



# PROGRAMMING AND OPTIMIZATION FOR INTEL<sup>®</sup> ARCHITECTURE

One-Day Workshop

*Andrey Vladimirov and Ryo Asai*  
*Colfax International — [colfaxresearch.com](http://colfaxresearch.com)*

October 2016

WELCOME

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.

## WHERE TO GET THESE SLIDES

[xeonphi.com/training/slides](http://xeonphi.com/training/slides)



# **§1. SNEAK PEAK**

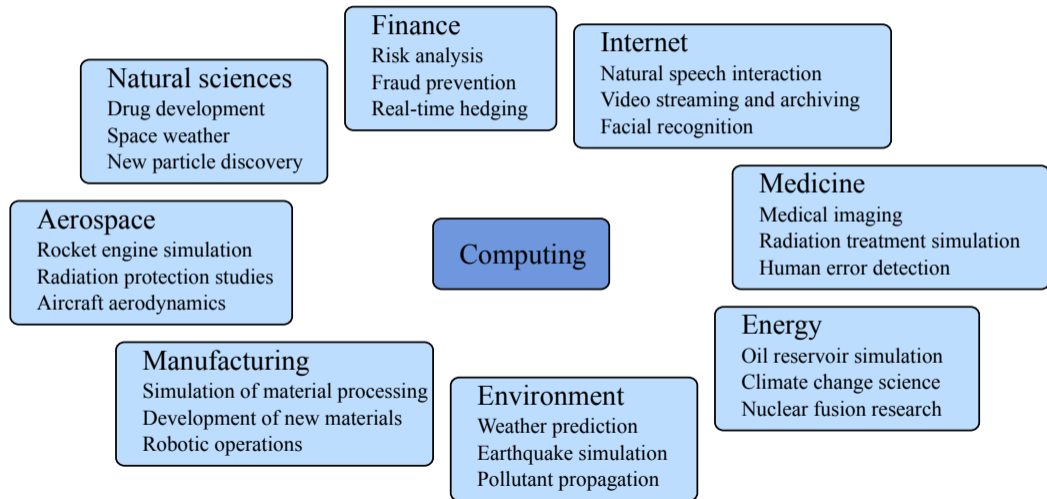


## **AGENDA AND WHAT'S IN IT FOR YOU**

# LIST OF TOPICS

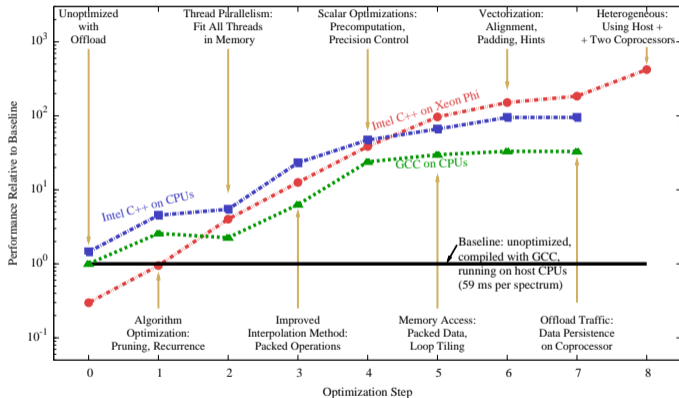
1. Programming, optimization by example
  - N-body simulation
  - Know architecture to work on performance
2. Optimization pointers
  - Scalar tuning - compiler usage, programming practices
  - Vectorization - making it happen and tuning containers, patterns
  - Multi-threading - OpenMP, common issues, tuning.
  - Memory access - avoiding it, streamlining it
  - Communication control - MPI strategies
3. Preparing for Knights Landing
  - AVX-512, high-bandwidth memory, clustering modes
  - Coprocessors and KNL-F; Intel Omni-Path Architecture
4. Intel Libraries
  - MKL, Python, DAAL, Caffe, etc.

## Just some examples

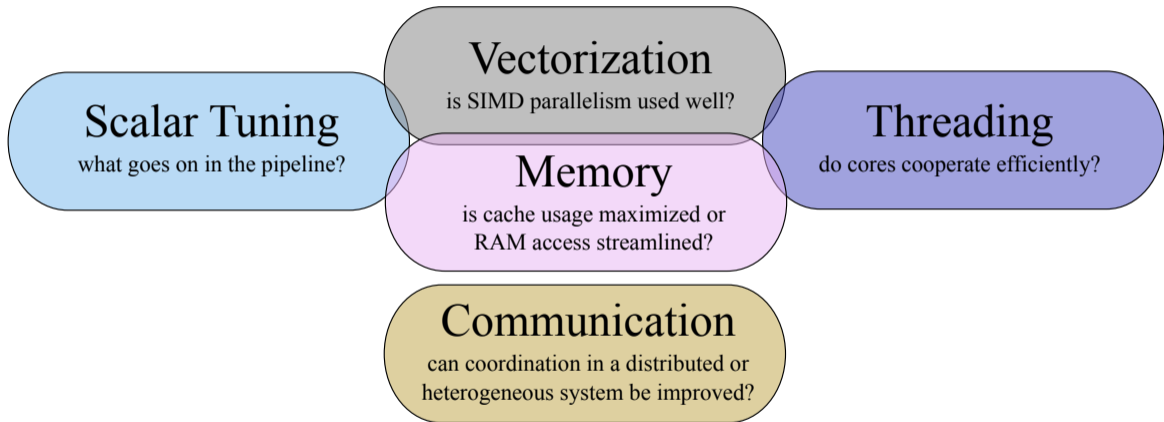


# PROGRAMMING MODEL CONTINUITY

Common story for many applications:



(see <http://xeonphi.com/papers/heatcode>)





**WHERE TO LEARN MORE**



THE "HOW" SERIES TRAINING

# DEEP DIVE

WITH CODE MODERNIZATION EXPERTS

It's free

→ [HOWSERIES.COM](https://howseries.com)

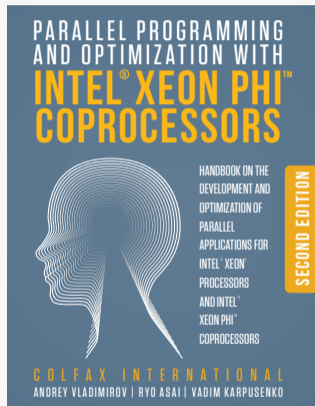
\*10x 2-hour sessions | 24-hour 2-weeks remote access to a system

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

## Parallel Programming and Optimization with Intel® Xeon Phi™ Coprorocessors

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

© Colfax International, 2015



<http://xeonphi.com/book>

## COLFAX RESEARCH

CONTRIBUTING TO INNOVATIONS IN COMPUTING

[Log In/Out](#) or [Register](#)

---

READ WATCH LEARN CONNECT JOIN



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**

---

Parallel Programming Book

**Introduction to parallel programming, deep discussion of optimization techniques, exercises.**  
 © 2015, Colfax International, 508 pages.

---

Featured Video

See Research material on vectorization in a training video



https://www.youtube.com/watch?v=7p-708

**Research and Educational Publications**



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



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



**Software Developer's Introduction to the HGST Ultrastar Archive H800 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 1 of 3: Multi-Threading and Parallel Reduction**



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

Consulting

[f](#) [t](#) [+](#) [s](#) [Share](#)



Colfax offers consulting services for enterprises, research help you:

- Optimize your existing application to take advantage of parallelism, from vectors to cores to clusters and
- Future-proof your application for upcoming innovations
- Accelerate your application using coprocessor tech
- Investigate the potential system configurations that satisfy your cost, power, performance requirements.
- Take a clean slate to develop a novel approach to reduce your computing pro

48 Videos Courses | 101 VHS | Chapter 1 | Episode 1.1

**Episode 2.1 — Purpose of the MIC architecture**



1.1 Purpose of the design  
 1.2 Introduction to the design  
 1.3 Summary of the design

Software Developer's Introduction to the HGST Ultrastar Archive H800 SMR Drives



In this paper we will discuss the new HGST Ultrastar Archive H800 SMR drives, their features, and how they compare to other high capacity storage solutions. These drives are well suited for large scale archival applications in a wide range of applications. The paper is intended for system architects.

Fluid Dynamics with Fortran on Intel Xeon Phi coprocessors



In this demonstration, a Fortran program is executed on a cluster of Intel Xeon Phi coprocessors. The program simulates the flow of a fluid around a cylinder. The results are visualized using a 3D visualization tool. The program is written in Fortran and is optimized for the Intel Xeon Phi coprocessors.

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



In this paper we will discuss the configuration and benchmarks of peer-to-peer communication over Gigabit Ethernet and InfiniBand in a cluster with Intel Xeon Phi coprocessors. The results are visualized using a 3D visualization tool. The program is written in Fortran and is optimized for the Intel Xeon Phi coprocessors.

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



In this video we will discuss the future of Intel MIC architecture, parallel programming, and education. The interview is with James Reinders, a senior software architect at Intel. The interview is conducted in a relaxed and conversational manner.

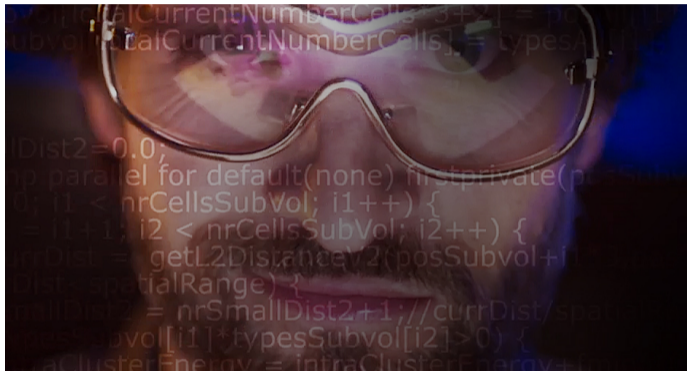
http://colfaxresearch.com/

colfaxresearch.com/events

WHERE TO LEARN MORE

© Colfax International, 2013–2016

# INTEL RESOURCES



[software.intel.com/modern-code](https://software.intel.com/modern-code)



[intel.com/xeonphi](https://intel.com/xeonphi)

## **§2. PROGRAMMING, OPTIMIZATION BY EXAMPLE**



# **DIRECT N-BODY SIMULATION**

# N-BODY SIMULATION ON CPU AND COPROCESSOR

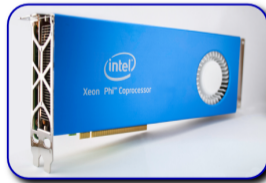


## N-body simulation on...

Two  
Intel® Xeon®  
CPUs



One  
Intel® Xeon Phi™  
coprocessor



Two  
Intel® Xeon Phi™  
coprocessors



Paper: <http://xeonphi.com/papers/nbody-basic>

Demo: [click here](#)

## Gravitational N-body dynamics:

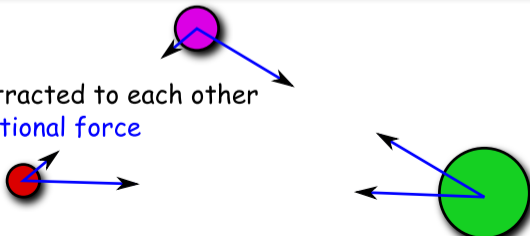
Newton's law of universal gravitation:

$$M_i \vec{R}_i''(t) = G \sum_j \frac{M_i M_j}{|\vec{R}_i - \vec{R}_j|^3} (\vec{R}_j - \vec{R}_i)$$

where:

$$|\vec{R}_i - \vec{R}_j| = \sqrt{(R_{i,x} - R_{j,x})^2 + (R_{i,y} - R_{j,y})^2 + (R_{i,z} - R_{j,z})^2}$$

particles are attracted to each other  
with the gravitational force



# APPLICATION

## 1. Astrophysics:

- planetary systems
- galaxies
- cosmological structures

## 2. Electrostatic systems:

- molecules
- crystals

This work: “toy model” with all-to-all  $O(n^2)$  algorithm. Practical N-body simulations may use tree algorithms with  $O(n \log n)$  complexity.



Source: [APOD](#), credit: Debra Meloy Elmegreen (Vassar College) et al., & the Hubble Heritage Team (AURA/ STScI/ NASA)

# ALL-TO-ALL APPROACH ( $O(n^2)$ COMPLEXITY SCALING)

Each particle is stored as a structure:

```
1 struct ParticleType {  
2     float x, y, z;  
3     float vx, vy, vz;  
4 };
```

main() allocates an array of ParticleType:

```
1 ParticleType* particle = new ParticleType[nParticles];
```

Particle propagation step is timed:

```
1 const double tStart = omp_get_wtime(); // Start timing  
2 MoveParticles(nParticles, particle, dt);  
3 const double tEnd = omp_get_wtime(); // End timing
```

# PARTICLE UPDATE ENGINE

```
1 void MoveParticles(int nParticles, ParticleType* particle, float dt) {
2     for (int i = 0; i < nParticles; i++) { // Particles that experience force
3         float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
4         for (int j = 0; j < nParticles; j++) { // Particles that exert force
5             // Newton's law of universal gravity
6             const float dx = particle[j].x - particle[i].x;
7             const float dy = particle[j].y - particle[i].y;
8             const float dz = particle[j].z - particle[i].z;
9             const float drSquared = dx*dx + dy*dy + dz*dz + 1e-20;
10            const float drPower32 = pow(drSquared, 3.0/2.0);
11            // Calculate the net force
12            Fx += dx/drPower32; Fy += dy/drPower32; Fz += dz/drPower32;
13        }
14        // Accelerate particles in response to the gravitational force
15        particle[i].vx+=dt*Fx; particle[i].vy+=dt*Fy; particle[i].vz+=dt*Fz;
16    }
```

# INCORPORATING THREAD PARALLELISM

Before:

```
1  for (int i = 0; i < nParticles; i++) { // Particles that experience force
2  float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
3  for (int j = 0; j < nParticles; j++) { // Particles that exert force
4  // Newton's law of universal gravity
5  ...
```

After:

```
1  #pragma omp parallel for
2  for (int i = 0; i < nParticles; i++) { // Particles that experience force
3  float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
4  for (int j = 0; j < nParticles; j++) { // Particles that exert force
5  // Newton's law of universal gravity
6  ...
```

# IMPROVING SCALAR EXPRESSIONS

Before:

```
1  const float drSquared = dx*dx + dy*dy + dz*dz + 1e-20;  
2  const float drPower32 = pow(drSquared, 3.0/2.0);  
3  // Calculate the net force  
4  Fx += dx/drPower32;  Fy += dy/drPower32;  Fz += dz/drPower32;
```

After:

```
1  const float drRecip    = 1.0f/sqrtf(dx*dx + dy*dy + dz*dz + 1e-20);  
2  const float drPowerN32 = drRecip*drRecip*drRecip;  
3  // Calculate the net force  
4  Fx += dx*drPowerN32;  Fy += dy*drPowerN32;  Fz += dz*drPowerN32;
```

- ▶ Strength reduction (division → multiplication by reciprocal)
- ▶ Precision control (suffix `-f` on single-precision constants and functions)
- ▶ Reliance on hardware-supported reciprocal square root

# VECTORIZING WITH UNIT-STRIDE MEMORY ACCESS

Before:

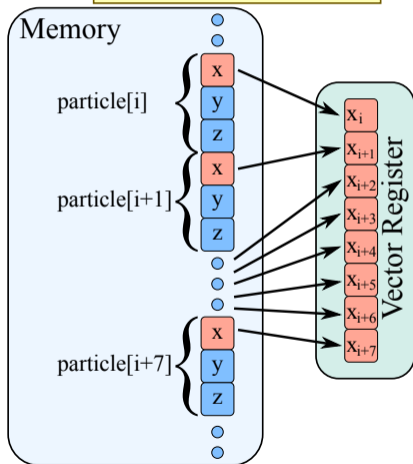
```
1 struct ParticleType {
2     float x, y, z, vx, vy, vz;
3 }; // ...
4     const float dx = particle[j].x - particle[i].x;
5     const float dy = particle[j].y - particle[i].y;
6     const float dz = particle[j].z - particle[i].z;
```

After:

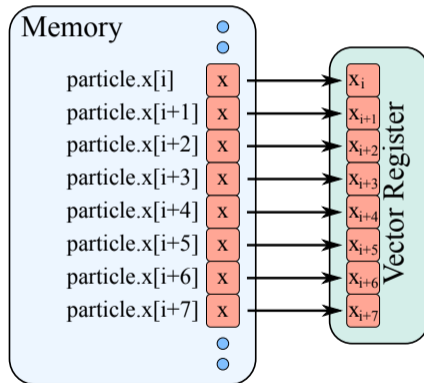
```
1 struct ParticleSet {
2     float *x, *y, *z, *vx, *vy, *vz;
3 }; // ...
4     const float dx = particle.x[j] - particle.x[i];
5     const float dy = particle.y[j] - particle.y[i];
6     const float dz = particle.z[j] - particle.z[i];
```

# WHY AOS TO SOA CONVERSION HELPS: UNIT STRIDE

Array of Structures  
(sub-optimal)



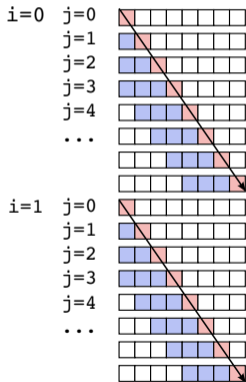
Structure of Arrays  
(optimal)



# 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

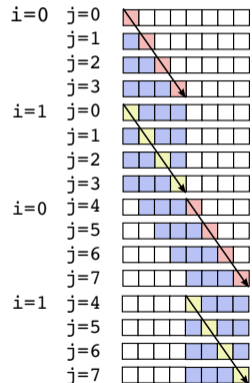
(must be tuned to cache size)

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];
```



# IMPROVING CACHE TRAFFIC

Before:

```

1  for (int i = 0; i < nParticles; i++) { // Particles that experience force
2  float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
3  for (int j = 0; j < nParticles; j++) { // Particles that exert force
4  // ...
5  Fx += dx*drPowerN32; Fy += dy*drPowerN32; Fz += dz*drPowerN32;

```

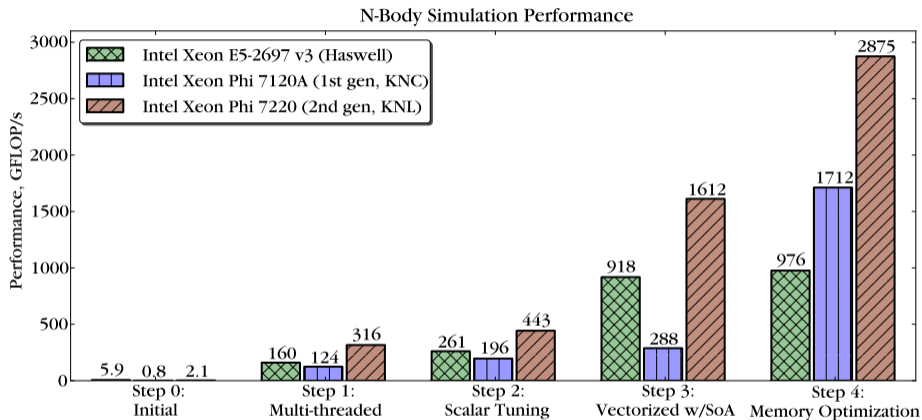
After: (tileSize = 16)

```

1  for (int ii = 0; ii < nParticles; ii += tileSize) { // Particle blocks
2  float Fx[tileSize], Fy[tileSize], Fz[tileSize]; // Force on particle block
3  Fx[:] = Fy[:] = Fz[:] = 0;
4  #pragma unroll(tileSize)
5  for (int j = 0; j < nParticles; j++) { // Particles that exert force
6  for (int i = ii; i < ii + tileSize; i++) { // Traverse the block
7  // ...
8  Fx[i-ii] += dx*drPowerN32; Fy[i-ii] += dy*drPowerN32; Fz[i-ii] += dz*drPowerN32;

```

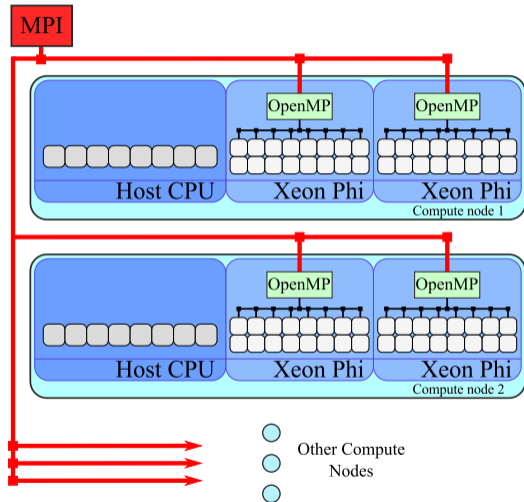
# IMPACT OF CODE OPTIMIZATION



Contributed as Chapter 23 in “[Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition](#)” (2016)

# SCALING ACROSS A CLUSTER WITH COPROCESSORS WITH MPI

- ▶ MPI processes only on CPUs
- ▶ Divide data between coprocessors
- ▶ Concurrent offload from multiple host threads
- ▶ Synchronize data between nodes with MPI



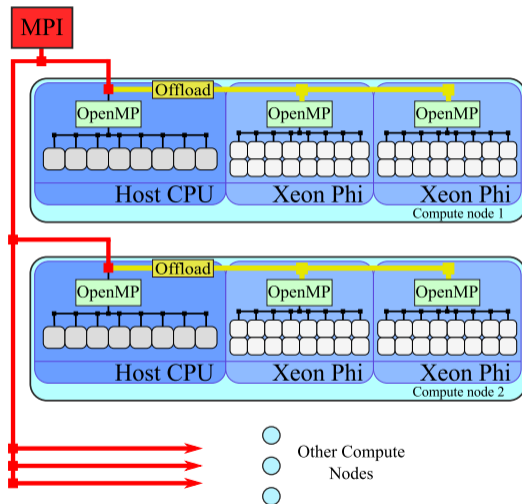
# CORE OF MPI-ONLY IMPLEMENTATION

Simple: all particles on each compute node; exchange updated particle coordinates.

```
1 void MoveParticles(int nParticles, ParticleSet& particle, float dt,
2                   int mpiRank, int mpiWorldSize) {
3     const int myParticles = nParticles/mpiWorldSize;
4     const int startParticle = (mpiRank    )*myParticles;
5     const int endParticle   = (mpiRank + 1)*myParticles;
6     // Outer loop over only the subset of particles processed by present process
7     #pragma omp parallel for schedule(guided)
8     for (int ii = startParticle; ii < endParticle; ii += tileSize) {
9         for (int j = 0; j < nParticles; j++) // ...But inner loop over all particles
10            //...
11    }
12    // ... Propagate results of time step across the cluster
13    MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, particle.x,
14                 myParticles, MPI_FLOAT, MPI_COMM_WORLD);
15    // ...
```

# SCALING ACROSS A CLUSTER WITH COPROCESSORS

- ▶ MPI processes only on CPUs
- ▶ Divide data between coprocessors
- ▶ Concurrent offload from multiple host threads
- ▶ Synchronize data between nodes with MPI



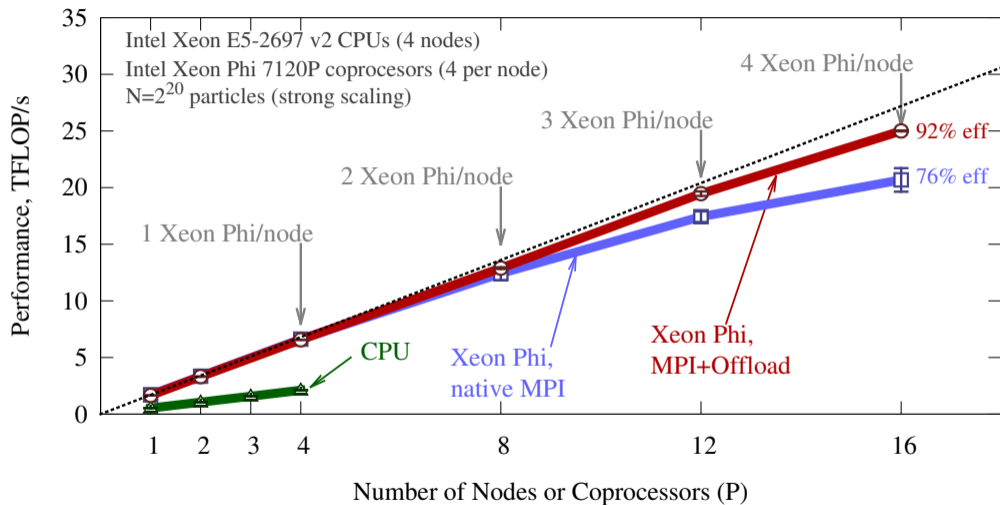
# MPI WITH OFFLOAD IMPLEMENTATION

```

1  const int nDevices = _Offload_number_of_devices();
2  const int particlesPerDevice=(nDevices==0 ? myParticles : myParticles/nDevices);
3  #pragma omp parallel num_threads(nDevices) if(nDevices>0)
4  {
5      const int iDevice = omp_get_thread_num();
6      const int startParticle = rankStartParticle + (iDevice )*particlesPerDevice;
7      #pragma offload target(mic:iDevice) if(nDevices>0) \
8          in (x : length(nParticles)          alloc_if(alloc==1) free_if(0)) \
9          out(x [startParticle:particlesPerDevice] : alloc_if(0) free_if(alloc==1)) \
10         in (vx: length(nParticles*alloc*alloc)          alloc_if(alloc==1) free_if(0)) \
11         //...
12         { // Loop over particles that experience force
13     #pragma omp parallel for schedule(guided)
14         for (int ii = startParticle; ii < endParticle; ii += tileSize) {
15             // ...

```

# RESULTS WITH MPI+OFFLOAD





# **INTEL ARCHITECTURE**

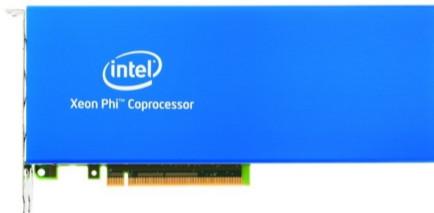
## Intel Xeon Processor



Current: Broadwell  
Upcoming: Skylake

Multi-Core Architecture

## Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

## Intel Xeon Phi Processor, 2nd generation\*



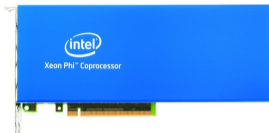
\* socket and coprocessor versions

Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture



- C/C++/Fortran
- Linux/Windows
- $\leq 3$  TiB DDR4
- $\leq 44$  cores (2-way)
- $\approx 3$  GHz
- 2 HT/core
- 256-bit AVX



- C/C++/Fortran
- Special Linux
- $\leq 16$  GiB GDDR5
- 57-61 cores
- $\approx 1.2$  GHz
- 4 HW THR/core
- 512-bit IMCI



- C/C++/Fortran
- Linux
- MCDRAM+DDR4
- 64-72 cores
- 1.3-1.5 GHz
- 4 HT/core
- 512-bit AVX-512

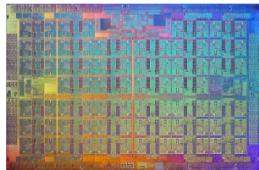
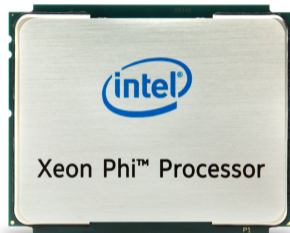


## **CORES AND VECTORS**

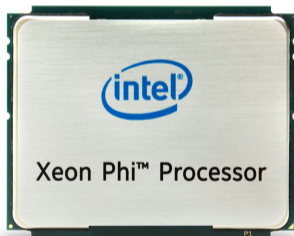
# INTEL XEON PHI PROCESSORS (2ND GEN)

Specialized platform for demanding computing applications.

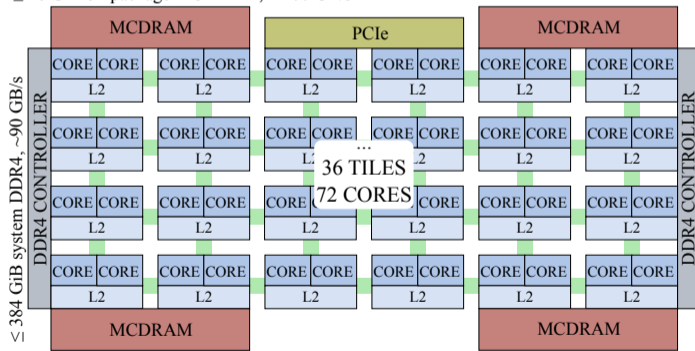
- ▶ Socket version or coprocessor
- ▶ 64-72 cores  $\times$  4 HT at 1.3-1.5 GHz
- ▶ 3+ TFLOP/s in DP (FMA)
- ▶ 6+ TFLOP/s in SP (FMA)
- ▶  $\leq 384$  GiB DDR4 ( $> 90$  GB/s)
- ▶ 16 GiB HBM (MCDRAM,  $> 400$  GB/s)
- ▶ Binary-compatible with Xeon
- ▶ Common OS  
(RHEL/CentOS/SUSE/Windows)



- ▶ Mesh interconnect relaxes data locality requirement [somewhat]
- ▶ All-to-all, quadrant or sub-numa domain communication in mesh

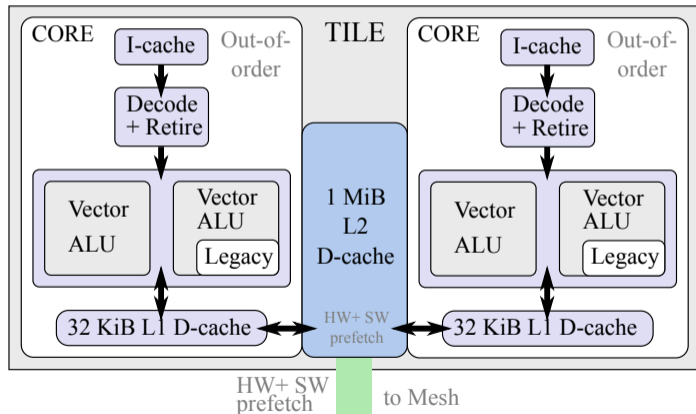
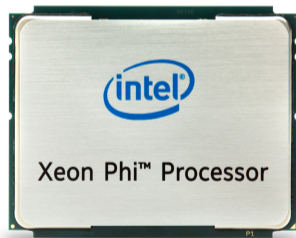


≤ 16 GiB on-package MCDRAM, ~ 400 GB/s

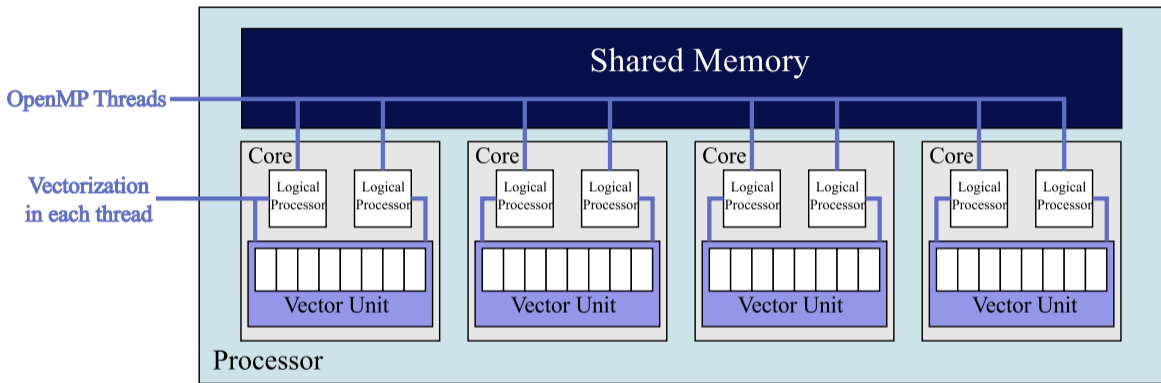


# KNL CORES

- ▶ Even more power in vector units
- ▶ Binary compatible with Xeon, but in legacy mode



# CO-EXISTENCE WITH VECTORS



**Utilize cores:** run multiple threads/processes (MIMD)

**Utilize vectors:** each thread (process) issues vector instructions (SIMD)

# SIMULTANEOUS THREADING AND VECTORIZATION

This approach often works:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++) // Thread parallelism in outer loop
3 #pragma simd
4   for (int j = 0; j < m; j++) // Vectorization in inner loop
5     DoSomeWork(A[i][j]);
```

That works as well:

```
1 #pragma omp parallel for simd
2 for (int i = 0; i < n; i++) // If the problem is all data-parallel
3   DoSomeWork(A[i]);
```

# SHORT VECTOR SUPPORT

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

Scalar Instructions

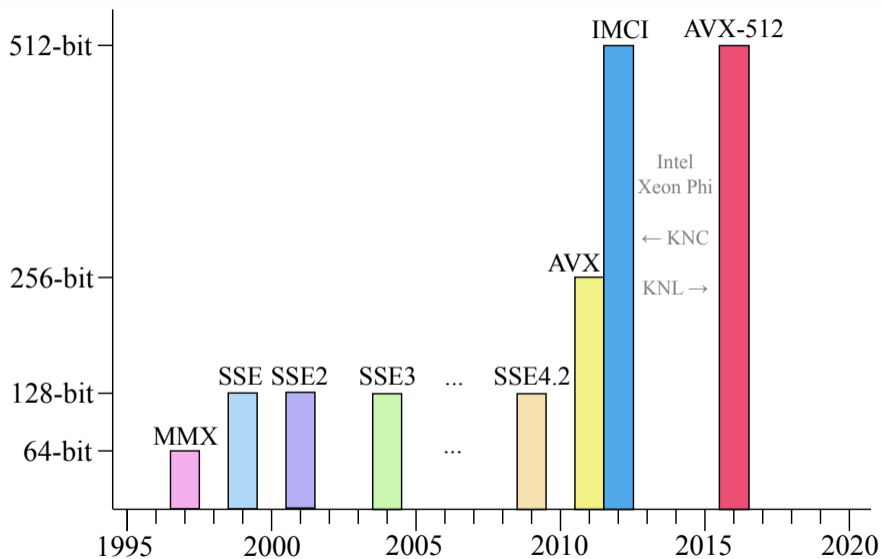
$$\begin{array}{r} 4 + 1 = 5 \\ 0 + 3 = 3 \\ -2 + 8 = 6 \\ 9 + -7 = 2 \end{array}$$

Vector Instructions

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

Vector Length

# INSTRUCTION SETS IN INTEL ARCHITECTURE

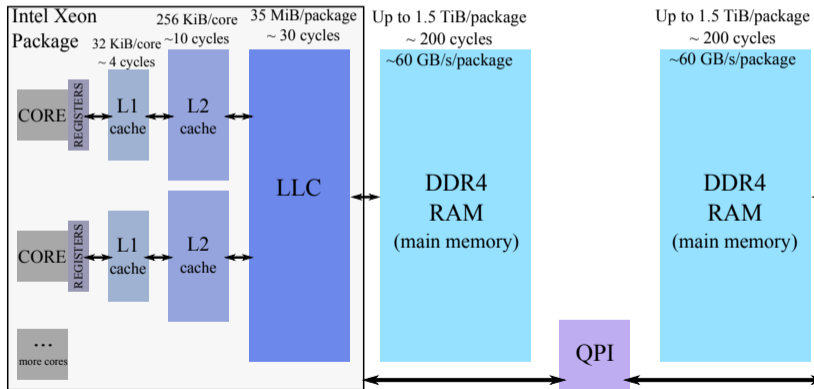




# **MEMORY ORGANIZATION**

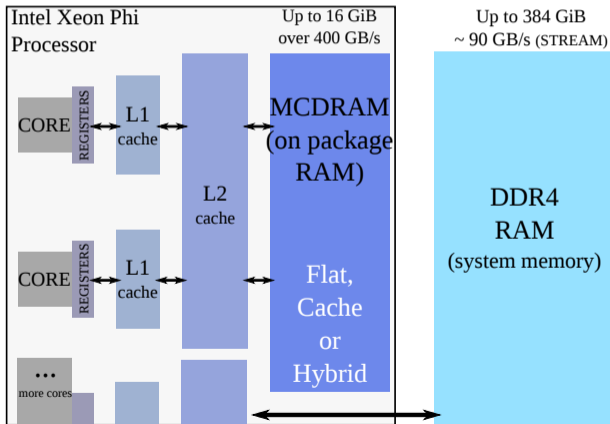
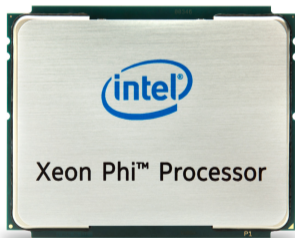
# INTEL XEON CPU: MEMORY ORGANIZATION

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



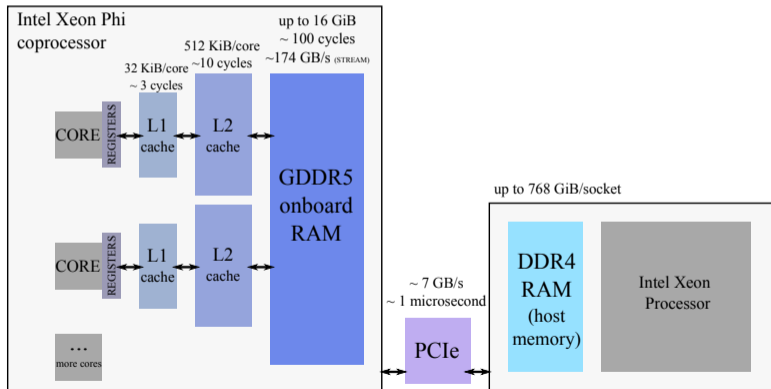
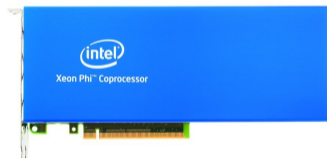
# KNL MEMORY ORGANIZATION (BOOTABLE)

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



# KNC MEMORY ORGANIZATION

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

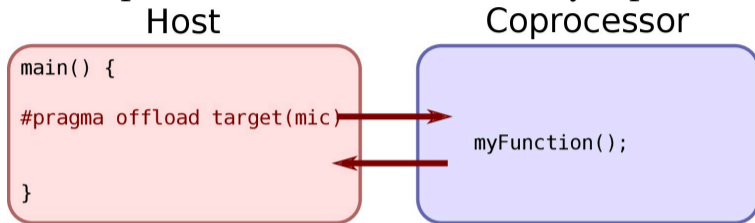




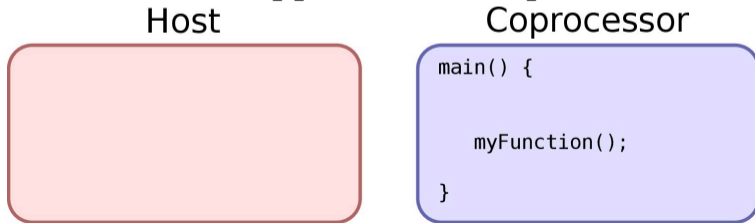
## **COPROCESSORS AND CLUSTERS**

# OFFLOAD AND NATIVE MODELS

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



- ▶ Native model (standalone application/MPI process):



# INTEL COMPILERS + INTEL XEON PROCESSOR

“Hello World” application:

```
1 #include <stdio>
2 #include <unistd.h>
3 int main(){
4     printf("Hello world! I have %ld logical processors.\n",
5         sysconf(_SC_NPROCESSORS_ONLN ));
6 }
```

Compile and run on host CPU:

```
vega@lyra% icpc hello.cc -xhost
vega@lyra% ./a.out
Hello world! I have 48 logical processors.
vega@lyra%
```

# NATIVE EXECUTION ON AN INTEL XEON PHI COPROCESSOR (KNC)

Compile and run the same code on the coprocessor in the native mode:

```
vega@lyra% icpc hello.cc -mmic # Cross-compile
vega@lyra% scp a.out mic0:~/ # Put executable on coprocessor
a.out 100% 10KB 10.4KB/s 00:00
vega@lyra% ssh mic0 # Log in to coprocessor
vega@mic0% pwd
/home/lyra
vega@mic0% ls
a.out
vega@mic0% ./a.out # Launch application
Hello world! I have 244 logical processors.
vega@mic0%
```

- ▶ Use `-mmic` to produce executable for MIC architecture
- ▶ Must transfer executable to coprocessor (or NFS-share) and run from shell
- ▶ Native MPI applications work the same way (need Intel MPI library)

# NATIVE APPLICATIONS WITH AUTOTOOLS

- ▶ Use the Intel compiler with flag `-mmic`
- ▶ Knights Landing: `-xMIC-AVX512`
- ▶ Eliminate assembly and unnecessary dependencies
- ▶ Use `--host=x86_64` to avoid “program does not run” errors

Example, the GNU Multiple Precision Arithmetic Library (GMP):

```
vega@lyra% wget https://ftp.gnu.org/gnu/gmp/gmp-5.1.3.tar.bz2
vega@lyra% tar -xf gmp-5.1.3.tar.bz2
vega@lyra% cd gmp-5.1.3
vega@lyra% ./configure CC=icc CFLAGS="-mmic" --host=x86_64 --disable-assembly
...
vega@lyra% make
...
```

# EXPLICIT OFFLOAD: PRAGMA-BASED APPROACH

“Hello World” in the explicit offload model:

```
1 #include <stdio>
2 int main() {
3     printf("Hello World from host!\n");
4     #pragma offload target(mic)
5     {
6         printf("Hello World from coprocessor!\n"); fflush(stdout);
7     }
8     printf("Bye\n");
9 }
```

Application runs on the host, but some parts of code and data are moved (“offloaded”) to the coprocessor.

Detailed syntax in the [Intel C++ Compiler Reference](#).

# COMPILING AND RUNNING AN OFFLOAD APPLICATION

```
vega@lyra% icpc hello_offload.cc -o hello_offload
vega@lyra% ./hello_offload
Hello World from host!
Bye
Hello World from coprocessor!
```

- ▶ No additional arguments (for Intel compiler)
- ▶ Launch on host as a regular application
- ▶ Code inside of `#pragma offload` is offloaded automatically
- ▶ Console output on coprocessor buffered, mirrored to the host
- ▶ If no coprocessor available, default behavior is error; may be overridden to fall back to host

# OFFLOADING MULTIPLE FUNCTIONS

```
1 #pragma offload_attribute(push, target(mic))
2 void MyFunctionOne() {
3 // ... implement function as usual
4 }
5
6 void MyFunctionTwo() {
7 // ... implement function as usual
8 }
9 #pragma offload_attribute(pop)
```

- ▶ To mark a long block of code with the offload attribute, use `#pragma offload_attribute(push/pop)`

# DATA MARSHALLING FOR DYNAMICALLY ALLOCATED DATA

```
1 double *p1=(double*)malloc(sizeof(double)*N);
2 double *p2=(double*)malloc(sizeof(double)*N);
3
4 #pragma offload target(mic) in(p1 : length(N)) out(p2 : length(N))
5 {
6     // ... perform operations on p1[] and p2[]
7 }
```

- ▶ #pragma offload recognizes clauses in, out, inout and nocopy
- ▶ Data size (length), alignment, redirection, and other properties may be specified
- ▶ Marshalling is required for pointer-based data

# OPTIONAL OFFLOAD, FALL-BACK TO HOST

```
1 #pragma offload target(mic) optional
2 {
3     printf("Hello World! I have %d logical processors.\n",
4         sysconf(_SC_NPROCESSORS_ONLN )); fflush(stdout);
5 }
```

```
vega@lyra% icpc Offload-Fallback.cc -o Offload-Fallback
vega@lyra% ./Offload-Fallback
Hello World! I have 244 logical processors.
vega@lyra% sudo systemctl stop mpss # Disabling coprocessors
vega@lyra% ./Offload-Fallback
Hello World! I have 48 logical processors.
```

## OPENMP 4.0 TARGET OFFLOAD

- ▶ Another API for offload: `#pragma omp target`
- ▶ Part of the OpenMP 4.0 standard
- ▶ Designed as portable solution (coprocessors, GPGPUs)
- ▶ On Xeon Phi, uses the same back-end as `#pragma offload`

```
1 #pragma omp target
2 {
3 #pragma omp parallel for
4   for(int i=0; i<size; i++)
5     data[i] = 0;
6 }
```

Application runs on the host, but some parts of code and data are moved (“offloaded”) the coprocessor. Scope-local scalars and stack arrays offloaded automatically.

# CLAUSES OF PRAGMA OMP TARGET

```
1 #pragma omp target [clause[, clause[, ...]]]
```

- ▶ `device(int)` – offload to a specific device (coprocessor)
- ▶ `map([type:] variables)` – create data environment. `type` is `to`, `from`, `tofrom` or `alloc`
- ▶ `if(expr)` – optional offload

Link to [reference manual](#).

# SHARED VIRTUAL MEMORY MODEL

```
1  _Cilk_shared int arr[N]; // This is a virtual-shared array
2
3  _Cilk_shared void Compute() { // This function may be offloaded
4      // ... function uses array arr[]
5  }
6
7  int main() {
8      // arr[] can be initialized on the host
9      _Cilk_offload Compute(); // and used on coprocessor
10     // and the values are returned to the host
11 }
```

- ▶ Alternative to Explicit Offload
- ▶ Data synced from host to coprocessor before the start of offload
- ▶ Data synced from coprocessor to host at the end of offload

# SHARED VIRTUAL MEMORY MODEL

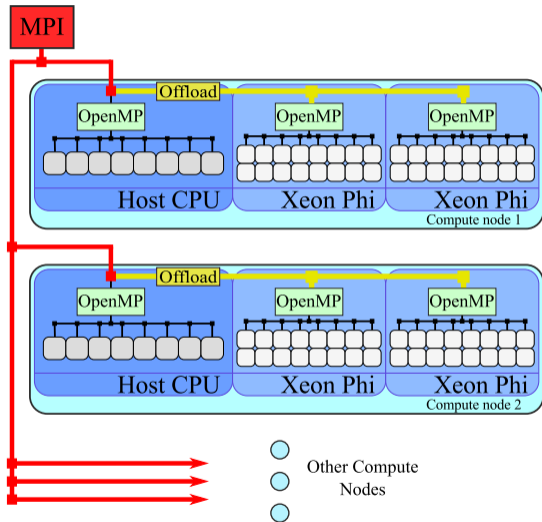
```
1  int* _Cilk_shared data; // Pointer to a virtual-shared array
2
3  int main() {
4      // Working with pointer-based data is illustrated below:
5      data = (_Cilk_shared int*)_Offload_shared_malloc(N*sizeof(float));
6      _Offload_shared_free(data);
7  }
```

- ▶ Addresses of virtual-shared pointers identical on host and coprocessors
- ▶ Synchronized before and after offload, with page granularity

# HETEROGENEOUS DISTRIBUTED COMPUTING WITH XEON PHI

## Option 1: MPI+OpenMP with Offload.

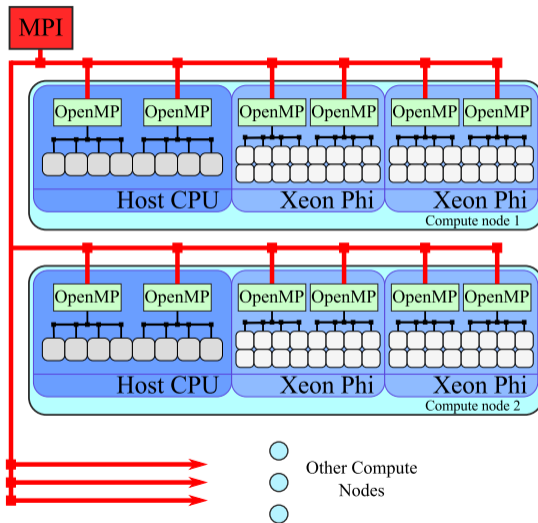
- ▶ MPI processes are multi-threaded with OpenMP.
- ▶ MPI runs only on CPUs.
- ▶ MPI processes offload to coprocessor(s).
- ▶ OpenMP in offload regions.



# HETEROGENEOUS DISTRIBUTED COMPUTING WITH XEON PHI

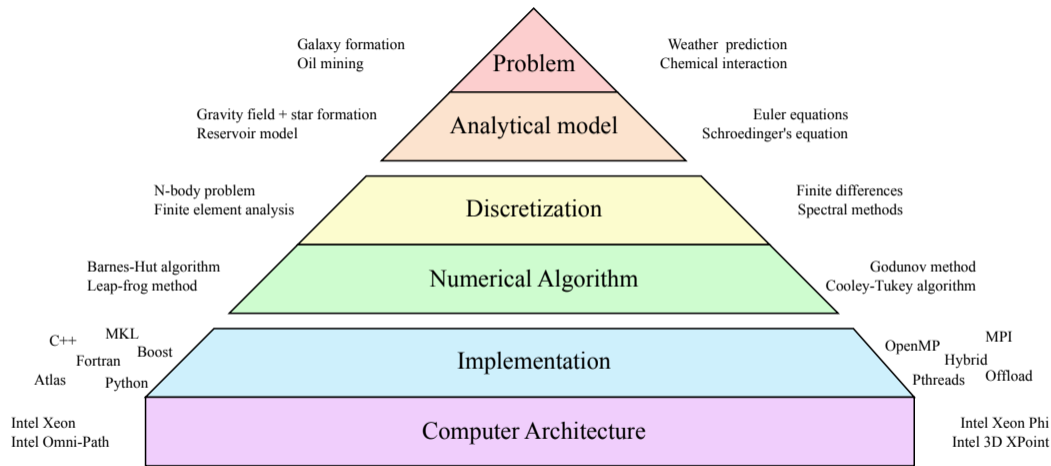
## Option 2: Symmetric hybrid MPI+OpenMP.

- ▶ MPI processes on hosts
- ▶ Native MPI processes on the coprocessor.
- ▶ Multi-threading with OpenMP.





## **§3. OPTIMIZATION POINTERS**



# SCOPE OF THIS COURSE

Areas of code optimization for Intel architecture:

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)



## SCALAR TUNING

# OPTIMIZATION LEVEL

## Default optimization level -O2

- ▶ optimization for speed
- ▶ automatic vectorization
- ▶ inlining
- ▶ constant propagation
- ▶ dead-code elimination
- ▶ loop unrolling

## Optimization level -O3

- ▶ aggressive optimization
- ▶ loop fusion
- ▶ block-unroll-and-jam
- ▶ if-statement collapse
- ▶ *may or may not be better than -O2*

For the entire file:

```
vega@lyra% icpc -o mycode -O3 source.cc
```

For a specific function:

```
1 #pragma intel optimization_level 3
2 void my_function() {
3     //...
4 }
```

# STRENGTH REDUCTION

## Common Subexpression Elimination.

```

1  for (int i = 0; i < n; i++) {
2      A[i] /= B;
3  }
```

```

1  const float Br = 1.0f/B;
2  for (int i = 0; i < n; i++)
3      A[i] *= Br;
```

## Replace division with multiplication.

```

1  for (int i = 0; i < n; i++) {
2      P[i] = (Q[i]/R[i])/S[i];
3  }
```

```

1  for (int i = 0; i < n; i++) {
2      P[i] = Q[i]/(R[i]*S[i]);
3  }
```

## Use functions with Hardware support.

```

1  double r = pow(r2, -0.5);
2  double v = exp(x);
3  double y = y0*exp(log(x/x0)*
4              log(y1/y0)/log(x1/x0));
```

```

1  double r = 1.0/sqrt(r2);
2  double v = exp2(x*1.44269504089);
3  double y = y0*exp2(log2(x/x0)*
4              log2(y1/y0)/log2(x1/x0));
```

# CONSISTENCY OF PRECISION: CONSTANTS

```
1 // Bad: 2 is "int"
2 long b=a*2;
3
4 // Bad: overflow
5 long n=100000*100000;
6
7 // Bad: excessive
8 float p=6.283185307179586;
9
10 // Bad: 2 is "int"
11 float q=2*p;
12
13 // Bad: 1e9 is "double"
14 float r=1e9*p;
15
16 // Bad: 1 is "int"
17 double t=s+1;
```

```
1 // Good: 2L is "long"
2 long b=a*2L;
3
4 // Good: correct
5 long n=100000L*100000L;
6
7 // Good: accurate
8 float p=6.283185f;
9
10 // Good: 2.0f is "float"
11 float q=2.0f*p;
12
13 // Good: 1e9f is "float"
14 float r=1e9f*p;
15
16 // Good: 1.0 is "double"
17 double t=s+1.0;
```

# CONSISTENCY OF PRECISION: FUNCTIONS

```
1 // Bad: 3.14 is a double
2 float x = 3.14;
3
4 // Bad: sin() is a
5 // double precision function
6 float s = sin(x);
7
8 // Bad: round() takes double
9 // and returns double
10 long v = round(x);
11
12 // Bad: abs() is not from IML
13 // it takes int and returns int
14 int v = abs(x);
```

```
1 // Good: 3.14f is a float
2 float x = 3.14f;
3
4 // Good: sin() is a
5 // single precision function
6 float s = sinf(x);
7
8 // Good: lroundf() takes float
9 // and returns long
10 long v = lroundf(x);
11
12 // Good: fabsf() is from IML
13 // It takes and returns a float
14 float v = fabsf(x);
```

# MOVE BRANCHES OUTSIDE OF LOOPS

```
1 // Elegant, but bad for performance
2 for (i = 0; i < n; i++) {
3     if (i == 0) {
4         // Absorbing boundary
5         B[i] = 0.0;
6     } else if (i == n - 1) {
7         // Injection at boundary
8         B[i] = B[i] + 1.0;
9     } else {
10        // Diffusion between boundaries
11        B[i] = 0.25*(A[i-1] +
12                    2.0*A[i] + A[i+1]);
13    }
14 }
```

```
1 // Moving branches out of loops
2
3
4 // Absorbing boundary
5 B[i] = 0.0;
6
7 for (i = 1; i < n - 1; i++) {
8     // Diffusion between boundaries
9     B[i] = 0.25*(A[i-1] + 2.0*A[i] +
10                 A[i+1]);
11 }
12
13 // Injection at boundary
14 B[n-1] = B[n-1] + 1.0;
```

# REDUNDANT CODE IS OK

```
1 // Elegant, but bad for performance
2 for (ii = 0; ii < n; ii+=16) {
3     for (i = ii; i < ii+16; i++)
4         // Branch causes unnecessary
5         // masking of vector iterations
6         if (i < n) {
7             A[k*n + i] = ...
8         }
9 }
```

```
1 // Redundant code, but faster
2 const int nTrunc = n - 16;
3 for (ii = 0; ii < nTrunc; ii+=16) {
4     for (i = ii; i < ii+16; i++)
5         A[k*n + i] = ...
6
7     for (i = nTrunc; i < n; i++)
8         A[k*n + i] = ...
9 }
```



**VECTORIZATION: MAKE IT HAPPEN**

# AUTOMATIC VECTORIZATION OF LOOPS

```

1  #include <stdio>
2
3  int main(){
4      const int n=8;
5      int i;
6      int A[n] __attribute__((aligned(64)));
7      int B[n] __attribute__((aligned(64)));
8
9      // Initialization
10     for (i=0; i<n; i++)
11         A[i]=B[i]=i;
12
13     // This loop will be auto-vectorized
14     for (i=0; i<n; i++)
15         A[i]+=B[i];
16
17     // Output
18     for (i=0; i<n; i++)
19         printf("%2d %2d %2d\n", i, A[i], B[i]);
20 }

```

```

vega@lyra% icpc autovec.cc -qopt-report
vega@lyra% cat autovec.optrpt
...
LOOP BEGIN at autovec.cc(14,3)
remark #15399: vectorization support:
unroll factor set to 2 [autovec.cc(14,3)]
remark #15300: LOOP WAS VECTORIZED
[autovec.cc(14,3)]
LOOP END
...
vega@lyra% ./a.out
0 0 0
1 2 1
2 4 2
3 6 3
4 8 4
5 10 5
6 12 6
7 14 7

```

## VECTORIZE MORE LOOPS: `#pragma simd`

Statement `#pragma simd` is used to “enforce vectorization of loops”, which includes:

- ▶ Loops with SIMD-enabled functions (see below)
- ▶ Second innermost loops
- ▶ Failed vectorization due to compiler decision
- ▶ Loops where guidance is required (vector length, reduction, etc.)

See compiler reference on `#pragma simd` for more information.

# EXTENSIONS FOR ARRAY NOTATION

Array notation is a method for specifying

- ▶ slices of arrays (begin, length)

```
1 A[0:16] += B[32:16]; // B[32]...B[47] added to A[0]...A[15]
```

- ▶ a stride (begin, length, stride)

```
1 A[0:16:2] += B[32:16:4]; // B[32],B[36]...B[92] added A[0],A[2]...A[30]
```

- ▶ Multi-dimensional arrays

```
1 A[:, :] += B[:, :]; // Add B to A; arrays are of the same shape
```

Better than strided loops (e.g., [this paper](#)).

# SIMD-ENABLED FUNCTIONS

(formerly “elemental functions”)

What if the implementation of a function is in a separate source code file (e.g., a library function)?

```
1 float my_simple_add(float x1, float x2){  
2     return x1 + x2;  
3 }
```

```
1 // ...in a separate source file:  
2 for (int i = 0; i < N, ++i) {  
3     output[i] = my_simple_add(inputa[i], inputb[i]);  
4 }
```

Compiler will refuse to automatically vectorize this loop.

# STRIP-MINING: METHOD

- ▶ Strip-mining is a programming technique that turns one loop into two nested loops.
- ▶ used to expose vectorization opportunities in the inner loop.

Original code:

```
1 for (int i = 0; i < n; i++) {  
2     // ... do work  
3 }
```

Strip-mined implementation:

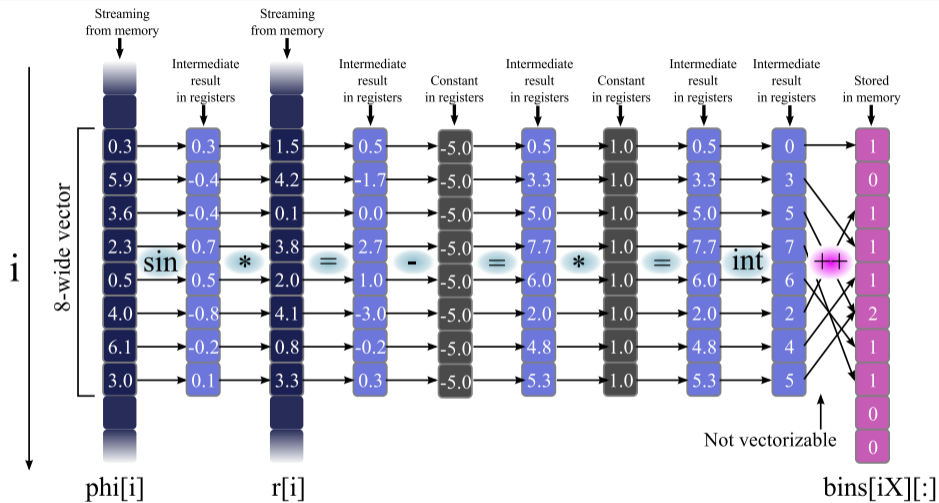
```
1 const int STRIP=1024;  
2 for (int ii = 0; ii < n; ii += STRIP)  
3     for (int i = ii; i < ii+STRIP; i++) {  
4         // ... do work  
5     }
```

# AUTO-VECTORIZED LOOPS MAY BE COMPLEX (EXAMPLE 1)

```
1  for (int i = ii; i < ii + tileSize; i++) { // Target for auto-vectorization
2
3  // Newton's law of universal gravity
4  const float dx = particle.x[j] - particle.x[i]; // x[j] is a const
5  const float dy = particle.y[j] - particle.y[i]; // x[i] makes SIMD vector
6  const float dz = particle.z[j] - particle.z[i];
7  const float rr = 1.0f/sqrtf(dx*dx + dy*dy + dz*dz + softening);
8  const float drPowerN32 = rr*rr*rr;
9
10 // Calculate the net force
11 Fx[i-ii] += dx * drPowerN32;
12 Fy[i-ii] += dy * drPowerN32;
13 Fz[i-ii] += dz * drPowerN32;
14 }
```

See also [this presentation](#)

# AUTO-VECTORIZED LOOPS MAY BE COMPLEX (EXAMPLE 2)



See [this paper](#) for more details



## **VECTOR-FRIENDLY DATA CONTAINERS**

# UNIT-STRIDE ACCESS

Unit-stride access is optimal:

```
1 for (int i = 0; i < n; i++)
2   A[i] += B[i];
```

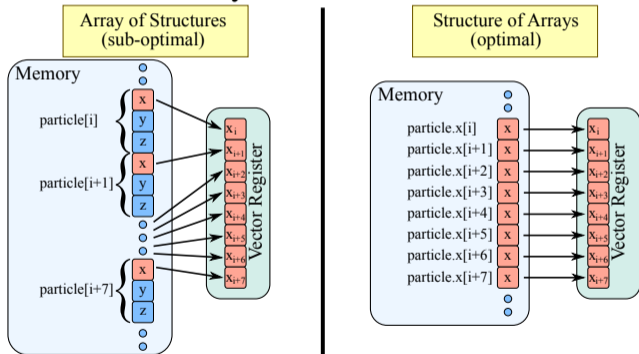
Non-unit stride is slower:

```
1 for (int i = 0; i < n; i++)
2   A[i*stride] += B[i];
```

Stochastic access may be  
vectorized (but not efficient):

```
1 for (int i = 0; i < n; i++)
2   A[offset[i]] += B[i];
```

It may be a question of changing the order of loop nesting, but sometimes you need to modify data structures:



# DATA ALIGNMENT REQUIREMENTS

Array `char* p` is `n`-byte aligned if  $((\text{size\_t})p \% n == 0)$ .

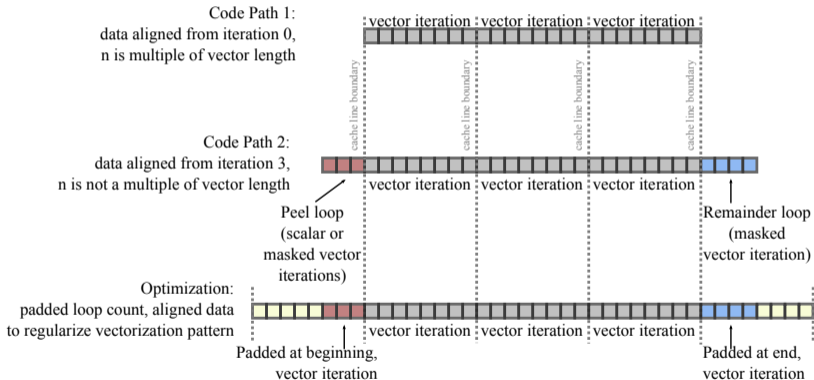
Processor	Operation	Alignment
Xeon (Westmere and earlier)	SSE load, store	16-byte
Xeon (Sandy Bridge and later)	AVX load, store	32-byte (relaxed)
Xeon Phi (1st gen)	IMCI load, store	64-byte (strict)
Xeon Phi (1st gen)	DMA transfer in offload	4096-byte (preferred)
Xeon Phi (2nd gen)	AVX-512 load, store	64-byte (relaxed)

Why align: speed up vector load/stores, avoid false sharing (see Session 7), accelerate RDMA.

# WHAT HAPPENS WITHOUT ALIGNMENT

Compiler may implement peel and remainder loops:

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



# CREATING ALIGNED DATA CONTAINERS

## ▷ Data alignment on the stack

```
1 float A[n] __attribute__((aligned(64))); // 64-byte alignment applied
```

## ▷ Data alignment on the heap

```
1 float *A = (float*) _mm_malloc(sizeof(float)*n, 64);
```

- ▷ A[0] is aligned on a 64-byte boundary.
- ▷ Very high alignment value may lead to wasted virtual memory.
- ▷ Fortran: directive or compiler argument `-align array64byte`

# PADDING MULTI-DIMENSIONAL CONTAINERS FOR ALIGNMENT

To use aligned instructions, you may need to pad inner dimension of multi-dimensional arrays to a multiple of 16 (in SP) or 8 (DP) elements.

Incorrect:

```
1 // A - matrix of size (n x n)
2 // n is not a multiple of 16
3 float* A =
4   _mm_malloc(sizeof(float)*n*n, 64);
5
6 for (int i = 0; i < n; i++)
7   // A[i*n + 0] may be unaligned
8   for (int j = 0; j < n; j++)
9     A[i*n + j] = ...
```

Correct:

```
1 // ... Padding inner dimension
2 int lda=n + (16-n%16); // lda%16==0
3 float* A =
4   _mm_malloc(sizeof(float)*n*lda, 64);
5
6 for (int i = 0; i < n; i++)
7   // A[i*lda + 0] aligned for any i
8   for (int j = 0; j < n; j++)
9     A[i*lda + j] = ...
```



## **VECTORIZATION FINE-TUNING**

# VECTORIZATION PRAGMAS, KEYWORDS AND COMPILER ARGUMENTS

- ▷ `#pragma simd`
- ▷ `#pragma vector always`
- ▷ `#pragma vector aligned | unaligned`
- ▷ `__assume_aligned` keyword
- ▷ `#pragma vector nontemporal | temporal`
- ▷ `#pragma novector`
- ▷ `#pragma ivdep`
- ▷ `restrict` qualifier and `-restrict` command-line argument
- ▷ `#pragma loop count`
- ▷ `-qopt-report -qopt-report-phase:vec`
- ▷ `-O[n]`
- ▷ `-x[code]`

## TARGETING A SPECIFIC INSTRUCTION SET

`-x [code]` instructs the compiler to target specific processor features, including instruction sets and optimizations.

code	Target architecture
MIC-AVX512	Intel Xeon Phi processors (KNL)
CORE-AVX512	Fugure Intel Xeon processors
CORE-AVX2	Intel Xeon processor E3 v3 family
CORE-AVX-I	Intel Xeon processor E3 v2, E5 v2 and E7 v2 family
AVX	Intel Xeon processor E3 and E5 family
SSE4.2	Intel Xeon processor 55XX, 56XX, 75XX and E7 family
host	architecture on which the code is compiled

# ASSUMED VECTOR DEPENDENCE

- ▶ True vector dependence makes vectorization impossible:

```

1 float *a, *b;
2 for (int i = 1; i < n; i++)
3     a[i] += b[i]*a[i-1]; // dependence on the previous element

```

- ▶ *Assumed vector dependence*: when compiler cannot determine whether vector dependence exists, auto-vectorization fails:

```

1 void mycopy(int n,
2             float* a, float* b) {
3     for (int i = 0; i < n; i++)
4         a[i] = b[i];
5 }

```

```

vega@lyra% icpc -c vdep.cc -qopt-report \
> -qopt-report-phase:vec
vega@lyra% cat vdep.optrpt
...
remark #15304: loop was not
vectorized: non-vectorizable loop
instance from multiversioning
...

```

# IGNORING ASSUMED VECTOR DEPENDENCE

## To ignore assumed vector dependence

```
#pragma ivdep
```

```
1 void mycopy(int n,  
2           float* a, float* b) {  
3     #pragma ivdep  
4     for (int i = 0; i < n; i++)  
5         a[i] = b[i];  
6 }
```

```
vega@lyra% icpc -c vdep.cc -qopt-report \  
> -qopt-report-phase:vec  
vega@lyra% cat vdep.optrpt  
...  
LOOP BEGIN at vdep.cc(4,1)  
<Multiversions v2>  
remark #15300: LOOP WAS VECTORIZED  
LOOP END
```

# MULTIVERSIONING

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat code.optrpt
...
LOOP BEGIN at code.cc(4,1)
<Multiversiomed v1>
    remark #25228: LOOP WAS VECTORIZED
LOOP END
...
LOOP BEGIN at code.cc(4,1)
<Multiversiomed v2>
    remark #15304: loop was not vectorized: non-vectorizable loop instance ....
LOOP END
```

Aliasing (true vector dependence) checked at *runtime* to choose the implementation.

# POINTER DISAMBIGUATION TO PREVENT MULTIVERSIONING

Prevent multiversioning by using `#pragma ivdep`

```
1  #pragma ivdep
2  for (int i = 0; i < n; i++)
3      // ...
```

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat vdep.optrpt
...
LOOP BEGIN at code.cc(4,1)
    remark #25228: LOOP WAS VECTORIZED
LOOP END
...
```

When keyword `restrict` is used instead, may not disambiguate different offsets of same pointer (e.g, `A[i*n+j] += A[b*n+j]`).

# DATA ALIGNMENT HINTS

Programmer may promise to the compiler (under penalty of segmentation fault) that alignment has been taken care of:

```
1 // Promising that A[i*lda + 0] is aligned for every i
2 // and the same for every other array in this loop
3 #pragma vector aligned
4     for (int j = 0; j < n; j++)
5         A[i*lda + j] -= ...
```

This can lead to significant speedups, because compiler will not implement runtime checks for alignment situation and *peel loops*.



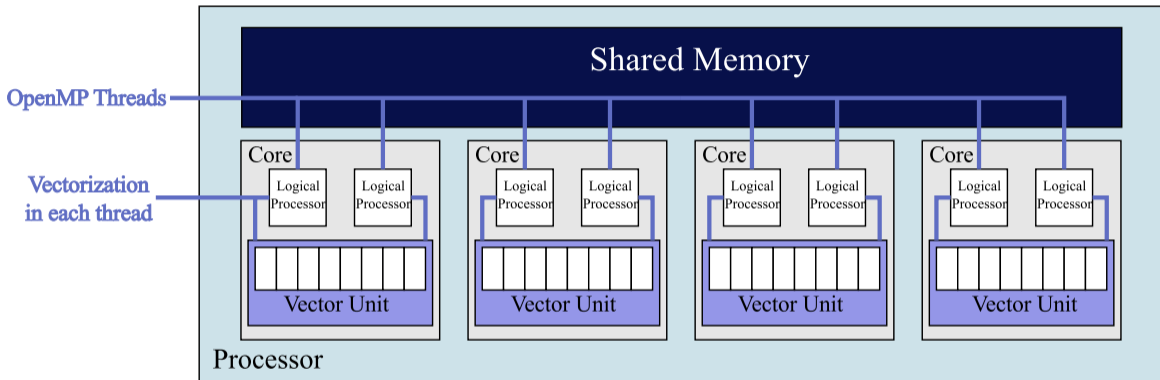
# **MULTI-THREADING WITH OPENMP**

# "HELLO WORLD" OPENMP PROGRAM

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(){
5      // This code is executed by only 1 thread
6      const int nt=omp_get_max_threads();
7      printf("OpenMP with %d threads\n", nt);
8
9      #pragma omp parallel
10     {
11         // This code is executed in parallel
12         // by multiple threads
13         printf("Hello World from thread %d\n",
14                omp_get_thread_num());
15     }
16 }
```

- ▶ OpenMP = “Open Multi-Processing” = computing-oriented framework for shared-memory programming
- ▶ Threads – streams of instructions that share memory address space
- ▶ Distribute threads across CPU cores for parallel speedup

# CO-EXISTENCE WITH VECTORS

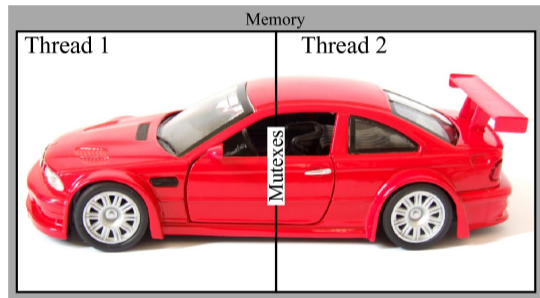
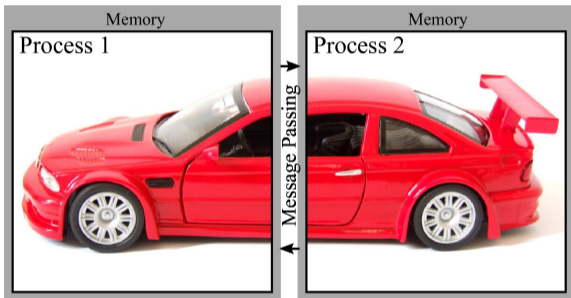


**Utilize cores:** run multiple threads/processes (MIMD)

**Utilize vectors:** each thread (process) issues vector instructions (SIMD)

# THREADS VERSUS PROCESSES

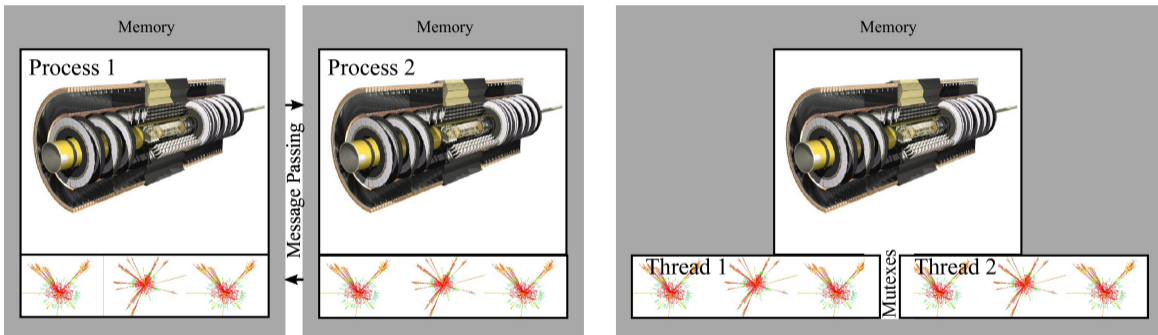
Option 1: Partitioning data set between threads/processes



Examples: computational fluid dynamics (CFD), image processing.

# THREADS VERSUS PROCESSES

## Option 2: Sharing data set between threads/processes



Examples: particle transport simulation, machine learning (inference).



## **MULTI-THREADING: COMMON ISSUES**

# RACE CONDITIONS AND UNPREDICTABLE PROGRAM BEHAVIOR

```

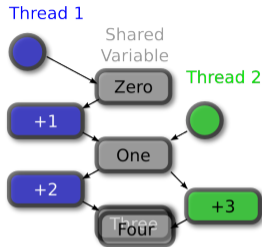
1  #include <omp.h>
2  #include <stdio.h>
3  int main() {
4      const int n = 1000;
5      int total = 0;
6      #pragma omp parallel for
7      for (int i = 0; i < n; i++) {
8          total = total + i; // Race condition
9      }
10     printf("total=%d (must be %d)\n", total,
11            ((n-1)*n)/2);
12 }

```

```

vega@lyra% icpc -o app omp-race.cc -qopenmp
vega@lyra% ./app
total=208112 (must be 499500)

```



**Race Condition!**

- Occurs when 2 or more threads access the same memory address, and at least one of these accesses is for writing

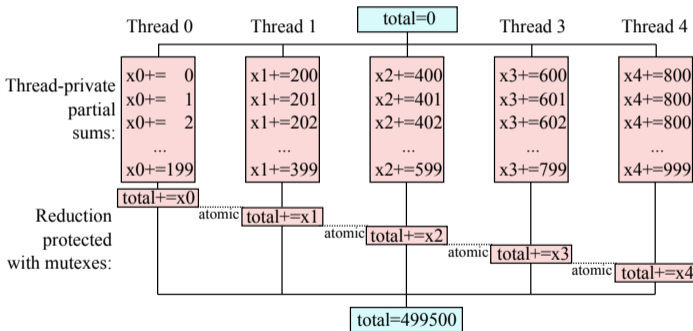
# AVOIDING RACES WITH THREAD-PRIVATE STORAGE

Correct and efficient code:

```

1  int total = 0;
2  #pragma omp parallel
3  {
4      int total_thr = 0;
5      #pragma omp for
6      for (int i=0; i<n; i++)
7          total_thr += i;
8
9      #pragma omp atomic
10     total += total_thr;
11
12 }

```



# EXAMPLE: BINNING PROBLEM

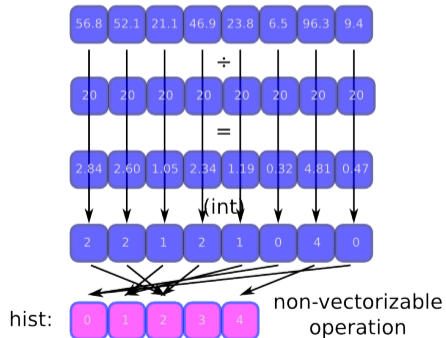
Computing a histogram ( $m \ll n$ ):

```

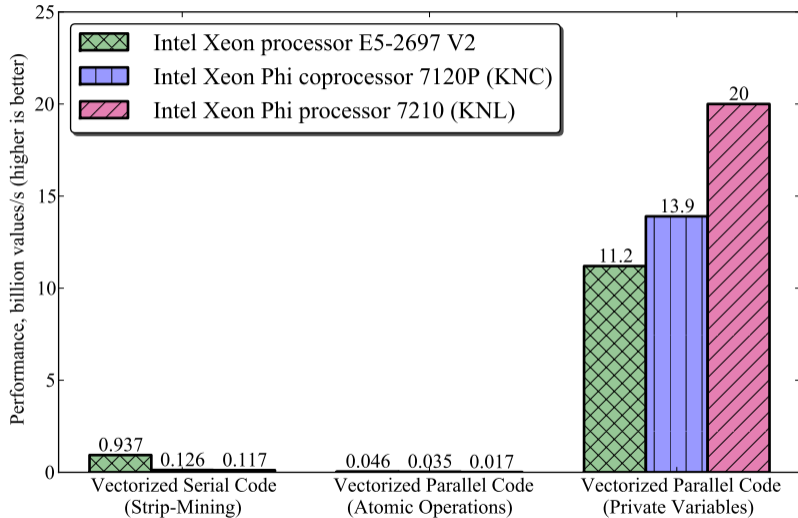
1 void Histogram(
2     // Ages, values from 0.0f to 100.0f:
3     const float* age,
4     // Size of array age, n=100000000:
5     const int n,
6     // Output: counts in groups:
7     int* const hist,
8     // Size of array hist, m=5:
9     const int m,
10    const float group_width) {
11    for (int i = 0; i < n; i++) {
12        const int j = int(age[i]/group_width);
13        hist[j]++;
14    }
15 }

```

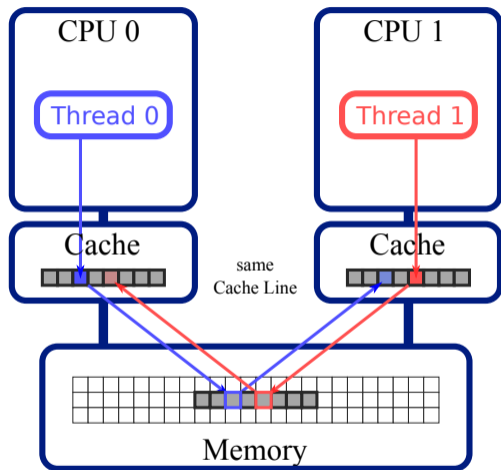
- ▶ Vector dependence in `hist[j]++`
- ▶ Strip-mine or use conflict detection



# USING REDUCTION INSTEAD OF SYNCHRONIZATION

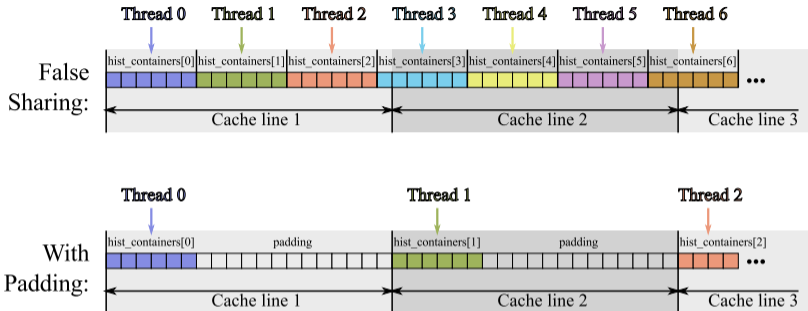


# FALSE SHARING. DATA PADDING AND PRIVATE VARIABLES



- ▶ Occurs when 2 or more threads access the same cache line, and at least one of the accesses is for writing
- ▶ Caused by *coherent caches*
- ▶ Cache line is 64-byte wide (in modern Intel architectures)

# PADDING TO AVOID FALSE SHARING

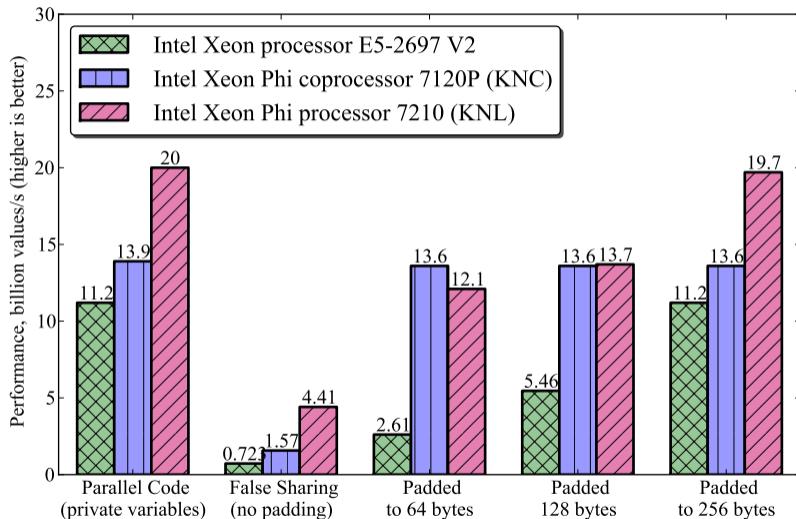


```

1 // Padding to avoid sharing a cache line between threads
2 const int paddingBytes = 64;
3 const int paddingElements = paddingBytes / sizeof(int);
4 const int mPadded = m + (paddingElements - m % paddingElements);
5 int hist_containers[nThreads][mPadded]; // New container

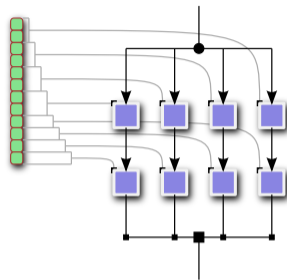
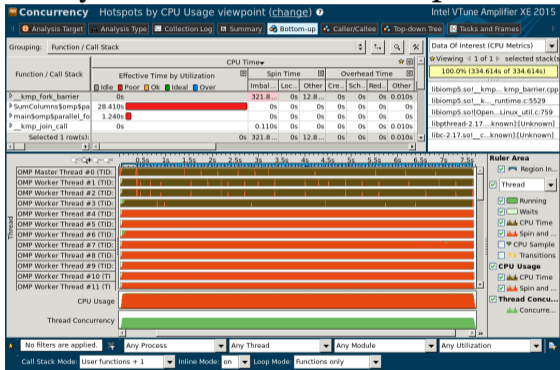
```

# PADDING TO AVOID FALSE SHARING



# INSUFFICIENT PARALLELISM

## Analysis in Intel VTune Amplifier XE



- ▶ Occurs when there are not enough iterations or parallel work-items exposed to the parallel loop in OpenMP.

## EXAMPLE: DEALING WITH INSUFFICIENT PARALLELISM

$$S_i = \sum_{j=0}^n M_{ij}, i = 0 \dots m. \quad (1)$$

- ▶  $m=4$  is small, smaller than the number of threads in the system
- ▶  $n \approx 10^8$  is large enough so that matrix does not fit into cache

```

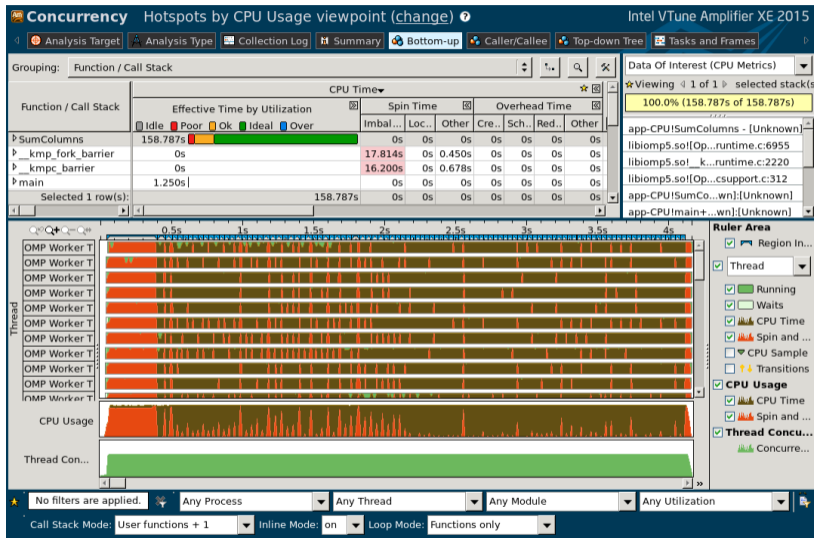
1 void sum_unoptimized(const int m, const int n, long* M, long* s){
2   #pragma omp parallel for
3     for (int i=0; i<m; i++) { // m=4
4       long total=0;
5       #pragma vector aligned
6         for (int j=0; j<n; j++) // n=100000000
7           total+=M[i*n+j];
8       s[i]=total; }}

```

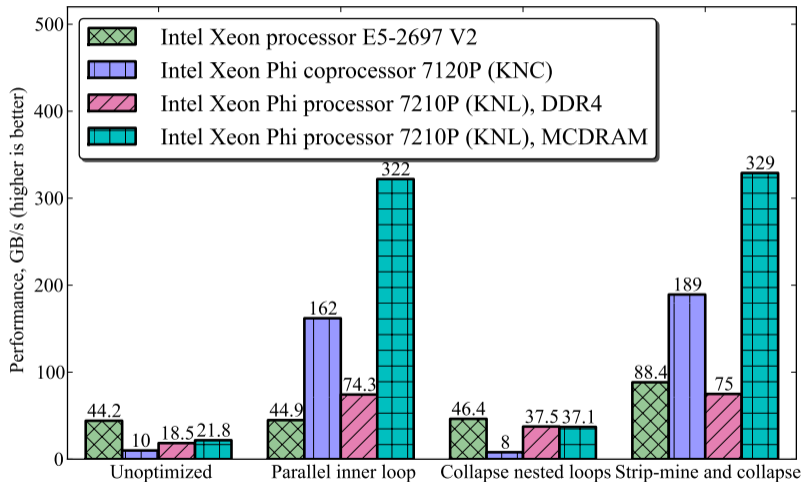
# EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE

```
1 void sum_stripmine(const int m, const int n, long* M, long* s){
2     const int STRIP=1024;
3     assert(n%STRIP==0);
4     s[0:m]=0;
5     #pragma omp parallel
6     {
7         long total[m];    total[0:m]=0;
8     #pragma omp for collapse(2) schedule(guided)
9         for (int i=0; i<m; i++)
10            for (int jj=0; jj<n; jj+=STRIP)
11    #pragma vector aligned
12                for (int j=jj; j<jj+STRIP; j++)
13                    total[i]+=M[i*n+j];
14        for (int i=0; i<m; i++)                // Reduction
15    #pragma omp atomic
16            s[i]+=total[i];
17    } }
```

# EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE



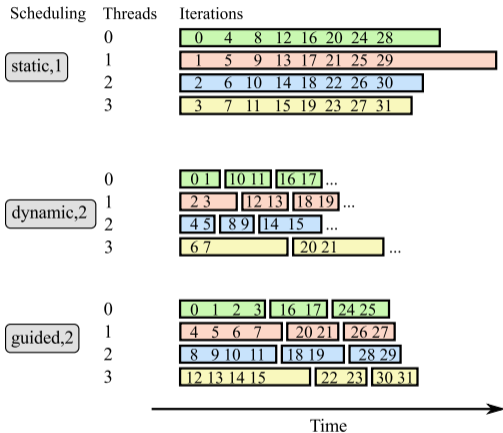
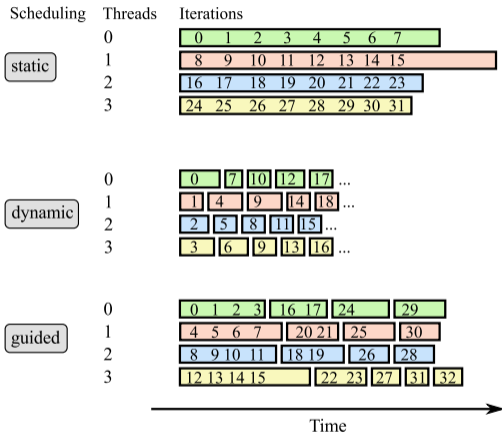
# DEALING WITH INSUFFICIENT PARALLELISM





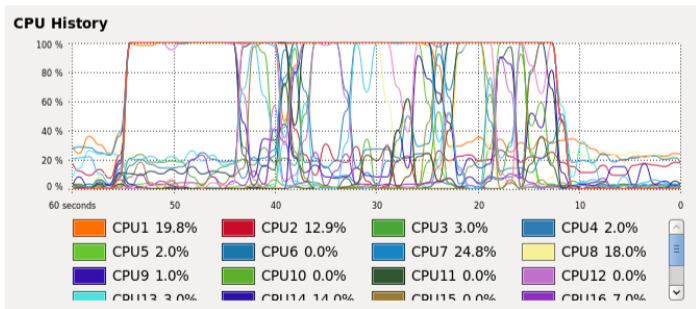
## **MULTI-THREADING: FINE-TUNING**

# LOOP SCHEDULING MODES IN OPENMP

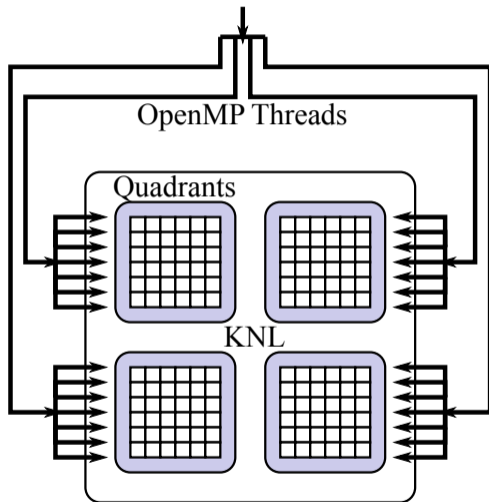


# WHAT IS THREAD AFFINITY

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



# NESTED PARALLELISM WITH OPENMP



```

1  #pragma omp parallel
2  {
3  #pragma omp parallel
4    {
5      // ...
6    }
7  }

```

- ▶ Tune granularity of parallelism
- ▶ Improve resource sharing in NUMA systems



## **CACHE AND MEMORY ACCESS**

# 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...

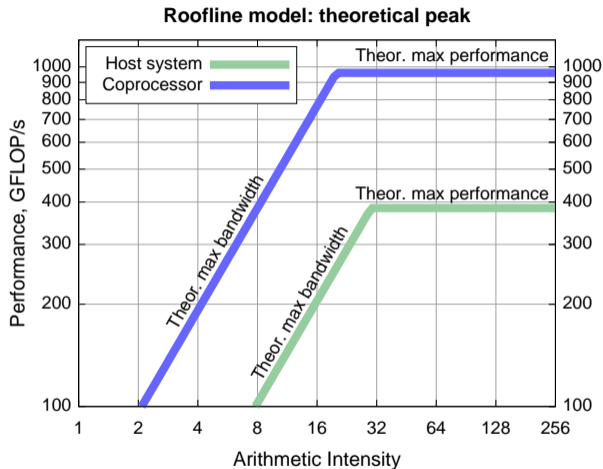
- ▶  $> 100$  (FLOPs)/(Memory Access) — Compute Bound Application
- ▶  $< 100$  (FLOPs)/(Memory Access) — Bandwidth Bound Application

# 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

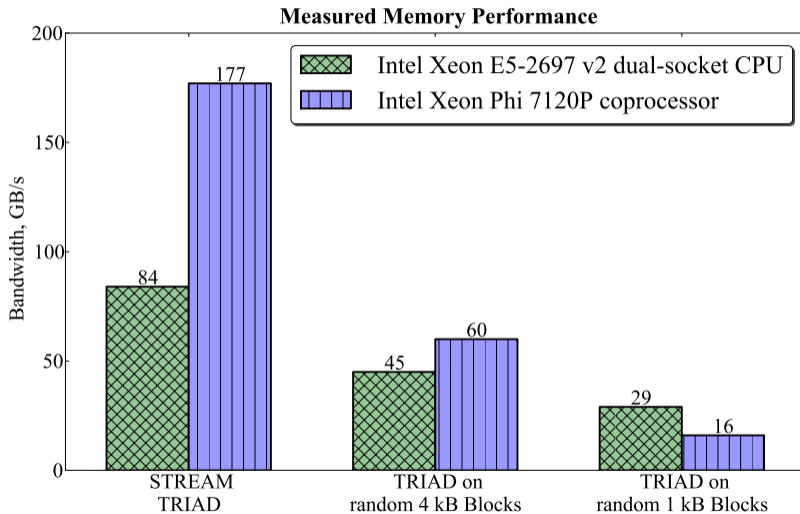
$N$  = data size

# ARITHMETIC INTENSITY AND ROOFLINE MODEL



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

# STREAMING VERSUS RANDOM ACCESS



## CHALLENGES WITH MEMORY ACCESS ON XEON PHI

- ▶ More threads than CPU, same amount of Level-2 cache (~30 MB)
- ▶ No hardware prefetching from Level-2 to Level-1
- ▶ High penalty for data page walks

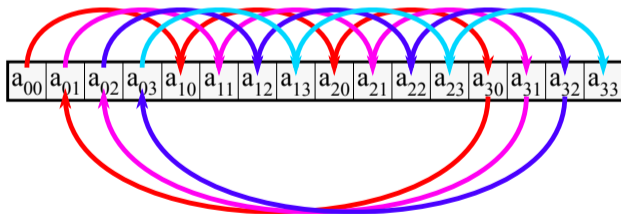
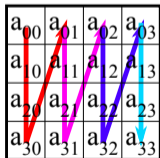
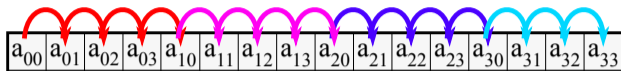
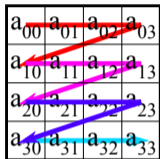
“Rule of Thumb” for memory optimization: locality of data access in space and in time.

Spatial locality = data structures (packing, reordering).

Temporal locality = order of operations (e.g., loop tiling).

# 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 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]);
```

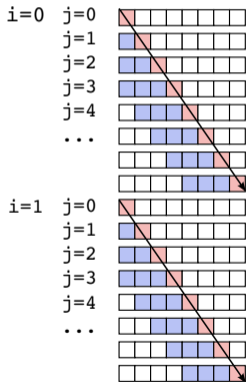
```
1 MyDataType* data = new MyDataType(n);
2
3 for (int i = 0; i < n; i++) {
4
5     Initialize(data[i]);
6
7     Stage1(data[i]);
8
9     Stage2(data[i]);
10 }
```

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

# LOOP TILING

## 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

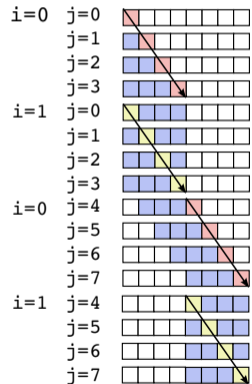
(must be tuned to cache size)

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

```
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 inner loop  
2  for (int i = 0; i < m; i++)  
3      for (int jj = 0; jj < n; jj += TILE)  
4          for (int j = jj; j < jj + TILE; j++)  
5              compute(a[i], b[j]); // Same order of operation as original
```

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

```

```

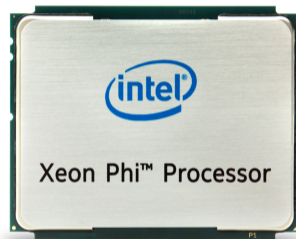
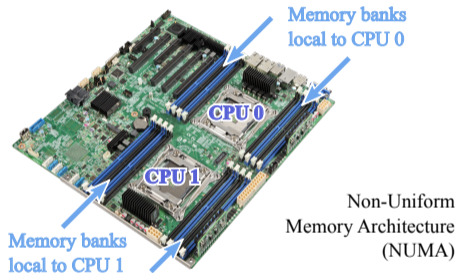
1  // Step 2: permute middle two loops
2  for (int ii = 0; ii < m; ii += TILE)
3      for (int jj = 0; jj < n; jj += VECLLEN)
4          for (int i = ii; i < ii + TILE; i++)
5              for (int j = jj; j < jj + VECLLEN; j++)
6                  compute(a[i], b[j]); // Use each vector in b[j] a total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

```



# NUMA ARCHITECTURES

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



Examples:

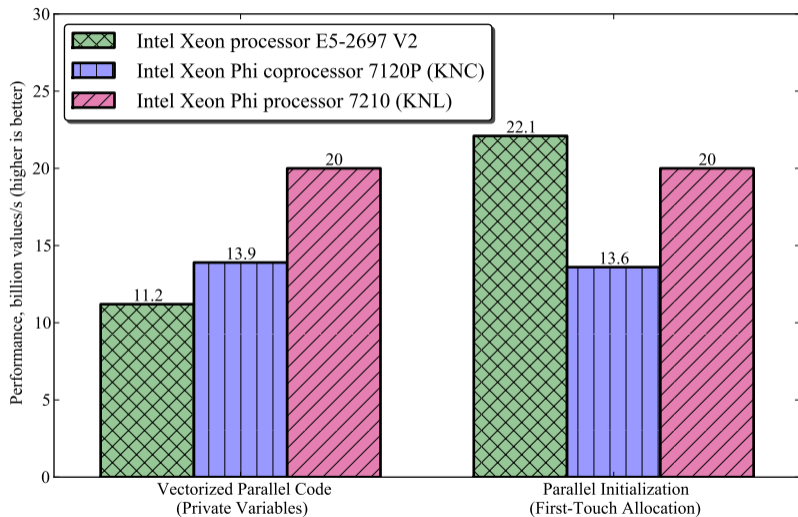
- ▶ Multi-socket Intel Xeon processors
- ▶ Second generation Intel Xeon Phi

## 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

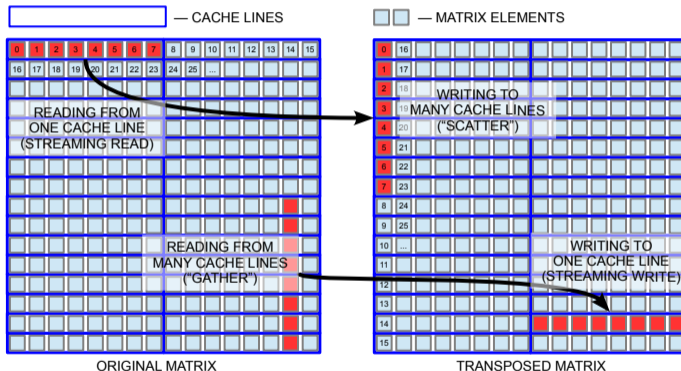
```
1 float* A = (float*)_mm_malloc(n*m*sizeof(float), 64);  
2  
3 // Initializing from parallel region for better performance  
4 #pragma omp parallel for  
5 for (int i = 0; i < n; i++)  
6     for (int j = 0; j < m; j++)  
7         A[i*m + j] = 0.0f;
```

# FIRST-TOUCH ALLOCATION POLICY



# 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: MATRIX-VECTOR MULTIPLICATION

$$c_i = \sum_{j=0}^n A_{ij} b_j, \quad i = 0, 1, \dots, (m-1). \quad (2)$$

```

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

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

# 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 } } }
```

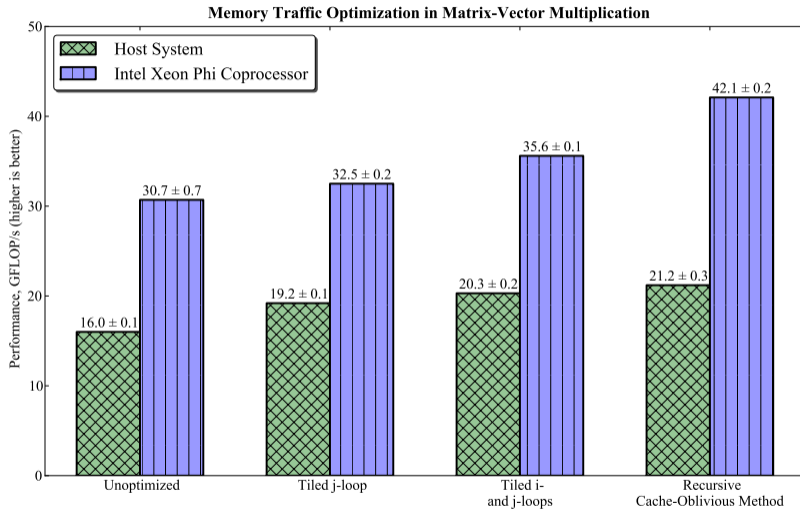
# EXAMPLE: MATRIX-VECTOR MULTIPLICATION

```

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





# **COMMUNICATION CONTROL**

# STRUCTURE OF MPI APPLICATIONS: HELLO WORLD

```
1 #include "mpi.h"
2 #include <stdio>
3 int main (int argc, char *argv[]) {
4     MPI_Init (&argc, &argv); // Initialize MPI environment
5     int rank, size, namelen;
6     char name[MPI_MAX_PROCESSOR_NAME];
7     MPI_Comm_rank (MPI_COMM_WORLD, &rank); // ID of current process
8     MPI_Get_processor_name (name, &namelen); // Hostname of node
9     MPI_Comm_size (MPI_COMM_WORLD, &size); // Number of processes
10    printf ("Hello World from rank %d running on %s!\n", rank, name);
11    if (rank == 0) printf("MPI World size = %d processes\n", size);
12    MPI_Finalize (); // Terminate MPI environment
13 }
```

# HETEROGENEOUS MPI APPLICATIONS: MACHINE FILE

```
vega@lyra% cat hosts.txt
localhost:2
mic0:2
mic1:2
vega@lyra% export I_MPI_MIC_POSTFIX=.MIC
vega@lyra% mpirun -machinefile hosts.txt ~/Hello
Hello World from rank 0 running on localhost!
Hello World from rank 1 running on localhost!
Hello World from rank 2 running on mic1!
Hello World from rank 3 running on mic1!
Hello World from rank 4 running on mic0!
Hello World from rank 5 running on mic0!
MPI World size = 6 ranks
```

- ▶ Specify Xeon executable for host processes
- ▶ MIC executable obtained by appending I\_MPI\_MIC\_POSTFIX

# COMPILING AND RUNNING MPI APPLICATIONS

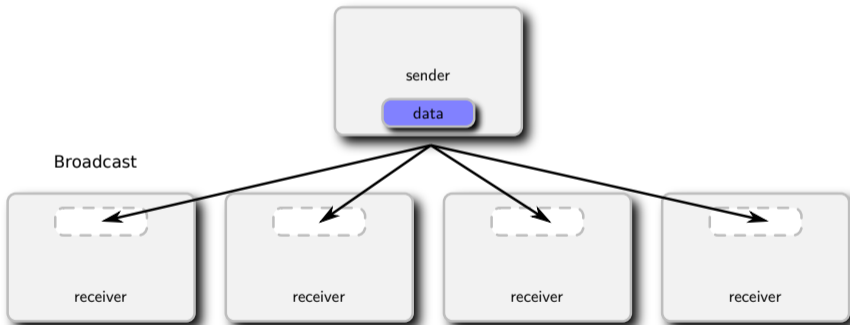
1. Compile and link with the MPI wrapper of the compiler:
  - `mpicc` for C,
  - `mpicxx` for C++,
  - `mpiifort` for Fortran 77 and Fortran 95.
2. Set up MPI environment variables *and* `I_MPI_MIC=1`
3. NFS-share or copy the MPI library and the application executable to the coprocessors
4. Launch with the tool `mpirun`
  - Colon-separated list of executables and hosts (argument `-host hostname`),
  - Alternatively, use the machine file to list hosts
  - Coprocessors have hostnames defined in `/etc/hosts`

# POINT TO POINT COMMUNICATION

```
1  if (rank == sender) {
2
3     char outgoingMsg[messageLength];
4     strcpy(outgoingMsg, "Hi There!");
5     MPI_Send(&outgoingMsg, messageLength, MPI_CHAR, receiver, tag, MPI_COMM_WORLD);
6
7
8  } else if (rank == receiver) {
9
10    char incomingMsg[messageLength];
11    MPI_Recv (&incomingMsg, messageLength, MPI_CHAR, sender,
12             tag, MPI_COMM_WORLD, &stat);
13    printf ("Received message with tag %d: '%s'\n", tag, incomingMsg);
14
15
16 }
```

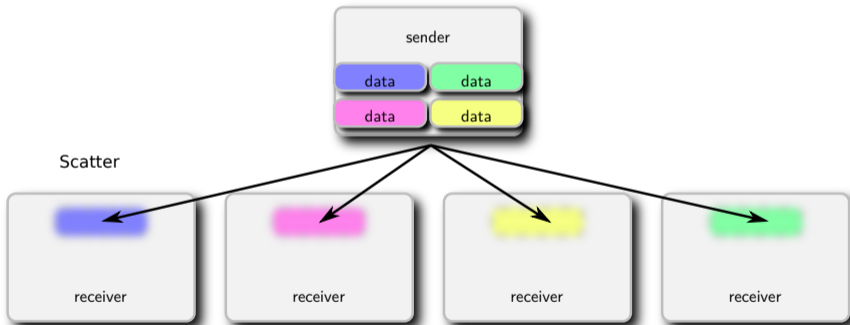
# COLLECTIVE COMMUNICATION: BROADCAST

```
1 int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype,  
2 int root, MPI_Comm comm );
```



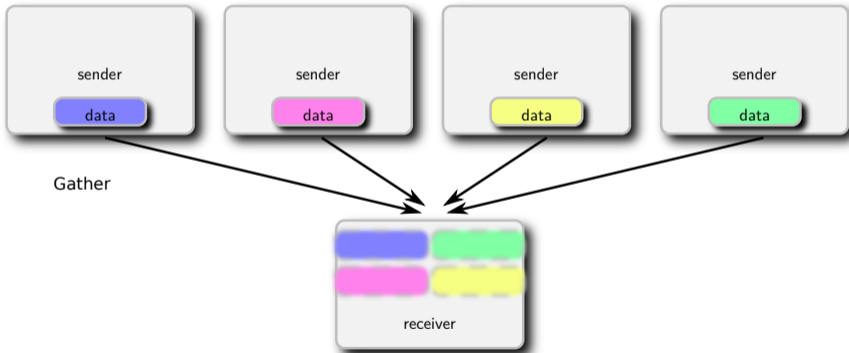
# COLLECTIVE COMMUNICATION: SCATTER

```
1 int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,  
2 int recvcnt, MPI_Datatype recvttype, int root, MPI_Comm comm);
```



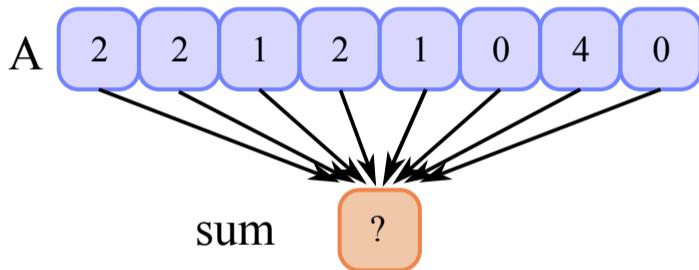
# COLLECTIVE COMMUNICATION: GATHER

```
1 int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,  
2 void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



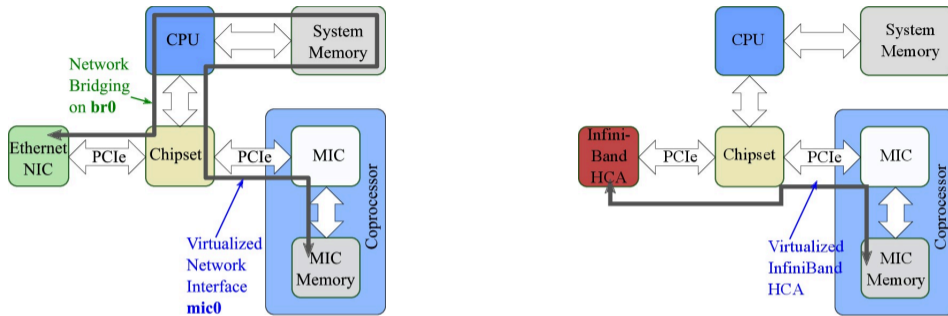
# COLLECTIVE COMMUNICATION: REDUCTION

```
1 int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,  
2 MPI_Op op, int root, MPI_Comm comm);
```



Available reducers: max/min, minloc/maxloc, sum, product, AND, OR, XOR (logical or bitwise).

# PEER-TO-PEER COMMUNICATION BETWEEN COPROCESSORS

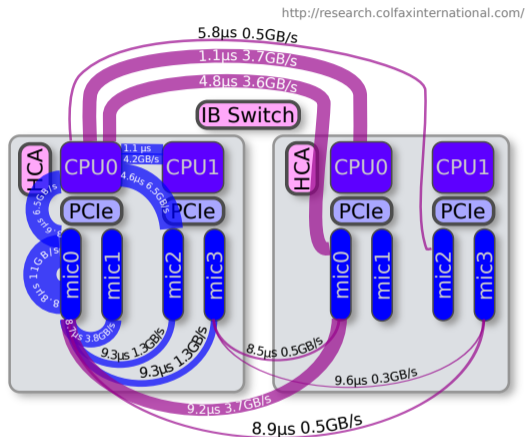


- ▶ Left: Gigabit Ethernet bridging on host allows to place coprocessors on the same subnet as hosts
- ▶ Right: Coprocessor Communication Link (CCL) – virtualization of an InfiniBand device on each coprocessor

# MPI FABRIC SELECTION

- ▶ MPI communication between CPU and coprocessors as efficient as offload
- ▶ Peer-to-peer communication not uniform, but better than with Gigabit Ethernet
- ▶ Control: environment variable `I_MPI_FABRICS`

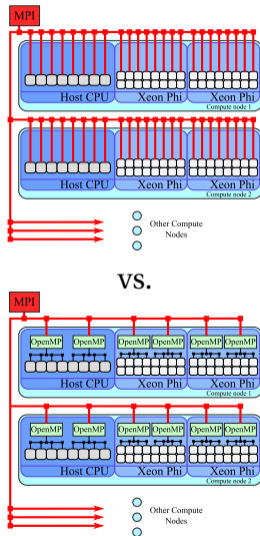
Our publication with details:  
<http://xeonphi.com/papers/p2p>



# HYBRID MPI+OPENMP

Using OpenMP inside of MPI processes:

- ▶ Reduces the memory footprint
- ▶ Decreases the number of MPI ranks, which reduces communication
- ▶ May incur thread synchronization overhead
- ▶ Optimal number of threads in MPI processes must be established empirically

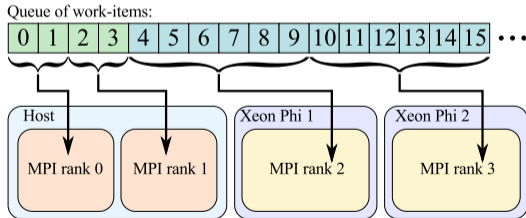


# STATIC LOAD BALANCING

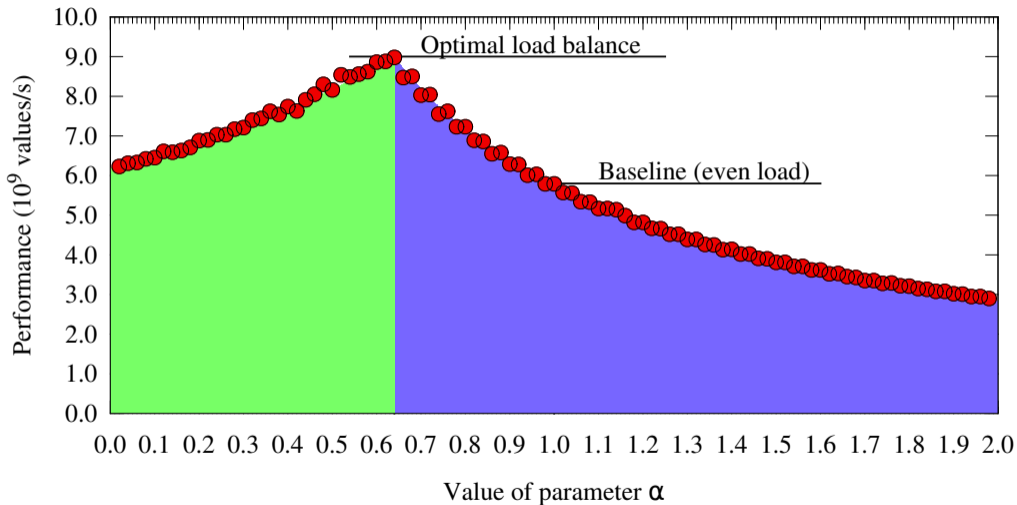
```

1  if (rankTypes[myRank] == 0) { // I am a MIC-based rank
2      double optionsPerProc = double(lastOptForCPUs)/double(cpuRanks.size());
3      myFirstOpt = int(optionsPerProc*(myGroupRank));
4      myLastOpt = int(optionsPerProc*(myGroupRank+1.0));
5  } else { // I am a CPU-based rank
6      double optionsPerProc = double(nOpts-lastOptForCPUs)/double(micRanks.size());
7      myFirstOpt=lastOptForCPUs+int(optionsPerProc*(myGroupRank));
8      myLastOpt=lastOptForCPUs+int(optionsPerProc*(myGroupRank+1.0)); }

```



# STATIC LOAD BALANCING: PARAMETER TUNING



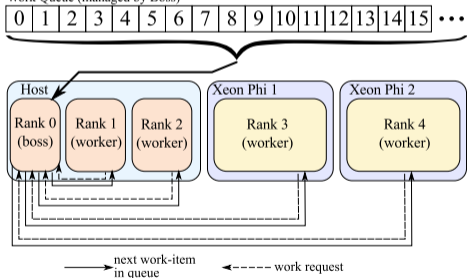
# DYNAMIC LOAD BALANCING

```

1  if (myRank == 0) // Boss's branch
2      DistributeWork(nOptions, option, mpiWorldSize);
3  else // Workers' branch
4      ReceiveWork(option, payoff, myRank);

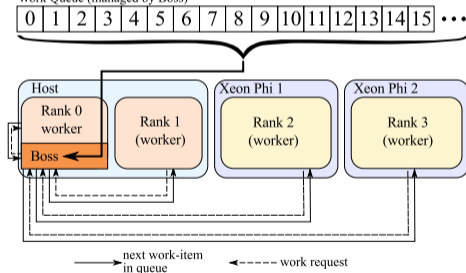
```

Work Queue (managed by Boss)



or

Work Queue (managed by Boss)

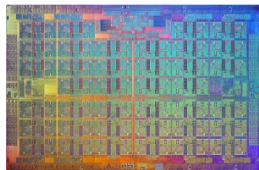
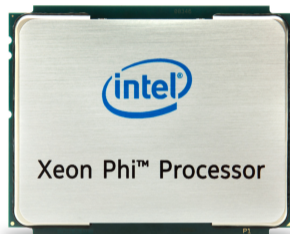


## **§4. PREPARING FOR INTEL XEON PHI PROCESSORS**

## INTEL XEON PHI PROCESSORS (2ND GEN)

Specialized platform for demanding computing applications.

- ▶ Socket version or coprocessor
- ▶ 64-72 cores × 4 HT at 1.3-1.5 GHz
- ▶ 3+ TFLOP/s in DP (FMA)
- ▶ 6+ TFLOP/s in SP (FMA)
- ▶ ≤ 384 GiB DDR4 (> 90 GB/s)
- ▶ 16 GiB HBM (MCDRAM, > 400 GB/s)
- ▶ Binary-compatible with Xeon
- ▶ Common OS  
(RHEL/CentOS/SUSE/Windows)

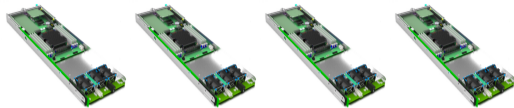


# BOOTABLE INTEL XEON PHI PROCESSORS

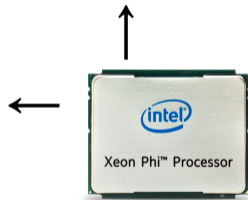
- ▶ Bootable Host Processor
- ▶ RHEL/CentOS/SUSE/Win
- ▶ 64 cores × 4 HT, 1.3 GHz
- ▶ ≤ 384 GiB DDR4, > 90 GB/s
- ▶ 16 GiB HBM, > 400 GB/s
- ▶ PCIe bus for networking

[dap.xeonphi.com](http://dap.xeonphi.com)

Servers:



Workstations:



# GET READY FOR INTEL® XEON PHI PROCESSORS (CODENAMED: KNIGHTS LANDING)



GET READY FOR KNL\*

**3**

papers

AVX-512 | CLUSTERING MODES | MCDRAM

\* Colfax series of webinars, training and white papers

[colfaxresearch.com/knl-ready](http://colfaxresearch.com/knl-ready)



HOW SERIES "KNIGHTS LANDING":

PROGRAMMING AND OPTIMIZATION FOR  
INTEL XEON PHI X200 FAMILY

Free 2-hour video course

[colfaxresearch.com/how-knl](http://colfaxresearch.com/how-knl)



## **COMPILING WITH AVX-512**

# AVX-512 FEATURES

- ▶ AVX-512F (Fundamentals)
  - Extension of most AVX2 instructions to 512-bit vector registers.
- ▶ AVX-512CD (Conflict Detection)
  - Efficient conflict detection (application: binning).
- ▶ AVX-512ER (Exponential and Reciprocal)
  - Transcendental function (exp, rcp and rsqrt) support.
- ▶ AVX-512PF (Prefetch)
  - Prefetch for scatter and gather.

Learn more: [colfaxresearch.com/avx-512](http://colfaxresearch.com/avx-512)

# INTEL COMPILER SUPPORT FOR AVX-512

Intel C, C++ and Fortran compilers  $\geq$  15.0 support AVX-512

```
user@knl% icc -v
icc version 16.0.1 (gcc version 4.8.5 compatibility)
user@knl% icc -help
// ... truncated output ... //
-x<code>
    ...
    MIC-AVX512
    CORE-AVX512
    COMMON-AVX512
```

- ▶ -xMIC-AVX512 : for KNL (supports F, CD, ER, PF)
- ▶ -xCORE-AVX512 : for future Xeon (supports F, CD, DQ, BW, VL)
- ▶ -xCOMMON-AVX512 : common to KNL and Xeon (supports F, CD)

# GCC SUPPORT FOR AVX-512

GCC  $\geq$  4.9.1 supports AVX-512 instruction set.

```
1 for(int i = 0; i < n; i++)  
2   B[i] = A[i] + B[i];
```

Basic automatic vectorization support: add -m flags and -O3:

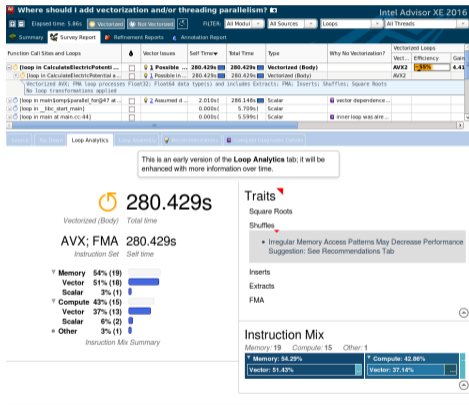
```
user@knl% g++ -v  
gcc version 4.9.2 (GCC)  
user@knl% g++ foo.cc -mavx512f -mavx512er -mavx512cd -mavx512pf -O3
```

Get assembly:

```
user@knl% g++ -s foo.cc -mavx512f -O3  
user@knl% cat foo.s  
...  
vmovapd -16432(%rbp,%rax), %zmm0  
vaddpd -8240(%rbp,%rax), %zmm0, %zmm0  
vmovapd %zmm0, -8240(%rbp,%rax)
```

# PERFORMANCE CONSIDERATIONS

Even if your code is vectorized, tuning may unlock more performance.



download paper to learn more

- ▶ Providing enough parallelism.
  - More consecutive vector operations required to overcome vectorization latency.
- ▶ Loop pipelining and unrolling.
  - Double the pipeline stages to populate.
- ▶ Better vectorization patterns.
  - Avoid long latency operations with unit-stride and unmasked operations.

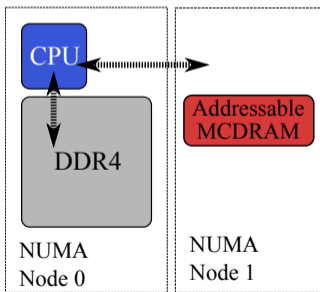


## **USING HIGH-BANDWIDTH MEMORY**

# MODES OF HBM OPERATION

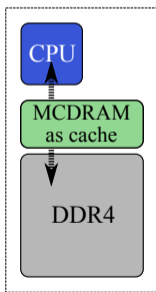
## Flat Mode

- ▶ MCDRAM treated as a NUMA node
- ▶ Users control what goes to MCDRAM



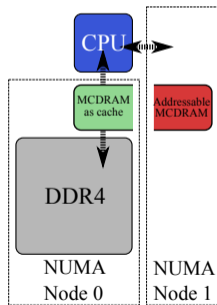
## Cache Mode

- ▶ MCDRAM treated as a Last Level Cache (LLC)
- ▶ MCDRAM is used automatically

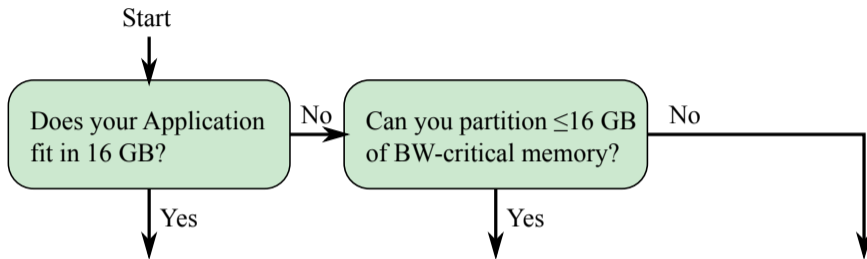


## Hybrid Mode

- ▶ Combination of Flat and Cache
- ▶ Ratio can be chosen in the BIOS



# FLOW CHART FOR BANDWIDTH-BOUND APPLICATIONS



<b>numactl</b>	<b>Memkind</b>	<b>Cache mode</b>
<ul style="list-style-type: none"> <li>▶ Run the whole program in HBM</li> <li>▶ No code modification</li> </ul>	<ul style="list-style-type: none"> <li>▶ Selectively allocate data to HBM</li> <li>▶ Add memkind calls</li> </ul>	<ul style="list-style-type: none"> <li>▶ Allow the chip to figure out how to use HBM</li> <li>▶ No code modification</li> </ul>

# RUNNING APPLICATIONS IN HBM WITH NUMACTL

- ▶ Finding information about the NUMA nodes in the system.

```
user@knl% # In Flat mode of MCDRAM
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 15 16 17 18 19 20 ... 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% gcc myapp.c -o runme -mavx512f -O2
user@knl% numactl --membind 1 ./runme
// ... Application running in HBM ... //
```

# ALLOCATION IN HBM WITH MEMKIND LIBRARY

Manual allocation to HBM possible with hbwmalloc and Memkind Library.

```
1 #include <hbwmalloc.h>
2
3 // Basic allocation in HBM
4 double* A = (double*) hbw_malloc(sizeof(double)*n);
5
6 // Allocation with alignment
7 double* B;
8 int ret = hbw_posix_memalign((void**) &B, 64, sizeof(double)*n);
9
10 hbw_free(A); hbw_free(b); // Special deallocator
```

In Fortran:

```
1 REAL, ALLOCATABLE :: A(:)
2 !DEC$ ATTRIBUTES FASTMEM :: A
3 ALLOCATE (A(1:N))
```

# COMPILATION WITH MEMKIND LIBRARY AND HBWMALLOC

To compile C/C++ applications:

```
user@knl% icpc -lmemkind foo.cc -o runme
user@knl% g++ -lmemkind foo.cc -o runme
```

To compile Fortran applications:

```
user@knl% ifort -lmemkind foo.f90 -o runme
user@knl% gfortran -lmemkind foo.f90 -o runme
```

Open source distribution of Memkind library can be found at:

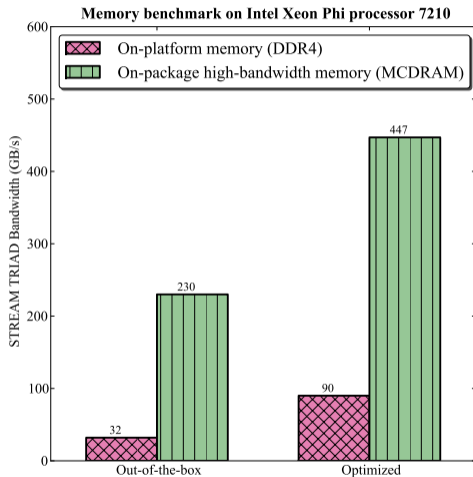
[memkind.github.io/memkind](https://memkind.github.io/memkind)

Learn more:

[colfaxresearch.com/knl-mcdram](https://colfaxresearch.com/knl-mcdram)

# 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/](http://www.cs.virginia.edu/stream/)

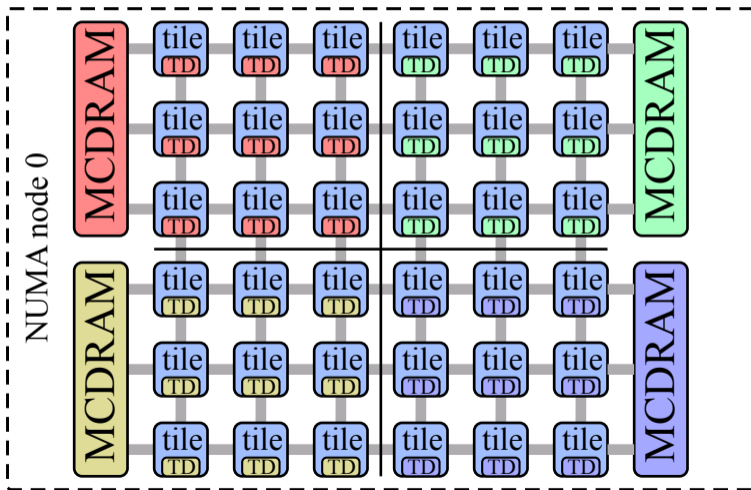




## **LEVERAGING CLUSTERING MODES**

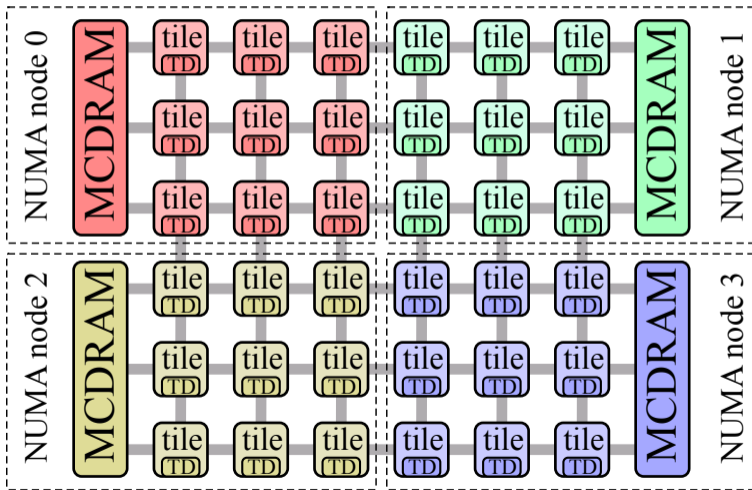
# CLUSTERING MODES: QUADRANT/HEMISPHERE

Tag Directory (TD) and memory reside in the same quadrant.



# CLUSTERING MODES: SNC-4/SNC-2

Cores appear as 4 (or 2) NUMA nodes.

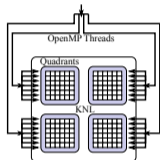


# HOW TO USE CLUSTERING MODES

## Nested OpenMP

```

1  #pragma omp parallel
2  {
3      // ...
4  #pragma omp parallel
5  {
6      // ...
7  }
8  }
```



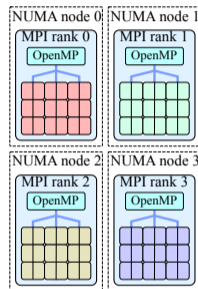
```

user@kn1% OMP_NUM_THREADS=4,72
user@kn1% OMP_NESTED=1
```

## MPI+OpenMP

```

1  stat = MPI_Init();
2  // ...
3  #pragma omp parallel
4  {
5      // ...
6  }
7  // ...
8  MPI_Finalize();
```



```

user@kn1% mpirun -host kn1 \
> -np 4 ./myparallel_app
```

Learn more: [colfaxresearch.com/knl-numa](http://colfaxresearch.com/knl-numa)

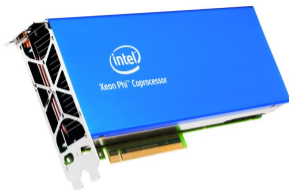


## **COPROCESSOR AND KNL-F**

# FUTURE FORM-FACTORS

## KNLF: KNL with Fabric

- ▶ Fabric integrated on CPU
  - Intel® Omni-Path Architecture
- ▶ Socket mount processor



\*KNC image

## KNL Coprocessor

- ▶ PCIe add-in card
  - Requires host
- ▶ Multiple KNLs in a system



## **§5. INTEL LIBRARIES**



# **INTEL MKL: THE MAGIC PILL**

Intel<sup>®</sup> Math Kernel Library (MKL) — standard mathematical functions optimized for Intel architecture.

## Dense Linear Algebra

BLAS + PBLAS  
LAPACK + ScaLAPACK  
Extended eigensolver

## Fourier Transform

Multi-threaded  
Cluster mode  
1D and multi-dimensional

## Statistics and Probability

Random number generators  
Convolution and correlation  
Summary statistics

## Sparse Linear Algebra

SpBLAS  
Iterative, direct solvers  
Preconditioners

## Numerical Analysis

Partial differential equations  
Nonlinear optimization  
Data fitting. Vector math.

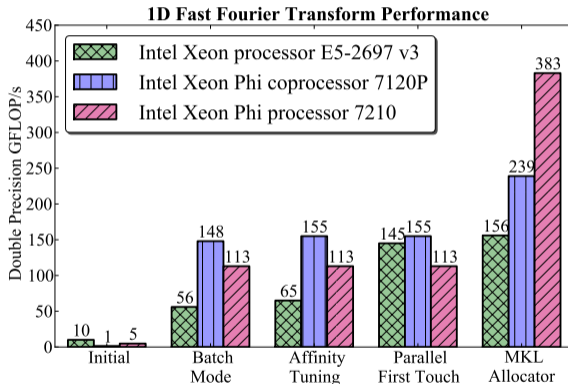
## Machine Learning

New

DNN Primitives:  
Convolution. Inner product.  
ReLU, LRNC, etc.

# ENVIRONMENT TUNING MAY BE NECESSARY

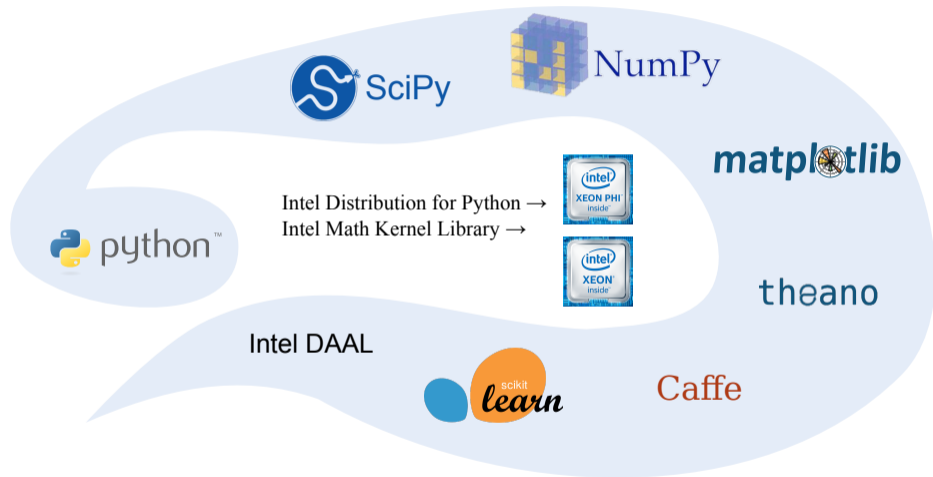
$2 \cdot 10^5$  FFTs of size 2048. See [HOW "KNL" for details](#).





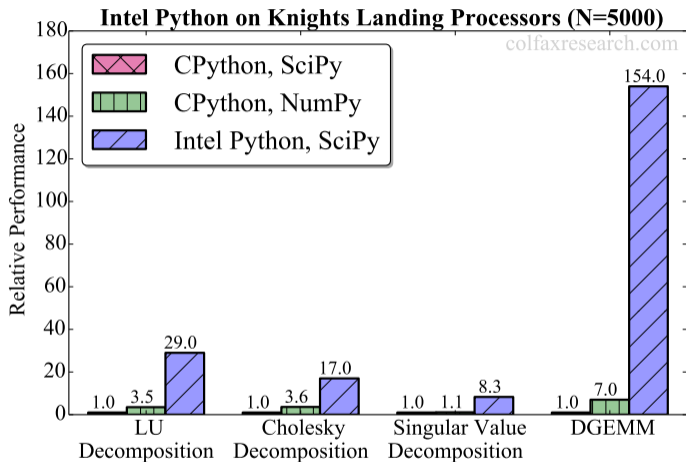
# **INTEL PYTHON: WRAP AROUND ANYTING**

# INTEL DISTRIBUTION FOR PYTHON



Portal: [software.intel.com/intel-distribution-for-python](https://software.intel.com/intel-distribution-for-python). See also: [CR paper](#).

# INTEL PYTHON PERFORMANCE



Portal: [software.intel.com/intel-distribution-for-python](https://software.intel.com/intel-distribution-for-python). See also: [CR paper](#).



# **INTEL DAAL: DATA ANALYTICS**

# DATA ANALYTICS BUILDING BLOCKS

## Computation

### Algorithm

What needs to be computed?

#### Analysis

- QR decomposition
- Correlation
- etc.

#### Training & Prediction

- Regression
- Classification

### Computation Modes

How should the computation be done?

#### Batch mode

#### Online mode

#### Distributed mode

## Data Management

### Loading Data

Where should the data be loaded from?

#### Local buffers (RAM)

#### Non-volatile Sources

- CSV Files
- MySQL
- C Byte-array Files

### Data Structure

How should the data be represented?

#### Homogeneous

- Dense
- Sparse

#### Heterogeneous

- SoA
- AoS

#### Matrices

- Symmetric
- Dense
- Triangular

Portal: [DAAL page](#). See also: [intro article](#), [CR papers](#).

# ALGORITHMS IN DAAL

## Analysis

- Low Order Moments
- Quantile
- Correlation and Variance
- Cosine Distance Matrix
- Correlation Distance Matrix
- K-Means Clustering
- Principal Component Analysis
- Cholesky Decomposition
- Singular Value Decomposition
- QR Decomposition
- Expectation-Maximization
- Multivariate Outlier Detection
- Univariate Outlier Detection
- Association Rules
- Kernel Functions
- Quality Metrics

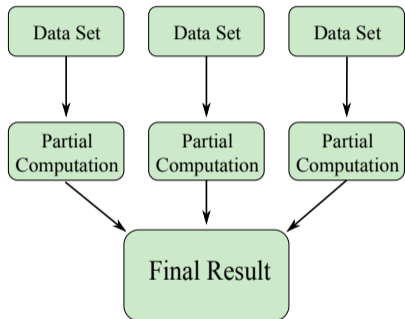
## Training & prediction

- Regression
  - Linear Regression
- Classification
  - Naive Bayes Classifier
  - Boosting
  - Support Vector Machine Classifier
  - Multi-Class Classifier

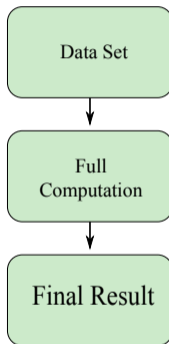
Portal: [DAAL page](#). See also: [intro article](#), [CR papers](#).

# ALGORITHMS IN DAAL

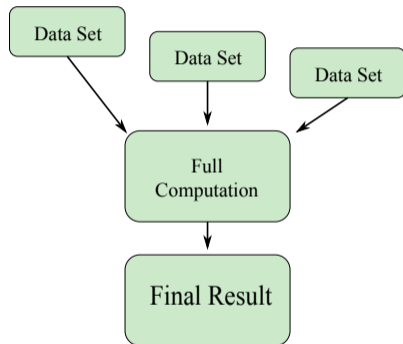
## Distributed Mode



## Batch Mode



## Online Mode

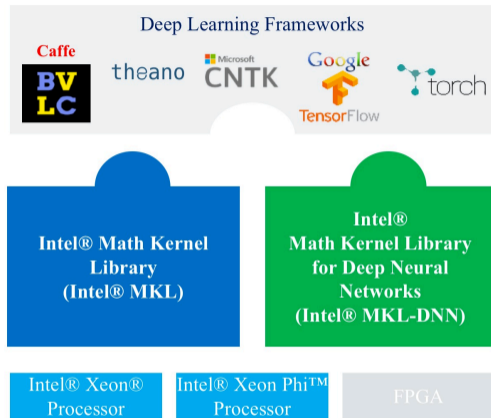


Portal: [DAAL page](#). See also: [intro article](#), [CR papers](#).



**AI ON IA**

# INTEL MKL AND INTEL MKL-DNN



Intel® MKL	Intel® MKL-DNN
DNN primitives + wide variety of other math functions	DNN primitives
C DNN APIs (C++ future)	C/C++ DNN APIs
Binary distribution	Open source DNN code <sup>1</sup>
Free community license. Premium support available as part of Intel® Parallel Studio XE	Apache 2.0 license
Broad usage DNN primitives; not specific to individual frameworks	Multiple variants of DNN primitives as required for framework integrations
Quarterly update releases	Rapid development ahead of Intel MKL releases

<sup>1</sup> GEMM building blocks are binary.

slide credit: Intel corp.

# MACHINE LEARNING ON INTEL ARCHITECTURE: RESOURCES

- ▶ The next wave of deep learning applications ([The Next Platform](#))
- ▶ Code Modernization Speeds Python and Other Machine Learning Packages ([TechEnablement](#))
- ▶ Intel Python ([Intel](#))
- ▶ Intel<sup>®</sup> MKL-DNN ([GitHub](#))
- ▶ Intel Caffe ([GitHub](#))
- ▶ Intel Theano ([GitHub](#))
- ▶ Intel Torch ([GitHub](#))





## **§6. CLOSING WORDS**

## 128-WORD SUMMARY

### Are you Realizing the Payoff of Parallel Processing?

As processor architectures evolve, you get performance boosts in some areas without doing anything with your code. For example, bigger caches, instruction pipelining, smarter branch prediction, and prefetching all improve your performance automatically. However, parallelism is different. You have to make your application explicitly aware of parallelism to reap the benefits of vector instruction support and of multiple cores.

That is what code modernization is about: it is the process of adapting applications to new hardware capabilities, especially parallelism on multiple levels. Once you have a robust version of code, you are future-ready. Just like in the past, when computing applications could ride the wave of increasing clock frequencies, your modernized code will be able to automatically take advantage of the ever-increasing parallelism in future x86-based computing platforms.

[link](#)



THE "HOW" SERIES TRAINING

# DEEP DIVE

WITH CODE MODERNIZATION EXPERTS

It's free

→ [HOWSERIES.COM](https://howseries.com)

\*10x 2-hour sessions | 24-hour 2-weeks remote access to a system