OpenCL for x86 CPU and Intel MIC

Jiří Filipovič

fall 2015

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

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

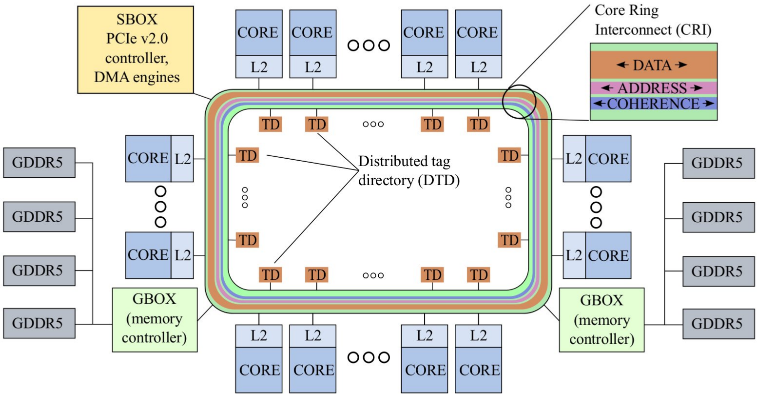
What is MIC?

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

Main features of the architecture

- large number of x86 cores
- a bidirectional ring bus connecting all cores
- cache-coherent system
- connected to high-throughput memory

MIC Architecture



Intel MIC

MIC core

- relatively simple (in-order in current architecture)
- use hyperthreading (4 threads per core)
    - needs to be used to exploit full performance
- 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
- PCI-E card with dedicated memory
  - bootable system in future generation
- runs own operating system (ssh from the host)

Xeon Phi 7120P

- 61 x86 cores at 1.2GHz
- 16GB RAM
- 1.2 TFlops SP, 2.4 TFlops DP
- 352 GB/sec global memory bandwidth

Programming Model

Native programming model

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

Offload programming model

- 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

MIC and OpenCL

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

- compute units creates threads
- processing elements 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. 61 cores need 244 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 predicated (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 of 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
  - MIC is in-order, so the memory access efficiency heavily depends on prefetching, which is more successful for consecutive access
  - CPU does not support gather/scatter, thus inefficiency comes also from forbidding vectorization

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 MIC

- 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 the same cache line from several threads
  - 64-byte block on modern Intel processors
- brings significant penalty

False sharing reasons

- multiple threads access the same addresses
  - it is better to create local copies and merge them when necessary (if possible)
  - reduces also synchronization
- multiple threads access different addresses in the same cache line
  - padding

Memory Access

NUMA

- Non-Uniform Memory Access
- realized usually at multi-socket setups
  - common with modern CPUs, can be also realized in single chip, or memory access can be uniform (FSB)
- each CPU has own local memory with faster access and non-local memory with slower access (local memory of other processors)
- when allocated, the block of memory is inserted in local memory
  - so access from threads running on different CPU is slower
  - thread-data affinity cannot be managed with current OpenCL specification

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, no problem at AMD
    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);
}
```

Histogram

We will show histogram computation in C++ using OpenMP

- to show more common way to implement highly efficient code for CPU and MIC
- to show that optimization is nontrivial task even in C++
- there are the same architecture restriction, changes are only in programming model

Histogram

- a distribution of numerical data (occurrence of values in defined intervals)
- in our case, we will create a histogram of age distribution with equally-sized intervals
- example taken from Parallel Programming and Optimization with Intel Xeon Phi Coprocessors (Colfax Research)

Histogram – serial version

```
void Histogram(const float* age, int* const hist, const int n,
    const float group_width, const int m) {
  for (int i = 0; i < n; i++) {
    const int j = (int) ( age[i] / group_width );
    hist[j]++;
  }
}
```

Issues

- vector dependence in loop
- inefficient division

Histogram – optimize division

```
void Histogram(const float* age, int* const hist, const int n,
    const float group_width, const int m) {
  const float invGroupWidth = 1.0f/group_width;
  for (int i = 0; i < n; i++) {
    const int j = (int) ( age[i] * invGroupWidth );
    hist[j]++;
  }
}
```

Issues

- vector dependence in loop

Histogram – vectorized

```
void Histogram(const float* age, int* const hist, const int n,
    const float group_width, const int m) {
  const int vecLen = 16;
  const float invGroupWidth = 1.0f/group_width;
  //XXX: this algorithm assumes n%vecLen == 0.
  for (int ii = 0; ii < n; ii += vecLen) {
    int histIdx[vecLen];
    for (int i = ii; i < ii + vecLen; i++)
      histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
    for (int c = 0; c < vecLen; c++)
      hist[histIdx[c]]++;
  }
}
```

Issues

- data are not aligned (if they are, compiler does not see it)

Histogram – vectorized

```
void Histogram(const float* age, int* const hist, const int n,
    const float group_width, const int m) {
  const int vecLen = 16;
  const float invGroupWidth = 1.0f/group_width;
  //XXX: this algorithm assumes n%vecLen == 0.
  for (int ii = 0; ii < n; ii += vecLen) {
    int histIdx[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
    for (int i = ii; i < ii + vecLen; i++)
      histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
    for (int c = 0; c < vecLen; c++)
      hist[histIdx[c]]++;
  }
}
```

Issues

- vectorized, but not thread-parallel

Histogram – thread-parallel

```
#pragma omp parallel for schedule(guided)
for (int ii = 0; ii < n; ii += vecLen) {
  int histIdx[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
  for (int i = ii; i < ii + vecLen; i++)
    histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
  for (int c = 0; c < vecLen; c++)
#pragma omp atomic
    hist[histIdx[c]]++;
  }
}
```

Issues

- too many atomics

Histogram – private storage

```
#pragma omp parallel
{
  int hist_priv[m];
  hist_priv[:] = 0;
  int histIdx[vecLen] __attribute__((aligned(64)));
#pragma omp for schedule(guided)
  for (int ii = 0; ii < n; ii += vecLen) {
#pragma vector aligned
    for (int i = ii; i < ii + vecLen; i++)
      histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
    for (int c = 0; c < vecLen; c++)
      hist_priv[histIdx[c]]++;
  }
  for (int c = 0; c < m; c++) {
#pragma omp atomic
    hist[c] += hist_priv[c];
  }
}
```

Issues

- false sharing for small `m`

Histogram – padding private storage

```
const int paddingBytes = 64;
const int paddingElements = paddingBytes / sizeof(int);
const int mPadded = m + (paddingElements−m%paddingElements);
int hist_priv[nThreads][mPadded];
hist_priv[:][:] = 0;
...
hist_priv[][histIdx[c]]++;
...
```

Performance

Xeon Phi 7120P (61 physical cores) vs. dual-socket Intel Xeon E5-2697 v2 (24 physical cores)

Implementation	CPU time	MIC time	CPU speedup	MIC speedup
Serial	5.06 s	71.3 s	1×	1×
Vectorized serial	1.27 s	9.23 s	3.98×	7.72×
Vectorized parallel	24.0 s	37.7 s	4.74×	1.89×
Removed atomics	0.116 s	0.073 s	43.6×	977×
Removed atomics, m=5	1.6 s	0.72 s	3.16×	99×
Padding to 256 bytes	0.114 s	0.068 s	44.4×	1049×