Introduction

What is CUDA?
- Compute Unified Device Architecture.
  - A powerful parallel programming model for issuing and managing computations on the GPU without mapping them to a graphics API.
  - Heterogenous - mixed serial-parallel programming
  - Scalable - hierarchical thread execution model
  - Accessible - minimal but expressive changes to C

Software Stack:
- Libraries:
  CUFFT & CUBLAS
- Runtime:
  Common component
  Device component
  Host component
- Driver:
  Driver API

CUDA SDK

Libraries:
- FFT, BLAS...
- Example Source Code
- Integrated CPU and GPU C Source Code
- NVIDIA C-Compiler
- CUDA Driver
- Debugger/Profiler
- Standard C Compiler
- GPU Host Code
- GPU
Motivation

GPU Programming Model
GPGPU Programming Model
CUDA Programming Model

Trick the GPU into general-purpose computing by casting problem as graphics

- Turn data into images ("texture maps")
- Turn algorithms into image synthesis ("rendering passes")
- Tough learning curve
- Potentially high overhead of graphics API
- Highly constrained memory layout & access model
- Need for many passes drives up bandwidth consumption
Motivation
GPGPU Programming to do A + B

Motivation
What’s wrong with GPGPU 1

Motivation
What’s wrong with GPGPU 2

Motivation
What’s wrong with GPGPU 2

Outline
• Introduction
• Motivation
• Programming Model
  • Memory Model
  • CUDA API
  • Example
  • Pro & Contra
  • Trend

Programming Model
CUDA: Unified Design
Advantages:
HW:
• fully generally data-parallel architecture
  • General thread launch
  • Global load-store
  • Parallel data cache
  • Scalar architecture
  • Integers, bit operation
SW:
• program the GPU in C
  • Scalable data parallel execution/ memory model
  • C with minimal yet powerful extensions
Motivation
From GPGPU to CUDA Programming Model

Programming Model
Feature 1:
- Thread not pixel
- Full Integer and Bit Instructions
- No limits on branching, looping
- 1D, 2D, 3D threadID allocation

Feature 2:
- Fully general load/store to GPU memory
- Untyped, not fixed texture types
- Pointer support

Feature 3:
- Dedicated on-chip memory
- Shared between threads for inter-thread communication
- Explicitly managed
- As fast as registers

Important Concepts:
- Device: GPU, viewed as a co-processor.
- Host: CPU
- Kernel: data-parallel, computed-intensive position of application running on the device.

Programming Model
Simple example (Matrix addition):

cpu c program:  
cuda program:

Hardware implementation:
A set of SIMD Multiprocessors with On-Chip shared memory.
Programming Model

G80 Example:
• 16 Multiprocessors, 128 Thread Processors
• Up to 12,288 parallel threads active
• Per-block shared memory accelerates processing.

Streaming Multiprocessor (SM)
• Processing elements
  o 8 scalar thread processors
  o 32 GFLOPS peak at 1.35GHz
  o 8192 32-bit registers (32KB)
  o usual ops: float, int, branch...

• Hardware multithreading
  o up to 8 blocks (3 active) residents at once
  o up to 768 active threads in total

• 16KB on-chip memory
  o supports thread communication
  o shared amongst threads of a block

Programming Model

Execution Model:

Single Instruction Multiple Thread (SIMT) Execution:
• Groups of 32 threads formed into warps
  o always executing same instruction
  o share instruction fetch/dispatch
  o some become inactive when code path diverges
  o hardware automatically handles divergence

• Warps are primitive unit of scheduling
  o pick 1 of 24 warps for each instruction slot.
  o all warps from all active blocks are time-sliced

Outline

• Introduction
• Motivation
• Programming Model
• Memory Model
• CUDA API
• Example
• Pro & Contra
• Trend

Memory Model

There are 6 Memory Types:
There are 6 Memory Types:

- **Registers**
  - on chip
  - fast access
  - per thread
  - limited amount
  - 32 bit

- **Local Memory**
  - in DRAM
  - slow
  - non-cached
  - per thread
  - relative large

- **Shared Memory**
  - on chip
  - fast access
  - per block
  - 16 KByte
  - synchronize between threads

- **Global Memory**
  - in DRAM
  - slow
  - non-cached
  - per grid
  - communicate between grids

- **Constant Memory**
  - in DRAM
  - cached
  - per grid
  - read-only

- **Texture Memory**
  - in DRAM
  - cached
  - per grid
  - read-only
Memory Model

- Registers
- Shared Memory
  - on chip
- Local Memory
- Global Memory
- Constant Memory
- Texture Memory
  - in Device Memory

Outline

- Introduction
- Motivation
- Programming Model
- Memory Model
  - CUDA API
  - Example
  - Pro & Contra
  - Trend

CUDA API

CUDA API provides a easy path for users to write programs for GPU device.

It consists of:

- A minimal set of extensions to C/C++
  - type qualifiers
  - call-syntax
  - built-in variables
- A runtime library to support the execution
  - host component
  - device component
  - common component

CUDA C/C++ Extensions:

- New function type qualifiers
  - __host__ void HostFunc(...); //executable on host
  - __global__ void KernelFunc(...); //callable from host
  - __device__ void DeviceFunc(...); //callable from device only

- Restrictions for device code (__global__ | __device__)
  - no recursive call
  - no static variable
  - no function pointer
  - __global__ function is asynchronous invoked
  - __global__ function must have void return type

CUDA C/C++ Extensions:

- New variable type qualifiers
  - __device__ int GlobalVar; //in global memory, lifetime of app
  - __const__ int ConstVar; //in constant memory, lifetime of app
  - __shared__ int SharedVar; //in shared memory, lifetime of blocks

- Restrictions
  - no external usage
  - only file scope
  - no combination with struct or union
  - no initialization for __shared__
CUDA API

CUDA C/C++ Extensions:
- New syntax to invoke the device code
  \[ \text{KernelFunc}<<<Dg, Db, Ns, S>>>(...); \]
  - \(Dg\): dimension of grid
  - \(Db\): dimension of block
  - \(Ns\): optional, shared memory for external variables
  - \(S\): optional, associated stream
- New build-in variables for indexing the threads
  - \(gridDim\): dimension of the whole grid
  - \(blockIdx\): index of the current block
  - \(blockDim\): dimension of each block in the grid
  - \(threadIdx\): index of the current thread

CUDA API

CUDA Runtime Library:
- Common component
  - Vector/Texture Types
  - Mathematical/Time Functions
- Device component
  - Mathematical/Time/Texture Functions
  - Synchronization Function
    - \texttt{__syncthreads()}\n  - Type Conversion/Casting Functions

CUDA API

CUDA Runtime Library:
- Host component
  - Structure
    - Driver API
    - Runtime API
  - Functions
    - Device, Context, Memory, Module, Texture management
    - Execution control
    - Interoperability with OpenGL and Direct3D

CUDA API

The CUDA source file uses \texttt{.cu} as extension. It contains host and device source codes.

The CUDA Compiler Driver \texttt{nvcc} can compile it and generate CPU/PTX binary code.

(PTX: Parallel Thread Execution, a device independent VM code)

PTX code may be further translated for special GPU-Arch.

CUDA API

Outline

- Introduction
- Motivation
- Programming Model
- Memory Model
- CUDA API
- Example
- Pro & Contra
- Trend

Programming Model

Simple example (Matrix addition):

- cpu c program:
  ```cpp
  #include <stdio.h>
  int main()
  
  
  ```

- cuda program:
  ```cpp
  #include <cuda_runtime.h>
  #include <stdio.h>
  int main()
  
  ```
Outline

- Introduction
- Motivation
- Programming Model
- Memory Model
- CUDA API
- Example
- **Pro & Contra**
- Trend

**Pro & Contra**

CUDA allows
- massive parallel computing
- with a relative low price
- high integrated solution
- personal supercomputing
- ecofriendly production
- easy to learn

Pro & Contra

- slightly low precision
- limited support for IEEE-754
- no recursive function call
- hard to use for irregular join/fork logic
- no concurrency between jobs

**Trend**

- More cores on-chip
- Better support for float point
- Flexible configuration & control/data flow
- Lower price
- Support higher level programming language

References

[3] Parallel Thread Execution, nVidia Corp.
Question?