MODERN CODE PRACTICES AND INTEL® ARCHITECTURE

Part 3 of 3

Colfax International — *colfaxresearch.com*

November 2016

DISCLAIMER

While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.

COURSE ROADMAP

- **Part 1**: Multi-threading strategies
  - Minimizing synchronization
  - Avoiding false sharing
  - Exposing parallelism
- **Part 2**: Vectorization tuning
  - Roles of compiler and developer
  - Tuning with directives
  - Data container optimization
  - Language extensions
- **Part 3**: Memory traffic control
  - Maximizing cache utilization
  - Optimizing memory bandwidth
  - Intel Xeon Phi processors: high-bandwidth memory

§2. INTEL ARCHITECTURE

COMPUTING PLATFORMS

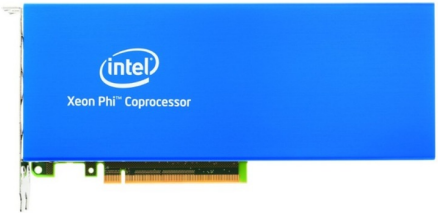
COMPUTING PLATFORMS

Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Multi-Core Architecture

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

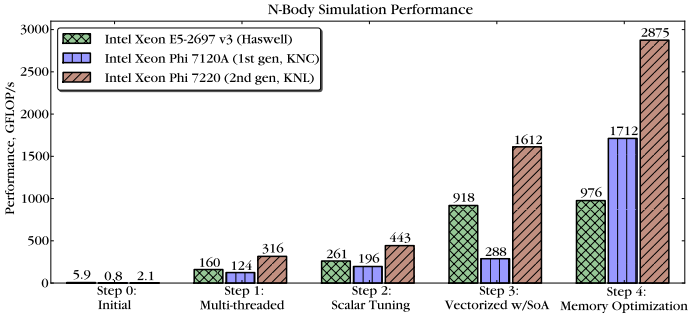
* socket and coprocessor versions

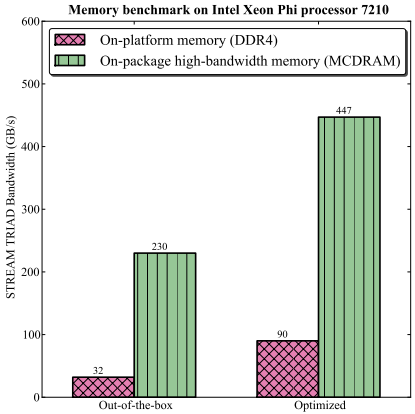
Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture

PERFORMANCE OPTIMIZATION

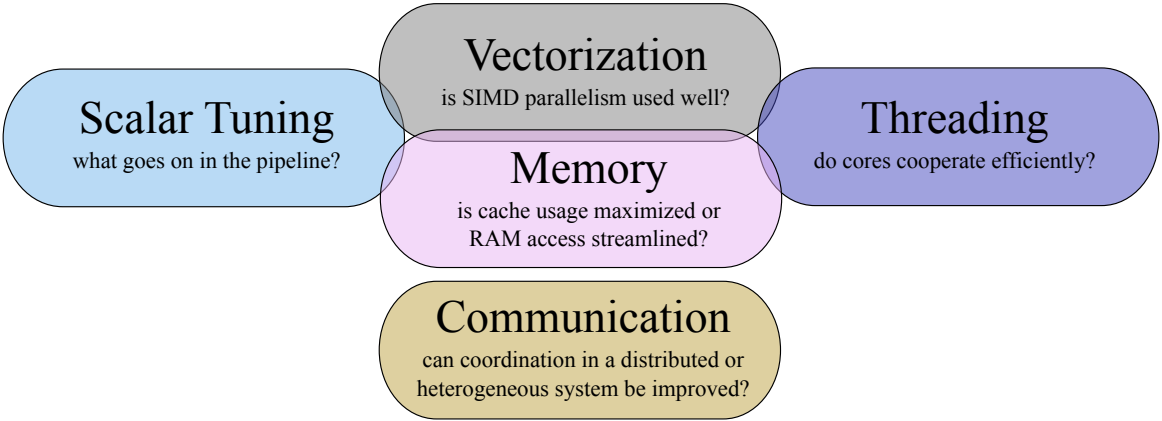
IT TAKES GOOD SOFTWARE TO UNLOCK THE PERFORMANCE!





Details on N-body simulation in Chapter 23 of this book

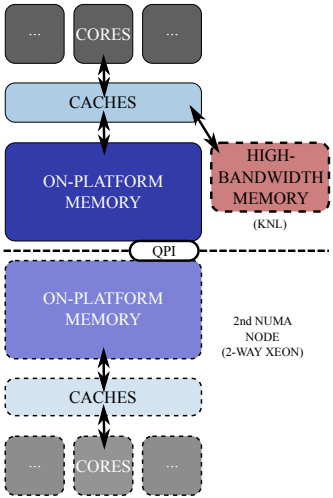
OPTIMIZATION AREAS



OVERVIEW

Optimizing memory traffic:

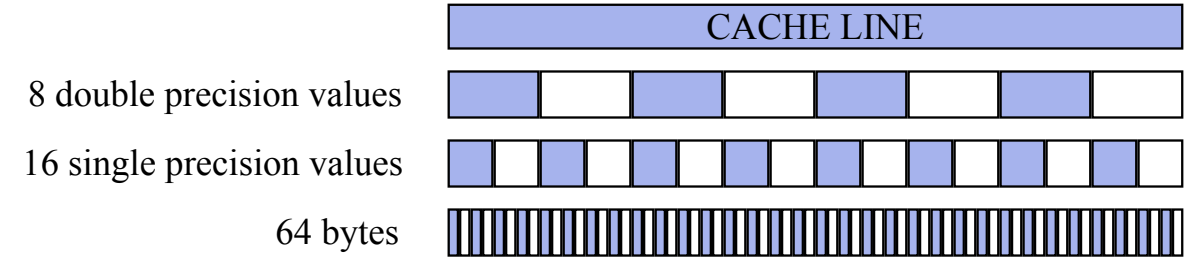
1. Does your algorithm re-use data?
   - 1.1 Loop fusion
   - 1.2 Loop tiling
   - 1.3 Recursion
2. Do you have to go to main memory?
   - 2.1 Unit stride
   - 2.2 Thread affinity
   - 2.3 NUMA locality
   - 2.4 High-bandwidth memory



§3. USING CACHES

CACHE LINES

- Minimal block of data transferred between memory and cache
- 64 bytes long in Intel Architecture
- Aligned on 64-byte boundaries in memory



HOW CHEAP ARE FLOPS?

Theoretical estimates, Intel Xeon E5-2697 V3 processor

$$\text{Performance} = 28\ \text{cores} \times 2.7\ \text{GHz} \times (256/64)\ \text{vec.lanes} \times 2\ \text{FMA} \times 2\ \text{FPU} \approx 1.2\ \text{TFLOP/s}$$

$$\text{Required Data Rate} = 1.2\ \text{TFLOP/s} \times 8\ \text{bytes} \approx 10\ \text{TB/s}$$

$$\text{Memory Bandwidth} = \eta \times 2 \times 59.7 \approx 0.1\ \text{TB/s}$$

$$\text{Ratio} = 10/0.1 \approx 100\ \text{(FLOPs)/(Memory Access)}$$

If the Arithmetic Intensity is...

- > 100 (FLOPs)/(Memory Access) — Compute Bound Application
- < 100 (FLOPs)/(Memory Access) — Bandwidth Bound Application

LOOP FUSION

LOOP FUSION TECHNIQUE

Re-use data in cache by fusing loops in a data processing pipeline

```
1  MyDataType* data = new MyDataType(n);
2
3  for (int i = 0; i < n; i++)
4    Initialize(data[i]);
5
6  for (int i = 0; i < n; i++)
7    Stage1(data[i]);
8
9  for (int i = 0; i < n; i++)
10   Stage2(data[i]);
```

```
1  MyDataType* data = new MyDataType(n);
2
3  for (int i = 0; i < n; i++) {
4
5    Initialize(data[i]);
6
7    Stage1(data[i]);
8
9    Stage2(data[i]);
10 }
```

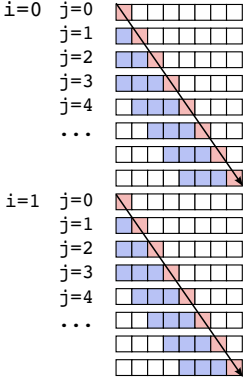
Potential positive side-effect: less data to carry between stages, reduced memory footprint, improved performance (see, e.g., this paper).

LOOP TILING

LOOP TILING

Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];
```



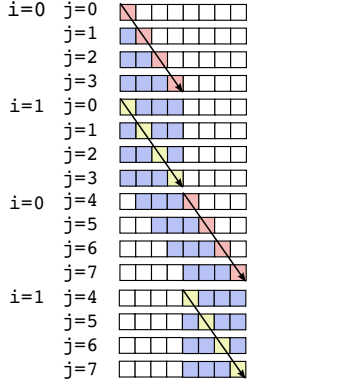
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

Cache size: 4
TILE=4
(must be tuned to cache size)

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

Tiled:

```
for (jj=0; jj<n; jj+=TILE)
  for (i=0; i<m; i++)
    for (j=jj; j<jj+TILE; j++)
      ...=...*b[j];
```



LOOP TILING (CACHE BLOCKING) -- PROCEDURE

```
for (int i = 0; i < m; i++)  // Original code:
  for (int j = 0; j < n; j++)
    compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
// Step 1: strip-mine inner loop
for (int i = 0; i < m; i++)
  for (int jj = 0; jj < n; jj += TILE)
    for (int j = jj; j < jj + TILE; j++)
      compute(a[i], b[j]); // Same order of operation as original
```

```
// Step 2: permute
for (int jj = 0; jj < n; jj += TILE)
  for (int i = 0; i < m; i++)
    for (int j = jj; j < jj + TILE; j++)
      compute(a[i], b[j]); // Re-use to j=jj sooner
```

LOOP TILING (UNROLL-AND-JAM/REGISTER BLOCKING)

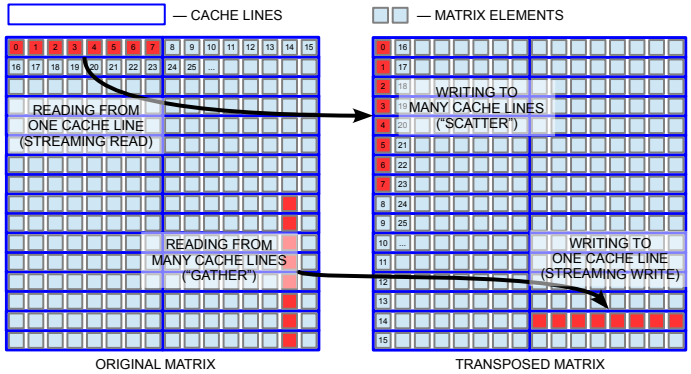
```
1  for (int i = 0; i < m; i++)  // Original code:
2    for (int j = 0; j < n; j++)
3      compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
1  // Step 1: strip-mine outer loop
2  for (int ii = 0; ii < m; ii += TILE)
3    for (int i = ii; i < ii + TILE; i++)
4      for (int j = 0; j < n; j++)
5        compute(a[i], b[j]); // Same order of operation as original
```

```
1  // Step 2: permute and vectorize outer loop
2  for (int ii = 0; ii < m; ii += TILE)
3#pragma simd
4    for (int j = 0; j < n; j++)
5      for (int i = ii; i < ii + TILE; i++)
6        compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times
```

LOOP TILING EXAMPLE: MATRIX TRANSPOSITION

$$B = A^T \quad \Leftrightarrow \quad B_{ij} = A_{ji}$$



See also this paper.

MATRIX TRANSPOSITION

Before:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int j = 0; j < n; j++)
4     B[i*n + j] = A[j*n + i];
```

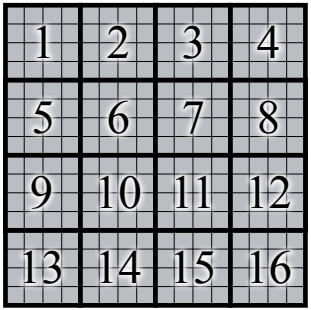
After:

```
1 const int tile = 200;
2 if (n%tile != 0) exit(1);
3
4 #pragma omp parallel for
5 for (int ii=0; ii<n; ii+=tile)
6  for (int jj=0; jj<n; jj+=tile)
7   for (int i=ii; i<ii+tile; i++)
8    for (int j=jj; j<jj+tile; j++)
9     B[i*n + j] = A[j*n + i];
```

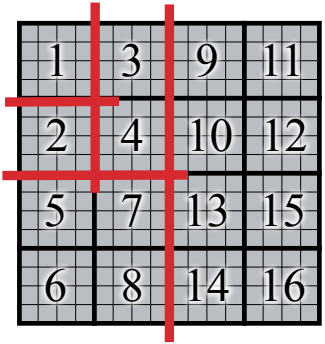
CACHE-OBLIVIOUS RECURSION

MATRIX TRANSPOSITION: TILING AND CACHE-OBLIVIOUS RECURSION

Tiling



Cache-Oblivious Recursion



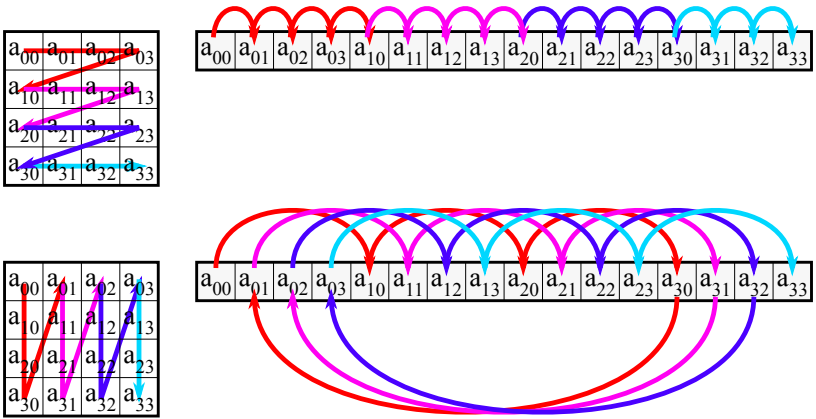
See also this paper.

§4. ACCESSING MAIN MEMORY

UNIT-STRIDE ACCESS

PRINCIPLE

Choose loop order to maintain unit-stride memory access



Compiler may or may not be able to automate loop permutation.

EXAMPLE: OVER-SIMPLIFIED MATRIX-MATRIX MULTIPLICATION

$$C = AB \quad \Leftrightarrow \quad C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$

Before:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int j = 0; j < n; j++)
4 #pragma vector aligned
5     for (int k = 0; k < n; k++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

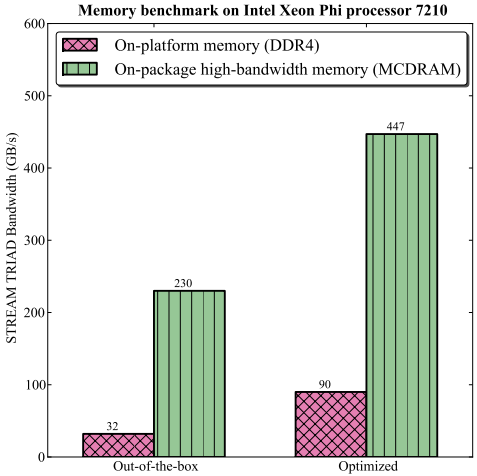
After:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int k = 0; k < n; k++)
4 #pragma vector aligned
5     for (int j = 0; j < n; j++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

THREAD AFFINITY

STREAM BENCHMARK

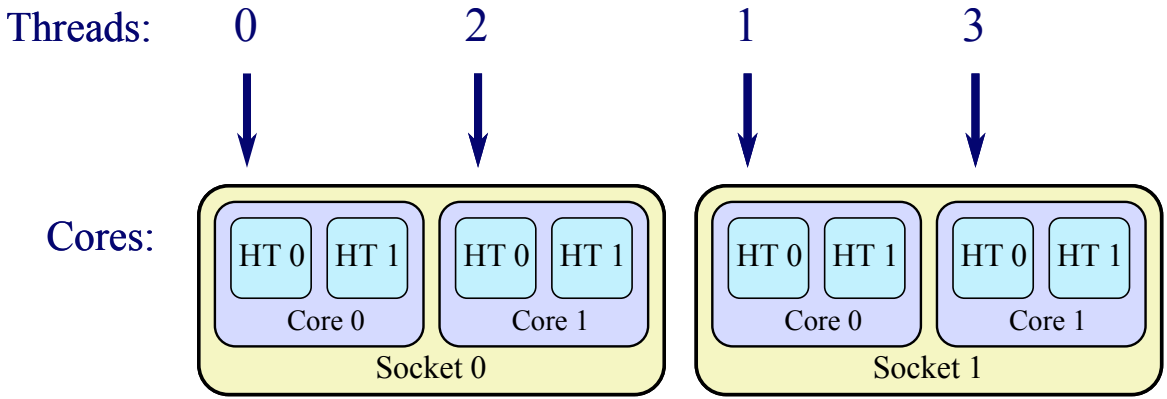
- Industry-standard tool for memory bandwidth measurement
- 4 tests: COPY, ADD, SCALE and TRIAD
- Download from Dr. John McCalpin's site: www.cs.virginia.edu/stream/



THREAD AFFINITY: SCATTER PATTERN

Generally beneficial for bandwidth-bound applications.

```
KMP_AFFINITY=scatter,granularity=fine
```



STREAM BENCHMARK TUNING

- KNL: Compile with `-xMIC-AVX512` (see also HOW Series "KNL")
- Set large enough array size: `-DSTREAM_ARRAY_SIZE=64000000`
- Set 1 thread per core (-1 for offload)
- Xeon CPU: set affinity "scatter" (default on Xeon Phi)
- KNC: Tune prefetching (learn more)

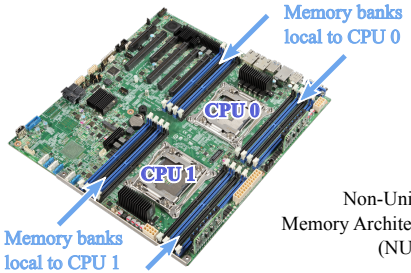
In addition, secret sauce for your own STREAM-like application:

- Parallel first touch (see Session 8 of the HOW Series)
- Essential element – streaming stores: discussion

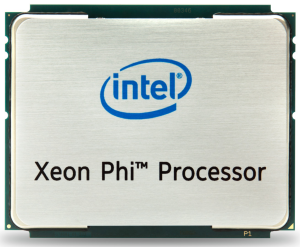
NUMA LOCALITY

NUMA ARCHITECTURES

NUMA = Non-Uniform Memory Access. Cores have fast access to local memory, slow access to remote memory.



Non-Uniform
Memory Architecture
(NUMA)

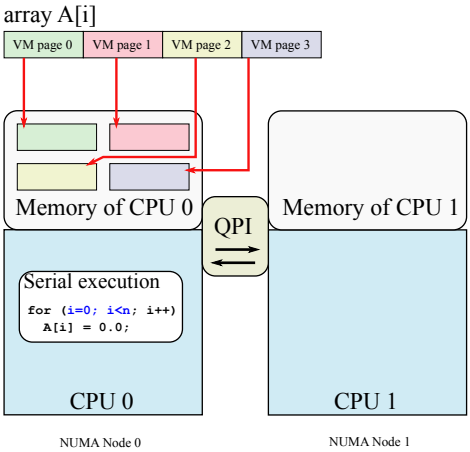


Examples:

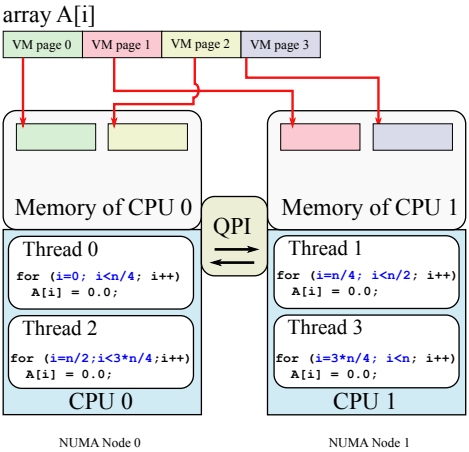
- Multi-socket Intel Xeon processors
- Second generation Intel Xeon Phi in sub-NUMA clustering mode

FIRST-TOUCH ALLOCATION POLICY

Poor First-Touch Allocation



Good First-Touch Allocation



BINDING TO NUMA NODES WITH `numactl`

- `libnuma` – a Linux library for fine-grained control over NUMA policy
- `numactl` – a tool for global NUMA policy control

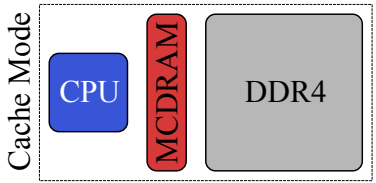
```
vega@lyra% numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 12 13 14 15 16 17
node 0 size: 65457 MB
node 0 free: 24426 MB
node 1 cpus: 6 7 8 9 10 11 18 19 20 21 22 23
node 1 size: 65536 MB
node 1 free: 53725 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
vega@lyra% numactl --membind=<nodes> --cpunodebind=<nodes> ./myApplication
```

HIGH-BANDWIDTH MEMORY

USING HIGH-BANDWIDTH MEMORY (MCDRAM) IN KNL

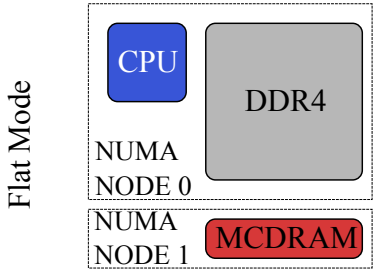
Option 1 : cache/hybrid mode

- Treat it as LLC
- Data locality techniques
- Miss latency 2x the direct DDR4 access



Option 2 : flat mode

- Application fits in 16 GiB? `numactl`
- More than 16 GiB data? Use special allocators (e.g., `memkind`)



HBM IN KNIGHTS LANDING

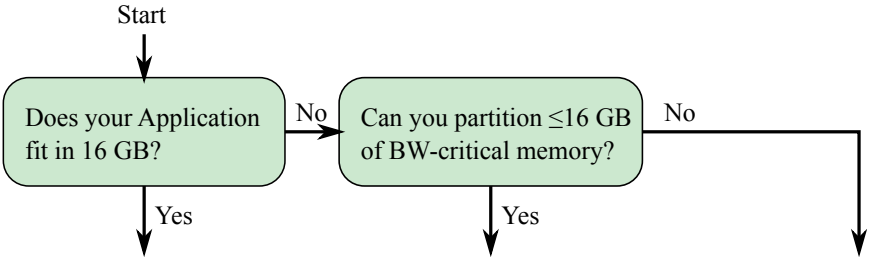
- Finding HBM (MCDRAM) in an Intel Xeon Phi processor x200 (KNL):

```
user@knl% # In Flat mode with All-to-All or Quadrant
user@knl% numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ... 249 250 251 252 253 254 255
node 0 size: 98207 MB
node 0 free: 94798 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15991 MB
```

- Binding the application to HBM (Flat/Hybrid)

```
user@knl% numactl --membind 1 ./runme
// ... Application running in HBM ... //
```

FLOW CHART FOR BANDWIDTH-BOUND APPLICATIONS



numactl	Memkind	Cache mode
▷ Simply run the whole program in MCDRAM ▷ No code modification required	▷ Manually allocate BW-critical memory to MCDRAM ▷ Memkind calls need to be added.	▷ Allow the OS to figure out how to use MCDRAM ▷ No code modification required

§5. LEARN MORE


THE "HOW" SERIES TRAINING
DEEP DIVE
WITH CODE MODERNIZATION EXPERTS
It's free
→ HOWSERIES.COM
*10x 2-hour sessions | 24-hour 2-weeks remote access to a system

HOW SERIES "KNIGHTS LANDING":

PROGRAMMING AND OPTIMIZATION FOR INTEL XEON PHI X200 FAMILY

Free 2-hour video course

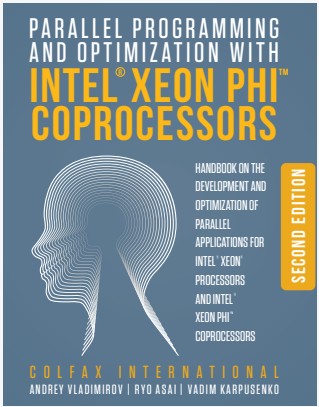
→ COLFAXRESEARCH.COM/HOW-KNL

TEXTBOOK

ISBN: 978-0-9885234-0-1 (508 pages, Electronic or Print)

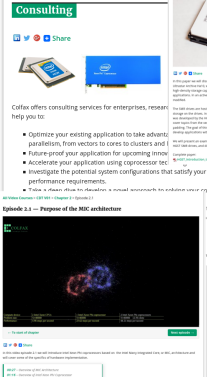
Parallel Programming
and Optimization with
Intel® Xeon Phi™
Coprocessors

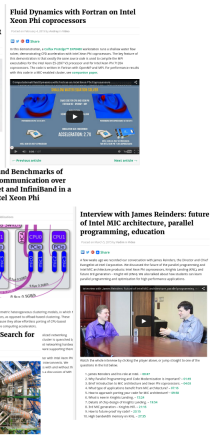
Handbook on the Development and
Optimization of Parallel Applications
for Intel® Xeon® Processors
and Intel® Xeon Phi™ Coprocessors



http://xeonphi.com/book

COLFAX RESEARCH





http://colfaxresearch.com/

DEVELOPER ACCESS PROGRAM (DAP)

Can't wait to get your hands on Knights Landing?

Find out more at dap.xeonphi.com
or contact us at dap@colfax-intl.com

BOOTABLE INTEL XEON PHI PROCESSORS

- Bootable Host Processor
- RHEL/CentOS/SUSE/Win
- 64 cores × 4 HT, 1.3 GHz
- ≤ 384 GiB DDR4, > 90 GB/s
- 16 GiB HBM, > 400 GB/s
- PCIe bus for networking

dap.xeonphi.com

Servers:

Workstations: