PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

One-Day Workshop

Andrey Vladimirov and Ryo Asai
Colfax International — colfaxresearch.com

October 2016
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.
§1. SNEAK PEAK
AGENDA AND WHAT'S IN IT FOR YOU
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 issue,s, 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

**Natural sciences**
- Drug development
- Space weather
- New particle discovery

**Aerospace**
- Rocket engine simulation
- Radiation protection studies
- Aircraft aerodynamics

**Manufacturing**
- Simulation of material processing
- Development of new materials
- Robotic operations

**Computing**

**Finance**
- Risk analysis
- Fraud prevention
- Real-time hedging

**Internet**
- Natural speech interaction
- Video streaming and archiving
- Facial recognition

**Medicine**
- Medical imaging
- Radiation treatment simulation
- Human error detection

**Energy**
- Oil reservoir simulation
- Climate change science
- Nuclear fusion research

**Environment**
- Weather prediction
- Earthquake simulation
- Pollutant propagation

**Finance**

**Internet**

**Medicine**

**Energy**

**Environment**
Common story for many applications:

(see http://xeonphi.com/papers/heatcode)
**OPTIMIZATION AREAS**

- **Scalar Tuning**: what goes on in the pipeline?
- **Vectorization**: is SIMD parallelism used well?
- **Threading**: do cores cooperate efficiently?
- **Memory**: is cache usage maximized or RAM access streamlined?
- **Communication**: can coordination in a distributed or heterogeneous system be improved?

---

**AGENDA AND WHAT’S IN IT FOR YOU**

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
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

© Colfax International, 2015

http://xeonphi.com/book

WHERE TO LEARN MORE
WHERE TO LEARN MORE

software.intel.com/modern-code

intel.com/xeonphi
§2. PROGRAMMING, OPTIMIZATION BY EXAMPLE
DIRECT N-BODY SIMULATION
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:

\[ M_i \ddot{R}_i(t) = G \sum_j \frac{M_i M_j}{\left| \vec{R}_i - \vec{R}_j \right|^3} \left( \vec{R}_j - \vec{R}_i \right) \]

where:

\[ \left| \vec{R}_i - \vec{R}_j \right| = \sqrt{(R_i, x - R_j, x)^2 + (R_i, y - R_j, y)^2 + (R_i, z - R_j, z)^2} \]

particles are attracted to each other with the gravitational force
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
```
void MoveParticles(int nParticles, ParticleType* particle, float dt) {
    for (int i = 0; i < nParticles; i++) { // Particles that experience force
        float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
        for (int j = 0; j < nParticles; j++) { // Particles that exert force
            // Newton's law of universal gravity
            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;
            const float drSquared = dx*dx + dy*dy + dz*dz + 1e-20;
            const float drPower32 = pow(drSquared, 3.0/2.0);
            // Calculate the net force
            Fx += dx/drPower32; Fy += dy/drPower32; Fz += dz/drPower32;
        }
        // Accelerate particles in response to the gravitational force
        particle[i].vx+=dt*Fx; particle[i].vy+=dt*Fy; particle[i].vz+=dt*Fz;
    }
}
Before:

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

After:

```c
#pragma omp parallel for
for (int i = 0; i < nParticles; i++) { // Particles that experience force
    float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
    for (int j = 0; j < nParticles; j++) { // Particles that exert force
        // Newton’s law of universal gravity
    ...
```
Before:

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

After:

```c
const float drRecip = 1.0f/sqrf(dx*dx + dy*dy + dz*dz + 1e-20);
const float drPowerN32 = drRecip*drRecip*drRecip;
// Calculate the net force
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
Before:

```c
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:

```c
struct ParticleSet {
}; // ...

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

Array of Structures (sub-optimal)

Structure of Arrays (optimal)

Memory

Vector Register

Array of Structures (sub-optimal)

Memory

Vector Register

Structure of Arrays (optimal)


colfaxresearch.com/events
Original:
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];

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

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

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

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

After: (tileSize = 16)

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

Impact of Code Optimization

N-Body Simulation Performance

Step 0: Initial
Step 1: Multi-threaded
Step 2: Scalar Tuning
Step 3: Vectorized w/SoA
Step 4: Memory Optimization

Performance, GFLOP/s

- Intel Xeon E5-2697 v3 (Haswell)
- Intel Xeon Phi 7120A (1st gen, KNC)
- Intel Xeon Phi 7220 (2nd gen, KNL)

MPI processes only on CPUs
Divide data between coprocessors
Concurrent offload from multiple host threads
Synchronize data between nodes with MPI
Simple: all particles on each compute node; exchange updated particle coordinates.

```c
void MoveParticles(int nParticles, ParticleSet& particle, float dt,
                    int mpiRank, int mpiWorldSize) {
    const int myParticles = nParticles / mpiWorldSize;
    const int startParticle = (mpiRank) * myParticles;
    const int endParticle = (mpiRank + 1) * myParticles;
    // Outer loop over only the subset of particles processed by present process
    #pragma omp parallel for schedule(guided)
    for (int ii = startParticle; ii < endParticle; ii += tileSize) {
        for (int j = 0; j < nParticles; j++) // ...But inner loop over all particles
            //...
    }
    // ... Propagate results of time step across the cluster
    MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, particle.x,
                  myParticles, MPI_FLOAT, MPI_COMM_WORLD);
    // ...
```
MPI processes only on CPUs

Divide data between coprocessors

Concurrent offload from multiple host threads

Synchronize data between nodes with MPI
```cpp
const int nDevices = Offload_number_of_devices();
const int particlesPerDevice = (nDevices == 0 ? myParticles : myParticles/nDevices);
#pragma omp parallel num_threads(nDevices) if(nDevices>0)
{
  const int iDevice = omp_get_thread_num();
  const int startParticle = rankStartParticle + (iDevice)*particlesPerDevice;
  #pragma offload target(mic:iDevice) if(nDevices>0) \
  in (x : length(nParticles) alloc_if(alloc==1) free_if(0)) \
  out(x [startParticle:particlesPerDevice] : alloc_if(0) free_if(alloc==-1)) \
  in (vx: length(nParticles*alloc*alloc) alloc_if(alloc==1) free_if(0)) \
  //...
  // Loop over particles that experience force
  #pragma omp parallel for schedule(guided)
  for (int ii = startParticle; ii < endParticle; ii += tileSize) {
    // ...
```

MPI WITH OFFLOAD IMPLEMENTATION
RESULTS WITH MPI+OFFLOAD

Intel Xeon E5-2697 v2 CPUs (4 nodes)
Intel Xeon Phi 7120P coprocessors (4 per node)
N=2^{20} particles (strong scaling)

1 Xeon Phi/node
2 Xeon Phi/node
3 Xeon Phi/node
4 Xeon Phi/node

Xeon Phi, native MPI
Xeon Phi, MPI+Offload

CPU

Performance, TFLOP/s

Number of Nodes or Coprocessors (P)
INTEL ARCHITECTURE
Computing Platforms

Intel Xeon Processor
- Current: Broadwell
- Upcoming: Skylake

Intel Xeon Phi Coprocessor, 1st generation
- Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*
- Knights Landing (KNL)
  - * socket and coprocessor versions

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture
- C/C++/Fortran
- Linux/Windows
- ≤3 TiB DDR4
- ≤44 cores (2-way)
- ≈3 GHz
- 2 HT/core
- 256-bit AVX

- C/C++/Fortran
- Special Linux
- ≤ 16 GiB GDDR5
- 57-61 cores
- ≈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
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)
Mesh interconnect relaxes data locality requirement [somewhat]

All-to-all, quadrant or sub-numa domain communication in mesh
Even more power in vector units

Binary compatible with Xeon, but in legacy mode
Utilize cores: run multiple threads/processes (MIMD)
Utilize vectors: each thread (process) issues vector instructions (SIMD)
This approach often works:

```c
#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:

```c
#pragma omp parallel for simd
for (int i = 0; i < n; i++) // If the problem is all data-parallel
    DoSomeWork(A[i]);
```
Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

Scalar Instructions

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>+</td>
<td>3</td>
</tr>
<tr>
<td>-2</td>
<td>+</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>+</td>
<td>-7</td>
</tr>
</tbody>
</table>

= 5
= 3
= 6
= 2

Vector Instructions

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>+</td>
<td>3</td>
</tr>
<tr>
<td>-2</td>
<td>+</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>+</td>
<td>-7</td>
</tr>
</tbody>
</table>

= 5
= 3
= 6
= 2
INSTRUCTION SETS IN INTEL ARCHITECTURE

- MMX
- SSE
- SSE2
- SSE3
- SSE4.2
- ...
- IMCI
- AVX
- AVX-512

Intel Xeon Phi

KNC

KNL

64-bit
128-bit
256-bit
512-bit


CORES AND VECTORS

colfaxresearch.com/events

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
- Direct access to \( \leq 16 \text{ GiB} \) 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):
  Host
  ```
  main() {
  #pragma offload target(mic)
  }
  ```
  Coprocessor
  ```
  myFunction();
  ```

- Native model (standalone application/MPI process):
  Host
  ```
  ```
  Coprocessor
  ```
  main() {
  myFunction();
  }
  ```
“Hello World” application:

```c
#include <cstdio>
#include <unistd.h>

int main(){
    printf("Hello world! I have %ld logical processors.\n", 
       sysconf(_SC_NPROCESSORS_ONLN ));
}
```

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

```bash
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 ...
```
“Hello World” in the explicit offload model:

```c
#include <cstdio>
int main() {
    printf("Hello World from host!\n");
    #pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n");
        fflush(stdout);
    }
    printf("Bye\n");
}
```

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
#pragma offload_attribute(push, target(mic))

```c
void MyFunctionOne() {
    // ... implement function as usual
}
```

```c
void MyFunctionTwo() {
    // ... implement function as usual
}
```

```c
#pragma offload_attribute(pop)
```

To mark a long block of code with the offload attribute, use `#pragma offload_attribute(push/pop)`
double *p1 = (double*)malloc(sizeof(double)*N);
double *p2 = (double*)malloc(sizeof(double)*N);

#pragma offload target(mic) in(p1 : length(N)) out(p2 : length(N))
{
    // ... perform operations on p1[] and p2[]
}

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

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

```c
#pragma omp target
{
#pragma omp parallel for
    for(int i=0; i<size; i++)
        data[i] = 0;
}
```

Application runs on the host, but some parts of code and data are moved (“offloaded”) to the coprocessor. Scope-local scalars and stack arrays offloaded automatically.
#pragma omp target [clause[, clause[,...]]]

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

Link to reference manual.
 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
int* _Cilk_shared data; // Pointer to a virtual-shared array

int main() {
    // Working with pointer-based data is illustrated below:
    data = (_Cilk_shared int*)_Offload_shared_malloc(N*sizeof(float));
    _Offload_shared_free(data);
}

- Addresses of virtual-shared pointers identical on host and coprocessors
- Synchronized before and after offload, with page granularity
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.
Option 2: Symmetric hybrid MPI+OpenMP.

- MPI processes on hosts
- Native MPI processes on the coprocessor.
- Multi-threading with OpenMP.
§3. OPTIMIZATION POINTERS
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
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
- \textit{may or may not be better than \(-O2\)}
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.

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

Replace division with multiplication.

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

Use functions with Hardware support.

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

```c
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;
```
// 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);
// 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] = B[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
for (i = 1; i < n - 1; i++) {
    // 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] = B[n-1] + 1.0;
// 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 - 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: MAKE IT HAPPEN
```c
#include <cstdio>

int main()
{
    const int n=8;
    int i;
    int A[n] __attribute__((aligned(64)));
    int B[n] __attribute__((aligned(64)));

    // Initialization
    for (i=0; i<n; i++)
        A[i]=B[i]=i;

    // This loop will be auto-vectorized
    for (i=0; i<n; i++)
        A[i]+=B[i];

    // Output
    for (i=0; i<n; i++)
        printf("%2d %2d %2d\n", i, A[i], B[i]);
}
```

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
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)

\[
A[0:16] += B[32:16]; \quad // \text{B}[32]...\text{B}[47] \text{ added to A}[0]...\text{A}[15]
\]

▷ a stride (begin, length, stride)

\[
A[0:16:2] += B[32:16:4]; \quad // \text{B}[32],\text{B}[36]...\text{B}[92] \text{ added } \text{A}[0],\text{A}[2]...\text{A}[30]
\]

▷ Multi-dimensional arrays

\[
A[:][:] += B[:][:]; \quad // \text{Add B to A; arrays are of the same shape}
\]

Better than strided loops (e.g., this paper).
(formerly “elemental functions”)

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

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

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

Compiler will refuse to automatically vectorize this loop.
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:

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

Strip-mined implementation:

```c
const int STRIP=1024;
for (int ii = 0; ii < n; ii += STRIP) {
    for (int i = ii; i < ii+STRIP; i++) {
        // ... do work
    }
}
```
for (int i = ii; i < ii + tileSize; i++) { // Target for auto-vectorization

    // Newton's law of universal gravity
    const float dx = particle.x[j] - particle.x[i]; // x[j] is a const
    const float dy = particle.y[j] - particle.y[i]; // x[i] makes SIMD vector
    const float dz = particle.z[j] - particle.z[i];
    const float rr = 1.0f/sqrtf(dx*dx + dy*dy + dz*dz + softening);
    const float drPowerN32 = rr*rr*rr;

    // Calculate the net force
    Fx[i-ii] += dx * drPowerN32;
    Fy[i-ii] += dy * drPowerN32;
    Fz[i-ii] += dz * drPowerN32;
}
See this paper for more details
VECTOR-FRIENDLY DATA CONTAINERS
Unit-stride access is optimal:

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

Non-unit stride is slower:

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

Stochastic access may be vectorized (but not efficient):

```plaintext
for (int i = 0; i < n; i++)
    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:
Array char* p is n-byte aligned if ((size_t)p%n==0).

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

Why align: speed up vector load/stores, avoid false sharing (see Session 7), accelerate RDMA.
Compiler may implement peel and remainder loops:

\[
\text{for } (i = 0; i < n; i++) \quad A[i] = \ldots
\]
Creating Aligned Data Containers

▷ Data alignment on the stack

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

▷ Data alignment on the heap

```c
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`
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:

```c
// A - matrix of size (n x n)
// n is not a multiple of 16
float* A = _mm_malloc(sizeof(float)*n*n, 64);
for (int i = 0; i < n; i++)
    // A[i*n + 0] may be unaligned
    for (int j = 0; j < n; j++)
        A[i*n + j] = ...
```

Correct:

```c
// ... Padding inner dimension
int lda=n + (16-n%16); // lda%16==0
float* A = _mm_malloc(sizeof(float)*n*lda, 64);
for (int i = 0; i < n; i++)
    // A[i*lda + 0] aligned for any i
    for (int j = 0; j < n; j++)
        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 nocount`
- `#pragma ivdep`
- `restrict` qualifier and `-restrict` command-line argument
- `#pragma loop count`
- `-qopt-report` `-qopt-report-phase:vec`
- `-O[n]`
- `-x[code]`
-x[<code> code] instructs the compiler to target specific processor features, including instruction sets and optimizations.

<table>
<thead>
<tr>
<th>code</th>
<th>Target architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIC-AVX512</td>
<td>Intel Xeon Phi processors (KNL)</td>
</tr>
<tr>
<td>CORE-AVX512</td>
<td>Fugure Intel Xeon processors</td>
</tr>
<tr>
<td>CORE-AVX2</td>
<td>Intel Xeon processor E3 v3 family</td>
</tr>
<tr>
<td>CORE-AVX-I</td>
<td>Intel Xeon processor E3 v2, E5 v2 and E7 v2 family</td>
</tr>
<tr>
<td>AVX</td>
<td>Intel Xeon processor E3 and E5 family</td>
</tr>
<tr>
<td>SSE4.2</td>
<td>Intel Xeon processor 55XX, 56XX, 75XX and E7 family</td>
</tr>
<tr>
<td>host</td>
<td>architecture on which the code is compiled</td>
</tr>
</tbody>
</table>
True vector dependence makes vectorization impossible:

```c
float *a, *b;
for (int i = 1; i < n; i++)
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:

```c
void mycopy(int n,
    float* a, float* b) {
    for (int i = 0; i < n; i++)
a[i] = b[i];
}
```

```bash
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 ...
```
To ignore assumed vector dependence

```c
#pragma ivdep

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

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

Aliasing (true vector dependence) checked at runtime to choose the implementation.
Prevent multiversioning by using `#pragma ivdep`

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

```
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, \( A[i*n+j] += A[b*n+j] \)).
Programmer may promise to the compiler (under penalty of segmentation fault) that alignment has been taken care of:

```c
// Promising that A[i*lda + 0] is aligned for every i
// and the same for every other array in this loop
#pragma vector aligned
for (int j = 0; j < n; j++)
    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
```c
#include <omp.h>
#include <cstdio>

int main(){
  // This code is executed by only 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
Utilize cores: run multiple threads/processes (MIMD)
Utilize vectors: each thread (process) issues vector instructions (SIMD)
Option 1: Partitioning data set between threads/processes

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

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

int main() {
    const int n = 1000;
    int total = 0;

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        total = total + i; // Race condition
    }

    printf("total=%d (must be %d)\n", total, ((n-1)*n)/2);
}
```

▶ Occurs when 2 or more threads access the same memory address, and at least one of these accesses is for writing
Correct and efficient code:

```c
int total = 0;
#pragma omp parallel
{
    int total_thr = 0;
    #pragma omp for
    for (int i=0; i<n; i++)
        total_thr += i;

    #pragma omp atomic
    total += total_thr;
}
```

Thread-private partial sums:

- Thread 0: x0+=0, x0+=1, x0+=2, ..., x0+=199
- Thread 1: x1+=200, x1+=201, x1+=202, ..., x1+=399
- Thread 3: x3+=600, x3+=601, x3+=602, ..., x3+=799
- Thread 4: x4+=800, x4+=800, x4+=800, ..., x4+=999

Reduction protected with mutexes:

- total+=x0
- total+=x1
- total+=x2
- total+=x3
- atomic total+=x4

Total = 499500
Computing a histogram ($m << n$):

```c
void Histogram(
    // Ages, values from 0.0f to 100.0f:
    const float* age,
    // Size of array age, $n=100000000$:
    const int n,
    // Output: counts in groups:
    int* const hist,
    // Size of array hist, $m=5$:
    const int m,
    const float group_width) {
    for (int i = 0; i < n; i++) {
        const int j = int(age[i]/group_width);
        hist[j]++;
    }
}
```

- Vector dependence in `hist[j]++`
- Strip-mine or use conflict detection
Using Reduction Instead of Synchronization

Vectorized Serial Code (Strip-Mining)

Vectorized Parallel Code (Atomic Operations)

Vectorized Parallel Code (Private Variables)

Performance, billion values/s (higher is better)

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210 (KNL)
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 sharing a cache line between threads

```
const int paddingBytes = 64;
const int paddingElements = paddingBytes / sizeof(int);
const int mPadded = m + (paddingElements - m % paddingElements);
int hist_containers[nThreads][mPadded]; // New container
```
PADDOING TO AVOID FALSE SHARING

Parallel Code (private variables) False Sharing (no padding) Padded to 64 bytes Padded to 128 bytes Padded to 256 bytes

Performance, billion values/s (higher is better)

- Intel Xeon processor E5-2697 V2
- Intel Xeon Phi coprocessor 7120P (KNC)
- Intel Xeon Phi processor 7210 (KNL)

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210 (KNL)

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

\[ S_i = \sum_{j=0}^{n} M_{ij}, \ i = 0 \ldots m. \]  

- \( m=4 \) 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

```c
void sum_unoptimized(const int m, const int n, long* M, long* s) {
    #pragma omp parallel for
    for (int i=0; i<m; i++) { // m=4
        long total=0;
        #pragma vector aligned
        for (int j=0; j<n; j++) // n=100000000
            total+=M[i*n+j];
        s[i]=total; }
}
```
```c
void sum_stripmine(const int m, const int n, long* M, long* s){
    const int STRIP=1024;
    assert(n%STRIP==0);
    s[0:m]=0;
    #pragma omp parallel
    {
        long total[m]; total[0:m]=0;
        #pragma omp for collapse(2) schedule(guided)
        for (int i=0; i<m; i++)
            for (int jj=0; jj<n; jj+=STRIP)
                #pragma vector aligned
                for (int j=jj; j<jj+STRIP; j++)
                    total[i]+=M[i*n+j];
        for (int i=0; i<m; i++)                // Reduction
            #pragma omp atomic
            s[i]+=total[i];
    } }
```
Dealing with Insufficient Parallelism

- Unoptimized Parallel inner loop Collapse nested loops Strip-mine and collapse

Performance, GB/s (higher is better)

- Intel Xeon processor E5-2697 V2
- Intel Xeon Phi coprocessor 7120P (KNC)
- Intel Xeon Phi processor 7210P (KNL), DDR4
- Intel Xeon Phi processor 7210P (KNL), MCDRAM

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210P (KNL), DDR4
Intel Xeon Phi processor 7210P (KNL), MCDRAM

Performance, GB/s (higher is better)

- Unoptimized
- Parallel inner loop
- Collapse nested loops
- Strip-mine and collapse

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210P (KNL), DDR4
Intel Xeon Phi processor 7210P (KNL), MCDRAM

Performance, GB/s (higher is better)

- Unoptimized
- Parallel inner loop
- Collapse nested loops
- Strip-mine and collapse

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210P (KNL), DDR4
Intel Xeon Phi processor 7210P (KNL), MCDRAM

Performance, GB/s (higher is better)

- Unoptimized
- Parallel inner loop
- Collapse nested loops
- Strip-mine and collapse

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210P (KNL), DDR4
Intel Xeon Phi processor 7210P (KNL), MCDRAM

Performance, GB/s (higher is better)

- Unoptimized
- Parallel inner loop
- Collapse nested loops
- Strip-mine and collapse

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210P (KNL), DDR4
Intel Xeon Phi processor 7210P (KNL), MCDRAM

Performance, GB/s (higher is better)

- Unoptimized
- Parallel inner loop
- Collapse nested loops
- Strip-mine and collapse

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210P (KNL), DDR4
Intel Xeon Phi processor 7210P (KNL), MCDRAM

Performance, GB/s (higher is better)
MULTI-THREADING: FINE-TUNING
Loop Scheduling Modes in OpenMP

Scheduling | Threads | Iterations
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>static 0</td>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16 17 18 19 20 21 22 23</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24 25 26 27 28 29 30 31</td>
</tr>
</tbody>
</table>

Scheduling | Threads | Iterations
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>static,1 0</td>
<td></td>
<td>0 4 8 12 16 20 24 28</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1 5 9 13 17 21 25 29</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2 6 10 14 18 22 26 30</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3 7 11 15 19 23 27 31</td>
</tr>
</tbody>
</table>

Scheduling | Threads | Iterations
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic,2 0</td>
<td></td>
<td>0 7 10 12 17 ...</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1 4 9 14 18 ...</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2 5 8 11 15 ...</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3 6 9 13 16 ...</td>
</tr>
</tbody>
</table>

Scheduling | Threads | Iterations
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic,2 0</td>
<td></td>
<td>4 5 8 9 14 15 ...</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2 3 12 13 18 19 ...</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4 5 8 9 14 15 ...</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6 7 12 20 21 30 31 ...</td>
</tr>
</tbody>
</table>

Scheduling | Threads | Iterations
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>guided,2 0</td>
<td></td>
<td>0 1 2 3 16 17 24 25</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4 5 6 7 20 21 25 30</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8 9 10 11 18 19 26 28</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12 13 14 15 22 23 27 31</td>
</tr>
</tbody>
</table>

Scheduling | Threads | Iterations
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>guided,2 0</td>
<td></td>
<td>4 5 6 7 20 21 26 27</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8 9 10 11 18 19 28 29</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12 13 14 15 22 23 30 31</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12 13 14 15 22 23 30 31</td>
</tr>
</tbody>
</table>
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

```
#pragma omp parallel
{
    #pragma omp parallel
    {
        // ...
    }
}
```

▷ Tune granularity of parallelism
▷ Improve resource sharing in NUMA systems
CACHE AND MEMORY ACCESS
Theoretical estimates, Intel Xeon E5-2697 V3 processor

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

Required Data Rate = 1.2 TFLOP/s × 8 bytes ≈ 10 TB/s

Memory Bandwidth = η × 2 × 59.7 ≈ 0.1 TB/s

Ratio = 10/0.1 ≈ 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

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

$N = \text{data size}$
Roofline model: theoretical peak

More on roofline model: Williams et al.
STREAMING VERSUS RANDOM ACCESS

Measured Memory Performance

- Intel Xeon E5-2697 v2 dual-socket CPU
- Intel Xeon Phi 7120P coprocessor

Bandwidth, GB/s

<table>
<thead>
<tr>
<th>Test Case</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAM TRIAD</td>
<td>177 GB/s</td>
</tr>
<tr>
<td>TRIAD on random 4 kB Blocks</td>
<td>60 GB/s</td>
</tr>
<tr>
<td>TRIAD on random 1 kB Blocks</td>
<td>16 GB/s</td>
</tr>
</tbody>
</table>

Intel Xeon E5-2697 v2 dual-socket CPU

Measured Memory Performance

- STREAM TRIAD: 177 GB/s
- TRIAD on random 4 kB Blocks: 60 GB/s
- TRIAD on random 1 kB Blocks: 16 GB/s

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).
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**

\[ C = AB \quad \iff \quad C_{ij} = \sum_{k=0}^{n-1} A_{ik}B_{kj} \]

**Before:**

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

**After:**

```c
#pragma omp parallel for
for (int i = 0; i < n; i++)
    #pragma vector aligned
    for (int k = 0; k < n; k++)
        for (int j = 0; j < n; j++)
            C[i*n+j] += A[i*n+k] * B[k*n+j];
```
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.
Re-use data in cache by fusing loops in a data processing pipeline

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

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

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

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

Potential positive side-effect: less data to carry between stages, reduced memory footprint, improved performance (see, e.g., this paper).
Original:
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];

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

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

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%
```c
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
```c
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
```c
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
```
for (int i = 0; i < m; i++)  // Original code:
    for (int j = 0; j < n; j++)
        compute(a[i], b[j]);  // Memory access is unit-stride in j

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

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

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

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

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

Vector b

Access pattern to vector b

Tiled Algorithm (serial order)

Recursive (serial order)
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
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

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

// Initializing from parallel region for better performance
#pragma omp parallel for
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        A[i*m + j] = 0.0f;
```
Vectorized Parallel Code
(Private Variables)

Parallel Initialization
(First-Touch Allocation)

Performance, billion values/s (higher is better)

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210 (KNL)

Intel Xeon processor E5-2697 V2
Intel Xeon Phi coprocessor 7120P (KNC)
Intel Xeon Phi processor 7210 (KNL)
EXAMPLE: MATRIX TRANSPOSITION

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

See also this paper.
Before:

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

After:

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

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

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

(2)

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

Non-optimal performance due to inefficient cache use
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];
    }
}
const long iTile = 64L;    assert(m%iTile == 0);
const long jTile = 4096L; assert(n%jTile == 0);
#pragma omp parallel
{
  double temp_c[m] __attribute__((aligned(64))); temp_c[:] = 0;
#pragma omp for collapse(2)
  for (long ii = 0; ii < m; ii += iTile)
    for (long jj = 0; jj < n; jj += jTile)
      for (long i = ii; i < ii+iTile; i++)
        #pragma vector aligned
        for (long j = jj; j < jj+jTile; j++)
          temp_c[i] += A[i*n+j] * b[j];
  for(long i = 0; i < m; i++) {
    #pragma omp atomic
    c[i] += temp_c[i];
  } } }
void RecursMultiply(const double* const A, const double* const b, 
                      double* const c, const long n, const long m, const long lda){
    const long jThreshold = 8192L; assert(n%jThreshold == 0);
    const long iThreshold = 64L; assert(m%iThreshold == 0);
    if ((m<=iThreshold) && (n<=jThreshold)) { // Recursion threshold
        // .... Base Case: Compute the result inside the tile ... //
    } else { // Recursive divide-and-conquer
        if (m*jThreshold > n*iThreshold) { // Split i-wise
            double c1[m/2] __attribute__((aligned(64)));
            #pragma omp task
            { RecursMultiply(&A[0*lda + 0], &b[0], c1, n, m/2, lda); }
            double c2[m/2] __attribute__((aligned(64)));
            RecursMultiply(&A[(m/2)*lda + 0], &b[m/2], c2, n, m/2, lda);
            #pragma omp taskwait
            c[0:m/2] += c1[0:m/2]; c[m/2:m/2] += c2[0:m/2]; // Reduction
        } else { // .... Split j-wise .... // }
    } }
Performance of Matrix Vector Multiplication

Memory Traffic Optimization in Matrix-Vector Multiplication

- Unoptimized
- Tiled j-loop
- Tiled i- and j-loops
- Recursive Cache-Oblivious Method

Performance, GFLOP/s (higher is better):
- Unoptimized: 16.0 ± 0.1
- Tiled j-loop: 19.2 ± 0.1
- Tiled i- and j-loops: 20.3 ± 0.2
- Recursive Cache-Oblivious Method: 21.2 ± 0.3

Host System
Intel Xeon Phi Coprocessor

Performance of Matrix Vector Multiplication

- Unoptimized
- Tiled j-loop
- Tiled i- and j-loops
- Recursive Cache-Oblivious Method

Performance, GFLOP/s (higher is better):
- Unoptimized: 30.7 ± 0.7
- Tiled j-loop: 32.5 ± 0.2
- Tiled i- and j-loops: 35.6 ± 0.1
- Recursive Cache-Oblivious Method: 42.1 ± 0.2
COMMUNICATION CONTROL
```c
#include "mpi.h"
#include <cstdio>

int main (int argc, char *argv[]) {
    MPI_Init (&argc, &argv);  // Initialize MPI environment
    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!
", rank, name);
    if (rank == 0) printf("MPI World size = %d processes\n", size);
    MPI_Finalize ();  // Terminate MPI environment
}
```
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
1. Compile and link with the MPI wrapper of the compiler:
   - mpiicc for C,
   - mpiicpc for C++,
   - mpiifort for Fortran 77 and Fortran 95.

2. Set up MPI environment variables and \texttt{I\_MPI\_MIC=1}

3. NFS-share or copy the MPI library and the application executable to the coprocessors

4. Launch with the tool \texttt{mpirun}
   - Colon-separated list of executables and hosts (argument \texttt{-host hostname}),
   - Alternatively, use the machine file to list hosts
   - Coprocessors have hostnames defined in /etc/hosts
if (rank == sender) {
    char outgoingMsg[messageLength];
    strcpy(outgoingMsg, "Hi There!");
    MPI_Send(&outgoingMsg, messageLength, MPI_CHAR, receiver, tag, MPI_COMM_WORLD);
}

} else if (rank == receiver) {

    char incomingMsg[messageLength];
    MPI_Recv (&incomingMsg, messageLength, MPI_CHAR, sender, 
                tag, MPI_COMM_WORLD, &stat);
    printf ("Received message with tag %d: '%s'
", tag, incomingMsg);

}
Collective Communication: Broadcast

```c
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm );
```
COLLECTIVE COMMUNICATION: SCATTER

```c
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
int recvnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
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).
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
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
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)); }

Queue of work-items:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | ... |

Host

MPI rank 0

MPI rank 1

Xeon Phi 1

MPI rank 2

Xeon Phi 2

MPI rank 3
Static Load Balancing: Parameter Tuning

Value of parameter $\alpha$

Baseline (even load)

Optimal load balance

Performance ($10^9$ values/s)
if (myRank == 0)  // Boss’s branch
    DistributeWork(nOptions, option, mpiWorldSize);
else  // Workers’ branch
    ReceiveWork(option, payoff, myRank);
§4. PREPARING FOR INTEL XEON PHI PROCESSORS
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

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

HOW SERIES “KNIGHTS LANDING”:

PROGRAMMING AND OPTIMIZATION FOR INTEL XEON PHI X200 FAMILY

Free 2-hour video course

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

```bash
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 ≥ 4.9.1 supports AVX-512 instruction set.

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

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

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

Get assembly:

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

- **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

Start

Does your Application fit in 16 GB?

Yes

No

Can you partition ≤16 GB of BW-critical memory?

Yes

No

numactl

▷ Run the whole program in HBM
▷ No code modification

Memkind

▷ Selectively allocate data to HBM
▷ Add memkind calls

Cache mode

▷ Allow the chip to figure out how to use HBM
▷ No code modification

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 ... //
```
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:

```fortran
REAL, ALLOCATABLE :: A(:)
!DEC$ ATTRIBUTES FASTMEM :: A
ALLOCATE (A(1:N))
```
To compile C/C++ applications:

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

To compile Fortran applications:

```bash
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/

Memory benchmark on Intel Xeon Phi processor 7210

- On-platform memory (DDR4)
- On-package high-bandwidth memory (MCDRAM)

Using High-Bandwidth Memory
LEVERAGING CLUSTERING MODES
Tag Directory (TD) and memory reside in the same quadrant.
Cores appear as 4 (or 2) NUMA nodes.
How to Use Clustering Modes

### Nested OpenMP

```cpp
#pragma omp parallel
{
    // ...
    #pragma omp parallel
    {
        // ...
    }
    // ...
}
```

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

### MPI+OpenMP

```cpp
stat = MPI_Init();
// ...
#pragma omp parallel
{
    // ...
}
// ...
MPI_Finalize();
```

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

Learn more: colfaxresearch.com/knl-numa
COPROCESSOR AND KNL-F
KNLF: KNL with Fabric

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

KNL Coprocessor

- PCIe add-in card
  - Requires host
- Multiple KNLs in a system
§5. INTEL LIBRARIES
INTEL MKL: THE MAGIC PILL
Intel® Math Kernel Library (MKL) — standard mathematical functions optimized for Intel architecture.

### Dense Linear Algebra
- BLAS + PBLAS
- LAPACK + ScaLAPACK
- Extended eigensolver

### Sparse Linear Algebra
- SpBLAS
- Iterative, direct solvers
- Preconditioners

### Fourier Transform
- Multi-threaded
- Cluster mode
- 1D and multi-dimensional

### Numerical Analysis
- Partial differential equations
- Nonlinear optimization
- Data fitting. Vector math.

### Statistics and Probability
- Random number generators
- Convolution and correlation
- Summary statistics

### Machine Learning
- DNN Primitives:
  - Convolution. Inner product.
  - ReLU, LRNC, etc.

### New
- DNN Primitives:
  - Convolution. Inner product.
  - ReLU, LRNC, etc.
$2 \cdot 10^5$ FFTs of size 2048. See **HOW “KNL”** for details.
INTEL PYTHON: WRAP AROUND ANYTHING
Intel Distribution for Python → Intel Math Kernel Library → Intel DAAL

SciPy → NumPy → matplotlib → theano → Caffe

Portal: software.intel.com/intel-distribution-for-python. See also: CR paper.
Intel on Knights Landing Processors (N=5000)

- CPython, SciPy
- CPython, NumPy
- Intel Python, SciPy

Relative Performance

<table>
<thead>
<tr>
<th>Performance</th>
<th>LU</th>
<th>Cholesky</th>
<th>Singular Value</th>
<th>DGEMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>3.5</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>29.0</td>
<td></td>
<td>3.6</td>
<td>1.1</td>
<td>7.0</td>
</tr>
<tr>
<td>17.0</td>
<td></td>
<td></td>
<td>8.3</td>
<td></td>
</tr>
<tr>
<td>154.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Portal: 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.
### Analysis

- Low Order Moments
- Quantile
- Correlation and Variance
- Cosine Distance Matrix
- Correlation Distance Matrix
- K-Means Clustering
- Principal Component Analysis
- Cholesky Decomposition

### Training & prediction

- Regression
- QR Decomposition
- Expectation-Maximization
- Multivariate Outlier Detection
- Univariate Outlier Detection
- Association Rules
- Kernel Functions
- Quality Metrics
- Multi-Class Classifier

Portal: DAAL page. See also: intro article, CR papers.
AI ON IA
# INTEL MKL AND INTEL MKL-DNN

## Deep Learning Frameworks

- Caffe
- BVLC
- theano
- Microsoft CNTK
- Google TensorFlow
- torch

## Intel® MKL

**Intel® Math Kernel Library**

- Led Kernel
- Math Kernel Library for Deep Neural Networks

**Intel® Xeon® Processor**

## Intel® MKL-DNN

**Intel® Xeon Phi™ Processor**

**FPGA**

<table>
<thead>
<tr>
<th>Intel® MKL</th>
<th>Intel® MKL-DNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNN primitives + wide variety of other math functions</td>
<td>DNN primitives</td>
</tr>
<tr>
<td>C DNN APIs (C++ future)</td>
<td>C/C++ DNN APIs</td>
</tr>
<tr>
<td>Binary distribution</td>
<td>Open source DNN code¹</td>
</tr>
<tr>
<td>Free community license. Premium support available as part of Intel® Parallel Studio XE</td>
<td>Apache 2.0 license</td>
</tr>
<tr>
<td>Broad usage DNN primitives; not specific to individual frameworks</td>
<td>Multiple variants of DNN primitives as required for framework integrations</td>
</tr>
<tr>
<td>Quarterly update releases</td>
<td>Rapid development ahead of Intel MKL releases</td>
</tr>
</tbody>
</table>

¹ GEMM building blocks are binary.

*slide credit: Intel corp.*
The next wave of deep learning applications (The Next Platform)

Code Modernization Speeds Python and Other Machine Learning Packages (TechEnablement)

Intel Python (Intel)

Intel® MKL-DNN (GitHub)

Intel Caffe (GitHub)

Intel Theano (GitHub)

Intel Torch (GitHub)
§6. CLOSING WORDS
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.
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