



DAY 6: OPTIMIZATION ON PARALLEL INTEL® ARCHITECTURES

Lecture day 6

Ryo Asai

Colfax International — colfaxresearch.com

April 2017

WELCOME

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.

COLFAX RESEARCH

COLFAX RESEARCH
CONTRIBUTING TO INNOVATIONS IN COMPUTING

READ WATCH LEARN CONNECT JOIN

Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation

To search, type and hit enter

Popular

The Hands-On Tutorials (HOT) webinars: details on efficient programming for Intel architecture

The Hands-On Workshop (HOW) Series

Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation

Research and Educational Publications

Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation

Optimization Techniques for the Intel MIC Architecture, Part 1 of 3: False Sharing and Padding

Software Developer's Introduction to the HOST Ultrastar Archive Hard SMR Drives

Optimization Techniques for the Intel MIC Architecture, Part 2 of 3: Strip-Mining for Vectorization

Optimization Techniques for the Intel MIC Architecture, Part 3 of 3: Multi-Threading and Parallel Reduction

Performance to Power and Performance to Cost Ratios with Intel Xeon Phi Coprocessors (and why Acceleration May Be Enough)

Events

Presentations

Log In/Out or Register

Consulting

Share

Colfax offers consulting services for enterprises, research help you to:

- Optimize your existing application to take advantage of parallelism, from vectors to cores to clusters and I
- Future-proof your application for upcoming Intel® processors
- Accelerate your application using coprocessor tec
- Investigate the potential system configurations that satisfy your cost, power performance requirements.

View Details

Introduction to Intel MIC, chapter 1: Overview

Episode 2.3 — Purpose of the MIC architecture

Parallel Computing in the Search for New Physics at LHC

Fluid Dynamics with Fortran on Intel Xeon Phi coprocessors

In this demonstration, a fluid dynamics program simulates flow of air over two flat rectangular blocks. The simulation shows that the air flow over each block is much more intense than the air flow between the blocks. This is due to the fact that the air flow around each block is much faster than the air flow between the blocks. The simulation results are visualized in a 3D visualization tool.

View Article

Configuration and Benchmarks of Peer-to-Peer Communication over Gigabit Ethernet and InfiniBand in a Cluster with Intel Xeon Phi Coprocessors

View Details

Interview with James Reinders: future of Intel MIC architecture, parallel programming, education

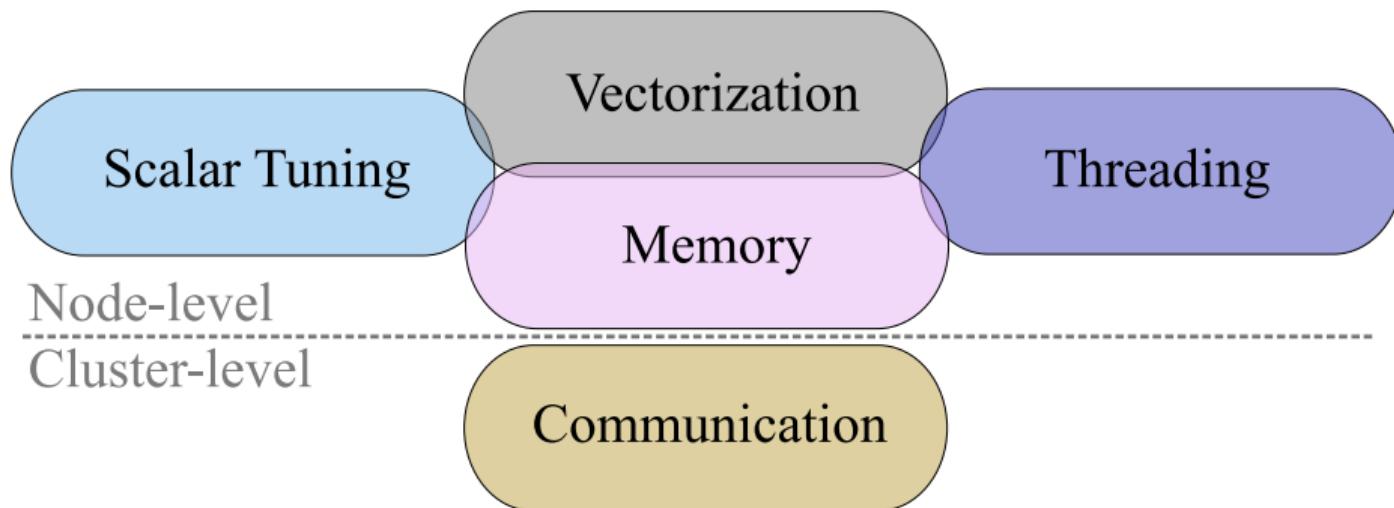
A few weeks ago we had another great conversation with James Reinders, the Director and Chief Architect of Intel's Parallel Computing Division. In this video he discusses the future of Intel MIC, and how it can produce fast Intel MIC processors, bring Intel MIC and Intel Xeon Phi closer together, and how Intel can support parallel programming and optimization for high performance applications.

Watch Video

<http://colfaxresearch.com/>

Code Modernization

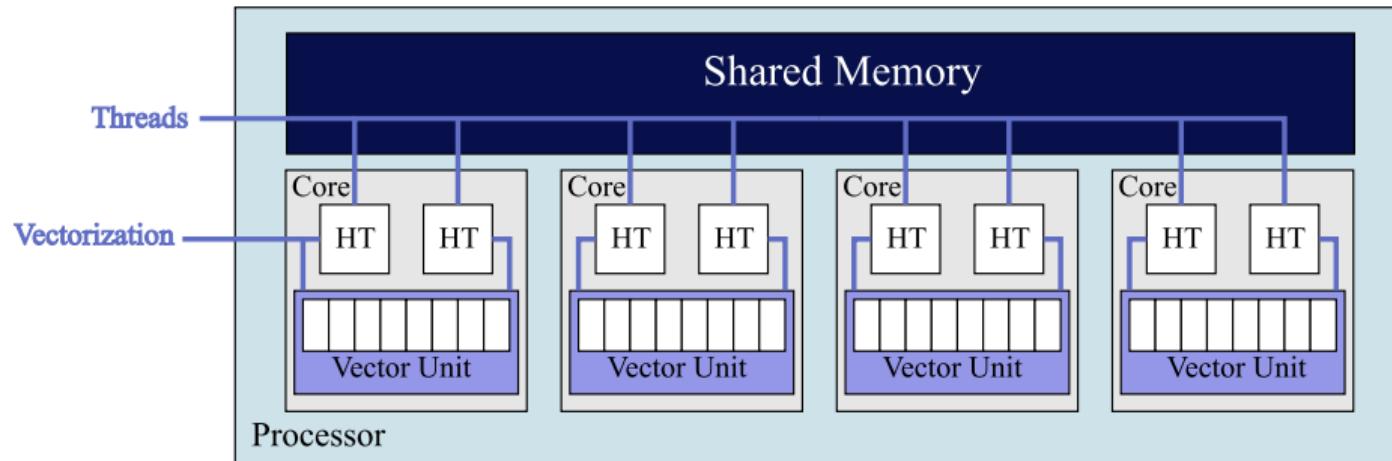
Optimizing software to better utilize features available in modern computer architectures.



§1. QUICK REVIEW

Task Parallelism – multiple instructions multiple data elements (MIMD)

Data Parallelism – single instruction multiple data elements (SIMD)



Unbounded growth opportunity, but **not automatic**

SHORT VECTOR SUPPORT

7

Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

Scalar Instructions

$$\begin{array}{ccc} 4 & + & 1 \\ \hline 5 \end{array}$$
$$\begin{array}{ccc} 0 & + & 3 \\ \hline 3 \end{array}$$
$$\begin{array}{ccc} -2 & + & 8 \\ \hline 6 \end{array}$$
$$\begin{array}{ccc} 9 & + & -7 \\ \hline 2 \end{array}$$

Vector Instructions

$$\begin{array}{c} 4 \\ 0 \\ -2 \\ 9 \end{array} + \begin{array}{c} 1 \\ 3 \\ 8 \\ -7 \end{array} = \begin{array}{c} 5 \\ 3 \\ 6 \\ 2 \end{array}$$

↑
Vector Length
↓

NOW WHAT?

8

I have a vectorized and multi-threaded code!

Some people stop here. But even if your application is multi-threaded and vectorized, it may not be optimal. Optimization could unlock more performance for your application.

Example areas for consideration:

▷ Multi-threading

- Do my threads have enough work?
- Are my threads independent?
- Is work distributed properly?

▷ Vectorization

- Is my data organized well for vectorization?
- Do I have regular loop patterns?

§2. OPTIMIZATION EXAMPLES

EXAMPLE: NUMERICAL INTEGRATION

$$I(a, b) = \int_0^a \frac{1}{\sqrt{x}} dx$$

Rectangle method:

$$\Delta x = \frac{a}{n},$$

$$x_i = (i+1)\Delta x,$$

$$I(a, b) = \sum_{i=0}^{n-1} \frac{1}{\sqrt{x_i}} \Delta x + O(\Delta x).$$

```

1 float Integrate(const float a,
                  const int N) {
2     const float dx = a/float(n);
3     float S = 0.0f;
4     for (int i = 0; i < n; i++) {
5         const float xi = dx*float(i+1);
6         S += 1.0f/sqrts(xi) * dx;
7     }
8     return S;
9 }
10 }
```

HOW NOT TO PARALLELIZE

Incorrect: Race condition on integral

```
1 const double dx = (x_upper_bound - x_lower_bound)/nSteps;
2 double integral = 0.0;
3 #pragma omp parallel for
4     for(int i = 0; i < nSteps; i++) {
5         const double x = x_lower_bound + dx*(double(i) + 0.5);
6         integral += 1.0/sqrt(x) * dx;
7     }
```

Non-optimal: Too much synchronization. Slower than serial implementation.

```
1 #pragma omp parallel for
2     for(int i = 0; i < nSteps; i++) {
3         const double x = x_lower_bound + dx*(double(i) + 0.5);
4 #pragma omp atomic
5         integral += 1.0/sqrt(x) * dx;
6     }
```

REDUCTION

Manual reduction implementation

```
1  double integral = 0.0;
2  #pragma omp parallel
3  {
4      double integral_th=0.0;
5  #pragma omp for
6      for(int i = 0; i < nSteps; i++) {
7          const double x = x_lower_bound + dx*(double(i) + 0.5);
8          integral_th += 1.0/sqrt(x) * dx;
9      }
10 #pragma omp atomic
11     integral += integral_th;
12 }
```

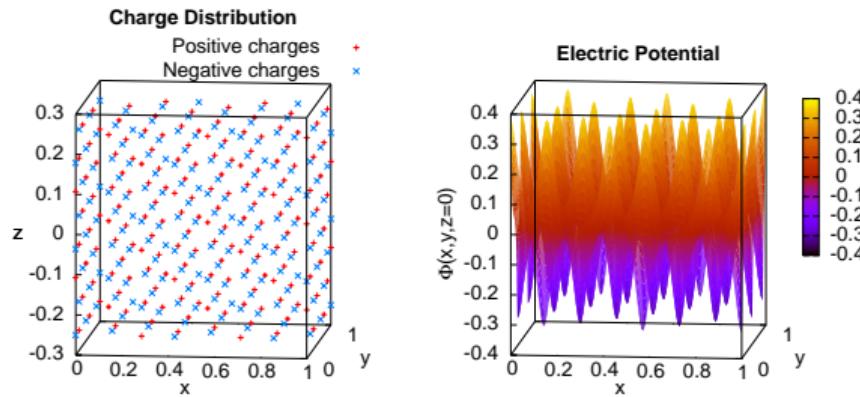
OpenMP reduction. Array support from 4.5.

```
1 #pragma omp parallel for reduction(+:integral)
```

EXAMPLE: COULOMB'S LAW APPLICATION

$$\Phi(\vec{R}_j) = - \sum_{i=1}^m \frac{q_i}{|\vec{r}_i - \vec{R}_j|}, \quad (1)$$

$$|\vec{r}_i - \vec{R}| = \sqrt{(r_{i,x} - R_x)^2 + (r_{i,y} - R_y)^2 + (r_{i,z} - R_z)^2}. \quad (2)$$



INITIAL IMPLEMENTATION

```
1 struct Charge {  
2     float x, y, z, q;  
3 };
```

```
1 for (int i = 0; i < n; i++)  
2     for (int j = 0; j < n; j++)  
3         for (int k = 0; k < n; k++) {  
4             const float Rx = (float) i, Ry = (float) j, Rz = (float) k;  
5             float phi_p = 0.0;  
6             for (int l=0; l<m; l++) {  
7                 const float dx=chg[l].x - Rx;  
8                 const float dy=chg[l].y - Ry;  
9                 const float dz=chg[l].z - Rz;  
10                phi_p -= chg[l].q / (dx*dx+dy*dy+dz*dz); // Coulomb's law  
11            }  
12            phi[i*n*n+j*n+k] = phi_p;  
13        }
```

ADDING THREAD PARALLELISM

Bad: parallelizing the inner loop increases overhead of OpenMP

```

1  for (int i = 0; i < n; i++)
2    for (int j = 0; j < n; j++)
3      #pragma omp parallel for
4        for (int k = 0; k < n; k++) {
```

Better: if n is small, not enough work to occupy all threads.

```

1  #pragma omp parallel for
2  for (int i = 0; i < n; i++)
3    for (int j = 0; j < n; j++)
4      for (int k = 0; k < n; k++) {
```

Good: collapse loops means threads have more iterations to work with.

```

1  #pragma omp parallel for collapse(2)
2  for (int i = 0; i < n; i++)
3    for (int j = 0; j < n; j++)
4      for (int k = 0; k < n; k++) {
```

LOOP COLLAPSE: PRINCIPLE

Idea: combine iterations spaces of the inner loop and the outer loop.

```
1 #pragma omp parallel for collapse(2)
2   for (int i = 0; i < m; i++)
3     for (int j = 0; j < n; j++) {
4       // ...
5       // ...
6     }
```

```
1 #pragma omp parallel for
2   for (int c = 0; c < m*n; c++) {
3     i = c / n;
4     j = c % n;
5     // ...
6   }
```

LOOP SCHEDULING MODES IN OPENMP

Scheduling	Threads	Iterations
static	0	0 1 2 3 4 5 6 7
	1	8 9 10 11 12 13 14 15
	2	16 17 18 19 20 21 22 23
	3	24 25 26 27 28 29 30 31

dynamic	0	0 7 10 12 17 ...
	1	1 4 9 14 18 ...
	2	2 5 8 11 15 ...
	3	3 6 9 13 16 ...

guided	0	0 1 2 3 16 17 24 29
	1	4 5 6 7 20 21 25 30
	2	8 9 10 11 18 19 26 28
	3	12 13 14 15 22 23 27 31 32

Time



Scheduling	Threads	Iterations
static,1	0	0 4 8 12 16 20 24 28
	1	1 5 9 13 17 21 25 29
	2	2 6 10 14 18 22 26 30
	3	3 7 11 15 19 23 27 31

dynamic,2	0	0 1 10 11 16 17 ...
	1	2 3 12 13 18 19 ...
	2	4 5 8 9 14 15 ...
	3	6 7 20 21 ...

guided,2	0	0 1 2 3 16 17 24 25
	1	4 5 6 7 20 21 26 27
	2	8 9 10 11 18 19 28 29
	3	12 13 14 15 22 23 30 31

Time



CONTROL OF SCHEDULING MODES

To set scheduling for a particular loop in code (example):

```
1 #pragma omp parallel for schedule(dynamic,4)
2 // ...
```

To set scheduling for the entire application at run time (example):

```
1 #pragma omp parallel for schedule(runtime)
2 // ...
```

```
vega@lyra% export OMP_SCHEDULE=dynamic,4
vega@lyra% ./run-my-app
```

ARRAYS OF STRUCTURES VERSUS STRUCTURES OF ARRAYS

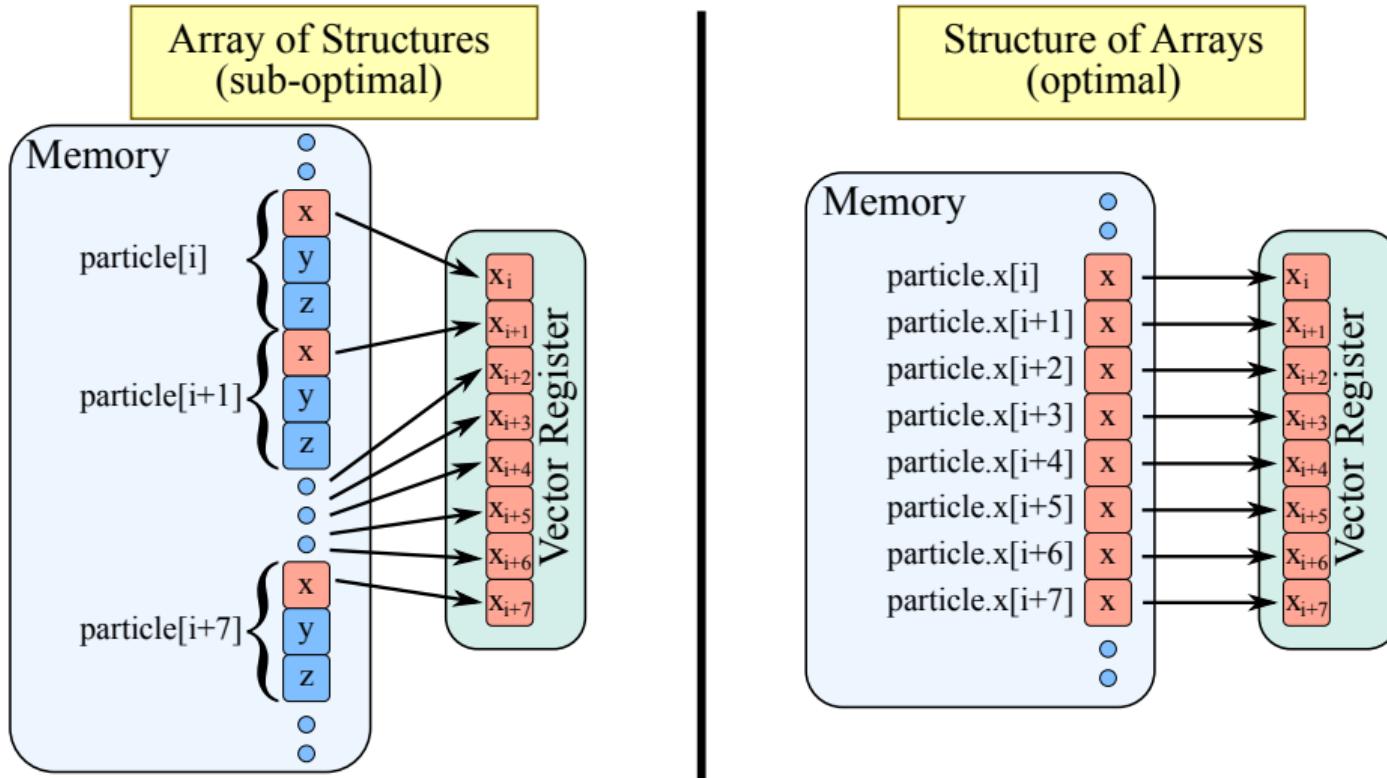
Array of Structures (AoS)

```
1 struct Charge { // Elegant, but ineffective data layout
2     float x, y, z, q; // Coordinates and value of this charge
3 };
4 // The following line declares a set of m point charges:
5 Charge chg[m];
```

Structure of Arrays (SoA)

```
1 struct Charge_Distribution {
2     // Data layout permits effective vectorization of Coulomb's law application
3     const int m; // Number of charges
4     float * x; // Array of x-coordinates of charges
5     float * y; // ...y-coordinates...
6     float * z; // ...etc.
7     float * q; // These arrays are allocated in the constructor
8 };
```

ARRAYS OF STRUCTURES VERSUS STRUCTURES OF ARRAYS



OPTIMIZED SOLUTION: STRUCTURE OF ARRAYS, UNIT-STRIDE ACCESS

```
1 struct Charge_Distribution {  
2     // Data layout permits effective vectorization of Coulomb's law application  
3     const int m; // Number of charges  
4     float *x, *y, *z, *q; // Arrays of x-, y- and z-coordinates of charges  
5 };
```

```
1 // This version vectorizes better thanks to unit-stride data access  
2 for (int i=0; i<chg.m; i++) {  
3     // Unit stride: ( $\&chg.x[i+1]$  -  $\&chg.x[i]$ ) == sizeof(float)  
4     const float dx=chg.x[i] - Rx;  
5     const float dy=chg.y[i] - Ry;  
6     const float dz=chg.z[i] - Rz;  
7     phi -= chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);  
8 }
```

§3. ADDITIONAL TOPIC: UNROLL AND JAM

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

```

UNROLL AND JAM

```

1  for (int k = 0; k < n; k+=4) {
2      const float Rx = (float) i, Ry = (float) j, Rz1 = (float) k,
3                  Rz2 = (float) k+1, Rz3 = (float) k+2, Rz4 = (float) k+3;
4      float phi_p1 = 0.0, phi_p2 = 0.0, phi_p3 = 0.0, phi_p4 = 0.0;
5 #pragma omp simd reduction(+: phi_p1, phi_p2, phi_p3, phi_p4)
6      for (int l=0; l<m; l++) {
7          const float dx = chg.x[l] - Rx, dy=chg.y[l] - Ry;
8                      dz1=chg.z[l] - Rz1, dz2=chg.z[l] - Rz2,
9                      dz3=chg.z[l] - Rz3, dz4=chg.z[l] - Rz4;
10         phi_p1 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz1*dz1);
11         phi_p2 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz2*dz2);
12         phi_p3 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz3*dz3);
13         phi_p4 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz4*dz4);
14     }
15     phi[i*n*n+j*n+k] = phi_p1;   phi[i*n*n+j*n+k+1] = phi_p2;
16     phi[i*n*n+j*n+k+2] = phi_p3;   phi[i*n*n+j*n+k+3] = phi_p4;
17 } // Remainders to follow (dealing with case k%4 != 0) ...

```

§4. MEMORY CONSIDERATIONS

FLOPS VS BANDWIDTH

HOW CHEAP ARE FLOPS?

Theoretical estimates, Intel Xeon E5-2697 V3 processor

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

Required Data Rate = 1.2 TFLOP/s \times 8 bytes \approx 10 TB/s

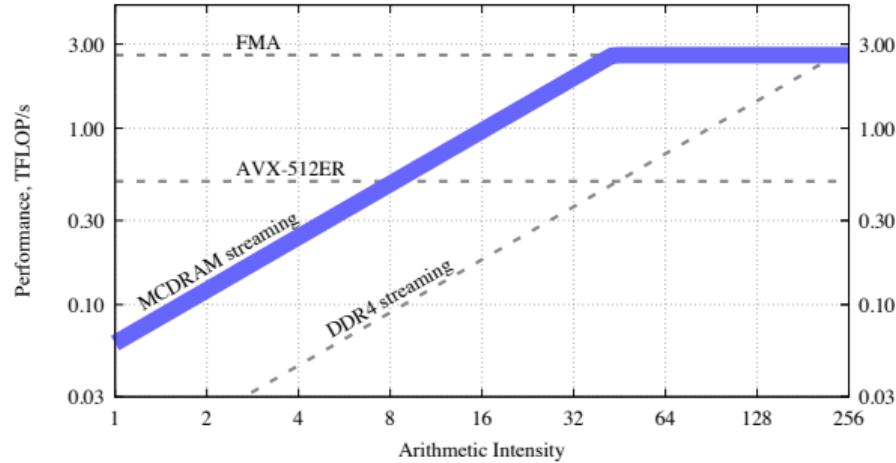
Memory Bandwidth = $\eta \times 2 \times 59.7 \approx 0.1$ TB/s

Ratio = 10/0.1 \approx 100 (FLOPs)/(Memory Access)

If the Arithmetic Intensity is...

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

ARITHMETIC INTENSITY AND ROOFLINE MODEL



More on roofline model: [Williams et al.](#)

ON COMPUTATIONAL COMPLEXITY OF ALGORITHMS

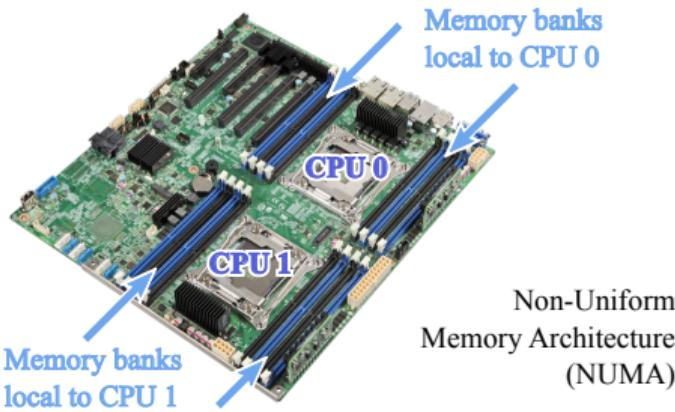
Type	Properties	Examples
$O(N)$	Each data element is used a fixed number of times. Memory-bound unless the number of times is large.	Array scaling, image brightness adjustment, vector dot-product.
$O(N^\alpha)$	Each element is used $N^{\alpha-1}$ times. A lot of data reuse for $\alpha > 1$. Good implementation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

NUMA ARCHITECTURE

NUMA ARCHITECTURES

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

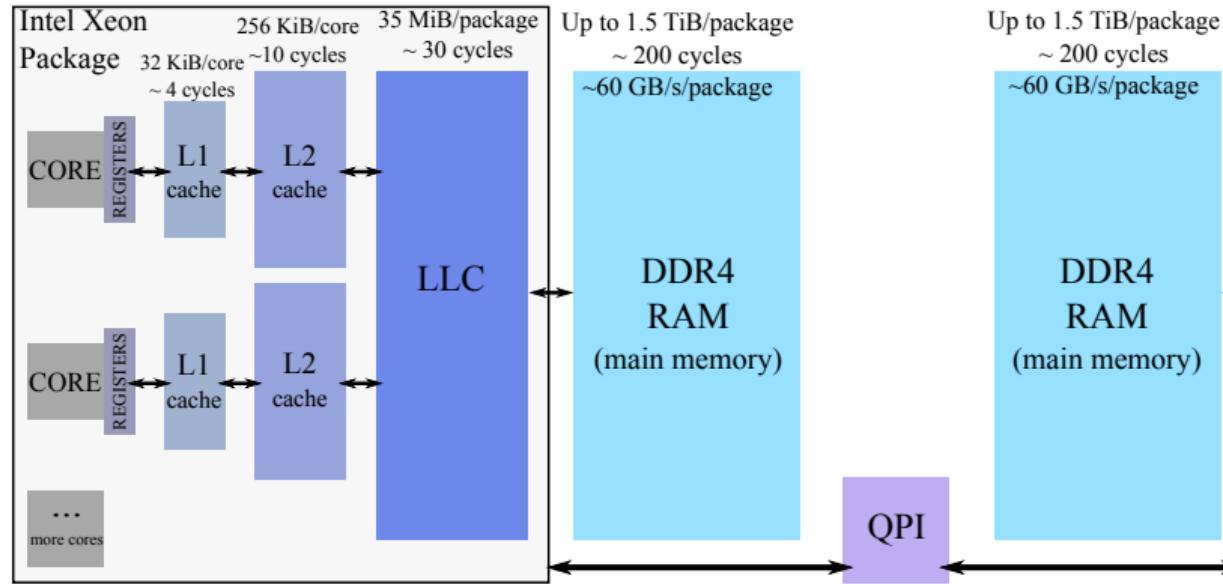


Examples:

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

INTEL XEON CPU: MEMORY ORGANIZATION

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



WORKING WITH NUMA ARCHITECTURES

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

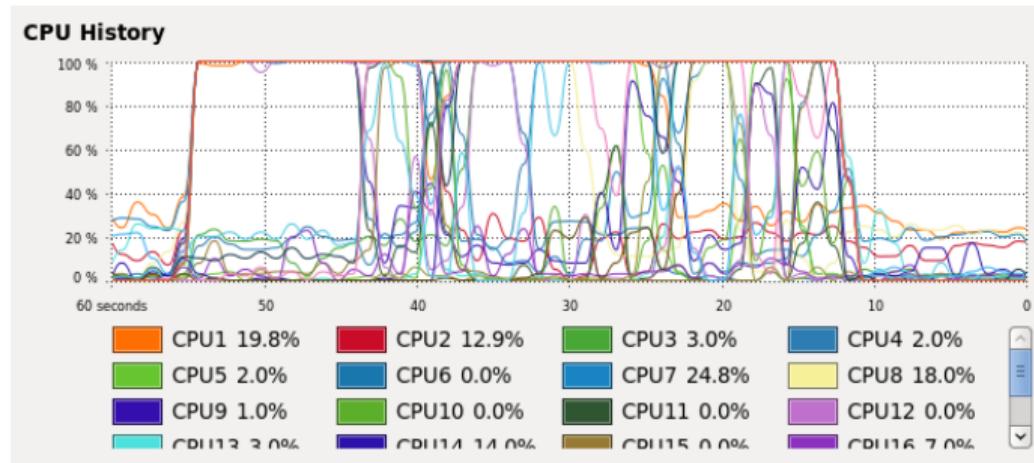
```
vega@lyra% numactl --hardware
```

```
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
```

WHAT IS THREAD AFFINITY

- ▶ OpenMP threads may migrate between cores
- ▶ Forbid migration — improve locality — increase performance
- ▶ Affinity patterns “scatter” and “compact” may improve cache sharing, relieve thread contention



OMP_PROC_BIND AND OMP_PLACES VARIABLES

Control the binding pattern, including nested parallelism:

```
OMP_PROC_BIND=type[,type[,...]]
```

Here type=true, false, spread, close or master.

Comma separates settings for different levels of nesting (OMP_NESTED must be enabled).

Control the granularity of binding:

```
OMP_PLACES=<threads|cores|sockets|(explicit)>
```

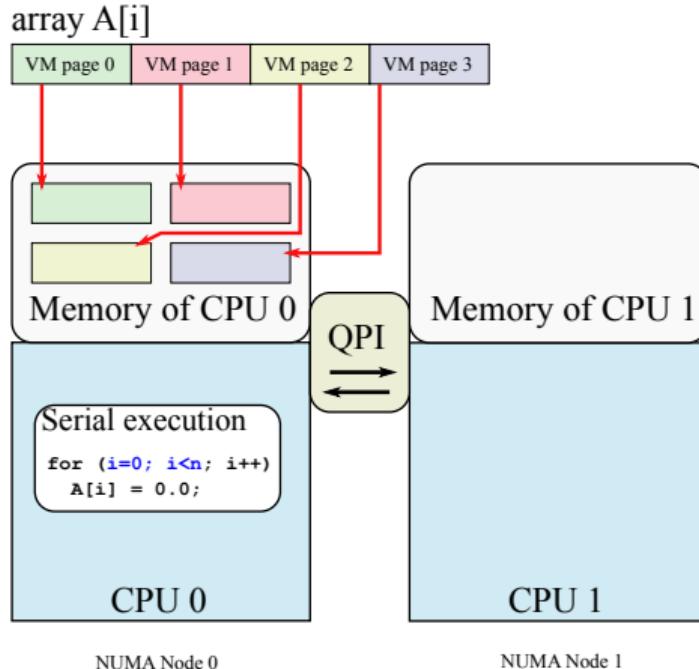
ALLOCATION ON FIRST TOUCH

- ▶ Memory allocation occurs not during `_mm_malloc()`, but upon the first write to the buffer (“first touch”)
- ▶ Default NUMA allocation policy is “on first touch”
- ▶ For better performance in NUMA systems, initialize data with the same parallel pattern as during data usage

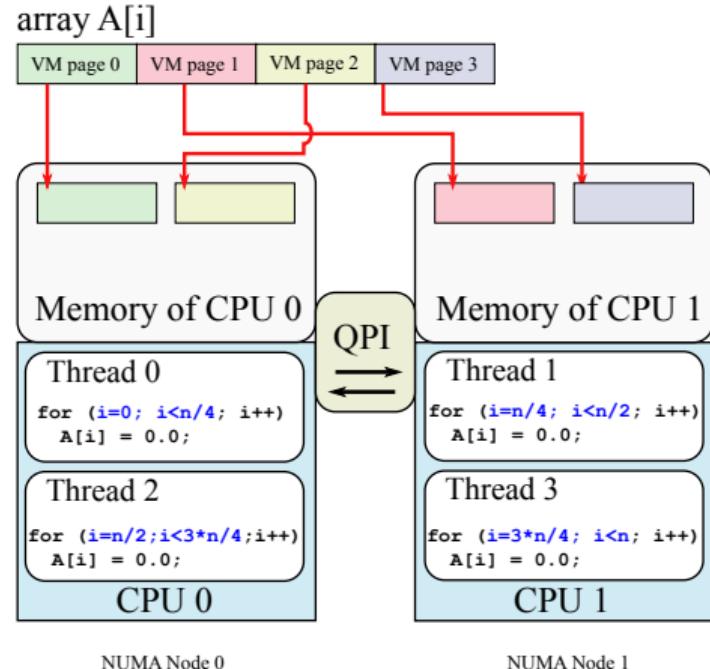
```
1 float* A = (float*)_mm_malloc(n*m*sizeof(float), 64);  
2  
3 // Initializing from parallel region for better performance  
4 #pragma omp parallel for  
5 for (int i = 0; i < n; i++)  
6     for (int j = 0; j < m; j++)  
7         A[i*m + j] = 0.0f;
```

FIRST-TOUCH ALLOCATION POLICY

Poor First-Touch Allocation



Good First-Touch Allocation



IMPACT OF FIRST-TOUCH ALLOCATION

