Programming and Optimization for Intel® Architecture

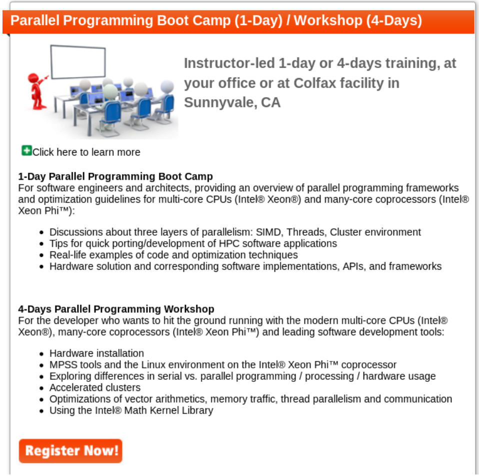
The Hands-On Workshop (HOW) Series

Colfax International — @colfaxintl

April 2016 , Rev. 02b

About This Document

This document represents the materials of a Web-based training "Programming and Optimization with Intel Architecture" developed and run by Colfax International.



colfaxresearch.com/how-series

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

1. Why Intel Parallel Architectures?
   - Parallelism and specialization – April 18
   - Programming model continuity – April 18
2. Programming models for Xeon Phi coprocessors
   - Native programming – April 18
   - Offload programming – April 19
3. Expressing Parallelism
   - Introduction to vectorization – April 20
   - Crash-course on OpenMP – April 21
4. Optimization – intro on April 22
   - Vectorization tuning – April 25
   - Multi-threading – April 26, 27
   - Memory traffic – April 28
5. Distributed Computing: MPI – April 29

April 2016

S	M	T	W	H	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

— Lecture+remote access

May 2016

S	M	T	W	H	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

— Self-study/remote access

HOW Online

Course page: colfaxresearch.com/how-16-04

- Slides (including this one), code downloads
- Video of recorded sessions o
- Chat (during webinars or offline)

Additional resources:

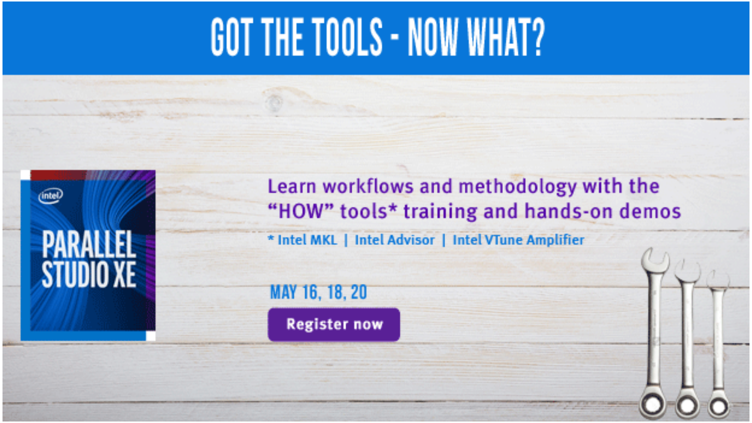
- More workshops like this one: colfaxresearch.com/training
- Video courses: colfaxresearch.com/video-courses
- Intel Many Integrated Core Architecture Forum

HOW Series "Deep Dive" in May

Wish you had joined us earlier? Want to recomment the HOW series to a friend? Another HOW Series run coming up in May.

colfaxresearch.com/how-series/

HOW Series "Tools"



http://colfaxresearch.com/how-tools/

Developer's Guide to Knights Landing

colfaxresearch.com/knl-webinar/

§2. Refresh

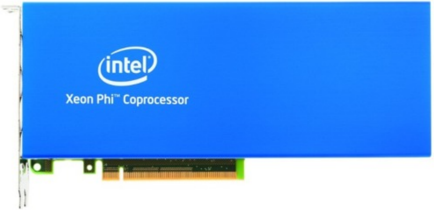
Performance Optimization

Computing Platforms

Intel Xeon
Processor

Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi
Corocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi
Processor, 2nd generation*

* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

Optimization Areas

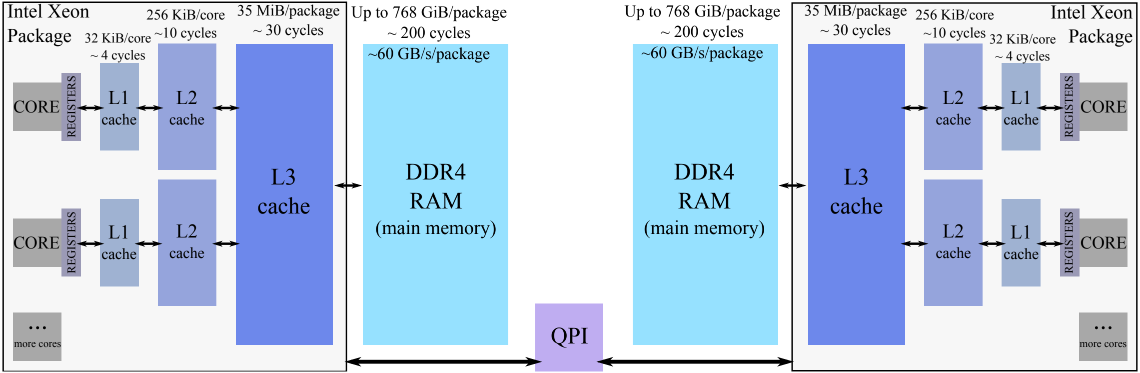
1. **Scalar optimization** (compiler-friendly practices)
2. **Vectorization** (must use 16- or 8-wide vectors)
3. **Multi-threading** (must scale to 100+ threads)
4. **Memory access** (streaming access or tiling)
5. **Communication** (offload, MPI traffic control)

§3. Memory Traffic Tuning

Memory Hierarchy

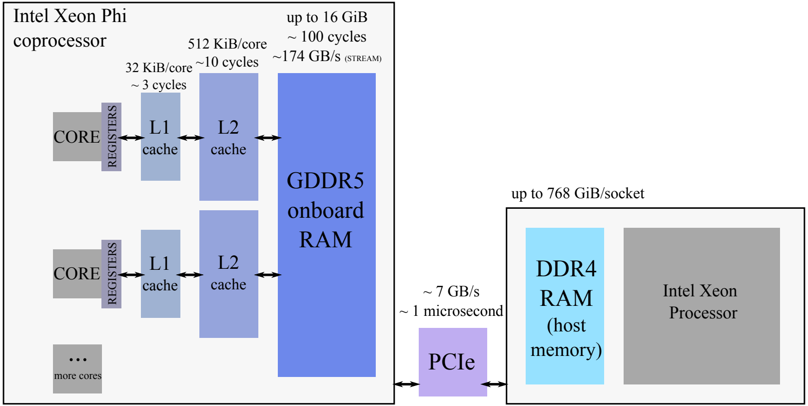
Intel Xeon CPU: Memory Organization

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



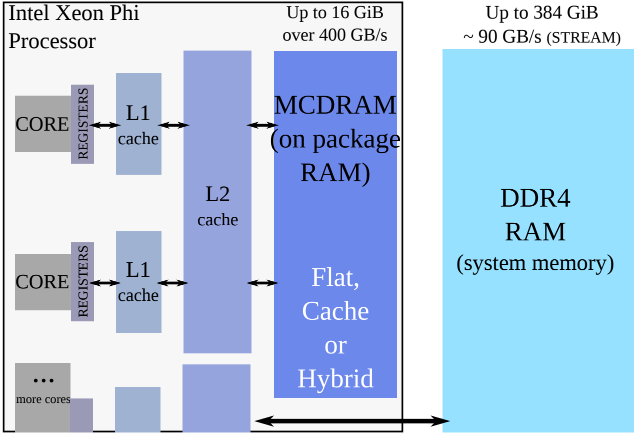
KNC Memory Organization

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



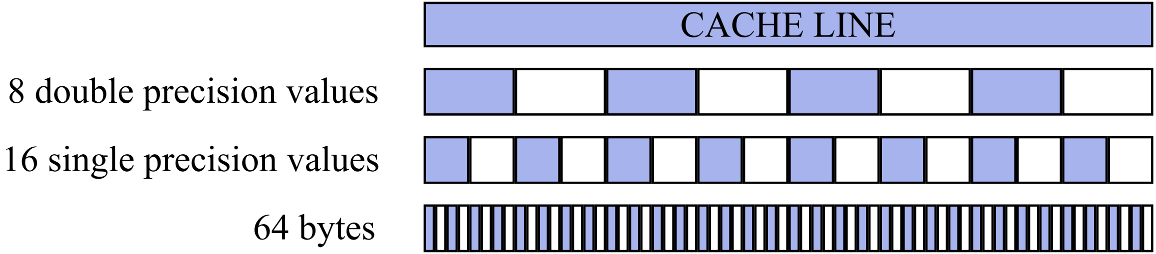
KNL Memory Organization

- Direct access to on-package MCDRAM *and* system DDR4 (socket)
- Use MCDRAM as cache, in flat mode, or as hybrid



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

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

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

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

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

If the Arithmetic Intensity is...

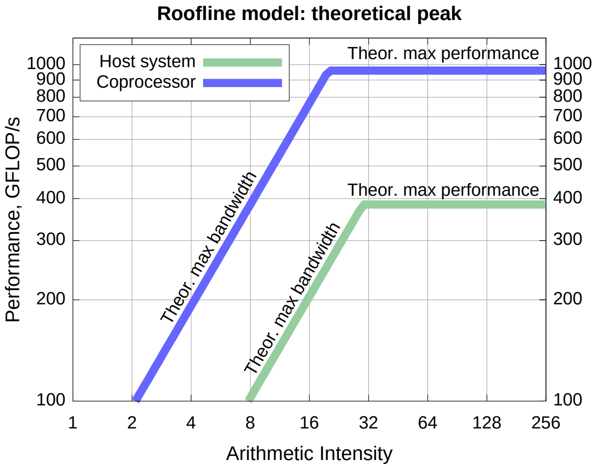
- $>$ 100 (FLOPs)/(Memory Access) — Compute Bound Application
- $<$ 100 (FLOPs)/(Memory Access) — Bandwidth Bound Application

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 $\alpha \gtrsim 2$. 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

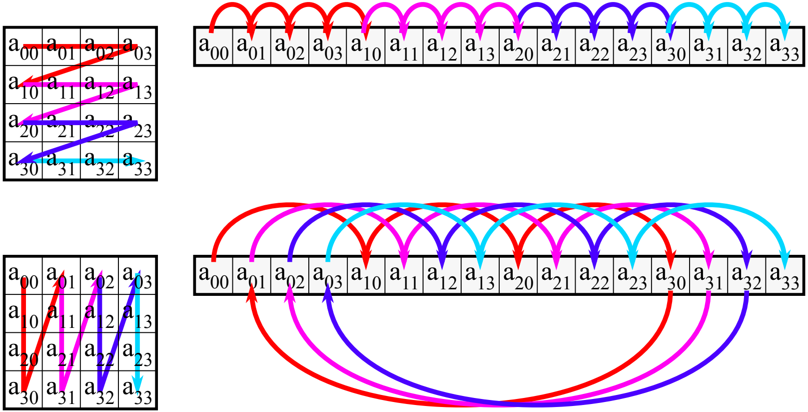


More on roofline model: Williams et al.

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:

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

After:

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
  for (int k = 0; k < n; k++)
#pragma vector aligned
    for (int j = 0; j < n; j++)
      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

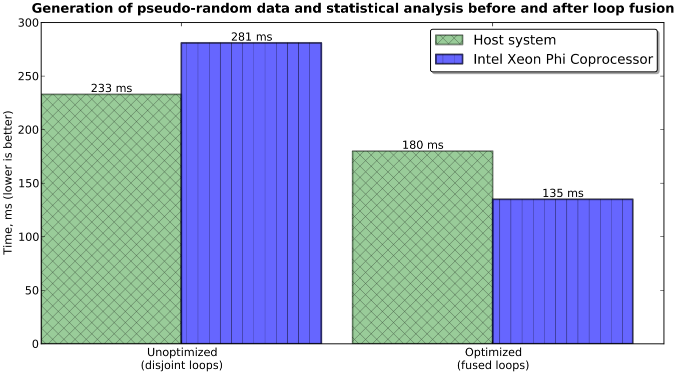
Re-use data in cache by fusing loops in a data processing pipeline

```
MyDataType* data = new MyDataType(n);

for (int i = 0; i < n; i++)
  Initialize(data[i]);

for (int i = 0; i < n; i++)
  Stage1(data[i]);

for (int i = 0; i < n; i++)
  Stage2(data[i]);
```

```
MyDataType* data = new MyDataType(n);

for (int i = 0; i < n; i++) {

  Initialize(data[i]);

  Stage1(data[i]);

  Stage2(data[i]);
}
```

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

Example Application – Performance



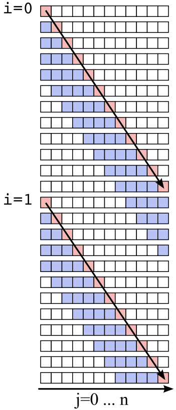
See `labs/4/4.09-memory-loop-fusion-statistics/`

Loop Tiling

Loop Tiling

Original:

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



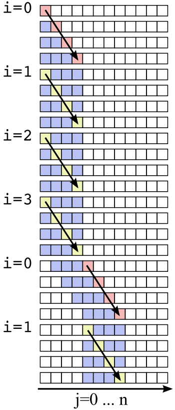
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

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

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 75%

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; j += 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; j += 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)

```
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 outer loop
for (int ii = 0; ii < m; ii += TILE)
  for (int i = ii; i < ii + TILE; i++)
    for (int j = 0; j < n; j++)
      compute(a[i], b[j]); // Same order of operation as original
```

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

Loop Tiling (Unroll-and-Jam) – Alternative Implementation

```
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
```

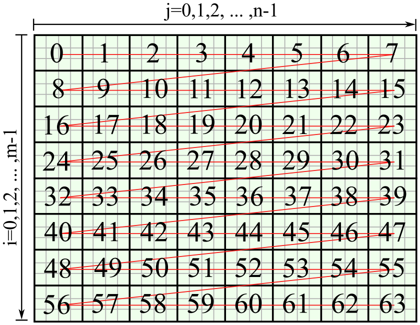
```
// Step 1: strip-mine both loops
for (int ii = 0; ii < m; ii += TILE)
  for (int i = ii; i < ii + TILE; i++)
    for (int jj = 0; jj < n; jj += VECLEN)
      for (int j = jj; jj < jj + VECLEN; j++)
        compute(a[i], b[j]); // Same order of operation as original
```

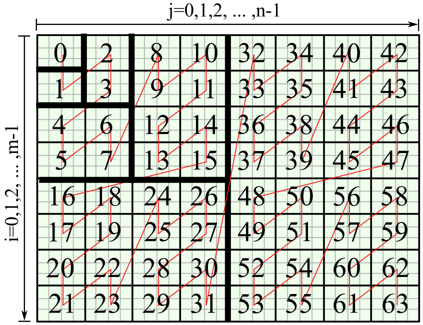
```
// Step 2: permute middle two loops
for (int ii = 0; ii < m; ii += TILE)
  for (int jj = 0; jj < n; jj += VECLEN)
    for (int i = ii; i < ii + TILE; i++)
      for (int j = jj; j < jj + VECLEN; j++)
        compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times
```


Cache-Oblivious Recursion

Principle

Matrix A

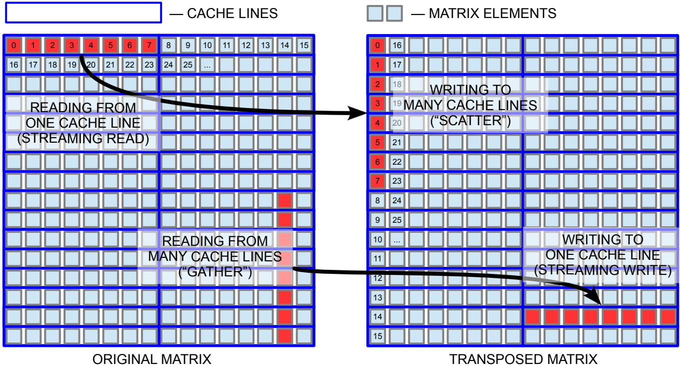




Example 1: Matrix Transposition, Tiling

Example: Matrix Transposition

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



See also this paper.

Matrix Transposition

Before:

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

After:

```
const int tile = 200;
if (n%tile != 0) exit(1);

#pragma omp parallel for
for (int ii=0; ii<n; ii+=tile)
 for (int jj=0; jj<n; jj+=tile)
  for (int i=ii; i<ii+tile; i++)
   for (int j=jj; j<jj+tile; j++)
    B[i*n + j] = A[j*n + i];
```

Example 2: Matrix-Vector Multiplication, Tiling

Example: Matrix-vector Multiplication

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

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

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

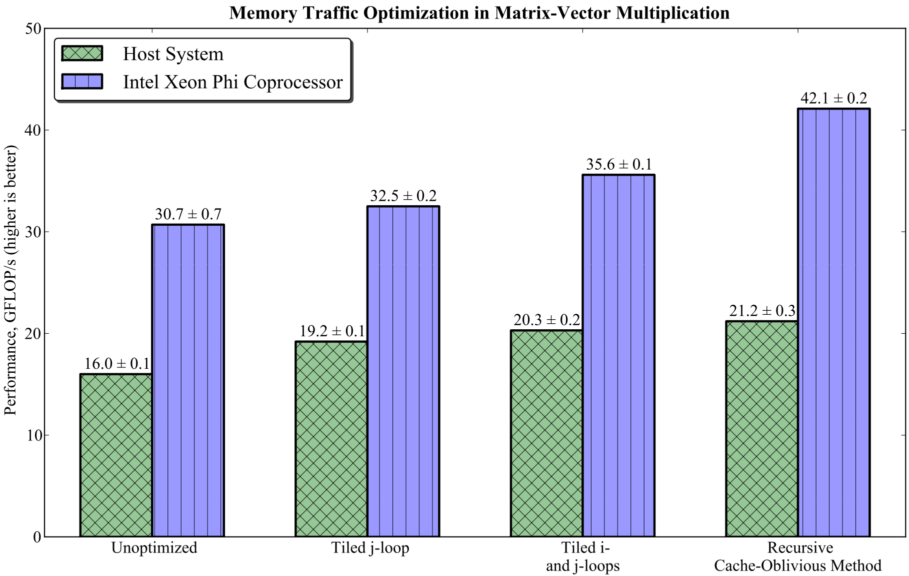
```
  const long iTile = 64L;   assert(m%iTile == 0);
  const long jTile = 4096L; assert(n%jTile == 0);
#pragma omp parallel
  {
    double temp_c[m] __attribute__((aligned(64))); temp_c[:] =0;
#pragma omp for collapse(2)
    for (long ii = 0; ii < m; ii += iTile)
      for (long jj = 0; jj < n; jj += jTile)
        for (long i = ii; i < ii+iTile; 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++) {
#pragma omp atomic
      c[i]+= temp_c[i];
} } }
```

Example 3: Matrix-Vector Multiplication, Recursion

Example: Matrix-Vector Multiplication

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

Performance of Matrix Vector Multiplication

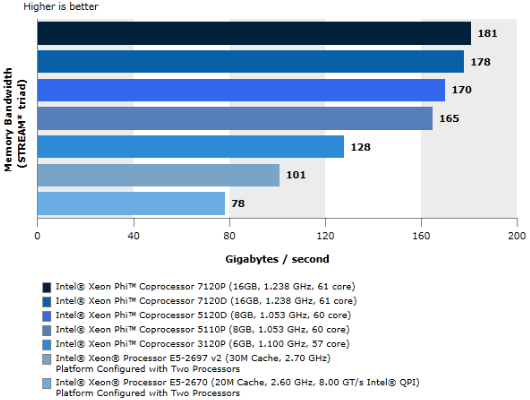


Bandwidth Tuning

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/

Expected for KNL: ≈400 GB/s



STREAM Benchmark on Tuning

- Set 1 thread per core on all platforms (-1 for offload)
- Set affinity "scatter": white paper (Colfax Research)
- Tune prefetching: white paper (Intel)

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

- Parallel first touch (see NUMA discussion in Session 08)
- Essential element – streaming stores: discussion

§4. Review and What's Next

Summary

This session:

1. Memory traffic optimization: spatial and temporal data locality
2. Loop tiling: cache blocking and unroll-and-jam
3. Cache-oblivious recursion
4. First-touch allocation in NUMA systems
5. Loop fusion

Next session: distributed computation with MPI, specifics of message passing with coprocessors.