Blockseminar:
Verteiltes Rechnen und Parallelprogrammierung
Tim Conrad
Today

• Introduction to GPGPUs
• “Hands on” to get you started
• Assignment 3

• Project ideas?
  Examples: http://wikis.fu-berlin.de/display/mathinfppdc/Home
Von Neumann architecture: instructions are sent from memory to the CPU

Serial execution: Instructions are executed one after another on a single Central Processing Unit (CPU)

Problems:
• More expensive to produce
• More expensive to run
• Bus speed limitation
Official-sounding definition: The simultaneous use of multiple compute resources to solve a computational problem.

Benefits:
• Economical – requires less power and cheaper to produce
• Better performance – bus/bottleneck issue

Limitations:
• New architecture – Von Neumann is all we know!
• New debugging difficulties – cache consistency issue
Flynn’s Taxonomy

Classification of computer architectures, proposed by Michael J. Flynn

- **SISD** – traditional serial architecture in computers.
- **SIMD** – parallel computer. One instruction is executed many times with different data (think of a for loop indexing through an array)
- **MISD** - Each processing unit operates on the data independently via independent instruction streams. Not really used in parallel
- **MIMD** – Fully parallel and the most common form of parallel computing.
CUDA is NVIDIA’s general purpose parallel computing architecture

- Designed for calculation-intensive computation on GPU hardware
- CUDA is not a language, it is an API
- We will look at the C implementation of CUDA
- There exist wrapper for Java (jCUDA) and Python (PyCUDA)
What is GPGPU?

• General Purpose computation using GPU in applications other than 3D graphics
  – GPU accelerates critical path of application
• Data parallel algorithms leverage GPU attributes
  – Large data arrays, streaming throughput
  – Fine-grain SIMD parallelism
  – Low-latency floating point (FP) computation
• Applications – see GPGPU.org
  – Game effects (FX) physics, image processing
  – Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting
Terms

• What is GPGPU?
  – General-Purpose computing on a Graphics Processing Unit
  – Using graphic hardware for non-graphic computations

• What is CUDA?
  – Compute Unified Device Architecture
  – Software architecture for managing data-parallel programming
CUDA Goals

• Scale code to hundreds of cores running thousands of threads
• The task runs on the GPU independently from the CPU
Graphic cards?

Logical Representation of Visual Information

Output Signal
History of Graphics Hardware

• ... - mid ’90s
  – SGI mainframes and workstations
  – PC: only 2D graphics hardware
• mid ’90s
  – Consumer 3D graphics hardware (PC)
    • 3dfx, nVidia, Matrox, ATI, ...
  – Triangle rasterization (only)
  – Cheap: pushed by game industry
• 1999
  – PC-card with TnL [ Transform and Lighting ]
    • nVIDIA GeForce: Graphics Processing Unit (GPU)
  – PC-card more powerful than specialized workstations
• Modern graphics hardware
  – Graphics pipeline partly programmable
  – Leaders: ATI and nVidia
    • “ATI Radeon X1950” and “nVidia GeForce 8800”
  – Game consoles similar to GPUs (Xbox)
• Why are GPUs used for computing?
Motivation

• High CPU load
  – Physics, AI, sound, network, …

• Graphics demand:
  – Fast memory access
    • Many lookups [ vertices, normal, textures, … ]
  – High bandwidth usage
    • A few GB/s needed in regular cases!
  – Large number of flops
    • Flops = Floating Point Operations [ ADD, MUL, SUB, … ]
    • Illustration: matrix-vector products
      \[(16 \text{ MUL} + 12 \text{ ADD}) \times (#\text{vertices} + #\text{normals}) \times \text{fps} = \]
      \[(28 \text{ Flops}) \times (6.000.000) \times 30 \approx 5\text{GFlops}\]

• Conclusion: Real time graphics needs supporting hardware!
CPU vs. GPU

• CPU
  – Fast caches
  – Branching adaptability
  – High performance

• GPU
  – Multiple ALUs (arithmetic logic unit)
  – Fast onboard memory
  – High throughput on parallel tasks
    • Executes program on each fragment/vertex

• CPUs are great for *task* parallelism
• GPUs are great for *data* parallelism
• More transistors devoted to data processing
GPU Architecture
• Processing element = thread processor = ALU
Memory Architecture

- Constant Memory
- Texture Memory
- Device Memory
Data-parallel Programming

- Think of the CPU as a massively-threaded co-processor
- Write “kernel” functions that execute on the device -- processing multiple data elements in parallel

- Keep it busy! ⇒ massive threading
- Keep your data close! ⇒ local memory
Modern Graphics Hardware

• Features of nvidia Tesla C870
  – Core speed 1.35 Ghz
  – 128 Cores (16x8 shader processors)
  – 51 GB/s memory bandwidth
  – 1.5 GB memory
Modern Graphics Hardware

• High Memory Bandwidth

- GPU: 1.35Mhz
- Graphics memory: 1.5GB
- CPU: 3GHz
- Main memory: 2GB
- AGP bus: 2GB/s
- AGP memory: ½GB
- Processor Chip: High bandwidth 77GB/s
- High bandwidth: 51GB/s
- Parallel Processes
- High Memory Bandwidth

- Graphics Card Output
• How to program a GPGPU?
GPU Intro

CUDA Software Developer’s Kit (SDK)

Libraries: FFT, BLAS, ..., Example Source Code

Integrated CPU and GPU C Source Code

NVIDIA C Compiler

NVIDIA Assembly for Computing

CUDA Driver

Debugger Profiler

CPU Host Code

Standard C Compiler

GPU

CPU
NVCC Compiler’s Role: Partition Code and Compile for Device

mycode.cu

```c
int main_data;
__shared__ int sdata;

Main() {}
__host__ hfunc () {
  int hdata;
  <<<gfunc(g,b,m)>>>();
}

__global__ gfunc() {
  int gdata;
}

__device__ dfunc() {
  int ddata;
}
```

Compiled by native compiler: gcc, icc, cc

```c
int main_data;
__shared__ sdata;

Main() {}
__host__ hfunc () {
  int hdata;
  <<<gfunc(g,b,m)>>>();
}

__global__ gfunc() {
  int gdata;
}
```

Compiled by nvcc compiler

```c
int main_data;
__shared__ sdata;

Main() {}
__host__ hfunc () {
  int hdata;
  <<<gfunc(g,b,m)>>>();
}

__global__ gfunc() {
  int gdata;
}

__device__ dfunc() {
  int ddata;
}
```
CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few
A kernel is executed as a grid of thread blocks
  - All threads share data memory space

A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory

Two threads from two different blocks cannot cooperate
Block and Thread IDs

• Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})

• Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...
Important Concepts:

- **Device**: GPU, viewed as a co-processor.
- **Host**: CPU
- **Kernel**: data-parallel, computed-intensive positions of application running on the device.
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 execute same kernel. Threads in different block in the same grid cannot directly communicate with each other
Simple matrix multiplication example:

**CPU C program:**
```c
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:**
```c
__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
    // Execute on N/256 blocks of 256 threads each
    addVector <<< N/256, 256 >>> (d_A, d_B, d_C);
    ....
}
```
Hardware implementation:

A set of SIMD Multiprocessors with On-Chip shared memory

See also:

http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
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
  - 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
Execution Model:

1. Batch of blocks

- **Block 1**
  - Warp 1
  - Warp 2
  - Warp 8
  - Warp 9
  - Warp 16
- **Block 2**
  - Warp 17
- **Block 3**
  - Thread 32
  - Thread 1

Warp 1 and Warp 2 go to MT IU.
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
• Memory Model
There are 6 Memory Types:

- Grid
  - Block (0, 0)
    - Shared Memory
    - Registers
    - Thread (0, 0)
    - Local Memory
    - Global Memory
  - Block (1, 0)
    - Shared Memory
    - Registers
    - Thread (0, 0)
    - Local Memory
    - Global Memory

- Constant Memory

- Texture Memory
There are 6 Memory Types:

- **Registers**
  - on chip
  - fast access
  - per thread
  - limited amount
  - 32 bit
There are 6 Memory Types:

- Registers
- **Local Memory**
  - in DRAM
  - slow
  - non-cached
  - per thread
  - relative large
There are 6 Memory Types:

- Registers
- Local Memory
- **Shared Memory**
  - on chip
  - fast access
  - per block
  - 16 KByte
  - synchronize between threads
There are 6 Memory Types:

- Registers
- Local Memory
- Shared Memory
- **Global Memory**
  - in DRAM
  - slow
  - non-cached
  - per grid
  - communicate between grids
There are 6 Memory Types:

- Registers
- Local Memory
- Shared Memory
- Global Memory
- **Constant Memory**
  - in DRAM
  - cached
  - per grid
  - read-only
There are 6 Memory Types:

- Registers
- Local Memory
- Shared Memory
- Global Memory
- Constant Memory
- Texture Memory
  - in DRAM
  - cached
  - per grid
  - read-only
CUDA Device Memory Space Overview

• Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

• The host can R/W global, constant, and texture memories
• Global memory
  – Main means of communicating R/W Data between host and device
  – Contents visible to all threads

• Texture and Constant Memories
  – Constants initialized by host
  – Contents visible to all threads
Memory Model

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

- Global Memory
- Constant Memory
- Texture Memory
  - managed by host code
  - persistent across kernels
• CUDA API
CUDA Highlights:
Easy and Lightweight

• The API is an extension to the ANSI C programming language
  → Low learning curve

• The hardware is designed to enable lightweight runtime and driver
  → High performance
To compute, we need to:

- **Allocate** memory that will be used for the computation (variable declaration and allocation)
- **Read** the data that we will compute on (input)
- **Specify** the **computation** that will be performed
- **Write** to the appropriate device the results (output)
A GPU is a specialized computer

- We need to allocate space in the video card’s memory for the variables.
- The video card does not have I/O devices, hence we need to copy the input data from the memory in the host computer into the memory in the video card, using the variable allocated in the previous step.
- We need to specify code to execute.
- Copy the results back to the memory in the host computer.
Initially:

- **Host’s Memory**
  - array
- **GPU Card’s Memory**
Allocate Memory in the GPU card

- **array**
  - Host’s Memory

- **array_d**
  - GPU Card’s Memory
Copy content from the host’s memory to the GPU card memory
Execute code on the GPU

array
Host’s Memory

GPU MPs

array_d
GPU Card’s Memory
Copy results back to the host memory

array
Host’s Memory

array_d
GPU Card’s Memory
The Kernel

• It is necessary to write the code that will be executed in the stream processors in the GPU card
• That code, called the kernel, will be downloaded and executed, simultaneously and in lock-step fashion, in several (all?) stream processors in the GPU card
• How is every instance of the kernel going to know which piece of data it is working on?
Grid Size and Block Size

- Programmers need to specify:
  - The grid size: The size and shape of the data that the program will be working on
  - The block size: The block size indicates the sub-area of the original grid that will be assigned to an MP (a set of stream processors that share local memory)
• Recall that the “stream processors” of the GPU are organized as MPs (multi-processors) and every MP has its own set of resources:
  – Registers
  – Local memory

• The block size needs to be chosen such that there are enough resources in an MP to execute a block at a time.
Block boundaries in the GPU:

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

It consists of:

- A minimal set of extensions to C/C++
  - type qualifiers
  - call-syntax
  - build-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

  ```c
  __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 C/C++ Extensions:

- New syntax to invoke the device code

```
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 Runtime Library:

• **Common component**
  - Vector/Texture Types
  - Mathematical/Time Functions

• **Device component**
  - Mathematical/Time/Texture Functions
  - Synchronization Function
    - `__syncthreads()`
  - Type Conversion/Casting Functions
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
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.
Simple working code example

• Goal for this example:
  – Really simple but illustrative of key concepts
  – Fits in one file with simple compile command
  – Can absorb during lecture

• What does it do?
  – Scan elements of array of numbers (any of 0 to 9)
  – How many times does “6” appear?
  – Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

Known as a cyclic data distribution
CUDA Pseudo-Code

**MAIN PROGRAM:**

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call *host* function

Calculate final output from per-thread output

Print result

**HOST FUNCTION:**

Allocate memory on device for copy of *input* and *output*

Copy input to *device*

Set up grid/block

Call *global* function

Copy *device* output to host

**GLOBAL FUNCTION:**

Thread scans subset of array elements

Call *device* function to compare with “6”

Compute local result

**DEVICE FUNCTION:**

Compare current element and “6”

Return 1 if same, else 0
Main Program: Preliminaries

**Main Program:**

Initialization
- Allocate memory on host for input and output
- Assign random numbers to input array

Call *host* function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

int main(int argc, char **argv)
{
    int *in_array, *out_array;
    ...
}
```
Main Program: Invoke Global Function

**MAIN PROGRAM:**

Initialization *(OMIT)*
- Allocate memory on host for input and output
- Assign random numbers to input array

Call *host* function

Calculate final output from per-thread output

Print result

```c
#include <stdio.h>
define SIZE 16
define BLOCKSIZE 4
__host__ void outer_compute (int *in_arr, int *out_arr);

int main(int argc, char **argv)
{
    int *in_array, *out_array;
    /* initialization */ …
    outer_compute(in_array, out_array);
    …
}
```
Main Program: Calculate Output & Print Result

Main Program:

Initialization (OMIT)
- Allocate memory on host for input and output
- Assign random numbers to input array

Call host function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute(int *in_arr, int *out_arr);

int main(int argc, char **argv) {
    int *in_array, *out_array;
    int sum = 0;
    /* initialization */ ...
    outer_compute(in_array, out_array);
    for (int i=0; i<BLOCKSIZE; i++) {
        sum+=out_array[i];
    }
    printf ("Result = %d\n",sum);
}
```
HOST FUNCTION:

Allocate memory on device for copy of
*input* and *output*

Copy input to *device*

Set up grid/block

Call *global* function

Copy *device* output to host

```c
__host__ void outer_compute (int *h_in_array, int *h_out_array) {

    int *d_in_array, *d_out_array;

    cudaMemcpy((void **)&d_in_array, SIZE*sizeof(int));
    cudaMemcpy((void **)&d_out_array, BLOCKSIZE*sizeof(int));

    ...
}
```
HOST FUNCTION:

Allocate memory on device for copy of input and output

Copy input to device

Set up grid/block

Call global function

Copy device output to host

__host__ void outer_compute (int *h_in_array, int *h_out_array) {
    int *d_in_array, *d_out_array;

    cudaMalloc((void **) &d_in_array,
               SIZE*sizeof(int));

    cudaMalloc((void **) &d_out_array,
               BLOCKSIZE*sizeof(int));

    cudaMemcpy(d_in_array, h_in_array,
               SIZE*sizeof(int),
               cudaMemcpyHostToDevice);

    ... do computation ...

    cudaMemcpy(h_out_array, d_out_array,
               BLOCKSIZE*sizeof(int),
               cudaMemcpyDeviceToHost);
}

HOST FUNCTION:
Allocate memory on device for copy of input and output
Copy input to device
Set up grid/block
Call global function
Copy device output to host

```c
__host__ void outer_compute (int *h_in_array, int *h_out_array) {
    int *d_in_array, *d_out_array;

    cudaMalloc((void **) &d_in_array,
               SIZE*sizeof(int));
    cudaMalloc((void **) &d_out_array,
               BLOCKSIZE*sizeof(int));
    cudaMemcpy(d_in_array, h_in_array,
                SIZE*sizeof(int),
                cudaMemcpyHostToDevice);
    msize = (SIZE+BLOCKSIZE) * sizeof (int);
    compute<<<1,BLOCKSIZE,msize>>>(d_in_array,
                                  d_out_array);
    cudaMemcpy(h_out_array, d_out_array,
               BLOCKSIZE*sizeof(int),
               cudaMemcpyDeviceToHost);
}
```
GLOBAL FUNCTION:

Thread scans subset of array elements

Call device function to compare with “6”

Compute local result

```c
__global__ void compute(int *d_in, int *d_out) {
    d_out[threadIdx.x] = 0;
    for (int i=0; i<SIZE/BLOCKSIZE; i++) {
        int val = d_in[i*BLOCKSIZE + threadIdx.x];
        d_out[threadIdx.x] += compare(val, 6);
    }
}
```
DEVICE FUNCTION:

Compare current element and "6"
Return 1 if same, else 0

```c
__device__ int compare(int a, int b) {
    if (a == b) return 1;
    return 0;
}
```
Another Example: Adding Two Matrices

**CPU C program**

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

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

**CUDA C program**

```c
__global__ void add_matrix_gpu(float *a, float *b, float *c, int N) {
  int i =blockIdx.x*blockDim.x+threadIdx.x;
  int j=blockIdx.y*blockDim.y+threadIdx.y;
  int index =i+j*N;
  if( i <N && j <N)
    c[index]=a[index]+b[index];
}
```

```c
void main() {
  dim3 dimBlock(blocksize,blocksize);
  dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
  add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
}
```
• Define 2-d set of blocks, and 2-d set of threads per block

```c
    dim3 dimBlock(blocksize,blocksize);
    dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

• Each thread identifies what element of the matrix it operates on

```c
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    int j=blockIdx.y*blockDim.y+threadIdx.y;
    int index =i+j*N;
    if( i <N & & j <N)
        c[index]=a[index]+b[index];
```
That’s it for today!