PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

One-Day Workshop

Andrey Vladimirov and Ryo Asai
Colfax International — *colfaxresearch.com*

August 2017

DISCLAIMER

While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.

WHERE TO GET LAB AND SLIDES

Labs:

```
~> git clone https://github.com/ColfaxResearch/HOW-Series-Labs.git
```

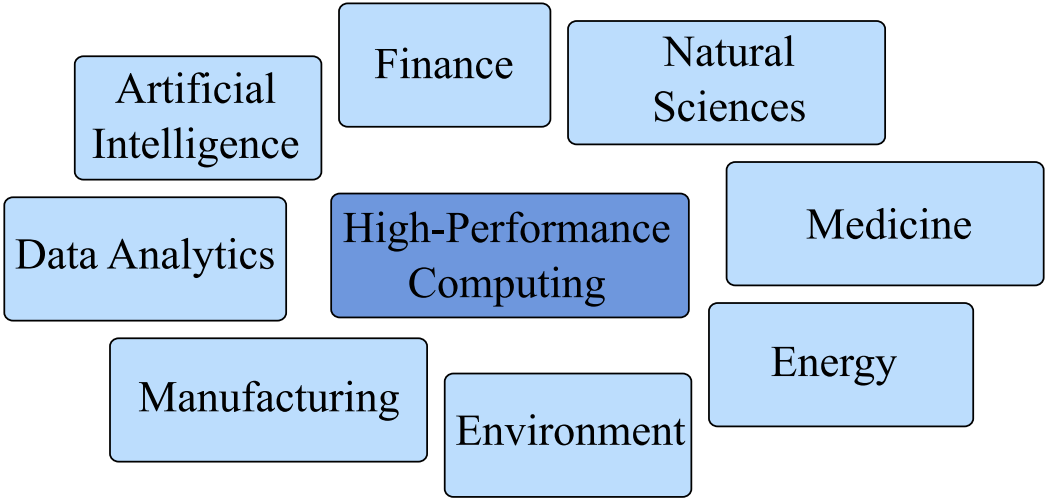
Slides:

https://colfaxresearch.com/training-slides/

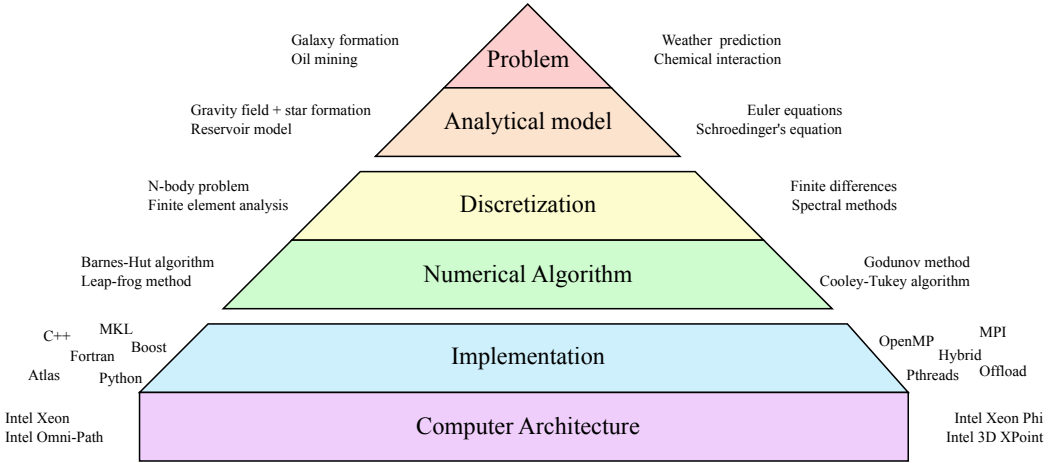
§1. SNEAK PEAK

AGENDA AND WHAT'S IN IT FOR YOU

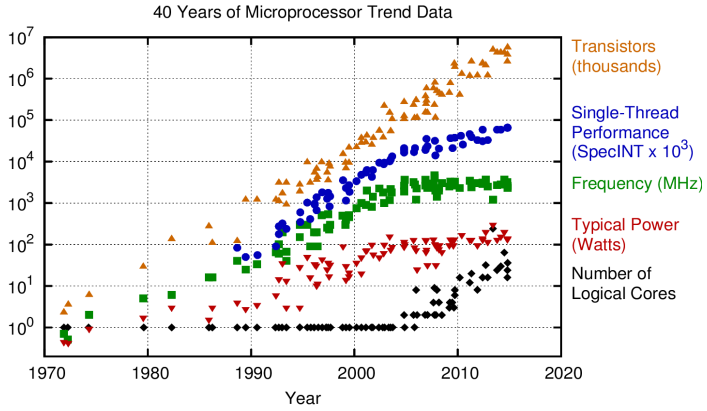
COMPUTING DOMAINS



COMPUTING IN SCIENCE AND ENGINEERING



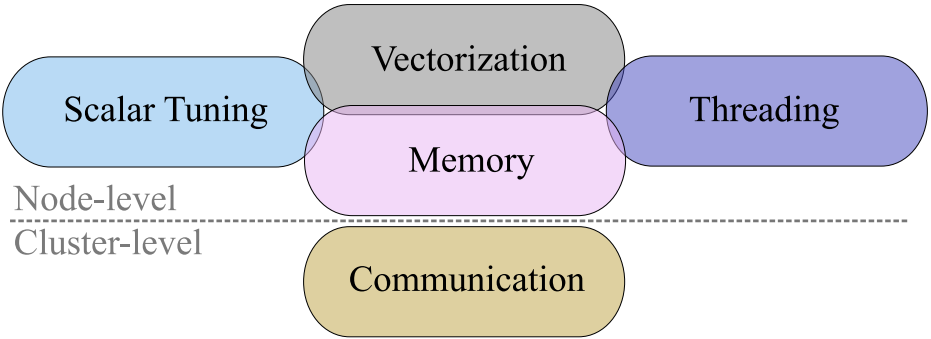
40 YEARS OF MICROPROCESSOR DATA



Source: karlrupp.net

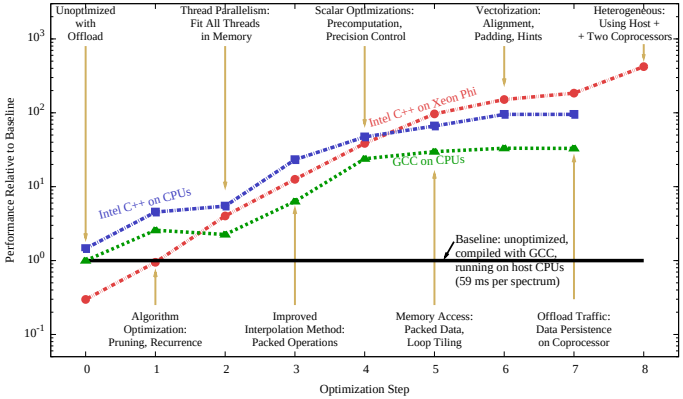
Code Modernization

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



PROGRAMMING MODEL CONTINUITY

Common story for many applications:



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

§2. PROGRAMMING, OPTIMIZATION BY EXAMPLE

DIRECT N-BODY SIMULATION

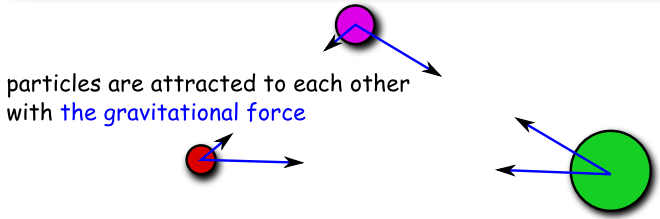
Gravitational N-body dynamics:

Newton's law of universal gravitation:

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

where:

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



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:

```
struct ParticleType {
  float x, y, z;
  float vx, vy, vz;
};
```

`main()` allocates an array of `ParticleType`:

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

Particle propagation step is timed:

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

PARTICLE UPDATE ENGINE

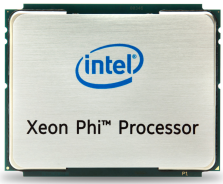
```
1  void MoveParticles(int nParticles, ParticleType* particle, float dt) {
2    for (int i = 0; i < nParticles; i++) {   // Particles that experience force
3      float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
4      for (int j = 0; j < nParticles; j++) { // Particles that exert force
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   }
```

INTEL ARCHITECTURE

COMPUTING PLATFORMS

Highly parallel, for general-purpose computing

Multi-Core Architecture



Highly parallel, specialized for high-performance computing

Intel Many Integrated Core (MIC) Architecture

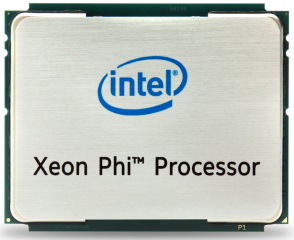
BIRD'S EYE VIEW

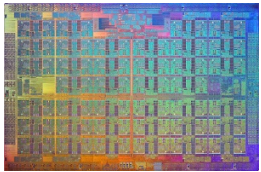
- C/C++/Fortran
- Linux/Windows
- ≤3 TiB DDR4
- ≤44 cores (2-way)
- ≈3 GHz
- 2 HT/core
- AVX2/AVX-512

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

INTEL XEON PHI PROCESSORS (2ND GEN)

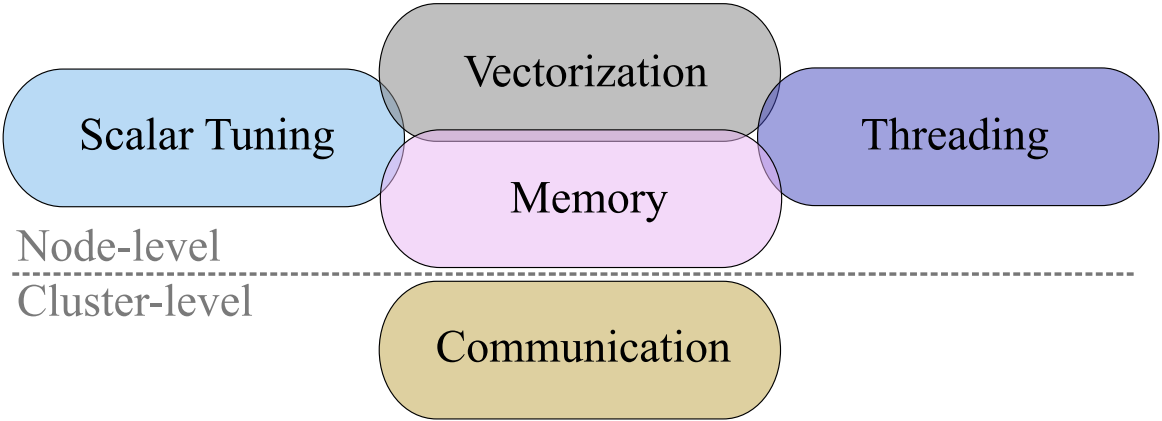
- Specialized for floating-point computing
- Highly-parallel (72 cores*)
- Balanced for compute
- Less forgiving than Xeon
- Theor. ~ 3.0 TFLOP/s in DP*
- Meas. ~ 490 GB/s bandwidth*





* Intel Xeon Phi processor, Knighs Landing architecture (2016), top-of-the-line (e.g., 7290P)

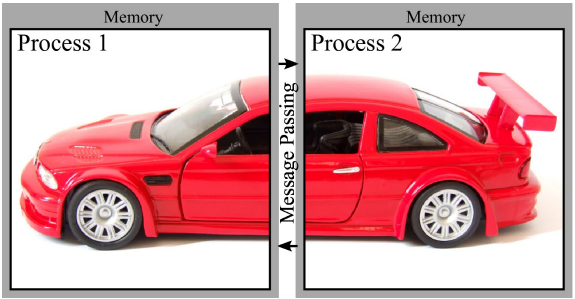
OPTIMIZATION AREAS

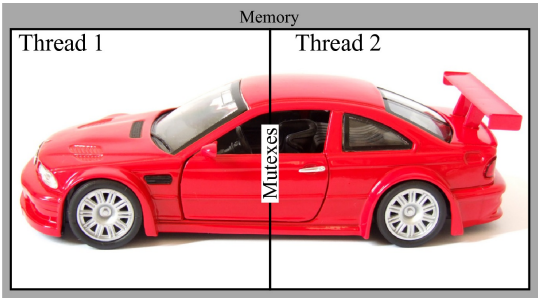


TASK PARALLELISM

THREADS VERSUS PROCESSES

Option 1: Partitioning data set between threads/processes

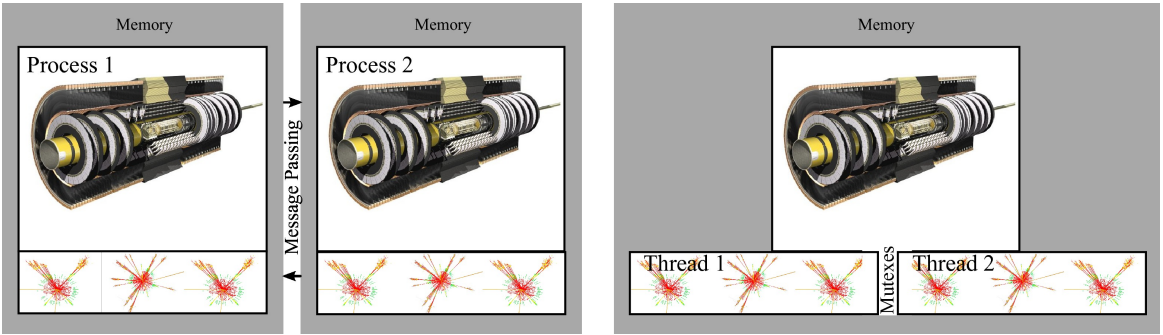




Examples: computational fluid dynamics (CFD), image processing.

THREADS VERSUS PROCESSES

Option 2: Sharing data set between threads/processes



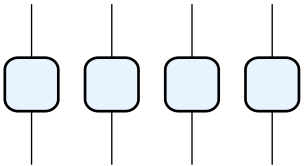
Examples: particle transport simulation, machine learning (inference).

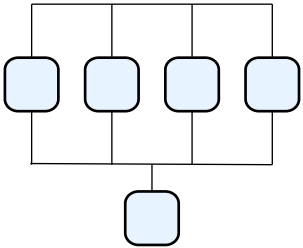
THREADING FRAMEWORKS

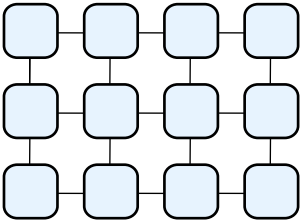
Framework	Functionality
C++11 Threads	Asynchronous functions; only C++
POSIX Threads	Fork/join; C/C++/Fortran; Linux
Cilk Plus	Async tasks, loops, reducers, load balance; C/C++
TBB	Trees of tasks, complex patterns; only C++
OpenMP	Tasks, loops, reduction, load balancing, affinity, nesting, C/C++/Fortran (+SIMD, offload)

PARALLEL PATTERNS

Common parallel patterns call for collective communication







"HELLO WORLD" OPENMP PROGRAM

```
#include <omp.h>
#include <cstdio>

int main(){
  // This code is executed by 1 thread
  const int nt=omp_get_max_threads();
  printf("OpenMP with %d threads\n", nt);

#pragma omp parallel
  { // This code is executed in parallel
    // by multiple threads
    printf("Hello World from thread %d\n",
                    omp_get_thread_num());
  }
}
```

- 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

COMPILING THE "HELLO WORLD" OPENMP PROGRAM

```
vega@lyra% icpc -qopenmp hello_omp.cc
vega@lyra% export OMP_NUM_THREADS=5
vega@lyra% ./a.out
OpenMP with 5 threads
Hello World from thread 0
Hello World from thread 3
Hello World from thread 1
Hello World from thread 2
Hello World from thread 4
```

`OMP_NUM_THREADS` controls number of OpenMP threads (default: logical CPU count)

CONTROL OF VARIABLE SHARING

Method 1: using clauses in pragma omp parallel (C, C++, Fortran):

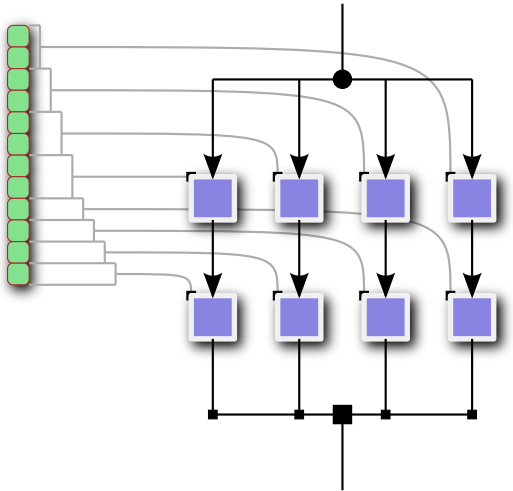
```
1 int A, B; // Variables declared at the beginning of a function
2 #pragma omp parallel private(A) shared(B)
3   {
4     // Each thread has its own copy of A, but B is shared
5   }
```

Method 2: using scoping (only C and C++):

```
1 int B; // Variable declared outside of parallel scope - shared by default
2 #pragma omp parallel
3   {
4     int A; // Variable declared inside the parallel scope - always private
5     // Each thread has its own copy of A, but B is shared
6   }
```

LOOP-CENTRIC PARALLELISM: FOR-LOOPS IN OPENMP

- Simultaneously launch multiple threads
- Scheduler assigns loop iterations to threads
- Each thread processes one iteration at a time



Parallelizing a `for`-loop.

LOOP-CENTRIC PARALLELISM: FOR-LOOPS IN OPENMP

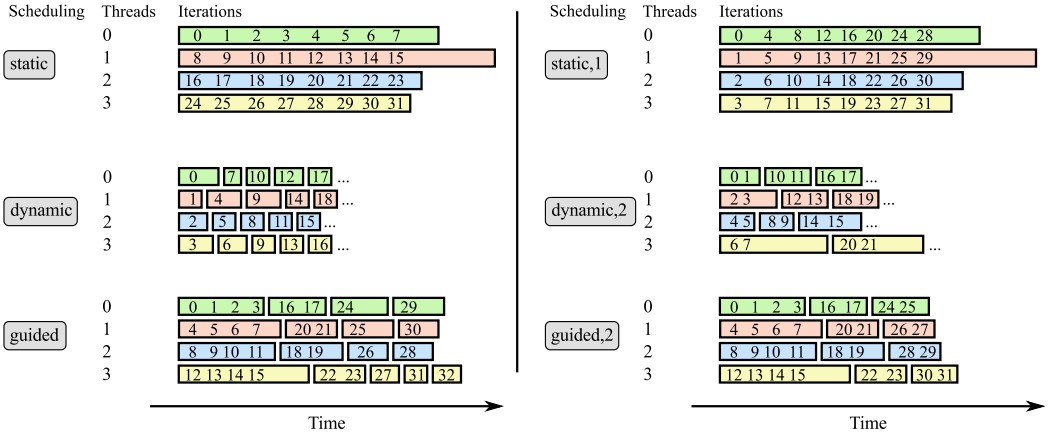
The OpenMP library will distribute the iterations of the loop following the `#pragma omp parallel for` across threads.

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++) {
3   printf("Iteration %d is processed by thread %d\n",
4          i, omp_get_thread_num());
5   // ... iterations will be distributed across available threads...
6 }
```

LOOP-CENTRIC PARALLELISM: FOR-LOOPS IN OPENMP

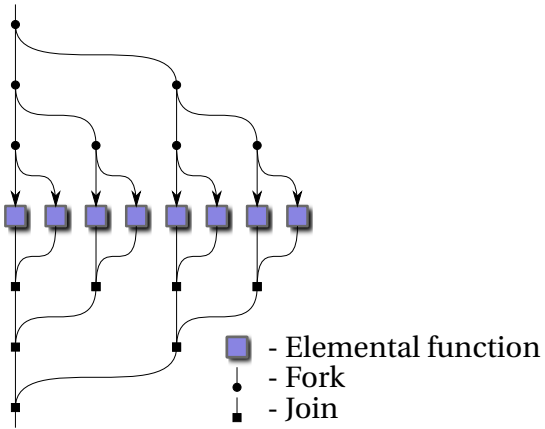
```
1  #pragma omp parallel
2  {
3    // Code placed here will be executed by all threads.
4
5    // Alternative way to specify private variables:
6    // declare them in the scope of pragma omp parallel
7    int private_number=0;
8
9  #pragma omp for
10   for (int i = 0; i < n; i++) {
11     // ... iterations will be distributed across available threads...
12   }
13   // ... code placed here will be executed by all threads
14 }
```

LOOP SCHEDULING MODES IN OPENMP



FORK-JOIN MODEL OF PARALLEL EXECUTION

- Each thread can spawn daughter threads
- Available threads pick up queued tasks
- Expresses algorithms that cannot be expressed in the loop model (e.g., parallel recursion)



Fork-join model of parallel execution.

(`#pragma omp task` functionality)

TASKS IN OPENMP: EXAMPLE

```
// Starting the first task:
#pragma omp parallel
{ // Enter a parallel region
#pragma omp single
  { // Start the first task
    // from only one thread
    RecursiveWorkload(args);
  }
}
```

```
// Recursive task spawning:
void RecursiveWorkload(Arg* args) {
  if (args->size > threshold) {
    // Split work
    Arg* args1=args->FirstHalf();
    Arg* args2=args->SecondHalf();

    // Parallel divide-and-conquer
#pragma omp task firstprivate(args1)
    { RecursiveWorkload(args1); }
#pragma omp task firstprivate(args2)
    { RecursiveWorkload(args2);  }
  } else {
    // End of recursion
    args->ProcessSmallestSubTask();
  }
}
```

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         ...
```

SCALAR OPTIMIZATION

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

PRECISION CONTROL FOR TRANSCENDENTAL FUNCTIONS

`-fimf-precision=value[:funclist]` Defines the precision for math functions. `value` is one of: `high`, `medium` or `low`

`-fimf-max-error=ulps[:funclist]` The maximum allowable error expressed in ulps (*units in last place*)

`-fimf-accuracy-bits=n[:funclist]` The number of correct bits required for mathematical function accuracy.

`-fimf-domain-exclusion=n[:funclist]` Defines a list of special-value numbers that do not need to be handled.
`int n` derived by the bitwise OR of types:
extremes: 1, NaNs: 2, infinites: 4, denormals[1]: 8, zeroes: 16.

[1] by default, on Intel Xeon Phi, denormals are flushed to zero in hardware, but supported in SVML

FLOATING-POINT SEMANTICS

The Intel C++ Compiler may represent floating-point expressions in executable code differently, depending on the *floating-point semantics*.

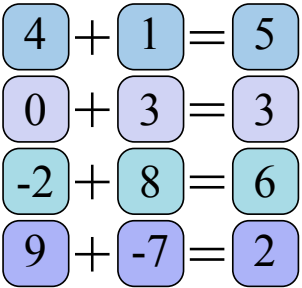
`-fp-model strict`	Only value-safe optimizations
`-fp-model precise`	calculations are reproducible from run to run exceptions controlled using `-fp-model except`
`-fp-model fast=1`	(default) Value-unsafe optimizations are allowed
`-fp-model fast=2`	better performance at the cost of lower accuracy
`-fp-model source`	Intermediate arithmetic results are rounded to the precision defined in the source code.
`-fp-model double`	Intermediate arithmetic results are rounded to 53-bit (double) precision.
`-fp-model extended`	Intermediate arithmetic results are rounded to 64-bit (extended) precision.
`-fp-model [no-]except`	controls floating-point exception semantics.

DATA PARALLELISM

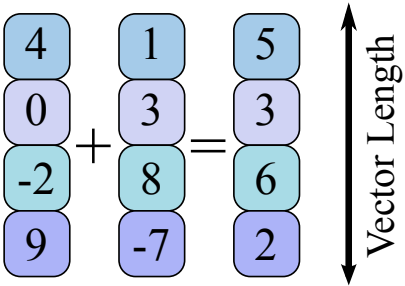
SHORT VECTOR SUPPORT

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

Scalar Instructions



Vector Instructions



SIMULTANEOUS THREADING AND VECTORIZATION

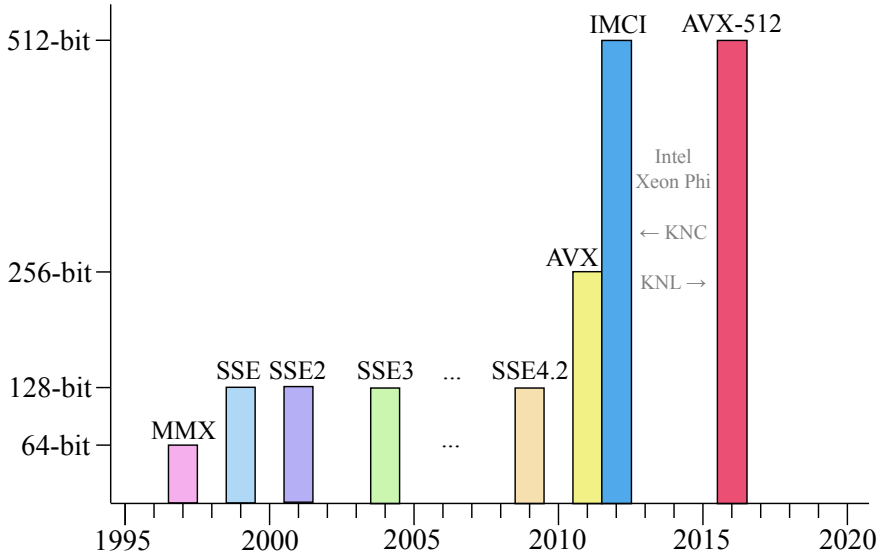
This approach often works:

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

That works as well:

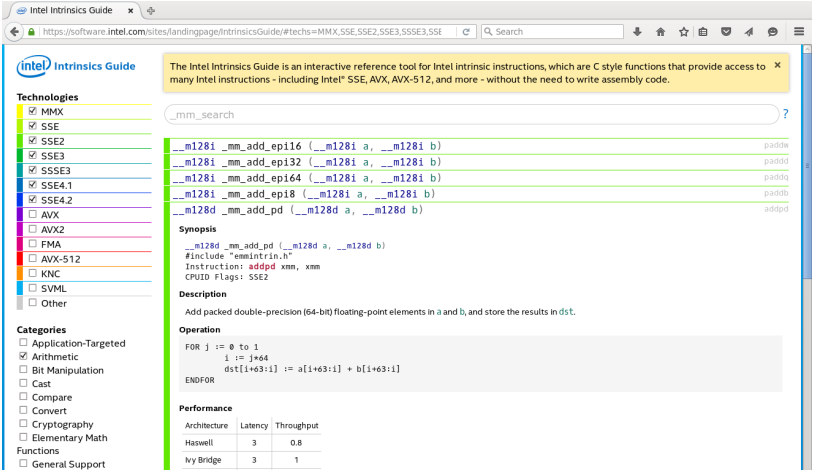
```
#pragma omp parallel for simd
for (int i = 0; i < n; i++) // If the problem is all data-parallel
    DoSomeWork(A[i]);
```

INSTRUCTION SETS IN INTEL ARCHITECTURE



INTEL INTRINSICS GUIDE

https://software.intel.com/sites/landingpage/IntrinsicsGuide



DETECTING AVAILABLE INSTRUCTIONS

In the OS:

```
[student@cdt ~]% cat /proc/cpuinfo
...
fpu_exception   : yes
cpuid level     : 11
wp              : yes
flags           : fpu vme de pse tsc msr pae mce
cx8 apic mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ss ht syscall nx lm constant_tsc
unfair_spinlock pni ssse3 cx16 sse4_1 sse4_2
x2apic popcnt aes hypervisor lahf_lm fsgsbase
bogomips        : 5985.17
clflush size    : 64
cache_alignment: 64
address sizes   : 46 bits physical, 48 bits virtual
...
```

In code (see also):

```
1  // Intel compiler
2  // preprocessor macros:
3
4  #ifdef __SSE__
5  // ...SSE code path
6  #endif
7
8  #ifdef __SSE4_2__
9  // ...SSE code path
10 #endif
11
12 #ifdef __AVX__
13 // ...AVX code path
14 #endif
```

AUTOMATIC VECTORIZATION OF LOOPS

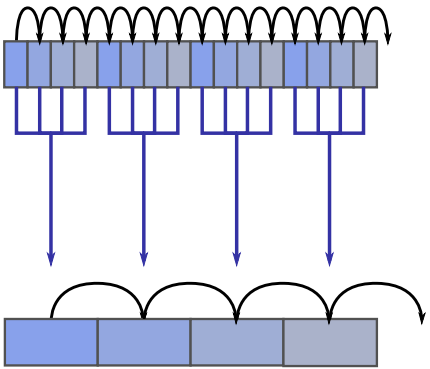
```
1  #include <cstdio>
2
3  int main(){
4    const int n=1024;
5    int A[n] __attribute__((aligned(64)));
6    int B[n] __attribute__((aligned(64)));
7
8    for (int i = 0; i < n; i++)
9      A[i] = B[i] = i;
10
11   // This loop will be auto-vectorized
12   for (int i = 0; i < n; i++)
13     A[i] = A[i] + B[i];
14
15   for (int i = 0; i < n; i++)
16     printf("%2d %2d %2d\n",i,A[i],B[i]);
17 }
```

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

LIMITATIONS ON AUTOMATIC VECTORIZATION

- Innermost loops*
- Known number of iterations
- No vector dependence
- Functions must be SIMD-enabled

* `#pragma omp simd` to override



VECTORIZE MORE LOOPS: `#pragma omp simd`

Used to “enforce vectorization of loops”, which includes:

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

See OpenMP reference for syntax; `#pragma simd`

VECTORIZING WITH UNIT-STRIDE MEMORY ACCESS

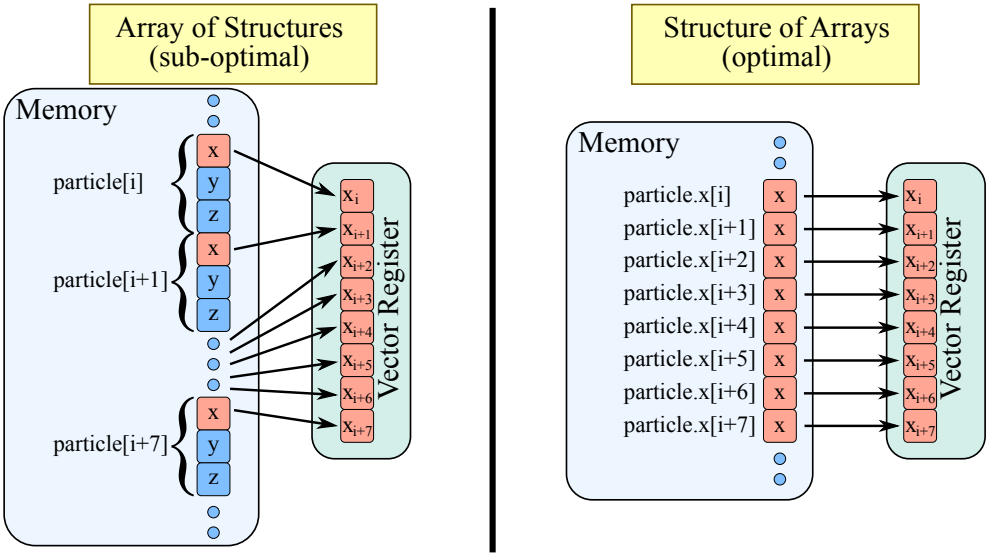
Before:

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

After:

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

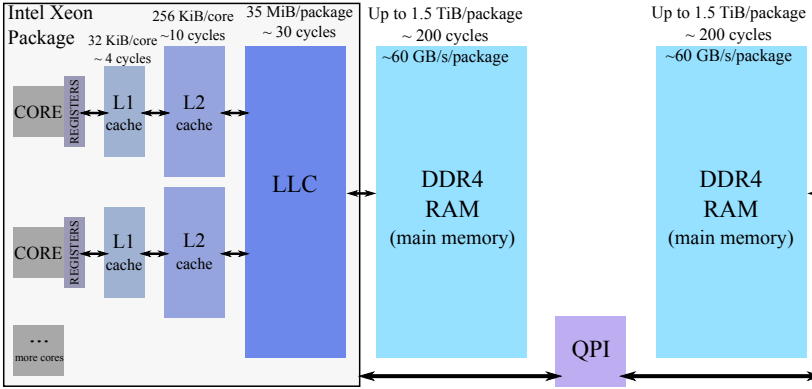
WHY AOS TO SOA CONVERSION HELPS: UNIT STRIDE



MEMORY OPTIMIZATION

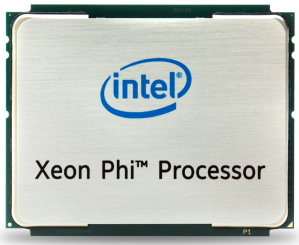
INTEL XEON CPU: MEMORY ORGANIZATION

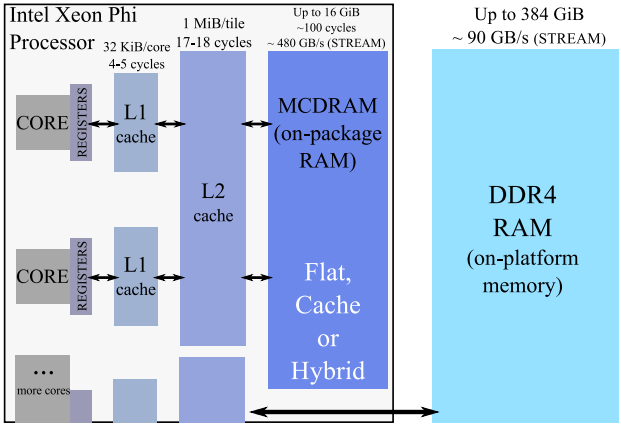
- Hierarchical cache structure
- Two-way processors have NUMA architecture



KNL MEMORY ORGANIZATION (BOOTABLE)

- On-package high-bandwidth memory (HBM) – MCDRAM
- Optimized for arithmetic performance and bandwidth (not latency)

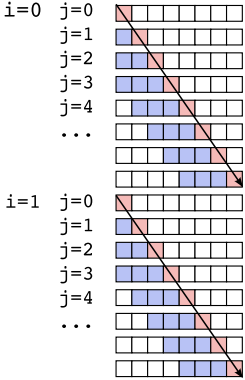




LOOP TILING: CACHE BLOCKING

Original:

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



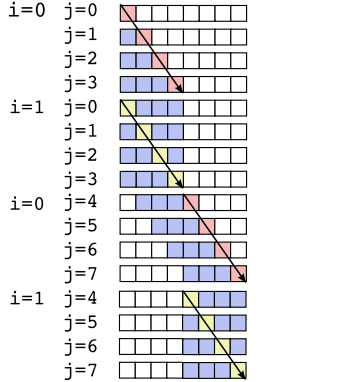
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

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

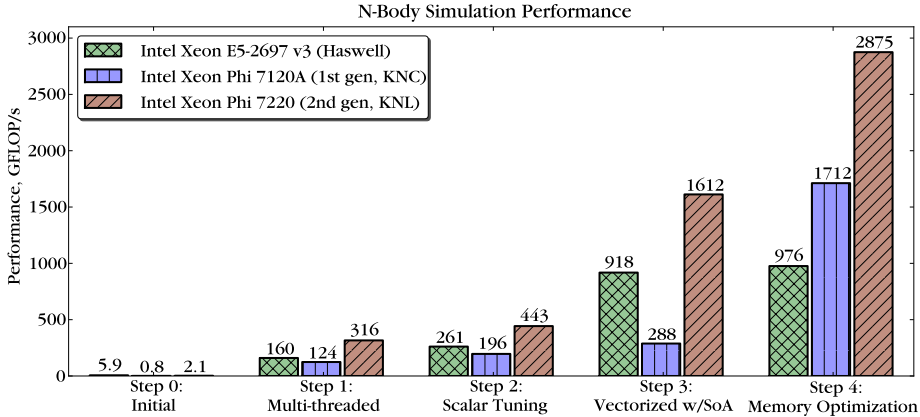
Before:

```
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)

§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`

SETTING OPTIMIZATION LEVEL

For the entire file:

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

For a specific function:

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

STRENGTH REDUCTION

Common Subexpression Elimination.

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

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

Replace division with multiplication.

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

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

Use functions with Hardware support.

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

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

CONSISTENCY OF PRECISION: CONSTANTS

```
// Bad: 2 is "int"
long b=a*2;

// Bad: overflow
long n=100000*100000;

// Bad: excessive
float p=6.283185307179586;

// Bad: 2 is "int"
float q=2*p;

// Bad: 1e9 is "double"
float r=1e9*p;

// Bad: 1 is "int"
double t=s+1;
```

```
// Good: 2L is "long"
long b=a*2L;

// Good: correct
long n=100000L*100000L;

// Good: accurate
float p=6.283185f;

// Good: 2.0f is "float"
float q=2.0f*p;

// Good: 1e9f is "float"
float r=1e9f*p;

// Good: 1.0 is "double"
double t=s+1.0;
```

CONSISTENCY OF PRECISION: FUNCTIONS

```
// Bad: 3.14 is a double
float x = 3.14;

// Bad: sin() is a
// double precision function
float s = sin(x);

// Bad: round() takes double
// and returns double
long v = round(x);

// Bad: abs() is not from IML
// it takes int and returns int
int v = abs(x);
```

```
// Good: 3.14f is a float
float x = 3.14f;

// Good: sin() is a
// single precision function
float s = sinf(x);

// Good: lroundf() takes float
// and returns long
long v = lroundf(x);

// Good: fabsf() is from IML
// It takes and returns a float
float v = fabsf(x);
```

MOVE BRANCHES OUTSIDE OF LOOPS

```
// Elegant, but bad for performance
for (i = 0; i < n; i++) {
  if (i == 0) {
    // Absorbing boundary
    B[i] = 0.0;
  } else if (i == n - 1) {
    // Injection at boundary
    B[i] = A[i] + 1.0;
  } else {
    // Diffusion between boundaries
    B[i] = 0.25*(A[i-1] +
                 2.0*A[i] + A[i+1]);
  }
}
```

```
// Moving branches out of loops



// Absorbing boundary
B[i] = 0.0;

for (i = 1; i < n - 1; i++) {
  // Diffusion between boundaries
  B[i] = 0.25*(A[i-1] + 2.0*A[i] +
                              A[i+1]);
}

// Injection at boundary
B[n-1] = A[n-1] + 1.0;
```

REDUNDANT CODE IS OK

```
// Elegant, but bad for performance
for (ii = 0; ii < n; ii+=16) {
  for (i = ii; i < ii+16; i++)
    // Branch causes unnecessary
    // masking of vector iterations
    if (i < n) {
      A[k*n + i] = ...
    }
}
```

```
// Redundant code, but faster
const int nTrunc = n - n%16;
for (ii = 0; ii < nTrunc; ii+=16) {
  for (i = ii; i < ii+16; i++)
      A[k*n + i] = ...

for (i = nTrunc; i < n; i++)
  A[k*n + i] = ...
}
```

VECTORIZATION

AUTOMATIC VECTORIZATION OF LOOPS

```
1  #include <cstdio>
2
3  int main(){
4    const int n=1024;
5    int A[n] __attribute__((aligned(64)));
6    int B[n] __attribute__((aligned(64)));
7
8    for (int i = 0; i < n; i++)
9      A[i] = B[i] = i;
10
11   // This loop will be auto-vectorized
12   for (int i = 0; i < n; i++)
13     A[i] = A[i] + B[i];
14
15   for (int i = 0; i < n; i++)
16     printf("%2d %2d %2d\n",i,A[i],B[i]);
17 }
```

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

VECTORIZE MORE LOOPS: `#pragma omp simd`

Used to “enforce vectorization of loops”, which includes:

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

See OpenMP reference for syntax; `#pragma simd`

SIMD-ENABLED FUNCTIONS

Define function in one file (e.g., library), use in another

```
// Compiler will produce 3 versions:
#pragma omp declare simd
float my_simple_add(float x1, float x2){
  return x1 + x2;
}
```

```
// May be in a separate file
#pragma omp simd
for (int i = 0; i < N, ++i) {
  output[i] = my_simple_add(inputa[i], inputb[i]);
}
```

SIMD-ENABLED FUNCTIONS

The solution is to design and declare the function as *SIMD-enabled*:

```
__attribute__((vector)) float my_simple_add(float x1, float x2) {
  return x1 + x2;
}
```

When using SIMD-enabled functions, use `#pragma simd`.

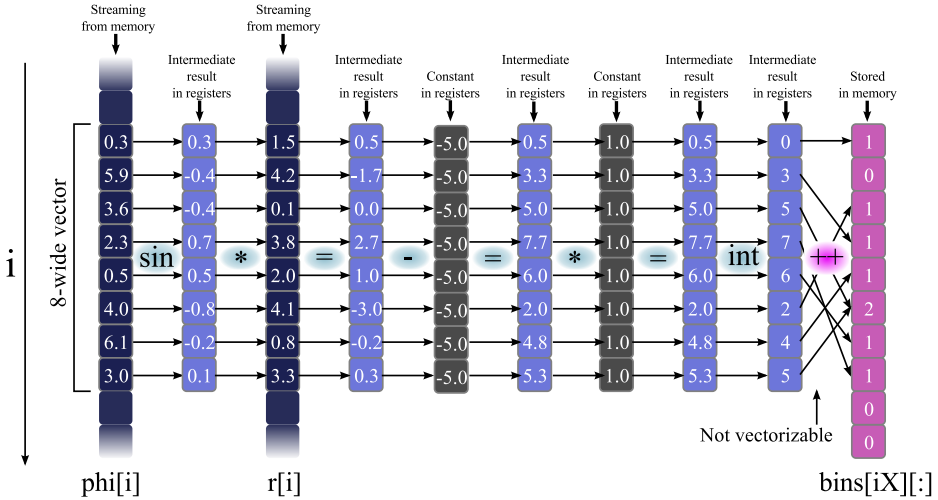
```
// ...in a separate source file:
#pragma simd
for (int i = 0; i < N, ++i) {
  output[i] = my_simple_add(inputa[i], inputb[i]);
}
```

In this case, automatic vectorization succeeds.

AUTO-VECTORIZED LOOPS MAY BE COMPLEX

```
1  for (int i = ii; i < ii + tileSize; i++) { // Auto-vectorized
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] -> 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 }
```

AUTO-VECTORIZED LOOPS MAY BE COMPLEX (EXAMPLE 2)



See this paper for more details

ASSUMED VECTOR DEPENDENCE

- True vector dependence – 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* – compiler suspects dependence

```
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
vega@lyra% cat vdep.optrpt
...
remark #15304: loop was not
vectorized: non-vectorizable loop
instance from multiversioning
...
```

RESOLVING ASSUMED DEPENDENCY

- Restrict: Keyword indicating that there is no pointer aliasing (C++11)

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

```
vega@lyra% icpc -c vdep.cc -qopt-report \
%  -restrict
vega@lyra% cat vdep.optrpt
...
remark #15304: LOOP WAS VECTORIZED
...
```

- `#pragma ivdep`: ignores assumed dependency for a loop (Intel Compiler)

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

UNIT-STRIDE ACCESS

Unit-stride access is optimal:

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

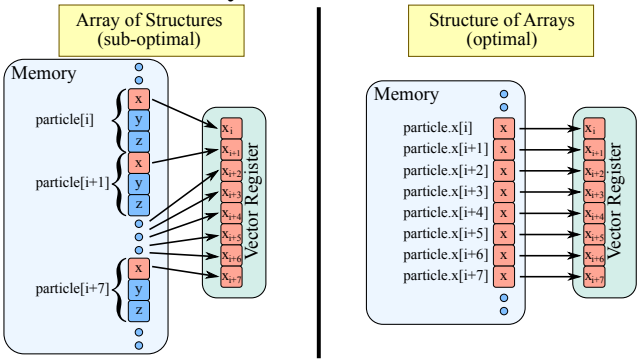
Non-unit stride is slower:

```
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 `((size_t)p%n==0)`.

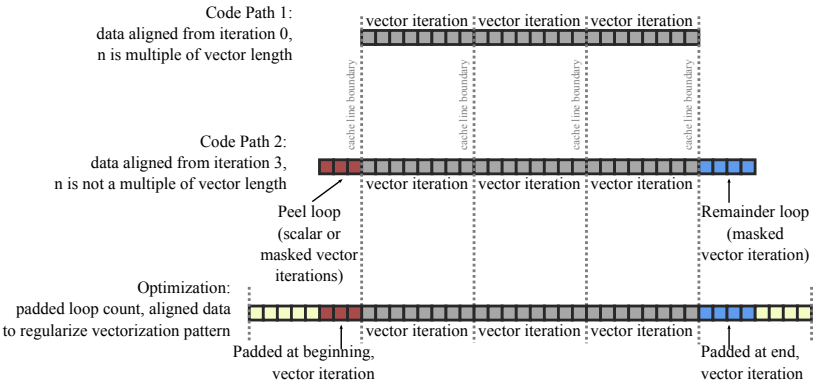
Processor	Operation	Alignment
Xeon (Westmere and earlier)	SSE load, store	16-byte
Xeon (Sandy Bridge and later)	AVX load, store	32-byte (relaxed)
Xeon Phi (1st gen)	IMCI load, store	64-byte (strict)
Xeon Phi (1st gen)	DMA transfer in offload	4096-byte (preferred)
Xeon Phi (2nd gen)	AVX-512 load, store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

Why align: speed up vector load/stores, avoid false sharing, 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] = ...
```

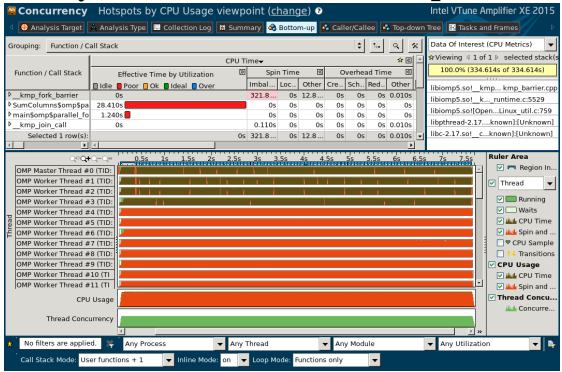
VECTORIZATION DIRECTIVES

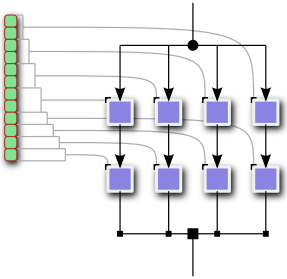
- `#pragma omp simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `__assume_aligned` keyword
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `restrict` qualifier and `-restrict` command-line argument
- `#pragma loop count`

MULTI-THREADING

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

$$S_i = \sum_{j=0}^{n} M_{ij},\ i = 0 \ldots m. \tag{1}$$

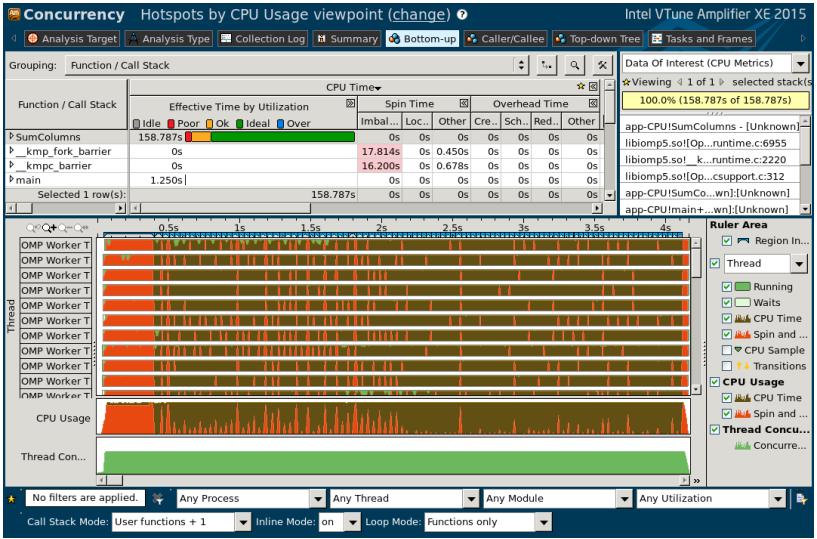
- `m`=4 is small, smaller than the number of threads in the system
- `n` $\approx 10^8$ is large enough so that matrix does not fit into cache

```
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; }}
```

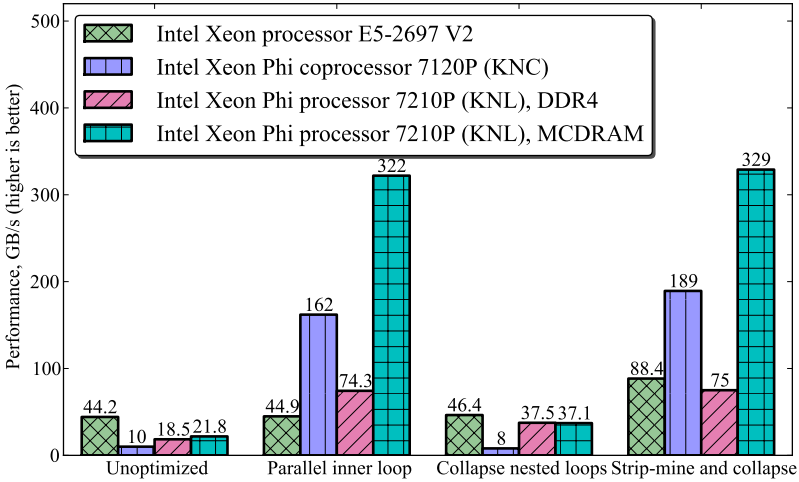
EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE

```
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



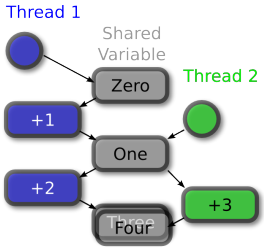
DEALING WITH INSUFFICIENT PARALLELISM



RACE CONDITIONS AND UNPREDICTABLE PROGRAM BEHAVIOR

```
1  #include <omp.h>
2  #include <cstdio>
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)
```



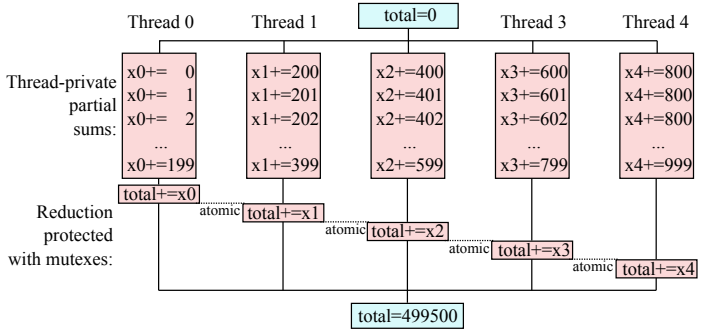
Race Condition!

- 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:

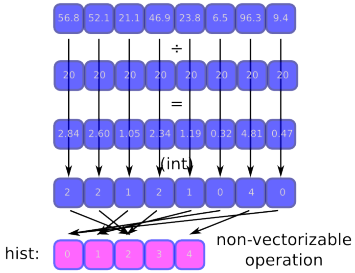
```
1  int total = 0;
2  #pragma omp parallel
3  {
4    int total_thr = 0;
5  #pragma omp for
6    for (int i=0; i<n; i++)
7      total_thr += i;
8
9  #pragma omp atomic
10   total += total_thr;
11
12 }
```



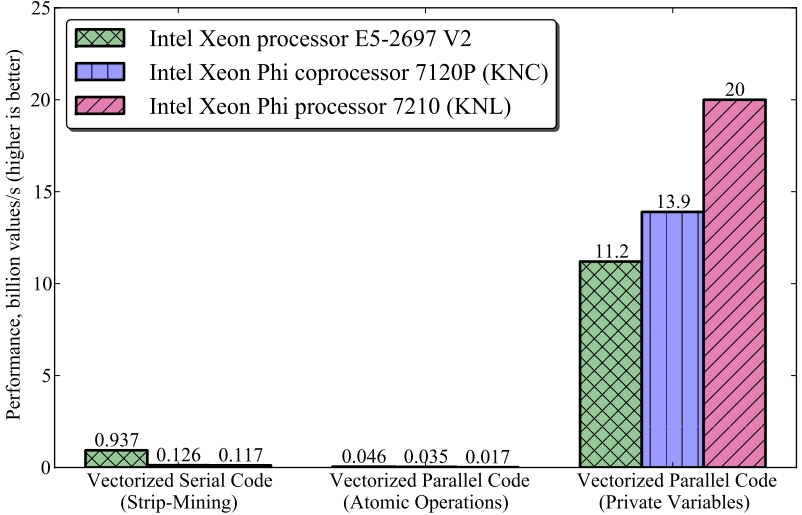
EXAMPLE: BINNING PROBLEM

```
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 grpWidth) {
11     for (int i = 0; i < n; i++) {
12       const int j = int(age[i]/grpWidth);
13       hist[j]++;
14     }
15 }
```

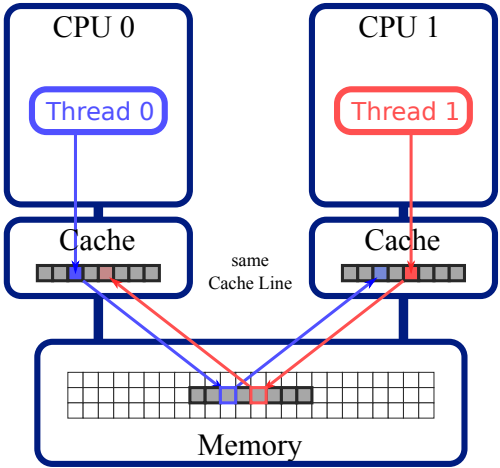
- Vector dependence in `hist[j]++`
- Strip-mine or use conflict detection



USING REDUCTION INSTEAD OF SYNCHRONIZATION

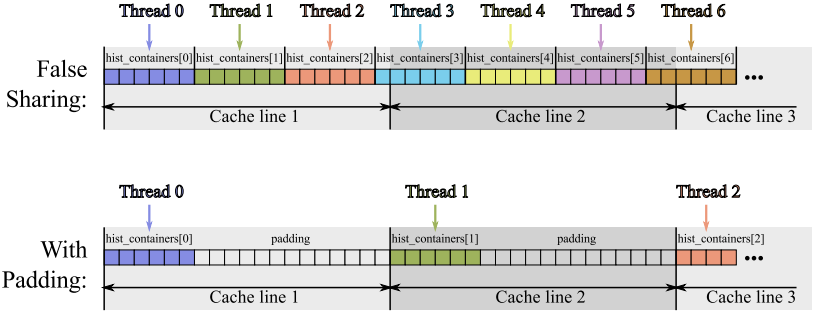


FALSE SHARING. DATA PADDING AND PRIVATE VARIABLES



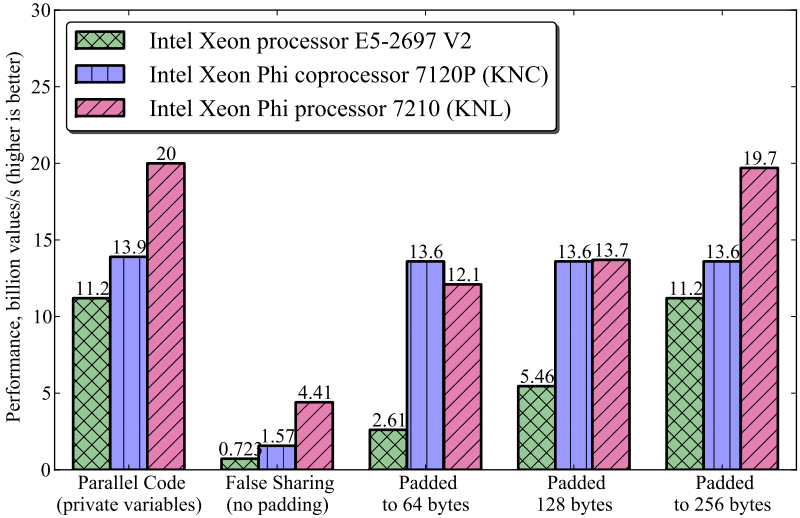
- Occurs when 2 or more threads acess the same cache line, and at least one of the accesses is for writing
- Caused by *coherent caches*
- Cache line is 64-byte wide (in modern Intel architectures)

PADDING TO AVOID FALSE SHARING

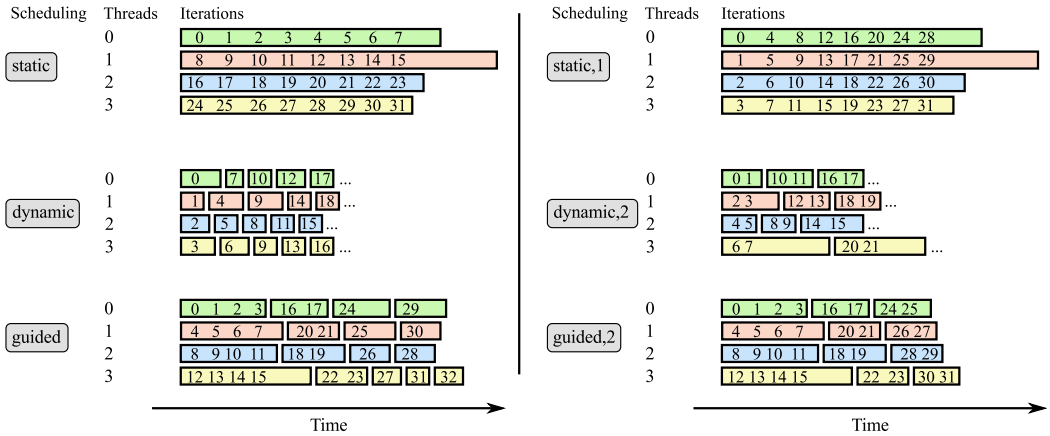



```
1 // Padding to avoid sharing a cache line between threads
2 const int paddingBytes = 64;
3 const int paddingElements = paddingBytes / sizeof(int);
4 const int mPadded = m + (paddingElements-m%paddingElements);
5 int hist_containers[nThreads][mPadded]; // New container
```

PADDING TO AVOID FALSE SHARING

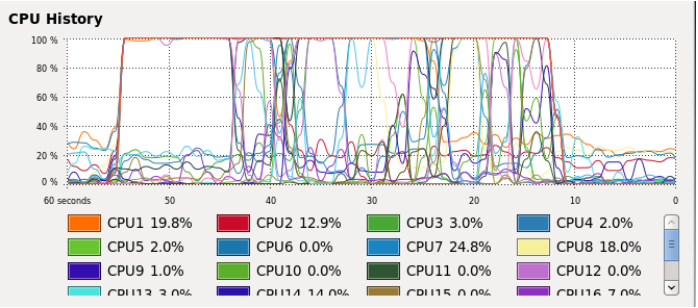


LOOP SCHEDULING MODES IN OPENMP

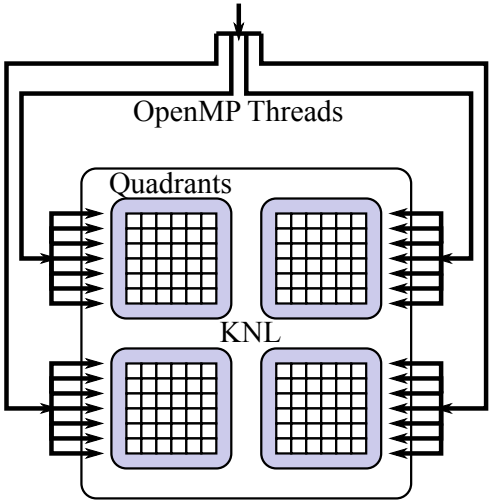


WHAT IS THREAD AFFINITY

- 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?

Intel Xeon Phi processor 7250

$$68 \text{ cores} \times 1.2 \text{ GHz} \times 8 \text{ vec.lanes} \times 2 \text{ FMA} \times 2 \text{ IPC} \approx 2.6 \text{ TFLOP/s}$$

$$2.6 \text{ TFLOP/s} \times 8 \text{ bytes} \approx 21 \text{ TB/s}$$

$$\text{MCDRAM bandwidth} \approx 0.48 \text{ TB/s}$$

$$\text{Ratio} = 21/0.48 \approx 43 \text{ (FLOPs)/(Memory Access)}$$

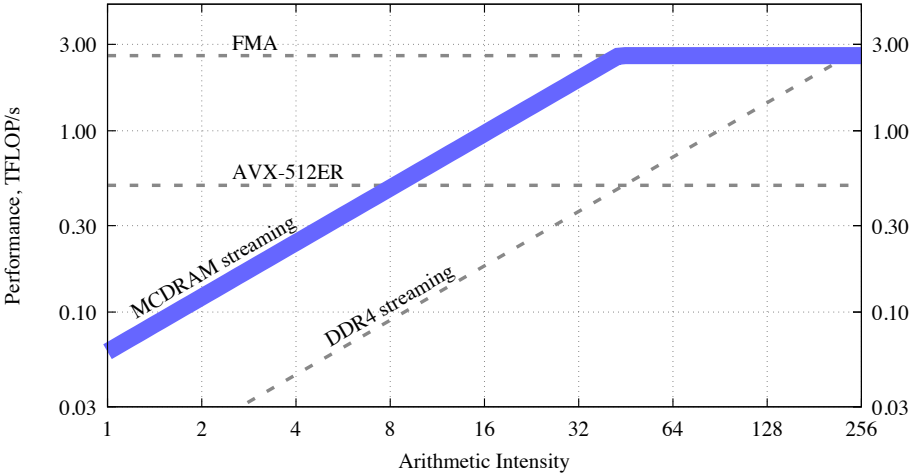
- > 50 FLOPs/Memory Access — Compute-bound Application
- < 50 FLOPs/Memory Access — Bandwidth-bound Application

ON COMPUTATIONAL COMPLEXITY OF ALGORITHMS

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

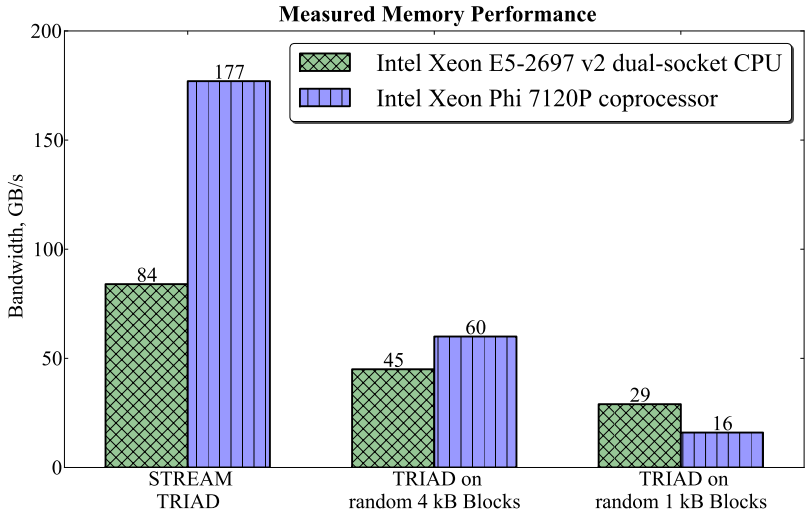
N = data size

ARITHMETIC INTENSITY AND ROOFLINE MODEL



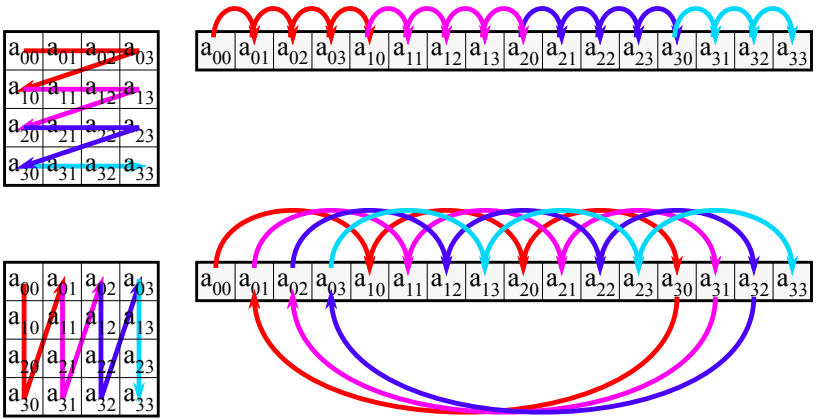
More on roofline model: Williams et al.

STREAMING VERSUS RANDOM ACCESS



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:

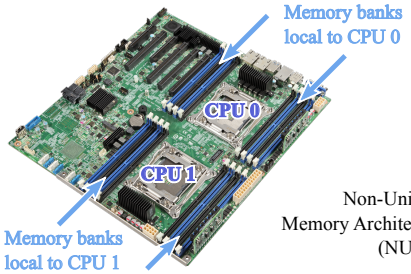
```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int j = 0; j < n; j++)
4 #pragma vector aligned
5     for (int k = 0; k < n; k++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

After:

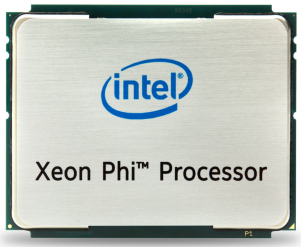
```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int k = 0; k < n; k++)
4 #pragma vector aligned
5     for (int j = 0; j < n; j++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

NUMA ARCHITECTURES

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



Non-Uniform
Memory Architecture
(NUMA)



Examples:

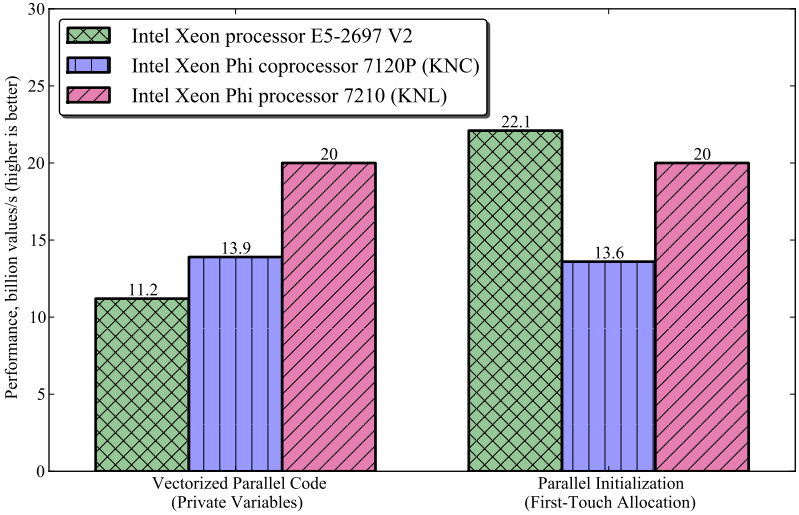
- Multi-socket Intel Xeon processors
- Second generation Intel Xeon Phi in sub-NUMA clustering mode

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



PRINCIPLE

- For best spatial locality, order loops to get unit-stride
- At `-O2` and above, the compiler may interchange loops
- In complex cases, investigate loop interchange manually
- May need to re-design data containers to get unit stride

LOOP FUSION TECHNIQUE

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

```
1  MyData* data = new MyData(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]);
```

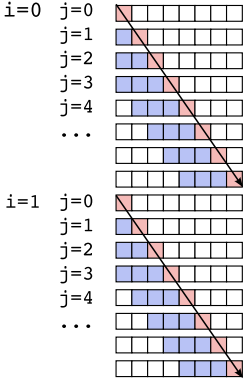
```
1  MyData* data = new MyData(n);
2
3  for (int i = 0; i < n; i++) {
4
5    Initialize(data[i]);
6
7    Stage1(data[i]);
8
9    Stage2(data[i]);
10 }
```

Potential positive side-effect: less data to carry between stages, reduced memory footprint, improved performance.

LOOP TILING: CACHE BLOCKING

Original:

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



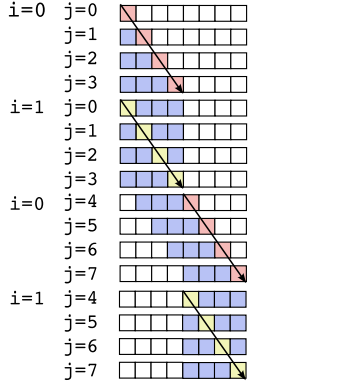
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

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

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

Tiled:

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



LOOP TILING (CACHE BLOCKING) -- PROCEDURE

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

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

```
// Step 2: permute
for (int jj = 0; jj < n; jj += TILE)
  for (int i = 0; i < m; i++)
    for (int j = jj; j < jj + TILE; j++)
      compute(a[i], b[j]); // Re-use to j=jj sooner
```

EXAMPLE: MATRIX-VECTOR MULTIPLICATION

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

```
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

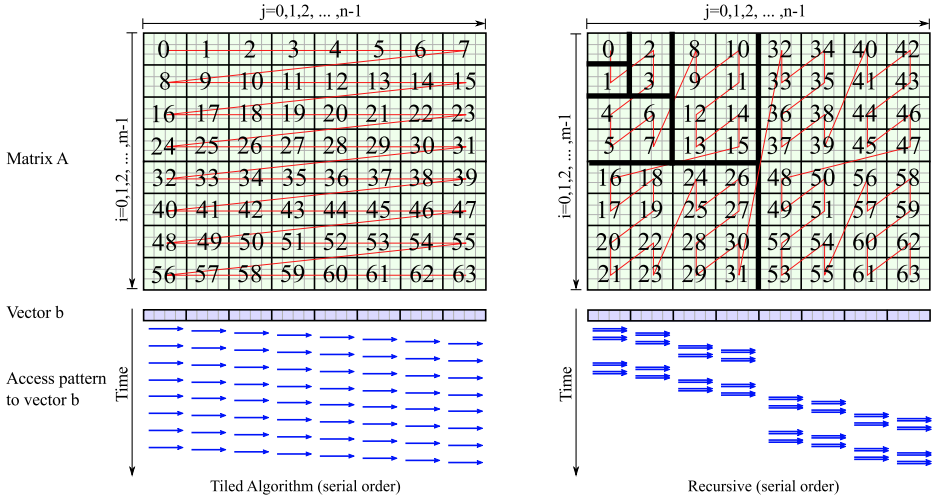
```
  const long jTile = 4096L; assert(n%jTile == 0);
#pragma omp parallel
  {
    double temp_c[m] __attribute__((aligned(64)));
    temp_c[:] =0;
#pragma omp for
    for (long jj =0; jj < n; jj+=jTile) // Loop Tiling in j
      for (long i = 0; i < m; i++)
#pragma vector aligned
        for (long j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

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

CACHE BLOCKING + STRIP-MINE AND COLLAPSE

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

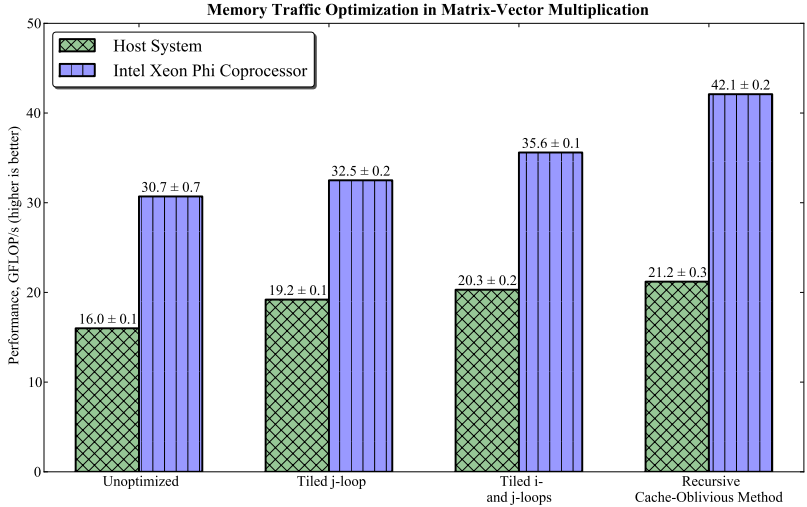
PRINCIPLE



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 } }
```

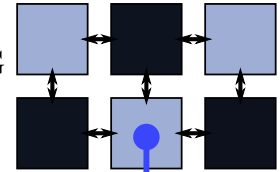
PERFORMANCE OF MATRIX VECTOR MULTIPLICATION



OPTIMIZING COMMUNICATION

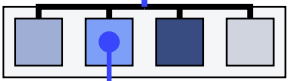
PARALLEL PROGRAMMING LAYERS

CLUSTER COMPUTING
in distributed memory



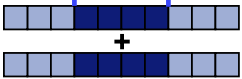
```
MPI_Sendrecv(data, k,
   MPI_DOUBLE, data2,
   ... );
```

MULTITHREADING
in shared memory



```
#pragma omp parallel for
for (j = 0; j < m; j++)
  ComputeSubset(j);
```

VECTORIZATION
of floating-point math



```
#pragma omp simd
for (i = 0; i < n; i++)
  A[i] += B[i];
```

STRUCTURE OF MPI APPLICATIONS: HELLO WORLD

```
#include "mpi.h"
#include <cstdio>
int main (int argc, char *argv[]) {
  MPI_Init (&argc, &argv);  // Initialize MPI envirnmnt
  int rank, size, namelen;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Comm_rank (MPI_COMM_WORLD, &rank); // ID of current process
  MPI_Get_processor_name (name, &namelen); // Hostname of node
  MPI_Comm_size (MPI_COMM_WORLD, &size); // Number of processes
  printf ("Hello World from rank %d running on %s!\n", rank, name);
  if (rank == 0) printf("MPI World size = %d processes\n", size);
  MPI_Finalize (); // Terminate MPI environment
}
```

MPICH site contains a list of MPI 3.2 routines

COMPILING AND RUNNING MPI APPLICATIONS ON LOCALHOST

```
u100@c005% mpiicpc -o HelloMPI HelloMPI.cc
```

Command file `mympi`:

```
#PBS -l nodes=1
cd $PBS_O_WORKDIR
mpirun -host localhost -np 2 ./HelloMPI
```

Results:

```
u100@c005% qsub mympi
2000
u100@c005% cat mympi.o2000
Hello World from rank 1 running on c005-n001!
Hello World from rank 0 running on c005-n001!
MPI World size = 2 processes
```

MPI APPLICATIONS: MACHINE FILE

Machine files are especially useful when working on large clusters.

Machine file `hosts.txt`:

```
c005-n001:1
c005-n002:1
```

Job submission:

```
vega@lyra% mpirun -machinefile hosts.txt ~/hello_mpi
```

Torque job file (assumes same executable path):

```
#PBS -l nodes=4
cd $PBS_O_WORKDIR
mpirun -machinefile $PBS_NODEFILE ./hello_mpi
```

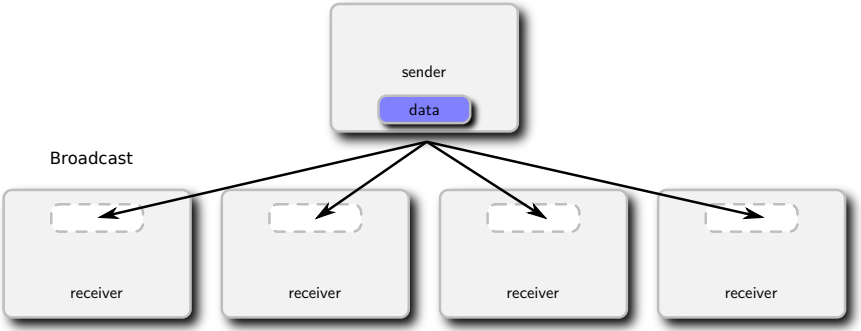
COMMUNICATION PATTERNS

POINT TO POINT COMMUNICATION

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

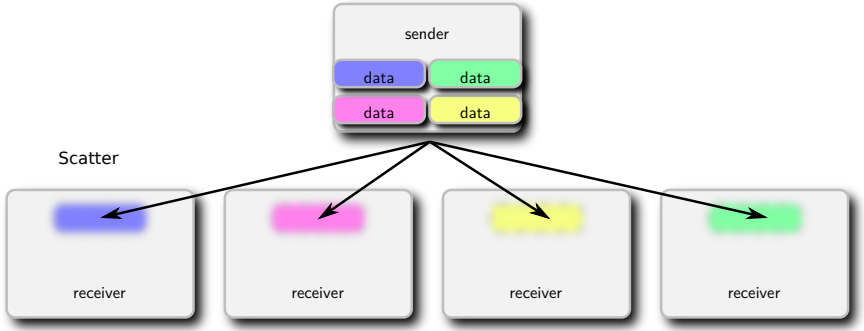
COLLECTIVE COMMUNICATION: BROADCAST

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



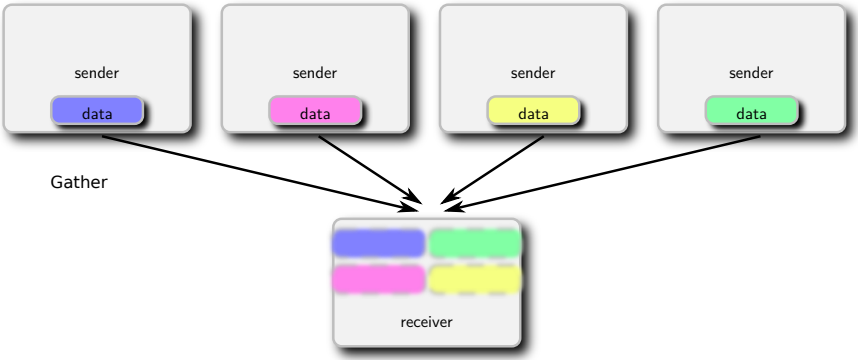
COLLECTIVE COMMUNICATION: SCATTER

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



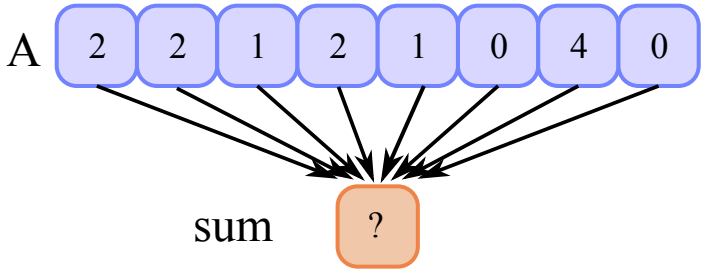
COLLECTIVE COMMUNICATION: GATHER

```
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

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



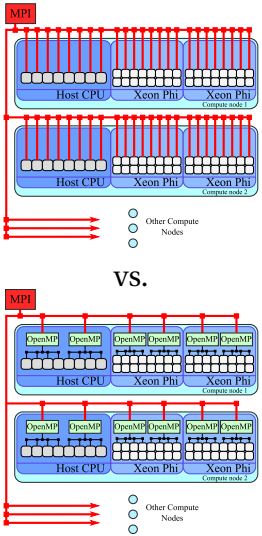
Available reducers: max/min, minloc/maxloc, sum, product, AND, OR, XOR (logical or bitwise).

INTER-OPERATION WITH OPENMP

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

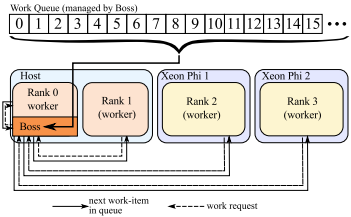


HYBRID MPI+OPENMP

- MPI pins processes to cores and sets OpenMP affinity for them.
- To tune pinning: `I_MPI_PIN`, `I_MPI_PIN_DOMAIN`
- To diagnose process pinning: `I_MPI_DEBUG=4`
- For MPI calls from multiple OpenMP threads, use `-mt_mpi`
- More information in the MPI Reference Manual

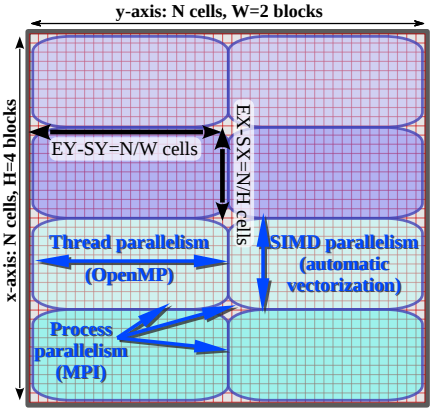
MULTI-THREADING WITHIN MPI PROCESSES

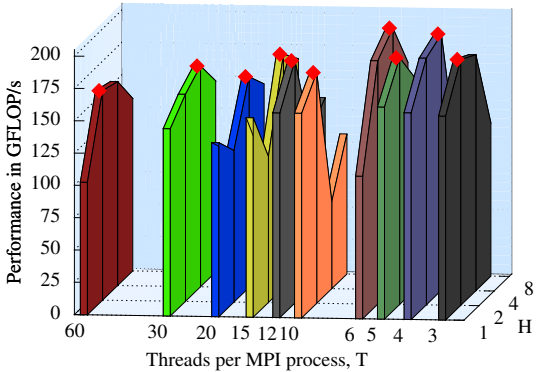
```
1  if(myRank == 0) { // Rank 0 has both a boss and a worker inside:
2      const int nThreads = omp_get_max_threads();  omp_set_nested(1);
3  #pragma omp parallel sections num_threads(2)
4      {
5  #pragma omp section
6        { DistributeWork(nOptions, option, mpiWorldSize); } // Boss
7  #pragma omp section
8        { omp_set_num_threads(nThreads-1); // Worker in rank 0:
9          ReceiveWork(option, payoff, myRank, optioncount);  } // ...
```



EXAMPLE OF OPENMP AND MPI INTER-OPERATION

Number of threads per process may be a tuning parameter:





Case study: this paper

EXAMPLE: ASIAN OPTIONS, HETEROGENEOUS DISTRIBUTED COMPUTING

EXAMPLE: THE MONTE CARLO METHOD OF ASIAN OPTION PRICING

1) Simulate Random-Walk of Asset Price

$$dS(t) = \mu S(t)dt + \sigma S(t)dB(t)$$

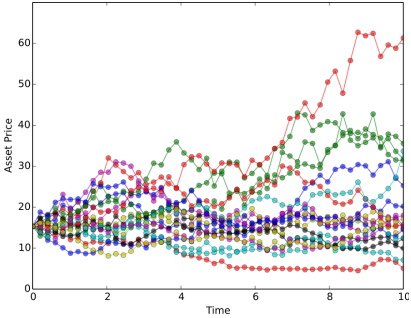
Using the Solution

$$S(t) = S(0)e^{(\mu - \frac{1}{2}\sigma^2)t + \sigma\sqrt{t}N(0,1)}$$

2) Perform Asian Option Price Averaging

$$\langle S \rangle_{\text{arithm}} = \frac{1}{N}\sum_{i=0}^{N-1} S(t_i),$$

$$\langle S \rangle_{\text{geom}} = \exp\left(\frac{1}{N}\sum_{i=0}^{N-1} \log S(t_i)\right)$$



3) Compute Discounted Pay-off

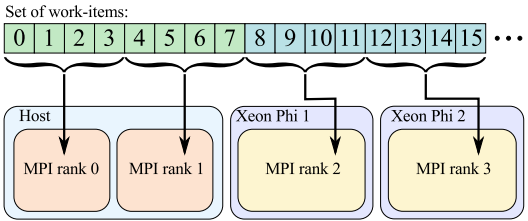
$$P_{\text{put}} = e^{-rT}\mathbb{E}\left(\max\left\{0; K - \langle S \rangle\right\}\right),$$

$$P_{\text{call}} = e^{-rT}\mathbb{E}\left(\max\left\{0; \langle S \rangle - K\right\}\right)$$

More information: this paper

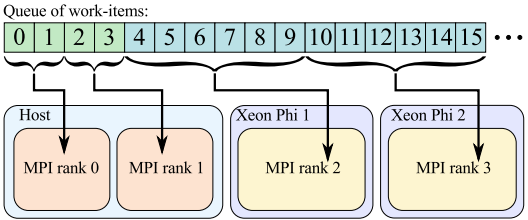
HETEROGENEOUS CALCULATION WITHOUT LOAD BALANCING

```
1  const double optionsPerProcess = double(nOptions)/double(mpiWorldSize);
2  const int myFirstOption = int(optionsPerProcess*(myRank));
3  const int myLastOption  = int(optionsPerProcess*(myRank+1));
4
5  // Static, even load distribution: assign options to ranks
6  for (int i = myFirstOption; i < myLastOption; i++)
7    ComputeOptionPayoffs(option[i], payoff[i]);
```

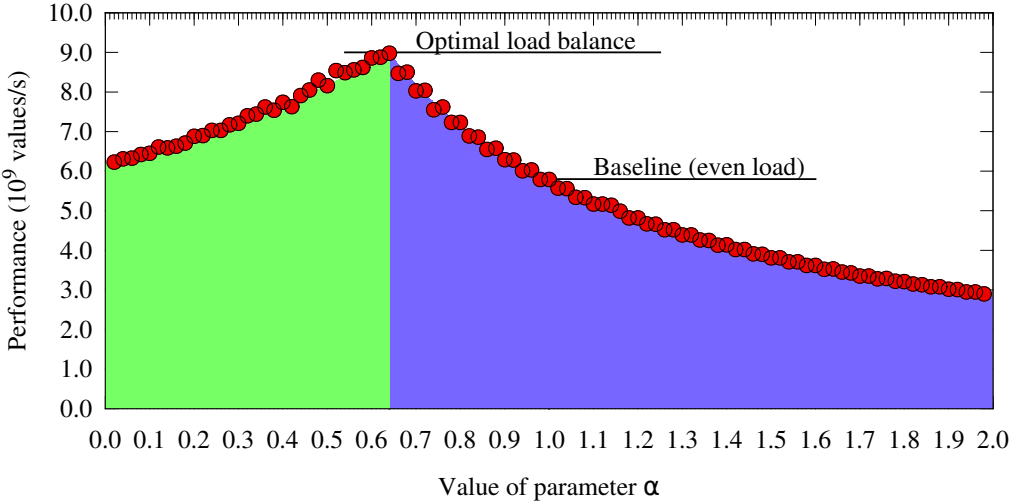


STATIC LOAD BALANCING

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

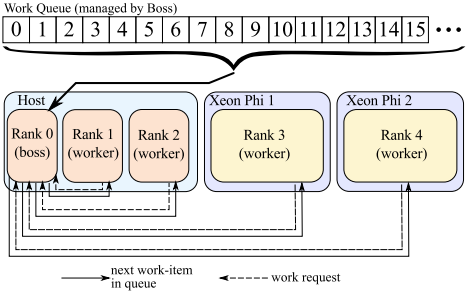


STATIC LOAD BALANCING: PARAMETER TUNING

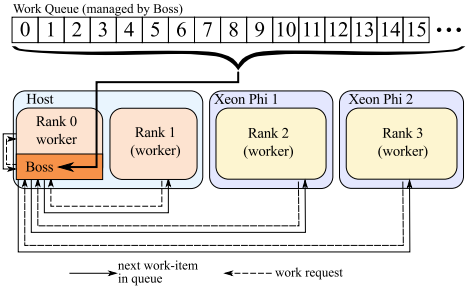


DYNAMIC LOAD BALANCING

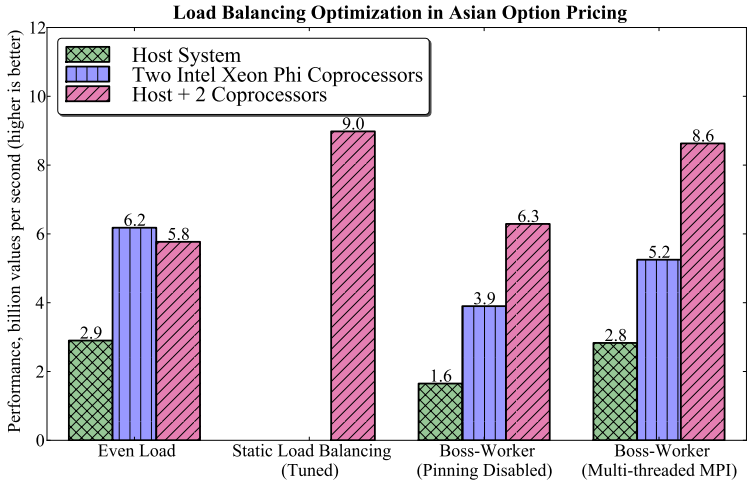
```
1  if (myRank == 0) // Boss's branch
2    DistributeWork(nOptions, option, mpiWorldSize);
3  else // Workers' branch
4    ReceiveWork(option, payoff, myRank);
```



or



PERFORMANCE WITH DIFFERENT SCHEDULING MODES



Refer to the book for explanation on the last two results.

WHERE TO LEARN MORE

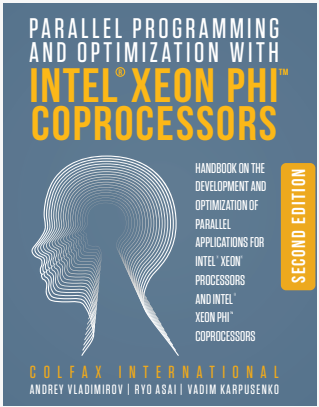

THE "HOW" SERIES TRAINING
DEEP DIVE
WITH CODE MODERNIZATION EXPERTS
It's free
→ HOWSERIES.COM
*10x 2-hour sessions | 24-hour 2-weeks remote access to a system

TEXTBOOK

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

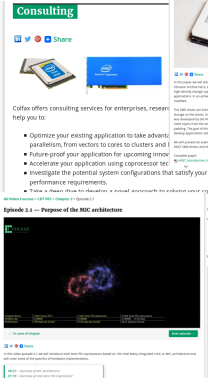
Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors

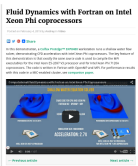
Handbook on the Development and Optimization of Parallel Applications for Intel® Xeon® Processors and Intel® Xeon Phi™ Coprocessors

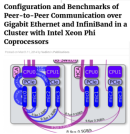


http://xeonphi.com/book

COLFAX RESEARCH

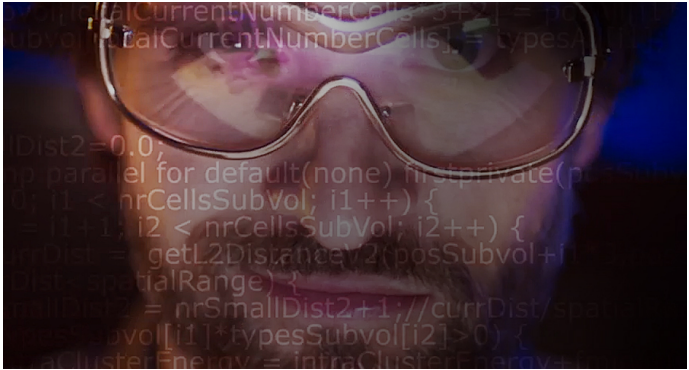






http://colfaxresearch.com/

INTEL RESOURCES



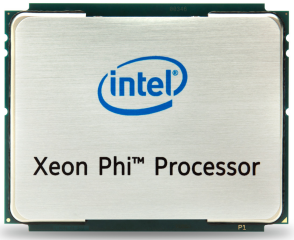
software.intel.com/modern-code

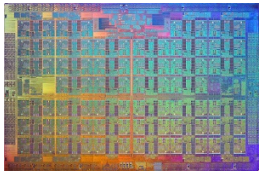
intel.com/xeonphi

§4. PREPARING FOR INTEL XEON PHI PROCESSORS

INTEL XEON PHI PROCESSORS (2ND GEN)

- Specialized for floating-point computing
- Highly-parallel (72 cores*)
- Balanced for compute
- Less forgiving than Xeon
- Theor. ~ 3.0 TFLOP/s in DP*
- Meas. ~ 490 GB/s bandwidth*

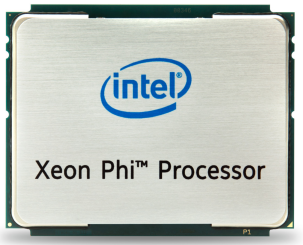


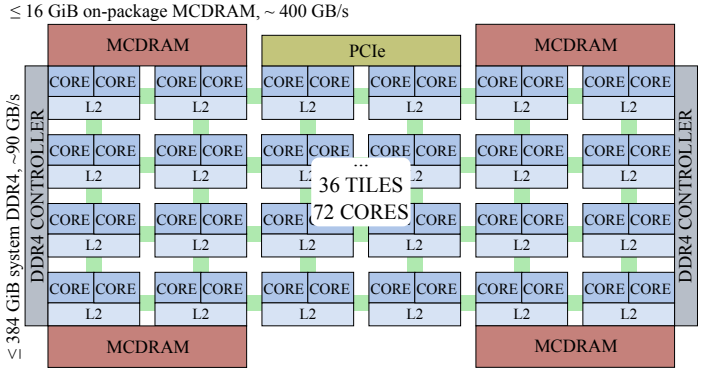


* Intel Xeon Phi processor, Knighs Landing architecture (2016), top-of-the-line (e.g., 7290P)

KNL DIE ORGANIZATION

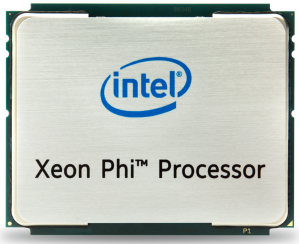
- Mesh interconnect relaxes data locality requirement [somewhat]
- All-to-all, quadrant or sub-numa domain communication in mesh

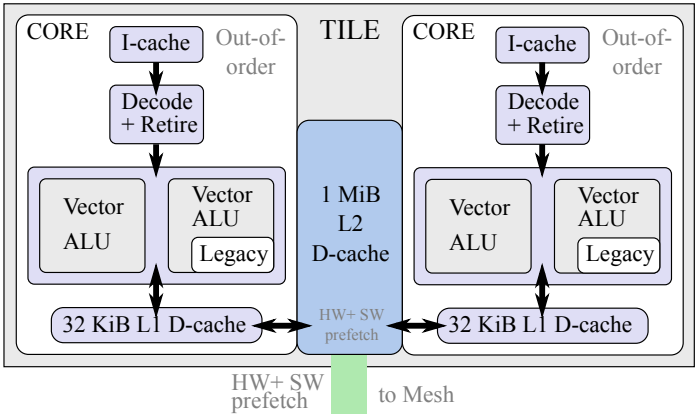




KNL CORES

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





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

colfaxresearch.com/knl-ready

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/knl-avx512

INTEL COMPILER SUPPORT FOR AVX-512

Intel C, C++ and Fortran compilers ≥ 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 ≥ 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:

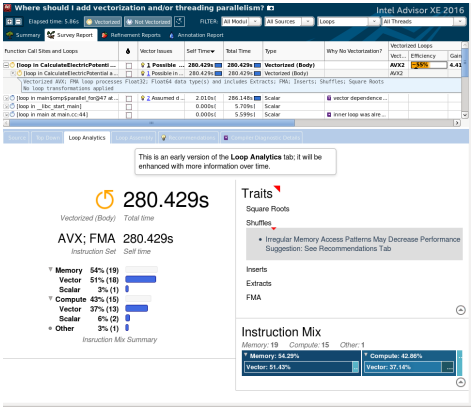
```
user@knl% g++ -v
gcc version 4.9.2 (GCC)
user@knl% g++ foo.cc -mavx512f -mavx512er -mavx512cd -mavx512pf -O3
```

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

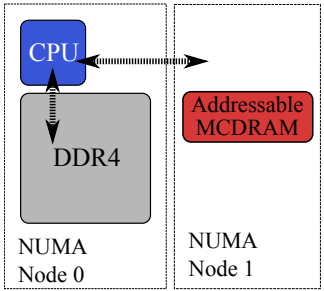
- Providing enough parallelism.
  - More consecutive vector operations required to overcome vectorization latency.
- Loop pipelining and unrolling.
  - Double the pipeline stages to populate.
- Better vectorization patterns.
  - Avoid long latency operations with unit-stride and unmasked operations.

USING HIGH-BANDWIDTH MEMORY

MODES OF HBM OPERATION

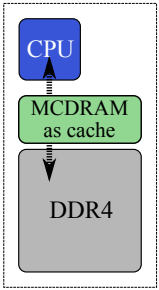
Flat Mode

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



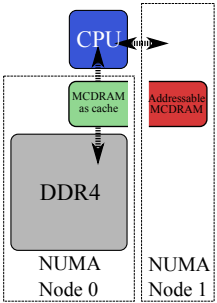
Cache Mode

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

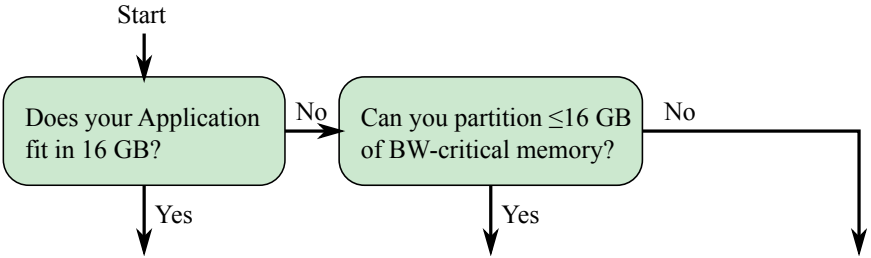


Hybrid Mode

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



FLOW CHART FOR BANDWIDTH-BOUND APPLICATIONS



numactl	Memkind	Cache mode
▷ Run the whole program in HBM	▷ Selectively allocate data to HBM	▷ Allow the chip to figure out how to use HBM
▷ No code modification	▷ Add memkind calls	▷ No code modification

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.

```c
#include <hbwmalloc.h>

// Basic allocation in HBM
double* A = (double*) hbw_malloc(sizeof(double)*n);

// Allocation with alignment
double* B;
int ret = hbw_posix_memalign((void**) &B, 64, sizeof(double)*n);

hbw_free(A); hbw_free(b); // Special deallocator
```

In Fortran:

```
REAL, ALLOCATABLE :: A(:)
!DEC$ ATTRIBUTES FASTMEM :: A
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:

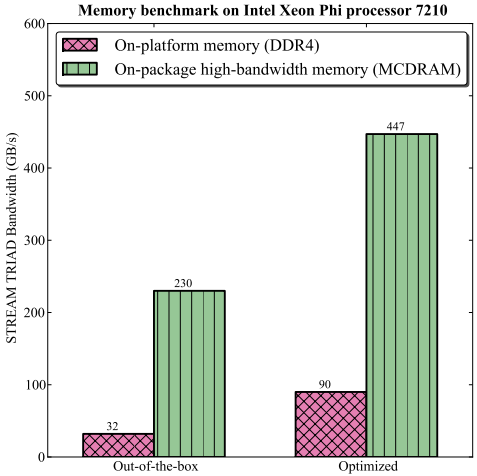
```
user@knl%  ifort -lmemkind foo.f90 -o runme
user@knl%  gfortran -lmemkind foo.f90 -o runme
```

Open source distribution of Memkind library can be found at:
memkind.github.io/memkind

Learn more:
colfaxresearch.com/knl-mcdram

STREAM BENCHMARK

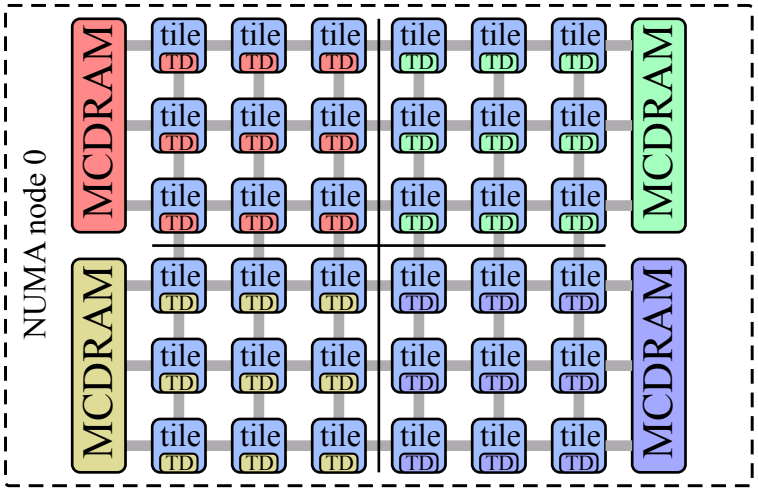
- Industry-standard tool for memory bandwidth measurement
- 4 tests: COPY, ADD, SCALE and TRIAD
- Download from Dr. John McCalpin's site: www.cs.virginia.edu/stream/



LEVERAGING CLUSTERING MODES

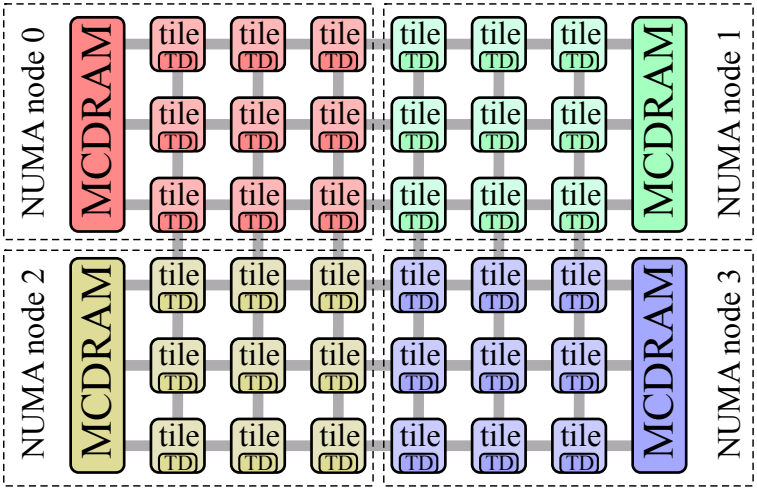
CLUSTERING MODES: QUADRANT/HEMISPHERE

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



CLUSTERING MODES: SNC-4/SNC-2

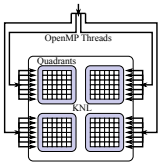
Cores appear as 4 (or 2) NUMA nodes.



HOW TO USE CLUSTERING MODES

Nested OpenMP

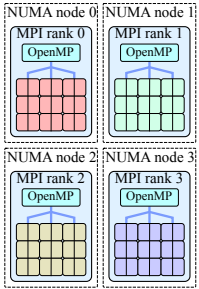
```
1 #pragma omp parallel
2 {
3   // ...
4 #pragma omp parallel
5   {
6     // ...
7   }
8 }
```




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

MPI+OpenMP

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




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

Learn more: colfaxresearch.com/knl-numa

§5. INTEL LIBRARIES

INTEL MKL: THE MAGIC PILL

INTEL® MATH KERNEL LIBRARY

Intel® Math Kernel Library (MKL) — free 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.

MKL C/C++ CODE SAMPLES

Thread-safe, vectorized RNG.

```
float *matrix = (float *) malloc(sizeof(float)*N*M);
#pragma omp parallel
  { VSLStreamStatePtr rnStream;
    vslNewStream( &rnStream, VSL_BRNG_MT19937, random_seed);
#pragma omp for
    for(size_t i = 0; i < N; i++)
      vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, M, matrix, -1.0, 1.0);
  }
```

Optimized BLAS API.

```
float A[N*N], B[N*N], C[N*N];
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            N, N, N, 1.0, A, N, B, N, 0.0, C, N);
```

For more, see the MKL documentation

MKL WITH R

Install R with MKL support.

```
user@host% ./configure --with-blas="-L/opt/intel/mkl/lib/intel64          \
%  -lmkl_intel_lp64  -lmkl_core -lmkl_intel_thread -lpthread -lm"         \
%  --with-lapack CC=icc CFLAGS="-O2 -qopenmp -I/opt/intel/mkl/include"    \
%  CXX=icpc CXXFLAGS="-O2 -qopenmp -I/opt/intel/mkl/include" F77=ifort    \
%  FFLAGS="-O2 -qopenmp -I/opt/intel/mkl/include" FC=ifort FCFLAGS="-O2 \
%  -qopenmp -I/opt/intel/mkl/include" --prefix=/opt/R-3.4.0
user@host% make && make install
```

In-place upgrade: no code change required.

```
user@host% cat gemm.R
N=5000
A<-matrix(runif(N*N), N, N)
B<-matrix(runif(N*N), N, N)
C = tcrossprod(A, B)
```

MKL_VERBOSE

Use MKL_VERBOSE to see the calls to MKL and overhead of the calls.

```
user@host% cat gemm.R
N=5000
A<-matrix(runif(N*N), N, N)
B<-matrix(runif(N*N), N, N)
for(i in 1:10) {
  Sys.time()->start
  C = tcrossprod(A, B)
  print(Sys.time()-start)
}
user@host% export MKL_VERBOSE=1
user@host% Rscript gemm.R
// ... //
MKL_VERBOSE DGEMM(N,T,5000,5000,5000,0x7fffc4b0fc60, ... ) 153.73ms  ...
Time difference of 0.4924946 secs
MKL_VERBOSE DGEMM(N,T,5000,5000,5000,0x7fffc4b0fc60, ... ) 151.93ms  ...
Time difference of 0.4966147 secs
```

MKL WITH VARIOUS LANGUAGES

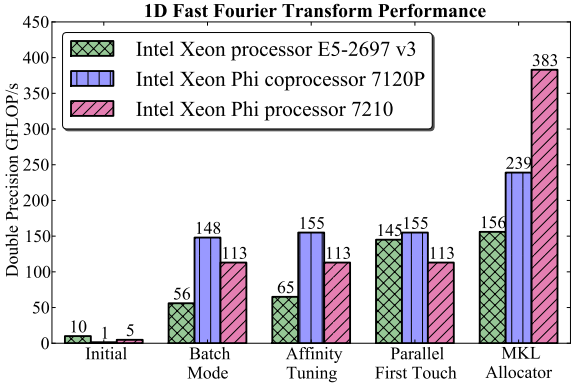
MKL is supported on various languages. For example:

- C/C++ - linked to library Intel MKL Developer Reference for C
- R - compiled in Intel article.
- Fortran -linked to library Intel MKL Developer Reference for Fortran
- Python - compiled in Intel Distribution for Python
- Julia - compiled in git repo (under section "Intel compilers and Math Kernel Library (MKL)")
- Matlab - compiled in Intel article with instructions

Intel MKL download can be found at https://software.intel.com/en-us/mkl.

ENVIRONMENT TUNING MAY BE NECESSARY

$2 \cdot 10^5$ FFTs of size 2048. See HOW “KNL” for details.



INTEL DAAL: DATA ANALYTICS

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/Ridge Regresion
- Clasification
  - Naive Bayes Classifier
  - Boosting
  - SVM
  - Neural Networks
  - Multi-Class Classifier

Portal: DAAL page. See also: intro article, CR papers.

DAAL C++ CODE SAMPLE

```
using namespace daal;
using namespace daal::data_management;
using namespace daal::algorithms::linear_regression;
// Setting up the training sets.
services::SharedPtr<NumericTable> trnFeatures(trnFeatNumTable);
services::SharedPtr<NumericTable> trnResponse(trnRespNumTable);
// Setting up the algorithm object
training::Batch<> algorithm;
algorithm.input.set(training::data, trnFeatures);
algorithm.input.set(training::dependentVariables, trnResponse);
// Training
algorithm.compute();
// Extracting the result
services::SharedPtr<training::Result> trainingResult;
trainingResult = algorithm.getResult();
```

See Colfax Research paper for more.

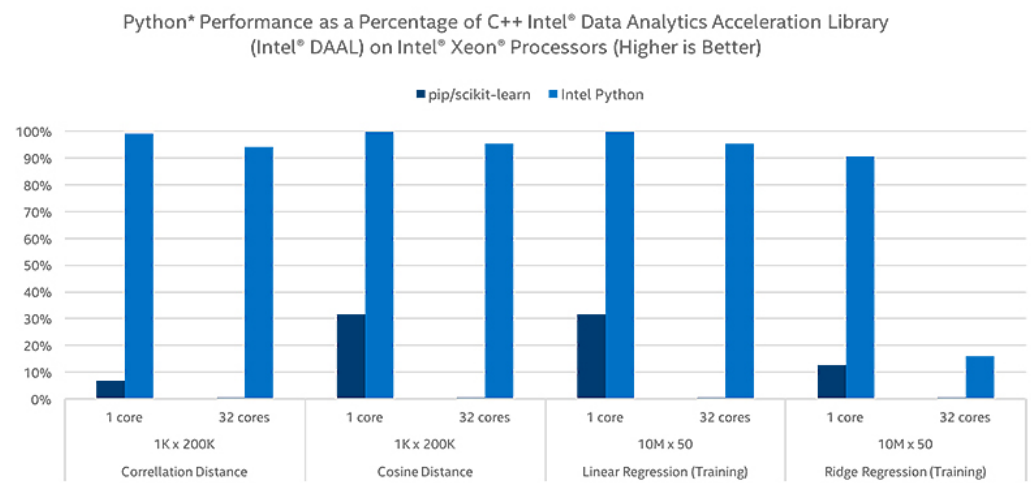
SCIKIT-LEARN WITH DAAL

Parts of scikit-learn have been integrated with DAAL (Intel video)

```
user@host% cat kmeans.py
from sklearn.cluster import KMeans
from sklearn.datasets import fetch_mldata
import time
data = fetch_mldata("MNIST original").data
start = time.time()
kmeans = KMeans(n_clusters=64).fit(data)
print(str(time.time()-start)+" seconds")
user@host% # default sklearn
user@host% /usr/local/python kmeans.py
1944.01453304 seconds      # 10 runs
user@host% # Intel DAAL enhanced sklearn
user@host% /opt/intel/intelpython_update_3/intelpython2/bin/python kmeans.py
5.6174788475 seconds       # 5  runs
```

Optimization of K-Means clustering: Colfax Research Paper

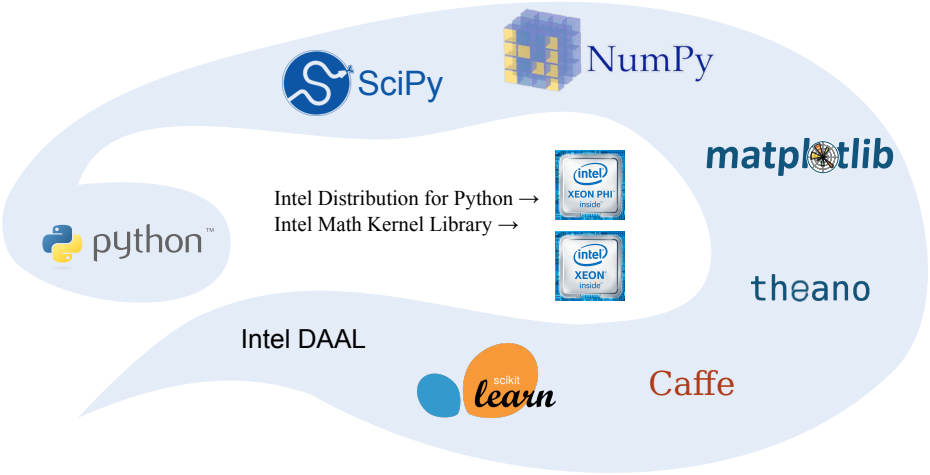
SCIKIT-LEARN PERFORMANCES



Source: Intel DAAL features page

§6. INTEL DISTRIBUTION FOR PYTHON

INTEL DISTRIBUTION FOR PYTHON



Portal: software.intel.com/intel-distribution-for-python. See also: CR paper.

GETTING INTEL PYTHON

Intel Python can be downloaded from Intel website

```
user@host% tar xzf l_python2_pu3_2017.3.053.tgz
user@host% cd l_python2_pu3_2017.3.053
user@host% ./install.sh
// Follow on-screen instructions
user@host% export PATH=/opt/intel/intelpython27/bin:${PATH}
```

Also supported for Anaconda (Intel article)

```
user@host% # Requires conda 4.1.11 and above
user@host% conda config --add channels intel     # use intel as default
user@host% conda create -n idp intelpython2_core python=2   # core only
user@host% # conda create -n idp intelpython2_full python=2   # full install
user@host% source activate idp
```

DIMENSIONALITY REDUCTION

Dimensionality Reduction – reducing number of variables with minimal loss of information.

```
import numpy as np
from sklearn.datasets import fetch_mldata
from sklearn.decomposition import PCA
# getting data, and preprocessing
data = fetch_mldata("MNIST original").data[0:60000,:]/255
data = data - np.mean(data, axis=0)
# dimensionality reduction
print("Original data: "+str(transformed_data.shape))
pca = PCA(n_components=200).fit(data)
print(str(np.sum(pca.explained_variance_ratio_)*100)+"% of variance")
transformed_data = np.dot(data, np.transpose(pca.components_))
print("Transformed data: "+str(transformed_data.shape))
```

```
Original data: (60000, 784)
87.98% of variance
Transformed data: (60000, 200)
```

DIMENSIONALITY REDUCTION PERFORMANCE

```
t_pca_start = time.time()
pca = PCA(n_components=200).fit(data)
print("PCA :"+str(time.time()-t_pca_start)+" seconds")
# ... same for np.dot ...
```

```
user@host% numactl -m 1 python dimensionality-reduction.py
```

On Intel Xeon Phi processor 7250:

- System Python (numpy ver. 1.13.1, scikit-learn ver. 0.18.2)

```
PCA : 81.3773860931 seconds    Dot : 6.7592010498 seconds
```

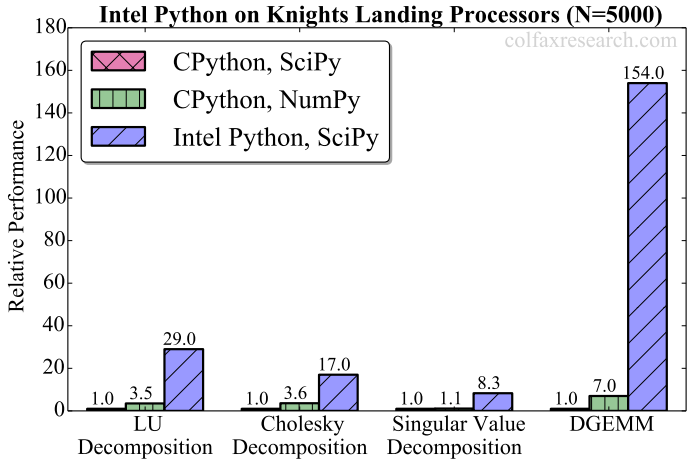
- Intel Distribution for Python (2017 update 3)

```
PCA : 5.84652495384 seconds    Dot : 0.038341999054 seconds
```

*Similar performance for `fit_transform()`

HOW TO GET THE MOST OUT OF NUMPY AND SCIPY

INTEL PYTHON PERFORMANCE



Portal: software.intel.com/intel-distribution-for-python. See also: CR paper.

DGEMM WITH SCIPY

```
import numpy as np
from numpy.random import rand as rn
from scipy.linalg.blas import dgemm as dgemm

m=5000
n=5000
k=5000

// Using column-major format with order="F"
A = np.array(np.random.rand(m, n), dtype = np.double, order="F");
B = np.array(np.random.rand(n, k), dtype = np.double, order="F");

// Scipy matrix-matrix multiplication
C=dgemm(alpha=1.0, a=A, b=B, c=C, overwrite_c=1, trans_b=1)
```

DGEMM WITH NUMPY

```
import numpy as np
from numpy.random import rand as rn

m=5000
n=5000
k=5000

// Using row-major by default (you can also set order="C")
A = np.array(np.random.rand(m, n), dtype = np.double);
B = np.array(np.random.rand(n, k), dtype = np.double);

// Numpy matrix-matrix multiplication
C = np.dot(A, B)
```

MISC. SCIPY WORKLOADS

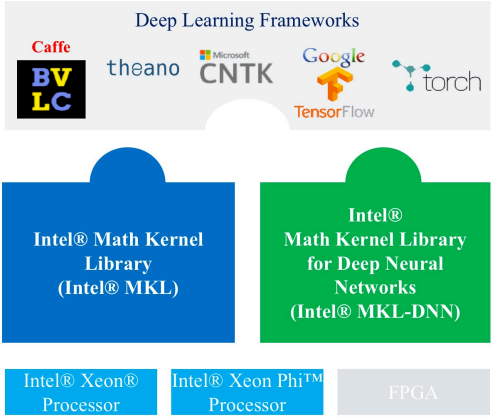
```
from scipy.linalg import lu as lu;
# ... #
A = np.array(np.random.rand(k, k), dtype = np.double, order='F')
P,L,U = lu(A, permute_l=False, overwrite_a=True, check_finite=False)
```

```
from scipy.linalg import cholesky as cholesky
# ... #
A = np.array(tempA, dtype=np.double, order='F')
L = cholesky(A, overwrite_a=True, check_finite=False)
```

```
from scipy.linalg import svd as svd
# ... #
A = np.array(np.random.rand(k, k), dtype = np.double, order='F');
U, s, V = svd(A)
```

§7. INTEL-OPTIMIZED DL FRAMEWORKS

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[1]
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

[1] GEMM building blocks are binary.

slide credit: Intel corp.

OPTIMIZED DEEP LEARNING FRAMEWORKS

Deep Learning frameworks that have been optimized for Intel Architectures. All are available on Colfax Cluster.

- Intel Caffe – GitHub

```
u100@c005% /opt/caffe-master
```

- Intel Theano/Keras – Github Theano/Keras/

```
import theano; import keras
```

- Intel Torch – GitHub

```
u100@c005% /opt/torch
```

- TensorFlow – GitHub (up-streamed)

```
import tensorflow
```

- Intel MXNet – GitHub

```
import mxnet
```

EXAMPLE: MNIST WITH TENSORFLOW

MNIST AND TENSORFLOW™

MNIST database

- Handwritten digit images (28p×28p)
- Public database
- 60000 train set, 10000 test set
- LeCun et. al.

sample digits

TensorFlow™*

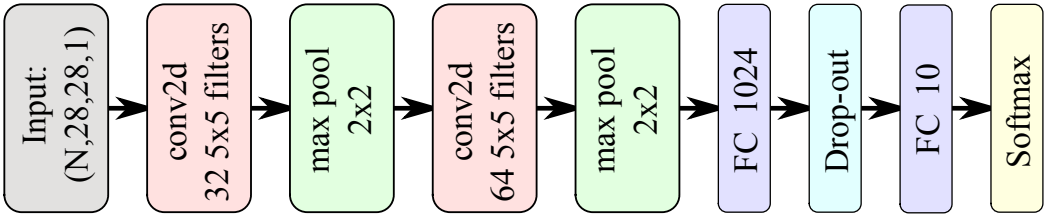
- ML framework developed by Google Inc.
- Uses data flow graphs
- Open source library
- Supports MKL DNN back-end**

* TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

** Not all versions supports MKL DNN

OUT-OF-BOX MNIST TENSORFLOW CONVOLUTIONAL NEURAL NETWORK EXAMPLE

Convolutional Neural Network from TensorFlow tutorial, Deep MNIST for Experts



Code and parameters are from the tutorial, but a few changes:

- Number of batches: 20000 → 1000
- Mini-batch size: 50 → 256
- Using `time.time()` to get the execution time

RUNNING EXAMPLE TENSORFLOW ON COLFAX CLUSTER

Use Intel Python 2.7 Update 3 and GCC 6.3.0 by setting up `~/.bash_profile`

```
u100@c005% cat ~/.bash_profile
>  # ... at the end ...#
>  export LD_LIBRARY_PATH=/opt/gcc-6.3.0/lib64/:${LD_LIBRARY_PATH}
>  export PATH=/opt/gcc-6.3.0/bin/:${PATH}
>  export PATH=/opt/intel/intelpython_update_2/intelpython2/bin/:${PATH}
```

Inside the job script:

```
numactl -m 1 python mnist_deep.py --data_dir=/opt/data/MNIST
```

Out-of-the-box result (Intel Xeon Phi processor 7210):

```
step 100, accuracy= 0.875
// .... //
step 900, accuracy= 0.957031
Completed in 2498.61 seconds
Test accuracy = 0.9764
```

NHWC VS NCHW

By default TensorFlow uses **NHWC** (batch, height, width, channel) tensors.

MKL DNN prefers **NCHW** (batch, channel, height, width)

- Use NCHW for `tf.reshape()`

```
# NCHW (batch, channel, height, width)
tf.reshape(x, [-1,1,28,28])
```

- Specify `data_format='NCHW'` an change strides for `conv` and `pool` layers

```
# Using NCHW strides
tf.nn.conv2d( ... , strides=[1,1,1,1], data_format='NCHW')
tf.nn.max_pool( ... , ksize=[1,1,2,2],  strides=[1,1,2,2], data_format='NCHW')
```

- The + operator uses NHWC by default, so instead use the `tf.nn.bias_add`

```
tf.nn.relu(tf.nn.bias_add(conv(h_pool1, w_conv2),b_conv2,data_format='NCHW'))
```

- No need to change weights or bias, or fully connected layers

ENVIRONMENT TUNING

Tuning parameters for OpenMP can improve performance.

- Method 1: inside the job script

```
export KMP_BLOCKTIME=1
export OMP_NUM_THREADS=32
export KMP_AFFINITY=granularity=fine,compact,1,0
```

- Method 2: inside the python script

```
os.environ["KMP_BLOCKTIME"] = "1"
os.environ["OMP_NUM_THREADS"] = "32"
os.environ["KMP_AFFINITY"]= "granularity=fine,compact,1,0"
```

- `KMP_BLOCKTIME` – time in `ms` that threads stay active after parallel regions
- `OMP_NUM_THREADS` – number of threads that OpenMP uses
- `KMP_AFFINITY` – controlling mapping from software threads to physical cores

TUNED PERFORMANCE

The same training runs approximately **37x** faster on Xeon Phi processor 7210.

```
# NCHW tensor data format
# KMP_BLOCKTIME=1
# KMP_AFFINITY=granularity=fine,compact,1,0
# OMP_NUM_THREADS=32
# using MCDRAM with "numactl -m 1"
step 100, accuracy= 0.878906
// ... //
step 900, accuracy= 0.972656
Completed in 67.7578 seconds
Test accuracy = 0.9757
```

Conclusion

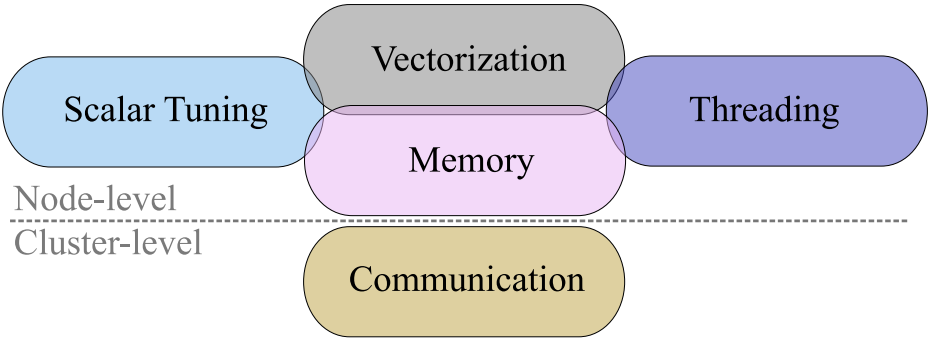
Use proper data format and tune environment variables when using ML libraries

For more, see the Intel article

§8. CLOSING WORDS

Code Modernization

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




THE "HOW" SERIES TRAINING
DEEP DIVE
WITH CODE MODERNIZATION EXPERTS
It's free
→ HOWSERIES.COM
*10x 2-hour sessions | 24-hour 2-weeks remote access to a system