Lecture 9

ENABLING PORTABLE PERFORMANCE VIA OPENCL

Portable performance in OpenCL

- Portable performance is always a challenge, more so when OpenCL devices can be so varied (CPUs, GPUs, …)
- But OpenCL provides a powerful framework for writing performance portable code
- The following slides are general advice on writing code that should work well on most OpenCL devices

Optimization issues

- Efficient access to memory
  - Memory coalescing
    - Ideally get work-item i to access data[i] and work-item j to access data[j] at the same time etc.
  - Memory alignment
    - Padding arrays to keep everything aligned to multiples of 16, 32 or 64 bytes
- Number of work-items and work-group sizes
  - Ideally want at least 4 work-items per PE in a Compute Unit on GPUs
  - More is better, but diminishing returns, and there is an upper limit
    - Each work item consumes PE finite resources (registers etc)
- Work-item divergence
  - What happens when work-items branch?
  - Actually a SIMD data parallel model
  - Both paths (if-else) may need to be executed (*branch divergence*), avoid where possible (non-divergent branches are termed *uniform*)

Memory layout is critical to performance

- "Structure of Arrays vs. Array of Structures" problem:

```
struct { float x, y, z, a; } Point;
```

- Structure of Arrays (SoA) suits memory coalescence on GPUs

Adjacent work-items like to access adjacent memory

- Array of Structures (AoS) may suit cache hierarchies on CPUs

Individual work-items like to access adjacent memory

Other optimisation tips

- Use a profiler to see if you're getting good performance
  - Occupancy is a measure of how active you're keeping each PE
  - Occupancy measurements of >0.5 are good (>50% active)
- Other measurements to consider with the profiler:
  - Memory bandwidth – should aim for a good fraction of peak
    - E.g. 148 GBytes/s to Global Memory on an M2050 GPU
  - Work-Item (Thread) divergence – want this to be low
  - Registers per Work-Item (Thread) – ideally low and a nice divisor of the number of hardware registers per Compute Unit
    - E.g. 32,768 on M2050 GPUs
    - These are statically allocated and shared between all Work-Items and Work-Groups assigned to each Compute Unit
    - Four Work-Groups of 1,024 Work-Items each would result in just 8 registers per Work-Item! Typically aim for 16-32 registers per Work-Item

Portable performance in OpenCL

- Don't optimize too hard for any one platform, e.g.
  - Don't write specifically for certain warp/wavefront sizes etc
  - Be careful not to rely on specific sizes of local/global memory
  - OpenCL's vector data types have varying degrees of support – faster on some devices, slower on others
  - Some devices have caches in their memory hierarchies, some don't, and it can make a big difference to your performance without you realizing
  - Choosing the allocation of Work-Items to Work-Groups and dimensions on your kernel launches
  - Performance differences between unified vs. disjoint host/global memories
  - Double precision performance varies considerably from device to device
  - Some OpenCL SDKs give useful feedback about how well they can compile your code (but you have to turn on this feedback)

- It is a good idea to try your code on several different platforms to see what happens (profiling is good!)
  - At least two different GPUs (ideally different vendors) and at least one CPU

Advice for performance portability

- Discover what devices you have available at run-time, e.g.

C

```
// Get available platforms
cl_uint nPlatforms;
cl_platform_id platforms[MAX_PLATFORMS];
int ret = clGetPlatformIDs(MAX_PLATFORMS, platforms, &nPlatforms);

// Loop over all platforms
for (int p = 0; p < nPlatforms; p++) {
  // Get available devices
  cl_uint nDevices = 0;
  cl_device_id devices[MAX_DEVICES];
  clGetDeviceIDs(platforms[p], deviceType, MAX_DEVICES, devices, &nDevices);

  // Loop over all devices in this platform
  for (int d = 0; d < nDevices; d++)
    getDeviceInformation(devices[d]);
}
```

Advice for performance portability

- Micro-benchmark all your OpenCL devices at run-time to gauge how to divide your total workload across all the devices
  - Ideally use some real work so you're not wasting resource
  - Keep the microbenchmark very short otherwise slower devices penalize faster ones
- Once you've got a work fraction per device calculated, it might be worth retesting from time to time
  - The behavior of the workload may change
  - The host or devices may become busy (or quiet)
- It is most important to keep the fastest devices busy
  - Less important if slower devices finish slightly earlier than faster ones (and thus become idle)
- Avoid overloading the CPU with both OpenCL host code and OpenCL device code at the same time

Timing microbenchmarks (C)

```
for (int i = 0; i < numDevices; i++) {
    // Wait for the kernel to finish
    ret = clFinish(oclDevices[i].queue);
    // Update timers
    cl_ulong start, end;
    ret = clGetEventProfilingInfo(oclDevices[i].kernelEvent,
                CL_PROFILING_COMMAND_START,
                sizeof(cl_ulong), &start, NULL);
    ret |= clGetEventProfilingInfo(oclDevices[i].kernelEvent,
                CL_PROFILING_COMMAND_END,
                sizeof(cl_ulong), &end, NULL);
    long timeTaken = (end - start);
    speeds[i] = timeTaken / oclDevices[i].load;
}
```

Advice for performance portability

- Optimal Work-Group sizes will differ between devices
  - E.g. CPUs tend to prefer 1 Work-Item per Work-Group, while GPUs prefer lots of Work-Items per Work-Group (usually a multiple of the number of PEs per Compute Unit, i.e. 32, 64 etc.)
- From OpenCL v1.1 you can discover the preferred Work-Group size multiple for a kernel once it's been built for a specific device
  - Important to pad the total number of Work-Items to an exact multiple of this
  - Again, will be different per device
- The OpenCL run-time will have a go at choosing good EnqueueNDRangeKernel dimensions for you
  - With very variable results

- **Your mileage will vary**, the best strategy is to write ***adaptive*** code that makes decisions at run-time

Tuning Knobs
some general issues to think about

- Tiling size (work-group sizes, dimensionality etc.)
  - For block-based algorithms (e.g. matrix multiplication)
  - Different devices might run faster on different block sizes
- Data layout
  - Array of Structures or Structure of Arrays (AoS vs. SoA)
  - Column or Row major
- Caching and prefetching
  - Use of local memory or not
  - Extra loads and stores assist hardware cache?
- Work-item / work-group data mapping
  - Related to data layout
  - Also how you parallelize the work
- Operation-specific tuning
  - Specific hardware differences
  - Built-in trig / special function hardware
  - Double vs. float (vs. half)

From Zhang, Sinclair II and Chien: Improving Performance Portability in OpenCL Programs – ISC13

Auto tuning

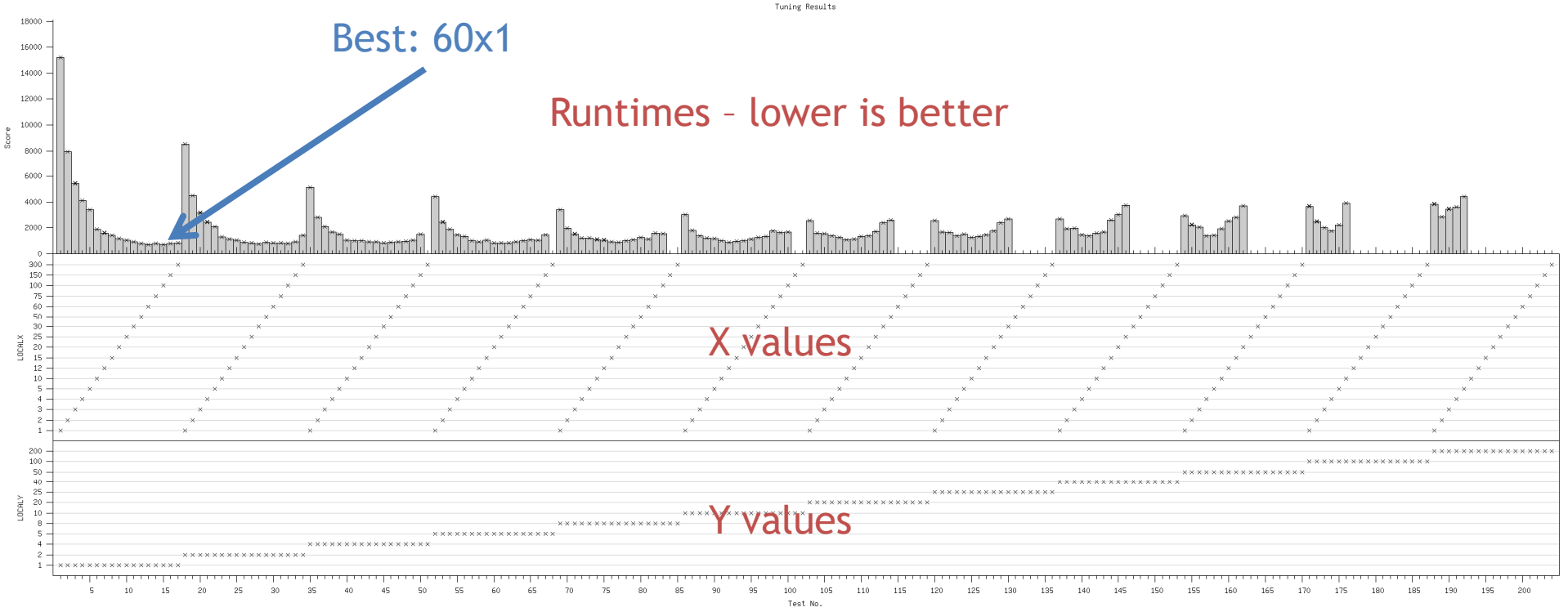
- Q: How do you know what the ***best*** parameter values for your program are?
  - What is the best work-group size, for example
- A: Try them all! (Or a well chosen subset)
- This is where auto tuning comes in
  - Run through different combinations of parameter values and optimize the runtime (or another measure) of your program.

Auto tuning example - Flamingo

- http://mistymountain.co.uk/flamingo/
- Python program which compiles your code with different parameter values, and calculates the “best” combination to use
- Write a simple config file, and Flamingo will run your program with different values, and returns the best combination
- Remember: scale down your problem so you don’t have to wait for “bad” values (less iterations, etc.)

Auto tuning - Example

- D2Q9 Lattice-Boltzmann
- What is the best work-group size for a specific problem size (3000x2000) on a specific device (NVIDIA Tesla M2050)?



Exercise 11: Optimize matrix multiplication

- Goal:
  - To understand portable performance in OpenCL
- Procedure:
  - Optimize a matrix multiply solution step by step, saving intermediate versions and tracking performance improvements
  - After you've tried to optimize the program on your own, study the blocked solution optimized for an NVIDIA GPU. Apply these techniques to your own code to further optimize performance
  - As a final step, go back and make a single program that is adaptive so it delivers good results on both a CPU and a GPU
- Expected output:
  - A message confirming that the matrix multiplication is correct
  - Report the runtime and the MFLOPS