Massively Parallel Computing with CUDA

Carlos Alberto Martínez Angeles
Cinvestav-IPN
What is a GPU?

• A graphics processing unit (GPU)
  • specialized electronic circuit
  • designed to rapidly manipulate and alter memory
  • accelerates the building of images in a frame buffer

• The term GPU was popularized by Nvidia in 1999
  • marketed the GeForce 256 as "the world's first 'GPU'

• There are 3 GPU forms
  • Dedicated: have RAM that is dedicated to the card's use, does not mean that it is removable
  • Integrated: use the computer's RAM. They are integrated into the motherboard or into the CPU die.
  • Hybrid: use both types of memory.

• GPU companies
  • Nvidia
  • ATI
Why Use the GPU?

• The GPU has evolved into a very flexible and powerful processor:
  • It’s programmable using high-level languages
  • It supports 32-bit and 64-bit floating point IEEE-754 precision
  • It offers lots of GFLOPS

• GPU in every PC and workstation
Figure 1  Floating-Point Operations per Second for the CPU and GPU
Figure 2  Memory Bandwidth for the CPU and GPU
What is behind such an Evolution?

- The GPU is specialized for compute-intensive, highly parallel computation (exactly what graphics rendering is about)

- So, more transistors can be devoted to data processing rather than data caching and flow control

- The fast-growing video game industry exerts strong economic pressure that forces constant innovation
• Each NVIDIA GPU has **thousands** of parallel cores
• Within each core
  • Floating point unit
  • Logic unit (add, sub, mul, madd)
  • Move, compare unit
  • Branch unit
• Cores managed by thread manager
  • Thread manager can spawn and manage 12,000+ threads per core
  • Zero overhead thread switching
CUDA is C for Parallel Processors

- CUDA is industry-standard C with minimal extensions
  - Write a program for one thread
  - Instantiate it on many parallel threads
  - Familiar programming model and language

- CUDA is a scalable parallel programming model
  - Program runs on any number of processors without recompling
CUDA Parallel Computing Architecture

- Includes a C compiler plus support for OpenCL and DX11 Compute
- Architected to natively support all computational interfaces (standard languages and APIs)
- NVIDIA GPU architecture accelerates CUDA
  - Hardware and software designed together for computing
Application Software
(written in C)

CUDA Libraries
- cuFFT
- cuBLAS
- cuDPP

CPU Hardware
- 1U
- PCI-E Switch

CUDA Compiler
- C
- Fortran

CUDA Tools
- Debugger
- Profiler

4 cores

thousands of cores
Massively parallel computing has become a commodity technology!
CUDA Computing with Fermi

- 512 cuda cores: 1.581 TFLOPS peak
- 67,108,864 threads per processor: 34,359,738,368 threads total
- Fermi PCI-e board: 580 Gtx
Compute Capability

- What the GPU can do
- Is defined by a major revision number and a minor revision number
- Devices with the same major revision number are of the same core architecture
  - 3 for devices based on the Kepler architecture
  - 2 for devices based on the Fermi architecture
  - 1 for devices based on the Tesla architecture
- The minor revision number corresponds to an incremental improvement to the core architecture
  - possibly including new features

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
<th>3.0</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of instructions per kernel</td>
<td>2 million</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>512 million</td>
<td></td>
</tr>
</tbody>
</table>
CUDA Uses Extensive Multithreading

- **CUDA threads** express fine-grained data parallelism
  - Map threads to GPU threads
  - Virtualize the processors
  - You must rethink your algorithms to be aggressively parallel

- **CUDA thread blocks** express coarse-grained parallelism
  - Blocks hold arrays of GPU threads, define shared memory boundaries
  - Allow scaling between smaller and larger GPUs

- GPUs execute thousands of lightweight threads
  - (In graphics, each thread computes one pixel)
  - One CUDA thread computes one result (or several results)
  - Hardware multithreading & zero-overhead scheduling
CUDA Computing Sweet Spots

Parallel Applications

- **High bandwidth:**
  - Sequencing (virus scanning, genomics), sorting, database, …

- **Visual computing:**
  - Graphics, image processing, tomography, machine vision, …

- **High arithmetic intensity:**
  - Dense linear algebra, PDEs, n-body, finite difference, …
A Highly Multithreaded Coprocessor

- The GPU is a highly parallel compute coprocessor
  - serves as a coprocessor for the host CPU
  - has its own device memory with high bandwidth interconnect
- The application run its parallel parts on GPU, via kernels.
  - Many threads execute same kernel
  - SIMT = Single Instruction Multiple Threads
  - GPU Threads are extremely lightweight
    - Very little creation overhead,
    - Instant switching
  - GPU uses 1000s of threads for efficiency
Heterogeneous Programming

- CUDA application = serial program executing parallel kernels, all in C
  - Serial C code executed by a CPU thread
  - Parallel kernel C code executed by GPU, in threads (grouped in blocks)
__global__ void function(arguments)
{
    //Each thread will execute this code
}

int main()
{
    ...
    dim3 blocks(A, B, C);
    dim3 threads(X, Y, Z);
    function<<<blocks, threads>>>(arguments);
    ...
}
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
- All threads run the same program, SIMT (Single Instruction multiple threads)
- Each thread uses its ID to compute addresses and make control decisions

```plaintext
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...```
CUDA Programming Model

- A kernel is executed by a **grid**, which contain **blocks**.

- These blocks contain our **threads**.

- A **thread block** is a batch of threads that can cooperate:
  - Sharing data through shared memory
  - Synchronizing their execution

- Threads from different blocks operate independently
Thread IDs

- To know its ID within his block
  - `threadIdx.x`
  - `threadIdx.y`
  - `threadIdx.z`

- To know the ID of the block
  - `blockIdx.x`
  - `blockIdx.y`
  - `blockIdx.z`

- To know the size of the grid
  - `blockDim.x`
  - `blockDim.y`
  - `blockDim.z`

Common IDs

- `id = threadIdx.x`
- `id = threadIdx.x + blockIdx.x * blockDim.x`
Example

Grid

Blocks

Threads
Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory**
  - Threads in different blocks cannot cooperate
- Enables programs to **transparently scale** to any number of processors!
Transparent Scalability

- Hardware is free to schedule thread blocks on any streaming multiprocessor
- Kernels scale to any number of these multiprocessors
Thread Cooperation

- Thread cooperation is a powerful feature of CUDA
  - Threads can cooperate via on-chip shared memory and synchronization
- The on-chip shared memory within one block allows:
  - Share memory accesses, drastic memory bandwidth reduction
  - Share intermediate results, thus: save computation
- Makes algorithm porting to GPUs a lot easier
Thread Synchronization

`__syncthreads()`

- Acts as a **barrier**
  - threads idle when they reach it
  - they continue when all threads have reached it
- Only works for threads in the **same block**
- It is lightweight
- Should be avoided if possible

How can I synchronize threads in different blocks?
Memory model seen from CUDA Kernel

- Registers (per thread)
- Shared Memory
  - Shared among threads in a single block
  - On-chip, small
  - As fast as registers
- Global Memory
  - Kernel inputs and outputs reside here
  - Off-chip, large
  - Cached (use coalescing)

Note: The host can read & write global memory but not shared memory
Registers

- Stack and variables are stored here
- Fastest type of memory in the GPU
- Automatically assigned
- Limit the number of threads per block
  - if the needed number of registers is exceeded (register spilling), global memory is used
  - slows down the application

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
<th>3.0</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
<td>64 K</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shared Memory

__shared__ type name [optional_size][optional_size]

- As fast as registers
- Race conditions are possible
  - use '__syncthreads()'
- Sometimes limits the number of threads per block
  - global memory is not used
  - the application simply fails

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
<th>3.0</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16 KB</td>
<td>48 KB</td>
</tr>
</tbody>
</table>
Global Memory

- Slowest memory available
  - still faster than RAM

- Large but depends on the card
  - usually between 256MB and 4GB
  - screen and other programs can reduce this number
  - no virtual memory, application fails if you exceed it

- Race conditions are possible
  - use the thread's id to avoid them
  - atomic operations are also an option

- CPU can copy to and from this memory
  - very slow

- No double pointers
  - matrices must be 'flattened' to vectors using offsets

- Access to this memory in the kernel should always be coalesced
Coalesced Memory Access

Coalesced

Semi-coalesced

Non-coalesced
Global Memory Instructions

- To declare global memory
  - `type* name;`
  - `cudaMalloc(&name, size);`

- To copy memory in different locations
  - From the CPU to the GPU
    - `cudaMemcpyp(destination, source, size, cudaMemcpyHostToDevice);`
  - From the GPU to the CPU
    - `cudaMemcpyp(destination, source, size, cudaMemcpyDeviceToHost);`
  - From the GPU to the CPU
    - `cudaMemcpyp(destination, source, size, cudaMemcpyDeviceToDevice);`

- To free memory
  - `cudaFree(name);`
Atomic Functions

- Perform a read-modify-write atomic operation
  - on one 32-bit or 64-bit word
  - in global or shared memory.
- It is guaranteed to be performed without interference from other threads
- Can slow down the entire application
- The only way of cooperation between threads of different blocks

```c
int atomicAdd(int* address, int val);
```

- Reads the word 'old' located at the address 'address', computes (old + val), and stores the result back to memory at the same address.
- These three operations are performed in one atomic transaction.
- The function returns 'old'.
Events

- Closely monitor the device’s progress
- Perform accurate timing
- Asynchronously record events at any point in the program
- An event has completed when all tasks preceding it have completed

```c
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);
    // Code to monitor
cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```
Error Checking

- All runtime functions return an error code
- The runtime maintains an error variable
  - initialized to cudaSuccess
  - overwritten by the error code every time an error occurs
- To check for asynchronous errors
  - synchronize just after the call

```c
cudaDeviceSynchronize();
cudaError_t error;
error = cudaGetLastError();
```

- cudaErrorMemoryAllocation
  - unable to allocate enough memory to perform the requested operation
- cudaErrorInitializationError
  - CUDA driver and runtime could not be initialized
- cudaErrorLaunchFailure
  - an exception occurred on the device while executing a kernel
    - dereferencing an invalid device pointer
    - accessing out of bounds shared memory.
Libraries

- **NVIDIA cuRAND**
  - The CUDA Random Number Generation library performs high quality GPU-accelerated random number generation (RNG) over 8x faster than typical CPU only code.

- **CUFFT**
  - The NVIDIA CUDA Fast Fourier Transform library (cuFFT) provides a simple interface for computing FFTs up to 10x faster.

- **NVIDIA cuSPARSE**
  - NVIDIA CUDA Sparse (cuSPARSE) Matrix library provides a collection of basic linear algebra subroutines used for sparse matrices that delivers over 8x performance boost.

- **GPU AI – Path Finding**
  - Libraries and samples applications demonstrating CUDA-accelerated path planning suitable for domains such as robotics, video games, synthetic environments and AI.

- **NVIDIA CUDA Math Library**
  - An industry proven, highly accurate collection of standard mathematical functions, providing high performance on NVIDIA GPUs.
Code Optimization

- Maximize **Utilization**
  - Use blocks of multiples of 32
  - Use at least as many blocks as streaming multiprocessors
  - Prefer more threads per block than more blocks

- Maximize **Memory Throughput**
  - Use coalesced access
  - Reduce exchange between CPU and GPU
  - Use shared memory and registers when possible

- Maximize **Instruction Throughput**
  - Try not to use control flow instructions (if, while, switch, ...)
  - Unless necessary, prefer single-precision
Installation on Linux

• Download the latest distribution
  • https://developer.nvidia.com/cuda-downloads

• Install it
  • If the included drivers do not work, use the ones for your Linux distribution or the ones from Nvidia's web page

• Add /usr/local/cuda-5.0/bin to PATH, /usr/local/cuda-5.0/lib and /usr/local/cuda-5.0/lib64 to LD_LIBRARY_PATH
  • Use the export command
  • To make it permanent place them in ~/.bash_profile

• Usually, the latest version of gcc and g++ is not compatible with CUDA
  • You must install an older version and link its binaries to the CUDA binaries directory
Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc

- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...

- NVCC can output:
  - Either an executable file
  - Or CUDA object code
    - That must then be compiled with the rest of the application using another tool
    - Or PTX object code directly

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (cudart)
  - The CUDA core library (cuda)
Compiling C for CUDA Applications

C CUDA Key Kernels

NVCC

CUDA object files

Rest of C Application

CPU Code

CPU object files

Linker

CPU-GPU Executable
Keys to GPU Computing Performance

● Hardware Thread Management
  ● Thousands of lightweight concurrent threads
  ● No switching overhead
  ● Hide instruction and memory latency

● On-Chip Shared Memory
  ● User-managed data cache
  ● Thread communication / cooperation within blocks

● Random access to global memory
  ● Any thread can read/write any location(s)
  ● Direct host access
Other CUDA Stuff

- Other 'types' of memory
  - Mapped memory
  - Texture memory
  - Surface memory

- Streams

- Driver API

- Direct X

- New functions for devices of compute capability 3.5
References

- CUDA C Programming Guide

- CUDA Getting Started Guide for Linux

- CUDA C Best Practices Guide

- CUDA Compiler Driver NVCC

- CUDA Toolkit v5.0 Release Notes