Advanced Performance Topics

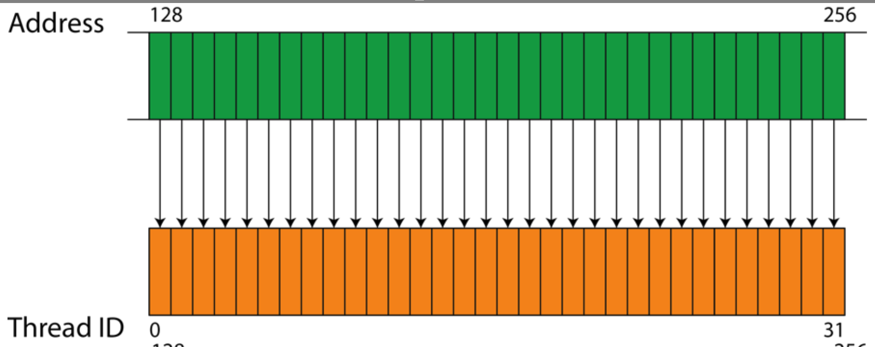
- Minimize execution divergence
  - Thread divergence serializes the execution
- Maximize on-chip memory (per-block shared, and per-thread)
  - Global memory is slow (~200GB/s)
- Optimize memory access
  - Coalesced memory access

Advanced Performance Topics

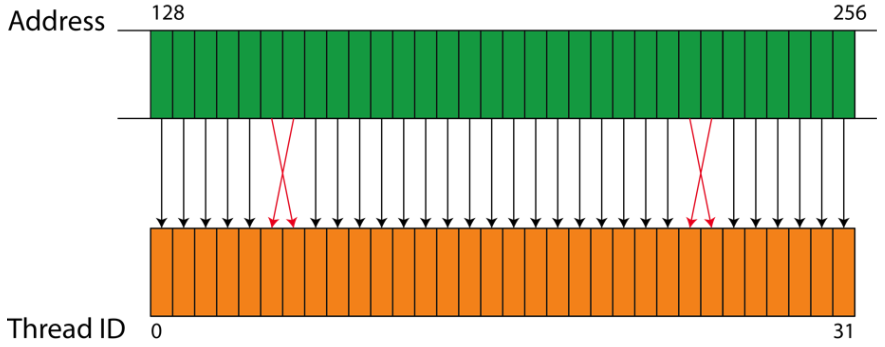
- What is *coalesced memory access?*
  - Combine all memory transactions into a single warp access
  - K20: 32 threads * 4-byte word = 128 bytes
- What are the requirements?
  - Memory alignment
  - Sequential memory access
  - Dense memory access

Advanced Performance Topics

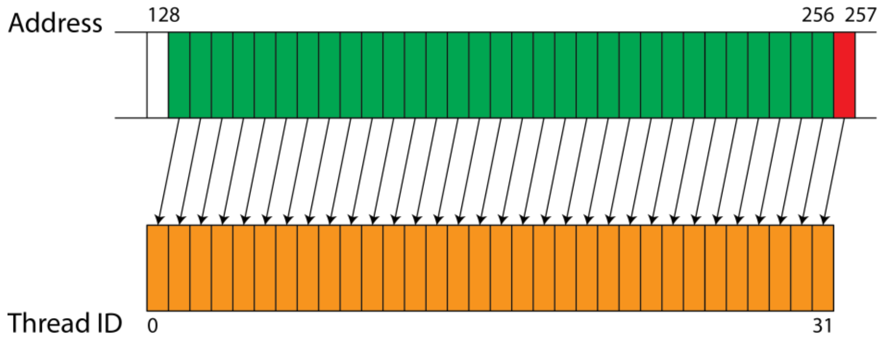
1 Transaction
Sequential and
aligned
(Stride 1)



1 Transaction
Non-sequential and
Aligned
(Stride 1)



2 Transactions
Sequential and
Misaligned
(Stride 2)



Advanced Performance Topics

- Consider the following code:
  - Is memory access aligned?
  - Is memory access sequential?

```
//The variable, offset, is a constant
int i=blockDim.x * blockIdx.x + threadIdx.x;
int j=blockDim.x * blockIdx.x + threadIdx.x + offset;
d_B2[i]=d_A2[j];
```

Summary

- GPU is very good at massive parallel jobs, and CPU is very good at serial processing
- Avoid thread divergence
- Use on-chip memory
- Always try to perform coalesced memory access

Final | Lab 3: Matrix Multiplication

```
$ cd $HOME/Intro_CUDA/matrix_mul
$ nvcc -arch=sm_30 matrix_mul.cu -o matmul
$ sbatch batch.sh
```

- Things to try on your own *(After the talk)*:
  - Compare the performance to the CUDA BLAS matrix multiplication routine
  - Can you improve the performance of it?
    - Hints:
      - Use on-chip memory
      - Use *page-locked* memory (see ***cudaMallocHost()***)

References

Recommended Reading:

- CUDA Documentation

- Hwu, Wen-Mei; Kirk, David. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*.