CS/EE 217
GPU Architecture and Programming

Lecture 2:
Introduction to CUDA C
CUDA /OpenCL – Execution Model

• Integrated host+device app C program
  – Serial or modestly parallel parts in **host** C code
  – Highly parallel parts in **device** SPMD kernel C code

Serial Code (host) → Parallel Kernel (device) → Serial Code (host)

KernelA<<< nBlk, nTid >>>(args);

Serial Code (host) → Parallel Kernel (device) → Serial Code (host)

KernelB<<< nBlk, nTid >>>(args);
From Natural Language to Electrons

Natural Language (e.g., English)

Algorithm

High-Level Language (C/C++…)

Instruction Set Architecture

Microarchitecture

Circuits

Electrons

©Yale Patt and Sanjay Patel, From bits and bytes to gates and beyond
The ISA

• An Instruction Set Architecture (ISA) is a contract between the hardware and the software.

• As the name suggests, it is a set of instructions that the architecture (hardware) can execute.
A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.

- Program instructions operate on data stored in memory or provided by Input/Output (I/O) device.
The Von-Neumann Model

- Memory
- I/O
- Processing Unit
  - ALU
  - Reg File
- Control Unit
  - PC
  - IR
Arrays of Parallel Threads

• A CUDA kernel is executed by a grid (array) of threads
  – All threads in a grid run the same kernel code (SPMD)
  – Each thread has an index that it uses to compute memory addresses and make control decisions

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

...
Thread Blocks: Scalable Cooperation

- Divide thread array into multiple blocks
  - Threads within a block cooperate via *shared memory, atomic operations* and *barrier synchronization*
  - Threads in different blocks cannot cooperate

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

...
blockIdx and threadIdx

• Each thread uses indices to decide what data to work on
  – blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  – threadIdx: 1D, 2D, or 3D

• Simplifies memory addressing when processing multidimensional data
  – Image processing
  – Solving PDEs on volumes
  – …
Vector Addition – Conceptual View

vector A

\[
\begin{align*}
\end{align*}
\]

\[+\]

vector B

\[
\begin{align*}
\end{align*}
\]

\[+\]

vector C

\[
\begin{align*}
\end{align*}
\]

\[+\]
Vector Addition – Traditional C Code

// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
    for (i = 0, i < n, i++)
        C[i] = A[i] + B[i];
}

int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    ...
    vecAdd(A_h, B_h, C_h, N);
}
Heterogeneous Computing vecAdd
Host Code

```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n) {
    int size = n* sizeof(float);
    float* A_d, B_d, C_d;
    ...
    1. // Allocate device memory for A, B, and C
       // copy A and B to device memory
    2. // Kernel launch code – to have the device
       // to perform the actual vector addition
    3. // copy C from the device memory
       // Free device vectors
}
```
Partial Overview of CUDA Memories

- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory

- Host code can
  - Transfer data to/from per grid global memory

We will cover more later.
CUDA Device Memory Management API functions

- **cudaMalloc()**
  - Allocates object in the device **global memory**
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes

- **cudaFree()**
  - Frees object from device global memory
    - **Pointer** to freed object

- **cudaMalloc()**
  - Allocates object in the device **global memory**
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes

- **cudaFree()**
  - Frees object from device global memory
    - **Pointer** to freed object
Host-Device Data Transfer API functions

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous
void vecAdd(float* A, float* B, float* C, int n) {
    int size = n * sizeof(float);
    float* A_d, B_d, C_d;

1. // Transfer A and B to device memory
    cudaMemcpy((void **) &A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy((void **) &B_d, B, size, cudaMemcpyHostToDevice);
    cudaMemcpy((void **) &C_d, C, size, cudaMemcpyHostToDevice);

2. // Kernel invocation code – to be shown later

3. // Transfer C from device to host
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
Check for API Errors in Host Code

cudaError_t err = cudaMalloc((void**)&d_A,size);

if (err!=cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
}
```
Example: Vector Addition Kernel

// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddkernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256),256>>>(A_d, B_d, C_d, n);
}
More on Kernel Launch

```c
int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(n/256, 1, 1);  dim3 DimGrid((n-1)/256 + 1, 1, 1);
    if (n%256) DimGrid.x++;
    dim3 DimBlock(256, 1, 1);

    vecAddKernel<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking
Kernel execution in a nutshell

```c
__host__
Void vecAdd()
{
    dim3 DimGrid = (ceil(n/256,1,1);
    dim3 DimBlock = (256,1,1);
    vecAddKernel<<<DimGrid,DimBlock>>>(A_d,B_d,C_d,n);
}

__global__
void vecAddKernel(float *A_d, float *B_d, float *C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```

Schedule onto multiprocessors
More on CUDA Function Declarations

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- **__global__** defines a kernel function
- Each “__” consists of two underscore characters
- A kernel function must return **void**
- **__device__** and **__host__** can be used together
Compiling A CUDA Program

Integrated C programs with CUDA extensions

NVCC Compiler

Host Code

Host C Compiler/Linker

Device Code (PTX)

Device Just-in-Time Compiler

Heterogeneous Computing Platform with CPUs, GPUs
QUESTIONS?