An introduction to

The CUDA programming environment

- Introduction : the rise of GPUs
- NVIDIA GPUS and CUDA
- Basic syntax
- Memory organization
- Examples

GPUs and the CUDA environment

► GPUs [Graphics Processing Units] are very powerful co-processors for graphics.

- Idea: why not use them for numerical computing?
- GPUs are present in every workstation for graphics processing

Find out what graphics card you have on your desktop computer or laptop..

Characteristics:

-large data arrays, streaming data

-fine-grain SIMD computations

-single precision floating point computation

- Difficulty: software.
- Solution: CUDA
- CUDA = Compute Unified Device Architecture
- Introduced in 2006 for NVIDIA GPUs
- Idea of attached processor [or co-processor] Not new [e.g. FPS AP-120B 'array processor' unveiled in 1981]

5-1

Terminology

- ► GPGPU : General purpose GPU
- ► GPU-accelerated computing: use GPUs along a CPU to speed-up computing

5-3

GPUs and the CUDA environment

Currently a very popular approach to: inexpensive supercomputing

5-2

See a series of articles in 2008 - when this whole thing started: CUDA - supercomputing for the masses by Rob Farber in <u>'Dr. Dobbs'</u>

- You can buy a Teraflop peak power for around \$1,500.
- \blacktriangleright Amazingly this price has remained \sim the same Difference: you get more from one GPU Example Tesla Products

Megatrend: GPU Performance being tuned for Deep Learning (single precison 'tensor-flops', vs FP64 teraflops).

5-4

GPU model	Price	FP64 Perf.	\$ / TFLOPS	DL (FP32) Perf.	\$/ TensFPS
V100 16GB	\$10,664*	7 TFLOPS	\$1,523	112 TFLOPS	\$95.21
32GB	\$11,458*		\$1,637		\$102.3
P100 (16GB)	\$ 7,374	4.7 TFLOPS	\$ 1,569	18.7 TFLOPS	\$394.33

* Note the huge jump in performance for Deep learning made in recent generation GPUs (Tesla V100).

 $^* \sim$ 10 years ago: 1 TFLOPS for approximately \$1,350 (Tesla C2050) [see that Dr. Dobbs article]

The NVIDIA products

4 families

- **Tegra:** Mobile and embedded devices (e.g., phones)
- GeForce: Consumer graphics, gaming
- Quadro: High-performance visualization
- **Tesla:** High performance computing (Tesla M2050)



CUDA Cores	2560
NVIDIA Turing Tensor Cores	320
Memory	16 GB GDDR6 w. ECC
Memory Interface	256-bit
Memory Bandwidth	320 GB/s
Single Precision Floating Point Perf.	8.1 TFLOPS
	(w. GPU Boost Clock)
Mixed Precision (FP16 / FP32)	65 TFLOPS
INT8-Precision	130 TOPS
INT4-Precision	260 TOPS
System Interface	PCI-Express 3.0 x16
Max. Power Consumption	70 Watt

Example: NVIDIA GeForce RTX 2080 Ti

- CUDA cores: 4.352
- Base Clock speed: 1350MHz
- Boosted Clock speed: 1545MHz
- FP32 peak speak: 13.44 TFlops
- RTX-OPS : 76T
- Memory capacity: 11GB GDDR6
- Memory bandwidth: 616 GB/sec
- Memory speed: 14 Gbps
- Memory interface width: 352-bit
- Memory bandwidth: 616GBps See Comparisons



- ► Host processor (CPU) and Device (GPU)
- > Model built around many threads executed on the device

5-9



SIMT: Single Instuction Multiple Threads

- cuda

- > A Kernel == a piece of code executed on the device
- > Each kernel is run in a thread. Blocks of threads are executed on a Streaming Multiprocessor (SM). Details later.
- > Idea: generate many threads (in the form of an SIMT code) which will be run on the GPU

5-11

 \blacktriangleright Host code may be C, C++, fortran90, ...

5-11

> Kernels are in C with CUDA syntax extensions

The CUDA environment: The big picture

5-10

> A host (CPU) and an attached device (GPU)

Typical program:

1. Generate data on CPU 2. Allocate memory on GPU cudaMalloc(...) 3. Send data Host \rightarrow GPU cudaMemcpy(...) 4. Execute GPU 'kernel': kernel <<<(...)>>>(..) 5. Copy data GPU \rightarrow CPU cudaMemcpy(...)





- > A group of 32 Threads is a Warp
- ► Warps grouped into thread Blocks
- \blacktriangleright Blocks have $\leq 1,024$ threads
- > Thread blocks are grouped into grids.

Thread \rightarrow Block of Threads \rightarrow Grid of Blocks

> Lots of flexibility in selecting block/grid shapes and dimensions

5-13

Documentation



- ▶ Blocks may be 1-D, 2-D, or 3-D,
- Grids can also be 1-D, 2-D, or 3-D
- > Related kernel variables:

Grid: gridDim, blockIdx, Block: blockDim, threadIdx

blockIdx, threadIdx are 3-Dimensional - can invoke

blockldx.x, blockldx.y, blockldx.z

and:

5-13

threadIdx.x, threadIdx.y, threadIdx.z

Function Type Qualifiers

__device__ : declares a function which executes on device. [Callable from the device only.]

5-14

__global__ declares a *kernel* function - which is Executed on device, Callable from host only.

__host__ declares a host function [executed on host, callable from host only]

If no qualifiers \rightarrow considered *host* [but can also combine __host__ and __device__]

► There are some restrictions – see docs. For example recursion not supported on device. ...

5-16

5-15

– cuda

Hello World in Cuda-ish:

```
#include <stdio.h>
__global__ void helloFromGPU(){
    printf("Hello World-Thread: %d\n",threadIdx.x);
}
int main(void) {
    helloFromGPU<<<1,16>>>();
    cudaDeviceSynchronize();
    return(0);
}
```

Example:

```
// Kernel definition:
__global__ void vecAdd(float *x,float *y,float *z)
{
    int i = threadIdx.x;
        z[i] = x[i] + y[i];
}
```

int main {

```
/* Kernel call: [1 Block of $n$ threads] */
vecAdd <<<1, n>>> (xd, yd, zd);
}
```

CUDA environment: Basic syntax

Kernels are called with the <<<>>> construct:

some_kernel_fun <<< Dg, Db, Ns>>>

5-17

- Dg = dimensions of the grid (type dim3)
- Db = dimensions of the block (type dim3)

▶ What is type dim3? An integer vector type [uint3] - used to specify dimensions

5-19

- Declare as: dim3 var(dimx, dimy, dimz),
- ... retrieve components as: var.x, var.y, var.z
- Unspecified components set to 1

5-19

Built-in variables

gridDim is of type dim3. Contains dimension of grid. Similarly for blockDim

5-18

> Can retrieve block dimensions from

blockDim.x, blockDim.y, blockDim.z

- blockIdx (type: uint3) contains block ID within grid
- threadIdx (type: uint3) contains thread index within block.

5-20

5-20

– cuda

Example:

Example:

__global__ void KernelFun(..)
//host:
dim3 DimGrid(200,10); //2000 thread blocks
dim3 DimBlock(4,8,8); //256 threads per block
size_t SharedMemBytes=64;//shared mem. per block
KernelFun<<<DimGrid,DimBlock,SharedMemBytes>>>(..)

- How to get index of a thread?
- > For a 1-D block: Index of a thread & its thread ID are the same

For a 2-D block of size (Dx, Dy): thread ID of a thread of index (x, y) is (x + y*Dx);

➤ For 3-D blocks of size (Dx, Dy, Dz): thread ID of a thread of index (x, y, z) is (x + y*Dx + z*Dx*Dy).



CUDA environment: Memory Hierarchy



Threads can access their local memories, shared memory of their block, and global memory.

5 - 23

$CUDA \ environment: \ Device \ {\mathcal C} \ Host \ Memory$

- > Device (GPU) memory distinct from that of host.
- Kernels operate only on device memory
- Also: Texture memory [called CUDA arrays] –
- Can allocate device memory with cudaMalloc()
- Copy from host to device with cudaMemcpy()
- Can also use cudaMallocPitch(),cudaMalloc3D(), cudaMemcpy2D(), cudaMemcpy3D(), [see prog. guide]

5-24

5-23

CUDA environment: Snarea vs. Giobal Memory	$CUDA \ documentation, \ resources$
 By default, the kernel will use global memory However, shared memory is *much* faster and should be used when possible Declarations: shared float, int, 	 Main document from the <u>CUDA main site</u> A PDF document also available [short-cut available in Canvas] General documentation site: <u>Here</u> CUDA sample source codes: <u>Here</u>
5-25 cuda 5-25	5-26 cuda 5-26
 New: openACC Note: Under development. Main Idea: use directives – Very similar to openMP Supported by vendors: there is a chance it will replace CUDA (?) Importantly: it is now part of gcc.7.xx Example: : product of two vectors Much simpler than under CUDA [used to be test1.cu] Available and works on Veggie cluster [gcc version 9.xx installed] Docs: See this page → https://gcc.gnu.org/wiki/OpenACC for status and the openACC homepage → https://www.openacc.org/ 	<pre>file main(void){ float *x, *y; /*</pre>

5-27

5-28

openACC: A few important directives

- *#pragma acc parallel* Defines a parallel region
- *#pragma acc kernels* Gives hint to compiler that a block that can 'kernelized'
- *#pragma acc loop* defines the type of parallelism in parallel or kernels region.
- #pragma acc data defines and copies data to / from device
- > On Canvas: full documentation from openACC-standard.ORG
- Book by Kirk and Hwu (see syllabus): Chapter 19
- > See sample code VecAdd on Canvas for illustration
- \blacktriangleright Group assignment: Mat-Add in openACC \rightarrow with a goal of getting best performance.

5-29	- cuc	ta
	5-29	_

Run-time functions

```
int acc_get_num_devices( acc_device_t );
void acc_set_device_type( acc_device_t );
acc_device_t acc_get_device_type( void );
int acc_get_device_num( acc_device_t );
void acc_set_device_num( int, acc_device_t );
...
```

Some of these can also be set via environment variables

30		– cuda
• • •		
export	ACC_DEVICE_NUM=1	
setenv	ACC_DEVICE_NUM 1	
tcsh:		
export	ACC_DEVICE_TYPE=NVIDIA	
setenv	ACC_DEVICE_TYPE NVIDIA	
tcsh:		