GPGPU COMPUTE ON AMD

Udepta Bordoloi
April 6, 2011
WHY USE GPU COMPUTE

CPU: scalar processing
- + Latency
- + Optimized for sequential and branching algorithms
- + Runs existing applications very well
- - Throughput

GPU: parallel processing
- + Throughput computing
- + High aggregate memory bandwidth
- - Latency
- ? Software

Serial/Task-Parallel Workloads
Graphics Workloads
Other Highly Parallel Workloads

CPU+GPU provides optimal performance combinations for a wide range of platform configurations
HETEROGENEOUS SYSTEM CONSIDERATIONS

- Where will the programs run?
  - CPU vs GPU

- Where do the data reside?
  - CPU vs GPU vs both

- When to communicate and synchronize between CPU and GPU?
  - Data transfer
  - Pipeline tasks
  - Sync points
AMD GPU Architecture
ATI 5800 SERIES (CYPRESS) GPU ARCHITECTURE

- New Compute Features:
  - Full Hardware Implementation of DirectCompute 11 and OpenCL™ 1.1
  - IEEE754 Compliance Enhancements
  - 32-bit Atomic Operations
  - 32kB Local Data Shares
  - 64kB Global Data Share
  - Global synchronization
  - Append/consume buffers
ATI 5800 SERIES (CYPRESS) GPU ARCHITECTURE

- 2.72 Teraflops Single Precision
- 544 Gigaflops Double Precision
- 153.6 GB/s memory bandwidth

- 5870 has 20 SIMDs

- Apart from registers, two options for fast memory in SIMD
  - 32KB Local (aka shared) memory
  - 8KB L1 Cache (aka texture) memory
SIMD Engine

Each SIMD:

- Has its own control logic and can run multiple batches of threads
- Comprises of 16 VLIW Thread Processing Units
  - Each Thread Processing Unit has 5 ALUs
- Has 8KB L1 cache accessed via a texture fetch unit
- Has 32KB Local Data Share
Wavefront

All threads in a “Wavefront” execute the same instruction

- 16 Thread Processing Units in a SIMD * 4 batches of 16-threads
  = 64 threads on same instruction (Cypress)

What if there is a branch?

1. First, full wavefront executes left branch, threads supposed to go to right branch are masked
2. Next, full wavefront executes right branch, left branch threads are masked

Try to avoid conditional code. (Case study: Lookup tables.)

OpenCL workgroup = 1 to 4 wavefronts on same SIMD

- Wavefront sizes not a multiple of 64 are inefficient!
LATENCY HIDING

- GPU memory latency is high compared to ALU rates
- When a wavefront needs to read in memory, it will stall on the SIMD until the memory arrives
- GPU hides the stall (potential SIMD idle time) by switching the SIMD onto another wavefront
- Hence, need more wavefronts than available SIMDS for efficient SIMD utilization
- Number of wavefronts that can be scheduled on a SIMD is dependent on per-wavefront resource (register, LDS) usage
Each Thread processing unit consists of
- 4 standard ALU cores, plus
- 1 special function ALU core
- Branch unit
- Registers

The GPU compiler tries to find parallelism to utilize all cores

Programmer can write code that fits well with the 4 standard ALU cores (say, using 128-bit data types such as a float4)
Programming Perspective (OpenCL™)
WHAT IS OPENCL™?

- Open, industry-standard specification
- Already supported by many vendors including AMD, Apple, IBM, Intel, Nvidia etc.
- Cross-platform, portable solution for running code on different target hardware and different OS.
  - Implementations exist for AMD and Nvidia GPUs, IBM Cell, x86 CPUs from AMD and Intel, Apple etc.
  - Windows, Linux, MacOS etc.
- How to run a program on the GPU (device)?
  - C language kernel
- How to control the GPU (device)?
  - C language API
OPENCL ARCHITECTURE

- **GPU execution**
  - Kernel: C like language
    - Performs GPU calculations
    - Reads from, and writes to GPU memory (simplistic view!)

- **GPU control**
  - Host program: C API (other bindings, e.g., C++)
    1. Initialize the GPU
    2. Allocate memory buffers on GPU
    3. Send data to GPU
    4. Run Kernel on GPU
    5. Read data from GPU
Kernel: OpenCL™ C Language

- Language based on ISO C99
  - Some restrictions, e.g., no recursion
- Additions to language for parallelism
  - Vector data types (e.g., int4, float2)
  - Work-items/group functions (e.g., get_global_id)
  - Synchronization (e.g., barrier)
- Built-in Functions (e.g., sin, cos)
Kernel: Work-item

- Define N-dimensional computation domain
  - N = 1, 2, or 3
  - Each element in the domain is called a **work-item**
  - N-D domain (**global dimensions**) defines the total number of work-items that will execute

- The kernel is executed for each work-item

C:

```c
for (int i=0; i<1000; i++)
{
    a[i] = i;
}
```

OpenCL kernel:

```c
for (int i=0; i<1000; i++)
{
    i = get_global_id(0);
    a[i] = i;
}
```
Kernel: Work-group

- Work-items are grouped into **work-groups**

- Work-items inside the same work-group
  - Execute on the same SIMD
  - Can share local memory and exchange data
  - Can synchronize using barrier calls

- Work-items in different work-groups
  - May or may not execute on the same SIMD
  - Cannot share local memory and cannot exchange data
  - Cannot synchronize using barrier calls

- How would you synchronize and exchange data between work-items in different work-groups?
Kernel: Work-group example

N (# of dimensions) = 2
Global size = 32 x 32
Work-group size = 8 x 8 = 64
Total # of work-items = 32 x 32 (assuming one pixel per work-item)
HOST PROGRAM: BASIC SEQUENCE

- Initialization
  - Find the GPU
  - Initialize the GPU
  - Compile the program for GPU (kernel)

- Memory
  - Create input, output buffers on the GPU
  - Copy data from CPU memory to GPU memory

- Execution
  - Run kernel on the GPU
  - Loop if needed
  - Wait till GPU is finished

- Memory
  - Copy data from GPU memory to CPU memory
HOST PROGRAM: TERMINOLOGY

- **Context** – OpenCL handles and resources will live inside a context.
- **Device** – The kernel code will execute on one or more devices. Can be GPU, CPU etc.
- **Command Queue** – Commands for memory, kernel execution etc. are pushed into one or more queues.
- **Memory objects** – Objects to hold the data. Can be of two types – buffers or images.
- **Kernels** – Code that executes on the device.
- **Program Objects** – Contains one or more kernels in source or binary form.
HOST PROGRAM: COMMAND QUEUE

- Enables **asynchronous** execution of OpenCL commands
  - Look for OpenCL commands `clEnqueue…()`
- Accepts:
  - Kernel execution commands
  - Memory commands
  - Synchronization commands
- Queued in-order
- Execute in-order (default) or out-of-order
EXAMPLE WALK THROUGH – KERNEL

```c
__kernel void vec_add (__global const float *a,
                      __global const float *b,
                      __global float *c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Compare to the C version:

```c
void vec_add (const float *a,
              const float *b,
              float *c)
{
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```
EXAMPLE – HOST PROGRAM (FIND PLATFORM)

    // get number of platforms available on this system
    status = clGetPlatformIDs(0, NULL, &numPlatforms);
    
    // search list of platforms for vendor name
    if (numPlatforms > 0) {
        platforms = (cl_platform_id *)
            malloc(numPlatforms*sizeof(cl_platform_id));
        status = clGetPlatformIDs(numPlatforms, platforms, NULL);
        
        for (unsigned i = 0; i < numPlatforms; i++) {
            char pbuf[100];
            status = clGetPlatformInfo(platforms[i],
                CL_PLATFORM_VENDOR, sizeof(pbuf), pbuf, NULL);
            platform = platforms[i];
            if (!strcmp(pbuf, "Advanced Micro Devices, Inc.")) break;
        }
        free(platforms); platforms = NULL;
    }
    if (platform == NULL) return -1;
    
    cps = {CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0};
EXAMPLE – HOST PROGRAM (INITIALIZATION)

// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(cps, CL_DEVICE_TYPE_GPU,
                                     NULL, NULL, &status);

// get the list of GPU devices associated with context
cGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
cGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                             sizeof(cl_float)*n, NULL, NULL);
EXAMPLE – HOST PROGRAM (COMPILE)

// create the program
program = clCreateProgramWithSource(context, 1, &program_source,
                                   NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);
EXAMPLE – HOST PROGRAM (EXECUTE)

// set the args values
err = clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2], sizeof(cl_mem));

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
        global_work_size,
        NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE, 0,
        n*sizeof(cl_float),
        dst, 0, NULL, NULL);
In Order Queue

Out of Order Queue

GPU

Images

Buffers

Kernels

Memory Objects

Programs

Context

CPU

GPU

__kernel void vec_add(const float *a, const float *b, float *c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}

vec_add

CPU program

binary

vec_add

GPU program

binary

arg[0] value

arg[1] value

arg[2] value

Compile code

Create data & arguments

Send to execution
OpenCL™ Memory Spaces

- **Private** – only accessible to one work-item. Fast access on GPU.

- **Local** – used for sharing data within one work-group. Read/write by all work-items in same work-group, not visible outside the work-group. Can function as user-managed cache on GPU.

- **Global** – read and write by all work-items. Higher latency access on GPU.

- **Constant** – read-only by work-items; initialized by host program.
**OPENCL™ MEMORY SPACE ON AMD GPU**

- Registers/LDS
- Thread Processor Unit
- SIMD
- Local Data Share
- Board Mem/Constant Cache
- Board Memory

Compute Device Memory

Global Memory

Global / Constant Memory Data Cache

Local Memory

Private Memory

Work Item 1

Work Item M

Private Memory

Work Item 1

Work Item M

Compute Unit 1

Compute Unit N
OPENCL VIEW OF AMD GPU

- Constants (cached global)
- Workgroups
- Image cache
- Global memory (uncached)
- Local memory (user cache)
- L2 cache