HOW TO EVALUATE YOUR APPLICATION
EVALUATING YOUR APPLICATIONS

Can your workload scale to over 100 threads?

NO

YES

Can your workload benefit from large vectors?

NO

YES

Can your workload benefit from more memory bandwidth?

NO

YES

Use Intel® Xeon Phi™ coprocessors for applications that scale with:

- Threads
- Vectors
- Memory Bandwidth

Click to see Animation video on
Exploiting Parallelism on Intel Xeon Phi Coprocessors

Representative example workload for illustration purposes only
BREAKTHROUGH PROGRAMMING EFFICIENCY
For Accelerating Highly Parallel Applications

Maintain a single code base
• Software that runs on Intel® Xeon® processors also runs on Intel® Xeon Phi™ coprocessors

Use familiar tools
• No need to learn new tools, languages, or development models

Optimize code just once
• Optimizations for Intel® Xeon Phi™ coprocessors also boost performance for Intel Xeon processors

Preserve your investment
• Don’t reinvent the wheel—accelerate performance for existing applications
FLEXIBLE USAGE MODELS

- **SOURCE CODE**
  - Compilers, Libraries and Parallel Models

- **Execution Models**
  - **Multicore Only (90% of applications)**
  - **Multicore Hosted with Manycore Offload**
  - **Symmetric**
  - **Manycore Only**

- **Percentage of Code**
  - Highly parallel code
  - Serial and moderately parallel code

- **XEON(s)**
  - MAIN ()
  - XEON(s)

- **Xeon Phi(s)**
  - Xeon Phi(s)

- **Intel Confidential**
## Examples of Highly Parallel Market Segments & Applications
Click on links for more information

<table>
<thead>
<tr>
<th>Sub-Segments:</th>
<th>Applications/Workloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public Sector</td>
<td>HPL, HPCC, NPB, LAMMPS, QCD, BQCD, GROMACS</td>
</tr>
<tr>
<td><strong>Energy</strong> (including Oil &amp; Gas)</td>
<td>RTM (Reverse Time Migration), WEM (Wave Equation Migration)</td>
</tr>
<tr>
<td>Climate Modeling and Weather Simulation</td>
<td>WRF, HOMME</td>
</tr>
<tr>
<td><strong>Financial Analysis</strong></td>
<td>Monte Carlo, Black-Scholes, Binomial model, Heston model</td>
</tr>
<tr>
<td><strong>Life Sciences</strong> (Molecular Dynamics, Gene Sequencing, Bio-Chemistry)</td>
<td>LAMMPS, NAMD, AMBER, HMMER, BLAST, QCD, CHARMM, BQCD, GROMACS</td>
</tr>
<tr>
<td><strong>Manufacturing</strong> (CAD/CAM/CAE/CFD/EDA)</td>
<td>Implicit, Explicit Solvers</td>
</tr>
<tr>
<td>Digital Content Creation</td>
<td>Ray Tracing, Animation, Effects</td>
</tr>
<tr>
<td>Software Ecosystem</td>
<td>Tools, Middleware</td>
</tr>
</tbody>
</table>

**Mix of ISV and End User Development**
A GROWING ECOSYSTEM:
Developing today on Intel® Xeon Phi™ coprocessors

Shown at SC’12, November 2012
## Intel® Xeon Phi™ Coprocessor: Performance Proof-points

Click on links for more information (SC12)

<table>
<thead>
<tr>
<th>Energy</th>
<th>Academic/ Government Labs</th>
<th>Financial Services</th>
<th>Manufacturing, Digital Content Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleware – 8th order Isotropic RTM (updated)</td>
<td>ASKAP – tHogbomClean (updated)</td>
<td>Monte Carlo (updated)</td>
<td>Altair - Radioss (finite element solver)</td>
</tr>
<tr>
<td>Sinopec iCluster PSDM (updated)</td>
<td>Sandia – miniFE (finite element solver)</td>
<td>Black-Scholes (updated)</td>
<td>Embree 2.0</td>
</tr>
<tr>
<td>CNPC BGP Geoeast PSTM</td>
<td>ZIB Ising 3D</td>
<td>Aneo HPClib</td>
<td>NEC – Realtime Super Program</td>
</tr>
<tr>
<td>CGG: WEM</td>
<td>Jefferson Labs Lattice QCD (updated)</td>
<td>LRZ/TUM SG++</td>
<td></td>
</tr>
<tr>
<td>Intel: 3DFD TTI Proxy (new)</td>
<td>CAS IPE MD Simulation</td>
<td>BQCD (updated)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gromacs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CAS – CNIC Wigeon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CAS – ICT FDTD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRF (updated)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AWE Cloverleaf</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TACC LB3d</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI-HMMER (updated)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Intel® Xeon Phi™ Coprocessor: Performance Proof-points

Click on links for more information (ISC13)

<table>
<thead>
<tr>
<th>Energy</th>
<th>Academic/Government Labs</th>
<th>Financial Services</th>
<th>Manufacturing, Digital Content Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleware – 8th order Isotropic RTM (updated)</td>
<td>ASKAP – thHogbomClean (updated)</td>
<td>Monte Carlo (updated)</td>
<td>Altair - Radioss (finite element solver)</td>
</tr>
<tr>
<td>Sinopec iCluster PSDM (updated)</td>
<td>Sandia – miniFE (finite element solver)</td>
<td>Black-Scholes (updated)</td>
<td>Embree 2.0</td>
</tr>
<tr>
<td>CNPC BGP Geoeast PSTM</td>
<td>ZIB Ising 3D</td>
<td>Aneo HPClib</td>
<td>NEC – Realtime Super Resolution Program</td>
</tr>
<tr>
<td>CGG: WEM</td>
<td>Jefferson Labs Lattice QCD (updated)</td>
<td></td>
<td>OpenFOAM (Symmetrec)</td>
</tr>
<tr>
<td>Intel: 3DFD TTI Proxy (new)</td>
<td>LRZ/TUM SG++</td>
<td></td>
<td>StartCD (Symmetrec)</td>
</tr>
<tr>
<td></td>
<td>CAS IPE MD Simulation</td>
<td></td>
<td>STARTCCM+ (Symmetrec)</td>
</tr>
<tr>
<td></td>
<td>BQCD (updated)</td>
<td></td>
<td>ANSYS-Fluent (Symmetrec)</td>
</tr>
<tr>
<td></td>
<td>Gromacs</td>
<td></td>
<td>Overflow (NASA, Symmetrec)</td>
</tr>
<tr>
<td></td>
<td>CAS – CNIC Wigeon</td>
<td></td>
<td>Numeca (native)</td>
</tr>
<tr>
<td></td>
<td>CAS – ICT FDTD</td>
<td></td>
<td>Phoenics (native)</td>
</tr>
<tr>
<td></td>
<td>WRF (updated)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AWE Cloverleaf</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TACC LB3d</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI-HMMER (updated)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CFD Application

- **ANSYS Fluent status:**
  - ansys Fluent has been trying to do a “beta” release of Fluent for Xeon Phi.

- **OpenFOAM:**
  - The current status is that OpenFOAM is about 2x slower on KNC compared to SNB-EP 2S, 1600 MHz. This is single-node, so no Infiniband involved.
  - open source development team did not want to do anything to customize for Phi.

- **Overflow (NASA): NASA Overflow has done the best of known structured solvers.**

- **StartCD|StartCCM+: Ongoing.**
Case Study: TACC -- LBM
Porting to the Intel Xeon Phi: Opportunities and Challenges

Carlos Rosales

carlos@tacc.utexas.edu
Motivation and main algorithm characteristics

THE LATTICE BOLTZMANN METHOD
Programming Models

• Traditional Cluster
  – Pure MPI and MPI+X
    • X: OpenMP, TBB, Cilk+, OpenCL, ...

• Native Phi
  – Use one Phi and run OpenMP or MPI programs directly

• MPI tasks on Host and Phi
  – Treat the Phi (mostly) like another host
    • Pure MPI and MPI+X

• MPI on Host, Offload to Xeon Phi
  – Targeted offload through OpenMP extensions
  – Automatically offload some library routines with MKL
Code Structure

- **Collision**: Local calculation. Most of the math takes place here.
- **PostCollision**: Boundary corrections for outward f values
- **Stream**: Move collision data along velocity directions
- **PostStream**: Boundary corrections for inward f and g
Governing equations

Navier-Stokes
\[ \frac{\partial (n\mathbf{u})}{\partial t} + \nabla \cdot (n\mathbf{u}\mathbf{u}) = -\nabla P + \mu \nabla^2 \mathbf{u} + \mathbf{F}_b \]

Mass Conservation
\[ \frac{\partial n}{\partial t} + \nabla \cdot (n\mathbf{u}) = 0 \]

Cahn-Hilliard
\[ \frac{\partial \phi}{\partial t} + \nabla \cdot (\phi \mathbf{u}) = \theta_M \nabla^2 \mu_\phi \]

Chemical Potential
\[ \mu_\phi = 4\alpha (\phi^3 - \phi^2) - \kappa \nabla^2 \phi \]
Challenges

• Industrially Relevant
  – Density and viscosity ratios
  – Volume fractions
  – Poly-disperse suspensions

• Capture the Physics
  – High time/space resolution
  – Large data sets
    • Storage
    • Visualization
PORTING THE LBM CODE
Setting Expectations

- Performance ratios between one mic and two E5-2680 sockets:
  - In FP Ops: $1074 / 345 = 3.1x$
  - In Memory BW (approx): $180 / 80 = 2.2x$

- So, the best acceleration I can expect is:
  - 3.1x for fp bound code
  - 2.2x for memory bw bound code

- And most likely I will get something in between these two results.
Original Code

- Multiphase LBM based on Free Energy formulation
- Single 4D array for f and single 4D array for g
- Double-buffered to avoid issues in Stream
- Parallelized using MPI
- Some common optimizations present
  - Fused collision-stream step for g
  - Neighbors calculated instead of read from array
  - Bidirectional MPI exchanges used
First Port to MIC

- Removed all MPI calls
- Included OMP in collision function
  - Most math is performed here
  - Takes 95% of run time on host
- Parallelized outermost loop (in z direction)
  - Innermost loop not broken – vectorization
  - Must be careful with number of threads
    - When using multiple threads per core, if zmax / OMP_NUM_THREADS not a multiple of the number of cores performance will be degraded
- No vector hints used
# First Port Results

**MIC**
- OMP_NUM_THREADS = 240
- KMP_AFFINITY = balanced,granularity=fine
- Performance - 7.7 MLUPS

**CPU**
- OMP_NUM_THREADS = 16
- KMP_AFFINITY = compact,granularity=fine
- Performance - 32.9 MLUPS

Domain Size = 240x240x240x240  
Number of Iterations = 100  
MLUPS = Millions of Lattice Updates Per Second

Disappointing!!  
CPU is roughly 4X FASTER than MIC  
WHY??
Finding the Hotspots

- Instrumented the code
- Timings revealed bottleneck moved from collision to stream
- Single thread on MIC too weak to achieve appropriate memory bandwidth
- Turns out profile from Host is misleading and does not predict behavior on the MIC
- Should have guessed that 😞

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (sec)</th>
<th>% Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collision</td>
<td>44.4</td>
<td>24.8</td>
</tr>
<tr>
<td>PostCollision</td>
<td>1.1</td>
<td>0.6</td>
</tr>
<tr>
<td>Stream</td>
<td>123.2</td>
<td>68.9</td>
</tr>
<tr>
<td>PostStream</td>
<td>10.3</td>
<td>5.7</td>
</tr>
</tbody>
</table>
Fixing the Bandwidth Bottleneck

- Added OMP to all heavy data movement sections
  - PostCollision
  - Stream
  - PostStream
- Re-run test case with identical settings
- Stream function performance improved by factor 89x
- Bottleneck moved back to Collision
- Both MIC and CPU performance improved
  - MIC improved by 3.9x
  - CPU improved by 1.6x
- Smaller improvement on CPU due to heavier threads

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (sec)</th>
<th>% Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collision</td>
<td>44.1</td>
<td>96.3</td>
</tr>
<tr>
<td>PostCollision</td>
<td>0.03</td>
<td>0.1</td>
</tr>
<tr>
<td>Stream</td>
<td>1.4</td>
<td>3.0</td>
</tr>
<tr>
<td>PostStream</td>
<td>0.3</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Performance (MIC) = 29.7 MLUPS
Performance (CPU) = 52.4 MLUPS

CPU is still FASTER by a factor 1.8x
What Could Be Missing?

- Adding OMP has only given us enough performance to roughly match a single Xeon E5 socket.
- Much of the performance of the MIC is achieved via the 512-bit vector width.
- We could be loosing up to a factor 8x in performance.
- It is time to look at the vectorization of the code.
Vector Reports

- Using the Intel compiler option -vec-report3 we find:

  collision.f90(40): (col. 6) remark: LOOP WAS VECTORIZED.
  collision.f90(40): (col. 6) remark: PEEL LOOP WAS VECTORIZED.
  collision.f90(40): (col. 6) remark: REMAINDER LOOP WAS VECTORIZED.
  collision.f90(39): (col. 4) remark: loop was not vectorized: not inner loop.
  collision.f90(38): (col. 2) remark: loop was not vectorized: not inner loop.
  collision.f90(51): (col. 6) remark: LOOP WAS VECTORIZED.
  collision.f90(51): (col. 6) remark: REMAINDER LOOP WAS VECTORIZED.
  collision.f90(50): (col. 4) remark: loop was not vectorized: not inner loop.
  collision.f90(49): (col. 2) remark: loop was not vectorized: not inner loop.
  collision.f90(95): (col. 6) remark: loop was not vectorized: existence of vector dependence.
  collision.f90(164): (col. 4) remark: vector dependence: assumed FLOW dependence between lbmparams_mp_g_ line 164 and lbmparams_mp_g_ line 100.
  collision.f90(164): (col. 4) remark: vector dependence: assumed FLOW dependence between lbmparams_mp_g_ line 164 and lbmparams_mp_g_ line 117.
  collision.f90(164): (col. 4) remark: vector dependence: assumed FLOW dependence between lbmparams_mp_g_ line 164 and lbmparams_mp_g_ line 164.
  collision.f90(94): (col. 4) remark: loop was not vectorized: not inner loop.
  collision.f90(93): (col. 2) remark: loop was not vectorized: not inner loop.
Vector Reports

- Using the Intel compiler option –vec-report3 we find:

```
collision.f90(40): (col. 6) remark: LOOP WAS VECTORIZED.
collision.f90(40): (col. 6) remark: PEEL LOOP WAS VECTORIZED.
collision.f90(40): (col. 6) remark: REMAINDER LOOP WAS VECTORIZED.
collision.f90(39): (col. 4) remark: loop was not vectorized: not inner loop.
collision.f90(38): (col. 2) remark: loop was not vectorized: not inner loop.
collision.f90(51): (col. 6) remark: LOOP WAS VECTORIZED.
collision.f90(51): (col. 6) remark: REMAINDER LOOP WAS VECTORIZED.
collision.f90(50): (col. 4) remark: loop was not vectorized: not inner loop.
collision.f90(49): (col. 2) remark: loop was not vectorized: not inner loop.
collision.f90(95): (col. 6) remark: loop was not vectorized: existence of vector dependence.
collision.f90(164): (col. 4) remark: vector dependence: assumed FLOW dependence between lbmparams_mp_g_ line 164 and lbmparams_mp_g_ line 100.
collision.f90(164): (col. 4) remark: vector dependence: assumed FLOW dependence between lbmparams_mp_g_ line 164 and lbmparams_mp_g_ line 117.
collision.f90(164): (col. 4) remark: vector dependence: assumed FLOW dependence between lbmparams_mp_g_ line 164 and lbmparams_mp_g_ line 164.
collision.f90(94): (col. 4) remark: loop was not vectorized: not inner loop.
collision.f90(93): (col. 2) remark: loop was not vectorized: not inner loop.
```

This is not good: Most of our math heavy operations are performed in the loop at line 95.
Testing Vectorization

- We compiled two version of the code:
  -O3 -openmp -mmic
  -O3 -openmp -mmic -novec

- MIC Performance (vec) = 29.7 MLUPS

- MIC Performance (novec) = 27.9 MLUPS

- This merely confirms what the vector report told us: no vectorization in the main loop of collision.

- Using IVDEP directives did not change the result.

- Because of the use of doubly-buffered arrays there should not be a real data dependence in the loop.

- But the compiler does not seem to realize this (changes during optimization?)
From AOS to SOA

\[ g(i, j, k, vel, buf) \rightarrow g_0(m), g_1(m), \ldots, g_{18}(m) \quad m = i + NX \cdot (j + NY \cdot k) + \text{offset} \]

\[ f(i, j, k, vel, buf) \rightarrow f_0(m), f_1(m), \ldots, f_6(m) \quad m = i + NX \cdot (j + NY \cdot k) + \text{offset} \]

- This maintains a doubly-buffered array but in a much simpler form
- Compiler is able to vectorize the main collision loop (-vec-report3)
- MIC performance increases by 3.3x to 127.7 MLUPS
- CPU performance decreases 4% to 50.1 MLUPS
- MIC is now 2.5x FASTER than a dual socket CPU node
KMP_AFFINITY and OMP scaling

SCALING AND AFFINITY
Binding Threads to Hardware

- Using the Intel KMP_AFFINITY interface
  - KMP_AFFINITY={compact, scatter, balanced}
  - KMP_AFFINITY=explicit, proclist=[0,1,2,3]
  - Adding *verbose* will dump the full affinity information when the run starts
  - Adding *granularity=fine* binds to specific thread contexts and may help in codes with heavy L1 cache reuse

- The MIC is a single chip, so there is no need for *numactl*

- We run all tests with *granularity=fine* in order to prevent thread migration
KMP_AFFINITY Example

compact

1 2 3 4
5 6 7 8

balanced

1 2
3 4
5 6
7 8

scatter

1 5
2 6
3 7
4 8
Scaling for Different Affinity Settings

![Graph showing scaling for different affinity settings with MLUPS on the y-axis and number of OMP threads on the x-axis. Lines represent Balanced, Scatter, and Compact.](image-url)
OMP + Offload

HOST + COPROCESSOR EXECUTION
Code Design

• Workload division
  – Partitioning in the Y direction
  – Avoids shortening the vectorizable innermost loop
  – Avoids having unaligned accesses due to uncommon partition in the Z direction
  – Avoids reducing maximum number of usable threads (mapped to Z direction blocks)

• Data transfer
  – Unlike CUDA codes the default is not for data to be persistent on the coprocessors in between offload sections
  – Carefully annotated data transfers to avoid MIC reallocation
  – Using asynchronous transfers is a requirement for host + MIC execution in this mode

• There are CPU and MIC versions of each function
Offload Data Transfer Examples

Some definitions to help handle transfers:

#define ALLOC alloc_if(.TRUE.)
#define REUSE alloc_if(.FALSE.)
#define FREE free_if(.TRUE.)
#define KEEP free_if(.FALSE.)

Transfer to mic device with id 0, allocating the data on the MIC, copying it over and returning to the host without freeing the array:

!DIR$ OFFLOAD_TRANSFER TARGET(MIC:0) IN( buff_mic : ALLOC KEEP )

Offload function PostCollisionMIC, reuse current values of array on device mic 0 during offload and return array values to host when offload is completed without freeing the array:

!DIR$ OFFLOAD BEGIN TARGET(MIC:0) OUT(buff_mic : REUSE KEEP )
   CALL PostCollisionMIC
!DIR$ END OFFLOAD
Offload Data Transfers

![Graph showing bandwidth (MB/s) vs. message size (Bytes) for ALLOC/FREE and REUSE/KEEP.]
Host + Coprocessor Results

- Since native execution is roughly 2.5x faster on the mic a naïve partition is $\frac{2.5}{2.5 + 1} = 0.71$ (71% workload for MIC)

- Tried in the range 60% - 80%, and optimal performance was found for 70% MIC workload

- MIC+CPU performance is 142.8 MLUPS

- Equivalent to 2.8 CPU nodes (44 CPU cores)

- Performance is relatively good, but this code had several issues
  - Not scalable (one node only)
  - Time consuming because of (essentially) duplicated functions
  - Increased development errors due to larger number of functions
MPI+OMP

SYMMETRIC EXECUTION
Code Design

- MPI partitioning allowed in 3D
  - But restricted to Y when possible to maximize length of innermost loop (vectorization) and length of outermost loop (OMP)
- MPI calls are made inside OMP MASTER regions
- MPI and OMP unaware of each other
- MPI exchanges are performed in PostCollision and PostStream
- Single ghost layer (or halo) used for MPI data exchange
- Load balance is coarse-grained
  - It depends on the number of tasks assigned to CPU and MIC
- No need to duplicate functions
- Requires two binaries
- Initially worried about MPI MIC to CPU data exchange
  - 5x slower than CPU to MIC
  - Fixed in latest Mvapich2
MIC/Host MPI Exchanges

Intel MPI

Mvapich2
Symmetric Execution Results

- We used 2 tasks on the MIC and 1 task on the CPU to assign a 66% workload fraction to the MIC
- The number of threads was 16 for the CPU and 120 for each of the MIC tasks
- Measured performance was 147.0 MLUPS
- This is equivalent to 2.8x nodes using only the CPUs
- This code design is scalable, and required only one major code change (the AOS to SOA modification)
Performance and Code Evolution
Summary

- Initial port of a multiphase LBM code to the MIC yielded low performance
  - 4x slower than a CPU node

- Final performance is very satisfactory for the amount of work performed
  - 2.8x faster than a cpu node alone

- Host profiles were not useful when porting

- There are some poorly understood alignment issues

- SOA + Symmetric execution mode provide good performance and scalability potential
FDTD Case Study -- ICT
Finite Difference Time Domain
Application background

• Widely used in many electromagnetics domains

Microwave
Electromagnetic detection
Radar

electromagnetic protection
Navigation
Antenna
Finite Difference Time Domain

Algorithm introduction

- An algorithm used for electromagnetic simulation by solving Maxwell equation.

\[
\begin{align*}
\nabla \times \vec{E} &= -\mu \frac{\partial \vec{H}}{\partial t} \\
\nabla \times \vec{H} &= \varepsilon \frac{\partial \vec{E}}{\partial t} + \sigma \vec{E}
\end{align*}
\]

For example, \( E_x \) obtained by:

\[
E_x^{n+1}(i+1/2, j, k) = \frac{2\varepsilon(m) - \sigma(m)\Delta t}{2\varepsilon(m) + \sigma(m)\Delta t} E_x^n(i+1/2, j, k) + \frac{2\Delta t}{2\varepsilon(m) + \sigma(m)\Delta t}
\]

\[
\begin{align*}
H_z^{n+1/2}(i+1/2, j+1/2, k) - H_z^{n+1/2}(i+1/2, j-1/2, k) \\
\frac{\Delta y}{\Delta z}
\end{align*}
\]

\[
\begin{align*}
H_y^{n+1/2}(i+1/2, j, k+1/2) - H_y^{n+1/2}(i+1/2, j, k-1/2) \\
\frac{\Delta z}{\Delta y}
\end{align*}
\]
Hotspot Analysis

- Define the candidate functions for optimization
  - update_E_PML(21.5%), update_H_PML(24.7%)
  - update_e(18.8% CPU time), update_h(19.1%)
Scalability Optimization

- Scalability was improved a lot compared to the baseline code after optimization by
  - Loop fusion
  - Merge the nested loop

```c
#pragma omp for nowait
for(i=0; i <= nx; i++)
    for(j=1; j <= ny; j++)
        for(k=1; k <= nz; k++) {
            ........
        }

int j_end = ny+1;
int ij_end=(nx+1)*(ny+1);
int ij_start = j_end + 1;
#pragma omp for nowait
for(int ij=ij_start; ij < ij_end; ij++) {
    i = ij/j_end;
    j = ij%j_end;
    if(!j) continue;
    for(k=1; k <= nz; k++) {
        ........
    }
}
```

```c
#pragma omp for nowait
for(i=0; i <= nx; i++)
    for(j=1; j <= ny; j++)
        for(k=1; k <= nz; k++) {
            Ex= ........
        }

#pragma omp for nowait
for(i=0; i <= nx; i++)
    for(j=1; j <= ny; j++)
        for(k=1; k <= nz; k++) {
            Ey= ........
        }

#pragma omp for nowait
for(i=0; i <= nx; i++)
    for(j=1; j <= ny; j++)
        for(k=1; k <= nz; k++) {
            Ez= ........
        }
```
Vectorization

- Compiler help to auto-vectorize the most inner loop by
  - Remove code dependency
  - #pragma simd
  - Adding compiler option “-ansi-alias”
- Check the speed up of scalar code vs vectorized code
Implement on Intel® Xeon Phi™ coprocessor

- Written by C and paralleled by OpenMP.

- Include three computation dataset mode, the performance shows different: small, medium, large dataset

- Migration to Intel® MIC in native mode, with medium dataset. (240*300*300)

- The optimized performance shows below,

![Speedup chart]

<table>
<thead>
<tr>
<th></th>
<th>SNB</th>
<th>KNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Dual E5-2670</td>
<td>Dual Intel® Xeon Phi™</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.6GHz</td>
<td>1.09 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>61 cores</td>
<td>61 cores</td>
</tr>
<tr>
<td>Memory size</td>
<td>8GB</td>
<td>8GB</td>
</tr>
<tr>
<td>Compiler</td>
<td>XE 13 0.079</td>
<td>Compiler XE 13 0.079</td>
</tr>
</tbody>
</table>
Case Study: Deep Learning
Case Study on IPDC

• Case: Deep Learning
  - Deep Learning is a new domain of machine learning, it use multiple layers of nonlinear neural network instead of traditional static model
  - Deep Learning bring subversive influence to human-machine interaction technology such as voice, image, hands writing and so on
  - Deep Learning greatly improve accuracy and processing time
  - Deep Learning is also time consuming part in model training
Hotspots Analysis

- 2S SNB Baseline: no optimization, single thread, time: 77.33s
- Hardware: 2Sockets Intel® Xeon E5-2670 (8 cores, 2.6GHz)
- Most time consuming function is ContrastiveDivergence: self time 69.8s, 90%
- CPI = 5.6 too high

![Hotspots Analysis Table]

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>CPU Time: Self by Utilization</th>
<th>Instr... Retire...</th>
<th>Ins... Ret...</th>
<th>CPI</th>
<th>CPI Rate...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td></td>
<td>100.0%</td>
<td>0</td>
<td>5.165</td>
<td>0.000</td>
</tr>
<tr>
<td>main</td>
<td></td>
<td>93.3%</td>
<td>0</td>
<td>5.492</td>
<td>0.000</td>
</tr>
<tr>
<td>▼ContrastiveDivergence</td>
<td></td>
<td>91.2%</td>
<td>23,..</td>
<td>5.608</td>
<td>7.595</td>
</tr>
<tr>
<td>▼ComputeStatusVisibleUnits</td>
<td>5.071s</td>
<td>18.6%</td>
<td>7,2...</td>
<td>1.813</td>
<td>1.813</td>
</tr>
<tr>
<td>▼ComputeStatusHiddenUnits</td>
<td>0.950s</td>
<td>5.8%</td>
<td>2,2...</td>
<td>1.083</td>
<td>1.083</td>
</tr>
<tr>
<td>▼ComputeStatusHiddenUnits</td>
<td>0.937s</td>
<td>5.5%</td>
<td>2,1...</td>
<td>1.152</td>
<td>1.152</td>
</tr>
<tr>
<td>▼init1</td>
<td>0.047s</td>
<td>0.7%</td>
<td>256,..</td>
<td>0.453</td>
<td>0.453</td>
</tr>
<tr>
<td>▼init</td>
<td>0.044s</td>
<td>0.7%</td>
<td>292,..</td>
<td>0.521</td>
<td>0.521</td>
</tr>
<tr>
<td>▼init</td>
<td>0.043s</td>
<td>0.7%</td>
<td>276,..</td>
<td>0.493</td>
<td>0.493</td>
</tr>
<tr>
<td>▼random_r</td>
<td>0.216s</td>
<td>3.4%</td>
<td>1,3...</td>
<td>0.442</td>
<td>0.442</td>
</tr>
<tr>
<td>▼random</td>
<td>0.149s</td>
<td>1.4%</td>
<td>542,..</td>
<td>0.642</td>
<td>0.642</td>
</tr>
<tr>
<td>▼▼ Selected 1 row(s): 76.755s</td>
<td>69.798s</td>
<td>91.2%</td>
<td>23,..</td>
<td>5.608</td>
<td>7.595</td>
</tr>
</tbody>
</table>
Hotspots Analysis

- L1 cache hit ratio: \( \frac{2640}{24978} = 10.6\% \), too low
Optimization on Intel Xeon

- Original code
  - Inconsecutive access in innermost loop, blocking vectorization and cache hits

```c
for (i = 0; j < J; j++)
{
    for (i = 0; i < I; i++)
    {
        float delta = (0.0);
        for (s = 0; s < samples/*->Rows()*/; s++)
        {
            delta += v[s * I + i] * h_data[s * J + j] - v_reco[s * I + i] * h_reco[s * J + j];
        }
        delta /= samples;
        w[j * I + i] = w[j * I + i] + momentum * lastDeltaW[j * I + i] + 0.004 * delta;
        lastDeltaW[j * I + i] = 0.004 * delta + momentum * lastDeltaW[j * I + i];
    }
    float deltaB = (0.0);
    for (s = 0; s < samples/*->Rows()*/; s++)
    {
        deltaB += h_data[s * J + j] - h_reco[s * J + j];
    }
    deltaB /= samples;
    b[j] = b[j] + momentum * lastDeltaB[j] + 0.004 * deltaB;
    lastDeltaB[j] = 0.004 * deltaB + momentum * lastDeltaB[j];
    //lastDeltaB[j] = 0.004 * deltaB + momentum*lastDeltaB[j];
    //b[j] = b[j] + lastDeltaB[j];
}
```
Optimization on Intel Xeon

- Optimization V1
  - transpose for consecutive access in innermost loop
  - SIMD optimization

```c
for (j = 0; j < J; j++)
{
    #ifdef OPT1
    float *h_data_trs_ptr = h_data_trs + j * samples;
    float *h_recon_trs_ptr = h_recon_trs + j * samples;
    #endif

    for (i = 0; i < I; i++)
    {
        float delta = (0.0f);
        float *v_trs_ptr = v_trs + i * samples;
        float *v_recon_trs_ptr = v_recon_trs + i * samples;

        for (s = 0; s < samples/*v->Rows()*/; s++)
        {
            if 1
            #ifdef USE_PRAG_SIMD
            #pragma loop_count=DS
            #pragma simd reduction(+:delta)
            #endif
            delta += v[s * I + i] * h_data[s * J + i] - v_recon[s * I + i] * h_recon[s * J + j];
            #endif
            else
                //delta += v_trs[s * samples + s] * h_data_trs[j * samples + s] - v_recon_trs[s * samples + s];
                delta += v_trs_ptr[s] * h_data_trs_ptr[s] - v_recon_trs_ptr[s] * h_recon_trs_ptr[s];
            #endif
        }
    }
    #ifdef OPT1
    h_data_trs[] * samples + s] = h_data[s * I + i];
    h_recon_trs[] * samples + s] = v_recon[s * I + i];
    #endif
}
```
Optimization on Intel Xeon

- Original code
  - Inconsecutive access in innermost loop, blocking vectorization and cache hits

```c
for (s = 0; s < samples; s++)
{
    for (i = 0; i < I; i++)
    {
        float sum = a[i];
        for (j = 0; j < J; j++)
        {
            sum += h[s * J + j] * v[j * I + i];
        }
    }
    v[s * I + i] = sum;
}
```
Optimization on Intel Xeon

- Optimization V2
  - Reconstruction code, directly write to result array and exchange outer loops
  - SIMD optimization

```c
for (s = 0; s < samples; s++)
{
    memcpy(v + s * I, a, I * sizeof(float));
    for (j = 0; j < J; j++)
    {
        #ifdef USE_PRAG_SIMD
        #pragma loop_count=DS
        #pragma simd
        #endif
        for(i = 0; i < I; i++)
        {
            v[s * I + i] += h[s * J + j] * w[j * I + i];
        }
    }
}
```
Optimization on Intel Xeon

- **Optimization V3**
  - Reconstruct code, split the value assigning to a different loop
  - Left codes can use MKL sgemm, which has been highly optimized both on Intel Xeon and Intel Xeon Phi

```c
for (s = 0; s < samples; s++)
{
    memcpy(v + s * I, a, I * sizeof(float));
    for (j = 0; j < J; j++)
    {
        if USE_PRAGSIMD
            pragma loop_count=DS
            pragma simd
            endif
        for(i = 0; i < I; i++)
            {
                v[s * I + i] += h[s * J + j] * w[j * I + i];
            }
    }
}
```

```c
for (s = 0; s < samples; s++)
{
    memcpy(v + s * I, a, I * sizeof(float));
}
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            samples, I, J, 1.0f, h, J, w, I, 1.0f, v, I);
```
Optimization on Intel Xeon

• Optimization V4
  - Use OpenMP multi-threads to scale code to multi/many cores
  - Use schedule(dynamic) to ensure load balance between different threads
  - Use KMP_AFFINITY=balanced,granularity=thread to bind threads to multi/many cores, avoiding threads migration

```c
#define USE_MULTITHREADS
#ifndef USE_MULTITHREADS
#pragma omp parallel for private(j) schedule(dynamic, OMP_GRAIN) num_threads(NUM_THREADS)
#else
#pragma omp parallel for private(j) schedule(dynamic, OMP_GRAIN) num_threads(NUM_THREADS)
#endif
for (j = 0; j < J; j++)
{
#define OPT1

float *h_data_trs_ptr = h_data_trs + j * samples;
float *h_recon_trs_ptr = h_recon_trs + j * samples;

float deltsB = (0.0f);
#endif

#ifndef USE_PRAG_SIMD
#pragma loop_count=DS
#pragma simd reduction(+::deltaB)
#endif
for (int s = 0; s < samples/*v->Rows()*/; s++)
{
#define OPT1

deltaB += h_data[s * J + j] - h_recon[s * J + j];
#endif

#else
//deltaB += h_data_trs[j * samples + s] - h_recon_trs[j * samples + s];
deltaB += h_data_trs_ptr[s] - h_recon_trs_ptr[s];
#endif
}
Optimization on Intel Xeon

- Optimization V5
  - Original code
  - Glibc `rand()` is not thread safe function, and will block parallelism

```c
for (s = 0; s < samples; s++)
{
    for (j = 0; j < J; j++)
    {
        float *v_ptr = v + s * I;
        float *w_ptr = w + j * I;
        float sum = b[j];

        #ifdef USE_PRAG_SIMD
        #pragma loop_count=D1
        #pragma simd reduction(+:sum)
        #endif

        for(i = 0; i < I; i++)
        {
            //sum += v[s * I + i] * w[j * I + i];
            sum += v_ptr[i] * w_ptr[i];
        }

        //h[s * J + j] = CUDA_SIGMOID(sum);
        h[s * J + j] = 1/(1.0f+expf(sum));

        h_recon[s * J + j] = (h[s * J + j] > ((float)rand()/RAND_MAX)) ? 1.0f:0.0f;
    }
}
```
Optimization on Intel Xeon

- Optimization V5
  - Original code
  - Glibc rand() is not thread safe function, and will block parallelism

```c
for (s = 0; s < samples; s++)
{
    for (j = 0; j < J; j++)
    {
        float *v_ptr = v + s * I;
        float *w_ptr = w + j * I;
        float sum = b[j];

        #ifdef USE_PRAG_SIMD
        #pragma loop_count=DI
        #pragma simd reduction(+:sum)
        #endif

        for(i = 0; i < I; i++)
        {
            //sum += v[s * I + i] * w[j * I + i];
            sum += v_ptr[i] * w_ptr[i];
        }

        //h[s * J + j] = CUDA_SIGMOID(sum);
        h[s * J + j] = 1/(1.0f+expf(sum));

        h_recon[s * J + j] = (h[s * J + j] > ((float)rand()/RAND_MAX)) ? 1.0f:0.0f;
    }
}
```
Optimization on Intel Xeon

- Optimization V5
  - Pre-calculate rand()
  - Increasing parallelism

```c
#pragma omp parallel for private(s,j) collapse(2) schedule(dynamic) num_threads(NUM_THREADS)
  for (s = 0; s < samples; s++)
  {
    for (j = 0; j < J; j++)
    {
      float *v_ptr = v + s * I;
      float *w_ptr = w + j * I;
      float sum = b[j];

      #ifdef USE_PRAG_SIMD
      #pragma loop_count=DI
      #pragma simd reduction(+:sum)
      #endif

      for(i = 0; i < I; i++)
      {
        //sum += v[s * I + i] * w[j * I + i];
        sum += v_ptr[i] * w_ptr[i];
      }

      //h[s * J + j] = CUDA_SIGMOID(sum);
      h[s * J + j] = 1/(1.0f+expf(sum));
      h_recon[s * J + j] = (h[s * J + j] > rand_array[s * J + j] / ((float)rand() / RAND_MAX)) ? 1.0f : 0.0f;
    }
  }
```
Optimization on Intel Xeon

- Optimization V6
  - Use MKL random generator instead of Glibc rand(), further increasing parallelism

```c
#ifdef USE_MKL_RAND
  vsrNewStream(&Randomstream, /*VSL_BRNG_MT19937*/VSL_BRNG_SFMT19937, RANDSEED);
#endif

#ifdef USE_MKL_RAND
  vsRngUniform (VSL_RNG_METHOD_UNIFORM_STD, Randomstream, samples * J, rand_array, 0.0f, 1.0f);
#else
  for (s = 0; s < samples; s++)
  {
    for (j = 0; j < J; j++)
      {
    #ifndef OPT7
      rand_array[s * J + j] = (float)rand() / RAND_MAX;
    #else
      rand_array[s * J + j] = (float)rand() * rcp_randmax;
    #endif
  ```
Optimization on Intel Xeon

- Optimization result on Intel Xeon
  - CPI reduced from 5.6 to 1.09
  - L1 cache hit ratio increased from 10.6% to 98.16%
  - Optimizations on Intel Xeon are all useful for Intel Xeon Phi
## Performance on Intel Xeon & Intel Xeon Phi

<table>
<thead>
<tr>
<th></th>
<th>Xeon® E5 processor – 1 threads baseline</th>
<th>Xeon® E5 processor – 32 threads optimization</th>
<th>Xeon Phi ™ coprocessor – 244 tasks optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time elapsed</td>
<td>77.335 s</td>
<td>0.2162 s</td>
<td>0.085 s</td>
</tr>
</tbody>
</table>

### Speedups vs. SNB Original 1 thread

- Original 1 thread: 1.00
- 2S SNB Optimization 32 threads: 357.70
- 1 MIC Optimization 244 threads: 910.90

SNB: 2S Intel® Xeon E5-2670 (8 cores, 2.6GHz)
MIC: Intel® Xeon Phi™ (B0, 61 cores, 1.1GHz; 8GB @ 5.5GT/s)
英特尔®软件学院2013年全新推出的课程
基于英特尔®集成众核架构的编程和优化

授课形式：课堂讲授 + 动手实验

课程长度：2-4天（可基于客户需求定制）

目标学员：

本课程适合集群和并行系统设计高性能，可扩充应用的软件工程师，项目负责人和解决方案架构师；计算机、高性能应用相关专业的教师等。

课程描述：

本课程总时长为期4天，采用课堂授课与动手实践相结合的方式，主要内容包括英特尔集成众核架构介绍，基于众核架构的开发环境搭建、众核编程、调试、优化以及实际应用案例分享等。本课程设计根据不同用户需求，由初级、中级、高级三部分组成，学生们可以从中阶梯式地学习和掌握课程内容。重点在于了解如何将实际应用移植到集成众核架构，利用英特尔®工具分析并提取应用特征，通过多线程、多任务、向量化等并行优化方法获得性能提升。案例研究说明如何将课堂上的方法和工具合理配合，提升实际应用的性能。
培训模式及联系方式

英特尔®软件学院:  http://software.intel.com/zh-cn/college

联系邮件: isc.apac@intel.com
Backup