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]

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

Currently a very popular approach to: inexpensive supercomputing

- See a series of articles in 2008 - when this whole thing started: CUDA - supercomputing for the masses by Rob Farber in ‘Dr. Dobbs’
  http://www.ddj.com/cpp/207200659
- You can buy a Teraflop peak power for around $1,500.
- Amazingly this price has remained ~ the same – Difference: you get more from one GPU

Megatrend: GPU Performance being tuned for Deep Learning (single precision 'tensor-flops', vs FP64 teraflops).
<table>
<thead>
<tr>
<th>GPU model</th>
<th>Price</th>
<th>FP64 Perf.</th>
<th>$ / TFLOPS</th>
<th>DL (FP32) Perf.</th>
<th>$ / TensFPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>V100 16GB</td>
<td>$10,664*</td>
<td>7 TFLOPS</td>
<td>$1,523</td>
<td>112 TFLOPS</td>
<td>$95.21</td>
</tr>
<tr>
<td>32GB</td>
<td>$11,458*</td>
<td>4.7 TFLOPS</td>
<td>$1,569</td>
<td>18.7 TFLOPS</td>
<td>$394.33</td>
</tr>
</tbody>
</table>

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

* ∼ 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)

### Example: The ‘cudaxx’ cluster in cselabs

To do in class: Look at the ‘cudaxx’ cluster – Analyze one node: “cuda01.cselabs.umn.edu” –

- What GPU?
  - Use the command `lspci`: Explore the unix command `lspci` before class. Look for “GPU” or “Graphics”
  - PCI: Peripheral Component Interconnet [bus that attaches peripheral devices, e.g., USB, audio, RAID, Ethernet, …]
  - Another (unix) command: `nvidia-smi` (Nvidia System Management Interface) – For nvidia GPUs only

Read about compute capability in Nvidia Documentation. What is it for the nodes of the cudaxx cluster?


### 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
CUDA environment: Device and Host

- Host processor (CPU) and Device (GPU)
- Model built around many threads executed on the device
  - SIMT: Single Instruction Multiple Threads
- 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
- Host code may be C, C++, fortran90, ..
- Kernels are in C with CUDA syntax extensions

Threads, Warps, Blocks, and Grids

- A group of 32 Threads is a Warp
- Warps grouped into thread Blocks
- Blocks have ≤ 1,024 threads
- Thread blocks are grouped into grids.
  - Thread → Block of Threads → Grid of Blocks
- Lots of flexibility in selecting block/grid shapes and dimensions

The CUDA environment: The big picture

- 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 → GPU
cudaMemcpy(...)
4. Execute GPU ‘kernel’:
kernel <<<(...)>>>(..)
5. Copy data GPU → CPU
cudaMemcpy(...)

Threads, Warps, Blocks, and Grids

- A group of 32 Threads is a Warp
- Warps grouped into thread Blocks
- Blocks have ≤ 1,024 threads
- Thread blocks are grouped into grids.
  - Thread → Block of Threads → Grid of Blocks
- Lots of flexibility in selecting block/grid shapes and dimensions
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
blockIdx.x, blockIdx.y, blockIdx.z
and:
threadIdx.x, threadIdx.y, threadIdx.z

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:

```c
some_kernel_fun <<< Dg, Db, Ns >>>
```

- **Dg** = dimensions of the grid (type `dim3`)
- **Db** = dimensions of the block (type `dim3`)
- **Ns** = number of bytes shared memory dynamically allocated / block (type `size_t`). Default 0

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

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

**Example:**

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
```

```
int main() {
    ...
    // Kernel invocation
    dim3 dimBlock(N, N);
    MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

**Built-in variables**

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

**Example:**

```c
__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.

CUDA environment: Device & 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]

CUDA environment: Shared vs. Global Memory

- By default, the kernel will use global memory
- However, shared memory is *much* faster and should be used when possible
- Declarations:
  ```
  __shared__ float, int, ...
  ```

CUDA documentation, resources

- Main document from the CUDA site:
- A PDF document also available [short-cut available in Canvas]
- General documentation site:
  [https://docs.nvidia.com/](https://docs.nvidia.com/)
- CUDA sample source codes:
  [https://docs.nvidia.com/cuda/cuda-samples/index.html](https://docs.nvidia.com/cuda/cuda-samples/index.html)
**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]
- Need to load module gcc version 7: [not default on cudaxx cluster]

```c
int main(void){
    float *x, *y;
    /* -------------------- size of arrays */
    const int N = 20;
    size_t size = N * sizeof(float);
    /*------------------------ Allocate array and set values */
    y = (float *)malloc(size);
    x = (float *)malloc(size);
    for (int i=0; i<N; i++) {
        y[i] = (float) (i+1);
        x[i] = (float) (i-1);
    }
    #pragma acc parallel loop
    for (int i=0; i<N; i++)
        y[i] = y[i]*x[i];
    /* -------------------- print result */
    for (int i=0; i<N; i++)
        printf("%d %8.2f\n", i, y[i]);
    /* --------------------- free memory */
    free(x); free(y);
}
```

Docs and Details: [https://gcc.gnu.org/wiki/OpenACC](https://gcc.gnu.org/wiki/OpenACC)