Introduction to GPU Parallel Programming

Data Heroes Summer HPC Workshop

June 27, 2016

Donald Frederick,
Livermore Computing
Agenda

- Why Accelerators, GPUs
- GPU architecture
- GPU Programming Models
  - CUDA
  - OpenCL
  - OpenMP
  - OpenACC
Agenda (continued)

- Focus on CUDA GPU Programming Model
- Simple CUDA Examples
- device_query
- Vector sum
- Compute Pi
Agenda (continued)

- LC Surface Linux Cluster Architecture
- LC Surface Linux Cluster node architecture
- How to compile, link and run simple CUDA examples on Surface
- References for further study
Objective

- Introduction to programming GPUs for general-purpose computing tasks
- Specific to NVIDIA GPUs
  - CUDA programming abstraction
- Compare performance to CPU threads
  - OpenMP – could also integrate (not shown)
- Show integration within clusters
  - Multiple nodes with GPUs
- Hands-on exercises - : laptop → LC surface Linux cluster

Not covered in this short course:

- Advanced GPU performance tuning (memory, async. kernels etc.)
- OpenCL
- PGI compiler directives for accelerators
Why GPU Computing
Recap of Important Parallel Computing Concepts

Single Instruction, Multiple Data (SIMD):

A type of parallel computer

**Single Instruction:** All processing units execute the same instruction at any given clock cycle

**Multiple Data:** Each processing unit can operate on a different data element

Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.

Synchronous (lockstep) and deterministic execution

Two varieties: Processor Arrays and Vector Pipelines

Examples:

Processor Arrays: Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV
Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10

Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.
Recap of Important Parallel Computing Concepts (continued)

Single Program Multiple Data (SPMD):

SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

SINGLE PROGRAM: All tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data parallel or hybrid.

MULTIPLE DATA: All tasks may use different data

SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.

The SPMD model, using message passing or hybrid programming, is probably the most commonly used parallel programming model for multi-node clusters.
Recap of Important Parallel Computing Concepts (continued)

Current Architectures
Recap of Important Parallel Computing Concepts (continued)

Designing Parallel Programs

Understand the Problem and the Program

Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

• Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

Partitioning

One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.

There are two basic ways to partition computational work among parallel tasks: domain decomposition and functional decomposition.

Domain Decomposition:

In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
Recap of Important Parallel Computing Concepts (continued)

Domain Decomposition:

In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
Add GPUs: Accelerate Science Applications
Small Changes, Big Speed-up

Application Code

Rest of Sequential CPU Code

GPU

Compute-Intensive Functions

Use GPU to Parallelize

CPU

© NVIDIA 2013
Fastest Performance on Scientific Applications

Tesla K20X Speed-Up over Sandy Bridge CPUs

**Engineering**  
MATLAB (FFT)*

**Physics**  
Chroma

**Earth Science**  
SPECFEM3D

**Molecular Dynamics**  
AMBER

---

CPU results: Dual socket E5-2687w, 3.10 GHz, GPU results: Dual socket E5-2687w + 2 Tesla K20X GPUs

*MATLAB results comparing one i7-2600K CPU vs with Tesla K20 GPU

Disclaimer: Non-NVIDIA implementations may not have been fully optimized

© NVIDIA 2013
Why Computing Perf/Watt Matters?

Traditional CPUs are not economically feasible

Era of GPU-accelerated computing is here

2.3 PFlops

7000 homes

7.0 Megawatts

7.0 Megawatts

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Many Parallel Tasks

10x performance/socket
> 5x energy efficiency

© NVIDIA 2013
World's Fastest, Most Energy Efficient Accelerator

Tesla K20X vs Xeon CPU
- 8x Faster SGEMM
- 6x Faster DGEMM

Tesla K20X vs Xeon Phi
- 90% Faster SGEMM
- 60% Faster DGEMM

© NVIDIA 2013
GPU Programming Models (APIs)

- **CUDA**
  - Developed by NVIDIA
  - Specific to NVIDIA hardware
  - Supports C/C++
    - Adds new syntactic elements to language
  - Often provides faster execution than other APIs
  - NVIDIA and 3-rdparty support for numerical libraries, infrastructure
    - cuBLAS
    - cuFFT
GPU Programming Models (APIs)

- OpenCL
  - Cross-platform, cross-vendor standard
  - Enables programming of diverse compute resources
    - CPU, GPU, DSP, FPGA
  - One code tree can be executed on CPUs, GPUs, DSPs and hardware
    - Dynamically interrogate system load and balance across available processors
  - Supports C/C++
  - More information: https://www.khronos.org/opencl/
GPU Programming Models (APIs)

- **OpenMP**
  - Multi-platform, shared-memory
  - Supports C/C++ and Fortran
    - Compiler-directive-based
  - Supported by many vendors’ compilers
  - Accelerator support defined in OpenMP 4.0
  - More information: [http://openmp.org](http://openmp.org)
    - OpenMP 4.5 latest (Fall 2015)
GPU Programming Models (APIs)

- **OpenACC**
  - Vendor-neutral API
  - Supports C/C++ and Fortran (similar to OpenMP)
    - Compiler-directive-based
CUDA C/C++ BASICS

NVIDIA Corporation
What is CUDA?

- **CUDA Architecture**
  - Expose GPU parallelism for general-purpose computing
  - Retain performance

- **CUDA C/C++**
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

- This session introduces CUDA C/C++
Introduction to CUDA C/C++

What will you learn in this session?

- Start from “Hello World!”
- Write and launch CUDA C/C++ kernels
- Manage GPU memory
- Manage communication and synchronization
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
HELLO WORLD!
Heterogeneous Computing

- **Terminology:**
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)
Heterogeneous Computing

serial code

parallel fn

parallel code

serial code
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$
```
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- Two new syntactic elements...
Hello World! with Device Code

```c
__global__ void mykernel(void) {

}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`
Hello World! with Device COde

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

- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- `mykernel()` does nothing, somewhat anticlimactic!

Output:

```
$ nvcc hello.cu
hello.cu
$ a.out
Hello World!
$

© NVIDIA 2013
Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!

- We need a more interesting example...

- We’ll start by adding two integers and build up to vector addition
Addition on the Device

- A simple kernel to add two integers

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

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host
Addition on the Device

- Note that we use pointers for the variables

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

- `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

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
Addition on the Device: add()

- Returning to our add() kernel

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

- Let’s take a look at main()…
Addition on the Device: `main()`

```c
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 = sizeof(int);

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

    // Setup input values
    a = 2;
    b = 7;
}
```
Addition on the Device: `main()`

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

// Launch add() kernel on GPU
add<<<1,1>>>(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;
```
RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

© NVIDIA 2013
Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```c
add<<< 1, 1 >>>();

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add&lt;&lt;&lt; N, 1 &gt;&gt;&gt;();</td>
</tr>
</tbody>
</table>
```

- Instead of executing `add()` once, execute N times in parallel
Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
  - The set of blocks is referred to as a **grid**
  - Each invocation can refer to its block index using `blockIdx.x`

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

- By using `blockIdx.x` to index into the array, each block handles a different index
Vector Addition on the Device

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

- On the device, each block can execute in parallel:

  ```c
  Block 0
  c[0] = a[0] + b[0];
  ```

  ```c
  Block 1
  c[1] = a[1] + b[1];
  ```

  ```c
  Block 2
  ```

  ```c
  Block 3
  ```
Vector Addition on the Device: \texttt{add()} 

- Returning to our parallelized \texttt{add()} kernel

\[
\begin{align*}
\texttt{\_\_global\_ void add(int \*a, int \*b, int \*c) \{} \\
&\quad c[\text{blockIdx.x}] = a[\text{blockIdx.x}] + b[\text{blockIdx.x}]; \\
&\quad \}\n\end{align*}
\]

- Let’s take a look at \texttt{main()}...
Vector Addition on the Device: `main()`

```c
#define N 512

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
    cudaMemcpy((void **)&d_a, size);
    cudaMemcpy((void **)&d_b, size);
    cudaMemcpy((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 on the Device: `main()`

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

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

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

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
Review (1 of 2)

- **Difference between host and device**
  - *Host* CPU
  - *Device* GPU

- Using **__global__** to declare a function as device code
  - Executes on the device
  - Called from the host

- Passing parameters from host code to a device function
Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - Launch $N$ copies of `add()` with `add<<<N,1>>>(...)`;
  - Use `blockIdx.x` to access block index
INTRODUCING THREADS

CONCEPTS

Heterogeneous Computing
Blocks
Threads
Indexing
Shared memory
__syncthreads()
Asynchronous operation
Handling errors
Managing devices
CUDA Threads

- Terminology: a block can be split into parallel threads
- Let’s change `add()` to use parallel threads instead of parallel blocks

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

- We use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in `main()`...
Vector Addition Using Threads:

main()

#define N 512

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 Using Threads:

main()

    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);
    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;

}
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Combining Blocks and Threads

- We’ve seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads

- Let’s adapt vector addition to use both blocks and threads

- Why? We’ll come to that…

- First let’s discuss data indexing…
Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

```
threadIdx.x    threadIdx.x    threadIdx.x    threadIdx.x
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
```

- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```
Indexing Arrays: Example

- Which thread will operate on the red element?

```
int index = threadIdx.x + blockIdx.x * M;
= 5 + 2 * 8;
= 21;
```
Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block
  
  ```c
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  ```

- Combined version of `add()` to use parallel threads and parallel blocks
  
  ```c
  __global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
  }
  ```

- What changes need to be made in `main()`?
Addition with Blocks and Threads:

main()

```c
#define N (2048*2048)
#define THREADS_PER_BLOCK 512

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);
```
Addition with Blocks and Threads:

```c
main()
{
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

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

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```
Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

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

- Update the kernel launch:

  ```
  add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
  ```
Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

- To look closer, we need a new example…
Review

- Launching parallel kernels
  - Launch $N$ copies of `add()` with `add<<<N/M,M>>>(...);`
  - Use `blockIdx.x` to access block index
  - Use `threadIdx.x` to access thread index within block

- Allocate elements to threads:

```c
int index = threadIdx.x + blockIdx.x * blockDim.x;
```
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

- If radius is 3, then each output element is the sum of 7 input elements:
Implementing Within a Block

- Each thread processes one output element
  - blockDim.x elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times
Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory

- Extremely fast on-chip memory, user-managed

- Declare using __shared__, allocated per block

- Data is not visible to threads in other blocks
Implementing With Shared Memory

- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory

- Each block needs a halo of radius elements at each boundary
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] =
        in[gindex + BLOCK_SIZE];
    }
}
Stencil Kernel

// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
Data Race!

- The stencil example will not work...

- Suppose thread 15 reads the halo before thread 0 has fetched it...

```c
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS] = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]  ❑❑❑❑❑❑❑❑❑❑❑❑❑❑❑

Skipped, threadIdx > RADIUS

Load from temp[19]  ❑❑❑❑❑❑❑❑❑❑❑❑❑❑❑

© NVIDIA 2013
\texttt{\_syncthreads()}

- \texttt{void \_syncthreads();}

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block
Stencil Kernel

```c
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
```
Stencil Kernel

// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
Review (1 of 2)

- Launching parallel threads
  - Launch $N$ blocks with $M$ threads per block with
    \[ \text{kernel} \lll N, M \rrr (...) \];
  - Use \text{blockIdx.x} to access block index within grid
  - Use \text{threadIdx.x} to access thread index within block

- Allocate elements to threads:

  \[
  \text{int} \text{ index} = \text{threadIdx.x} + \text{blockIdx.x} \times \text{blockDim.x};
  \]
Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards
MANAGING THE DEVICE

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Coordinating Host & Device

- Kernel launches are **asynchronous**
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

```plaintext
cudaMemcpy()                      Blocks the CPU until the copy is complete
Copy begins when all preceding CUDA calls have completed

cudaMemcpyAsync()                 Asynchronous, does not block the CPU

cudaDeviceSynchronize()           Blocks the CPU until all preceding CUDA calls have completed
```
Reporting Errors

- All CUDA API calls return an error code (\texttt{cudaError_t})
  - Error in the API call itself
    OR
  - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:
  \begin{verbatim}
  cudaError_t cudaGetLastError(void)
  \end{verbatim}

- Get a string to describe the error:
  \begin{verbatim}
  char *cudaGetErrorString(cudaError_t)
  \end{verbatim}

  \begin{verbatim}
  printf("%s\n",
         cudaGetErrorString(cudaGetLastError()));
  \end{verbatim}
Device Management

- Application can query and select GPUs
  
  ```
  cudaGetDeviceCount(int *count)
  cudaSetDevice(int device)
  cudaGetDevice(int *device)
  cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
  ```

- Multiple threads can share a device

- A single thread can manage multiple devices
  
  ```
  cudaSetDevice(i) to select current device
  cudaMemcpy(...) for peer-to-peer copies
  ```

† requires OS and device support
Introduction to CUDA C/C++

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - __global__, blockIdx.x, threadIdx.x, <<<>>>
  - Manage GPU memory
    - cudaMalloc(), cudaMemcpy(), cudaFree()
  - Manage communication and synchronization
    - __shared__, __syncthreads()
Compute Capability

- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>Selected Features</th>
<th>Tesla models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Fundamental CUDA support</td>
<td>870</td>
</tr>
<tr>
<td>1.3</td>
<td>Double precision, improved memory accesses, atomics</td>
<td>10-series</td>
</tr>
<tr>
<td>2.0</td>
<td>Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion</td>
<td>20-series</td>
</tr>
</tbody>
</table>
# Compute Capability

The **compute capability** of a device describes its architecture, e.g.

- Number of registers
- Sizes of memories
- Features & capabilities

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>Selected Features (see CUDA C Programming Guide for complete list)</th>
<th>Tesla models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Fundamental CUDA support</td>
<td>870</td>
</tr>
<tr>
<td>1.3</td>
<td>Double precision, improved memory accesses, atomics</td>
<td>10-series</td>
</tr>
<tr>
<td>2.0</td>
<td>Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P,</td>
<td>20-series</td>
</tr>
<tr>
<td></td>
<td>concurrent kernels/copies, function pointers, recursion</td>
<td></td>
</tr>
</tbody>
</table>

© NVIDIA 2013
### Compute Capability (continued)

- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>Selected Features</th>
<th>Tesla models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Fundamental CUDA support</td>
<td>870</td>
</tr>
<tr>
<td>1.3</td>
<td>Double precision, improved memory accesses, atomics</td>
<td>10-series</td>
</tr>
<tr>
<td>2.0</td>
<td>Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion</td>
<td>20-series</td>
</tr>
</tbody>
</table>

© NVIDIA 2013
IDs and Dimensions

- A kernel is launched as a grid of blocks of threads
  - blockIdx and threadIdx are 3D
  - We showed only one dimension (x)

- Built-in variables:
  - threadIdx
  - blockIdx
  - blockDim
  - gridDim
Textures

- Read-only object
  - Dedicated cache

- Dedicated filtering hardware
  (Linear, bilinear, trilinear)

- Addressable as 1D, 2D or 3D

- Out-of-bounds address handling
  (Wrap, clamp)
Topics we skipped

- We skipped some details, you can learn more:
  - CUDA Programming Guide
  - CUDA Zone – tools, training, webinars and more
    developer.nvidia.com/cuda

- Need a quick primer for later:
  - Multi-dimensional indexing
  - Textures


**LC Surface Linux Cluster Architecture**

- **LC Linux Cluster – CPUs + GPUs**
  - 158 compute (batch) nodes, 1 login node
  - 2 Sockets, 8 cores each, Intel Xeon E5-2670 @ 2.6 GHz; 2 GPUs/node – Tesla K40m

- **Workshop will be using 20 reserved nodes**
  - XXX
  - XXX
LC Surface Linux Cluster Architecture
Hands-on Exercises: Computing $\pi$ (Pi)

- Running Example: compute Pi with increasing parallelism
- Description of Numerical approach

1. C

2. C + CUDA
   1. Simple kernel (1 thread)
   2. 1 Block (512 threads)
   3. Shared memory
   4. Grid of blocks (32k threads)
   5. GPU reduction
Login on LC SurfaceCluster

- On Linux/OS X/Windows systems, use:
  - Putty / any ssh client
  - Connect to surface.llnl.gov
  - `ssh -X -l username surface.llnl.gov`

- Copy exercise file into $HOME directory
  - `mkdir GPU`
  - `cp /usr/global/docs/training/blaise/gpu/C/* ~/GPU`
  - `cd GPU`
Getting Information on GPU Environment

- In terminal session
  deviceQuery
  • What is output?
  • Now try mxterm 1 1 90 –q gpgpu, then when new X-window opens -
    deviceQuery

- Examine output – what information is provided?
Setting up User Software Environment

- Set up CUDA environment
  - module load cudatoolkit/7.5

- Set up Compiler environment
  - use gcc-4.4.6
Approximation of Pi by Monte Carlo

- The value of Pi can be calculated in a number of ways. Consider the following method of approximating Pi. Inscribed a circle in a square.
- Randomly generate points in the square.
- Determine the number of points in the square that are also in the circle.
- Let \( r \) be the number of points in the circle divided by the number of points in the square.
- \( \pi \sim 4r \)
- Note that the more points generated, the better the approximation.

\[
\begin{align*}
A_S &= (2r)^2 = 4r^2 \\
A_C &= \pi r^2 \\
\pi &= 4 \times \frac{A_C}{A_S}
\end{align*}
\]
Approximation of Pi - Monte Carlo

- Serial pseudo code for this procedure:

```
npoints = 10000 circle_count = 0

do j = 1,npoints
    generate 2 random numbers between 0 and 1
    xcoordinate = random1
    ycoordinate = random2
    If (xcoordinate, ycoordinate) inside circle then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```
Approximation of Pi by Monte Carlo – Parallel Version

- Another problem that's easy to parallelize: All point calculations are independent; no data dependencies
- Work can be evenly divided; no load balance concerns
- No need for communication or synchronization between tasks
- Parallel strategy: Divide the loop into equal portions that can be executed by the pool of tasks
- Each task independently performs its work
- A SPMD model is used
- One task acts as the master to collect results and compute the value of PI
Approximation of Pi by Monte Carlo – Parallel Version

- Pseudo code solution: red highlights changes for parallelism.

\[
\text{npoints} = \text{numthreads} \\
\text{circle\_count} = 0 \\
p = \text{number of tasks} \\
\text{num} = \frac{\text{npoints}}{p} \\
do \text{j} = 1, \text{num} \\
generate 2 random numbers between 0 and 1 \\
xcoordinate = \text{random1} \quad [\text{for each thread}] \\
ycoordinate = \text{random2} \quad [\text{for each thread}] \\
if (xcoordinate, ycoordinate) inside circle then circle\_count = circle\_count + 1 \quad [\text{for each thread}] \\
end do
Make Examples

- VectorAdd
  
  make vectorAdd

- Compute Pi – pure C
  
  • make pi-serial

- Compute Pi – C, CUDA
  
  • make cuda-pi
Running Examples

- Vector Add
  - ./vectorAdd

- Compute Pi – pure C
  - ./pi-serial

- Compute Pi – CUDA
  - ./cuda-pi
References for Additional Reading

- NVIDIA CUDA Developers Blog
  https://devblogs.nvidia.com/parallelforall/
- Oak Ridge Leadership Computing facility (OLCF) -
- “Programming Massively Parallel Processors”, by Kirk and Hwu
Acknowledgements

- Steven Rennich, NIDIA
- NVIDIA Corporation
- Oak Ridge National laboratory Leadership Computing facility (LCF)