Programming with GPUs - CUDA and OpenCL

Rohith Goparaju
Devarshi Ghoshal
GPUs- the beginning

• Design more focused on data processing rather than data caching & flow control

• Well suited to address problems that can be expressed as data-parallel computations

• The same program is executed on many data elements and hence low requirement for sophisticated flow control

• Since it is executed on many data elements & has high arithmetic intensity, the memory latency can be hidden with calculations instead of big data caches

• Data parallel processing maps data elements to parallel processing threads to speed up the computations
Why CUDA?

- The GPU could only be programmed through a graphics API (inadequate to non-graphics application)

- The GPU DRAM could be read in a general way - programs can gather data elements from any part of DRAM but could not be written in a general way because they cannot scatter information to any part of DRAM

- Bottlenecked by the DRAM memory bandwidth (since centralized)

*CUDA - Novel hardware & programming model exposing GPU as a truly generic data-parallel computing device.*
CUDA- Compute Unified Device Architecture

• A new hardware & software architecture for issuing & managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API

• Software stack is composed of several layers: a hardware driver, an API & its runtime, and 2 higher level math libraries CUFFT & CUBLAS

• Hardware supports lightweight driver & runtime layers

• Parallel data cache or on-chip shared memory

• Available for the GeForce 8 Series, Quadro FX 5600/4600, and Tesla solutions
CUDA Software Stack

Compute Unified Device Architecture Software Stack
Multithreaded coprocessor

• GPU viewed as a compute device capable of executing a very high no. of threads in parallel and operates as a coprocessor to the main CPU or host

• Kernel Function: a portion of a program that is executed many times but independently on different data

• Both the host & device maintain their own DRAM referred to as host memory & device memory and one can copy data from one DRAM to the other utilizing device’s high-performance DMA engines
CUDA- Gather Scatter Memops

The **Gather** and **Scatter** Memory Operations
Shared Memory

Without shared memory

With shared memory

Shared Memory Brings Data Closer to the ALUs
Thread Batching

• Thread block: a batch of threads, each identified by its thread-id, that can cooperate together by efficiently sharing data through some fast shared memory & synchronizing their execution to coordinate memory accesses

• Grid of blocks: blocks of same dimensionality and size that execute the same kernel are batched together

• Reduced thread cooperation because threads in different thread blocks from the same grid cannot communicate & synchronize with each other

• Allows kernels to run efficiently without recompilation on various devices with different parallel capabilities
Thread Batching

The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks.
Memory Model

- A thread executing on the device has only access to the device’s DRAM and on-chip memory.

- 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
Programming Pattern

• Local and global memory reside in device memory (DRAM) - much slower access than shared memory
• So, a profitable way of performing computation on the device is to block data to take advantage of fast shared memory:
  – Partition data into data subsets that fit into shared memory
  – Handle each data subset with one thread block by:
    ▪ Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    ▪ Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    ▪ Copying results from shared memory to global memory
Programming Pattern (contd.)

• Texture and Constant memory also reside in device memory (DRAM) - much slower access than shared memory
  – But... cached!
  – Highly efficient access for read-only data

• Carefully divide data according to access patterns
  – R/O no structure  constant memory
  – R/O array structured  texture memory
  – R/W shared within Block  shared memory
  – R/W registers spill to local memory
  – R/W inputs/results  global memory
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
Hardware Model

• The device is a set of 16 multiprocessors, where each one has an SIMD architecture.

• Each multiprocessor has on-chip memory of 4 types:
  – One set of local 32-bit registers per processor
  – A parallel data cache or shared memory implements the shared memory space
  – A read-only constant cache reads from the constant memory space
  – A read-only texture cache reads from the texture memory space

• At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a “warp”

• The number of threads in a warp is the “warp size”
GeForce 8800 Series Technical Specs

• Maximum number of threads per block: 512
• Maximum size of each dimension of a grid: 65,535
• Number of streaming multiprocessors (SM):
  - GeForce 8800 GTX: 16 @ 675 MHz
  - GeForce 8800 GTS: 12 @ 600 MHz
• Device memory:
  - GeForce 8800 GTX: 768 MB
  - GeForce 8800 GTS: 640 MB
• Shared memory per multiprocessor: 16KB divided in 16 banks
• Constant memory: 64 KB
• Warp size: 32 threads (16 Warps/Block)
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 processed by only one multiprocessor
  - 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
Compilation with NVCC

• NVCC- Compiler driver that simplifies the process of compiling CUDA code
• Separates device code from host code
• Compiles device code into binary form- cubin
• Cubin object
  – Load the cubin object onto the device and launch the device code using the CUDA driver API
  or
  – Link to the generated host code, which includes the cubin object as a global initialized data array
Performance

• Instruction Throughput
  – Arithmetic instructions: 4 – 16 clock cycles
  – Control flow instructions: 4 – 7 clock cycles
  – Memory instructions: 4 – 600 clock cycles
  – Synchronization instruction: 4 clock cycles

• Memory Bandwidth
  – Device memory is of much higher latency and lower bandwidth than on-chip memory
  – Global memory- no cache
  – Shared memory- bank conflicts
Example- Matrix Multiplication

Each thread block computes one sub-matrix $C_{sub}$ of $C$. Each thread within the block computes one element of $C_{sub}$.
OpenCL
(Open Computing Language)
Introduction

• A framework for writing programs that execute across a heterogeneous platform.
• CPU’s, GPU’s and other processors as peers.
• A language based on C99.
• Data and Task parallel model.
• OpenCL gives access to GPU for non graphical computations.
OpenCL Objects

• Compute Devices
• Memory Objects
  – Arrays
  – Images
• Executable Objects
  – Compute Program
  – Compute Kernel
Devices

• Device Object is some kind of a processor that executes parallel programs.

• Each Device can have more than one Processing element.

• Host – Group of Devices.

• Processing elements execute programs in SIMD or SPMD.
Memory Objects

• Arrays
  – Work like arrays in C.
  – Array read/write on CPU is cached.

• Images
  – Data is stored in an optimized non-linear format.
  – Reads use texture cache.
Compute Kernel

• A data parallel function executed by the compute object (CPU or GPU).

Listing:

```c
#include <global.h>

__kernel void sum(__global const float *a,
                 __global const float *b,
                 __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```
Compute Program

• A group of kernels and functions.

```plaintext
__kernel void sub{...}
__kernel void transpose{...}
float cross_product{...}
...
__kernel void fft_radix2{...}
```
Expressing Data Parallelism

• A unit of work is called a work item.
• Work items are grouped into a work group.

NDRange Size = **Global Size**
Work Group Size = **Local Size**
Expressing Data Parallelism.

• Kernels execute across a global domain of work items.
  – Global Dimensions define the range of computation.
  – One work-item per computation executed in parallel.

• Work Items are grouped in local work groups
  – Local Dimensions define the size of the work groups
  – Execute together on one device.
  – Share local memory.
Work Items and Work Group Functions

get_work_dim = 1
get_global_size = 26

input
6 1 1 0 9 2 4 1 1 9 7 6 1 2 2 1 9 8 4 1 9 2 0 0 7 8

workgroups
get_group_id = 0
get_num_groups = 2
get_local_size = 13
get_local_id = 8
get_global_id = 21
Synchronization.

- No global Synchronization
- Synchronization can be done within a work group.
Expressing Task Parallelism

- Executes as a Single Work Item.
- A kernel in OpenCL C or a Function.
- A task owns a core.
- Benefits from large private/local memory.
Address Space.

• There are four types of address space.
  
  – **Private** (CUDA Local)
    – Per Work Item
  
  – **local** (CUDA Shared)
    – Shared within a workgroup
  
  – **constant** (CUDA Constant)
    – Not Synchronized.
  
  – **global** (CUDA Global)
    – Host Memory.
Address Space
OpenCL Execution.

- There are five main steps to run an OpenCL Application.
  - Initialization
  - Allocate Resources
  - Creating Programs/Kernels.
  - Execution
  - Tear Down.
Initialization/Setup

• Setup
  – Get the Device(s)
  – Create a Context
  – Create Command Queues

```c
cl_int err;
cl_context context;
cl_device_id devices;
cl_command_queue cmd_queue;

err = clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &devices, NULL);
context = clCreateContext(0, 1, &devices, NULL, NULL, &err);
cmd_queue = clCreateCommandQueue(context, devices, 0, NULL);
```
Initialization

• Devices
  – Multiple Cores on a CPU or GPU are a single device
  – OpenCL executes kernels across all devices in a data parallel manner

• Contexts
  – Enable sharing of memory between devices
  – To share between devices both devices must be in same context

• Queues
  – All work submitted through queues
  – Each device must have a queue
Allocation/Read & Write Memory

• Allocating Data

```c
cl_mem ax_mem = clCreateBuffer(context, CL_MEM_READ_ONLY, atom_buffer_size, NULL, NULL);
```

• Explicit Commands to access memory object data
  – Read from a region in memory object to host memory
    ```c
    clEnqueueReadBuffer(queue, object, blocking, offset, size, *ptr, ...)
    ```
  – Write to a region in memory object from host memory
    ```c
    clEnqueueWriteBuffer(queue, object, blocking, offset, size, *ptr, ...)
    ```
Read & Write Memory

• Similar Methods to copy regions of memory objects and map a region in memory object to host address space

• Operate Synchronously (\texttt{blocking = CL\_TRUE}) or Asynchronously
Creating Programs and Kernels

• Programs and kernels are read from a source compiled or loaded as a binary

```c
cl_program program[1];
cl_kernel kernel[1];

program[0] = clCreateProgramWithSource(context, 1,
   (const char**)&program_source, NULL, &err);

err = clBuildProgram(program[0], 0, NULL, NULL, NULL, NULL);
kernel[0] = clCreateKernel(program[0], "mdh", &err);
```
Program and Kernel Objects

• Program Object Encapsulates:
  – program source or a binary
  – list of devices and latest successfully built executable for each device
  – a list of kernel objects

• Kernel Object Encapsulates:
  – A specific kernel function in the program declared with the kernel qualifier
  – argument values
  – Kernel objects created after the program object has been built
Execution

- Arguments to the kernel are set and the kernel is executed on all data

```c
size_t global_work_size[2], local_work_size[2];
global_work_size[0] = nx; global_work_size[1] = ny;
local_work_size[0] = nx/2; local_work_size[1] = ny/2;

err = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), &ax_mem);
err = clEnqueueNDRangeKernel(cmd_queue, kernel[0], 2, NULL,
                            &global_work_size, &local_work_size,
                            0, NULL, NULL);
```

- Kernel is executed asynchronously
- Use events to track execution status
Tear Down

• The results are written back to the host and the memory is cleaned up

```c
err = clEnqueueReadBuffer(cmd_queue, val_mem, CL_TRUE, 0,
                          grid_buffer_size, val, 0, NULL, NULL);

clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_queue);
clReleaseContext(context);
```
Synchronization Between Commands

• Each individual queue can execute inorder or out of order.
  – For an inorder queue all commands execute in order

• Explicit synchronization between queues
  – Multiple Devices have their own queue
  – Use events to synchronize
Synchronization: One Device/Queue

- Example: Kernel 2 uses the results of kernel 1
Two Devices/Queues

- Explicit Dependency: Kernel 1 must finish before kernel 2 starts
Two Devices/Queues

Kernel 2 starts before the results from Kernel 1 are ready

Kernel 2 waits for an event from Kernel 1 and does not start until the results are ready
Using Events on The Host

- `clWaitForEvents(num_events, *event_list)`
  - Blocks until events are complete
- `clEnqueueMarker(queue, *event)`
  - Returns an event for a marker that moves through the queue
- `clEnqueueWaitForEvents(queue, num_events, *event_list)`
  - Inserts a "WaitForEvents" into the queue
- `clGetEventInfo()`
  - Command type and status
    - `CL_QUEUED`, `CL_SUBMITTED`, `CL_RUNNING`, `CL_COMPLETE`, or error code
- `clGetEventProfilingInfo()`
  - Command queue, submit, start, and end times