Exploring Massive Parallel Computation with GPU

Ian Bond

Massey University, Auckland, New Zealand

2011 Sagan Exoplanet Workshop
Pasadena, July 25-29 2011
Assumptions/Purpose

- You are all involved in microlensing modelling and you have (or are working on) your own code
- this lecture shows how to get started on getting code to run on a GPU
- then its over to you . . .
Outline

1. Need for parallelism
2. Graphical Processor Units
3. Gravitational Microlensing Modelling
Parallel Computing is use of multiple computers, or computers with multiple internal processors, to solve a problem at a greater computational speed than using a single computer (Wilkinson 2002).

How does one achieve parallelism?
A grand challenge problem is one that cannot be solved in a reasonable amount of time with today's computers’

Examples:
- Modelling large DNA structures
- Global weather forecasting
- N body problem (N very large)
- Brain simulation

Has microlensing modelling become a grand challenge problem?
Achieving Parallelism

- History
  - Thinking Machines, Cray, Sun Starfire, Beowulf clusters, ...

- Three ways of achieving parallelism today
  - Shared memory multiprocessor
  - Distributed Memory multicomputer
  - Graphical processing units
Flynns Classifications

- **SISD.** Single instruction, single data stream
  - a single stream of instructions is generated by the program and operates on a single stream of data items.

- **SIMD.** Single instruction, multiple data stream
  - instructions from program are broadcast to more than one Processor. Each processor executes the same instruction in synchronism, but using different data.

- **MISD.** Multiple instruction, single data stream
  - a computer with multiple processors each sharing a common memory. There are multiple streams of instructions and one stream of data.

- **MIMD.** Multiple instruction, multiple data stream
  - each instruction stream operates upon different data.
Examples: most multicore PCs
- All memory shared across all processors via a single address space
- Program using threads. OpenMP makes it easier.
Distributed Memory Multicomputers: aka cluster computers. Two programming models:

- **Multiple Program Multiple Data (MPMD)**
  - Each processor will have its own program to execute
  - Parallel Virtual Machine (PVM) library

- **Single Program Multiple Data (SPMD)**
  - A single source program is written, and each processor executes its own personal copy of the program
  - MPI standard
Standard for communication across several processors, developed by group of academics and industrial partners

MPI is a standard - it defines routines, not implementations

Several free implementations exist: openmpi for Ubuntu.
Outline

1. Need for parallelism
2. Graphical Processor Units
3. Gravitational Microlensing Modelling
What are GPUs?

- A GPU (on a graphics card) offloads/accelerates graphics rendering from a CPU.
- Modern GPU functions:
  - rendering polygons
  - texture mapping
  - coordinate transformations
  - accelerated video decoding
- Manufacturers
  - NVIDIA
  - ATI
General Purpose Computing on Graphical Processing Units
– using a GPU for applications traditionally handled by a CPU

Stream Processing
– stream of data
– a series of operations applied to that stream—the kernel functions

SPMD
– single program, multiple data
– related to, but not the same as SIMD
GPU architecture

Physical layout varies among GPU makes and models, but follows this general idea:

- GPU divided into blocks
- Each block contains one or more streaming multiprocessors
- Each SM has a number of streaming processors
  - all share the same control logic and instruction cache within an SM
- All SPs from all SMs have access to up to 4 GB GDDR DRAM global memory
  - GDDR: graphics double data rate
  - DRAM: dynamic random access memory
Approach is to make use of the GPU AND the CPU

- **CUDA**
  - Compute Unified Device Architecture
  - Developed and distributed by NVIDIA

- **OpenCL**
  - tedious and not as good performance as CUDA (according to NVIDIA)

Now lets get started...
Setting Up CUDA

See http://www.r-tutor.com/gpu-computing/cuda-installation/cuda3.2-ubuntu

- Make sure you have a graphics card, install Ubuntu.
- Disable the nouveau nvidia driver that comes with Ubuntu. Reboot in safe graphics mode (hold down shift key).
- Install the linux developer tools and the OpenGL development driver.
- Install the CUDA development driver (after downloading from CUDA download site). Switch to console mode for this (ctrl-alt-f2).
- Download and install the CUDA toolkit. Usually in /usr/local/cuda
- Download and install the CUDA SDK samples. Usually in your personal home directory.
Check out your system

- Run the device query sample program from your CUDA SDK samples:
  $
  \text{cd } ~/\text{CUDASDK3.2/C/bin/linux/release/}
  
  \text{./deviceQuery}

- Look at the output:
  - How many graphics devices are there?
  - How many multiprocessors and cores?
  - How much global memory?
  - ...
Device Query Screenshot

CUDA Device Query (Runtime API) version (CUDART static linking)

There are 2 devices supporting CUDA

**Device 0: "GeForce GTX 480"**
- CUDA Driver Version: 3.20
- CUDA Runtime Version: 3.20
- CUDA Capability Major/Minor version number: 2.0
- Total amount of global memory: 1619285056 bytes
- Multiprocessors x Cores/MP = Cores: 15 (MP) x 32 (Cores/MP) = 480 (Cores)
- Total amount of constant memory: 65536 bytes
- Total amount of shared memory per block: 49152 bytes
- Total number of registers available per block: 32768
- Warp size: 32
- Maximum number of threads per block: 1024
- Maximum sizes of each dimension of a block: 1024 x 1024 x 64
- Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
- Maximum memory pitch: 2147483647 bytes
- Texture alignment: 512 bytes
- Clock rate: 1.40 GHz
- Concurrent copy and execution: Yes
- Run time limit on kernels: No
- Integrated: No
- Support host page-locked memory mapping: Yes
- Compute mode: Default (multiple host threads can use this device simultaneously)
- Concurrent kernel execution: Yes
- Device has ECC support enabled: No
- Device is using TCC driver mode: No

**Device 1: "GeForce 210"**
NVIDIA GeForce GTX 480
Need to identify those parts of the program that operate on the host (CPU) and the device (GPU).
First CUDA Program

Perform element-wise vector addition, with each vector element being handled by one thread

```
#include <stdio.h>
#include <cuda.h>

// Kernel that executes on the CUDA device. This is executed by ONE
// stream processor
__global__ void vec_add(float* A, float* B, float* C, int N) {
    // What element of the array does this thread work on
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```
// main routine that executes on the host
int main(void)
{
    int n;
    int N = 1000000;
    size_t size = N * sizeof(float);

    // Allocate in HOST memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize vectors
    for (n = 0; n < N; ++n) {
        h_A[n] = 3.2333 * n;
        h_B[n] = 8.09287 * n;
    }
}
// Allocate in DEVICE memory (note the address of pointer argument)
float *d_A, *d_B, *d_C;
cudaMalloc(&d_A, size);
cudaMalloc(&d_B, size);
cudaMalloc(&d_C, size);

// Copy vectors from host to device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
vec_add<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

// Copy result from device memory into host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

// Free host memory
free(h_A);
free(h_B);
free(h_C);

Build using cuda compiler and linker
$ nvcc -o testprog1 testprog1.cu
$ ./testprog1
$ cd /usr/local/cuda/compute prof/bin
$ ./compute prof
Important Constructs

- **Important Functions**

  ```c
  cudaMalloc(device_address, size);
  cudaMemcpy(dest, source, size, cudaMemcpyHostToDevice)
  cudaMemcpy(dest, source, size, cudaMemcpyDeviceToHost)
  ```

- **Function modifier keywords**

  - `__global__`: called by host, executed on device
  - `__device__`: called by and executed on device
  - `__host__`: called by and executed on host

- **Kernel Functions**

  Code to be run on an SP

  ```c
  mykernel<<<blocks_per_grid, threads_per_block>>>
  ```
- **host** (CPU) and **device** (GPU)
- **thread**
  - concurrent code executed on an SP
  - fine grain unit of parallelism
- **warp**
  - group of threads executed in parallel (up to a maximum number)
- **block**
  - group of threads executed together and form the unit of resource assignment
- **grid**
  - group of thread blocks that must all complete before control is returned to the host
Organizing threads

- Threads in a warp can share instruction stream
- Each thread has its own registers and local memory
- Threads in a block can communicate by shared memory
- All threads in a grid can access the same global memory (but 200–600 cycle penalty)
- Need to decide how many blocks in the grid, and how many threads in each block.
- Can arrange blocks and threads in 1, 2, or 3 dimensions
// Matrix multiplication kernel: C = A * B
__global__ void mx_mult(float* A, float* B, float* C, int width)
{
    // What is the matrix element for this thread?
    int col = blockDim.x * blockIdx.x + threadIdx.x;
    int row = blockDim.y * blockIdx.y + threadIdx.y;

    float sum = 0;
    for (int k = 0; k < width; ++k) {
        float elementA = A[row * width + k];
        float elementB = B[k * width + col];
        sum += elementA * elementB;
    }
    C[row * width + col] = sum;
}

int main(void) {
    ...
    // 2 dimensional arrangement of threads and blocks
    int blockWidth = 30;
    int gridWidth = 15;
    dim3 dimBlock(blockWidth, blockWidth);
    dim3 dimGrid(gridWidth, gridWidth);
    mx_mult<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, width);
    ...
}
CUDA Device Memory Types

- **GlobalMemory**
  - largest memory on GPU and accessible by all threads
  - slowest access time - $\sim$200–600 clock cycles
  - lifetime: application

- **Registers**
  - fastest memory, used to store local variables of a single thread
  - lifetime: thread

- **Local memory**
  - section of device memory used when variables of a thread do not fit the registers available
  - lifetime: thread

- **Shared memory**
  - fast on chip memory shared between all threads of a single block
  - lifetime: block

- **Texture memory**
  - a cached region of global memory
  - each SM has its own texture memory cache on chip
  - lifetime: application

- **Constant memory**
  - a cached read-only region of device memory on each SM
  - lifetime: application
GPU bundles several threads together for execution into "warps"

Thread index values within a warp are contiguous. For warp size of 32 (eg GTX480) we have

```
threadIdx.x 0 → 31 in warp 0
threadIdx.x 32 → 63 in warp 1
```

... 

Bit more complicated for multidimensional thread organization
Branching

- **Single Instruction Multiple Thread**
  - executes instruction for all threads in the warp, before moving onto next instruction

- **Divergence** occurs when threads in a warp follow different control flows. Sequential passes are then needed which can affect performance

- Also be careful of conditionals based on thread ID
DRAM memory access patterns
- **Fast**: accessing data from multiple and contiguous locations
- **Slow**: truly random access

Ideal access pattern in GPUs
- all threads in a warp access consecutive global memory locations

**Coalescing memory access**
- hardware can combine all of these accesses into a single request

Non-coalescing memory access affects performance
1 Need for parallelism

2 Graphical Processor Units

3 Gravitational Microlensing Modelling
Work done by PhD student Joe Ling, Massey University (thesis due soon!)

- **Unroll small loops**
  - Reducing a few instructions per loop can add up to significant saving when performing the computation billions of time.
  - There is 8%-9% improvement in performance by just unrolling the lens equation in ray shooting.
Magnification Map generation

- Coalescing memory read/write has significant impact on performance.
- When writing to random memory position, atomic instruction is needed. For example, binning rays in rays shooting.
- Make sure there are enough blocks to hide the memory latency.
- Use as less registers per threads as possible in order to fit more blocks into a MP.
- Number of threads per block should be multiple of warp size.
- Use constant memory (pass as argument) instead of loading input parameters from global memory.
Magnification Map reading
- Texture memory should be used instead of global memory as memory reading is usually not coalescing
- Take advantage of locality as texture memory is cached

Dynamic light curve calculation
- Ray shooting sum can be done very quickly on the GPU but solving the image positions is usually faster by using the CPU.
- Minimize divergent warps.
- Our programming model: Multi-thread images solving code by CPU & ray shooting sum code by GPU.
Other Stuff

- Fermi is now implemented the IEEE 754-2008 floating-point standard.
- Fermi’s double precision arithmetic is 8 times slower then single precision on consumer hardware (1/2 in commercial hardware).
- But better precision can be achieved by shifting even with single precision arithmetic.
- Only commercial hardware has ECC check, it is disabled on consumer hardware.
Further Reading

- *Parallel Programming*, B. Wilkinson & M. Allen – a classic text on parallel programming. Deals mainly with cluster computing and message passing programming, but concepts in parallelizing numerical algorithms are still relevant to GPU programmers

- *Programming Massively Parallel Processors*, D.B. Kirk & W. W. Hwu – the definitive guide to CUDA and programming GPUs

- Ken’s paper

- RTFM!