Introduction to NVIDIA CUDA

Why Massively Parallel Processor

- A quiet revolution and potential build-up (GRD numbers)
- Calculation: 544 GFLOPS vs. 264 GFLOPS (FP-64)
- Memory Bandwidth: 153.6 GB/s vs. 256 GB/s
- Until recently, programmed through graphics API
- GPU in every PC and workstation - massive volume and potential impact

Future Apps in Concurrent World

- Exciting applications in future mass computing market
  - Molecular dynamics simulation
  - Video and audio coding and manipulation
  - 3D imaging and visualization
  - Consumer game physics
  - Virtual reality products
- Various granularities of parallelism exist, but...
  - Programming model must not hinder parallel implementation
  - Data delivery needs careful management
- Introducing domain-specific architecture
  - CUDA for GPGPU
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 (single-instruction multiple-data) 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

GPU and CPU: The Differences

- **GPU**
  - More transistors devoted to computation, instead of caching or flow control
  - Suitable for data-intensive computation
  - High arithmetic/memory operation ratio

CUDA

- "Compute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
    - GPU - dedicated, super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation program into GPU
  - Standalone Driver - Optimized for computation
    - Guaranteed maximum download & readback speeds
    - Explicit GPU memory management
CUDA Programming Model

- 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
    - Hardware switching between threads (in 1 cycle) on long-latency memory reference
  - Overprovision (1000s of threads) → hide latencies
- 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

Thread Batching: Grids and Blocks

- Kernel executed as a grid of thread blocks
  - All threads share data memory space
- Thread block is a batch of threads, can cooperate with each other by:
  - Synchronizing their execution
  - For hazard-free shared memory access
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
  - (Unless thru slow global memory)
- Threads and blocks have IDs

Extended C

- Declspec
  - __device__ float filter(N);
  - __global__ void convolve (float *image) {
    __shared__ float region(M);
    ...}
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads();
- Runtime API
  - Memory, symbol, execution management
  // Allocate GPU memory
  void *image = cudaMalloc(nbytes);
  // 100 blocks, 10 threads per block
  cudaMemcpy(cudaImage, 100 * 10 * nbytes);
### CUDA Function Declarations

<table>
<thead>
<tr>
<th>Function</th>
<th>Executed on:</th>
<th>Only callable from:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>Host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>Host</td>
<td>Host</td>
</tr>
</tbody>
</table>

- __global__ defines a kernel function
  - Must return void
- __device__ and __host__ can be used together

### 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, Constant, and Texture Memories (Long Latency Accesses)

- 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
Calling Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

```c
__global__ void KernelFun(...) {
    dim3 DimGrid(100, 50); // 5000 thread blocks
    dim3 DimBlock(4, 8, 8); // 256 threads per block
    size_t SharedBytes = 64; // 64 bytes of shared memory
    KernelFun<<<DimGrid, DimBlock, SharedBytes>>>(...);
}
```

- Any call to a kernel function is asynchronous (CUDA 1.0 & later), explicit sync needed for blocking
- Recursion in kernels supported (in 5.0/Kepler+)

Sample Code: Increment Array

```c
model_t init_y(int x, int a, int b, size_t x_size = 16*1024); // A single thread initializes
model_t init_y(int x, int a, int b, size_t x_size = 16*1024) {
    int y = 1;
    for (int i = 0; i < x_size; i++)
        y += i;
    return y;
}
```

Execution model

- Multiple levels of parallelism
  - Thread block
    - Max. 1024 threads/block
    - Communication through shared memory (fast)
    - Thread guaranteed to be resident
    - threadIdx, blockIdx
    - __syncthreads()
    - barrier for this block only
      avoid RAW/WAR/WAW hazards when ref shared/global memory
  - Grid of thread blocks
    - F=blocks, nthreads=(a, b, c)
Compiling CUDA

- Call nvcc (driver) — also C++/Fortran
- LLVM front end (used to be EdG)
  — Separate GPU & CPU code
- LLVM back end (used to be Open64)
  — Generates GPU TPX assembly
- Parallel Threads eXecution (PTX)
  — Virtual machine and ISA
  — Programming model
  — Execution resources and state
- Extensions
  — OpenACC: see ARC web page,
    like OpenMP but for GPUs
  — OpenCL (not covered here)

Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  — Serial C code executes on CPU
  — Parallel Kernel C code executes on GPU thread blocks

Hardware Implementation:
Execution Model

- Each thread block of a grid is split into warps, each gets executed by one multiprocessor (SM)
  — The device processes only one grid at a time
- Each thread block is executed by one multiprocessor
  — So that the shared memory space resides in the on-chip shared memory
- A multiprocessor can execute multiple blocks concurrently
  — Shared memory and registers are partitioned among the threads of all concurrent blocks
  — So, decreasing shared memory usage (per block) and register usage
    (per thread) increases number of blocks that can run concurrently
Threads, Warps, Blocks

- There are (up to) 32 threads in a Warp
  - Only <32 when there are fewer than 32 total threads
- There are (up to) 32 Warps in a Block
- Each Block (and thus, each Warp) executes on a single SM
- GF110 has 16 SMs
- At least 16 Blocks required to "fill" the device
- More is better
  - If resources (registers, thread space, shared memory) allow, more than 1 Block can occupy each SM

More Terminology Review

- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = group of SIMD threads that execute a kernel and can communicate via shared memory

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Aligned</th>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads in block</td>
</tr>
<tr>
<td>Thread</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>

Access Times

- Register - dedicated HW - single cycle
- Shared Memory - dedicated HW - single cycle
- Local Memory - DRAM, no cache - "slow"
- Global Memory - DRAM, no cache - "slow"
- Constant Memory - DRAM, cached, 1.10s..100s of cycles, depending on cache locality
- Texture Memory - DRAM, cached, 1.10s..100s of cycles, depending on cache locality
- Instruction Memory (invisible) - DRAM, cached
**Memory Hierarchy**

- Thread
  - Kernel
    - Per-thread local Memory
  - Host memory
    - cudaMemcpy()
- Block
  - Kernel
    - Per-device Global Memory
  - Device 0 memory
  - Device 1 memory

**Using per-block shared memory**

- Variables shared across block
  ```
  int *begin, *end;
  ```

- Scratchpad memory
  ```
  __shared__ int scratch[blocksize];
  scratch[threadIdx.x] = begin[threadIdx.x];
  // compute on scratch values -
  begin[threadIdx.x] = scratch[threadIdx.x];
  ```

- Communicating values between threads
  ```
  scratch[threadIdx.x] = begin[threadIdx.x];
  __syncthreads();
  int left = scratch[threadIdx.x - 1];
  ```

**Example: Parallel Reduction**

- Summing up a sequence with 1 thread:
  ```
  int sum = 0;
  for(int i=0; i<N; ++i) sum += a[i];
  ```

- Parallel reduction builds a summation tree
  ```
  - each thread holds 1 element
  - stepwise partial sums
  - N threads need log N steps
  - one possible approach: Butterfly pattern
  ```
Parallel Reduction for 1 Block

```c
// INPUT: Thread i holds value x_i
int i = threadIdx.x;
__shared__ int sum(blocksize);

// One thread per element
sum[i] = x[i] __syncthreads();

for(int bit=blocksize/2; bit>0; bit/=2)
  int t=sum[i]+sum[i+bit]; __syncthreads();
  sum[i]=t __syncthreads();

// OUTPUT: Every thread now holds sum in sum[i]
```

Parallel Reduction Across Blocks

- Code lets B-thread block reduce B-element array
- For larger sequences:
  - reduce each B-element subsequence with 1 block
  - write N/B partial sums to temporary array
  - repeat until done

- P.S. this works for min, max, *, and friends too
  - as written requires associative & commutative function
  - can restructure to work with any associative function

Language Extensions

- **Built-in Variables**
  - dim3 gridDim;
    - Dimensions of the grid in blocks (gridDim.z unused)
  - dim3 blockDim;
    - Dimensions of the block in threads
  - dim3 blockIdx;
    - Block index within the grid
  - dim3 threadIdx;
    - Thread index within the block

- **Math Functions**
  - sin, cos, tan, asin, ...
  - Math device functions:
    - __sin, __cos (faster, less accurate)
  - Atomic device functions:
    - atomicAdd(), atomicCAS(), ...
    - Can implement locks

In Kernel Memory Management
- malloc()
- free()
### Tesla Architecture

- Used for Technical and Scientific Computing
- L1/L2 Data Cache
  - Allows for caching of global and local data
  - Some on-chip memory used for Shared and L1
  - Configurable at kernel invocation

### Fermi Architecture

- L1 cache for each SM
  - Shared memory/L1 use same memory
  - Configurable partitions at kernel invocation
- 45KB shared/16KB L1 or 32KB shared/4KB L1
- Unified 768KB L2 Data Cache
  - Services all load, store, and texture requests

### Kepler Architecture

- GK104/K10 early 2012
  - Configurable shared memory access bank width: 4 / 8 bytes
    - cudaDeviceSetSharedMemConfig
- GK104/K20 (late 2012)
  - Dynamic parallelism, HyperQ, more regs/thread & DP throughput
CUDA Toolkit Libraries

- NVIDIA GPU-accelerated math libraries:
  - cuFFT - Fast Fourier Transforms Library
  - cuBLAS - Complete BLAS library
  - cuSPARSE - Sparse Matrix library
  - cuRAND - Random Number Generation (RNG) Library
- Performance improved since 3.1

- CULA - linear algebra library (commercial add-on)
  - Single precision version free, double costs $x
- Thrust: C++ template lib -> STL-like
  - Boost-like aside:
    - thrust::transform(x.begin(), x.end(), y.begin(), y.begin(), a * 1 + 2);

Libraries & More

- Object linking
  - Plug-ins, libraries
- Dynamic parallelism
  - GPU threads can launch new kernels
- RDMA from GPU(node1) -> GPU(node2)

Tools

- Visual Profiler
  - Where is the time spent?
- CUDA-gdb: debugger
- Parallel Nsight • Eclipse
  - Debugger
  - Memory checker
  - Traces (GPU vs. GPU activity)
  - Profiler (memory, instruction throughput, std)
- Nvidia-smi
  - Turn off ECC
  - Read performance counters
Timing CUDA Kernels

- Real-time Event API
  ```
cudadEvent_t cstart, cstop;
  float cdiff;
  cudadEventCreate(&cstart);
  cudadEventCreate(&cstop);
  cudadEventRecord(cstart, 0);
  kernel<<<x, y, z>>>(a, b, c);
  cudadEventRecord(cstop, 0);
  cudadEventSynchronize(cstop, cstart, cstop);
  printf("CUDA time is %lf usec\n", cdiff);
  cudadEventDestroy(cstart);
  cudadEventDestroy(cstop);
  ```

Device Capabilities

- Need to compile for specific capability when needed
  - Flags in Makefile
- Capability levels:
  - 1:0: basic GPU (e.g., 8800 GTX)
  - 1:1: 32-bit atomic in global memory (e.g., GTX 280)
  - 1:2: 64-bit atomic in global-shared memory, warp voting
  - 1:3: double precision floating point
    - e.g., GTX 280/GTX 480, C2050/C2070, C2060/C2070
  - 2:0: caches for global-shared memory
    - e.g., GTX 480, C2050/C2070
  - 2:0: more wraps, threads, blocks, registers
    - e.g., GTX 680
  - 3:5: Dynamic parallelism, HyperQ
    - e.g., Tesla K20}

OpenACC

- Pseudo-based industry standard, the “OpenMP for GPUs”, V1.0
  ```
  #pragma acc (clause)
  ```
- For GPUs but also other accelerators
- For CUDA but also OpenCL...
- Data movement: sync/async
- Parallelism
- Data layout and caching
- Scheduling
- Mixes w/ MPI, OpenMP
- Works with C, Fortran

---

OpenACC

**Directives for Accelerators**
OpenACC Kernel Example

- CPU
  void domain...{
  #pragma acc data 
  copy(y[0:n], y[0:n])
  
  saxpy( n, a, x, y );
  }

- GPU
  void saxpy( int n, float a,
              float* x, float*
             restrict y )
  int i;
  #pragma acc kernels loop 
  present(x[0:n], y[0:n])
  for( i = 1; i < ng + 1i )
    y[i] += a*x[i];

OpenACC Execution Constructs

- kernels [classes.] in [structured block]
  - Run kernel on GPU
  - if (cond) only exec if cond is true
  - async: do not block when done

- Loop [classes.]
  - run iterations of loop on GPU
  - collapse(n) for next n loop nests
  - seq: sequential execution!
  - private (list): private copy of vars
  - firstprivate (list): copy in private
  - reduction (op:lis): +* &
  - min/max
  - gang/worker: scheduling options
  - vector: SIMD mode
  - independent: iterations w/o hazards

OpenACC Data Constructs

- data [classes.] in [structured block]
  - Declare data for GPU memory
  - async: as before

  Cluster:
  - copy( list ) Allocates list on GPU,
    copies data CPU→GPU when entering
    kernel and GPU→CPU when done
  - copy( list ) same but only
    CPU→GPU
  - copyarray( list ) same but only
    GPU→CPU
  - create list only allocate
  - present list ) data already on GPU (no
    copy)
  - present_or_copy(in/out) list ) if not
    present then copy [in/out]
  - present_or_create list ) if not present
    then allocate
  - device_ptr list ) List pointers of device
    addresses, such as from acc_malloc.
OpenACC Update

- CPU
  ```c
  for( intstep=0; ... ){
    ...compute...
    
    MPI_SENDBEV( x, ... )
    MPI_RECVBEV( x, ... )
    ...adjust...
  }
  ```

- GPU
  ```c
  #pragma acc data copy(x[0:n])
  for( intstep=0; ... ){ 
    ...compute on device...
    #pragma acc update host(x[0:n]) -> CPU
    #pragma acc update device(x[0:n])
    ...adjust on device
  }
  ```

OpenACC Async

- CPU
  ```c
  void domany(...){
    #pragma acc data \ create(x[0:n],y[0:n])
    { int i;
      #pragma acc update device \ #pragma acc kernels loop async
      (x[0:n], y[0:n]) async
      for( i = 0; i < n; ++i )
        x[i] = y[i] + z[i];
    }
  }
  ```

- GPU
  ```c
  void sapply( int n, float x, float* y, float* restrict y ){
    int i;
    for( i = 0; i < n; ++i )
      y[i] = x*y[i];
  }  
  ```

OpenACC Data Caching

- Uses shared memory (SM / scratch pad memory)
  ```c
  #pragma acc kernels loop present(i:0:[x[i]:y[i]], z[i]:x[i-2:j]=1)
  for( i = 0; ... ){
    #pragma acc cache( b[]:l=1+i:2[j]=1 )
    for( i = 0; ... ){
      a[i][j] = b[i][j] +
        w * ( b[i-1][j] + b[i+2][j] + b[i][j-1] + b[i][j+1])
    }
  }
  ```
OpenACC Parallel / Loop (for)

- GPU Parallel
  
  ```c
  #pragma acc parallel
  void saxpy(int n, float* x, float* y)
  restrict y)
  int i;
  
  #pragma acc loop
  for (i = 1; i < n; ++i)
  y[i] += a*x[i];
  ```

OpenACC Runtime Constructs

- #include "openacc.h"
- acc_malloc(size_t)
- acc_free(void*)
- acc_async_wait(expression)
- acc_async_wait_all()
- acc_async_wait_all()

Cray OpenACC

This Feedback Loop Unique to Compliers!

We can use this same methodology to enable effective migration of applications to Multi-core and Accelerators.