OpenCL for x86 CPU and Intel MIC

Jiří Filipovič

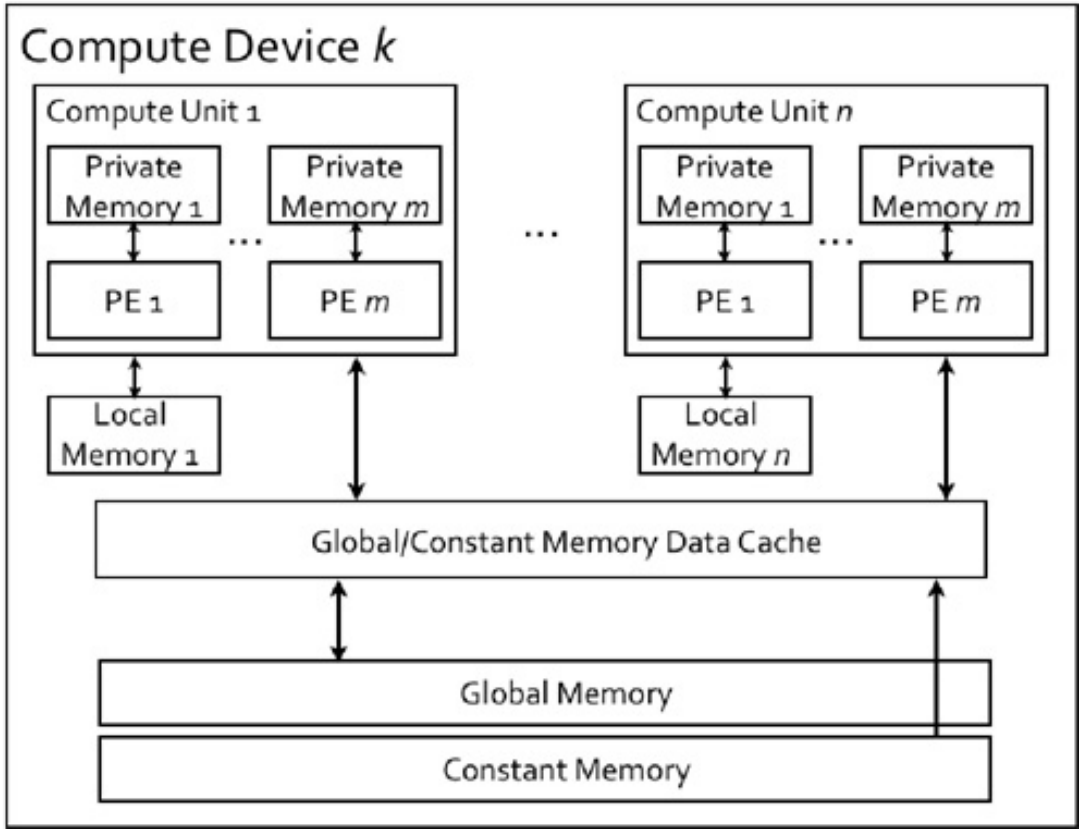
Fall 2021

x86 CPU Architecture

Common features of (nearly all) modern x86 processors

- core is complex, out-of-order instruction execution, large cache
- multiple cache coherent cores in single chip
- vector instructions (MMX, SSE, AVX)
- NUMA for multi-socket systems

OpenCL Device



CPU and OpenCL

The projection of CPU HW to OpenCL model

- CPU cores are compute units
- vector ALUs are processing elements
  - so the number of work-items running in lock-step is determined by instruction set (e.g., SSE, AVX) and data type (e.g., float, double)
- one or more work-groups create a CPU thread
  - the number of work-groups should be at least equal to the number of cores
  - higher number of work-groups allows to better workload balance (e.g., what if we have eight work-groups at six-core CPU?), but creates overhead
- work-items form serial loop, which may be vectorized

Implicit and Explicit Vectorization

Implicit vectorization

- we write scalar code (similarly as for NVIDIA and AMD GCN)
- the compiler generates vector instructions from work-items (creates loop over work-items and vectorizes this loop)
- better portability (we do not care about vector size and richness of vector instruction set)
- supported by Intel OpenCL, AMD OpenCL does not support it yet

Explicit vectorization

- we use vector data types in our kernels
- more complex programming, more architecture-specific
- potentially better performance (we do not rely on compiler ability to vectorize)

Differences from GPU

Images

- CPU does not support texture units, so they are emulated
- better to not use...

Local memory

- no special HW at CPU
- brings overhead (additional memory copies)
- but it is meaningful to use memory pattern common for using local memory, as it improves cache locality

Intel MIC

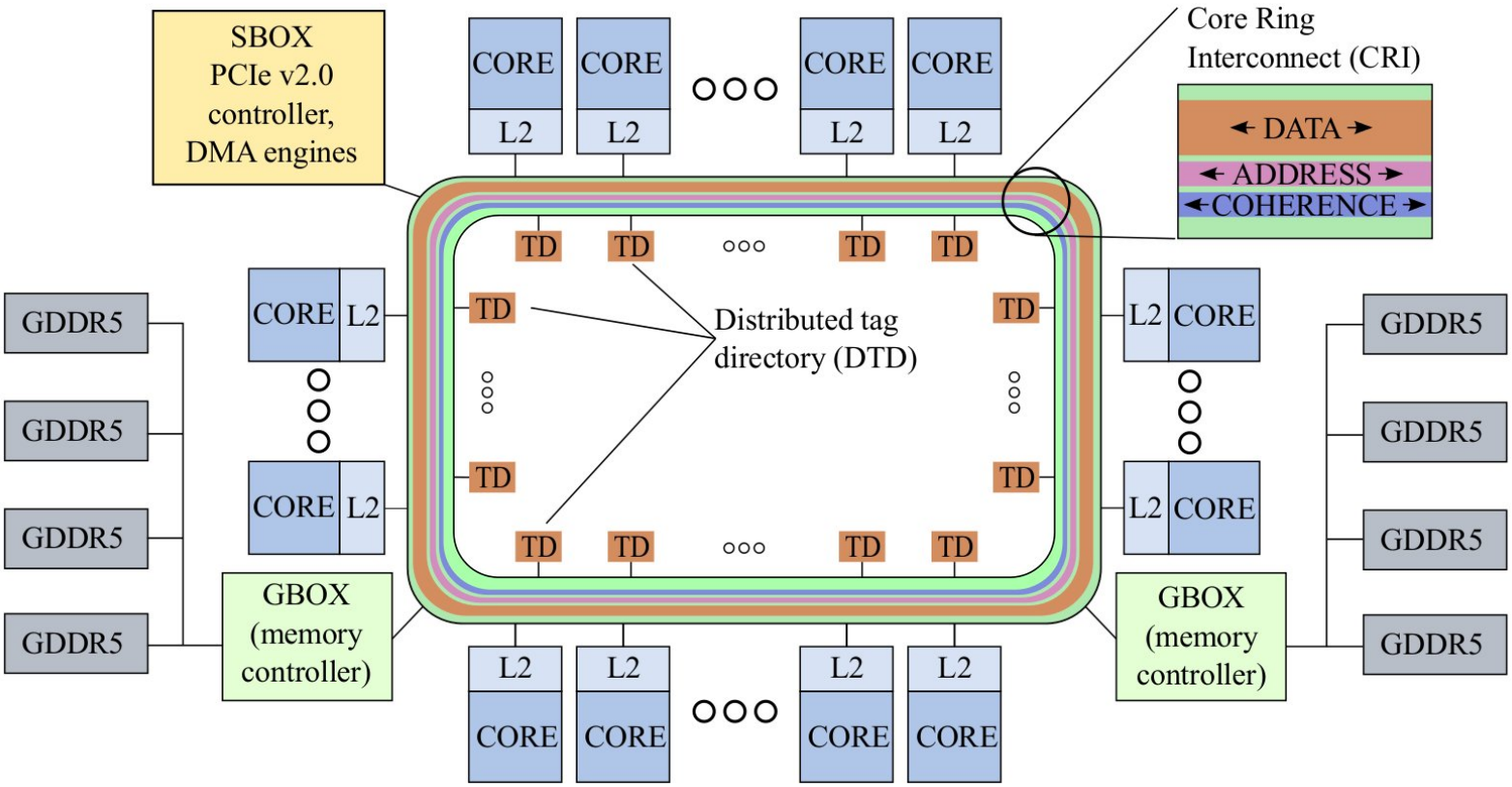
What is MIC?

- Many Integrated Core Architecture
- originated in Intel Larrabee project (x86 graphic card)

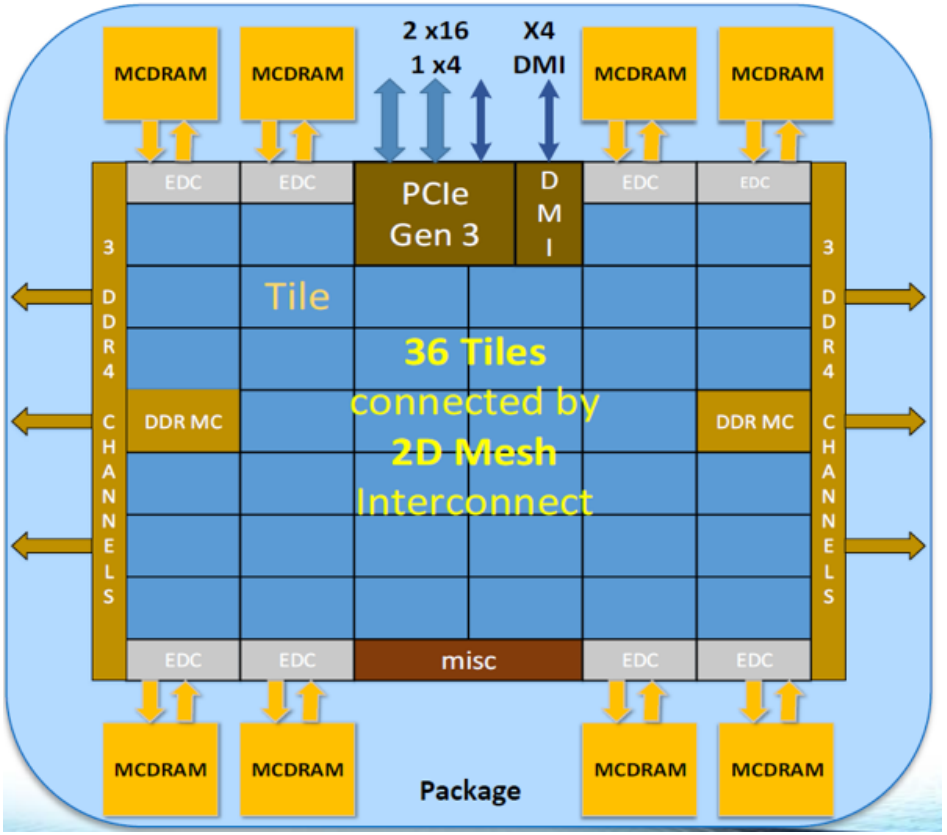
Existing hardware

- Knights Corner (KNC) and Knights Landing (KNL) generation
- large number of x86 cores
- cores are connected by bi-directional ring bus (KNC) or mesh (KNL)
- cache-coherent system
- connected to high-throughput memory

KNC Processor



KNL Processor



Intel MIC

MIC core

- relatively simple, KNC in-order, KNL based on Atom Airmont
- use hyperthreading (4 threads per core)
  - needs to be used to exploit full performance on KNC
- fully cache coherent, 32+32 KB L1 cache (I+D), 512 KB L2 cache
- contain wide vector units (512-bit vectors)
  - predicated execution
  - gather/scatter instructions
  - transcendentals

Current Hardware

Xeon Phi

- product based on MIC architecture
- bootable processor, or PCI-E card with dedicated memory
  - runs its own operating system

Xeon Phi 7210

- 64 x86 cores at 1.3 GHz
- 16 GB HBM RAM + DDR4 RAM up to 384 GB
- 2.25 TFlops DP, 4.5 TFlops SP
- 450 GB/sec HBM, 102 GB/s DDR4 memory bandwidth

Programming Models

Native programming model (KNC)

- we can execute the code directly at accelerator
- after recompilation, we can use the same code as for CPU
- programming via OpenMP, MPI

Offload programming model (KNC)

- application is executed at host
- code regions are offloaded to accelerator, similarly as in the case of GPUs
  - by using `#pragma offload` with intel tools
  - by using OpenCL

KNL is host processor.

MIC and OpenCL

The projection of MIC HW to OpenCL programming model is very similar to CPU case

- work-groups creates threads
- work-items creates iterations of vectorized loops
  - higher number of work-items due to wider vectors
  - less sensitive to divergence and uncoalesced memory access due to richer vector instruction set
- high need of parallelism
  - e.g., 64 cores executes 256 threads

OpenCL Optimization for CPU and MIC

We will discuss optimizations for CPU and MIC together

- many common concepts
- differences will be emphasized

Parallelism

How to set a work-group size?

- we do not need high parallelism to mask memory latency
- but we need enough work-items to fill vector width (if implicit vectorization is employed)
- the work-group size should be divisible by vector length, it can by substantially higher, if we don't use local barriers
  - Intel recommends 64-128 work-items without synchronizations and 32-64 work-items with synchronizations
  - general recommendation, needs experimenting ...
- we can let a compiler to choose the work-group size

How many work-groups?

- ideally multiple of (virtual) cores
- be aware of NDRange tile effect (especially at MIC)

Thread-level Parallelism

Task-scheduling overhead

- overhead of scheduling large number of threads
- issue mainly on MIC (CPU has too low cores)
- problematic for light-weight work groups
  - low workload per work-item
  - small work-groups
- can be detected by profiler easily

Barriers overhead

- no HW implementation of barriers, so they are expensive
- higher slowdown on MIC

Vectorization

Branches

- if possible, use uniform branching (whole work-group follows the same branch)
- consider the difference
  - `if (get_global_id(0) == 0)`
  - `if (kernel_arg == 0)`
- divergent branches
  - can forbid vectorization
  - can be masked (both then and else branches are executed)

Vectorization

Scatter/gather

- supported mainly on MIC
- for non-consecutive memory access, compiler tries to generate scatter/gatter instructions
  - instructions use 32-bit indices
  - `get_global_id()` returns `size_t` (64-bit)
  - we can cast indices explicitly
- avoid pointer arithmetics, use array indexing
  - more transparent for the compiler

Memory Locality

Cache locality

- the largest cache dedicated to core is L2
- cache blocking – create work-groups using memory regions fitting into L1, or not exceeding L2 cache

AoS

- array of structures
- more efficient for random access

SoA

- structure of arrays
- more efficient for consecutive access

Memory Access

Memory access pattern

- consecutive memory access is the most efficient in both architectures
- however, there are differences
  - KNC is in-order, so the memory access efficiency heavily depends on prefetching, which is more successful for consecutive access
  - CPU does not support vector gather/scatter

Alignment

- some vector instructions require alignment
  - IMCI (MIC): 64-byte
  - AVX: no requirements
  - SSE: 16-byte
- pad innermost dimension of arrays

Memory Access

Prefetching on KNC

- prefetching is done by HW and by SW
  - generated by the compiler
  - also can be explicitly programmed (function `void prefetch(const __global gentype *p, size_t num_elements)`)
- explicit prefetching helps, e.g., in irregular memory access pattern

Memory Access

False sharing

- accessing different addresses in the same cache line from several threads
  - cache line has 64 bytes on modern Intel processors
- brings significant penalty
- padding is the solution...

Concurrent R/W access to the same address

- it is better to create local copies and merge them when necessary (if possible)
- reduces also synchronization

Vector reduction

Rewritten CUDA version

- uses very similar concept as was demonstrated in former lecture, but run in constant number of threads
- reaches nearly peak theoretical bandwidth on both NVIDIA and AMD GPUs

Reduction for GPUs (1/2)

```
__kernel void reduce(__global const int* in, __global int* out,
    unsigned int n, __local volatile int *buf) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0)*(get_local_size(0)*2)
    + get_local_id(0);
  unsigned int gridSize = 256*2*get_num_groups(0);
  buf[tid] = 0;

  while (i < n) {
    buf[tid] += in[i];
    if (i + 256 < n)
      buf[tid] += in[i+256];
    i += gridSize;
  }
  barrier(CLK_LOCAL_MEM_FENCE);
```

Reduction for GPUs (2/2)

```
  //XXX hard optimization for 256-thread work groups
  if (tid < 128)
    buf[tid] += buf[tid + 128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid < 64)
    buf[tid] += buf[tid + 64];
  barrier(CLK_LOCAL_MEM_FENCE);

  //XXX hard optimization for 32-bit warp size
  //XXX problematic on new NVIDIA HW
  if (tid < 32) {
    buf[tid] += buf[tid + 32];
    buf[tid] += buf[tid + 16];
    buf[tid] += buf[tid + 8];
    buf[tid] += buf[tid + 4];
    buf[tid] += buf[tid + 2];
    buf[tid] += buf[tid + 1];
  }

  if (tid == 0) atomic_add(out, buf[0]);
}
```

Vector reduction

Execution of GPU code on CPU and Phi

- difficult to vectorize
- overhead of local reduction, which is not necessary

Optimizations for CPU and MIC

- the simplest solution is to use only necessary amount of parallelism
- work-groups of one vectorized work-item

Reduction for CPU and MIC

```
__kernel void reduce(__global const int16* in, __global int* out,
    const unsigned int n, const unsigned int chunk) {

  unsigned int start = get_global_id(0)*chunk;
  unsigned int end = start + chunk;
  if (end > n) end = n;

  int16 tmp = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
  for (int i = start/16; i < end/16; i++)
    tmp += in[i];

  int sum = tmp.s0 + tmp.s1 + tmp.s2 + tmp.s3 + tmp.s4
    + tmp.s5 + tmp.s6 + tmp.s7 + tmp.s8 + tmp.s9 + tmp.sa
    + tmp.sb + tmp.sc + tmp.sd + tmp.se + tmp.sf;

  atomic_add(out, sum);
}
```

Electrostatic Potential Map

Important problem from computational chemistry

- we have a molecule defined by position and charges of its atoms
- the goal is to compute charges at a 3D spatial grid around the molecule

In a given point of the grid, we have

$$V_i = \sum_j \frac{w_j}{4\pi\epsilon_0 r_{ij}}$$

Where w_j is charge of the j-th atom, r_{ij} is Euclidean distance between atom j and the grid point i and ϵ_0 is vacuum permittivity.

Algorithm Analysis

Parallelization

- each grid point can be processed in parallel
- not practical to parallelize loop going over atoms (reduction)

Performance bound of the naive algorithm

- 11 arithmetic operations per one atom per grid point
- atom's data require 16 bytes (4 floats – Cartesian position and charge)
- computation for one grid point is memory-bound
- caches maintain locality for multiple grid points (atom reads are synchronous)

Improving the Algorithm

We can compute a grid per 2D slices

- enough parallelism
- distance in z-dimension can be precomputed (stored instead of z-dimension of atom's data)
- reduce number of arithmetic operations per atom per grid point to 9

Implementation

```
int xIndex = get_global_id(0);
int yIndex = get_global_id(1);
int outIndex = get_global_size(0) * yIndex + xIndex;

float coordX = gridSpacing * xIndex;
float coordY = gridSpacing * yIndex;

float energyValue = 0.0f;
for (int i = 0; i < numberOfAtoms; i++) {
  float dX = coordX - atomInfo[i].x;
  float dY = coordY - atomInfo[i].y;
  energyValue += atomInfo[i].w
    * native_rsqrt(dX*dX + dY*dY + atomInfo[i].z);
}

energyGrid[outIndex] += energyValue;
```

Performance

Let's set slice size to 512×512, number of atoms to 4096, WG size to 16×16, and measure the performance in number of atoms evaluated per second.

Code	2×CPU	MIC	GPU
slices	25.8 Geval/s	48.1 Geval/s	45.0 Geval/s

Performance

Let's optimize WG size

- 8×2 for CPU, 8×1 for MIC, 16×4 for GPU

Code	2×CPU	MIC	GPU
slices	25.8 Geval/s	48.1 Geval/s	45.0 Geval/s
optimized WG	26.1 Geval/s	54.4 Geval/s	45.8 Geval/s

Removing Redundancy

Are there any redundant work among WIs?

- WIs in the same warp/vector read the same atom data
- WIs in the same row compute the same y-distance
- redundancy removing critical for GPU, but may also improve performance on CPU and MIC (if compiler fails to remove invariant code)

We can assign more work per WI

- "unrolling of the outer (parallelized) loop", so a WI computes several grid points at a row
- increases private memory locality (atom data are used for more grid points)
- removes some redundant computation of y-distance
- reduces strong scaling, uses more registers

Performance

We have tested from 1 to 8 grid points and re-optimize WG size.

- unroll 8× for CPU, 2× for MIC and 8× for GPU

Code	2×CPU	MIC	GPU
slices	25.8 Geval/s	48.1 Geval/s	45.0 Geval/s
optimized WG	26.1 Geval/s	54.4 Geval/s	45.8 Geval/s
unrolling	54.5 Geval/s	60.9 Geval/s	162.0 Geval/s

Memory Access Optimization

CPU and MIC often prefers SoA

- we can split x, y, z-dimensions and charge w into separate arrays

GPU caches global memory in L2 cache only

- we can use constant memory for atom data

Performance

We have tested from 1 to 8 grid points and re-optimize WG size.

- CPU and MIC prefers SoA, GPU prefers constant memory (more visible effect if unrolling is disabled)

Code	2×CPU	MIC	GPU
slices	25.8 Geval/s	48.1 Geval/s	45.0 Geval/s
optimized WG	26.1 Geval/s	54.4 Geval/s	45.8 Geval/s
unrolling	54.5 Geval/s	60.9 Geval/s	162.0 Geval/s
optimized mem.	60.2 Geval/s	61.1 Geval/s	164.9 Geval/s

Manual Vectorization

Vectorization of memory access

- we pack atoms data into vectors (both in SoA and AoS)
- usable to enforce vectorized data access

Vectorized computation

- we read vectorized data and perform vectorized computation in each WI

Performance

We have tested using vector from size 2 to size 8.

- CPU prefers to not vectorize, MIC prefers SoA with vector size 4 and scalar computation, GPU prefers scalar computation with AoS using vector size 8 (i.e. two atoms are packed into single vector)

Code	2×CPU	MIC	GPU
slices	25.8 Geval/s	48.1 Geval/s	45.0 Geval/s
optimized WG	26.1 Geval/s	54.4 Geval/s	45.8 Geval/s
unrolling	54.5 Geval/s	60.9 Geval/s	162.0 Geval/s
optimized mem.	60.2 Geval/s	61.1 Geval/s	164.9 Geval/s
vectorized		62.4 Geval/s	168.3 Geval/s

Performance without squere root

The performance of MIC is quite low and optimizations does not improve it

- slower implementation of `native_rsqrt`
- depsite it leads to uncorrect algorithm, we have tested performance with removed reciprocal square root

Performance without squere root

Code	2×CPU	MIC	GPU
slices	30.0 Geval/s	103.8 Geval/s	43.6 Geval/s
optimized WG	30.6 Geval/s	114.3 Geval/s	43.8 Geval/s
unrolling	68.3 Geval/s	148.9 Geval/s	221.8 Geval/s
optimized mem.	70.9 Geval/s	159.3 Geval/s	260.0 Geval/s
vectorized		175.4 Geval/s	266.4 Geval/s