Portable Performance ?

22 November, 2017 | Ilya Zhukov

OpenCL performance aspects

- Performance considerations
  - host-device efficiency
  - core utilisation
  - load balancing
  - diverged branches
  - memory optimisations

OpenCL portable

- OpenCL is a portable programming model, performance portability is **not guaranteed**

- Examples:
  - GPUs rely on fast shared local memory which needs to be programmed explicitly
  - Xeon Phi includes fully coherent cache hierarchy (similar to CPUs) automatically speeding up memory accesses

  - GPU performance is based on HW scheduling of many (small) threads
  - Xeon Phi relies on the device OS to schedule (medium size) threads

warp or wavefront of GPU threads

- programming GPUs: schedule many 1000s of threads organized in tiles of threads
  - threads – tiles of threads
  - CUDA threads – thread blocks
  - work items – work groups
- additional bunching of threads:
  - NVIDIA warp (32 threads) - AMD wavefront (64 threads)
  - most basic unit of scheduling, smallest executable unit of code, minimum size of the data processed in SIMD fashion, processes a single instruction over all processes at the same time
  - comparable CPU hardware concept: vector width given in total number of bits (128, 256, 512), floats, doubles
- avoid having **diverged** warps/wavefronts
  - all threads in a warp/wavefront execute the **same instruction** in lock-step (but on **different data**)
  - if the code causes the threads in a warp to be unable to execute the same instruction, some threads will be diverged during the execution of that instruction (leaving some hardware unused)

warp or wavefront of GPU threads II

- examples of divergence:
  - tile size not a multiple of the warp/wavefront size
  - branching where the whole warp/wavefront is unable to take the same branch
- simple line of code:

```
if (idx[0] >= XYZ)
   my_array[idx] += A;
else
   my_array[idx] *= B;
```

- if XYZ is a multiple of the warp/wavefront size no divergence occurs
- else there will be a warp/wavefront executed twice:
  - once with the first threads adding up A and the other ones idling
  - second with the last threads multiplying B an the first ones idling
- choose a tile size of a multiple of 64
- ensure that conditionals and loops would not diverge threads within the 64 boundary

General tuning issues

- 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

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 CU on GPUs
  - More is better, but there is an upper limit as each work-item consumes PE finite resources (registers etc.)
- Work-item divergence
  - A SIMD data parallel model
  - When work-items branch both paths (if-else) may need to be executed

Memory layout is critical to performance

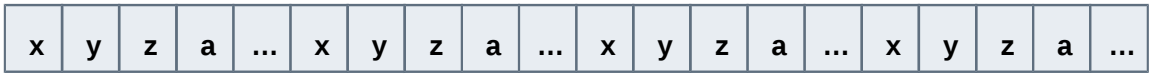
- "Structure of Arrays vs. Array of Structures" problem:

  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

Key performance aspects (Xeon Phi)

- Multi-threading parallelism
  - many cores (assume 60 here), each capable of running four HW threads
  - populating the 240 HW threads is essential
- In-core vectorization
  - SIMD vector size is 512 bit (8 double or 16 single precision floating point numbers)
  - core can issue one vector computation per cycle
- Xeon Phi coprocessor resides on the PCIe bus
  - high latency/low bandwidth → minimize traffic
- Memory subsystem
  - three levels of memory (GDDR, L2 and L1 cache)
  - Xeon Phi (KNC) is in-order machine → latency of memory accesses has significant performance impact

Caches (Xeon Phi)

	L1 (Data + Instructions)	Shared L2
Total Size	32 KB + 32 KB	512 KB
Miss Latency	15-30 cycles	500-1000 cycles

- L1 cache access involves latency of only one cycle
- consecutive memory access is fastest:
  - improves cache efficiency
  - reduces number of TLB misses
  - allows HW prefetcher to contribute

Mapping OpenCL constructs to Xeon Phi

- at initialization time, the Intel Xeon Phi OpenCL driver creates 240 SW threads and pins them to HW threads
- following a **clEnqueueNDRange()** call, the driver schedules the work groups of the current NDRange on the 240 threads

  → calling a kernel with less than 240 work groups leaves the coprocessor underutilized

Mapping OpenCL constructs to Xeon Phi

- the OpenCL compiler creates an optimized routine that executes a work group built from up to three nested loops

```
1  __Kernel ABC(…)
2  For (int i = 0; i < get_local_size(2); i++)
3      For (int j = 0; j < get_local_size(1); j++)
4          For (int k = 0; k < get_local_size(0); k++)
5              Kernel_Body;
```

- the innermost loop is used for dimension zero of the NDRange
  - impact on memory access pattern
  - impact on vectorization efficiency

```
1  __Kernel ABC(…)
2  For (int i = 0; i < get_local_size(2); i++)
3      For (int j = 0; j < get_local_size(1); j++)
4          For (int k = 0; k < get_local_size(0); k += VECTOR_SIZE)
5              Vector_Kernel_Body;
```

- implicitly vectorized work group routine based on dimension zero loop
- unrolled by the vector size (16, regardless of data types used)

Exposing algorithm parallelism

- OpenCL provides various ways to express parallelism and concurrency, some of them **will not map well** to Xeon Phi
- key OpenCL constructs that should be considered here:

Multi-threading

- best to have more than 1000 work groups per NDRange
- applications with NDRange of 1000 work groups or less will suffer from serious under-utilization of threads

Vectorization

- Xeon Phi coprocessor includes an implicit vectorization module
- automatically vectorized the implicit loop over the work items in dimension zero (vectorization width 16)
- don't manually vectorized kernels
- choose work group sizes as a multiple of 16 (in future 32)

Data alignment

- to guarantee that work groups start on a vector-size aligned address the work group size must be divisible by 16 (future 32)

  local size NULL lets the OpenCL driver choose the best size

Benefit from the Xeon Phi memory subsystem

- Xeon Phi is an in-order machine, i.e. sensitive to memory latencies:

Data reuse (intra work group)

- data reuse from the caches (blocking / tiling)

Data access pattern

- consecutive data access (at least wrt dimension zero loop)

Data layout

- **SOA:** efficient vector loads/stores but lower spatial locality
- **AOS:** load/store via gather/scatter but for random access pattern often better spatial locality

Data prefetching

- Xeon Phi includes a simple automatic prefetcher to the L2
- OpenCL kernel prefetch built-in for manual prefetches

Local memory and barriers

- traditional GPUs include Shared Local Memory whereas Xeon Phi includes a 2-level cache system
- local memory is allocated on the regular GDDR memory (overhead)
- no special hardware support for barriers (emulated)

Summary of Xeon Phi performance aspects

- include enough work groups in each NDRange (> 1000)
- avoid lightweight work groups, try maximum local size (1024)
- avoid ID(0) dependent control flow
  - allows for implicit vectorisation
- prefer consecutive data access
- data layout preferences:
  - AOS for sparse random access
  - pure SOA or AOSOA(32) otherwise
- exploit data reuse through the caches within the work group
  - use tiling, blocking
- if auto-prefetching fails, use PREFETCH built-in
  - brings global data to the cache 500-1000 cycles before use
- don't use local memory
- avoid using barriers

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