Lecture 7

CUDA (I)

Today's Content

- Introduction
  - Trends in HPC
  - GPGPU
- CUDA Programming

Trends in High-Performance Computing

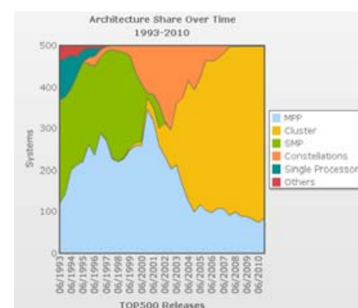
Trends

- HPC is never a commodity until 1994
- In 1990's
  - Performances of PCs are getting faster and faster
  - Proprietary computer chips offers lower and lower performance/price compared to standard PC chips
- 1994, NASA
  - Thomas Sterling, Donald Becker, … (1994)
  - 16 DX4 (40486) processors, channel bonded Ethernet.

Factors contributed to the growth of Beowulf class computers.

- The prevalence of computers for office automation, home computing, games and entertainment now provide system designers with new types of cost-effective components.
- The COTS industry now provides fully assembled subsystems (microprocessors, motherboards, disks and network interface cards).
- Mass market competition has driven the prices down and reliability up for these subsystems.
- The availability of open source software, particularly the Linux operating system, GNU compilers and programming tools and MPI and PVM message passing libraries.
- Programs like the HPCC program have produced many years of experience working with parallel algorithms.
- The recognition that obtaining high performance, even from vendor provided, parallel platforms is hard work and requires researchers to adopt a do-it-yourself attitude.
- An increased reliance on computational science which demands high performance computing.

http://www.beowulf.org/overview/history.html



MPP: Massive Parallel Processing

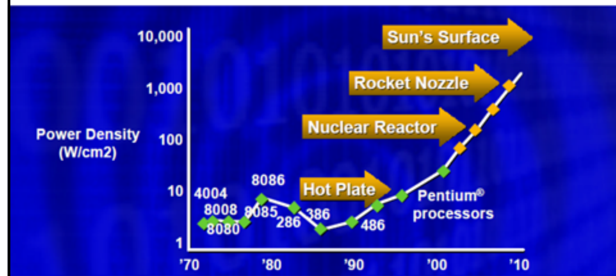
MPP vs. cluster: they are very similar. Multiple processors, each has its private memory, are interconnected through network. MPP tends to have more **sophisticated interconnection** than cluster.

Next 10 years development?

- **Multi-core CPU** is already commodity / standard.
  - That's why OpenMP is becoming more and more important ;)
  - Intel has 80-core prototype processors in their lab in 2006…
- Cluster/MPP approach is showing its age with diminishing increase in performance
  - Increasing in clock-speed for general purpose processors (GPP) is slowing down due to power consumption.
- Moore's law suggests number of transistors can be double every 18 – 24 months. These increases can be used better for special-purposed processing.

http://news.cnet.com/Intel-shows-off-80-core-processor/2100-1006_3-6158181.html
http://hpc.pnl.gov/projects/hybrid-computing/

Power Density



Source: Patrick Gelsinger, *Intel Developer's Forum*, Intel Corporation, 2004.

Next 10 years of development?

- **Hybrid** computing system
  - Systems that employ more than one type of computing engine to perform application tasks.
  - General purpose processors (e.g. x86) do not necessarily perform well on all tasks. Specialized processors can be built for some special-purpose tasks.
    - GPU → much better at 3D computer graphics
    - FPGA (Field Programmable Gate Arrays) – the processor's hardware can be "reconfigured" or "programmed" to invest its transistors on performing certain tasks.
    - DSP (Digital Signal Processors)

http://hpc.pnl.gov/projects/hybrid-computing/

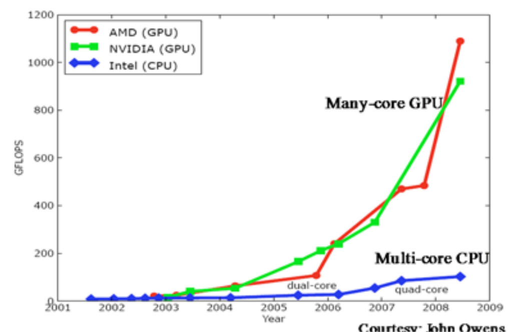
Introduction to GPGPU

GPGPU

General-**P**urpose computing using **G**raphical **P**rocessing **U**nit

- **GPGPU:** Has been a hot topic in research community since early 2000.
- High-performance (>10x of current generation General Purpose CPUs)
- Highly efficient
  - Cost efficient
  - Space efficient
  - Green / Energy-efficient

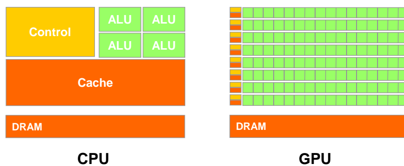
GPU Performance Growth



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, Spring 2010, University of Illinois, Urbana-Champaign

Why GPU is faster than CPU ?

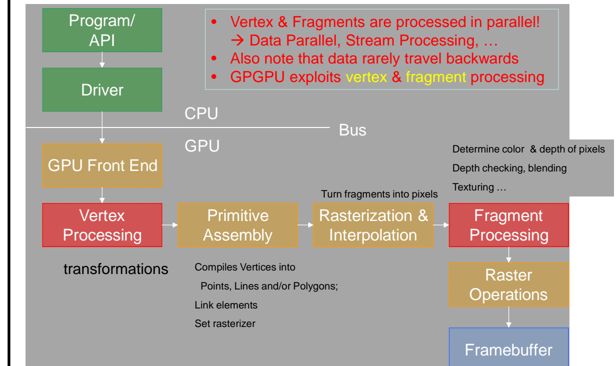
- The GPU is ***specialized*** for compute-intensive, highly data parallel computation (graphics rendering)
  - So, more transistors can be devoted to data processing rather than data caching and flow control



CPU GPU

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

GPU Pipeline



Shading Languages

- Cg
  - NVIDIA
- HLSL
  - Microsoft
  - High Level Shader Language
  - DirectX 8+
  - XBox, XBox360
- GLSL
  - OpenGL shading language

http://en.wikipedia.org/wiki/Shading_language

Barrier into GPGPU

- OpenGL, DirectX ?

- Graphic pipeline ?
- Map between computation versus drawing ?
  - Open a drawing canvas
  - Array → texture
  - Compute kernel → shader
  - Compute → drawing
  - http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html#feedback

Efforts for GPGPU

Programming Language Approach

- BrookGPU
  - Around 2003 @ Stanford University
  - Ian Buck, now with NVIDIA
  - Layering over DirectX 9, OpenGL, CTM, Cg
  - Incorporated into the Stream SDK, AMD to replace it CTM
- Sh
  - Conceived @ University of Waterloo, Canada
  - General release around 2005
  - Evolved and commercialized @ RapidMind, Inc.
- **CUDA**
  - Released at 2006 @ NVIDIA
- **OpenCL**
  - 2008 @ Apple WDC, now supported by Nvidia, AMD/ATI
- DirectCompute
  - Microsoft DirectX 10+ on Windows Vista & Windows 7

http://www.gpgpu.org/cgi-bin/blosxom.cgi/High-Level%20Languages/index.html

CUDA

Compute Unified Device Architecture

18

Outline

- CUDA Architecture
- CUDA programming model
- CUDA-C

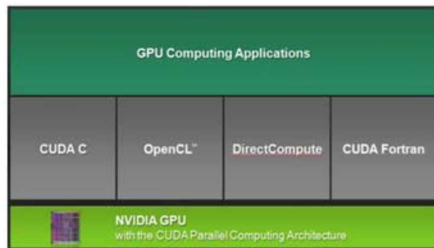
19

CUDA Architecture

20

CUDA™: a General-Purpose Parallel Computing Architecture

- CUDA is designed to support various languages or APIs



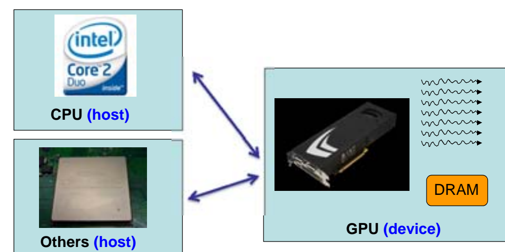
NVIDIA_CUDA_Programming_Guide_3.0

21

CUDA Device and Threads

- Host: usually the general purpose CPU running the computer system
- Host memory: DRAM on the host
- Initiates the execution of kernels

- Device: coprocessor to the CPU/Host
- Device memory: memory on the device
- Runs many threads in parallel

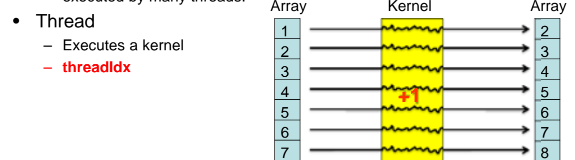


CUDA Programming Model

23

Programming Concepts

- CUDA considers "many-core" processors
- Need **lots and lots of threads** to make it efficient!
- Kernel
  - Kernels are executed by devices through **threads**. One kernel may be executed by many threads.
- Thread
  - Executes a kernel
  - **threadIdx**



Single thread vs. multiple threads

```
Single thread + loop
int a[100]={0};
for(int i=0;i<100;i++) {
    a[i]++;
}
```

25

Single thread vs. multiple threads

```
100 threads, each thread has an
Id starts from 0
a[id]++;
```

26

Thread Block

- Each thread has its unique ID within a block
- One thread block contains many threads arranged in 1D, 2D, or 3D
  - Process data in linear array, in matrices, or in volumes
  - Threads get their id by threadIdx.x, threadIdx.y, threadIdx.z
- Holds up to 512 threads in current GPU
  - The limit exists because threads in a block are expected to execute on the same processor core
  - Threads in the same block can cooperate
- Threads within a block can cooperate via **shared memory**, **atomic operations** and **barrier synchronization**

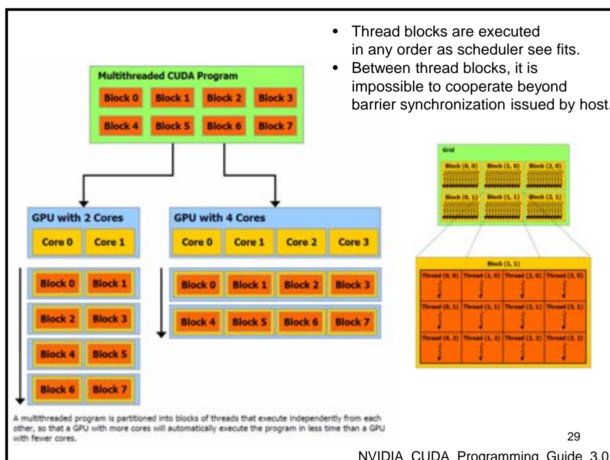
27

Grid

- Thread blocks are arranged into grid (1D or 2D)
- So CUDA programs have grid, each grid point has a thread block, and each thread block contains threads arranged in 1D, 2D, or 3D
- Dimensional Limits:
  - Grid: 65535 x 65536
  - Thread block : 512 x 512 x 64 (the total number must be < 512)
- NO cooperation between thread blocks
- This grid → blocks → threads alllows CUDA programs scale on different hardware.
- Programmers can use **blockIdx, blockDim,** and **threadIdx** to identify their global location/ranking/id to deduce its portion of work (similar to rank in MPI)
  - blockIdx.x, blockIdx.y, blockDim.x, blockDim.y, blockDim.z
  - threadIdx.x, threadIdx.y, threadIdx.z

28

- Thread blocks are executed in any order as scheduler see fits.
- Between thread blocks, it is impossible to cooperate beyond barrier synchronization issued by host.



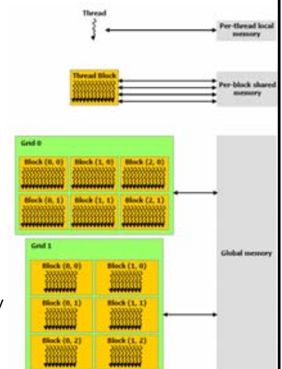
A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

29

NVIDIA_CUDA_Programming_Guide_3.0

Programming Concepts

- Memory Hierarchy
  - Local memory
  - Shared memory
  - Global memory (R/W)
    - Constant memory (R)
    - Texture memory (R)
- Global memories can be accessed by both device & host, but they are optimized for different purposes..
- Need to choose appropriate memory types to use. But for now, just keep this in mind.



Refresh

- Kernel
- Thread
- Thread Block
- Grid
- Memory
  - Local, shared, global / constant / texture

CUDA-C

CUDA-C

- CUDA-C extends C/C++
  - Declspecs (declaration specifications):
    - Functions declaration: global, device, host
    - Variable declaration: shared, local, constant
  - Built-in variables
    - threadIdx, blockIdx, blockDim
  - Intrinsics (intrinsic function)
    - __synthreads, cudaThreadSynchronize(), …
  - Runtime API
    - Memory, execution management, etc.
  - Launching kernels

intrinsic function: functions handled by compilers instead of library functions. (http://en.wikipedia.org/wiki/Intrinsic_function /)

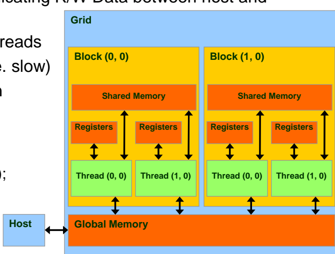
Examples

0. Memory Allocation & Initialization
1. Data Movement

00.cu
Memory allocation & initialization

- Before we use GPU to do any computation, we need data on GPU first. Unless, data are generated on GPU.
- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access (i.e. slow)
  - Content persist between kernel launches

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```



00.cu
Memory allocation & initialization

```
double host(const unsigned long size) {
  double t1=0;
  int count=0;
  stopWatch timer;

  char *data
  data  = malloc(size);
  do {
    timer.start();
    memset(data, 0, size);

    timer.stop();
    t1 += timer.elapsedTime();
    count++;
  } while(t1<1.0);
  free(data);
  return t1/count;
}
```

void* malloc(size_t size)
allocate dynamic memory

void *memset(void *s, int c, size_t n);
fill memory with a constant byte

void free(void *ptr);
free the memory space pointed by ptr

00.cu
Memory allocation & initialization

```
double host(const unsigned long size) {
  double t1=0;
  int count=0;
  stopWatch timer;

  char *data;
  data  = malloc(size);
  do {
    timer.start();
    memset(data, 0, size);

    timer.stop();
    t1 += timer.elapsedTime();
    count++;
  } while(t1<1.0);
  free(data);
  return t1/count;
}
```

```
double device(const unsigned long size) {
  double t2=0;
  int count=0;
  stopWatch timer;

  char *data;
  cudaMalloc((void**) &data, size);
  do {
    timer.start();
    cudaMemset(data, 0, size);
    cudaThreadSynchronize();
    timer.stop();
    t2 += timer.elapsedTime();
    count++;
  } while(t2 < 1.0);
  cudaFree(data);
  return t2/count;
}
```

37

00.cu
Bandwidth on erasing data

I7 950 3.07GHz + GeFore GT220		Xeon E5310 1.6GHz + GeForce 8800Ultra		Q9550 2.83GHz + GeForce GTX280		I7 950 3.07GHz + NVidia Tesla c2050	
Host	Device	Host	Device	Host	Device	Host	Device
11.04	16.13	2.73	50.27	5.42	66.00	11.05	104.52
Config #0		Config #1		Config #2		Config #3	

01.cu
Memory copy between two memory blocks

```
__host__ double memop2(void *dst, void *src, const unsigned long size, cudaMemcpyKind dir) {
    double t = 0;
    int count=0;
    stopWatch timer;
    do {
        timer.start();
        memcpy(dst, src, size);              cudaMemcpy(dst, src, size, dir);
                                             cudaThreadSynchronize();
        timer.stop();
        t += timer.elapsedTime();
        count++;
    } while(t < T_MIN);

    return t/count;
}
```

```
enum cudaMemcpyKind {
   cudaMemcpyHostToDevice,
   cudaMemcpyDeviceToHost,
   cudaMemcpyDeviceToDevice,
   cudaMemcpyHostToHost
}
```

Example 1. Memory performance
Copy

Config #0

Host2Device:0.0440951secs., bandwidth=5.66957GB/sec.
Device2Host:0.0454369secs., bandwidth=5.50213GB/sec.
Device2Devi:0.0244666secs., bandwidth=20.436GB/sec.
Host2Host :0.0403944secs., bandwidth=12.378GB/sec.

Config #1

Host2Device:0.0867472secs., bandwidth=2.88194GB/sec.
Device2Host:0.0808961secs., bandwidth=3.09038GB/sec.
Device2Devi:0.046854secs., bandwidth=10.6715GB/sec.
Host2Host :0.103345secs., bandwidth=4.83814GB/sec.

Config #2

Host2Device:0.0800946secs., bandwidth=3.12131GB/sec.
Device2Host:0.0800039secs., bandwidth=3.12485GB/sec.
Device2Devi:0.0349145secs., bandwidth=14.3207GB/sec.
Host2Host :0.0471595secs., bandwidth=10.6023GB/sec.

Config #3

Host2Device:0.0889612secs., bandwidth=5.62043GB/sec.
Device2Host:0.0822179secs., bandwidth=6.0814GB/sec.
Device2Devi:0.0121208secs., bandwidth=82.5028GB/sec.
Host2Host :0.0716965secs., bandwidth=13.9477GB/sec.

CUDA-C
APIs for memory management

- cudaError_t cudaMalloc (void *devPtr, size_t size)
- cudaError_t cudaMallocHost (void **ptr, size_t size)
  - Allocates size bytes of host memory that is **page-locked/pinned** and accessible to the device.
- cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - cudaMemcpyHostToDevice,
  - cudaMemcpyDeviceToHost,
  - cudaMemcpyDeviceToDevice,
  - cudaMemcpyHostToHost
- cudaError_t cudaFree (void *devPtr)
- cudaError_t cudaMemset (void *devPtr, int value, size_t count)

41

CUDA Reference manual 3.0: Section 4.8: memory management

Readings

- No programming assignment this week!
  - Serial implementation of the term project due on May. 10, 2011
- NVIDIA CUDA Programing Guide (4.0RC2)
  - Chap 1, 2 (p.1 – 14)
  - Most of Section 3.2 (p. 18 – 46)
  - Chap 4 (p. 85-88), optional
  - Chap 5 (p. 89 – 102)
- Quickly browse Appendix B, C (p.107 – 144)

42

http://140.118.5.6:8000/doc/CUDA/4.0RC2/CUDA_C_Programming_Guide.pdf