PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

The Hands-On Workshop (HOW) Series

Session 10

Colfax International — *colfaxresearch.com*

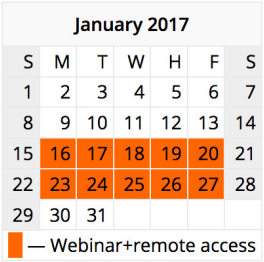
January 2017

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

- Module I. Programming
  - 01. Intel Architecture and Modern Code – Jan 16
  - 02. Xeon Phi, Coprocessors, Omni-Path – Jan 17
- Module II. Expresssing Parallelism
  - 03. Expressing Parallelism with Vectors – Jan 18
  - 04. Multi-threading with OpenMP – Jan 19
  - 06. Distributed Computing, MPI – Jan 20
- Module III. Optimization
  - 06. Optimization Overview: N-body – Jan 23
  - 07. Scalar tuning, Vectorization – Jan 24
  - 08. Common Multi-threading Problems – Jan 25
  - 09. Multi-threading, Memory Aspect – Jan 26
  - 10. Access to Caches and Memory – Jan 27



HOW SERIES ONLINE

Course page:
colfaxresearch.com/how-17-01

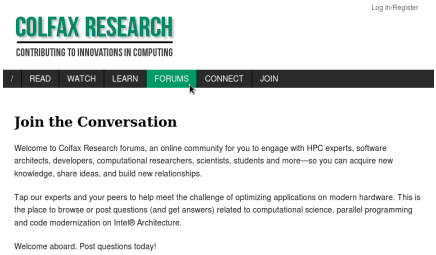
- Slides
- Code
- Video
- Chat

More workshops:
colfaxresearch.com/training

GET YOUR QUESTIONS ANSWERED

Chat (current):
colfaxresearch.com/how-17-01

Forums (technical):
colfaxresearch.com/discussion



Email (organizational):
`training@colfaxresearch.com`

HANDS-ON EXERCISES AND REMOTE ACCESS

- All registrants receive an invitation from cluster@colfaxresearch.com
- Queue-based access to Intel Xeon E5, Intel Xeon Phi (KNC and KNL)
- Can access the cluster the entire 2 weeks of the workshop

§2. REFRESH

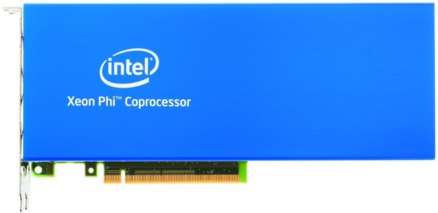
PERFORMANCE OPTIMIZATION

COMPUTING PLATFORMS

Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

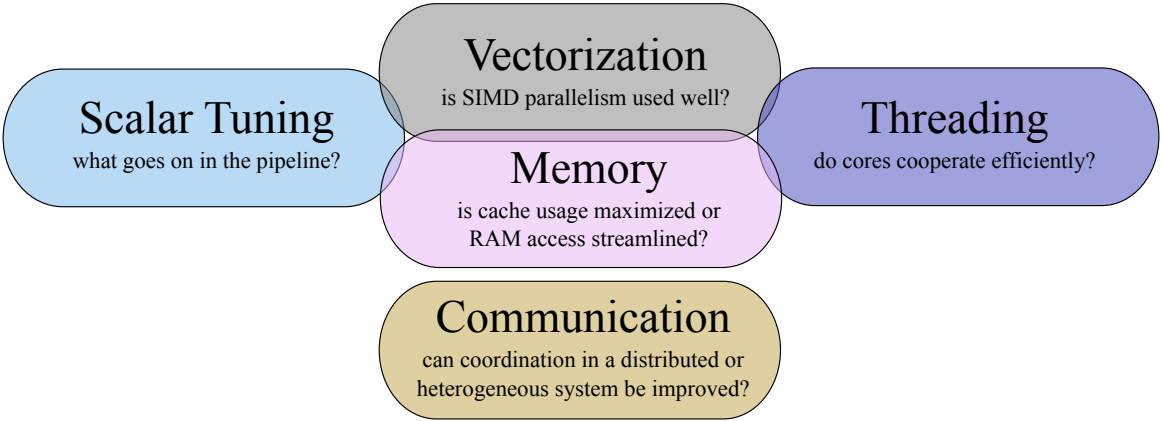
* socket and coprocessor versions

Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

OPTIMIZATION AREAS

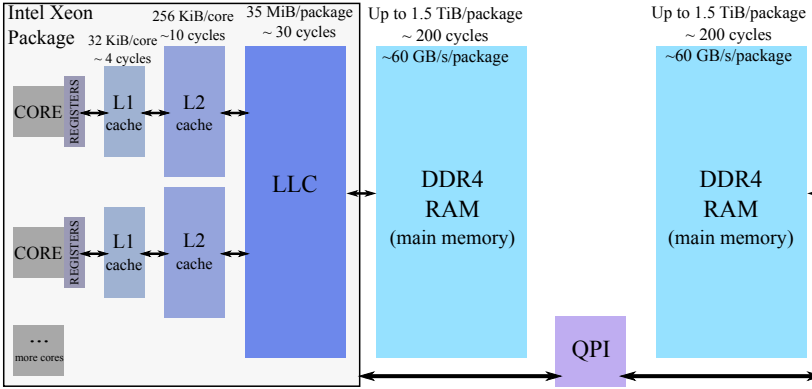


§3. MEMORY TRAFFIC TUNING

MEMORY HIERARCHY

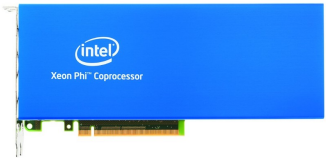
INTEL XEON CPU: MEMORY ORGANIZATION

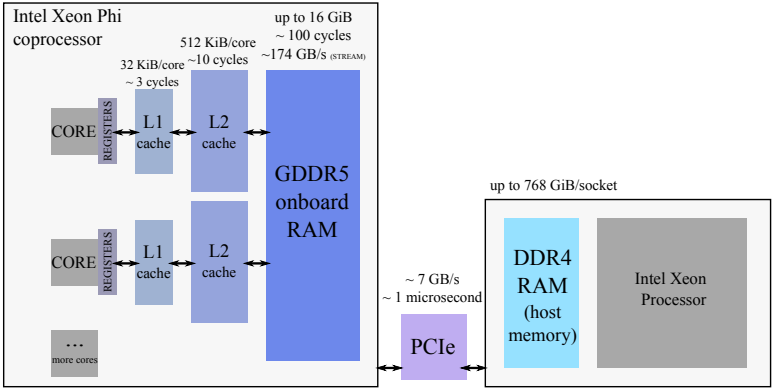
- Hierarchical cache structure
- Two-way processors have NUMA architecture



KNC MEMORY ORGANIZATION

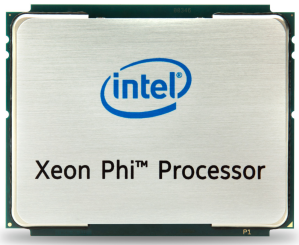
- Direct access to ≤ 16 GiB of cached GDDR5 memory on board
- No access to system DDR4, connected to host via PCIe

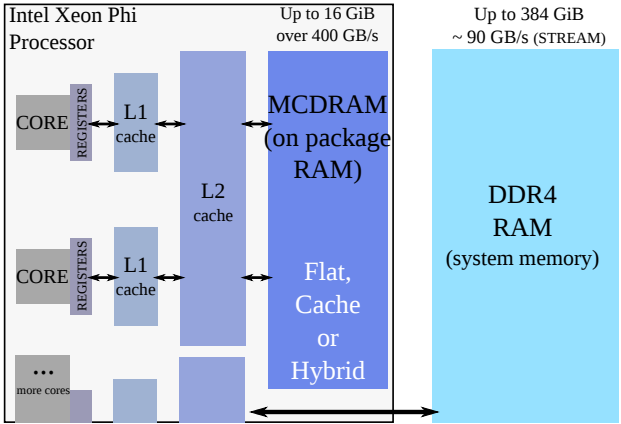




KNL MEMORY ORGANIZATION (BOOTABLE)

- Direct access to on-platform RAM and on-package HBM
- Use HBM as cache, in flat mode, or as hybrid

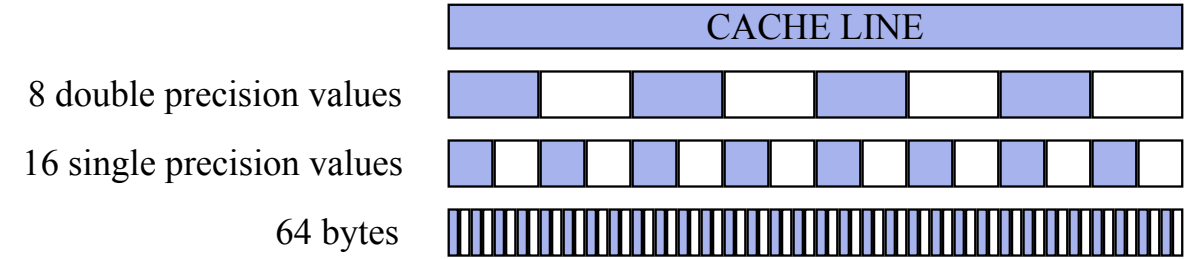




MEMORY HIERARCHY

CACHE LINES

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



MEMORY RE-USE AND ALGORITHMS

LOOP WAS VECTORIZED, NOW WHAT?

1. Ensure unit stride access
2. Align data
3. Pad multi-dimensional containers
4. Eliminate peel loops
5. Eliminate multiversioning
6. **Optimize data re-use in caches**

Good to Know

Vector FLOPs are cheap compared to memory access.

If your data is served by RAM and not caches, it does not matter if you have vectorization: you will be bottlenecked by memory access.

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...

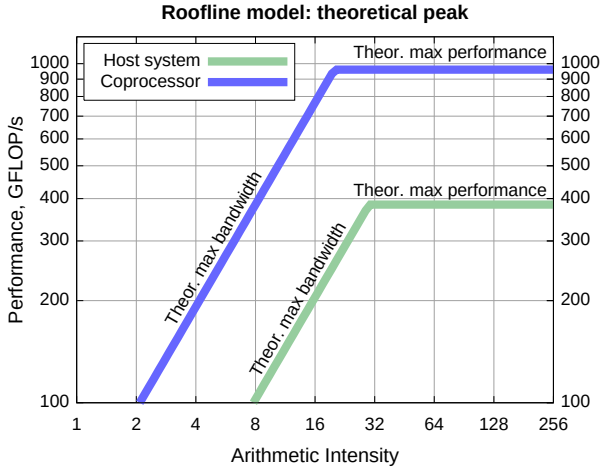
- > 100 (FLOPs)/(Memory Access) — Compute Bound Application
- < 100 (FLOPs)/(Memory Access) — Bandwidth Bound Application

ON COMPUTATIONAL COMPLEXITY OF ALGORITHMS

Type	Properties	Examples
$O(N)$	Each data element is used a fixed number of times. Memory-bound unless the number of times is large.	Array scaling, image brightness adjustment, vector dot-product.
$O(N^{\alpha})$	Each element is used $N^{\alpha-1}$ times. A lot of data reuse for $\alpha > 1$. Good implementation can be compute-bound, poor one – memory-bound.	Matrix-matrix multiplication: $O\left(N^{3/2}\right)$ (N = amount of data in matrix), direct N-body calculation: $O\left(N^2\right)$
$O(N \log N)$	Each element is used $\log N$ times. For small problems – memory-bound, for very large problems transitions to compute-bound	Fast Fourier transform, merge sort
$O(\log N)$	Always memory-bound.	Binary search

N = data size

ARITHMETIC INTENSITY AND ROOFLINE MODEL

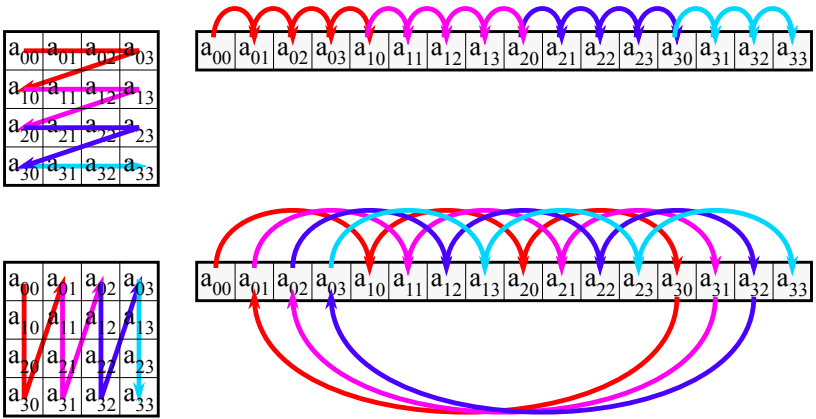


More on roofline model: Williams et al.

LOOP PERMUTATION

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];
```

PRINCIPLE

- The order of nested loops must be chosen for best locality of data access
- At `-O2` and above, the compiler automatically interchanges loops in some cases
- In other cases, loop interchange must be investigated manually

LOOP FUSION

LOOP FUSION TECHNIQUE

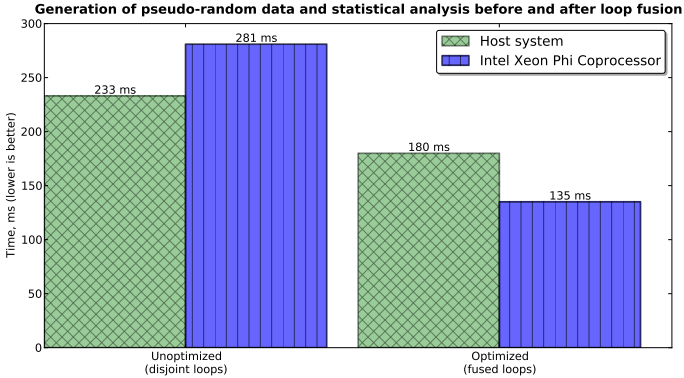
Re-use data in cache by fusing loops in a data processing pipeline

```
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).

EXAMPLE APPLICATION -- PERFORMANCE



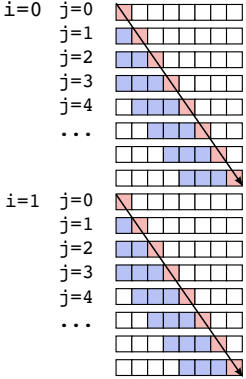
See `labs/4/4.09-memory-loop-fusion-statistics/`

LOOP TILING

LOOP TILING

Original:

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



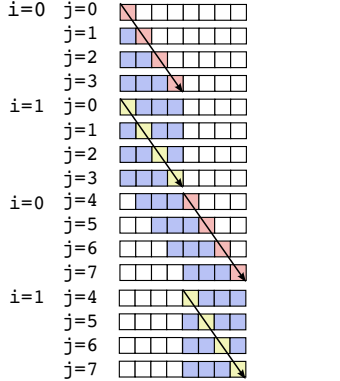
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

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 (UNROLL-AND-JAM) -- ALTERNATIVE IMPLEMENTATION

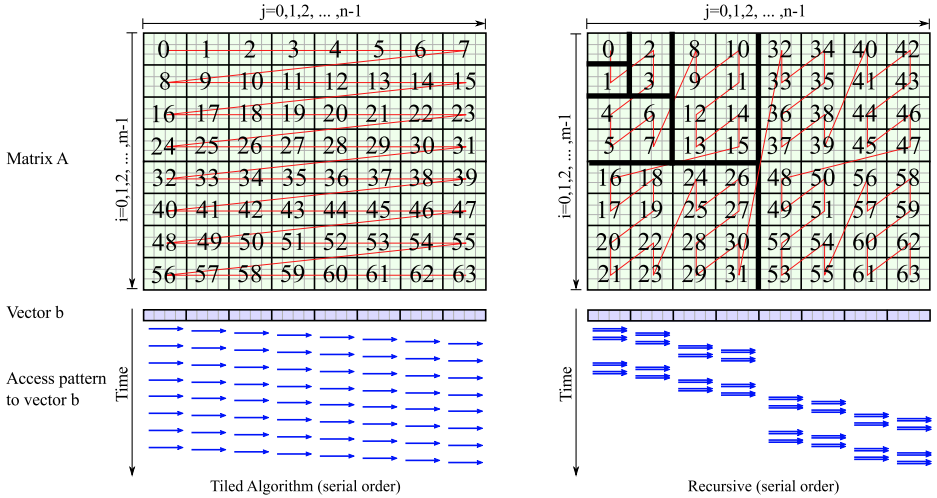
```
1  for (int i = 0; i < m; i++)   // Original code:
2    for (int j = 0; j < n; j++)
3      compute(a[i], b[j]); // Memory access is unit-stride in j
```

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

```
1  // Step 2: permute middle two loops
2  for (int ii = 0; ii < m; ii += TILE)
3    for (int jj = 0; jj < n; jj += VECLEN)
4      for (int i = ii; i < ii + TILE; i++)
5        for (int j = jj; j < jj + VECLEN; j++)
6          compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times
```

CACHE-OBLIVIOUS RECURSION

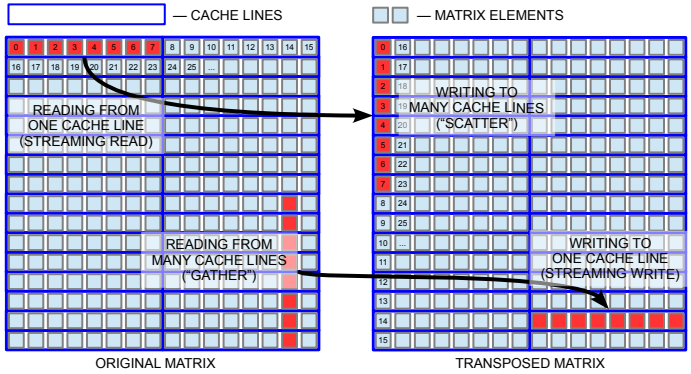
PRINCIPLE



EXAMPLE 1: MATRIX TRANSPOSITION, TILING

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];
```

EXAMPLE 2: MATRIX-VECTOR MULTIPLICATION, TILING

EXAMPLE: MATRIX-VECTOR MULTIPLICATION

$$c_i = \sum_{j=0}^{n} A_{ij} b_j, \quad i = 0, 1, \ldots, (m-1). \tag{1}$$

```
1  void Multiply(const double* const A, const double* const b,
2                double* const c, const long n, const long m){
3    assert(n%64 == 0);
4  #pragma omp parallel for
5    for (long i = 0; i < m; i++)
6  #pragma vector aligned
7      for (long j = 0; j < n; j++) // Each value of A[i*n+j] is used only once
8        c[i] += A[i*n+j] * b[j];   // Each value of b[j] is used a total of m times
9  }
```

Non-optimal performance due to inefficient cache use

APPLYING TILING

```
  const long jTile = 4096L; assert(n%jTile == 0);
#pragma omp parallel
  {
    double temp_c[m] __attribute__((aligned(64)));
    temp_c[:] =0;
#pragma omp for
    for (long jj =0; jj < n; jj+=jTile) // Loop Tiling in j
      for (long i = 0; i < m; i++)
#pragma vector aligned
        for (long j =jj; j < jj+jTile; j++)
          temp_c[i] += A[i*n+j] * b[j];

    for(long i = 0; i < m; i++) { // Reduction
#pragma omp atomic
      c[i]+= temp_c[i];
} } }
```

CACHE BLOCKING + STRIP-MINE AND COLLAPSE

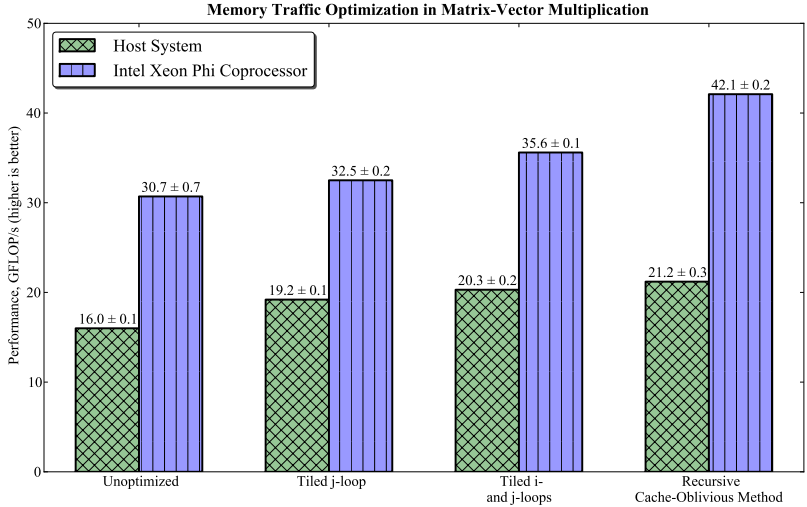
```
1   const long iTile = 64L;   assert(m%iTile == 0);
2   const long jTile = 4096L; assert(n%jTile == 0);
3  #pragma omp parallel
4   {
5     double temp_c[m] __attribute__((aligned(64))); temp_c[:] =0;
6  #pragma omp for collapse(2)
7     for (long ii = 0; ii < m; ii += iTile)
8       for (long jj = 0; jj < n; jj += jTile)
9         for (long i = ii; i < ii+iTile; i++)
10 #pragma vector aligned
11          for (long j =jj; j < jj+jTile; j++)
12            temp_c[i] += A[i*n+j] * b[j];
13
14    for(long i = 0; i < m; i++) {
15 #pragma omp atomic
16      c[i]+= temp_c[i];
17 } } }
```

EXAMPLE 3: MATRIX-VECTOR MULTIPLICATION, RECURSION

EXAMPLE: MATRIX-VECTOR MULTIPLICATION

```
1  void RecursMultiply(const double* const A, const double* const b,
2        double* const c, const long n, const long m, const long lda){
3    const long jThreshold = 8192L; assert(n%jThreshold == 0);
4    const long iThreshold = 64L;   assert(m%iThreshold == 0);
5    if ((m<=iThreshold) && (n<=jThreshold)) { // Recursion threshold
6      // .... Base Case: Compute the result inside the tile ... //
7    } else { // Recursive divide-and-conquer
8      if (m*jThreshold > n*iThreshold) { // Split i-wise
9        double c1[m/2] __attribute__((aligned(64)));
10 #pragma omp task
11       { RecursMultiply(&A[0*lda + 0], &b[0], c1, n, m/2, lda); }
12       double c2[m/2] __attribute__((aligned(64)));
13       RecursMultiply(&A[(m/2)*lda + 0], &b[m/2], c2, n, m/2, lda);
14 #pragma omp taskwait
15       c[0:m/2] += c1[0:m/2]; c[m/2:m/2] += c2[0:m/2]; // Reduction
16     } else { // .... Split j-wise .... // }
17 } }
```

PERFORMANCE OF MATRIX VECTOR MULTIPLICATION

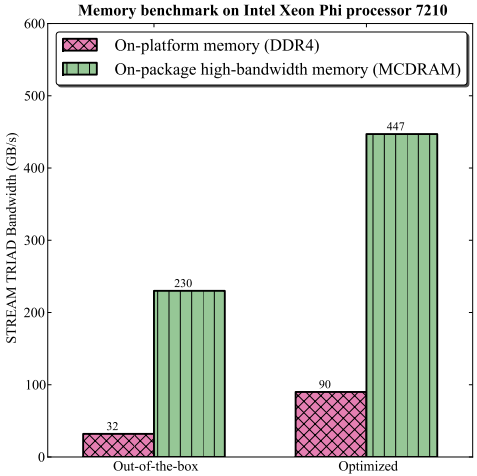


EXAMPLE 3: MATRIX-VECTOR MULTIPLICATION, RECURSION

BANDWIDTH TUNING

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/



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)

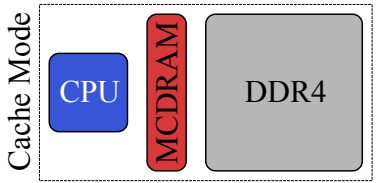
In addition, secret sauce for your own STREAM-like application:

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

USING HIGH-BANDWIDTH MEMORY (MCDRAM) IN KNL

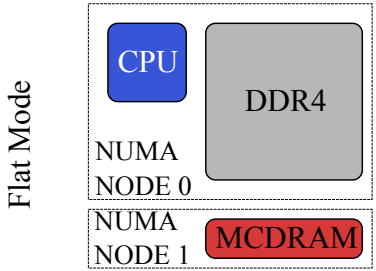
Option 1 : cache/hybrid mode

- 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`)



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
```

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 ... //
```

§4. REVIEW AND WHAT'S NEXT

SUMMARY

Memory Optimization:

1. Data re-use in caches: increase arithmetic intensity
   - 1.1 Loop permutation: achieve unit-stride access
   - 1.2 Loop fusion: re-use data as soon as possible
   - 1.3 Loop tiling: cache blocking and unroll-and-jam
   - 1.4 Cache-oblivious recursion: portable
2. Access to main memory:
   - 2.1 1 thread/core, “scatter” affinity
   - 2.2 “Secret sauce” compiler arguments for KNC
   - 2.3 First-touch allocation in NUMA systems
   - 2.4 High-bandwidth memory (HBM) in KNL: numactl or memkind

§5. SUMMARY

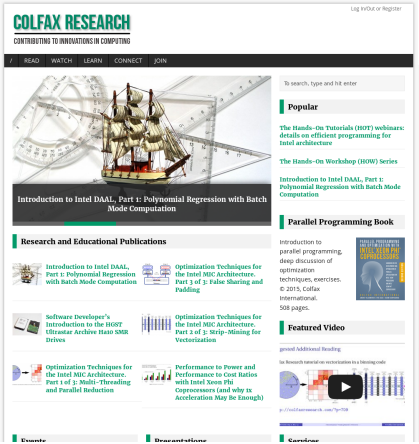
WHAT WE LEARNED

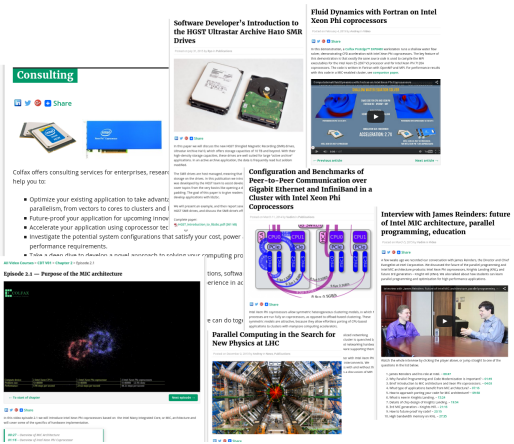
Session 1	Intel Architecture, Colfax Cluster
Session 2	Programming Xeon Phi: native, offload, HBM, OPA
Session 3	Expressing vectorization
Session 4	Expressing thread parallelism (OpenMP)
Session 5	Distributed computing (MPI)
Session 6	Optimization overview (N-body)
Session 7	Optimizing scalar component and vectorization
Session 8	Optimizing multi-threading (common errors)
Session 9	Optimizing multi-threading (memory aspect)
Session 10	Optimizing memory access

YOUR TURN

Let us know what you think via email!

COLFAX RESEARCH





http://colfaxresearch.com/