Parallel programming in OpenCL

Advanced Graphics & Image Processing

Rafał Mantiuk
Computer Laboratory, University of Cambridge
Consider the following vector addition example

Serial program:
one program completes
the entire task

Multiple copies of the same program execute on different data in parallel

SPMD program:
multiple copies of the
same program run on
different chunks of the
data
Parallel Software – SPMD

- In the vector addition example, each chunk of data could be executed as an independent thread.
- On modern CPUs, the overhead of creating threads is so high that the chunks need to be large.
  - In practice, usually a few threads (about as many as the number of CPU cores) and each is given a large amount of work to do.
- For GPU programming, there is low overhead for thread creation, so we can create one thread per loop iteration.
Parallel Software – SPMD

Single-threaded (CPU)

// there are N elements
for(i = 0; i < N; i++)
    C[i] = A[i] + B[i]

Multi-threaded (CPU)

// tid is the thread id
// P is the number of cores
for(i = 0; i < tid*N/P; i++)
    C[i] = A[i] + B[i]

Massively Multi-threaded (GPU)

// tid is the thread id

From: OpenCL 1.2 University Kit - http://developer.amd.com/partners/university-programs/
Parallel programming frameworks

- These are some of more relevant frameworks for creating parallelized code

Diagram:
- CPU:
  - OpenMP
- GPU:
  - CUDA
  - OpenCL
  - OpenACC
  - Metal
OpenCL

- OpenCL is a framework for writing parallelized code for CPUs, GPUs, DSPs, FPGAs and other processors
- Initially developed by Apple, now supported by AMD, IBM, Qualcomm, Intel and Nvidia (reluctantly)

Versions

- Latest: OpenCL 2.2
  - OpenCL C++ kernel language
  - SPIR-V as intermediate representation for kernels
    - Vulcan uses the same Standard Portable Intermediate Representation
  - AMD, Intel
- Mostly supported: OpenCL 1.2
  - Nvidia, OSX
OpenCL platforms and drivers

- To run OpenCL code you need:
  - Generic ICD loader
    - Included in the OS
  - Installable Client Driver
    - From Nvidia, Intel, etc.
  - This applies to Windows and Linux, only one platform on Mac

- To develop OpenCL code you need:
  - SDK from one of the vendors
    - Nvidia – CUDA Toolkit
    - Intel OpenCL SDK
    - AMD App SDK
Programming OpenCL

- OpenCL natively offers C99 API
- But there is also a standard OpenCL C++ API wrapper
  - Strongly recommended – reduces the amount of code
- Programming OpenCL is similar to programming shaders in OpenGL
  - Host code runs on CPU and invokes kernels
  - Kernels are written in C-like programming language
    - In many respects similar to GLSL
  - Kernels are passed to API as strings and compiled at runtime
    - Kernels are usually stored in text files
    - Kernels can be precompiled into SPIR from OpenCL 2.1
Example: Step 1 - Select device

// get all platforms (drivers)
std::vector<cl::Platform> all_platforms;
cl::Platform::get(&all_platforms);
if (all_platforms.size() == 0){
    std::cout << " No platforms found. Check OpenCL installation!\n";
    exit(1);
}
cl::Platform default_platform = all_platforms[0];
std::cout << " Using platform: " << default_platform.getInfo<CL_PLATFORM_NAME>() << "\n";

// get default device of the default platform
std::vector<cl::Device> all_devices;
default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
if (all_devices.size() == 0){
    std::cout << " No devices found. Check OpenCL installation!\n";
    exit(1);
}
cl::Device default_device = all_devices[0];
std::cout << " Using device: " << default_device.getInfo<CL_DEVICE_NAME>() << "\n";
Example: Step 2 - Build program

```cpp
cl::Context context({ default_device });

cl::Program::Sources sources;
// kernel calculates for each element C=A+B
std::string kernel_code =
  "__kernel void simple_add(__global const int* A, __global const int* B, __global int* C) {
  int index = get_global_id(0);
};";
sources.push_back({ kernel_code.c_str(), kernel_code.length() });

cl::Program program(context, sources);
try {
  program.build({ default_device });
} catch (cl::Error err) {
  std::cout << " Error building: " <<
    program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(default_device) << "\n"
  exit(1);
}
```
Example: Step 3 - Create Buffers and copy memory

Create Buffers → Create Queue → Enqueue Memory Copy

```c
// create buffers on the device
cl::Buffer buffer_A(context, CL_MEM_READ_WRITE, sizeof(int) * 10);
cl::Buffer buffer_B(context, CL_MEM_READ_WRITE, sizeof(int) * 10);
cl::Buffer buffer_C(context, CL_MEM_READ_WRITE, sizeof(int) * 10);

int A[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int B[] = { 0, 1, 2, 0, 1, 2, 0, 1, 2, 0 };

//create queue to which we will push commands for the device.
cl::CommandQueue queue(context, default_device);

//write arrays A and B to the device
queue.enqueueWriteBuffer(buffer_A, CL_TRUE, 0, sizeof(int) * 10, A);
queue.enqueueWriteBuffer(buffer_B, CL_TRUE, 0, sizeof(int) * 10, B);
```
Example: Step 4 - Execute Kernel and retrieve the results

Create Kernel → Set Kernel Arguments → Enqueue Kernel → Enqueue memory copy

```cpp
cl::Kernel kernel(program, "simple_add");

kernel.setArg(0, buffer_A);
kernelp.setArg(1, buffer_B);
kernelp.setArg(2, buffer_C);
queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(10), cl::NullRange);

int C[10];
//read result C from the device to array C
queue.enqueueReadBuffer(buffer_C, CL_TRUE, 0, sizeof(int) * 10, C);
queue.finish();

std::cout << " result: \n";
for (int i = 0; i < 10; i++){
    std::cout << C[i] << " ";
}
std::cout << std::endl;
```

Our Kernel was

```cpp
__kernel void simple_add(__read_only const int* A,
                         __read_only const int* B,
                         __write_only int* C) {
    int index = get_global_id(0);
};
```
OpenCL API Class Diagram

- **Platform** – Nvidia CUDA
- **Device** – GeForce 780
- **Program** – collection of kernels
- **Buffer / Image** – device memory
- **Sampler** – how to interpolate values for Image
- **Command Queue** – put a sequence of operations there
- **Event** – to notify that something has been done

From: OpenCL API 1.2 Reference Card
Platform model

- The host is whatever the OpenCL library runs on
  - Usually x86 CPUs for both NVIDIA and AMD
- Devices are processors that the library can talk to
  - CPUs, GPUs, DSPs and generic accelerators
- For AMD
  - All CPUs are combined into a single device (each core is a compute unit and processing element)
  - Each GPU is a separate device
Execution model

- Each kernel executes on 1D, 2D or 3D array (NDRange)
- The array is split into work-groups
- Work items (threads) in each work-group share some local memory
- Kernel can query
  - `get_global_id(dim)`
  - `get_group_id(dim)`
  - `get_local_id(dim)`
- Work items are not bound to any memory entity (unlike GLSL shaders)
Memory model

- **Host memory**
  - Usually CPU memory, device does not have access to that memory

- **Global memory** [__global__]
  - Device memory, for storing large data

- **Constant memory** [__constant__]

- **Local memory** [__local__]
  - Fast, accessible to all work-items (threads) within a workgroup

- **Private memory** [__private__]
  - Accessible to a single work-item (thread)
Memory objects

- **Buffer**
  - ArrayBuffer in OpenGL
  - Accessed directly via C pointers

- **Image**
  - Texture in OpenGL
  - Access via texture look-up function
  - Can interpolate values, clamp, etc.
Programming model

- Data parallel programming
  - Each NDRrange element is assigned to a work-item (thread)

- Task-parallel programming
  - Multiple different kernels can be executed in parallel
  - Each kernel can use vector-types of the device (float4, etc.)

- Command queue

```c
queue.enqueueWriteBuffer(buffer_A, CL_TRUE, 0, sizeof(int)*10, A);
```

- Provides means to both synchronize kernels and execute them in parallel
Big Picture

OpenCL

Context

Programs

Kernels

Memory Objects

Command Queues

CPU

GPU

Compile code

Create data & arguments

Send to execution
Thread Mapping

By using different mappings, the same thread can be assigned to access different data elements

The examples below show three different possible mappings of threads to data (assuming the thread id is used to access an element)

```
thread_id = get_global_id(0) * get_global_size(1) +
            get_global_id(1);
```

```
tid = get_group_id(1) *
      get_num_groups(0) *
      group_size +
      get_group_id(0) *
      group_size +
      get_local_id(1) *
      get_local_size(0) +
      get_local_id(0);
```
Thread Mapping

- Consider a serial matrix multiplication algorithm

```c
for(i1=0; i1 < M; i1++)
  for(i2=0; i2 < N; i2++)
    for(i3=0; i3 < P; i3++)
      C[i1][i2] += A[i1][i3]*B[i3][i2];
```

- This algorithm is suited for output data decomposition
  - We will create $N \times M$ threads
    - Effectively removing the outer two loops
  - Each thread will perform $P$ calculations
    - The inner loop will remain as part of the kernel

- Should the index space be $MxN$ or $NxM$?
Thread Mapping

- Thread mapping 1: with an MxN index space, the kernel would be:

```c
int tx = get_global_id(0);
int ty = get_global_id(1);
for (i3=0; i3 < P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

<table>
<thead>
<tr>
<th>Mapping for C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 4 8 12</td>
</tr>
<tr>
<td>1 5 9 13</td>
</tr>
<tr>
<td>2 6 10 14</td>
</tr>
<tr>
<td>3 7 11 15</td>
</tr>
</tbody>
</table>

- Thread mapping 2: with an NxM index space, the kernel would be:

```c
int tx = get_global_id (0);
int ty = get_global_id (1);
for (i3=0; i3 < P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

<table>
<thead>
<tr>
<th>Mapping for C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
</tr>
<tr>
<td>4 5 6 7</td>
</tr>
<tr>
<td>8 9 10 11</td>
</tr>
<tr>
<td>12 13 14 15</td>
</tr>
</tbody>
</table>

- Both mappings produce functionally equivalent versions of the program
Thread Mapping

- This figure shows the execution of the two thread mappings on NVIDIA GeForce 285 and 8800 GPUs

- Notice that mapping 2 is far superior in performance for both GPUs
Thread Mapping

- The discrepancy in execution times between the mappings is due to data accesses on the global memory bus
  - Assuming row-major data, data in a row (i.e., elements in adjacent columns) are stored sequentially in memory
  - To ensure coalesced accesses, consecutive threads in the same wavefront should be mapped to columns (the second dimension) of the matrices
    - This will give coalesced accesses in Matrices B and C
    - For Matrix A, the iterator \( i3 \) determines the access pattern for row-major data, so thread mapping does not affect it
Reduction

- GPU offers very good performance for tasks in which the results are stored independently
  - Process N data items and store in N memory location

- But many common operations require reducing N values into 1 or few values
  - sum, min, max, prod, min, histogram, …

- Those operations require an efficient implementation of reduction

```
float reduce_sum(float* input, int length)
{
    float accumulator = input[0];
    for(int i = 1; i < length; i++)
        accumulator += input[i];
    return accumulator;
}
```

- The following slides are based on AMD’s OpenCL™ Optimization Case Study: Simple Reductions
Reduction tree for the min operation

```c
__kernel
void reduce(__global float* buffer,
    __local float* scratch,
    __const int length,
    __global float* result) {

    int global_index = get_global_id(0);
    int local_index = get_local_id(0);
    // Load data into local memory
    if (global_index < length) {
        scratch[local_index] = buffer[global_index];
    } else {
        scratch[local_index] = INFINITY;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int offset = get_local_size(0) / 2;
        offset > 0; offset >>= 1) {
        if (local_index < offset) {
            float other = scratch[local_index + offset];
            float mine = scratch[local_index];
            scratch[local_index] = (mine < other) ? mine :
            other;
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if (local_index == 0) {
        result[get_group_id(0)] = scratch[0];
    }
}
```

- barrier ensures that all threads (work units) in the local group reach that point before execution continue
- Each iteration of the for loop computes next level of the reduction pyramid
Multistage reduction

- The local memory is usually limited (e.g. 50kB), which restricts the maximum size of the array that can be processed.
- Therefore, for large arrays need to be processed in multiple stages.
  - The result of a local memory reduction is stored in the array and then this array is reduced.
Two-stage reduction

Stage 1

Global memory

1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8

Local memory

Stage 2

Different colours denote different threads

First stage: serial reduction by N concurrent threads

Second stage: parallel reduction in local memory

```c
__kernel
void reduce(__global float* buffer,
            __local float* scratch,
            __const int length,
            __global float* result) {

    int global_index = get_global_id(0);
    float accumulator = INFINITY;
    // Loop sequentially over chunks of input vector
    while (global_index < length) {
        float element = buffer[global_index];
        accumulator = (accumulator < element) ?
            accumulator : element;
        global_index += get_global_size(0);
    }

    // Perform parallel reduction
    [The same code as in the previous example]
}
```
Reduction performance CPU/GPU

- Different reduction algorithm may be optimal for CPU and GPU
- This can also vary from one GPU to another

- The results from: http://developer.amd.com/resources/articles-whitpapers/opencl-optimization-case-study-simple-reductions/
Better way?

- **Halide** - a language for image processing and computational photography
  - Code written in a high-level language, then translated to x86/SSE, ARM, CUDA, OpenCL
  - The optimization strategy defined separately as a *schedule*
  - Auto-tune software can test thousands of schedules and choose the one that is the best for a particular platform
  - Automatically find the best trade-offs for a particular platform
  - Designed for image processing but similar languages created for other purposes
OpenCL resources

- https://www.khronos.org/registry/OpenCL/
- Reference cards
  - Google: “OpenCL API Reference Card”
- AMD OpenCL Programming Guide
  - http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OC