From Scalar & Serial to Vector & Multi-Threaded

The Hands-On Tutorial (HOT) Series

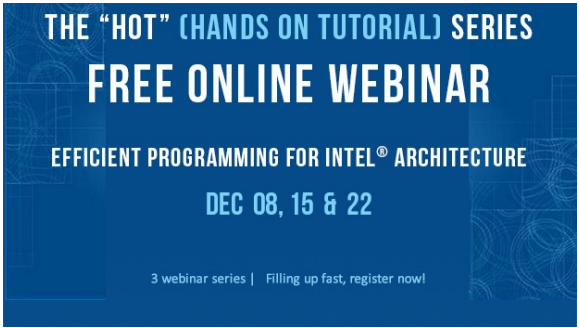
Andrey Vladimirov, PhD — Colfax International
colfaxresearch.com

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.

About the Series

Hands-On Tutorial (HOT) series: webinars on efficient programming for the Intel architecture. Select topics of parallel programming and performance optimization with detailed practical demonstrations.



colfaxresearch.com/hot-1512

Slides, Code, Video

You can download slides, code and watch the video recording of this webinar here (requires registration for a free Colfax Research account):

colfaxresearch.com/hot-1512

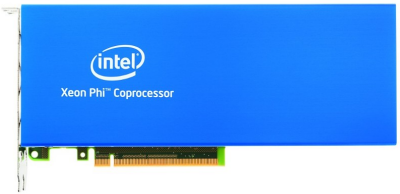
Next webinar on December 15, 2015: “Finding the Low-Hanging Fruit for Optimization”:

Register

§2. Intel Architecture

Intel Xeon Phi Coprocessors and the MIC Architecture

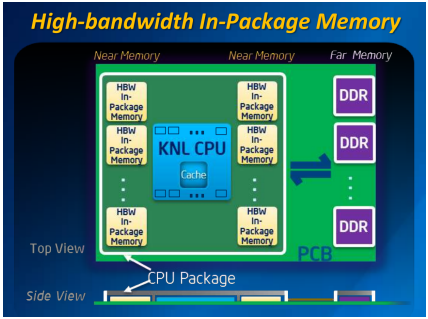
- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- ≤18 cores/socket ≈3 GHz
- 2-way hyper-threading
- 256-bit AVX vectors



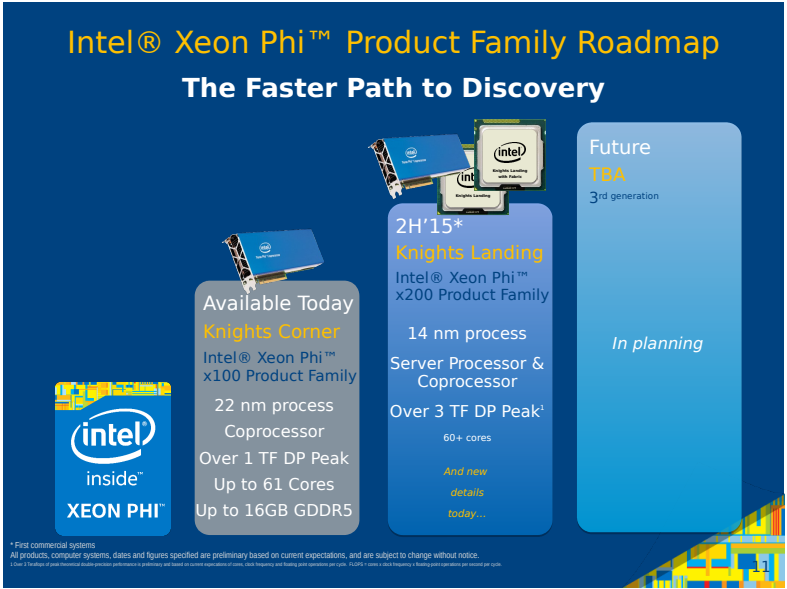
- C/C++/Fortran; OpenMP/MPI
- Special Linux distribution
- 6–16 GB cached GDDR5 RAM
- 57 to 61 cores at ≈1 GHz
- 4 hardware threads per core
- 512-bit IMCI vectors

Next Generation of the MIC Architecture

- 2nd generation MIC product: code name Knights Landing (KNL)
- Intel's 14 nm manufacturing process
- A processor (running the OS) or a coprocessor (PCIe device)
- On-package high-bandwidth memory w/flexible memory models: flat, cache, & hybrid
- Intel Advanced Vector Extensions AVX-512 (public)

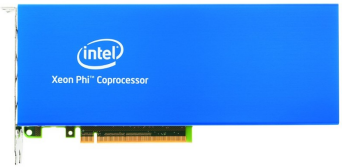


Source: Intel Newsroom



Source: https://www.brighttalk.com/webcast/10773/116329

“Standard Candle” Testbench



vs.

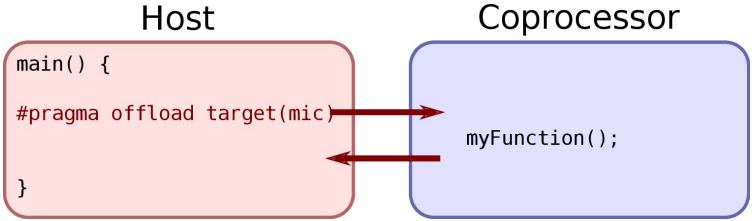
One Intel Xeon Phi 7120P coprocessor (2012)
TDP: 300 W, RCP: $4129

Two Intel Xeon E5-2697 v3 CPUs (2014)
TDP: 290 W, RCP: $5404

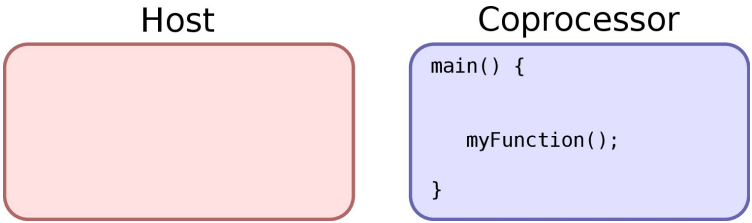
See also “Intel Xeon Product Family: Performance Brief”

Offload and Native models

- Offload model (explicit/virtual-shared memory/OpenMP 4.0):

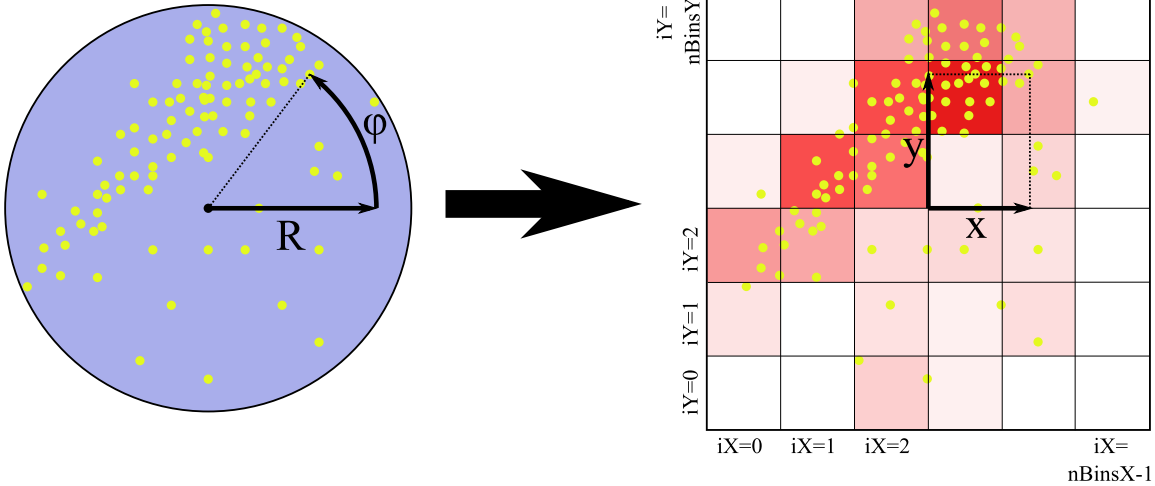


- Native model (standalone application/MPI process):



§3. Example Problem: Binning

Example Problem: Binning

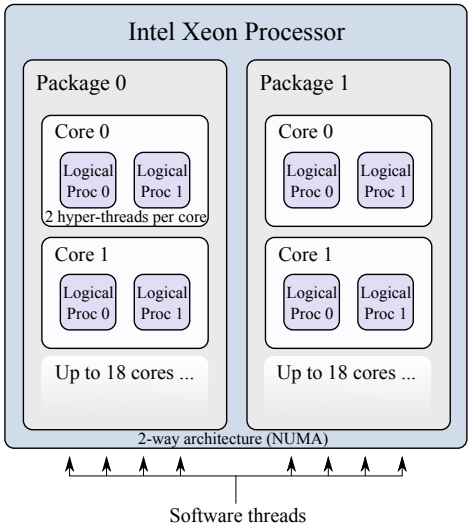


Initial Approach

```
1  // Reference implementation: scalar, serial code without optimization
2  void BinParticlesReference(const InputDataType & inputData,
3                                        BinsType & outputBins) {
4    // Loop through all particle coordinates
5    for (int i = 0; i < inputData.numDataPoints; i++) {
6      // Transforming from cylindrical to Cartesian coordinates:
7      const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
8      const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);
9
10     // Calculating the bin numbers for these coordinates:
11     const int iX = int((x - xMin)*binsPerUnitX);
12     const int iY = int((y - yMin)*binsPerUnitY);
13
14     // Incrementing the appropriate bin in the counter:
15     outputBins[iX][iY]++;
16   }
17 }
```

§4. Multi-Threading: Happy Cores

Cores and Threads

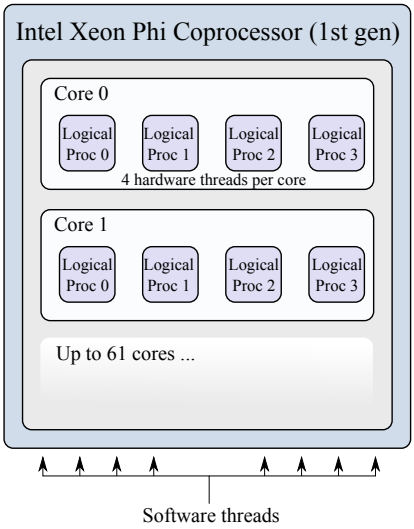


Hierarchy:
Packages ->
Cores ->
Logical Processors

Core presents itself as several logical processors due to one of:
a) hyper-threading (Xeon)
b) hardware threads (Xeon Phi)

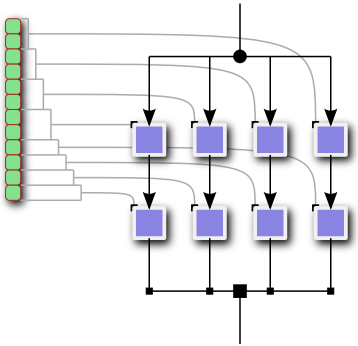
Terminology:
OS Proc - numeric ID of logical processor
Software thread - stream of instructions in a process

Jargon:
"physical core" = core
"logical core" = logical processor



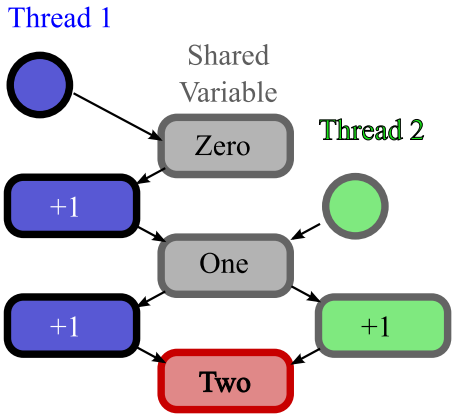
OpenMP Threads

```
1  // Parallel, but INCORRECT implementation
2  void BinParticlesReference(
3     const InputDataType & inputData,
4                 BinsType & outputBins) {
5  // Distribute loop iterations across threads
6  #pragma omp parallel for
7    for (int i=0; i<inputData.numDataPoints; i++)
8      {
9        // ...
10       // Incrementing the appropriate
11       // bin in the counter:
12       outputBins[iX][iY]++;
13     }
14 }
```

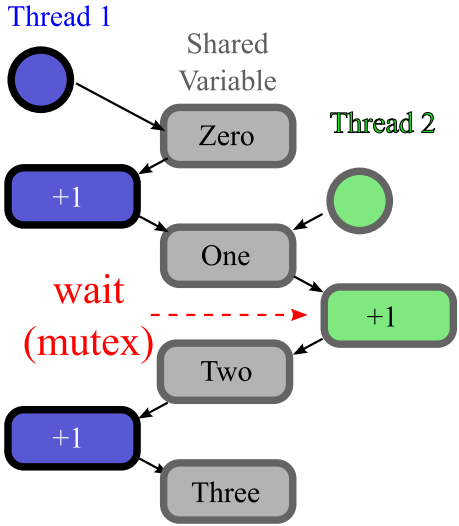


One software thread per logical processor or per core

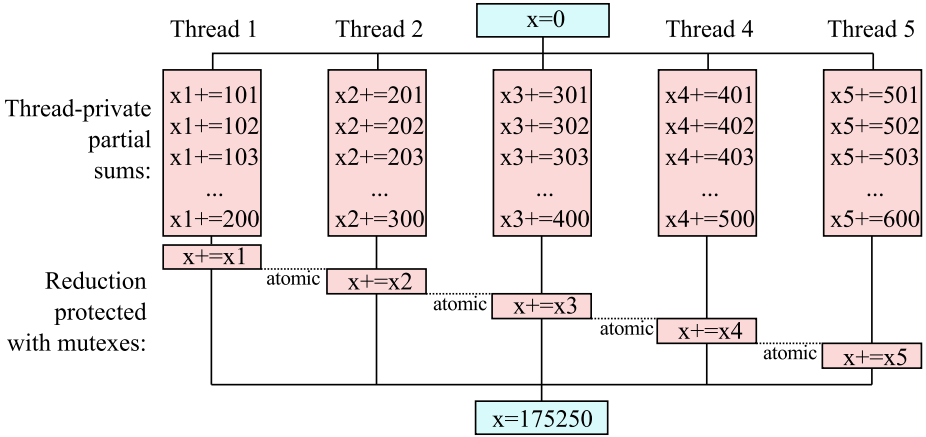
Data Races



Race Condition!



Parallel Reduction

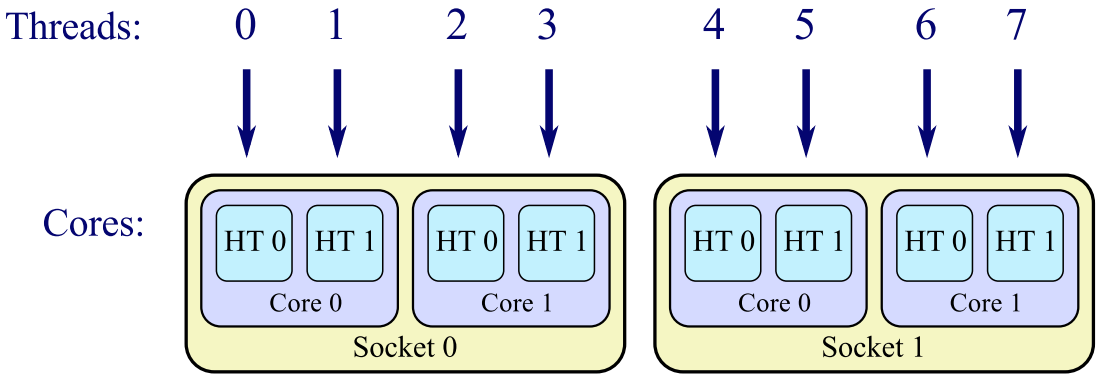


Key: using thread-private containers for partial sums; mutexes only after the loop

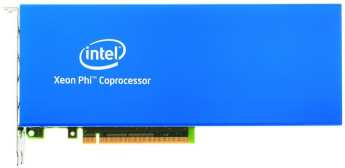
Multi-Threaded Version

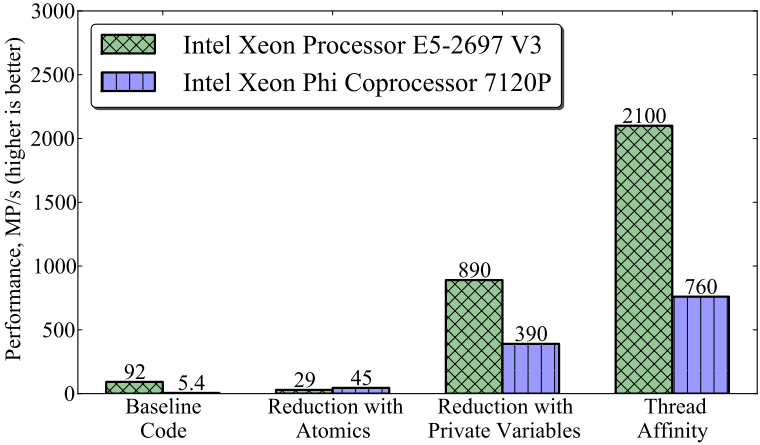
```
1  void BinParticles_2(const InputDataType& inputData, BinsType& outputBins){
2  #pragma omp parallel
3    { BinsType threadPrivateBins; // Declare thread-private containers
4      for (int i = 0; i < nBinsX; i++)
5        for (int j = 0; j < nBinsY; j++)
6          threadPrivateBins[i][j] = 0;
7  #pragma omp for
8      for (int i = 0; i < inputData.numDataPoints; i++) {
9        // ...transforming from cylindrical to Cartesian coordinates
10       // ...calculating the bin numbers for these coordinates
11       threadPrivateBins[iX][iY]++; // Incrementing thread-private counter:
12     }
13     for(int i = 0; i < nBinsX; i++) // Reduction outside the parallel loop
14       for(int j = 0; j < nBinsY; j++)
15 #pragma omp atomic
16         outputBins[i][j] += threadPrivateBins[i][j];
17 }  }
```

Thread Affinity



Performance with Thread Parallelism

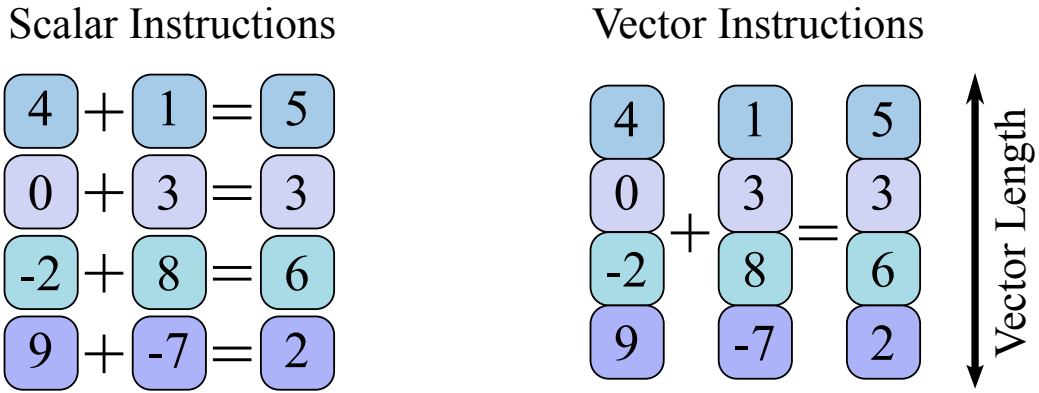




More work is needed to bring out coprocessor's performance!
Next step: vectorization.

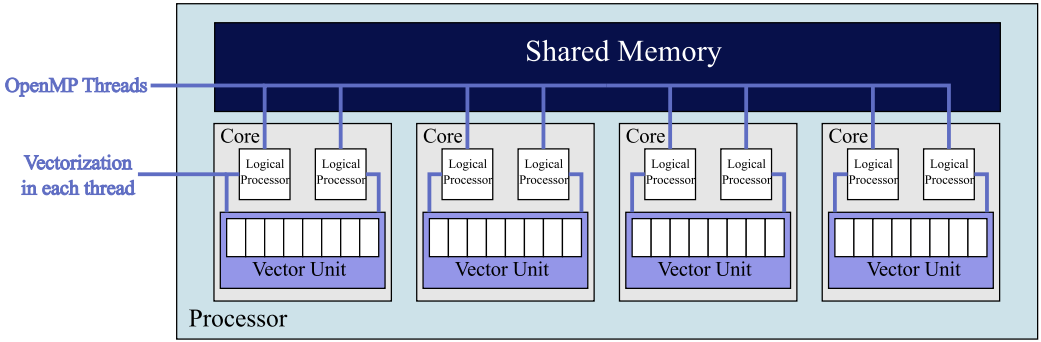
§5. SIMD Parallelism: Happy Vector Units

SIMD Instructions and Vectorization



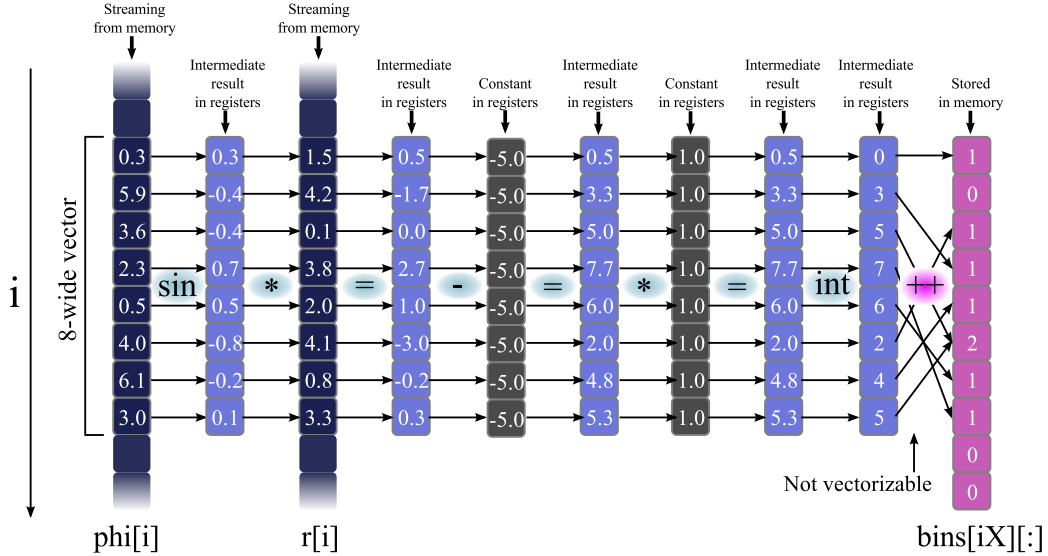
Intel archtecture implements SIMD processing with short vectors.

Vectors and Cores



Use threads to parallelize across cores, vector instructions for SIMD parallelism.

Vectorization Opportunity in Binning



Strip-Mining to the Rescue

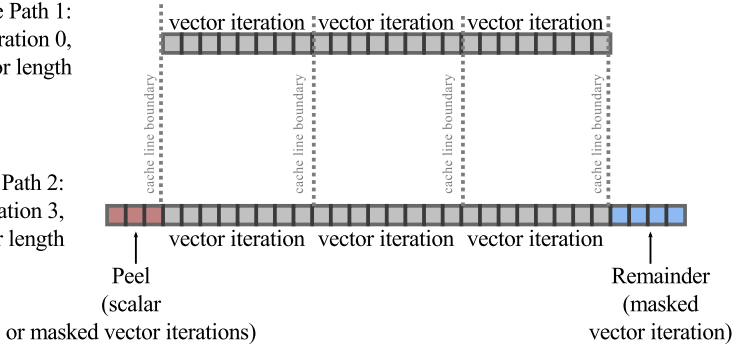
```
1  const int STRIP_WIDTH = 16;
2  for (int ii = 0; ii < inputData.numDataPoints; ii += STRIP_WIDTH) {
3    int iX[STRIP_WIDTH], iY[STRIP_WIDTH];
4    const FTYPE* r   = &(inputData.r[ii]);
5    const FTYPE* phi = &(inputData.phi[ii]);
6    for (int c = 0; c < STRIP_WIDTH; c++) { // Vector loop
7      const FTYPE x = r[c]*COS(phi[c]); // Transforming from cylindrical
8      const FTYPE y = r[c]*SIN(phi[c]); // to Cartesian coordinates
9      iX[c] = int((x - xMin)*binsPerUnitX); // Calculating the bin numbers
10     iY[c] = int((y - yMin)*binsPerUnitY); // for these coordinates
11   }
12   for (int c = 0; c < STRIP_WIDTH; c++) // Scalar loop
13     threadPrivateBins[iX[c]][iY[c]]++;
14 }
```

Fine-Tuning Vectorization

```
for (i = 0; i < n; i++)  A[i] = ...
```

Code Path 1:
data aligned from iteration 0,
n is multiple of vector length

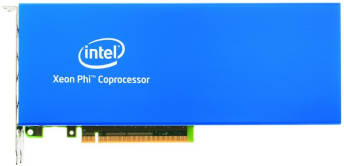
Code Path 2:
data aligned from iteration 3,
n is not a multiple of vector length

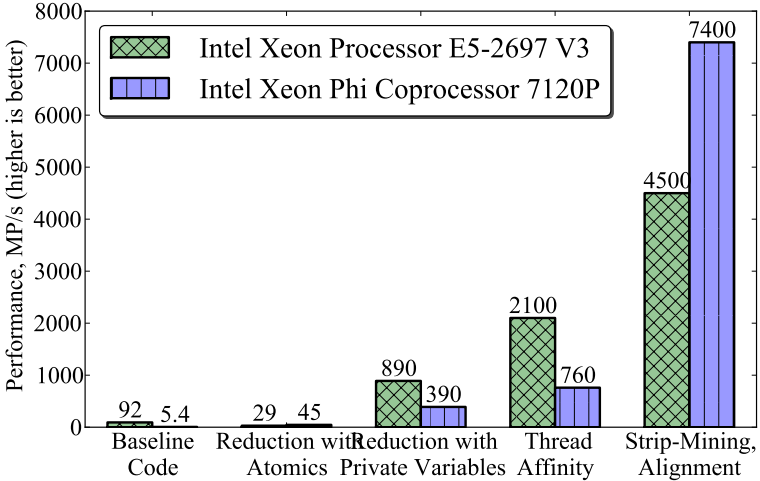


Better Vectorization

```
1  // Allocating data on a 64-byte aligned memory heap address
2  rawData.r   = (FTYPE*) _mm_malloc(sizeof(FTYPE)*n, 64);
3  rawData.phi = (FTYPE*) _mm_malloc(sizeof(FTYPE)*n, 64);
4
5  // Later in the code:
6  for (int ii = 0; ii < inputData.numDataPoints; ii += STRIP_WIDTH) {
7    int iX[STRIP_WIDTH] __attribute__((aligned(64))); // Aligned allocation
8    int iY[STRIP_WIDTH] __attribute__((aligned(64))); // on the stack
9    // ...
10
11   // Compiler hint: we promise alignment, no need for peeling
12 #pragma vector aligned
13   for (int c = 0; c < STRIP_WIDTH; c++) {
14     // ...
15   }
16 }
```


Performance with Vectorization



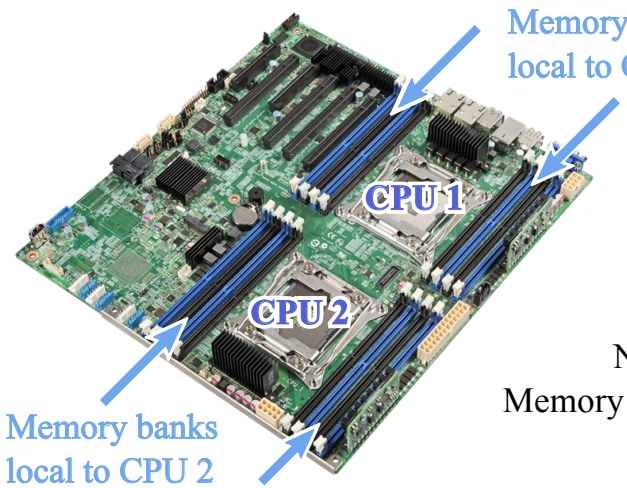


We can get more out of the CPU

Next step: memory traffic tuning.

§6. Memory Tuning: Happy Controllers

NUMA Architecture in Intel Xeon Processors

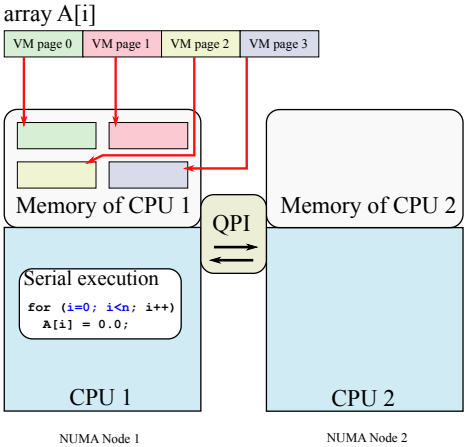


Non-Uniform
Memory Architecture
(NUMA)

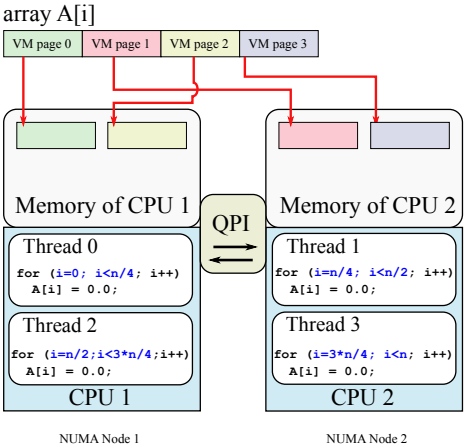
Intel Xeon Phi processors of 2nd generation (KNL) have NUMA support as well.

First-Touch Allocation Policy

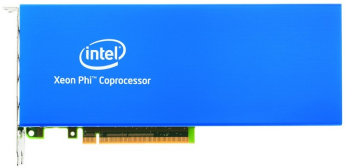
Poor First-Touch Allocation

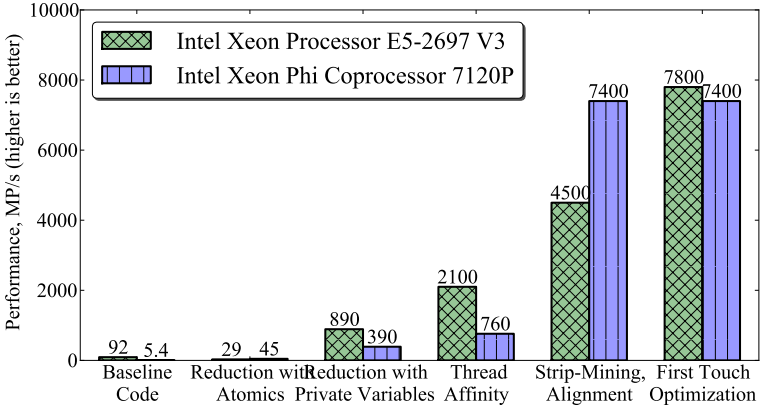


Good First-Touch Allocation



Performance with Parallel First Touch



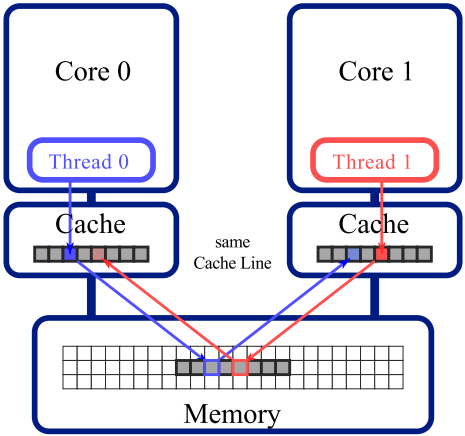


Code is ready to scale to future Intel architectures.

Is the Xeon Phi performance sufficient? See colfaxresearch.com/?p=11

§7. Extra Credit: False Sharing

False Sharing



False sharing occurs if one thread when multiple threads access the same cache line, and one of the accesses is a write operation.

Global Containers for Thread-Private Data

```
// Using a global container in
// threads-first layout
int nThrds=omp_get_max_threads();

// Instead of storing scalars in
// each bin, store an array with
// values for each thread =
int glBins[nBinsX][nBinsY][nThrds];

#pragma omp parallel
{
  int iThd = omp_get_thread_num();
  // ... later, memory access:
  glBins[iX[c]][iY[c]][iThrd]++;
}
```

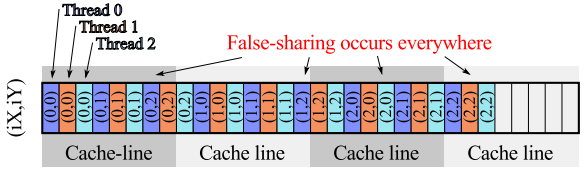
```
// Using a global container in
// threads-last layout
int nThrds=omp_get_max_threads();

// Instead of storing scalars in
// each bin, store an array with
// values for each thread
int glBins[nThrds][nBinsX][nBinsY];

#pragma omp parallel
{
  int iThd = omp_get_thread_num();
  // ... later, memory access:
  glBins[iThrd][iX[c]][iY[c]]++;
}
```

False Sharing and Data Layout

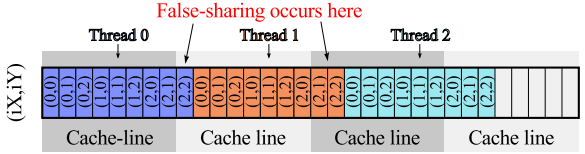
Case #1:
global container
threads-first layout



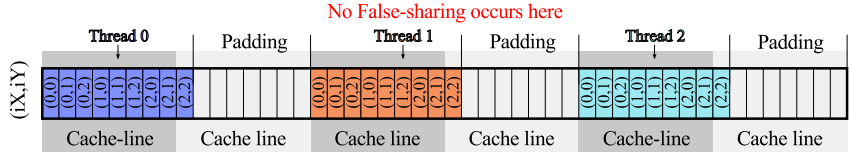
Bins

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)
(0,2)	(1,2)	(2,2)

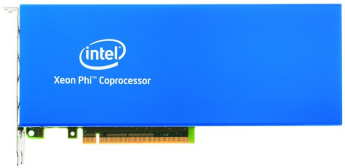
Case #2:
global container
threads-last layout

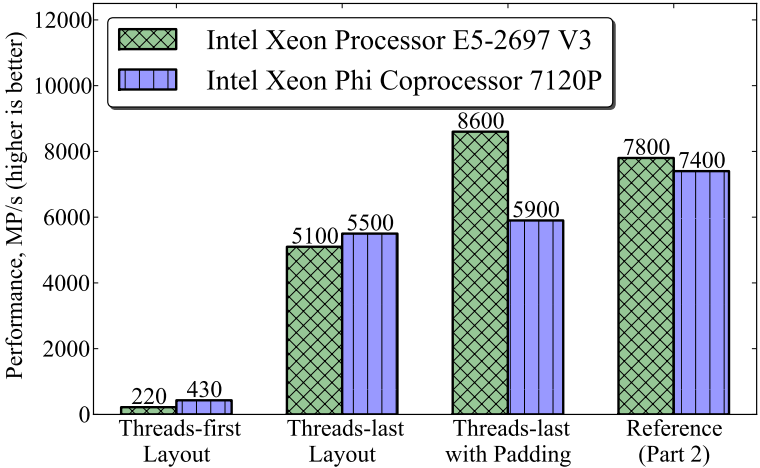


Case #3:
global container
threads-last layout
with padding



Performance with False Sharing and Padding





False sharing can be eliminated with padding.
Xeon needs more padding than Xeon Phi.

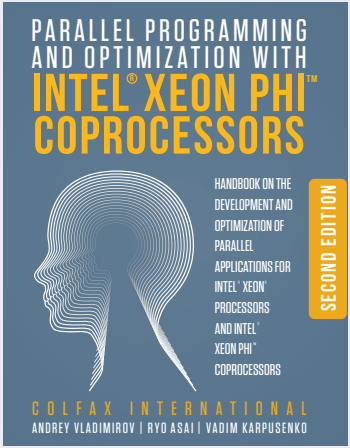
§8. Resources

Supplementary Materials: Textbook

ISBN: 978-0-9885234-0-1 (2nd edition, 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/

Supplementary Materials: Video Courses

colfaxresearch.com/video-courses

Slides, Code, Video

You can download slides, code and watch the video recording of this webinar here (requires registration for a free Colfax Research account):

colfaxresearch.com/hot-1512

Next webinar on December 15, 2015: “Finding the Low-Hanging Fruit for Optimization”:

Register