Parallel Programming and Optimization with Intel Xeon Phi Coprocessors

Colfax Developer Boot Camp

Vadim Karpusenko, PhD and Andrey Vladimirov, PhD
Colfax International

July 2014, Rev. 12
This document represents the materials of a one-day training “Parallel Programming Boot Camp” developed and run by Colfax International.

Disclaimer

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

Please sign in during any coffee break to receive an invitation to a survey. Completing the survey earns you a free electronic copy of our book “Parallel Programming and Optimization with Intel Xeon Phi Coprocessors”.

![Image of Colfax Workshop Sign Up form]
Supplementary Materials
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, 2013

Parallel Computing in the Search for New Physics at LHC
By: James Darbro
2 December 2013 06:00

Abstract
In the past few months we have had the pleasure of collaborating with ACT, a world leader in detector simulation. The aim of our project is to improve the performance of the event generator LHC, as it is used to model events at the event generator level. For this purpose, we have implemented a new algorithm for the event generator, which allows for faster simulations. This method allows faster computation of the event generator, which makes it possible to simulate larger event samples. The performance of the event generator has been measured in terms of speed and accuracy. The results show that the new algorithm significantly improves the performance of the event generator, making it possible to simulate larger event samples in a shorter time.

Accelerating Public Domain Applications: Lessons from Models of Radiation Transport in the Milky Way Galaxy
By: John Smith
25 November 2013 06:00

Abstract
In this paper, we present a case study of applying a high-performance computing (HPC) approach to a public domain application. We implemented a high-performance version of the HEATCODE library, which is used to model the propagation of cosmic rays. The HEATCODE library was optimized using a combination of parallel programming and hardware acceleration. The results show that the performance of the HEATCODE library was significantly improved, making it possible to simulate larger event samples in a shorter time. The results of this study can be used to improve the performance of public domain applications.

Heterogeneous Clustering with Homogeneous Code: Accelerate MPI Applications Without Code Surgery Using Intel Xeon Phi Coprocessors
By: Maria Garcia
25 November 2013 06:00

Abstract
In this paper, we present a case study of applying a high-performance computing (HPC) approach to a public domain application. We implemented a high-performance version of the HEATCODE library, which is used to model the propagation of cosmic rays. The HEATCODE library was optimized using a combination of parallel programming and hardware acceleration. The results show that the performance of the HEATCODE library was significantly improved, making it possible to simulate larger event samples in a shorter time. The results of this study can be used to improve the performance of public domain applications.

Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors
By: Richard Williams
25 November 2013 06:00

Abstract
In this paper, we present a case study of applying a high-performance computing (HPC) approach to a public domain application. We implemented a high-performance version of the HEATCODE library, which is used to model the propagation of cosmic rays. The HEATCODE library was optimized using a combination of parallel programming and hardware acceleration. The results show that the performance of the HEATCODE library was significantly improved, making it possible to simulate larger event samples in a shorter time. The results of this study can be used to improve the performance of public domain applications.

Download source code for the data
Research and Consulting

http://research.colfaxinternational.com/
http://nlreg.colfax-intl.com/
It all comes down to PARALLEL PROGRAMMING! (applicable to processors and Intel® Xeon Phi™ coprocessors both)

Forward, Preface
Chapters:
1. Introduction
2. High Performance Closed Track Test Drive!
3. A Friendly Country Road Race
4. Driving Around Town: Optimizing A Real-World Code Example
5. Lots of Data (Vectors)
6. Lots of Tasks (not Threads)
7. Offload
8. Coprocessor Architecture
9. Coprocessor System Software
10. Linux on the Coprocessor
11. Math Library
12. MPI
13. Profiling and Timing
14. Summary
Glossary, Index

This book belongs on the bookshelf of every HPC professional. Not only does it successfully and accessibly teach us how to use and obtain high performance on the Intel MIC architecture, it is about much more than that. It takes us back to the universal fundamentals of high-performance computing including how to think and reason about the performance of algorithms mapped to modern architectures, and it puts into your hands powerful tools that will be useful for years to come.

—Robert J. Harrison
Institute for Advanced Computational Science, Stony Brook University
List of Topics
List of Topics

1. Introduction
   - Intel Xeon Phi Architecture from the Programmer’s Perspective
   - Software Tools for Intel Xeon Phi Coprocessors
   - Will Application X benefit from the MIC architecture?

2. Programming Models for Intel Xeon Phi Applications
   - Native Applications for Coprocessors and MPI
   - Offload Programming Models
   - Using Multiple Coprocessors
   - MPI Applications and Heterogeneous Clustering
List of Topics

Porting Applications to the MIC Architecture

- Future-Proofing: Reliance on Compiler and Libraries
- Choosing the Programming Model
- Cross-Compilation of User Applications
- Performance Expectations

Parallel Scalability on Intel Architectures

- Vectorization (Single Instruction Multiple Data, SIMD, Parallelism)
- Multi-threading: OpenMP, Intel Cilk Plus
- Task Parallelism in Distributed Memory, MPI
List of Topics

- Optimization for the Intel Xeon Product Family
  - Optimization Checklist
  - Finding Bottlenecks with Intel VTune Amplifier
  - MPI Diagnostics Using Intel Trace Analyzer and Collector
  - Intel Math Kernel Library (MKL)
  - Scalar Optimization Considerations
  - Automatic Vectorization and Data Structures
  - Optimization of Thread Parallelism
List of Topics

- Advanced Optimization for the MIC Architecture
  - Memory Access and Cache Utilization
  - Data Persistence and PCIe Traffic
  - MPI Applications on Clusters with Coprocessors

- Conclusion
  - Course Recap
  - Additional Resources: Reading, Guides, Support
§1. Introduction to the Intel Many Integrated Core (MIC) Architecture
MIC Architecture from the Programmer’s Perspective
Intel Xeon Phi Coprocessors and the MIC Architecture

- PCIe end-point device
- High Power efficiency
- \( \sim 1 \text{ TFLOP/s in DP} \)
- Heterogeneous clustering

For highly parallel applications which reach the scaling limits on Intel Xeon processors
Xeon Family Product Performance

Many-core Coprocessors (Xeon Phi) vs Multi-core Processors (Xeon) —

- Better performance per system & performance per watt for parallel applications
- Same programming methods, same development tools.

Source: “Intel Xeon Product Family: Performance Brief”
### Intel Xeon Processors and the MIC Architecture

**Multi-core Intel Xeon processor**
- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- $\leq 12$ cores/socket $\approx 3$ GHz
- 2-way hyper-threading
- 256-bit AVX vectors

**Many-core Intel Xeon Phi coprocessor**
- C/C++/Fortran; OpenMP/MPI
- Special Linux $\mu$OS distribution
- 6–16 GB cached GDDR5 RAM
- 57 to 61 cores at $\approx 1$ GHz
- 4-way hyper-threading
- 512-bit IMCI vectors
Examples of Solutions with the Intel MIC Architecture

Colfax’s **CXP7450** workstation with two Intel Xeon Phi coprocessors

Colfax’s **CXP9000** server with eight Intel Xeon Phi coprocessors
N-body simulation on...

Two Intel® Xeon® CPUs

One Intel® Xeon Phi™ coprocessor

Two Intel® Xeon Phi™ coprocessors

Paper: research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx
Demo: http://www.youtube.com/watch?v=KxaSEcmkGTo
Source: “Intel Xeon Phi Coprocessor - the Architecture“
Core Topology

![Core Topology Diagram]

- **Instruction Decode**
- **Scalar Unit**
- **Scalar Registers**
- **Vector Unit**
- **Vector Registers**
- **L1 Cache (Data and Instruction)**
- **L2 Cache**
- **Interprocessor Network**

**4 Threads per Core**
- 64-bit
- 512-wide
- In Order
- VPU: integer, SP, DP; 3-operand, 16-instruction
- Specialized Instructions (new encoding)
- HW transcendental

**Fully Coherent**
- L2 Hardware Prefetching
- 32 KB per core
- 512 KB Slice per Core – Fast Access to Local Copy

**Ring interconnect**
## Cache Structure

The caches are 8-way associative, fully coherent with the LRU (Least Recently Used) replacement policy.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cache line size</strong></td>
<td>64B</td>
</tr>
<tr>
<td><strong>L1 size</strong></td>
<td>32KB data, 32KB code</td>
</tr>
<tr>
<td><strong>L1 latency</strong></td>
<td>1 cycle</td>
</tr>
<tr>
<td><strong>L2 size</strong></td>
<td>512KB</td>
</tr>
<tr>
<td><strong>L2 ways</strong></td>
<td>8</td>
</tr>
<tr>
<td><strong>L2 latency</strong></td>
<td>11 cycles</td>
</tr>
<tr>
<td><strong>Memory → L2 prefetching</strong></td>
<td>hardware and software</td>
</tr>
<tr>
<td><strong>L2 → L1 prefetching</strong></td>
<td>software only</td>
</tr>
<tr>
<td><strong>Translation Lookaside Buffer (TLB) coverage options (L1, data)</strong></td>
<td>64 pages of size 4KB (256KB coverage)</td>
</tr>
<tr>
<td></td>
<td>8 pages of size 2MB (16MB coverage)</td>
</tr>
</tbody>
</table>
Features of the IMCI Instruction Set

Intel IMCI is the instruction set supported by Intel Xeon Phi copr.

- **512-bit wide registers**
  - can pack up to eight 64-bit elements (long int, double)
  - up to sixteen 32-bit elements (int, float)

- **Arithmetic Instructions**
  - Addition, subtraction and multiplication
  - Fused Multiply-Add instruction (FMA)
  - Division and reciprocal calculation;
  - Error function, inverse error function;
  - Exponential functions (natural, base 2 and base 10) and the power function.
  - Logarithms (natural, base 2 and base 10).
  - Square root, inverse square root, hypothenuse value and cubic root;
  - Trigonometric functions (sin, cos, tan, sinh, cosh, tanh, asin, acos ...);
  - Rounding functions
Features of the IMCI Instruction Set

- Initialization, Load and Store, Gather and Scatter
- Comparison
- Conversion and type cast
- Bitwise instructions: NOT, AND, OR, XOR, XAND
- Reduction and minimum/maximum instructions
- Vector mask instructions
- Scalar instructions
- Swizzle and permute
Interactions between Operating Systems
Interactions between Operating Systems

Linux Host

- Host-side offload application
  - User code
  - Offload libraries, user-level driver, user-accessible APIs and libraries
- SSH

Intel® Xeon Phi™ coprocessor

- Virtual terminal session

Intel® Xeon Phi™ coprocessor support
- Libraries, tools, and drivers
- Linux OS
- PCIe Bus

Target-side “native” application
- User code
- Standard OS libraries plus any 3rd-party or Intel libraries
- Intel® Xeon Phi™ coprocessor communication and application-launching support
- PCIe Bus
- Linux uOS

Target-side offload application
- User code
- Offload libraries, user-accessible APIs and libraries

User-level code
- System-level code
Linux µOS on Intel Xeon Phi coprocessors (part of MPSS)

```
user@host% lspci | grep -i "co-processor"
06:00.0 Co-processor: Intel Corporation Device 2250 (rev 11)
82:00.0 Co-processor: Intel Corporation Device 2250 (rev 11)
user@host% sudo service mpss status
mpss is running
user@host% cat /etc/hosts | grep mic
172.31.1.1 host-mic0 mic0
172.31.2.1 host-mic1 mic1
user@host% ssh mic0
user@mic0% cat /proc/cpuinfo | grep proc | tail -n 3
processor : 237
processor : 238
processor : 239
user@mic0% ls /
amplxe  dev  home  lib64  oldroot  proc  sbin  sys  usr
bin  etc  lib  linuxrc  opt  root  sep3.10  tmp  var
```
Software Tools for Intel Xeon Phi Coprocessors
Execute MIC Applications (all free):

Drivers: Intel MIC Platform Software Stack (Intel MPSS) — mandatory — detect, boot and manage coprocessors

Libraries: Redistributable libraries — optional — run and distribute pre-built applications

OpenCL: Intel OpenCL SDK — optional

Monitoring MIC activity with micsmc (an MPSS tool)
MPSS Tools and Utilities

**micinfo** a system information query tool

**micsmc** a utility for monitoring and modifying the physical parameters: temperature, power modes, core utilization, etc.

**micctrl** a comprehensive configuration tool for the Intel Xeon Phi coprocessor operating system

**miccheck** a set of diagnostic tests for the verification of the Intel Xeon Phi coprocessor configuration

**micrasd** a host daemon logger of hardware errors reported by Intel Xeon Phi coprocessors

**micflash** an Intel Xeon Phi flash memory agent
Build Xeon Phi & Xeon CPU Applications (all licensed):

**Compilers**  : Intel C Compiler, Intel C++ Compiler, and Intel Fortran Compiler — mandatory

**Optimization tools**  : Intel VTune Amplifier XE and Intel Trace Analyzer and Collector (ITAC) — highly recommended

**Mathematics support**  : Intel Math Kernel Library (MKL) — highly recommended

**Development**  : Intel Inspector XE, Intel Advisor XE — optional

All-in-One Bundles, common for CPU and MIC
Will Application X Benefit from the MIC architecture?
Three Layers of Parallelism

- Instruction Pool
- Data Pool
- SIMD
- Vector Unit
- PU (Processor Unit)

Will Application X Benefit from the MIC architecture?
Three Layers of Parallelism
Three Layers of Parallelism

- Instruction Pool
- Data Pool
- SIMD
- Vector Unit
- PU
- MPI
- Host CPUs
- Xeon Phi coprocessor
- Compute Node 1
- Compute Node 2
- Compute Node 3

Will Application X Benefit from the MIC architecture?
Compute-Bound Application Performance

- Scalar & Single-thread
- Vector & Single-thread
- Scalar & Multi-threaded
- Vector & Multi-threaded

More Parallel

More Performance

- Intel Xeon Phi
- Intel Xeon

Will Application X Benefit from the MIC architecture?

One Size Does Not Fit All

An application must reach scalability limits on the CPU in order to benefit from the MIC architecture.

Use Xeon Phi if:
- Scales up to 100 threads
- Compute bound & vectorized, or bandwidth-bound

Use Xeon if:
- Serial or scales to \( \lesssim 10 \) threads
- Unvectorized or latency-bound
Xeon + Xeon Phi Coprocessors = Xeon Family

Programming models allow a range of CPU+MIC coupling modes

Xeon - Multi-Core Centric

MIC - Many-Core Centric

Multi-Core Hosted
General serial and parallel computing

Offload
Code with highly-parallel phases

Symmetric
Codes with balanced needs

Many Core Hosted
Highly-parallel codes

Will Application X Benefit from the MIC architecture?

§2. Programming Models for Intel Xeon Phi Applications
Native Execution

“Hello World” application:

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

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

Compile and run on host:

```
user@host% icc hello.c
user@host% ./a.out
Hello world! I have 32 logical cores.
user@host%
```
Native Execution

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

```
user@host% icc hello.c -mmic
user@host% scp a.out mic0:~/a.out
a.out 100% 10KB 10.4KB/s 00:00
user@host% ssh mic0
user@mic0% pwd
/home/user
user@mic0% ls
a.out
user@mic0% ./a.out
Hello world! I have 240 logical cores.
user@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 for Coprocessors with MPI

“Hello World” in MPI:

```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[]) {
    int i, rank, size, namelen;
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name (name, &namelen);
    printf ("Hello World from rank %d running on %s!\n", rank, name);
    if (rank == 0) printf("MPI World size = %d processes\n", size);
    MPI_Finalize ();
}
```
Running MPI Applications on Host

```
user@host% source /opt/intel/impi/4.1.0/intel64/bin/mpivars.sh
user@host% export I_MPI_FABRICS=shm:tcp
user@host% mpiicpc -o HelloMPI.XEON HelloMPI.c
user@host% mpirun -host localhost -np 2 ./HelloMPI.XEON
Hello World from rank 1 running on host!
Hello World from rank 0 running on host!
MPI World size = 2 processes
```

- Set up MPI environment variables
- Use wrapper script `mpiicpc` to compile
- Use automated tool `mpirun` to launch
Enable the MIC architecture in Intel MPI: \texttt{I\_MPI\_MIC=1}

Copy or NFS-share MPI library & executables with coprocessor

Use \texttt{mpiicpc} with \texttt{-mmic} to compile

Launch as if \texttt{mic0} is a remote host
Heterogeneous Clustering with Homogeneous Code: Asian Option Pricing

- Monte Carlo method
- MPI + OpenMP + automatic vectorization
- The same C code for clusters of
  a) CPUs
  b) Coprocessors
  c) CPUs+Coprocessors (heterogeneous)

- **No code modification** to run on the Intel MIC architecture
- No platform-specific tuning
- Bridged network configuration

Paper: research.colfaxinternational.com/post/2013/10/17/Heterogeneous-Clustering.aspx
Demo: http://youtu.be/GffmChTcWf8
Offload Programming Models
Explicit Offload: Pragma-based approach

“Hello World” in the explicit offload model:

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    printf("Hello World from host!\n");
    #pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n");
        fflush(0);
    }
    printf("Bye\n");
}
```

Application runs on the host, but some parts of code and data are moved ("offloaded") to the coprocessor.
Compiling and Running an Offload Application

- No additional arguments if compiled with an Intel compiler
- Run application on host as a regular application
- Code inside of #pragma offload is offloaded automatically
- Console output on Intel Xeon Phi coprocessor is buffered and mirrored to the host console
- If coprocessor is not installed, code inside #pragma offload runs on the host system
Functions used on coprocessor must be marked with the specifier
```
__attribute__((target(mic)))
```

Compiler produces a host version and a coprocessor version of such
functions (to enable fall-back to host)
Offloading Multiple Functions

```c
#pragma offload_attribute(push, target(mic))
void MyFunctionOne() {
    // ... implement function as usual
}

void MyFunctionTwo() {
    // ... implement function as usual
}
#pragma offload_attribute(pop)
```

- To mark a long block of code with the offload attribute, use `#pragma offload_attribute(push/pop)`
void MyFunction() {
    const int N = 1000;
    int data[N];
    #pragma offload target(mic)
    {
        for (int i = 0; i < N; i++)
            data[i] = 0;
    }
}

- Scope-local scalars and known-size arrays are offloaded automatically
- Data is copied from host to coprocessor at the start of offload
- Data is copied back from coprocessor to host at the end of offload
- Bitwise-copyable data only (arrays of basic types and scalars)
  C++ classes, etc. should use virtual-shared memory model
Global and static variables must be marked with the offload attribute.

#pragma offload_attribute(push/pop) may be used as well.
Data Marshalling for Dynamically Allocated Data

```c
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
Memory retention and data persistence on coprocessor

By default, memory on coprocessor is allocated before, deallocated after offload.

Specifiers `alloc_if` and `free_if` allow to avoid allocation/deallocation.

Can be combined with `length(0)` to avoid data transfer.

Why bother: data transfer across the PCIe bus is relatively slow (6 GB/s), and memory allocation on coprocessor is even slower (0.5 GB/s).
Precautions with persistent data

- Use explicit zero-based coprocessor number (e.g., `mic:0` as shown below)

- With multiple coprocessors, if target number is unspecified, any coprocessor can be used, which will result in runtime errors if persistent data cannot be found.

```c
#pragma offload target(mic:0) in(p : length(N)) alloc_if(1) free_if(0) 
{ /* allocate memory for array p on coprocessor, do not deallocate */ }```

- Do not change the value of the host pointer to a persistent array: the runtime system finds the data on coprocessor using the host pointer value, not variable name.
Virtual-shared Memory Model

_Cilk_shared int arr[N]; // This is a virtual-shared array

_Cilk_shared void Compute() { // This function may be offloaded
    // ... function uses array arr[]
}

int main() {
    // arr[] can be initialized on the host
    _Cilk_offload Compute(); // and used on coprocessor
    // and the values are returned to the host
}

- 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
Virtual-shared Memory Model

```c
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
Target-Specific Code

- During MIC architecture compilation, preprocessor macro `__MIC__` is defined.
- Allows to fine-tune application performance or output where necessary

```c
void __attribute__((target(mic))) MyFunction() {
    #ifdef __MIC__
        printf("I am running on a coprocessor.\n");
        const int tuningParameter = 16;
    #else
        printf("I am running on the host.\n");
        const int tuningParameter = 8;
    #endif
    // ... Proceed, using the variable tuningParameter
}
```
Using Multiple Coprocessors
Multiple Coprocessors with Explicit Offload

- Querying the number of coprocessors:

```c
const int numDevices = _Offload_number_of_devices();
printf("Number of available coprocessors: %d\n", numDevices);
```

- Specifying offload target:

```c
#pragma offload target(mic: 0)
{
    /* ... */
}
```
Multiple Blocking Offloads Using Host Threads
(Explicit Offload)

```c
const int nDevices = _Offload_number_of_devices();
#pragma omp parallel for
  for (int i = 0; i < nDevices; i++) {
#pragma offload target(mic: i)
  {
    MyFunction(/*...*/);
  }
}
```

- Up to 8 coprocessors, up to 32 host threads
- All offloads start simultaneously and block the respective thread
const int nDevices = _Offload_number_of_devices();
omp_set_num_threads(nDevices);
#pragma omp parallel for schedule(dynamic, 1)
for (int i = 0; i < nWorkItems; i++) {
    const int iDevice = omp_get_thread_num();
    #pragma offload target(mic: iDevice)
    {
        MyFunction(i);
    }
}
Asynchronous Offload

- By default, `#pragma offload` blocks until offload completes
- Use clause “signal” with any pointer to avoid blocking
- Use `#pragma offload_wait` to block where needed

```c
char* offload0;

#pragma offload target(mic:0) signal(offload0) in(data : length(N))
{ /* ... will not block code execution because of clause "signal" */ }

DoSomethingElse();

/* Now block until offload signalled by pointer "offload0" completes */
#pragma offload_wait target(mic:0) wait(offload0)
```

- Use the target number to avoid hanging
**Offload diagnostics**

- Set environment variable `OFFLOAD_REPORT` to 1 or 2 for automatic collection and output of offload information.
- Unset or set `OFFLOAD_REPORT=0` to disable offload diagnostics.
Environment variable forwarding with offload

- By default, all host environment variables on the host will be copied to the coprocessor when offload starts.
- In order to have different values for an environment variable on host and coprocessor, set `MIC_ENV_PREFIX`.
- The prefix is dropped when variables are copied to coprocessor.

```
user@host%  # This enables s
user@host%  export MIC_ENV_PREFIX=XEONPHI
user@host%
user@host%  # This sets the value of OMP_NUM_THREADS on the host:
user@host%  export OMP_NUM_THREADS=32
user@host%
user@host%  # This sets the value of OMP_NUM_THREADS on the coprocessor:
user@host%  export XEONPHI_OMP_NUM_THREADS=236
```
```cpp
const int nDevices = _Offload_number_of_devices();
char sig[nDevices];
for (int i = 0; i < nDevices; i++) {
  #pragma offload target(mic: i) signal(&sig[i])
  {
    MyFunction(/*...*/);
  }
}
for (int i = 0; i < nDevices; i++) {
  #pragma offload_wait target(mic: i) wait(&sig[i])
}
```

- Any pointer acts as a signal
- Must wait for all signals
MPI Applications and Heterogeneous Clustering
Heterogeneous MPI Applications: Host + Coprocessors

```
user@host% mpirun -host mic0 -n 2 ~/Hello.MIC : -host mic1 -n 2 ~/Hello.MIC : \
% -host localhost -n 2 ~/Hello.XEON
Hello World from rank 5 running on localhost!
Hello World from rank 4 running on localhost!
Hello World from rank 2 running on mic1!
Hello World from rank 3 running on mic1!
Hello World from rank 1 running on mic0!
Hello World from rank 0 running on mic0!
MPI World size = 6 ranks
```

- Specify Xeon executable for host processes
- Specify Xeon Phi executable for coprocessor processes
Option 1: Hybrid MPI+OpenMP with Offload.

- MPI processes are multi-threaded with OpenMP.
- MPI processes run only on CPUs.
- One or more OpenMP threads perform offload to coprocessor(s).
Option 2: Symmetric Pure MPI.

- MPI processes are single-threaded.
- Native MPI processes on the coprocessor.

E.g., 32 MPI processes on each CPU, 240 on each coprocessor.
Heterogeneous Distributed Computing with Xeon Phi

Option 1: **Symmetric Hybrid MPI+OpenMP.**

- MPI processes are multi-threaded with OpenMP.
- Native MPI processes on the coprocessor.

E.g., one 32-thr MPI proc on each CPU, 240-thr on each coprocessor.
File I/O in MPI Applications
RAM Filesystem

- Files are stored in the coprocessor RAM
- Does not survive MPSS restart or host reboot
- Fastest method
- Good for local pre-staged input or runtime scratch data
Files are stored on a physical or virtual drive on the host
Written data persistent across reboots
Fast method
Cannot share a drive between coprocessors
Good for distributed checkpointing
Network Storage

- Files are stored on a remote file server
- Can share a mount point across the cluster
- Lustre has scalable performance
- NFS is slow but easy to set up
Review: Programming Models
Programming Models

1. Native coprocessor applications
   - Compile with -mmic
   - Run with mcnativeloadex or scp+ssh
   - The way to go for MPI applications without offload

2. Explicit offload
   - Functions, global variables require __attribute__((target(mic)))
   - Initiate offload, data marshalling with #pragma offload
   - Only bitwise-copyable data can be shared

3. Clusters and multiple coprocessors
   - #pragma offload target(mic:i)
   - Use threads to offload to multiple coprocessors
   - Run native MPI applications
§3. Porting Applications to the MIC Architecture
Choosing the Programming Model
To Offload or Not To Offload

For a “MIC-friendly” application, Use offload if:

- Per-rank data set does not fit in the Xeon Phi onboard memory
- Need CPU: serial workload, intensive file I/O
- MPI bandwidth-bound or latency-bound workload
- Cannot compile some of dependencies for MIC

Use native/symmetric MPI if:

- Parallel work-items too small, so data transfer overhead is significant
- Peer-to-peer communication between workers is required
- Difficult to instrument data movement or sharing with coprocessor
PCIe Bandwidth Considerations

With data sent from host to coprocessor, communication overhead must be considered:

- PCIe bandwidth: 6 GB/s, theoretical max arithmetic performance 1 TFLOP/s, practical memory bandwidth 150-170 GB/s
- Offload if MIC performs $\gg 1000$ operations per transferred word
- Algorithms with strong complexity scaling (e.g., $O(n^2)$) likely less impacted by communication than with weak scaling (e.g., $O(n)$, $O(n \log n)$)
Cross-Compilation of User Applications
Simple CPU applications can be compiled for native execution on Xeon Phi coprocessors by supplying the flag “-mmic” to the Intel compiler:

```
user@host% icpc -c myobject1.cc -mmic
user@host% icpc -c myobject2.cc -mmic
user@host% icpc -o myapplication myobject1.o myobject2.o -mmic
```
Native Applications with Autotools

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

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

```bash
user@host% wget https://ftp.gnu.org/gnu/gmp/gmp-5.1.3.tar.bz2
user@host% tar -xf gmp-5.1.3.tar.bz2
user@host% cd gmp-5.1.3
user@host% ./configure CC=icc CFLAGS="-mmic" --disable-assembly --host=x86_64
...  
user@host% make
...  
```
Static Libraries with Offload

In offload applications, additional object files are produced:

```
user@host% # Program in myobject.cc contains #pragma offload
user@host% icpc -c myobject.cc
user@host% ls
myobject.cc  myobjectMIC.o  myobject.o
```

In order to compile the *MIC.o files into a static library with offload, use `xiar -qoffload-build` instead of `ar`.

See white paper for more details:
http://research.colfaxinternational.com/post/2013/05/03/Fast-Library-Xeon-Phi.aspx
Performance Expectations
Performance on MIC is a Function of Optimization Level

Performance Relative to Baseline

Optimization Step

Baseline: unoptimized, compiled with GCC, running on host CPUs (59 ms per spectrum)

Unoptimized with Offload

Thread Parallelism: Fit All Threads in Memory

Scalar Optimizations: Precomputation, Precision Control

Vectorization: Alignment, Padding, Hints

Heterogeneous: Using Host + Two Coprocessors

Algorithm Optimization: Pruning, Recurrence

Improved Interpolation Method: Packed Operations

Memory Access: Packed Data, Loop Tiling

Offload Traffic: Data Persistence on Coprocessor

GCC on CPUs

Intel C++ on CPUs

Intel C++ on Xeon Phi

Performance on MIC is a Function of Optimization Level

- Performance will be disappointing if code is not optimized for multi-core CPUs
- Optimized code runs better on the MIC platform and on the multi-core CPU
- Single code for two platforms + Ease of porting = Incremental optimization

HEATCODE Benchmarks

Caution on Comparative Benchmarks

- In most of our benchmarks, "Xeon Phi" = 5110P SKU (60 cores, TDP 225 W, $2.7k), "CPU" = dual Xeon E5-2680 (16 cores, TDP 260 W, $3.4k + system cost)

- Why dual CPU vs single coprocessor? Approximately the same Thermal Design Power (TDP) and cost.

Case study: 
Future-Proofing: Reliance on Compiler and Libraries
Future-Proofing: Reliance on Compiler and Libraries

**Thread Options**
- Intel® Math Kernel Library API*
- Intel® Threading Building Blocks
- Intel® Thilk™ Plus
- OpenMP*
- Pthreads*

**Ease of use**
- Intel® Math Kernel Library
- Array Notation: Intel® Thilk™ Plus
- Auto vectorization
- Semi-auto vectorization: #pragma (vector, simd)
- OpenCL*
- OpenMP*
- Pthreads*

**Vector Options**
- C/C++ Vector Classes (F32vec16, F64vec8)

Next Generation MIC: Knights Landing (KNL)

- 2nd generation MIC product: code name Knights Landing (KNL)
- Intel’s 14 nm manufacturing process
- A processor (running the OS) or a coprocessor (PCIe device)
- On-package high-bandwidth memory w/flexible memory models: flat, cache, & hybrid
- Intel Advanced Vector Extensions AVX-512 (public)

Source: Intel Newsroom
Getting Ready for the Future

- Porting HPC applications to today’s MIC architecture makes them ready for future architectures, such as KNL
- Xeon, KNC and KNL are not binary compatible, therefore assembly-level tuning will not scale forward.
- Reliance on compiler optimization and using optimized libraries (such as Intel MKL) ensures future-readiness.

Source: Intel Newsroom
Intel® Xeon Phi™ Product Family Roadmap

The Faster Path to Discovery

Available Today
Knights Corner
Intel® Xeon Phi™
x100 Product Family
22 nm process
Coprocessor
Over 1 TF DP Peak
Up to 61 Cores
Up to 16GB GDDR5

2H’15*
Knights Landing
Intel® Xeon Phi™
x200 Product Family
14 nm process
Server Processor &
Coprocesor
Over 3 TF DP Peak†
60+ cores
And new
details
today...

Future
TBA
3rd generation

* First commercial systems
† Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle. FLOPS = Cores x clock frequency x floating-point operations per second per cycle.

Knights Landing: Next-Generation Intel® Xeon Phi™

Architectural Enhancements = ManyX Performance

 knit 60+ cores
 3+ Teraflops
 3x Single-Thread
 2-D Core Mesh
 Cache Coherency

High-Performance Memory
Over 5X STREAM vs. DDR4
Up to 16 GB at launch
NUMA support

Server Processor

Binary-compatible with Intel® Xeon® processors

Core

with Enhancements for HPC

✓ 14nm process technology
✓ 4 Threads/Core
✓ Deep Out-of-Order Buffers
✓ Gather/Scatter
✓ Better Branch Prediction
✓ Higher Cache Bandwidth
... and many more

Based on Intel® Atom™ core (based on Silvermont microarchitecture)

Integrated Fabric

DDR4

Capacity Comparable to Intel® Xeon® Processors

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

14nm process technology
4 Threads/Core
Deep Out-of-Order Buffers
Gather/Scatter
Better Branch Prediction
Higher Cache Bandwidth
... and many more

Core

In partnership with

Integrated Fabric

DDR4

Capacity Comparable to Intel® Xeon® Processors

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

**Other logos, brands and names are the property of their respective owners.**

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

1 Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating-point operations per cycle.

2 Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner).

3 Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4.

Diagram is for conceptual purposes only and only illustrates a CPU, memory, integrated fabric and DDR memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.
Today’s Parallel Investment Carries Forward

MOST significant code modernizations carry forward

Recompile
Parallelization, threading, vectorization, cache-blocking, MPI+OpenMP hybridization & more.

TBB
MKL
OpenMP

Cilk
OpenCL
Knights Landing Enabled Performance Libraries & Runtimes

Intel® AVX-512
Cache Mode For High Bandwidth Memory
Knights Landing Enabled Compilers

ADDITIONAL tuning gains

Tuning
Exploiting NEW features and structures

Intel® Xeon Phi™ x100 Product Family
Native or Symmetric or Offload

Knights Landing
All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

A Paradigm Shift for Highly-Parallel

Server Processor with Leadership Integration are Keys to Future

Memory Bandwidth
Over 5x STREAM vs. DDR4

Memory Capacity
Comparable to Intel® Xeon® processors

Resiliency
Intel-server class reliability

Power Efficiency
>25% better than discrete card

I/O
Highest bandwidth

Cost
Less costly than discrete parts

Flexibility
Extensive server configurations

Density
3+ KNL with fabric in 1U
§4. Parallel Scalability on Intel Architectures
Vectorization (Single Instruction Multiple Data, SIMD, Parallelism)
SIMD Operations

SIMD — Single Instruction Multiple Data

Scalar Loop

```
for (i = 0; i < n; i++)
```

SIMD Loop

```
for (i = 0; i < n; i += 4)
    A[i:(i+4)] = A[i:(i+4)] + B[i:(i+4)];
```

Each SIMD addition operator acts on 4 numbers at a time.
### Instruction Sets in Intel Architectures

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Year and Intel Processor</th>
<th>Vector registers</th>
<th>Packed Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>1997, Pentium</td>
<td>64-bit</td>
<td>8-, 16- and 32-bit integers</td>
</tr>
<tr>
<td>SSE</td>
<td>1999, Pentium III</td>
<td>128-bit</td>
<td>32-bit single precision FP</td>
</tr>
<tr>
<td>SSE2</td>
<td>2001, Pentium 4</td>
<td>128-bit</td>
<td>8 to 64-bit integers; SP &amp; DP FP</td>
</tr>
<tr>
<td>SSE3–SSE4.2</td>
<td>2004 – 2009</td>
<td>128-bit</td>
<td>(additional instructions)</td>
</tr>
<tr>
<td>AVX</td>
<td>2011, Sandy Bridge</td>
<td>256-bit</td>
<td>Single and double precision FP</td>
</tr>
<tr>
<td>AVX2</td>
<td>2013, Haswell</td>
<td>256-bit</td>
<td>Single and double precision FP; integers, additional instructions</td>
</tr>
<tr>
<td>IMCI</td>
<td>2012, Knights Corner</td>
<td>512-bit</td>
<td>32- and 64-bit integers; single &amp; double precision FP</td>
</tr>
<tr>
<td>AVX-512</td>
<td>(future) Knights Landing</td>
<td>512-bit</td>
<td>32- and 64-bit integers; single &amp; double precision FP</td>
</tr>
</tbody>
</table>
The arrays float A[n] and float B[n] are aligned on a 16-byte (SSE2) and 64-byte (IMCI) boundary.

n is a multiple of 4 for SSE and a multiple of 16 for IMCI.

Variables Avec and Bvec are
128 = 4 × sizeof(float) bits in size for SSE2 and
512 = 16 × sizeof(float) bits for the Intel Xeon Phi architecture.
Automatic Vectorization of Loops

```c
#include <stdio.h>

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]);
}
```

```bash
user@host% icpc autovec.c -vec-report3
autovec.c(10): (col. 3) remark:
    loop was not vectorized:
    vectorization possible
    but seems inefficient.
autovec.c(14): (col. 3) remark:
    LOOP WAS VECTORIZED.
autovec.c(18): (col. 3) remark:
    loop was not vectorized:
    existence of vector
    dependence.
user@host% ./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
```
Automatic Vectorization of Loops on MIC architecture

Compilation and runtime output of the code for Intel Xeon Phi execution

```
user@host% icpc autovec.c -vec-report3 -mmic
autotest.c(10): (col. 3) remark: LOOP WAS VECTORIZED.
autotest.c(14): (col. 3) remark: LOOP WAS VECTORIZED.
autotest.c(18): (col. 3) remark: loop was not vectorized: existence of vector dependence.

user@host% mcnativeloadex 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
```
Automatic Vectorization of Loops

Limitations:

- Only for-loops can be auto-vectorized. Number of iterations must be known at a runtime and/or compilation time.
- Memory access in the loop must have a regular pattern, ideally with unit stride.
- Non-standard loops that cannot be automatically vectorized:
  - loops with irregular memory access pattern
  - calculations with vector dependence
  - while-loops, for-loops with undetermined number of iterations
  - outer loops (unless #pragma simd overrides this restriction)
  - loops with complex branches (i.e., if-conditions)
  - anything else that cannot be, or is very difficult to vectorize.
Multi-Threading: OpenMP
Parallelism in Shared Memory: OpenMP and Intel Cilk Plus

- **Intel Cilk Plus**
  - Good performance “out of the box”
  - Little freedom for fine-tuning
  - Programmer should focus on exposing the parallelism
  - Low-level optimization (thread creation, work distribution and data sharing) is performed by the Cilk Plus library
  - Novel framework

- **OpenMP**
  - Easy to use for simple algorithms
  - For complex parallelism, may require more tuning to perform well
  - Allows more control over synchronization, work scheduling and distribution
  - Well-established framework
Program Structure in OpenMP

```
main() {
  // Begin serial execution.
  ...
  // Only the initial thread executes
  #pragma omp parallel
  {
    // Begin a parallel construct and form
    // a team.
    #pragma omp sections
    {
      // Begin a work-sharing construct.
      #pragma omp section
      {
        // One unit of work.
        {...}
      }
      #pragma omp section
      {
        // Another unit of work.
        {...}
      }
      // Wait until both units of work complete.
      // This code is executed by each team member.
    }
    ...
    // This code is executed by each team member.
    #pragma omp for
    for(...) {
      // Each iteration chunk is unit of work.
      ...
      // Work is distributed among the team members.
    }
    // End of work-sharing construct.
  }
}
```
Program Structure in OpenMP

```c
#pragma omp critical // Begin a critical section.
{...} // Only one thread executes at a time.
#pragma omp task // Execute in another thread without blocking
{...}
...
#pragma omp barrier // Wait for all team members to arrive.
... // This code is executed by each team member.
} // End of Parallel Construct
// Disband team and continue serial execution.
...
// Possibly more parallel constructs.

Code outside #pragma omp parallel is serial, i.e., executed by only one thread
Code directly inside #pragma omp parallel is executed by each thread
Code inside work-sharing construct #pragma omp for is distributed across the threads in the team
```
#include <omp.h>
#include <stdio.h>

int main(){
    const int nt=omp_get_max_threads();
    printf("OpenMP with %d threads\n", nt);

    #pragma omp parallel
        printf("Hello World from thread %d\n", omp_get_thread_num());
}

"Hello World" OpenMP Programs
user@host% export OMP_NUM_THREADS=5
user@host% icpc -openmp hello_omp.cc
user@host% ./a.out
OpenMP with 5 threads
Hello World from thread 0
Hello World from thread 3
Hello World from thread 1
Hello World from thread 2
Hello World from thread 4
user@host% icpc -openmp-stubs hello_omp.cc
hello_omp.cc(8): warning #161: unrecognized #pragma
    #pragma omp parallel
  ~
user@host% ./a.out
OpenMP with 1 threads
Hello World from thread 0
Simultaneously launch multiple threads
Scheduler assigns loop iterations to threads
Each thread processes one iteration at a time
Loop-Centric Parallelism: For-Loops in OpenMP

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

```c
#pragma omp parallel for
for (int i=0; i<n; i++) {
    printf("Iteration %d is processed by thread %d\n", i, omp_get_thread_num());
    // ... iterations will be distributed across available threads...
}
```
Loop-Centric Parallelism: For-Loops in OpenMP

```
#pragma omp parallel
{
    // Code placed here will be executed by all threads.
    // Stack variables declared here will be private to each thread.
    int private_number=0;

#pragma omp for schedule(dynamic, 4)
    for (int i=0; i<n; i++) {
        // ... iterations will be distributed across available threads...
    }
    // ... code placed here will be executed by all threads
}
```
Fork-Join Model of Parallel Execution

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

Fork-join model of parallel execution.

(see #pragma omp task functionality, e.g., here)
Synchronization: Avoiding Unpredictable Program Behavior

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

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

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

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

```
user@host% icpc -o omp-race omp-race.cc -openmp
user@host% ./omp-race
total=208112 (must be 499500)
```
#include <omp.h>
#include <stdio.h>

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

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        #pragma omp critical
        {
            // Only one thread at a time can execute this section
            total = total + i;
        }
    }
}

user@host% icpc -o omp-critical omp-critical.cc -openmp
user@host% ./omp-race
Total=499500 (must be 499500)
This parallel fragment of code has predictable behavior, because the race condition was eliminated with an atomic operation:

```c
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    // Lightweight synchronization
    #pragma omp atomic
    sum += i;
}
```
Synchronization: Avoiding Unpredictable Program Behavior

**Read**: operations in the form $v = x$

**Write**: operations in the form $x = v$

**Update**: operations in the form $x++, x--, --x, ++x, x \ binop = \ expr$
and $x = x \ binop \ expr$

**Capture**: operations in the form $v = x++, v = x-, v = -x, v = ++x,$
$v = x \ binop \ expr$

- Here $x$ and $v$ are scalar variables
- $\ binop$ is one of $+, *, -, -, /, &, ^, |, <, >$.
- No “trickery” is allowed for atomic operations:
  - no operator overload,
  - no non-scalar types,
  - no complex expressions.
# Reduction: Avoiding Synchronization

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

int main() {
    const int n = 1000;
    int sum = 0;
    #pragma omp parallel for reduction(+: sum)
    for (int i = 0; i < n; i++) {
        sum = sum + i;
    }
    printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
}
```

```
user@host% icpc -o omp-reduction omp-reduction.cc -openmp
user@host% ./omp-reduction
sum=499500 (must be 499500)
```
Implementation of Reduction using Private Variables

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

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

#pragma omp parallel
    {
        int sum_th = 0;

#pragma omp for
        for (int i = 0; i < n; i++)
            sum_th = sum_th + i;

#pragma omp atomic
        sum += sum_th;
    }

    printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
}
```
Task Parallelism in Distributed Memory, MPI
The most commonly used framework for distributed memory HPC calculations is the Message Passing Interface (MPI).

Intel MPI library implements MPI for the x86 and for the MIC architectures.
Compiling and Running MPI applications

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 `I_MPI_MIC=1`.

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

4. Launch with the tool `mpirun`  
   - Colon-separated list of executables and hosts (argument `-host hostname`),  
   - Alternatively, use the machine file to list hosts  
   - Coprocessors have hostnames defined in `/etc/hosts`
Left: Gigabit Ethernet bridging on host allows to place coprocessors on the same subnet as hosts (I_MPI_FABRICS=tcp)

Right: Coprocessor Communication Link (CCL) – virtualization of an InfiniBand device on each coprocessor (I_MPI_FABRICS=dapl)
Structure of MPI Applications

```c
#include "mpi.h"

int main(int argc, char** argv) {
    int ret = MPI_Init(&argc,&argv);  // Set up MPI environment
    if (ret != MPI_SUCCESS) {
        MyErrorLogger("...");
        MPI_Abort(MPI_COMM_WORLD, ret);
    }

    int worldSize, myRank, myNameLength;
    char myName[MPI_MAX_PROCESSOR_NAME];
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Get_processor_name(myName, &myNameLength);
    // ... Perform work, exchange messages with MPI_Send, MPI_Recv, etc. ...
    // Terminate MPI environment
    MPI_Finalize();
}
```
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'\n", tag, incomingMsg);
} else if (rank == sender) {
    char outgoingMsg[messageLength];
    strcpy(outgoingMsg, "/Jenny");
    MPI_Send (&outgoingMsg, messageLength, MPI_CHAR, receiver, tag, MPI_COMM_WORLD);
}
Collective Communication: Broadcast

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

![Diagram showing a sender broadcasting data to multiple receivers]
Collective Communication: Scatter

```c
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
               int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```
Collective Communication: Gather

```c
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recv cnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
```
Collective Communication: Reduction

```c
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).
Review: Parallel Scalability
Expressing Parallelism

1. Data parallelism (vectorization)
   - Automatic vectorization by the compiler: portable and convenient
   - For-loops and array notation can be vectorized
   - Compiler hints (#pragma simd, #pragma ivdep, etc.) to assist the compiler

2. Shared-memory parallelism with OpenMP and Intel Cilk Plus
   - Parallel threads access common memory for reading and writing
   - Parallel loops: #pragma omp parallel for and _Cilk_for — automatic work distribution
   - In OpenMP: private and shared variables; synchronization, reduction.

3. Distributed-memory parallelism with MPI
   - MPI processes do not share memory, but can send information to each other
   - All MPI processes execute the same code; role is determined by its rank
   - Point-to-point and collective communication patterns
§5. Optimization for the Intel Xeon Product Family
Optimization Roadmap
Performance Expectations

One Intel Xeon Phi coprocessor vs. Two Intel Xeon Sandy Bridge CPUs

- Up to $2x-3x$ for linear algebraic workloads
- Up to $2x-4x$ for bandwidth-bound and transcendental arithmetics
- Why compare 1 coprocessor against 2 processors? Same thermal design power (TDP).

See also “Intel Xeon Product Family: Performance Brief”
Optimization Checklist

1. Scalar optimization
2. Vectorization
3. Scale above 100 threads
4. Arithmetically intensive or bandwidth-limited
5. Efficient cooperation between the host and the coprocessor(s)
Finding Bottlenecks with Intel VTune Amplifier
Intel VTune Parallel Amplifier XE

Hardware event-based profiler for parallel applications on Xeon CPUs and Xeon Phi coprocessors.

Bottleneck detection down to a single line of code, hardware event collection, minimal impact on performance.
Using VTune

Setting up a VTune project:

Results of profiling, bottom-up view:
Using VTune

Locating hotspots down to a single line of code:
Using VTune

Analyzing custom events
MPI Diagnostics Using Intel Trace Analyzer and Collector
Intel Trace Analyzer and Collector

Profiler for MPI Applications on Xeon and Xeon Phi architectures.

Graphical user interface, visualization of computation and communication.
Using Intel Trace Analyzer and Collector

user@host% source /opt/intel/itac/8.1.0.024/bin/itacvars.sh
user@host% source /opt/intel/itac/8.1.0.024/mic/bin/itacvars.sh
user@host% mpiicpc -mkl -o pi_mpi pi_mpi.c
user@host% mpiicpc -mmic -mkl -o pi_mpi.mic pi_mpi.c
user@host% scp pi_mpi.mic mic0:~/
pi_mpi.mic 100% 433KB 432.5KB/s 00:00
user@host% export VT_LOGFILE_FORMAT=stfsingle
user@host% mpirun -trace -n 32 -host localhost ./pi_mpi : \
% -n 240 -host mic0 ~/pi_mpi.mic
Time, s: 0.36
[0] Intel(R) Trace Collector INFO: Writing tracefile pi_mpi.single.stf
in /home/user/pi
user@host% traceanalyzer pi_mpi.single.stf
Using Intel Trace Analyzer and Collector
Intel Math Kernel Library (MKL)
Intel Math Kernel Library (MKL)

Linear algebra, fast Fourier transforms, vector math, parallel random numbers, statistics, data fitting, sparse solvers.

Intel MKL functions are optimized for Xeon Processors as well as for Xeon Phi coprocessors.
Using Intel MKL

Three modes of usage:

- **Automatic Offload**
  - No code change required to offload calculations to a Xeon Phi coprocessor
  - Automatically uses both the CPU and the coprocessor
  - The library takes care of data transfer and execution management

- **Compiler-Assisted Offload**
  - Programmer maintains explicit control of data transfer and remote execution
  - Requires using compiler offload pragmas and directives

- **Native Execution**
  - Uses an Intel Xeon Phi coprocessor as an independent compute node.
  - Data initialized & processed on the coprocessor, or communicated via MPI
Using MKL in Automatic Offload Mode

Calling an MKL function from host code:

```c
sgemm(&transa, &transb, &SIZE, &SIZE, &SIZE, &alpha,
    A, &newLda, B, &newLda, &beta, C, &SIZE);
```

Compiling and running the code. Calculation will be offloaded to a Xeon Phi coprocessor, if one is available at runtime.

```
user@host% icpc -c mycode.cc -mkl -o mycode
user@host% export MKL_MIC_ENABLE=1
user@host% ./mycode
```
Using MKL in Compiler-Assisted Offload Mode

Calling an MKL function from offloaded section:

```c
#pragma offload target(mic) \
in(transa, transb, N, alpha, beta) \ 
in(A:length(matrix_elements)) \ 
in(B:length(matrix_elements)) \ 
out(C:length(matrix_elements) alloc_if(0))
{
    sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, &beta, C, &N);
}
```

Compiling and running the code. If no coprocessor at runtime, MKL will fall back to CPU calculation.

```
user@host% icpc -c mycode.cc -mkl -o mycode
user@host% ./mycode
```
Using MKL Native Execution Mode

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    const size_t N = 1<<29L;
    const size_t F = sizeof(float);
    float* A = (float*)malloc(N*F);
    srand(0); // Initialize RNG
    for (int i = 0; i < N; i++) {
        A[i]=(float)rand() / (float)RAND_MAX;
    }
    printf("Generated %ld random \nnumbers\nA[0]=%e\n", N, A[0]);
    free(A);
}
```
Using MKL in Native Execution Mode

```
user@host% icpc -mmic -o rand % rand.cc
user@host% # Run on coprocessor
user@host% # and benchmark
user@host% time micnativeloadex % rand
Generated 536870912 random numbers
A[0]=8.401877e-01
real 0m56.591s
user 0m0.002s
sys 0m0.011s
```

```
user@host%  icpc -mkl -mmic -o % rand-mkl rand-mkl.cc
user@host% export SINK_LD_LIBRARY_PATH=/opt/intel/composerxe/mkl/lib/mic:
% /opt/intel/composerxe/lib/mic
user@host% time micnativeloadex rand-mkl
Generated 536870912 random numbers
A[0]=1.343642e-01
real 0m7.951s
user 0m0.053s
sys 0m0.168s
```

On Intel Xeon Phi coprocessor, *random number generation* with Intel MKL is 7x faster than with the C standard Library.
Scalar Optimization Considerations
Optimization Level

The default optimization level -O2

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

Optimization level -O3

- enables more aggressive optimization
- loop fusion
- block-unroll-and-jam
- if-statement collapse

```c
#pragma intel optimization_level 3
void my_function() {
  //...
}
```
#include <stdio.h>

int main() {
  const int N = 1 << 28;
  double w = 0.5;
  double T = (double)N;
  double s = 0.0;
  for (int i = 0; i < N; i++)
    s += w * (double)i / T;
  printf("%e\n", s);
}

user@host% icpc noconst.cc
user@host% time ./a.out
6.710886e+07
real 0m0.461s
user 0m0.460s
sys 0m0.001s

user@host% icpc const.cc
user@host% time ./a.out
6.710886e+07
real 0m0.097s
user 0m0.094s
sys 0m0.003s
Array Reference by Index instead of Pointer Arithmetics

With **Pointer arithmetics**, the code is 5x slower than with reference to array elements by index.
The value of \(\sin_A\) can be calculated once and re-used \(m\) times in the \(j\)-loop

In some cases, at -O2 compiler eliminates common subexpressions automatically
Ternary if-operator Trap

- Ternary if operator ( ? : ) is a short-hand for if ... else
- Example: the `min()` function as a pre-processor expression

```c
#define min(a, b) ((a) < (b) ? (a) : (b))

const float c = min(my_function(x), my_function(y));
```

- Problem: line 2 calls `my_function()` 3 times
- Optimization:

```c
#define min(a, b) ((a) < (b) ? (a) : (b))

const float result_a = my_function(x);
const float result_b = my_function(y);
const float c = min(result_a, result_b);
```
Strength Reduction

Replace expensive operations with a combination of fast operations.

**Example 1**: replacing division with multiplication by the precomputed reciprocal:

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

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

**Example 2**: algebraic transformations to replace two divisions with one

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

```c
for (int i = 0; i < n; i++) {
    A[i] = B[i]/(C[i]*D[i]);
    E[i] = (A[i]*D[i] + B[i]*C[i])/(B[i]*D[i]);
}
```
Consistency of Precision: Constants

1. Operations on type `float` is faster than operations on type `double`. Avoid type conversions and define single-precision literal constants with suffix `-f`.

```cpp
const double twoPi = 6.283185307179586;
const float phase = 0.3f; // single precision
```

2. Use 32-bit `int` values including 64-bit `long` where possible, including array indices. Avoid type conversions and define 64-bit literal constants with suffix `-L` or `UL`.

```cpp
const long N2 = 1000000*1000000; // Overflow error
const long N3 = 1000000L*1000000L; // Correct
```
Consistency of Precision: Functions

1. `math.h` contains fast single precision versions of arithmetic functions ending with suffix `-f`:
   ```c
   double sin(double x);
   float sinf(float x);
   ```

2. `math.h` contains fast base 2 exponential and logarithmic functions:
   ```c
   double exp(double x); // Double precision, natural base
   float expf(float x); // Single precision, natural base
   double exp2(double x); // Double precision, base 2
   float exp2f(float x); // Single precision, base 2
   ```
Floating-Point Semantics

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-fp-model strict</td>
<td>Only value-safe optimizations calculations are reproducible from run to run exceptions controlled using -fp-model except (default) Value-unsafe optimizations are allowed better performance at the cost of lower accuracy</td>
</tr>
<tr>
<td>-fp-model fast=1</td>
<td>Intermediate arithmetic results are rounded to the precision defined in the source code.</td>
</tr>
<tr>
<td>-fp-model fast=2</td>
<td>Intermediate arithmetic results are rounded to 53-bit (double) precision.</td>
</tr>
<tr>
<td>-fp-model source</td>
<td>Intermediate arithmetic results are rounded to 64-bit (extended) precision.</td>
</tr>
<tr>
<td>-fp-model double</td>
<td>controls floating-point exception semantics.</td>
</tr>
</tbody>
</table>
Precision Control for Transcendental Functions

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

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

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

-fimf-domain-exclusion= n[:funclist] Defines a list of special-value numbers that do not need to be handled.

int n derived by the bitwise OR of types:
- extremes: 1, NaNs: 2, infinites: 4, denormals\(^1\): 8, zeroes: 16.

\(^1\)by default, on Intel Xeon Phi, denormals are flushed to zero in hardware, but supported in SVML
```c
#include <stdio.h>
#include <math.h>

int main() {
    const int N = 1000000;
    const int P = 10;
    double A[N];
    const double startValue = 1.0;
    A[:] = startValue;
    for (int i = 0; i < P; i++)
        #pragma simd
        for (int r = 0; r < N; r++)
            A[r] = exp(-A[r]);

    printf("Result=%.17e\n", A[0]);
}
```
Precision Control for Transcendental Functions

User@host% icpc -o precision-1 -mmic \%
%fimf-precision=low precision.cc
User@host% scp precision-1 mic0:~/%
precision-1 100% 11KB 11.3KB/s
User@host% ssh mic0 time ./precision-1
Result=5.68428695201873779e-01
real 0m 0.08s
user 0m 0.06s
sys 0m 0.02s
User@host%

User@host% icpc -o precision-2 -mmic \%
%fimf-precision=high precision.cc
User@host% scp precision-2 mic0:~/%
precision-2 100% 19KB 19.4KB/s
User@host% ssh mic0 time ./precision-2
Result=5.68428725029060722e-01
real 0m 0.14s
user 0m 0.12s
sys 0m 0.02s
User@host%
Automatic Vectorization: Making it Happen and Tuning
Challenges with Optimizing Vectorization on Xeon Phi

- Must utilize 512-bit vector registers (16 float or 8 double)
- Must convince compiler that vectorization is possible
- Preferably unit-stride access to data
- Preferably align data on 64-byte boundary
- Avoid branches in vector loops
- Guide compiler regarding expected iteration count, memory alignment, outer loop vectorization, etc.

This section:
Ensuring that automatic vectorization succeeds where it must exist.
Diagnosing the Utilization of Vector Instructions

When porting and optimizing an application:
- Find performance-critical parts
- Use `-vec-report3` to get information about automatic vectorization
- Use Intel VTune Amplifier XE to diagnose the executable
- Benchmark regular compilation *vs. -no-vec -no-simd case*
- Provide additional information to the compiler about loops in form of `#pragma`
Assumed Vector Dependence. The restrict Keyword.

- **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
user@host% icpc -vec-report3 \
    -c vdep.cc
vdep.cc(2): (col. 3) remark:
    loop skipped: multiversioned.
    loop was not vectorized:
    not inner loop.
```
Ignoring Assumed Vector Dependence

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];
}
```

```
user@host% icpc -vec-report3 -c vdep.cc
vdep.cc(3): (col. 3) remark:
  LOOP WAS VECTORIZED.
vdep.cc(3): (col. 3) remark:
  loop was not vectorized:
  not inner loop.
```
Pointer Disambiguation (alternative to \#pragma ivdep)

- restrict keyword applies to each pointer variable qualified with it
- The object accessed by the pointer is only accessed by that pointer in the given scope
- The compiler argument -restrict must be used.

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

```bash
user@host% icpc -vec-report3 -restrict -c vdep.cc
vdep.cc(2): (col. 3) remark: LOOP WAS VECTORIZED.
vdep.cc(2): (col. 3) remark: loop was not vectorized: not inner loop.
```
Automatic Vectorization: Data Structures
Challenges with Optimizing Vectorization on Xeon Phi

- Must utilize 512-bit vector registers (16 float or 8 double)
- Must convince compiler that vectorization is possible
- Preferably unit-stride access to data
- Preferably align data on 64-byte boundary
- Avoid branches in vector loops
- Guide compiler regarding expected iteration count, memory alignment, outer loop vectorization, etc.

The rule of thumb for achieving unit-stride access
Use structures of arrays (SoA) instead of arrays of structures (AoS)
Example: Unit-Stride Access in Coulomb’s Law Application

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

(1)

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

(2)
Elegant, but Inefficient Solution: Array of Structures

```c
struct Charge { // Elegant, but ineffective data layout
    float x, y, z, q;
} chgs[m]; // Coordinates and value of this charge

for (int i=0; i<m; i++) { // This loop will be auto-vectorized
    // Non-unit stride: (&chg[i+1].x - &chg[i].x) != sizeof(float)
    const float dx=chg[i].x - Rx;
    const float dy=chg[i].y - Ry;
    const float dz=chg[i].z - Rz;
    phi -= chg[i].q / sqrtf(dx*dx+dy*dy+dz*dz); // Coulomb’s law
}
```
Arrays of Structures versus Structures of Arrays

Array of Structures (AoS)

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

Structure of Arrays (SoA)

```c
struct Charge_Distribution {
    // Data layout permits effective vectorization of Coulomb’s law application
    const int m; // Number of charges
    float * x; // Array of x-coordinates of charges
    float * y; // ...y-coordinates...
    float * z; // ...etc.
    float * q; // These arrays are allocated in the constructor
};
```
struct Charge_Distribution {
    // Data layout permits effective vectorization of Coulomb’s law application
    const int m;  // Number of charges
    float *x, *y, *z, *q;  // Arrays of x-, y- and z-coordinates of charges
};

// This version vectorizes better thanks to unit-stride data access
for (int i=0; i<chg.m; i++) {
    // Unit stride: (&chg.x[i+1] - &chg.x[i]) == sizeof(float)
    const float dx=chg.x[i] - Rx;
    const float dy=chg.y[i] - Ry;
    const float dz=chg.z[i] - Rz;
    phi -= chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);
}
Electric Potential Calculation with Coulomb’s Law

Electric potential calculation

Host system
Intel Xeon Phi Coprocessor

Non-unit stride
(array of structures)
Unit-stride
(structure of arrays)
Unit-stride with
relaxed precision

Time, s (lower is better)

<table>
<thead>
<tr>
<th></th>
<th>Host system</th>
<th>Intel Xeon Phi Coprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-unit stride</td>
<td>0.90 s</td>
<td>0.73 s</td>
</tr>
<tr>
<td>Unit-stride</td>
<td>0.51 s</td>
<td>0.37 s</td>
</tr>
<tr>
<td>Unit-stride with relaxed precision</td>
<td>0.51 s</td>
<td>0.22 s</td>
</tr>
</tbody>
</table>
Automatic Vectorization: Data Alignment
Challenges with Optimizing Vectorization on Xeon Phi

- Must utilize 512-bit vector registers (16 float or 8 double)
- Must convince compiler that vectorization is possible
- Preferably unit-stride access to data
- Preferably align data on 64-byte boundary
- Avoid branches in vector loops
- Guide compiler regarding expected iteration count, memory alignment, outer loop vectorization, etc.

This section: Data alignment and compiler hints.
Data Alignment

- `char* p` points to an address aligned on an `n`-byte boundary if `((size_t)p%n==0)`.
- 128-bit SSE load and store instructions require 16-byte alignment,
- 256-bit AVX load and store instructions do not require alignment,
- 512-bit IMCI load and store instructions require 64-byte alignment.
Data Alignment

Data alignment on the stack

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

- The address of A[0] is a multiple of 64, i.e., aligned on a 64-byte boundary.
- Setting a very high alignment value may lead to wasted virtual memory.

Alignment of memory blocks on the heap

#include <malloc.h>

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

_mm_malloc and _mm_free are aligned version of malloc and free:
- the header file malloc.h must be included
Data Alignment Hints

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

```c
float* packedData = _mm_malloc(sizeof(float)*nData, 64);
float* inVector = _mm_malloc(sizeof(float)*nRows, 64);
// ... Pragma vector aligned promises to the compiler that elements of array
// used in the first iteration are 64-byte boundary aligned.
#pragma vector aligned
for (int c = 0; c < blockLen[idx]; c++) // blockLen[idx] are multiples of 64
    sum += packedData[offs+c]*inVector[j0+c];
    outVector[i] += sum;
// ...
_mm_free(packedData); _mm_free(inVector);
```

This can lead to significant speedups, because compiler will not implement runtime checks for alignment situation.
Data Alignment and Padding

Note: when relying on `#pragma vector aligned`, may need to pad the inner dimension on data structures to a multiple of 16 (in single precision) or 8 (double precision).

```c
void GaussEl(const int n, const int m, const int start, float* const matrix) {
    for (int i = start+1; i < n; i++) {
        const float factor = matrix[(i-1)*m]/matrix[i*m];
        #pragma vector aligned
        for (int j = 0; j < m; j++)
            matrix[i*m + j] += factor*matrix[(i-1)*m + j];
    }
    // ... Padding inner dimension and allocating matrix
    if (m % 16 != 0) m += (16 - m%16);
    matrix = (float*)_mm_malloc(n*m*sizeof(float), 64);
    //...
    GaussEl(n, m, 0, matrix);
```

MIC Developer Boot Camp Rev. 12
Automatic Vectorization: Data Alignment
Vectorization Pragmas, Keywords and Compiler Arguments

- `#pragma simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `restrict qualifier and -restrict command-line argument`
- `#pragma loop count`
- `__assume_aligned keyword`
- `-vec-report[n]`
- `-O[n]`
- `-x[code]`
Vectorization Pragmas, Keywords and Compiler Arguments

- #pragma simd
- #pragma vector always
- #pragma vector aligned | unaligned
- #pragma vector nontemporal | temporal
- #pragma novector
- #pragma ivdep
- restrict qualifier and -restrict command-line argument
- #pragma loop count
- __assume_aligned keyword
- -vec-report[n]
- -O[n]
- -x[code]
Thread Parallelism: Reducing Synchronization
Challenges with Thread Parallelism on Xeon Phi

- Multi-core CPU: 4–48 threads, Xeon Phi: 228–244 threads.
- Must have enough parallelism to keep all cores busy
- Must have less synchronization than on CPU
- Must have lower per-thread memory overhead
- Must access core-local data whenever possible
- Must co-exist with vectorization in each core
Example: Dealing with Excessive Synchronization

Computing a histogram \((m << n)\) with a serial code:

```c
void Histogram(const float* age, int* const hist, const int n,
               const float group_width, const int m) {
    for (int i = 0; i < n; i++) {
        const int j = (int) (age[i] / group_width);
        hist[j]++;
    }
}
```

- Code cannot be automatically vectorized
- True vector dependence
void Histogram(const float* age, int* const hist, const int n, 
    const float group_width, const int m) {
    const int vecLen = 16; // Length of vectorized loop
    const float invGroupWidth = 1.0f/group_width; // Pre-compute the reciprocal
    // Strip-mining the loop in order to vectorize the inner short loop
    // Note: this algorithm assumes n%vecLen == 0.
    for (int ii = 0; ii < n; ii += vecLen) {
        // Temporary store vecLen indices
        int histIdx[vecLen] __attribute__((aligned(64)));
        // Vectorize the multiplication and rounding
        #pragma vector aligned
        for (int i = ii; i < ii + vecLen; i++)
            histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
        // Scattered memory access, does not get vectorized
        for (int c = 0; c < vecLen; c++)
            hist[histIdx[c]]++;
    }
}
# Adding Thread Parallelism

```c
#pragma omp parallel for schedule(guided)
    for (int ii = 0; ii < n; ii += vecLen) {
        int histIdx[vecLen] __attribute__((aligned(64)));
        #pragma vector aligned
        for (int i = ii; i < ii + vecLen; i++)
            histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
        for (int c = 0; c < vecLen; c++)
            // Protect the ++ operation with the atomic mutex (inefficient!)
            #pragma omp atomic
            hist[histIdx[c]]++;
    }
```

MIC Developer Boot Camp Rev. 12

Thread Parallelism: Reducing Synchronization

Improving Thread Parallelism

```c
#pragma omp parallel
{
    int hist_priv[m]; // Better idea: thread-private storage
    hist_priv[:] = 0;
    int histIdx[vecLen] __attribute__((aligned(64)));

#pragma omp for schedule(guided)
    for (int ii = 0; ii < n; ii += vecLen) {

#pragma vector aligned
        for (int i = ii; i < ii + vecLen; i++)
            histIdx[i-ii] = (int) ( age[i] * invGroupWidth );
        for (int c = 0; c < vecLen; c++)
            hist_priv[histIdx[c]]++;
    }

    for (int c = 0; c < m; c++) {

#pragma omp atomic
        hist[c] += hist_priv[c];
    }
}
```
Dealing with Excessive Synchronization

Computing a histogram: elimination of synchronization

- Scalar Serial Code: 71.30 s
- Vectorized Serial Code (Atomic Operations): 5.06 s
- Vectorized Parallel Code: 1.27 s
- Vectorized Parallel Code (Private Variables): 9.23 s
- Vectorized Parallel Code (Atomic Operations): 24.00 s
- Vectorized Parallel Code (Private Variables): 37.70 s

Host system: 24.00 s
Intel Xeon Phi coprocessor: 0.07 s
Thread Parallelism: False Sharing
False Sharing is similar to a race condition. Threads accessing the same cache line are caused by coherent caches. The cache line is 64-byte wide in modern Intel architectures.
The value of \( m = 5 \) is small

Array elements \( \text{hist}_\text{thr}[0][:] \) are within \( m \times \text{sizeof}(\text{int}) = 20 \) bytes of array elements \( \text{hist}_\text{thr}[1][:] \)
Padding to Avoid False Sharing

// Padding for hist_thr[][] in order to avoid a situation
// where two (or more) rows share a cache line.
const int paddingBytes = 64;
const int paddingElements = paddingBytes / sizeof(int);
const int mPadded = m + (paddingElements - m % paddingElements);
// Shared histogram with a private section for each thread
int hist_thr[nThreads][mPadded];
hist_thr[::][::] = 0;
Padding to Avoid False Sharing

Computing a histogram: elimination of false sharing

- Host system
- Intel Xeon Phi coprocessor
Thread Parallelism: Expanding Iteration Space
Example: Dealing with Insufficient Parallelism

\[ S_i = \sum_{j=0}^{n} M_{ij}, \quad i = 0 \ldots m. \] (3)

- \( m \) is small, smaller than the number of threads in the system
- \( n \) is large, large enough so that the 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++) {
        long sum=0;
        #pragma simd
        #pragma vector aligned
        for (int j=0; j<n; j++)
            sum+=M[i*n+j];
        s[i]=sum; }}
```
Dealing with Insufficient Parallelism

VTune Analysis: Row-Wise Reduction of a Short, Wide Matrix
Strip-Mining: Simultaneous Thread and Data Parallelism

```c
// Compiler may be able to simultaneously parallelize and auto-vectorize it
#pragma omp parallel for
#pragma simd
for (int i = 0; i < n; i++) {
    // ... do work
}

// The strip-mining technique separates parallelization from vectorization
const int STRIP=1024;
#pragma omp parallel for
for (int ii = 0; ii < n; ii += STRIP)
#pragma simd
    for (int i = ii; i < ii + STRIP; i++) {
        // ... do work
    }
```
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 sum[m]; sum[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 simd
                #pragma vector aligned
                for (int j = jj; j < jj + STRIP; j++)
                    sum[i] += M[i*n + j];
        // Reduction
        for (int i = 0; i < m; i++)
            #pragma omp atomic
            s[i] += sum[i];
    }  }
Exposing Parallelism: Strip-Mining and Loop Collapse
Dealing with Insufficient Parallelism

Row-Wise Reduction of a Short, Wide Matrix

Parallel row-wise matrix reduction

- Host system
- Intel Xeon Phi Coprocessor

Performance, GB/s (higher is better)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Performance, GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>47.5</td>
</tr>
<tr>
<td>Parallel inner loop</td>
<td>84.9</td>
</tr>
<tr>
<td>Collapse nested loops</td>
<td>28.3</td>
</tr>
<tr>
<td>Strip-mine and collapse</td>
<td>131.6</td>
</tr>
</tbody>
</table>

Host system
Intel Xeon Phi Coprocessor
Thread Parallelism: Affinity
Setting Thread Affinity

- OpenMP threads may migrate from one core to another according to OS decisions.
- Forbid migration — increase the performance.
- Control: environment variable `KMP_AFFINITY`
Uses of Thread Affinity

- **Bandwidth-bound applications:** 1 thread per core + prevent migration. Optimizes utilization of memory controllers.

- **Compute-bound applications:** 2 (Xeon) or 4 (Xeon Phi) threads per core + prevent migration. Ensures that threads consistently access local L1 cache data (+L2 for Xeon Phi).

- **Offload applications:** physical core 0 on Xeon Phi is used by $\mu$OS for offload tasks. Prevent placing compute threads on that core.

- **Applications in multi-socket NUMA (Non-Uniform Memory Access) systems:** partition the system for two independent tasks, pin tasks to respective CPUs.
The KMP_AFFINITY Environment Variable

KMP_AFFINITY= [<modifier>,...]<type>[,<permute>][,<offset>]

modifier:
- verbose/nonverbose
- respect/norespect
- warnings/nowarnings
- granularity=core or thread

- type=compact, scatter or balanced
- type=explicit, proclist=[<proc_list>]
- type=disabled or none.

user@host% export MIC_ENV_PREFIX=MIC
user@host% export KMP_AFFINITY=compact,granularity=fine
user@host% export MIC_KMP_AFFINITY=balanced,granularity=fine
Bandwidth-bound, KMP_AFFINITY=scatter

```
user@host% export OMP_NUM_THREADS=32
user@host% export KMP_AFFINITY=none
user@host% for i in {1..4} ; do ./rowsum_stripmine | tail -1; done
Problem size: 2.980 GB, outer dimension: 4, threads: 32
Strip-mine and collapse: 0.061 +/- 0.002 seconds (52.89 +/- 1.31 GB/s)
Strip-mine and collapse: 0.059 +/- 0.002 seconds (54.11 +/- 1.56 GB/s)
Strip-mine and collapse: 0.077 +/- 0.001 seconds (41.71 +/- 0.69 GB/s)
Strip-mine and collapse: 0.070 +/- 0.005 seconds (45.59 +/- 3.14 GB/s)
user@host% export OMP_NUM_THREADS=16
user@host% export KMP_AFFINITY=scatter
user@host% for i in {1..4}; do ./rowsum_stripmine | tail -1 ; done
Problem size: 2.980 GB, outer dimension: 4, threads: 16
Strip-mine and collapse: 0.059 +/- 0.004 seconds (54.47 +/- 3.25 GB/s)
Strip-mine and collapse: 0.061 +/- 0.004 seconds (52.30 +/- 3.30 GB/s)
Strip-mine and collapse: 0.062 +/- 0.005 seconds (51.37 +/- 4.29 GB/s)
Strip-mine and collapse: 0.058 +/- 0.001 seconds (55.48 +/- 1.27 GB/s)
```
double* A = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
double* B = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
double* C = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);

for(int k = 0; k < nIter; k++) {
    dgemm(&tr, &tr, &N, &N, &N, &v, A, &Nld, B, &Nld, &v, C, &N);
    double flopsNow = (2.0*N*N*N+1.0*N*N)*1e-9/(t2-t1);
    printf("Iteration %d: %.1f GFLOP/s\n", k+1, flopsNow);
}
_mm_free(A); _mm_free(B); _mm_free(C);
Compute-Bound, KMP_AFFINITY=compact/balanced

```
user@host% icpc -o bench-dgemm -mkl -mmic bench-dgemm.cc
user@host% micnativeloadex ./bench-dgemm
Iteration 1: 312.7 GFLOP/s
Iteration 2: 346.5 GFLOP/s
Iteration 3: 348.5 GFLOP/s
Iteration 4: 347.2 GFLOP/s
Iteration 5: 348.3 GFLOP/s

user@host% micnativeloadex ./bench-dgemm -e "KMP_AFFINITY=compact"
Iteration 1: 626.8 GFLOP/s
Iteration 2: 769.1 GFLOP/s
Iteration 3: 769.4 GFLOP/s
Iteration 4: 769.3 GFLOP/s
Iteration 5: 769.4 GFLOP/s
```
Other Optimization Topics for Thread Parallelism

Examples found in our 4-day training and in the book:

- Avoiding excessive synchronization with reduction
- Load balancing across threads
- Using thread affinity to partition a multi-socket NUMA system
§6. Advanced Optimization for the MIC Architecture
Memory Access and Cache Utilization
Challenges with Memory Access on Xeon Phi

- More threads than CPU, same amount of Level-2 cache (~30 MB)
- No hardware prefetching from Level-2 to Level-1
- High penalty for data page walks
- Dynamic memory allocation is serial → greater penalty than CPU per Amdahl’s law

“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).
Loop Tiling (Blocking)

/* Nested loops without tiling. Array B[] does not fit into cache */
for (int i = 0; i < iMax; ++i)
    for (int j = 0; j < jMax; ++j)
        PerformWork(A[i], B[j]);

/* Tiled nested loops */
for (int jj = 0; jj < jMax; jj+= T)
    for (int i = 0; i < iMax; ++i)
        for (int j = jj; j < jj+T; ++j)
            PerformWork(A[i], B[j]);

Example: tile size T=2
cache size=3

Cache Hit Rate = 6/16  
SLOWER

Cache Hit Rate = 10/16  
FASTER
Loop Tiling (Blocking)

// Plain nested loops
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        compute(a[i], b[j]); // Memory access is unit-stride in j

// Tiled nested loops
for (int ii = 0; ii < m; ii += TILE)
    for (int j = 0; j < n; j++)
        for (int i = ii; i < ii + TILE; i++) // Re-use data for each j with several i
            compute(a[i], b[j]); // Memory access is unit-stride in j

// Doubly tiled nested loops
for (int ii = 0; ii < m; ii += TILE)
    for (int jj = 0; jj < n; jj += TILE)
        for (int i = ii; i < ii + TILE; i++) // Re-use data for each j with several i
            for (int j = jj; j < jj + TILE; j++)
                compute(a[i], b[j]); // Memory access is unit-stride in j
Optimization Example: In-Place Square Matrix Transposition

```c
#pragma omp parallel for
    for (int i = 0; i < n; i++) {
        // Distribute across threads
        for (int j = 0; j < i; j++) {
            // Employ vector load/stores
            const double c = A[i*n + j]; // Swap elements
            A[i*n + j] = A[j*n + i];
            A[j*n + i] = c;
        }
    }
```

Unoptimized code:

- Large-stride memory accesses
- Inefficient cache use
- Does not reach memory bandwidth limit
Tiling a Parallel For-Loop (Matrix Transposition)

```c
#pragma omp parallel for
for (int ii = 0; ii < n; ii += TILE) { // Distribute across threads
    const int iMax = (n < ii+TILE ? n : ii+TILE); // Adapt to matrix shape
    for (int jj = 0; jj <= ii; jj += TILE) { // Tile the work
        for (int i = ii; i < iMax; i++) {
            const int jMax = (i < jj+TILE ? i : jj+TILE); // for whole matrix
            #pragma loop count avg(TILE) // Vectorization tuning
            #pragma simd // Vectorization hint
            for (int j = jj; j<jMax; j++) {
                const double c = A[i*n + j]; // Swap elements
                A[i*n + j] = A[j*n + i];
                A[j*n + i] = c;
            }
        }
    }
}
```

Better (but not optimal) solution:
- Loop tiling to improve locality of data access
- Not enough outer loop iterations to keep 240 threads busy
Further Optimization of Matrix Transposition

- Multi-versioned inner loop for diagonal, edges and body
- Tuning pragma to enforce non-temporal stores
- Expand parallel iteration space occupy all threads
- Control data alignment
- OpenMP thread affinity for bandwidth optimization
Further Optimization: Code Snippet

```c
#pragma omp parallel
{
#pragma omp for schedule(guided)
    for (int k = 0; k < nTilesParallel; k++) { // Bulk of calculations here
        const int ii = plan[HEADER_OFFSET + 2*k + 0]*TILE; // Planned order
        const int jj = plan[HEADER_OFFSET + 2*k + 1]*TILE; // of operations
        for (int j = jj; j < jj+TILE; j++) { // Simplified main microkernel
            #pragma simd // Vectorization hint
            #pragma vector nontemporal // Cache traffic hint
            for (int i = ii; i < ii+TILE; i++) { // Constant loop count (good)
                const double c = A[i*n + j]; // Swap elements
                A[i*n + j] = A[j*n + i];
                A[j*n + i] = c;
            }
        }
    }
    // Transposing the tiles along the main diagonal and edges...
    // ...
}
```

- Longer code but still in the C language; works for CPU and MIC
Arithmetic Intensity and Roofline Model

Theoretical estimates, Intel Xeon Phi coprocessor

Arithmetic Performance = \(60 \times 1.0 \times (512/64) \times 2 = 960\) GFLOP/s.

Memory Bandwidth = \(\eta \times 6.0 \times 8 \times 2 \times 4 = \eta \times 384\) GB/s,

Peak performance for:
- 60-core Intel Xeon Phi
- clocked at 1.0 GHz
- 512-bit SIMD registers
- 64-bit floating-point numbers
- fused multiply-add

The peak memory bandwidth:
- \(\eta \approx 0.5\) – practical efficiency
- 6.0 GT/s (Transfers)
- 8 memory controllers
- 2 channels in each
- 4 bytes per channel
**Arithmetic Intensity and Roofline Model**

Theoretical estimates, Intel Xeon Phi coprocessor

Arithmetic Performance = \(60 \times 1.0 \times \frac{512}{64} \times 2 = 960\) GFLOP/s.

Memory Bandwidth = \(\eta \times 6.0 \times 8 \times 2 \times 4 = \eta \times 384\) GB/s,

To saturate Arithmetic and Logic Units (ALUs):

384/8 = 48 billion floating-point numbers per second should be delivered from memory to the cores (double precision)

960/48 = 20 floating-point operations (multiplication/addition) must be performed on every number fetched from the main memory
Arithmetic Intensity and Roofline Model

Theoretical estimates, 2x 8-core Intel Xeon E5 processors at 3.0 GHz

Arithmetic Performance = 2 sockets × 8 × 3.0 × (256/64) × 2 = 384 GFLOP/s,

Memory Bandwidth = 2 sockets × η × 6.4 × 8 = η × 102 GB/s,

Peak performance for:

- 16 Intel Xeon cores
- clocked at 3.0 GHz
- 256-bit SIMD registers
- 64-bit floating-point numbers
- 2 ALUs

The peak memory bandwidth:

- η ≈ 0.5 – practical efficiency
- 6.4 GT/s (Transfers)
- 8 bytes per transfer
Arithmetic Intensity and Roofline Model

Theoretical estimates, 2x 8-core Intel Xeon E5 processors at 3.0 GHz

Arithmetic Performance = 2 sockets × 8 × 3.0 × (256/64) × 2 = 384 GFLOP/s,

Memory Bandwidth = 2 sockets × η × 6.4 × 8 = η × 102 GB/s,

To saturate Arithmetic and Logic Units (ALUs):

102/8 ≈ 13 billion floating-point numbers per second should be delivered from memory to the cores (double precision)

384/13 = 30 floating-point operations (multiplication/addition) must be performed on every number fetched from the main memory
Arithmetic Intensity and Roofline Model

Roofline model: theoretical peak

More on roofline model: Williams et al.
Other Topics on Memory Traffic Optimization

Discussions found in our 4-day training and in the book:

- Recursive cache-oblivious algorithms
- Cross-procedural loop fusion
- Software prefetching
Data Persistence and PCIe Traffic
Memory Retention Between Offloads

// Allocate arrays on coprocessor during the first iteration;
// retain allocated memory for subsequent iterations
#pragma offload target(mic:0) \
    in(data: length(size) alloc_if(k==0) free_if(k==nTrials-1) align(64))
{
    // offloaded code here...
}

- Data transfer across the PCIe bus rate is 6 GB/s
- To allocate memory on the coprocessor – 0.5 GB/s
- The memory allocation operation is serial and therefore slow
- Memory retention reduces the latency by a factor of 10x
- For smaller arrays, the effect is even more dramatic
Offload Latency With and Without Memory/Data Retention

Offload latencies

- Default offload (allocation + data transfer + deallocation)
- With memory retention (data transfer only)
- With data persistence (no memory allocation or data transfer)

Array Size

- 1 KB
- 2 KB
- 4 KB
- 8 KB
- 16 KB
- 32 KB
- 64 KB
- 128 KB
- 256 KB
- 512 KB
- 1 MB
- 2 MB
- 4 MB
- 8 MB
- 16 MB
- 32 MB
- 64 MB
- 128 MB
- 256 MB
- 512 MB
- 1 GB

Latency, ms

- 0.1
- 1
- 10
- 100
- 1000
MPI Applications on Clusters with Coprocessors
MPI: Fabrics
**MPI Fabric Selection: Ethernet and InfiniBand**

- Ethernet+TCP between coprocessors slower than the hardware limit
- InfiniBand approaches the hardware limit from CPU to coprocessors

![Graphs showing latency and bandwidth for different message sizes and fabric combinations.](http://research.colfaxinternational.com/)
MPI Fabric Selection: Ethernet and InfiniBand

- InfiniBand requires additional software on top of MPSS
- Environment variable `I_MPI_FABRICS`
- More information in white paper
MPI Fabric Selection: Intra-Device Fabric

- Part of CCL: virtual interface ibscif for communication between coprocessors within a system
- Default Combination: I_MPI_FABRICS=shm:dapl
- shm provides better latency, dap1 – greater bandwidth

![Graph showing latency and bandwidth](http://research.colfaxinternational.com/)
Communication Efficiency with Symmetric Clustering

- MPI communication between CPU and coprocessors as efficient as offload
- Peer-to-peer communication not uniform, but better than with Gigabit Ethernet

White paper with details:
http://research.colfaxinternational.com/post/2014/03/11/InfiniBand-for-MIC.aspx
Process Parallelism: MPI Optimization Strategies

- Dynamic scheduling
- Load balancing
- Communication-efficient algorithms
- OpenMP/MPI hybrid
The Monte Carlo Method of Computing the Number $\pi$

\[ A_{\text{quarter circle}} = \frac{1}{4}\pi R^2 \]

\[ A_{\text{square}} = L^2. \]

\[ \langle N_{\text{quarter circle}} \rangle = \frac{A_{\text{quarter circle}}}{A_{\text{square}}} N. \]

\[ 4 \frac{\langle N_{\text{quarter circle}} \rangle}{N} = 4 \frac{\pi R^2}{4L^2} = \pi. \]

\[ \pi \approx 4 \frac{N_{\text{quarter circle}}}{N}. \]

\[ \pi = 3.141592653589793\ldots \]
The Monte Carlo Method of Computing the Number $\pi$

```c
#include <mkl_vsl.h>
const long BLOCK_SIZE=4096;

// Random number generator from MKL
VSLStreamStatePtr stream;
vslNewStream( &stream, VSL_BRNG_MT19937, seed );

for (long j = 0; j < nBlocks; j++) {
    vsRngUniform( 0, stream, BLOCK_SIZE*2, r, 0.0f, 1.0f );
    for (i = 0; i < BLOCK_SIZE; i++) {
        const float x = r[i];
        const float y = r[i+BLOCK_SIZE];
        if (x*x + y*y < 1.0f) dUnderCurve++;
    }
}

const double pi = (double)dUnderCurve / (double)iter * 4.0
```

The Monte Carlo Method of Computing the Number $\pi$

```c
int rank, nRanks, trial;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nRanks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

const double blocksPerProc = (double)nBlocks / (double)nRanks;
const long myFirstBlock = (long)(blocksPerProc*rank);
const long myLastBlock = (long)(blocksPerProc*(rank+1));

RunMonteCarlo(myFirstBlock, myLastBlock, stream, dUC);
// Compute pi
MPI_Reduce(&dUC, &UnderCurveSum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank==0)
    const double pi = (double)UnderCurveSum / (double) iter * 4.0 ;

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
```
The Monte Carlo Method of Computing the Number $\pi$

Host, coprocessor, heterogeneous

<table>
<thead>
<tr>
<th>Command</th>
<th>Time, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpirun -np 32 -host localhost ./pi_mpi</td>
<td>0.84</td>
</tr>
<tr>
<td>mpirun -np 240 -host mic0 ~/pi_mpi</td>
<td>0.44</td>
</tr>
<tr>
<td>mpirun -np 32 -host localhost ./pi_mpi : -np 240 -host mic0 ~/pi_mpi</td>
<td>0.36</td>
</tr>
</tbody>
</table>

- Coprocessor is 1.9x faster than the host system
- $T_{\text{host}} \approx 0.84$ seconds, $T_{\Phi} \approx 0.44$ seconds
- Expect $T_{\text{both}} = 1/(1/0.84 + 1/0.44) \approx 0.29$ seconds
- $T_{\text{measured}} \approx 0.36$ seconds, which is 25% worse than expected. Why?
Using Intel Trace Analyzer and Collector

CPU finishes its share of work faster than coprocessors.
Load Balancing with Static Scheduling

Solution: assign more work to CPU ranks.

\[ \alpha = \frac{b_{\text{host}}}{b_{\text{MIC}}}, \]

\[ B_{\text{total}} = b_{\text{host}} P_{\text{host}} + b_{\text{MIC}} P_{\text{MIC}}, \]

\[ b_{\text{host}} = \frac{B_{\text{total}}}{\alpha P_{\text{host}} + P_{\text{MIC}}}, \]

\[ b_{\text{MIC}} = \frac{\alpha B_{\text{total}}}{\alpha P_{\text{host}} + P_{\text{MIC}}}. \]
Load Balancing with Static Scheduling

Load balance: execution times

- Xeon only (32 processes)
- Xeon Phi only (240 processes)
- Xeon + Xeon Phi, $\alpha = 1.0$
- Xeon + Xeon Phi, $\alpha = 3.4$

Time, s (lower is better)

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon only (32 processes)</td>
<td>0.839</td>
</tr>
<tr>
<td>Xeon Phi only (240 processes)</td>
<td>0.449</td>
</tr>
<tr>
<td>Xeon + Xeon Phi, $\alpha = 1.0$</td>
<td>0.366</td>
</tr>
<tr>
<td>Xeon + Xeon Phi, $\alpha = 3.4$</td>
<td>0.283</td>
</tr>
</tbody>
</table>
§7. Conclusion
Course Recap
Programming Models for Xeon Phi Coprocessors

1. Native coprocessor applications
   - Compile with -mmic
   - Run with micnativeloadex or scp+ssh
   - The way to go for MPI applications without offload

2. Explicit offload
   - Functions, global variables require __attribute__((target(mic)))
   - Initiate offload, data marshalling with #pragma offload
   - Only bitwise-copyable data can be shared

3. Clusters and multiple coprocessors
   - #pragma offload target(mic:i)
   - Use threads to offload to multiple coprocessors
   - Run native MPI applications
Optimization Checklist

1. Scalar optimization
2. Vectorization
3. Scale above 100 threads
4. Arithmetically intensive or bandwidth-limited
5. Efficient cooperation between the host and the coprocessor(s)
Additional Resources: Reading, Guides, Support
Reference Guides

- Intel C++ Compiler 14.0 User and Reference Guide
- Intel VTune Amplifier XE User's Guide
- Intel Trace Analyzer and Collector Reference Guide
- Intel MPI Library for Linux* OS Reference Manual
- Intel Software Documentation Library
- MPI Routines on the ANL Web Site
- OpenMP Specifications
Intel’s Top 10 List

1. Download programming books: “Intel Xeon Phi Coprocessor High Performance Programming” by Jeffers & Reinders, and “Parallel Programming and Optimization with Intel Xeon Phi Coprocessors” by Colfax.

2. Watch the parallel programming webinar

3. Bookmark and browse the mic-developer website

4. Bookmark and browse the two developer support forums: “Intel MIC Architecture” and “Threading on Intel Parallel Architectures”.

5. Consult the “Quick Start” guide to prepare your system for first use, learn about tools, and get C/C++ and Fortran-based programs up and running

Link to TOP10 List for Starter Kit Developers
Intel’s Top 10 List (continued)

6. Try your hand at the **beginning lab exercises**
7. Try your hand at the **beginner/intermediate real world app exercises**
8. Browse the **case studies webpage** to view examples from many segments
9. Begin optimizing your application(s); consult your programming books, the ISA reference manual, and the support forums for assistance.
10. Hone your skills by watching more **advanced video workshops**

Link to **TOP10 List for Starter Kit Developers**
Intel Xeon Phi Starter Kit

Intel® Xeon Phi™ Coprocessor – Developer Starter Kits

- Breakthrough performance for highly-parallel applications
- A single programming model for all your code
- Experience the technology inside the world’s fastest supercomputer

OVERVIEW STARTER KITS APPLICATIONS WHERE TO BUY FAQ

<table>
<thead>
<tr>
<th>Kit</th>
<th>Workstation Kit</th>
<th>Server Kit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Intel® Xeon Phi™ 3120A Coprocessor</td>
<td>Intel® Xeon Phi™ 5110P Coprocessor</td>
</tr>
<tr>
<td></td>
<td>• Actively-cooled thermal solution (fan)</td>
<td>• Passively-cooled thermal solution</td>
</tr>
<tr>
<td></td>
<td>• Over 1 Teraflop Peak Double Precision²</td>
<td>• Over 1 Teraflop Peak Double Precision²</td>
</tr>
<tr>
<td></td>
<td>• 8 GB DDR3 memory capacity</td>
<td>• 8 GB DDR3 memory capacity</td>
</tr>
<tr>
<td></td>
<td>• 250 GB/s memory bandwidth</td>
<td>• 320 GB/s memory bandwidth</td>
</tr>
<tr>
<td></td>
<td>• 300 Watts</td>
<td>• 205 Watts</td>
</tr>
<tr>
<td></td>
<td>• Full specifications</td>
<td>• Full specifications</td>
</tr>
</tbody>
</table>

To purchase a Starter Kit, click on a partner logo in the Where To Buy tab to visit their Intel® Xeon Phi™ Coprocessor starter kit webpages.
# Intel Xeon Phi Starter Kit

Intel Xeon Phi™ Coprocessor – Developer Starter Kits

Developers: Get your Intel® Xeon Phi™ coprocessor starter kit to access all the tools you need to go parallel – act now on this limited time offer!

- Breakthrough performance for highly-parallel applications
- A single programming model for all your code
- Experience the technology inside the world’s fastest supercomputer!

## OVERVIEW

<table>
<thead>
<tr>
<th>Kit</th>
<th>Workstation Kit</th>
<th>Server Kit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Intel® Xeon Phi™ 3120A Coprocessor</td>
<td>Intel® Xeon Phi™ 5110P Coprocessor</td>
</tr>
<tr>
<td>Software</td>
<td>Intel® Parallel Studio XE 2013</td>
<td>Intel® Cluster Studio XE 2013</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Single User commercial license including 1 year Intel® Premier Support)</td>
<td>Deliver top application performance with C, C++ and Fortran compilers, libraries and analysis tools</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Industry-leading application performance that scales as processor core count and vector width increase</td>
<td>High Performance Comprehensive Cluster Development Tools for HPC</td>
</tr>
<tr>
<td></td>
<td>Efficiently scale on tomorrow’s hardware while preserving investment in existing code</td>
<td>Scale Development Efforts with Standards Driven Compilers, Programming Models and Tools</td>
</tr>
<tr>
<td></td>
<td>More details</td>
<td>Supports the Latest Multicore and Manycore Based Systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>More details</td>
</tr>
</tbody>
</table>

To purchase a Starter Kit, click on the partner logo in the Where To Buy tab to visit their Intel® Xeon Phi™ Coprocessor starter kit webpages.
# Intel Xeon Phi Starter Kit

## Intel® Xeon Phi™ Coprocessor – Developer Starter Kits

Developers: Get your Intel® Xeon Phi™ coprocessor starter kit to access all the tools you need to go parallel – act now on this limited time offer!

- Breakthrough performance for highly-parallel applications
- A single programming model for all your code
- Experience the technology inside the world’s fastest supercomputer!

## Overview

<table>
<thead>
<tr>
<th>Kit</th>
<th>Workstation Kit</th>
<th>Server Kit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Intel® Xeon Phi™ 3120A Coprocessor</td>
<td>Intel® Xeon Phi™ 5110P Coprocessor</td>
</tr>
<tr>
<td>Software</td>
<td>Intel® Parallel Studio XE 2013</td>
<td>Intel® Cluster Studio XE 2013</td>
</tr>
<tr>
<td>Programming Books</td>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
<tr>
<td>(one digital copy per book)</td>
<td>Intel® Xeon Phi™ Coprocessor High Performance Programming</td>
<td>Intel® Xeon Phi™ Coprocessor High Performance Programming</td>
</tr>
<tr>
<td></td>
<td>- A practical guide to the essentials of the Intel® Xeon Phi™ coprocessor</td>
<td>- A practical guide to the essentials of the Intel® Xeon Phi™ coprocessor</td>
</tr>
<tr>
<td></td>
<td>- Presents best practices for portable, high-performance computing and a familiar and proven threaded, scalar-vector programming model</td>
<td>- Presents best practices for portable, high-performance computing and a familiar and proven threaded, scalar-vector programming model</td>
</tr>
<tr>
<td></td>
<td>- Includes simple but informative code examples that explain the unique aspects of this new highly parallel and high performance computational product</td>
<td>- Includes simple but informative code examples that explain the unique aspects of this new highly parallel and high performance computational product</td>
</tr>
<tr>
<td></td>
<td>- Covers wide vectors, many cores, many threads and high bandwidth cache/memory architecture</td>
<td>- Covers wide vectors, many cores, many threads and high bandwidth cache/memory architecture</td>
</tr>
</tbody>
</table>

## Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors book

- An example-based intensive guide for programming Intel® Xeon Phi™ coprocessors
- Introduction to task- and data-parallel programming with MPI, OpenMP, Intel® C++ and Fortran, and automatic vectorization with the Intel C++ compiler
- Extensive discussion of high performance computing (HPC) application optimization on the Intel® Xeon® and Intel® Xeon Phi™ platforms, including scalar optimizations, improvement of SIMD operations, multithreading, efficient cache utilization, and scaling across heterogeneous distributed-memory computing platforms
Intel Xeon Phi Starter Kit

**Intel® Xeon Phi™ Coprocessor – Developer Starter Kits**

Developers: Get your Intel® Xeon Phi™ coprocessor starter kit to access all the tools you need to go parallel – act now on this limited time offer!

- Breakthrough performance for highly-parallel applications
- A single programming model for all your code
- Experience the technology inside the world’s fastest supercomputer!

**Starter Kit Provider**

To purchase a Starter Kit, click on a partner logo in the Where To Buy tab to visit their Intel® Xeon Phi™ Coprocessor starter kit webpages.

<table>
<thead>
<tr>
<th>Kit</th>
<th>Workstation Kit</th>
<th>Server Kit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Intel® Xeon Phi™ 3120A Coprocessor</td>
<td>Intel® Xeon Phi™ 5110P Coprocessor</td>
</tr>
<tr>
<td>Software</td>
<td>Intel® Parallel Studio XE 2013</td>
<td>Intel® Cluster Studio XE 2013</td>
</tr>
<tr>
<td>Programming Books</td>
<td>Intel® Xeon Phi™ Coprocessor High Performance Programming Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors book</td>
<td></td>
</tr>
</tbody>
</table>

Other Resources

- Quick start guide
- Access to the TOP 10 useful developer resources to get you started
## Workstations with Intel Xeon Phi Coprocessors (Jan 2014)

<table>
<thead>
<tr>
<th>Model</th>
<th>Max Coprocessors</th>
<th>Graphics</th>
<th>CPU</th>
<th>Max Memory</th>
<th>Max Drives</th>
<th>Form Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXP9600a</td>
<td>4</td>
<td>5110P with Passive Heatsink</td>
<td>On-board</td>
<td>Dual Intel® Xeon® E5-2600 V2 Series</td>
<td>Up to 512GB</td>
<td>8 3.5'' SAS/SATA</td>
</tr>
<tr>
<td>SXP8600</td>
<td>4</td>
<td>3120A with Active Heatsink</td>
<td>On-board OR Discrete graphics card</td>
<td>Dual Intel® Xeon® E5-2600 V2 Series</td>
<td>Up to 512GB</td>
<td>8 3.5'' SAS/SATA</td>
</tr>
<tr>
<td>SXP7450</td>
<td>2</td>
<td>3120A with Active Heatsink</td>
<td>Discrete graphics card</td>
<td>Dual Intel® Xeon® E5-2600 V2 Series</td>
<td>Up to 512GB</td>
<td>8 2.5'' HDDs/SSDs</td>
</tr>
<tr>
<td>SXP2300</td>
<td>1</td>
<td>3120A with Active Heatsink</td>
<td>Discrete graphics card</td>
<td>Dual Intel® Xeon® E5-2600 V2 Series</td>
<td>Up to 256GB</td>
<td>4 3.5'' SATA</td>
</tr>
<tr>
<td>SXP1300</td>
<td>1</td>
<td>3120A with Active Heatsink</td>
<td>Discrete graphics card</td>
<td>Single Intel® Xeon® E5-2600 V2 or E5-1600 Series</td>
<td>Up to 64GB</td>
<td>4 3.5'' SATA</td>
</tr>
</tbody>
</table>

## Servers with Intel Xeon Phi Coprocessors (Jan 2014)

<table>
<thead>
<tr>
<th>Model</th>
<th>Form Factor</th>
<th>Sockets</th>
<th>CPU/SPU</th>
<th>Max. Memory</th>
<th>Max. Hdds</th>
<th>Features</th>
</tr>
</thead>
</table>
| C720-8680 | EU          | 1       | Intel Xeon E5-2680 | Max 256GB   | Max 4 Hdds | Single Intel Xeon Phi Coprocessor  
+ Intel 10 x PCIe 3.0 x8  
+ Intel Xeon Phi 5000 Series  
+ Intel Reliable Error Detection (RED)  
+ Intel Remote Management Module  
+ Intel Integrated RAID  
+ Intel 750W Platinum PS |
| C720-8680 | EU          | 2       | Intel Xeon E5-2680 V2 | Max 256GB   | Max 6 Hdds | Single Intel Xeon Phi Coprocessor  
+ Intel 20 x PCIe 3.0 x8  
+ Intel Xeon Phi 7000 Series  
+ Intel Reliable Error Detection (RED)  
+ Intel Remote Management Module  
+ Intel Integrated RAID  
+ Intel 750W Platinum PS |
| C720-8680 | EU          | 3       | Intel Xeon E5-2680 V2 | Max 256GB   | Max 6 Hdds | Single Intel Xeon Phi Coprocessor  
+ Intel 40 x PCIe 3.0 x8  
+ Intel Xeon Phi 7000 Series  
+ Intel Reliable Error Detection (RED)  
+ Intel Remote Management Module  
+ Intel Integrated RAID  
+ Intel 750W Platinum PS |
| C720-8680 | EU          | 4       | Intel Xeon E5-2680 V2 | Max 256GB   | Max 6 Hdds | Single Intel Xeon Phi Coprocessor  
+ Intel 80 x PCIe 3.0 x8  
+ Intel Xeon Phi 7000 Series  
+ Intel Reliable Error Detection (RED)  
+ Intel Remote Management Module  
+ Intel Integrated RAID  
+ Intel 750W Platinum PS |

Research and Consulting

Colfax offers consulting services for Enterprises, Research Labs, and Universities. We can help you to:

- Optimize your existing application to take advantage of all levels of hardware parallelism
- Future-proof for upcoming innovations in computing solutions.
- Accelerate your application using coprocessor technologies.
- Investigate the potential system configurations that satisfy your cost, power and performance requirements.
- Take a deep dive to develop a novel approach.

For more details, contact us at phi@colfax-intl.com to discuss what we can do together.
Remote Access Systems

Intel-supported options for Academia:

- Manycore Testing Lab through SSG ([more info](#))
- Intel Science & Technology Center (ISTC) and Intel Collaborative Research Institutes (ICRI) programs through Intel Labs ([more info](#))
- Texas Advanced Computing Center (TACC) and National Institute for Computational Sciences (NICS) both offer allocations through the NSF XSEDE program ([more info](#))

Colfax Code Treadmill:

- Seven-day, 24/7 remote access to a personal HPC server at Colfax with training materials, Intel® Xeon® processors, Intel® Xeon Phi™ coprocessors and software development tools
- More Information: [HERE](#)

Loaner Programs

Intel Demo Depot:

- Contact your local Intel sales representative for requesting an Intel® Xeon Phi™ coprocessor-based system

Colfax Loaner Program:

- 30-day access to a loaner system, complete with Colfax hardware and software programming support
- More information please send email to: phi@colfax-intl.com

Please contact your Intel BDM or local OEM representative for more remote access and system loaner options
Go parallel today with a fully-configured system starting below $5K*
Thank you for tuning in, and have a wonderful journey to the Parallel World!