High Performance Computing for Science and Engineering II

15.4.2019 - Lecture 8: Many-Core Processor Programming (CUDA)

Lecturer: Dr. Sergio Martin
Today's Class

A Brief History of Hardware Acceleration
- From x87 to NVIDIA Turing

CUDA Model
- Programming, rationale and examples.

Running CUDA
- Configuration, compilation, and running.
ANNouncing Today:
Intel to Build First Exascale Supercomputer for U.S. DoE

- Intel – partnered with Argonne National Laboratory – driving the convergence of HPC and AI

Aurora will accelerate the convergence of traditional HPC, data analytics, and AI

Intel's data centric portfolio at the heart of Aurora, integrated with Cray's “Shasta” system

Leading research & academia programs already engaged to harness Aurora and enable software ecosystem

"Achieving Exascale is imperative not only to better the scientific community, but also to better the lives of everyday Americans. Aurora and the next-generation of Exascale supercomputers will apply HPC and AI technologies to areas such as cancer research, climate modeling, and veterans' health treatments. The innovative advancements that will be made with Exascale will have an incredibly significant impact on our society."

- Rick Perry, US Secretary of Energy
The telescopes contributing to this result were ALMA, APEX, the IRAM 30-meter telescope, the James Clerk Maxwell Telescope, the Large Millimeter Telescope Alfonso Serrano, the Submillimeter Array, the Submillimeter Telescope, and the South Pole Telescope.

**Petabytes of raw data** from the telescopes were combined by **highly specialised supercomputers** hosted by the Max Planck Institute for Radio Astronomy and MIT Haystack Observatory.

Hardware Acceleration: A brief History
The Intel 8086 Processor

1978 - Intel Releases 8086, the first 16-bit processor of the x86 architecture.

- 29k Transistors - 5Mhz
- No FP-Unit
- No FP Registers

Address Calculator Unit
Segment Selectors (16-bit)
General-Purpose Registers (16-bit)

Further Read: Computer Organization and Design: The Hardware/Software Interface

Picture: https://commons.wikimedia.org/wiki/File:Intel_8086_block_scheme.svg
First Personal Computers

1980 - IBM releases DisplayWriter, one of the first word processing machines

Based on the 8086 Processor
Mainly for Word Processing

Further Read: https://en.wikipedia.org/wiki/IBM_Displaywriter_System

Picture Source: We Look 4 Things - Research Gate
Coprocessors

1980 - Intel Releases 8087, a floating-point math coprocessor

Add-on Chip on a separate Socket
Works together with the 8086 CPU
Improved FP Performance 50x

8-Deep 80-bit FP Register Stack

Further Read: https://en.wikipedia.org/wiki/Coprocessor
Further Read: https://en.wikipedia.org/wiki/X87
Personal Computers

1981 - IBM Releases the 5150, the first (widely successful) Personal Computer

Word Processing + CAD + DB + Spreadsheets
Gaming Industry as driver of progress

80's - 90's - PC/Console Gaming becomes mainstream.

Layers of 2D Sprites
Easy for CPU
(Mostly Integers)
Gaming Industry as driver of progress

90's - Commercial Success of 3D-Perspective Games

Higher Computational Demands

3D Algebra

Floating Point Operations

X-Rotation in 3D
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos\phi & -\sin\phi & 0 \\
0 & \sin\phi & \cos\phi & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Z-Rotation in 3D
\[
\begin{bmatrix}
\cos\phi & -\sin\phi & 0 & 0 \\
\sin\phi & \cos\phi & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Scale in 3D
\[
\begin{bmatrix}
x & 0 & 0 & 0 \\
y & 0 & 0 & 0 \\
z & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

(4x4)*(4x1) = (4x1)

Y-Rotation in 3D
\[
\begin{bmatrix}
\cos\phi & 0 & \sin\phi & 0 \\
0 & 1 & 0 & 0 \\
-\sin\phi & 0 & \cos\phi & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Translation in 3D
\[
\begin{bmatrix}
1 & 0 & 0 & Tx \\
0 & 1 & 0 & Ty \\
0 & 0 & 1 & Tz \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Matrix Multiplication
\[
\begin{bmatrix}
a & b & c & d \\
e & f & g & h \\
i & j & k & l \\
m & n & o & p
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
q
\end{bmatrix}
\]

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
q'
\end{bmatrix}
\]
Vectorization

1997 - Intel releases MMX, the first widely-deployed desktop SIMD instruction set.

A key development for multimedia operations.

Further Read: https://en.wikipedia.org/wiki/MMX_(instruction_set)

Image Source: https://www.pcmag.com/encyclopedia/term/38706/bitmap
Accelerator Cards.

"[...] A single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second." -NVIDIA

1998 - 3DFx Launches Voodoo2

1999 - Nvidia Launches Geforce 256

In other words... a Coprocessor

Source: https://www.youtube.com/watch?v=8leTSKPJLXc

Source: https://www.youtube.com/watch?v=CjHVDB5PbCw
Low Latency vs High Throughput

CPU
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU
- Optimized for data-parallel, throughput computation
- High memory latency tolerance
Low Latency vs High Throughput

CPU minimizes latency within each thread.

GPU hides latency with computation by scheduling different thread sets (warp).

**CPU core – Low Latency Processor**

**GPU Stream Multiprocessor – High Throughput Processor**

**Computation Thread/Warp**

- Processing
- Waiting for data
- Ready to be processed
- Context switch
Why GPUs?

General-Purpose Computing on GPU
1992 - SGI Releases OpenGL, the first open-source 2D/3D graphics pipeline API

Programming in Graphical Terms
- Vertices
- Triangles
- Colors
- Viewpoints
- Light sources

Greatly Simplified the task of developing multimedia (games)

What API's exist today?
General Purpose computing on GPUs (GPGPU)

Graphics Rendering = Lots of Algebra

Matrix & Vector Operations:
- Multiplication
- Inversion
- Rotation

But these are also useful for:
- Physics simulations
- Signal processing
- Computational geometry
- Computer vision
- Computational biology
- Computational finance
- Meteorology

For each frame, for every vertex...

\[
\begin{bmatrix}
X \\
Y \\
Z
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x_0 & y_0 & z_0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
X_p \\
Y_p \\
Z_p
\end{bmatrix} =
\begin{bmatrix}
x_0 & y_0 & z_0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

\[
x_p = x_0 \cdot \text{Camera Lens factor} \\
y_p = y_0 \\
z_p = z_0
\]

- 14 multiplies
- 12 adds
- 1 reciprocal

Image Source: https://www.anandtech.com/show/255
Nvidia CUDA

2007 - NVIDIA Releases CUDA, the first general purpose API for Many-core Programming

C extension to write GPU code

Only supported by Nvidia GPUs
(Open Source Alternatives: OpenCL, DirectCompute, etc.)

Many GPU-accelerated scientific computation libraries.

NVIDIA cuBLAS
NVIDIA cuRAND
NVIDIA cuSPARSE
NVIDIA NPP
GPU VSIPL
CUDA tools
IMSL Library
Cusp
GPU Accelerated Linear Algebra
Matrix Algebra on GPU and Multicore
Sparse Linear Algebra
Building-block Algorithms for CUDA
C++ STL Features for CUDA
2018 - Nvidia Launches the *Turing* GPU Microarchitecture

- 4608 Cuda Cores (GP)
- 576 Tensor Cores (AI/ML)
- 72 Ray Tracing Cores (Graphics)
- 16.3 Peak FP32 TFLOPS
- 6Mb L2 Cache

72 Streaming Multiprocessing Units

Source: nvidia.com
Turing TU102 Streaming Multiprocessor (SM)

Up to 4608 threads executing concurrently.
Warp Scheduling

A warp is a team of (software) threads that:

- Are scheduled together.
- Share a common instruction dispatcher.
- Execute the same code simultaneously (SIMT)
- Each executed by a different CUDA core in the SM.
- Share a common shared cache space.
- Operate on different data.

Every SM (hardware) has a warp scheduler.
CUDA Example: Code

```c
#include <iostream>
#include <cassert>

__global__ void scaleVector(float scale, float *input, float *output)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N)
    {
        output[tid] = input[tid] * scale;
    }
}

int main()
{
    float *a = new float[N];
    float *b = new float[N];
    float *dev_a; float *dev_b;
    const float scale = 2.1;
    for (int i=0; i<N; i++)
        all = (float)i/2;

    std::cout << "Initializing data on GPU\n";
    cudaMalloc( (void**)&dev_a, N*sizeof(float) );
    cudaMalloc( (void**)&dev_b, N*sizeof(float) );
    cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice);

    std::cout << "Launching kernels on GPU\n";
    const int nblocks = 128;
    const int nthreads = 256;
    scaleVector<<< nblocks, nthreads >>>(scale, dev_a, dev_b);

    std::cout << "Downloading data\n";
    cudaMemcpy( b, dev_b, N*sizeof(float), cudaMemcpyDeviceToHost);

    std::cout << "Verifying results\n";
    for (int i=0; i<N; i++)
    {
        std::cout << b[i] << std::endl;
        assert((double)i == b[i]);
    }

    std::cout << "Done\n";
    delete[] a;
    delete[] b;
}
```

device code (runs on GPU)

host code (runs on CPU)
CUDA Example: Compilation

Device Code

```
__global__ void kernel()
{
    // do GPU stuff
}
```

cpu.cpp

CPU Code

```
int main()
{
}
```

CUDA Function Qualifiers:

- `__global__` : called from CPU, runs on GPU
- `__device__` : called from GPU, runs on GPU
- `__host__` : called from CPU, runs on CPU (default)

`__host__` and `__device__` can be combined
__global__ void mykernel(void)
{
    printf("Hello World, I'm GPU!\n");
}

int main(void)
{
    mykernel<<<1,1>>>();
    printf("Hello World, I'm CPU!\n");
    return 0;
}
1. Copy input data from CPU memory to GPU memory
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory
Example: Vector/Vector Addition

A simple kernel to add two integers

```c
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}
```

In this case `__global__` determines that `add()` will execute on the device;
`add()` will be called from the host.

- We use pointers for the variables.
- `add()` runs on the device, so `a`, `b` and `c` must point to device memory.
- We need to allocate memory on the GPU.
Memory Management

Host and device memory are separate entities

Device pointers point to GPU memory
- May be passed to/from host code
- May not be dereferenced in host code

Host pointers point to CPU memory
- May be passed to/from device code
- May not be dereferenced in device code

CUDA API for handling device memory:
cudaMalloc(), cudaFree(), cudaMemcpy()

Similar to the C equivalents
malloc(), free(), memcpy()
cudaMemcpy(void* dst, void* src, size_t num_bytes, enum cudaMemcpyKind direction)

Where direction can be any of:

- cudaMemcpyHostToDevice  Host -> Device
- cudaMemcpyDeviceToHost  Device -> Host
- cudaMemcpyDeviceToDevice Device -> Device
Kernel Instantiation

mykernel<<<gridDim, blockDim>>>(...);

Triple angle brackets mark a call from host code to device code.

This is called a “Kernel Launch”.

- **gridDim** is the number of instances of the kernel
- **blockDim** is the number of threads within each instance

These values may be 1D, 2D, or 3D vectors (type vec3).

This topology helps with representing multi-dimensional problems.
GPU Parallelism

1D Vector Add (1k Elements)

Software

1024 Threads

64 Blocks

1 Grid

Hardware

256 GPU Cores

16 SM

1 GPU

Example

(times 4)

(times 4)

(times 16)

1D Vector Add (1k Elements)

1024 Threads

64 Blocks

1 Grid

256 GPU Cores

16 SM

1 GPU

Example

(times 4)

(times 4)

(times 16)
Kernel Topology

\[
\text{mykernel}<<<1, 1>>>(\ldots); \\
\text{1 Block / 1 Thread each}
\]

\[
\text{mykernel}<<<16, 1>>>(\ldots); \\
\text{N Blocks / 1 Thread each}
\]
Kernel Topology

mykernel<<<1, 16>>>(...);
1 Block / 16 Threads each

mykernel<<<16, 16>>>(...);
16 Blocks / 16 Threads each
Block Scheduling

With 16 SM?

16 Blocks * 1 Thread

With 4 SM, which is better?

With 16 SM?

1 Blocks * 16 Thread
#define N 1024
int main(void)
{
    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c
    // and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
Vector Addition: `main()`

// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;
__global__ void add(int *a, int *b, int *c)
{
    int myElement = ...;
    c[myElement] = a[myElement] + b[myElement];
}

We need to define myElement for each CUDA thread.

• Thread/Block Identification Variables
  • blockDim size (or dimensions) of each block
  • blockIdx index (or 2D/3D indices) of block
  • blockDim size (or dimensions) of each block
  • threadIdx index (or 2D/3D indices) of thread
Vector Addition on the Device

```c
__global__ void add(int *a, int *b, int *c)
{
  int myElement = blockIdx.x;
  c[myElement] = a[myElement] + b[myElement];
}

mykernel<<<1024, 1>>>(...);

1024 Blocks / 16 SM = 64 "Latencies"

mykernel<<<1, 1024>>>(...);

__global__ void add(int *a, int *b, int *c)
{
  int myElement = threadIdx.x;
  c[myElement] = a[myElement] + b[myElement];
}

1024 Threads / 16 Threads/SM = 64 "Latencies"
```
#define N 1024
#define THREADS_PER_BLOCK 16

mykernel<<<N/ThreadsPerBlock, ThreadsPerBlock>>>(…);

__global__ void add(int *a, int *b, int *c)
{
    int myElement = blockIdx.x * blockDim.x + threadIdx.x;
    c[myElement] = a[myElement] + b[myElement];
}

1024 Threads / (16 SM*16 Threads/SM) = 4 "Latencies"
Arbitrary Vector Sizes

**Problem:** Not every problem can be set as even multiples of `blockDim.x`

**Solution:** Avoid accessing beyond the end of the arrays:

```c
__global__ void add(int *a, int *b, int *c, int n)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x
    if (index < n)
        c[index] = a[index] + b[index];
}
```
Warp Coherency

Control Flow

if (...)
{
    // then-clause
}
else
{
    // else-clause
}

Execution within warps is coherent

© NVIDIA 2011
Divergent Execution

Control Flow

if ( ... )
{
    // then-clause
}
else
{
    // else-clause
}

Execution diverges within a warp
2D Geometry: Square Matrix Multiplication

2D Matrices (NxN)
1D (row by column) operation
Complexity: O(N^3)

each thread executes a 1D kernel

```
for (int i=0; i<N; i++)
for (int j=0; j<N; j++)
for (int k=0; k<N; k++)
c[i*N+j] += a[i*N+k] * b[k*N+j];
```

Parallelize
2D Geometry

```
for (int i=0; i<N; i++)
for (int j=0; j<N; j++)
```

Each thread
Executes a 1D Kernel

```
for (int k=0; k<N; k++)
c[i*N+j] += a[i*N+k] * b[k*N+j];
```
A matrix multiplication kernel

```c
__global__ void matrix(float * a, float * b, float * c, int N) {
    int ix = threadIdx.x + blockIdx.x*blockDim.x;
    int iy = threadIdx.y + blockIdx.y*blockDim.y;
    if (ix<N && iy<N) {
        c[ix*N + iy] = 0;
        for (int k=0; k<N; k++)
            c[ix*N + iy] += a[ix*N + k] * b[k*N + iy];
    }
}
```

And call it by creating a 2D grid

```c
dim3 blocks(N/4,N/4);
dim3 threads(4,4);
matrix<<< blocks, threads >>>(dev_a, dev_b, dev_c,N);
```
Kernel launches are **asynchronous**

CPU needs to synchronize before consuming the results

```c
cudaDeviceSynchronize()
```

Blocks the CPU until all preceding CUDA calls have completed

```c
cudaMemcpy()
```

Blocks the CPU until the copy is complete.
Copy begins when all preceding CUDA calls have completed.

```c
cudaMemcpyAsync()
```

Asynchronous, does not block the CPU
Reporting Errors

All CUDA API calls return an error code (`cudaError_t`) 

Get the error code for the last error:

```c
    cudaError_t cudaGetLastError()
```

Get a string to describe the error:

```c
char *cudaGetErrorString(cudaError_t)
if(cudaGetLastError() != cudaSuccess)
    std::cerr << cudaGetErrorString(cudaGetLastError());
```
Device Management

Application can query and select GPUs

- `cudaGetDeviceCount(int *count)`
- `cudaSetDevice(int device)`
- `cudaGetDevice(int *device)`
- `cudaGetDeviceProperties(cudaDeviceProp, prop, int device)`

A single host thread can manage multiple devices

- `cudaSetDevice(i)` to select current device
- `cudaMemcpy(...)` for peer-to-peer copies