Parallem programming in OpenCL

Advanced Graphics
Rafal Mantiuk
Computer Laboratory, University of Cambridge

Parallel programming in OpenCL

Parallem programming in OpenCL

Advanced Graphics
Rafal Mantiuk
Computer Laboratory, University of Cambridge

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 Program Multiple Data (SPMD)

- Consider the following vector addition example:

```
for (i = 0:3) {
  C[i] = A[i] + B[i]
}
```

```
for (i = 4:7) {
  C[i] = A[i] + B[i]
}
```

```
for (i = 8:11) {
  C[i] = A[i] + B[i]
}
```

- Serial program: one program completes the entire task.
- SPMD program: multiple copies of the same program run on different chunks of the data.

```
for (i = 0:11) {
  C[i] = A[i] + B[i]
}
```

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

- 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:

```
CPU        GPU
OpenCL     OpenMP  CUDA
OpenACC
```

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
      - Vulkan 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
  - Kernels are usually stored in text files

Example: Step 1 - Select device

- Get all Platforms
- Select Platform
- Get all Devices
- Select Device

Example: Step 2 - Build program

- Create context
- Load sources (usually from file)
- Create Program
- Build Program

Example: Step 3 - Create Buffers and copy memory

- Create Buffers
- Create Queue
- Enqueue Memory Copy

Example: Step 4 - Execute Kernel and retrieve the results

- Create Kernel
- Set Kernel Arguments
- Enqueue Kernel
- Enqueue memory copy

Our Kernel was
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 NDRange 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
  - queue.enqueueWriteBuffer(buffer_A, CL_TRUE, 0, sizeof(int)*10, A);
    - CL_TRUE – Execute in-order
    - CL_FALSE – Execute out-of-order
  - Provides means to both synchronize kernels and execute them in parallel
Big Picture

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).

\[
\text{int tid = get_global_id(1) * get_global_size(0) + get_global_id(0);}\\
\text{int tid = get_global_id(0) * get_global_size(1) + get_global_id(1);}\\
\text{int group_size = get_local_size(0) * get_local_size(1);}\\
\text{int tid = get_group_id(1) * get_num_groups(0) * group_size + get_group_id(0) * group_size + get_local_id(1) * get_local_size(0) + get_local_id(0);}
\]

Thread Mapping

Consider a serial matrix multiplication algorithm

\[
\text{int tid = get_global_id(0);}\\
\text{int ty = get_global_id(1);}\\
\text{for (i=0; i < M; i++) }\\
\text{for (j=0; j < N; j++) }\\
\text{C[i][j] = A[i][i] + B[i][j];}
\]

This algorithm is suited for output data decomposition

- We will create NM threads
- 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

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.
GPU offers very good performance for tasks in which the results are stored independently.

Process \( N \) data items and store in \( N \) memory locations.

But many common operations require reducing \( N \) values into 1 or few values:

- \( \text{sum} \), \( \text{min} \), \( \text{max} \), \( \text{prod} \), \( \text{min} \), \( \text{histogram} \), ... Those operations require an efficient implementation of reduction.

The following slides are based on AMD's OpenCL Optimization Case Study: Simple Reductions.


Reduction tree for the \( \text{min} \) operation:

- \( \text{barrier} \) ensures that all threads (work units) in the local group reach that point before execution continues.
- Each iteration of the for loop computes next level of the reduction pyramid.

Two-stage reduction:

Stage 1: Serial reduction by \( N \) concurrent threads.

Stage 2: Parallel reduction in local memory.

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.

Better way?

- Halide - a language for image processing and computational photography.
  - http://halide-lang.org/
  - 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.

Reduction Performance CPU/GPU:

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

OpenCL resources

- https://www.khronos.org/registry/OpenCL/
- Reference cards
  - Google: “OpenCL API Reference Card”
- Reductions
- OpenCL Courses
  - OpenCL 1.2 University Kit
    - Perhaad Mistry & Dana Schaa, Northeastern Univ Computer Architecture Research Lab, with Ben Gaster, AMD © 2011
  - OpenCL 2.0 University Kit
    - Zhongliang Chen and Rush Udawale, Northeastern University Computer Architecture Research Lab with Perhaad Mistry and Dana Schaa, AMD © 2015