GPGPUs and CUDA

Raj Singh
National Center for Microscopy and Imaging Research
What are we going to talk about?

- How did GPUs evolve?
- Why is the scientific community interested?
- Multi-core parallel architecture
- Programming and Memory models
- CUDA concepts and introduction
- Optimizations and gotchas
- How we use GPUs at NCMIR / CALIT2
Graphics Processing Unit (GPU)

- Development driven by the multi-billion dollar game industry
  - Bigger than Hollywood
- Need for physics, AI and complex lighting models
- Impressive Flops / dollar performance
  - Hardware has to be affordable
- Evolution speed surpasses Moore’s law
  - Performance doubling approximately 6 months
GPU evolution curve

![Graph showing the evolution of GPU performance from NV35 to GT200 with corresponding dates and model numbers.](image)

*Courtesy: Nvidia Corporation*
GPGPUs (General Purpose GPUs)

- A natural evolution of GPUs to support a wider range of applications
- Widely accepted by the scientific community
- Cheap high-performance GPGPUs are now available
  - It's possible to buy a $500 card which can provide a TeraFlop of computing.
Teraflop computing

- Supercomputers are still rated in Teraflops
  - Expensive and power hungry
  - Not exclusive and have to be shared by several organizations
  - Custom built in several cases
- National Center for Atmospheric Research, Boulder installed a 12 Tflop supercomputer in 2007
What does it mean for the scientist?

- Desktop supercomputers are possible
  - No sharing
  - No network latency. Good for real-time applications
- Very efficient
  - Approx 200 Watts / Teraflop
- Turnaround time can be cut down by magnitudes.
  - Simulations can take several days
GPU hardware

• Highly parallel architecture
  – SIMD

• Designed initially for efficient matrix operations and pixel manipulations pipelines

• Computing core is lot simpler
  – No memory management support
  – 32-bit native cores
  – Little or no cache
  – Largely single-precision support. Though its changing.
Multi-core Horsepower

- Latest Nvidia card has 240 cores for simultaneous processing
- Very high memory bandwidth
  - > 100 GBytes / sec and increasing
- Perfect for embarrassingly parallel compute intensive problems
- Cluster of GPGPUs are possible

Figures courtesy: Nvidia programming guide 2.0
CPU v/s GPU
Some Nomenclature

• **Device** = GPU
• **Host** = PC on which the code is executed and has a physical connection to the device
• **Cores / SPU** = Individual processing unit on GPUs
• **Kernel** = The function/code that is executed on the device
• **Global Memory** = Where most of the data structures for the kernel are stored on the device
Programming model

- The GPU is seen as a **compute device** to execute a portion of an application that
  - Has to be executed many times
  - Can be isolated as a function
  - Works independently on different data
- Such a function can be compiled to run on the **device**. The resulting program is called a Kernel
  - C like language helps in porting existing code.
- Copies of kernel execute simultaneously as threads.

Figure courtesy: Nvidia programming guide 2.0
Programming model

• Data is copied to the GPUs for initializations
• Kernel is invoked
• Result is copied back to host memory.
  – Graphical output can be shown directly in OpenGL contexts.
• Unfortunately currently the API forces the programmer to be aware of the underlying architecture
Look Ma no cache ..

- Cache is expensive
- By running thousands of fast-switching light threads large memory latency can be masked
- Context switching of threads is handled by CUDA
  - Users have little control, only synchronization
CUDA (Compute Unified Device Architecture)

- A non-OpenGL oriented API to program the GPUs
- Compiler and tools allow porting of existing C code fairly rapidly
- Libraries for common math functions like trigonometric, pow(), exp()
- Provides support for general DRAM memory addressing
  - Scatter / gather operations
Threads, Blocks and Grids

- Parallel threads execute on different portions of the data.
- Threads are batched as Blocks.
- Threads in a block access the fast shared memory between them and can synchronize.
- Since number of threads per block is limited, Grids of blocks can be executed
  - Future devices support more blocks.
- Threads in different blocks cannot synchronize or use shared memory.
Memory Model

- A thread that executes on the device has only access to the device’s DRAM and on-chip memory through the following memory spaces:
  
  - Read-write per-thread *registers*,
  
  - Read-write per-thread *local memory*,
  
  - Read-write per-block *shared memory*,
  
  - Read-write per-grid *global memory*,
  
  - Read-only per-grid *constant memory*,
  
  - Read-only per-grid *texture memory*. 
Memcpy operations

- Host memory to device memory
- Device memory to host memory
- CUDA memcpy functions can only be invoked from the host.
  - Not possible from inside a kernel
- There are restrictions on which memory sections on the device can be accessed
Function Type Qualifiers

- **__device__**
  - Executed on the device
  - Callable from the device only.

- **__global__**
  - Executed on the device
  - Callable from the host only.

- **__host__**
  - Executed on the host
  - Callable from the host only.
Variable Type Qualifiers

__device__
- Resides in the global memory space
- Has the lifetime of an application
- Accessible from all threads within a grid and from the host through the runtime library.

__constant__
- Resides in constant memory space
- Has the lifetime of an application
- Accessible from all threads within a grid and from the host through the runtime library

__shared__
- Resides in shared memory space
- Has the lifetime of the block
- Is only accessible from all threads within a block
NVCC – The compiler

- Comes with the toolkit
- Allows compilation of a subset of C into device binary objects
  - Cuda code files need to have a .cu extension
  - Linked as C functions to other C/C++ code
- Supports emulation mode
- Supports optimization flags and directives
Simple ‘Hello World’

main.cu

```c
#include <stdio.h>
#include <sys/time.h>
#include <cutil.h>

extern __global__ void kernel_func(float);

int main(int argc,char *argv[])
{
    dim3 grids(1, 1);
    dim3 block(1, 1);

    float inFloat, result;
    void *resultPtr;

    CUT_DEVICE_INIT(); // function provided with CUDA toolkit

    printf("Enter float : ");
    scanf("%f", &inFloat);

    // invoke kernel
    kernel_func <<< grids, block >>> (inFloat);

    // copy back the result
    cudaMemcpy((void *)&result, (void *)resultPtr, sizeof(float), cudaMemcpyDeviceToHost);

    printf("Input = %f and Result = %f
", inFloat, result);

    // check if kernel execution generated an error
    CUT_CHECK_ERROR("Kernel execution failed");

    CUT_EXIT(argc, argv); // function provided with CUDA toolkit
    return 0;
}
```

kernel.cu

```c
#include <stdio.h>
#include <cutil.h>

__device__ float floatVal_d;

__global__ void kernel_func(float val)
{
    floatVal_d = fabsf(val);
    return;
}
```
More practical real world case

main.c

```c
#include <stdio.h>
#include <sys/time.h>

extern float compute_on_gpu(float);

int main(int argc, char *argv[]) {
    float inFloat
    printf("Enter float : ");
    scanf("%f", &inFloat);
    result = compute_on_gpu(inFloat);
    printf("Input = %f and Result = %f\n", inFloat, result);
    return 0;
}
```

kernel.cu

```c
#include <stdio.h>
#include <cutil.h>
__device__ float floatVal_d;
void *resultPtr;

float compute_on_gpu(float in) {
    float result;
    dim3 grids(1, 1);
    dim3 block(1, 1);
    CUT_DEVICE_INIT(); // function provided with CUDA toolkit
    // invoke kernel
    kernel_func <<< grids, block >>> (in);
    // check if kernel execution generated and error
    CUT_CHECK_ERROR("Kernel execution failed");
    // copy back the result
    cudaGetSymbolAddress(&resultPtr, "floatVal_d");
    cudaMemcpy((void *)&result, (void *)resultPtr, sizeof(float), cudaMemcpyDeviceToHost);
    return result;
}

__global__ void kernel_func(float val) {
    floatVal_d = fabsf(val);
    return;
}
```

GPGPUs and CUDA

GPGPUs and CUDA

Guest Lecture, CSE167, Fall 2008
# Handling multiple threads in the kernel

```c
#include <stdio.h>
#include <cutil.h>

__device__ char * imageBuffer[4096];

__global__ void kernel_func(int imageW, int imageH)
{
    // first thing find out the index of the thread
    unsigned int threadIndex = threadIdx.x + threadIdx.y * blockDim.y;
    unsigned int totalNoOfThreads = blockDim.x * blockDim.y;

    unsigned int bufSize = imageW * imageH;

    for(int i = threadIndex; i < bufSize; i += totalNoOfThreads)
    {
        // Do the main pixel related computation
        imageBuffer[i] ++;
    }

    // synchronize all threads before quitting
    __syncthreads();
    return;
}
```
Optimization Techniques

• 3 main rules to optimize
  – Optimize memory transfers
  – Maximize processor occupancy
  – Maximize arithmetic intensity
    • recompute instead of loading (counter intuitive)
    • reduce data interaction v/s computation

• Visual Profiler

• Use shared memory
  – Each Block of threads has 16 KB of shared mem accessible at register speeds
  – use shared memory versions of variables since access to shared memory is faster
  – after you are done with data, store it back in the global memory
Coalesced memory access

- Be aware of the memory access patterns of the cores.
  - Split tasks in a way the data access for each lock step of threads is spatially closer
- Non-coalesced memory access can hurt performance by several factors
- Can be counter-intuitive
What do we do at NCMIR / CALIT2?

- Research on large data visualization, optical networks and distributed system.
- Collaborate with Earth sciences, Neuroscience, Gene research, Movie industry.
- Large projects funded by NSF / NIH.
- Basically we play with a lot of toys ;)

NSF EarthScope
Instruments Generating A Lot of Data
Osaka’s 3Mev Microscope
Cluster Driven High-Resolution displays
Electron Tomography

- Used for constructing 3D view of a thin biological samples
- Sample is rotated around an axis and images are acquired for each ‘tilt’ angle
- Electron tomography enables high resolution views of cellular and neuronal structures.
- 3D reconstruction is a complex problem due to high noise to signal ratio, curvilinear electron path, sample deformation, scattering, magnetic lens aberrations…
Challenges

- Use a Bundle Adjustment procedure to correct for curvilinear electron path and sample deformation
- Evaluation of electron micrographs correspondences needs to be done with double precision when using high-order polynomial mappings
- Non-linear electron projection makes reconstruction computationally intensive.
- Wide field of view for large datasets
  - CCD cameras are up to 8K x 8K
Reconstruction on GPUs

• Large datasets take up to several days to reconstruct on a fast serial processor.
• Goal is to achieve real-time reconstruction
• Computation is embarrassingly parallel at the tilt level
• GTX 280 with double-precision support and 240 cores has shown speedups between 10X – 50X for large data
• Tesla units with 4Tflops are the next target for the code.
Really? Awesome free Lunch?

• **C-like** language support
  – Missing support for function pointers, recursion, double precision not very accurate, no direct access to I/O
  – Cannot pass structures, unions

• **Code has to be fairly simple and free of dependencies**
  – Completely self contained in terms of data and variables.

• **Speedups depend on efficient code**
  – Programmers have to code the parallelism.
    • No magic spells available for download
  – Combining CPU and GPU code might be better in cases
And more cons …

• Performance is best for computation intensive apps.
  – Data intensive apps can be tricky.
• Bank conflicts hurt performance
• It’s a black-box with little support for runtime debugging.
Resources

- [http://www.gpgpu.org](http://www.gpgpu.org)
Q ‘n A

Thanks and you can contact me at raj@ncmir.ucsd.edu