CUDA Programming Model

Xing Zeng, Dongyue Mou

Outline

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

Outline

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

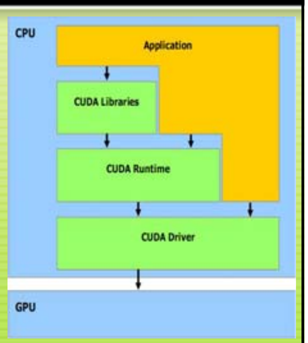
Introduction

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

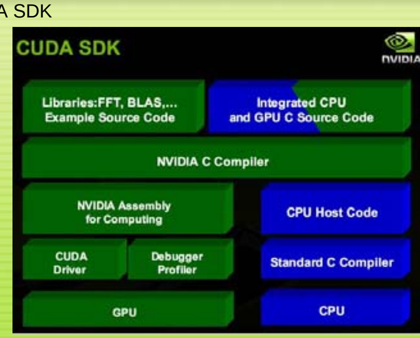
Introduction

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



Introduction

CUDA SDK



Outline

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

Motivation

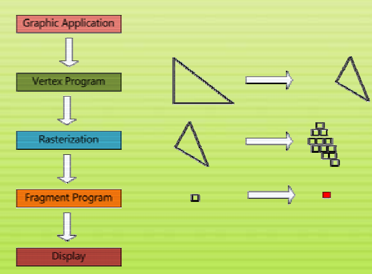
GPU Programming Model
GPGPU Programming Model
CUDA Programming Model

Motivation

GPU Programming Model
GPGPU Programming Model
CUDA Programming Model

Motivation

GPU Programming Model for Graphics



Motivation

GPU Programming Model
GPGPU Programming Model
CUDA Programming Model

Motivation

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

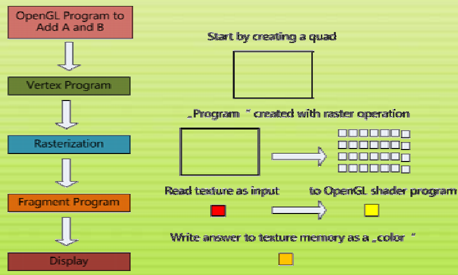
- Turn data into images ("texture maps")
- Turn algorithms into image synthesis ("rending passes")

Drawback:

- Tough learning curve
- potentially high overhead of graphics API
- highly constrained memory layout & access model
- Need for many passes drives up bandwidth consumption

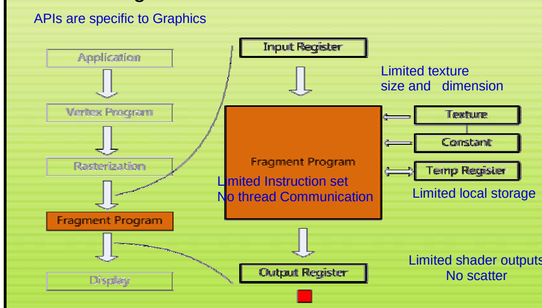
Motivation

GPGPU Programming to do A + B



Motivation

What's wrong with GPGPU 1



Motivation

What's wrong with GPGPU 2

Motivation

GPU Programming Model
GPGPU Programming Model
CUDA Programming Model

Outline

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

Programming Model

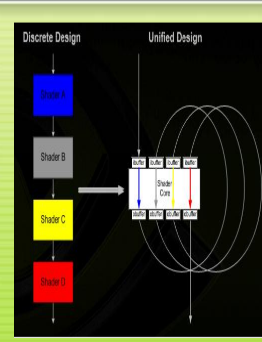
CUDA: Unified Design
Advantage:

HW: fully generally data-parallel arch-tecture.

- General thread launch
- Global load-store
- Parallel data cache
- Scalar architecture
- Integers, bit operation

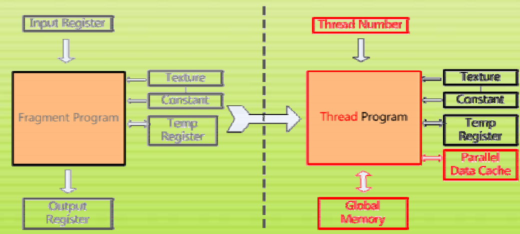
SW: program the GPU in C

- Scalable data parallel execuation/ memory model
- C with minimal yet powerful extensions



Motivation

From GPGPU to CUDA Programming Model

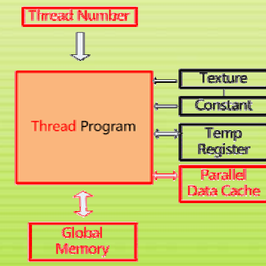


Programming Model

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

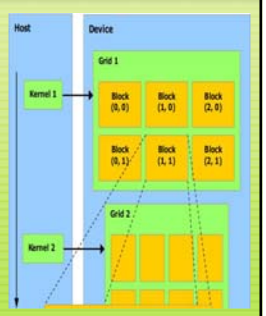
Feature 2:
- Fully general load/store to GPU memory
- Untyped, not fixed texture types
- Pointer support

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



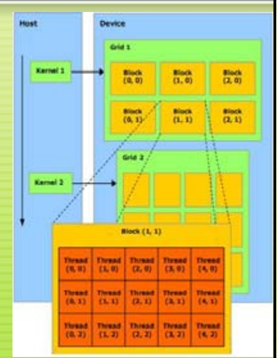
Programming Model

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



Programming Model

Important Concepts:
- Thread: basic execution unit
- Thread block:
  A batch of thread. Threads in a block cooperate together, efficiently share data.
  Thread/block have unique id
- Grid:
  A batch of thread block.
  that excuate same kernel.
  Threads in different block in the same grid cannot directly communicate with each other



Programming Model

Simple example (Matrx addition):

cpu c program:

```
void addVector (float *a, float *b,
                float *c, int N)
{
    int i, index;
    for (i = 0; i<N, i++) {
        c[index] = a[index] + b[index];
    }
}

void main()
{
   .....
   addVector(a, b, c, N);
   .....
}
```

cuda program:

```
__global__ void addVector (float *a, float *b,
                           float *c)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    c[i] = a[i] + b[i];
}

Void main()
{
   ...
   // allocation & transfer data to GPU
   //Excuate on N/256 blocks of 256 threads each
   addVector << N/256, 256 >> ( d_A, d_B, d_C);
   .....
}
```

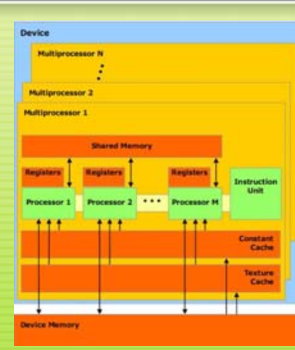
Device code

Host code

Programming Model

Hardware implementation:

A set of SIMD Multiprocessors with On-Chip shared memory



Programming Model

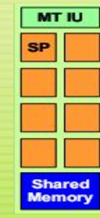
G80 Example:

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

Programming Model

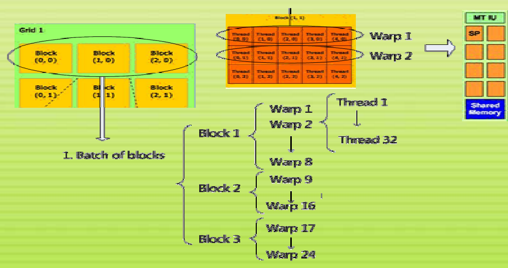
Streaming Multiprocessor (SM)

- Processing elements
  - 8 scalar thread processors
  - 32 GFLOPS peak at 1.35GHz
  - 8192 32-bit registers (32KB)
  - usual ops: float, int, branch...
- Hardware multithreading
  - up to 8 blocks (3 active) residents at once
  - up to 768 active threads in total
- 16KB on-chip memory
  - supports thread communication
  - shared amongst threads of a block



Programming Model

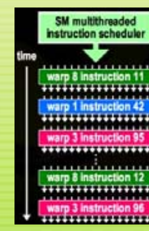
Execution Model:



Programming Model

Single Instruction Multiple Thread (SIMT) Execution:

- Groups of 32 threads formed into warps
  - always executing same instruction
  - share instruction fetch/dispatch
  - some become inactive when code path diverges
  - hardware automatically handles divergence
- Warps are primitive unit of scheduling
  - pick 1 of 24 warps for each instruction slot.
  - all warps from all active blocks are time-sliced

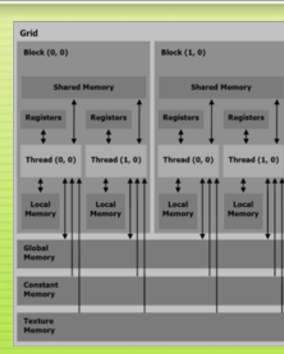


Outline

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

Memory Model

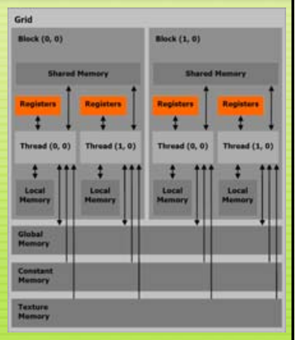
There are 6 Memory Types :



Memory Model

There are 6 Memory Types :

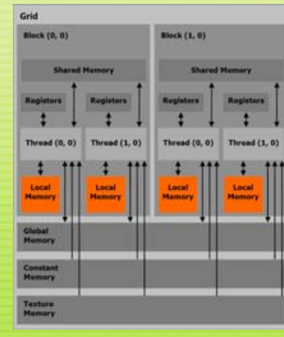
- **Registers**
  - o on chip
  - o fast access
  - o per thread
  - o limited amount
  - o 32 bit



Memory Model

There are 6 Memory Types :

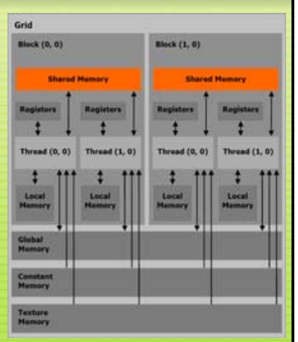
- Registers
- **Local Memory**
  - o in DRAM
  - o slow
  - o non-cached
  - o per thread
  - o relative large



Memory Model

There are 6 Memory Types :

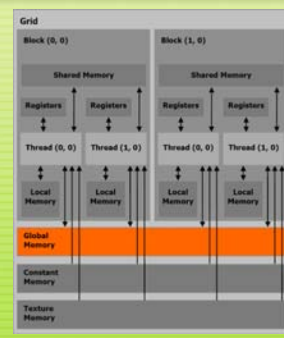
- Registers
- Local Memory
- **Shared Memory**
  - o on chip
  - o fast access
  - o per block
  - o 16 KByte
  - o synchronize between threads



Memory Model

There are 6 Memory Types :

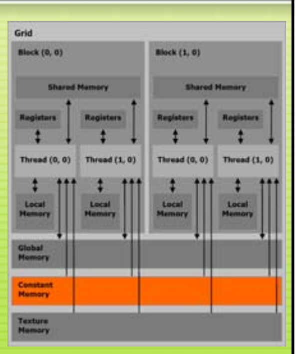
- Registers
- Local Memory
- Shared Memory
- **Global Memory**
  - o in DRAM
  - o slow
  - o non-cached
  - o per grid
  - o communicate between grids



Memory Model

There are 6 Memory Types :

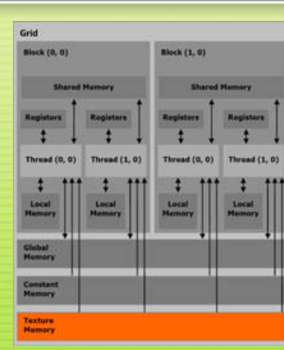
- Registers
- Local Memory
- Shared Memory
- Global Memory
- **Constant Memory**
  - o in DRAM
  - o cached
  - o per grid
  - o read-only



Memory Model

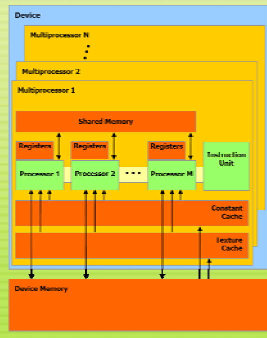
There are 6 Memory Types :

- Registers
- Local Memory
- Shared Memory
- Global Memory
- Constant Memory
- **Texture Memory**
  - o in DRAM
  - o cached
  - o per grid
  - o read-only



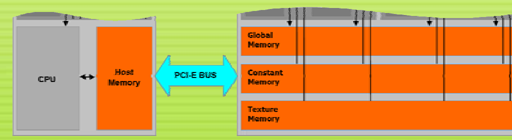
Memory Model

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



Memory Model

- Global Memory
- Constant Memory
- Texture Memory
  - o managed by host code
  - o persistent across kernels



Outline

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

CUDA API

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

It consists of:

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

CUDA API

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
```

- o 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 API

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
```

- o 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

`KernelFunc<<< Dg, Db, Ns, S >>>(...);`

  - o Dg: dimension of grid
  - o Db: dimension of block
  - o Ns: optional, shared memory for external variables
  - o S : optional, associated stream

- New build-in variables for indexing the threads
  - o `gridDim:` dimension of the whole grid
  - o `blockIdx:` index of the current block
  - o `blockDim:` dimension of each block in the grid
  - o `threadIdx:` index of the current thread

CUDA API

CUDA Runtime Library:

- Common component
  - o Vector/Texture Types
  - o Mathematical/Time Functions
- Device component
  - o Mathematical/Time/Texture Functions
  - o Synchronization Function
    - o `__syncthreads()`
  - o Type Conversion/Casting Functions

CUDA API

CUDA Runtime Library:

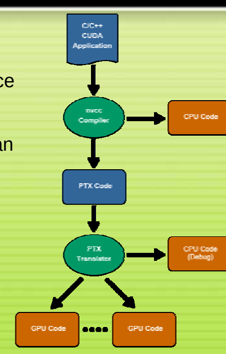
- Host component
  - o Structure
    - ▪ Driver API
    - ▪ Runtime API
  - o Functions
    - ▪ Device, Context, Memory, Module, Texture management
    - ▪ Execution control
    - ▪ Interoperability with OpenGL and Direct3D

CUDA API

The CUDA source file uses **.cu** as extension. It contains host and device source codes.

The CUDA Compiler Driver **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.



Outline

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

Programming Model

Simple example (Matrx addition):

cpu c program:

```
void addVector (fload *a, float *b,
                fload *c, int N)
{
    int i, index;
    for (i = 0; i<N, i++) {
        c[index] = a[index] + b[index];
    }
}

void main()
{
  .....
  addVector(a, b, c, N);
  .....
}
```

cuda program:

```
__global__ void addVector (fload *a, float *b,
                           float *c)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    c[i] = a[i] + b[i];
}

Void main()
{
  ...
  // allocation & transfer data to GPU
  //Excuate on N/256 blocks of 256 threads each
  addVector << N/256, 256 >> ( d_A, d_B, d_C);
  .....
}
```

Device code

Host code

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

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

Outline

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

Trend

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

References

[1] CUDA Programming Guide, nVidia Corp.
[2] The CUDA Compiler Driver, nVidia Corp.
[3] Parallel Thread Execution, nVidia Corp.
[4] CUDA: A Heterogeneous Parallel Programming Model for Manycore Computing, ASPLOS 2008, gpgpu.org


Question?