Exploiting Parallelism for Intel® Xeon Processors & Intel® Xeon Phi™ Coprocessors

going for low hanging fruits using the same tools and techniques for multi & many core architectures

J.D. Patel
Jayesh.Patel@intel.com
Agenda

1. Enabling Rapidly Growing Parallelism
2. Intel Compiler Key Optimization Features
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO
   • Parallel programming models – Cilk Plus
3. Phi Hardware & Software-stack Overview
4. Phi Programming Models
   • Native mode
   • Offload mode – Synchronous & Asynchronous
5. Other Important Stuff
   • Data alignment
   • Numeric string conversion library
   • FP accuracy, reproducibility, performance
Agenda

1. Enabling Rapidly Growing Parallelism
2. Intel Compiler Key Optimization Features
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO
   • Parallel programming models
3. Phi Hardware & Software-stack Overview
4. Phi Programming Models
   • Native mode
   • Offload mode – Synchronous & Asynchronous
5. Other Important Stuff
   • Data alignment
   • Numeric string conversion library
   • FP accuracy, reproducibility, performance
Types of parallelism in Intel processors / coprocessors / platforms

- Instruction Level Parallelism (ILP)
  - Micro-architectural techniques
    - Pipelined Execution
    - Out-of/In-order execution
    - Super-scalar execution
    - Branch prediction...

- Vector Level Parallelism (VLP)
  - Using SIMD vector processing instructions for SSE, AVX, Phi
    - SIMD registers width:
      - 64-bit (MMX) ➔ 128-bit (SSE) ➔ 256-bit (AVX) for host-CPU
      - 512-bit for Phi coprocessors

- Thread-Level Parallelism (TLP)
  - Multi-core architecture w/ & w/o Hyper-Threading (HT)
  - Many-core architecture w/ “smart” RR h/w multithreading

- Node Level Parallelism (NLP) (Distributed/Cluster/Grid Computing)
Rapidly Growing Parallelism Capability
An Inflection Point

1. **Multiple-cores** w/ HT on CPU to **Many-cores** on Phi w/ “smart” RR h/w multithreading ➔ Thread level parallelism
   - Difference in CPU-core HT vs. Phi-core multithreading
   - Over 240 threads on Phi (61 cores * 4 threads/core = 244 threads)
   - **Call to action** ➔ thread-parallelize to fully utilize all cores/threads

2. **Wider vectors** per core ➔ Vector level parallelism
   - SIMD parallelism
   - CPUs w/ AVX support has vector register width of 256 bits, 32 bytes
   - Phi coprocessors has vector register width to 512 bits, 64 bytes
   - **Call to action** ➔ vectorize to fully utilize the wider vectors

• **BOTH** must be exploited to maximize performance on Phi!
• You can start optimization on CPU and then scale it to Phi
Heterogeneous Environment

• Heterogeneous parallel hardware within each node
  • One or more CPUs
  • One or more Phi coprocessors
  • Different # of cores for CPU vs. Phi
  • Different vector-size for CPU vs. Phi

• Different configurations across nodes
  • Node w/ AVX capable CPU(s) w/ Phi coprocessor(s)
  • Node w/ SSEx capable CPU(s) w/ Phi coprocessor(s)

• Heterogeneity may create load imbalance
  • Various software architectures
    – Host only programs
    – Native only programs
    – Hybrid programs where host uses Phi via compute-offloads
    – Combinations of all of the above across nodes
  • Leads to load imbalance!
  • Different ways to load-balance and exploit performance
Enabling Advancing Parallelism

• **Vision mantra:**
  span from few cores to many cores with consistent models, languages, tools, and techniques

• One software architecture ➔ common programming models
• One software tuning method ➔ common tools for optimization

• Preserving precious investment of time, effort & money!
Intel® Parallel Studio XE 2013 and Intel® Cluster Studio XE 2013

More Cores
- Multicore
  - Xeon
  - Xeon Phi
  - 50+ cores
- Many-core

Wider Vectors
- 128 Bits
- 256 Bits
- 512 Bits

Scaling Performance Efficiently
- Serial Performance
- Task & Data Parallel Performance
- Distributed Performance

Industry-leading performance from advanced compilers
- Comprehensive libraries
- Parallel programming models
- Insightful analysis tools

Serial, Threaded & Cluster Application Development Suites
Agenda

1. Enabling Rapidly Growing Parallelism

2. **Intel Compiler Key Optimization Features**
   - **Vectorization** – auto, semi-auto, and explicit
   - IPO, PGO, HLO
   - Parallel programming models

3. Phi Hardware & Software-stack Overview

4. Phi Programming Models
   - Native mode
   - Offload mode – Synchronous & Asynchronous

5. Other Important Stuff
   - Data alignment
   - Numeric string conversion library
   - FP accuracy, reproducibility, performance
Why Use Intel® Compilers? ➔ Performance

- Goal is better performance!

- Performance to be gained in a variety of ways:
  - **Scale Up** using Intel compilers & Intel performance libraries
    - Vector Level Parallelism (VLP)
      - Ever improving SIMD capabilities of each core (MMX➔SSE➔AVX, Phi)
      - Vectorization - auto, semi-auto (pragmas/keyword), or explicit (Array Notations, SIMD-pragmas, Elemental Functions)
    - Thread Level Parallelism (TLP)
      - Easy to use task-parallel models for effective usage of all cores
      - Cilk Plus, OpenMP, TBB, Auto-Parallelism
  - **Scale Out** using Intel cluster toolkit

- Intel Compilers support the latest Features
  - Older binaries/code may not extract the best possible performance
  - Stay on the cutting edge w/ latest instructions for latest micro-architectures

- Highly Optimized libraries
  - MKL - Math functions (BLAS, FFT, LAPACK, etc.)
  - IPP - (compression, video encoding, image processing, etc.)
Why Use Intel® Compilers? ➔ Ease Of Use & Compatibility

- Multiple OS Support w/ IDE Integration
  - Visual Studio* in Windows*
  - Eclipse* in Linux*
  - Xcode* in Mac OS X*

- Quick ROI
  - May just want to recompile w/ appropriate switches
  - Simple compiler-guiding changes for better ROI

- Let Intel compilers do heavy lifting for you
  - Avoid writing & maintaining different code for different processors
  - Lower TTM & TCO thanks to much better portability & maintainability

- Source and binary compatibility
  - Mix and match components/files compiled with different compilers (e.g. icc & gcc)
  - Mix and match components/files compiled with different optimization options
**VLP / SIMD / Vectorization**

**Vectorization** is the process of transforming a scalar operation acting on single data elements at a time (Single Instruction Single Data – SISD), to an operation acting on multiple data elements at once (Single Instruction Multiple Data – SIMD)

- **SSE** → 128-bit SIMD registers → 4 FP (32-bit) or 2 DP (64-bit) calculations
- **AVX** → 256-bit SIMD registers → 8 FP (32-bit) or 4 DP (64-bit) calculations
- **Phi** → 512-bit SIMD registers → 16 FP (32-bit) or 8 DP (64-bit) calculations

**Scalar mode**
- one instruction produces one result (SISD)

**SIMD processing**
- one instruction can produce multiple results (SIMD)
- using SSE or AVX or MIC instructions

```c
for (i=0; i<=MAX; i++)
    c[i] = a[i] + b[i];
```

```
```

```
```

Copyright © 2013, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.*
VLP / SIMD: Role of Intel Compilers & Libraries

- **Hand-tuning for different processors**
  - Inline Assembly Language support
    - Most control but much harder to learn, code, debug, maintain...
  - SIMD intrinsics
    - Access to low level details similar to assembler but same issues
  - SIMD vector classes for C++
    - Nicely fits into programming methodology of C++ but still the same issues

- **Compiler based Vectorization** (details coming up!)
  - The fastest & easiest way; recommended for most cases
    - **Auto-Vectorization**
      - No code-changes; compiler vectorizes automatically for specified processor(s)
    - **Semi-Auto-Vectorization**
      - Use pragmas to guide compiler for missed auto-vectorization opportunities
    - **Explicit Vector Programming**
      - SIMD-pragmas, Elemental functions, Array notation (Cilk Plus)
      - Go after the opportunities missed by auto/semi-auto vectorization

- **MKL & IPP** library exploits SIMD capabilities for you!
  - Flexible & best performing implementation using manually written & optimized SSE/AVX/Phi-code
Vectorization (Host)

- Vectorizer helps exploit SIMD/VLP opportunities
  - Vectorizes sequential operations using SSE and/or AVX instructions
  - No significant changes to source-code but may need to *guide the* compiler
  - Much easier to learn, debug, maintain, enhance
  - Forward looking *w.r.t.* compilers and processors!

- Optimized code for targeted processor(s)
  - Both Intel and AMD* host-CPU’s
  - Mixed processors environment supported as well

- Processor Specific Optimization
  - Targeting specific Intel Processor(s)
  - e.g. for AVX capable CPU use /QxAVX (Windows) or -xAVX (Linux)

- Auto-dispatch: Processor Optimized Optimization
  - Includes both optimized and generic (SSE2) code-paths
  - e.g. for AVX capable CPU use /QaxAVX (Windows) or -axAVX (Linux)
  - e.g. for AVX and SSE4.2 capable CPUs use/QaxAVX,SSE4.2 (Windows) or -axAVX,SSE4.2 (Linux)
Ways to Write Vectorizable Code

Auto-Vectorization

```c
for(i = 0; i < num_elem; i++){
    A[i] = B[i] + C[i];
}
```

Semi-Auto-Vectorization

```c
#pragma ivdep
for(i = 0; i < num_elem; i++){
    A[i] = B[i] + C[i];
}
```

Explicit vector programming using Intel® Cilk™ Plus

**SIMD Pragma/Directive**

```c
#pragma simd
for(i = 0 i < num_elem; i++) {
    A[i] = B[i] + C[i];
}
```

**Elemental Function**

```c
__declspec(vector)
float work(float b, float c) {
    return b + c;
}
...
for(i = 0; i < num_elem; i++) {
    A[i] = work(B[i],C[i]);
}
```

Array Notation for C/C++ (Fortran like)

```c
A[:] = B[:] + C[:];
```
Explicit Vector Programming with Cilk Plus, OpenMP 4.0 SIMD, Fortran

Input: C/C++/FORTRAN source code

- Array Notation
- Elemental Function
- SIMD pragma/directive
- Semi Automatic Using Vectorization Hints (ivdep/vector pragmas)
- Fully Automatic Analysis

Express/expose vector parallelism

Map vector parallelism to vector ISA

Vectorizer

Optimize and Code Generation

Intel® SSE  Intel® AVX  Intel® MIC
Semi*-Auto-Vectorization Example
Using `#pragma ivdep` to help auto-vectorize

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

```
$ icpc -c work.cpp -xAVX -vec-report2
work.cpp(3): (col. 6) remark: LOOP WAS VECTORIZED.
```

![Vectorized loop diagram](image)
static double N(const double& z) {
    return (1.0/sqrt(2.0*PI))*exp(-0.5*z*z);
}

double option_price_call_black_scholes(
    double S, double K, double r, double sigma, double time) {
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}

void test_option_price_call_black_scholes(
    double S[], double K[], double r, double sigma, double time[],
    double call[], int num_options) {
    #pragma ivdep
    for (int i=0; i < num_options; i++) {
        call[i] = option_price_call_black_scholes(S[i],K[i],r,sigma,time[i]);
    }
}

$ icpc -c -xAVX -vec_report3 BlackScholes.cpp
BlackScholes.cpp(22): (col. 4) remark: LOOP WAS VECTORIZED.
Semi/Auto-Vectorization – Important to know

- **Focus on hot-loops only** and make sure they vectorize
- Get advice on how to help the compiler to vectorize loops
  - vectorization reports using -vec-reportN switch
  - guide switch would generate GAP report w/ suggestions
- Guidance to compiler (pragma/switch) may help vectorize
  - Pragmas
    - #pragma ivdep
    - #pragma vector always
    - #pragma loop count (n)
  - Switches
    - -fargument-noalias or -ansi-alias
    - restrict switch with restrict keyword usage
- Many loops only vectorize with High-Level Optimizer (HLO at -O3)
  - additional loop optimizations that may help vectorize transformed loops
- IPO and PGO can help a lot for vectorization:
  - IPO to handle procedure calls in loop body
  - PGO to handle unknown trip count or control flow
Guidelines for Writing Vectorizable Code

- Prefer countable single entry and single exit “for” loops
- Write straight line code. Avoid:
  - most function calls, goto/switch-statement
  - Branches that can’t be treated as masked assignments.
- Avoid dependencies between loop iterations
  - Or at least, avoid read-after-write dependencies
- Prefer array notation to the use of pointers
  - Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
  - Try to use the loop index directly in array subscripts, instead of incrementing a separate counter for use as an array address.
- Use efficient memory accesses
  - Favor inner loops with unit stride
  - Minimize indirect addressing
  - Align your data where possible to
    - 32 byte boundaries (for AVX)
    - 64 byte boundaries (for MIC)
Intel® Cilk™ Plus Array Notation

• Example:

```
A[:] = B[:] + C[:];
```

• An extension to C/C++

• Perform operations on sections of arrays in parallel

• Well suited for code that:
  – Performs per-element operations on arrays,
  – Without an implied order between them
  – With an intent to execute in vector instructions
Intel® Cilk™ Plus Array Notation Syntax

- Use a “:” in array subscripts to operate on multiple elements
- Array notation returns a subset of the referenced array
- “length” specifies number of elements of subset
- “stride”: distance between elements for subset
- “length” and “stride” are optional (all & stride 1 are default)

\[
A[:]) // all of array A \\
A[start\_index : length] \\
A[start\_index : length : stride]
\]

Explicit Data Parallelism Based on C/C++ Arrays
Elemental Functions

- Allows use of scalar syntax to describe an operation on a single element (or single set of elements)

- The programmer:
  - Writes a standard function with a scalar syntax and uses `__declspec (vector)` or `__attribute ((vector))`

- The compiler:
  - Generates a scalar and a short vector version(s).
  - Can call the vector function from vectorized loop

- The invocation
  - Deploy the function across a collection of elements, e.g. arrays
  - Each invocation will produce a vector of results instead of a single result
Elemental Functions

• Write a function for one element and add `__declspec(vector)`:

```cpp
__declspec(vector) float work(float a, float b, float c)
{
    return a * b + c;
}
```

• Call the scalar version:

```
d = work(a, b, c);
```

• Call elemental function version via SIMD loop:

```cpp
#pragma simd
for(i = 0; i < num_elem; i++) {
    D[i] = work(A[i], B[i], C[i]);
}
```

• Call it with array notations:

```
D[:] = work(A[:], B[:], C[:]);
```
# Invoking Elemental Functions

<table>
<thead>
<tr>
<th>Construct</th>
<th>Example</th>
<th>Semantics</th>
</tr>
</thead>
</table>
| Standard for loop       | `for (j = 0; j < N; j++) {`  
                          | `    a[j] = my_ef(b[j]);`  
                          | `}`                                      | Single thread,  
                                      | auto vectorization                                        |
| #pragma simd            | `#pragma simd`  
                          | `for (j = 0; j < N; j++) {`  
                          | `    a[j] = my_ef(b[j]);`  
                          | `}`                                      | Single thread,  
                                      | vectorized, use the  
                                      | vector version if matched                      |
| cilk for loop           | `cilk_for (j = 0; j < N; j++) {`  
                          | `    a[j] = my_ef(b[j]);`  
                          | `}`                                      | Both vectorization and  
                                      | concurrent execution                                |
| Array notation          | `a[:] = my_ef(b[:]);`                                                   | Vectorization                                  |
Restrictions using Elemental Functions

- The following language constructs are disallowed within elemental functions:
  - The GOTO statement
  - The switch statement with 16 or more case statements
  - Operations on classes and structs (other than member selection)
  - Expressions with array notations
  - No functions calls unless inlined or vector functions
    - Most math library functions are vector functions
  - No parallel constructs
    - The _Cilk_spawn keyword, array notation, OpenMP, native threads
#pragma simd

- Provides ability to describe vectorizable loops in a similar way to describing parallelizable loops in OpenMP

```c
void add_fl(float *a, float *b, float *c, float *d, float *e, int n) {
    #pragma simd
    for (int i=0; i<n; i++)
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Without SIMD directive, vectorization will fail since there are too many pointer references to do a run-time check for overlapping.
Auto-Vectorization – Limited by Serial Semantics

Compiler checks for:

- Is *p loop invariant?
- Are A, B and C loop invariant?
- Is A[] aliased with B[], C[], and/or sum?
- Is sum aliased with B[] and/or C[]?
- Is + operator associative? (Does the order matter?)
- Vector computation on the target expected to be faster than scalar code? (efficiency heuristic)

```c
float add( float* A, float* B, float* C, int* p) {
    float sum = 0.0f;
    for(int i = 0; i < *p; i++) {
        A[i] = B[i] * C[i];
        sum = sum + A[i];
    }
}
```

Auto vectorization is limited by the language rules: you can’t say what you mean!
Explicit Vector Programming with 
#pragma pragma simd

Programmer asserts:

- 

*\textit{p} is loop invariant
- \(A[\cdot]\) not aliased with \(B[\cdot], C[\cdot], \) & \textit{sum}
- \textit{sum} not aliased with \(B[\cdot]\) and \(C[\cdot]\)
- + operator is associative
  (compiler can reorder for better vectorization)
- Vectorized code generated even if efficiency heuristic does not indicate a gain

```
float add( float* A, float* B, float* C, int* p) {
    float sum = 0.0f;
    #pragma simd reduction(+:sum)
    for(int i = 0; i < *p; i++) {
        A[i] = B[i] * C[i];
        sum = sum + A[i];
    }
}
```

Explicit vector programming lets you express what you mean!

```
icpc -c -xAVX -vec-report3 add-simd.cpp
add-simd.cpp(4): (col. 4) remark: SIMD LOOP WAS VECTORIZED
```
Solution: If, for example, offsets are at least 4 elements, vectorization is still possible as vector length can be controlled via \texttt{#pragma simd}:

\begin{verbatim}
#pragma simd
private(X,Y,seed)
reduction(+:darts_in,darts_out)
for(int i=0; i<darts; i++){
    X = erand48(seed);
    Y = erand48(seed);
    if ((X*X + Y*Y) <= 1.0) {
        darts_in++;
    } else {
        darts_out++;
    }
}
\end{verbatim}

This program results in good utilization of vector level parallelism and provides measureable speedups.
#pragma simd Example: Mandelbrot

// vectorizable outer loop
#pragma simd
for (i=0; i<n; i++) {
    complex<float> c = a[i];
    complex<float> z = c;
    int j = 0;
    while ((j < 255)
        && (abs(z)< limit)) {
        z = z*z + c;
        j++;
    };
    color[i] = j;
}

This program results in good utilization of vector level parallelism and provides measureable speedups.
// This sample is derived from code published by Bernt Arne Odegaard
// http://finance.bi.no/~bernt/gcc_prog/recipes/recipes/
__declspec(vector)
static double N(const double& z) {
    return (1.0/sqrt(2.0*PI))*exp(-0.5*z*z);
}
__declspec(vector(uniform(r,sigma)))
double option_price_call_black_scholes(
    double S, double K, double r, double sigma, double time) {
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}
void test_option_price_call_black_scholes(
    double S[], double K[], double r, double sigma, double time[],
    double call[], int num_options) {
    call[0:num_options] = option_price_call_black_scholes(
        S[0:num_options],K[0:num_options],r,sigma,time[0:num_options]);
}

$ icpc -c -xAVX -vec_report3 BlackScholes.cpp
BlackScholes.cpp(22): (col. 1) remark: FUNCTION WAS VECTORIZED.
BlackScholes.cpp(48): (col. 4) remark: LOOP WAS VECTORIZED.
Agenda

1. Enabling Rapidly Growing Parallelism

2. Intel Compiler Key Optimization Features
   - Vectorization – auto, semi-auto, and explicit
   - IPO, PGO, HLO
   - Parallel programming models

3. Phi Hardware & Software-stack Overview

4. Phi Programming Models
   - Native mode
   - Offload mode – Synchronous & Asynchronous

5. Other Important Stuff
   - Data alignment
   - Numeric string conversion library
   - FP accuracy, reproducibility, performance
InterProcedural Optimization (IPO)

- Optimization at build-time by performing static analysis of code
- Cross-module optimization

Benefits of IPO
- Optimization of large number of frequently used small & medium functions, especially those called in loops
- Function Inlining
  - Eliminates need for arguments setup, call branch/return overhead
  - Enables opportunities for other optimizations (const propagation, DCE, etc.)
- Dead code elimination, Better register usage
- Improved alias analysis for better auto-vectorization & loop transformations

Better to use IPO w/ PGO to guide function inlining

May increase build-time/binary size
IPO (contd.)

- Two step build process:
  1. Compilation phase - creates info file containing Intermediate Representation (IR) of source & summary for optimization
  2. Linking Phase - performs IPO on all files with IR info

- Single and multi-files IPO possible
  icpc -ip \(\rightarrow\) single file IPO
  Inlining functions defined within same file
  icpc -ipo \(\rightarrow\) multi-file IPO
  Inlining functions defined across multiple

Usability Tips:
- Try IPO on performance critical files/libs
- Don’t run ipo on 10,000’s object files, avoid unnecessary increased build time
- Remember to link with -ipo option
Profile Guided Optimization (PGO)

- Optimization with runtime feedback
- Static analysis leaves many questions open for the optimizer like:
  
  ```
  if (x > y)
    do_this();
  else
    do_that();
  
  How often is x > y? What is the size of count?
  Which code is touched how often?
  ```

- Use execution-time feedback to guide final optimization
PGO (contd.)

- PGO is 3-step process:
  1. Instrumented Compilation phase (using “prof-gen”)
  2. Instrumented Execution Phase
     1. Instrumented program is run with *typical* workloads
     2. Dynamic runtime info gathered separately for each run
  3. Feedback Compilation phase (using “prof-use”)
     1. Dynamic info merged into a profile that guides compiler optimizations

- Benefits of PGO
  - Better data & code layout
    - Frequently accessed code placed adjacent
    - Better instruction cache usage & fetching
  - Improved branch prediction – good for branchy apps
  - Switch-statement optimization
  - Better function inlining (inline hot functions, not cold)
  - Can optimize function ordering
  - Better vectorization decisions
PGO: Three Step Process

**Step 1**
Compile + link to add instrumentation
```
icc -prof_gen prog.c
```

**Step 2**
Execute instrumented program
```
prog.exe (on a typical dataset)
```

**Step 3**
Compile + link using feedback
```
icc -prof_use prog.c
```

- **Instrumented executable:** foo.exe
- **Dynamic profile:** 12345678.dyn
- **Merged .dyn files:** pgopti.dpi
- **Optimized executable:** foo.exe
Auto-Parallelization

- Serial portion of code automatically translated into multi-threaded code when possible
- Frees developers from having to:
  - Determine good work-sharing portion of serial code
  - Perform dataflow analysis to verify correct parallel execution
  - Partition data for threaded code
- Parallel runtime support offers same features as in OpenMP
  - Handling details of loop iteration modification
  - Thread scheduling
  - Synchronization
- Enabled by “-parallel” switch
  - /Qparallel on Windows
- “#pragma parallel” if you know it’s safe to parallelize a loop
  - Ok to ignore potential aliasing of pointers or array references
High-Level Optimizations (HLO)

- Enabled with -O3 (/O3 on Windows)
  - With auto-vectorization does more aggressive data dependency analysis than at /O2
  - Exploits properties of source code (loops & arrays)
  - Best chance for performing loop transformations

- Performs loop transformations:
  - Loop distribution
  - Loop interchange
  - Loop fusion
  - Loop unrolling
  - Data pre-fetching
  - PGO based loop unrolling
  - etc.
Agenda

1. Enabling Rapidly Growing Parallelism

2. Intel Compiler Key Optimization Features
   - Vectorization – auto, semi-auto, and explicit
   - IPO, PGO, HLO
   - Parallel programming models – Cilk Plus

3. Phi Hardware & Software-stack Overview

4. Phi Programming Models
   - Native mode
   - Offload mode – Synchronous & Asynchronous

5. Other Important Stuff
   - Data alignment
   - Numeric string conversion library
   - FP accuracy, reproducibility, performance
A Family of Parallel Programming Models
Developer Choice

Choice of high-performance parallel programming models

- Libraries for pre-optimized and parallelized functionality
- Intel® Cilk™ Plus and Intel® Threading Building Blocks supports composable parallelization of a wide variety of applications.
- OpenCL* addresses the needs of customers in specific segments, and provides developers an additional choice to maximize their app performance
- MPI supports distributed computation, combines with other models on nodes
Parallel Models: Few Recommendations

- **Cilk Plus**
  - `#pragma simd` & Array Notation for vector-level parallelism
    - Easily apply to new & existing apps for predictable vectorization
    - Interoperate with other threading models - TBB, Cilk, native-threads...
  - `cilk_for` & `cilk_spawn/cilk_sync` for task-level parallelism
    - Simplest & most debuggable parallel code
  - **Reducers**
    - Provide lock-free mechanism that allows parallel code to use private "views" of a variable which are merged at next sync
    - Merge is done in ordered manner to maintain serial semantics

- **OpenMP 4.0 for task & vector-level (simd) parallelism**

- **TBB for task-level parallelism**
  - Portability to non-Intel compilers and CPUs
Intel® Cilk™ Plus

Cilk Plus is made up of Five main features:

1. Set of **Keywords** for expressing task level parallelism
2. **Array Notations** for expressing vector/data level parallelism
3. **Reducers** to resolve data races for shared variables
4. **Elemental functions** for expressing vector/data level parallelism for scalar user-defined functions to vectorize and then apply to array sections
5. **simd pragma** to enable and enforce vectorization of loops
Intel® Cilk™ Plus keywords

• Cilk Plus adds three keywords to C and C++:
  _cilk_spawn
  _cilk_sync
  _cilk_for

• If you #include <cilk/cilk.h>, you can write the keywords as cilk_spawn, cilk_sync, and cilk_for.

• Cilk Plus runtime controls thread creation and scheduling. A thread pool is created prior to use of Cilk Plus keywords.

• The number of threads matches the number of cores by default, but can be controlled by the user.
**cilk_spawn & cilk_sync example**

- Recursive computation of a Fibonacci number:
  ```c
  int fib(int n) {
    int x, y;

    if (n < 2) return n;

    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x+y;
  }
  ```

Execution can continue while fib(n-1) is running.

Asynchronous call must complete before using x.
cilk_for loop

• Looks like a normal for loop:
  
  ```cilk_for (int x = 0; x < 1000000; ++x) { … }```
  
  ```cilk_for (vector<int>::iterator x = y.begin();
              x != y.end(); ++x) { … }```

• Any or all iterations may execute in parallel with one another.

• All iterations complete before program continues.

• Constraints:
  
  – Limited to a single control variable.
  – Must be able to jump to the start of any iteration at random.
  – Iterations should be independent of one another.

• Not allowed:
  
  ```cilk_for (list<int>::iterator x = y.begin();
              x != y.end(); ++x) { … }```

  – Loop count cannot be computed in constant time for a list. (y.end() – y.begin() is not defined.)
  – Do not have random access to the elements of the list. (y.begin() + n is not defined.)
Black Scholes w/ Cilk Plus:
Elemental Function & cilk_for

// This sample is derived from code published by Bernt Arne Odegaard http://finance.bi.no/~bernt/gcc_prog/recipes/recipes/

__declspec(vector)
static double N(const double& z) {
  return (1.0/sqrt(2.0*PI))*exp(-0.5*z*z);
}

__declspec(vector(uniform(r,sigma)))
double option_price_call_black_scholes(
    double S, double K, double r, double sigma, double time) {
  double time_sqrt = sqrt(time);
  double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
  double d2 = d1-(sigma*time_sqrt);
  return S*N(d1) - K*exp(-r*time)*N(d2);
}

void test_option_price_call_black_scholes(
    double S[], double K[], double r, double sigma, double time[],
    double call[], int num_options) {
  cilk_for (int i=0; i < num_options; i++) {
    call[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
  }
}
#pragma simd

Pi Using Monte Carlo example

```c
#pragma simd private(X,Y,seed) \n    reduction(+:darts_in,darts_out)

for(int i=0; i<darts; i++)
{
    X = erand48(seed);
    Y = erand48(seed);
    if ((X*X + Y*Y) <= 1.0) {
        darts_in++;
    } else {
        darts_out++;
    }
}
```

#pragma simd exploits SIMD capability
```c
#pragma omp parallel for private(X,Y,seed) \
    reduction(+:darts_in,darts_out)
#pragma simd private(X,Y,seed) \
    reduction(+:darts_in,darts_out)
  for(int i=0; i<darts; i++)
  {
    X = erand48(seed);
    Y = erand48(seed);
    if ((X*X + Y*Y) <= 1.0) {
      darts_in++;
    } else {
      darts_out++;
    }
  }
```

`#pragma simd` exploits SIMD capability, and
`#pragma omp` exploits multiple cores/threads
#pragma simd with OpenMP 4.0: Pi Using Monte Carlo example

```c
#pragma omp parallel for simd private(X,Y,seed) \
    reduction(+:darts_in,darts_out)
for(int i=0; i<darts; i++)
{
    X = erand48(seed);
    Y = erand48(seed);
    if ((X*X + Y*Y) <= 1.0) {
        darts_in++;
    } else {
        darts_out++;
    }
}
```

Starting w/ OpenMP 4.0: simd clause added to #pragma omp so now it exploits both SIMD capability and multiple cores/threads
Host mode – Black Scholes example app

1. **Host Mode**
   - i. Single-Thread no-vec
   - ii. Single-Thread Vectorized
   - iii. Multi-Thread Vectorized

2. **Native Mode**
   - i. Single-Thread no-vec
   - ii. Single-Thread Vectorized
   - iii. Multi-Thread Vectorized

3. **Offload Mode**
   - i. Synchronous Offload
   - ii. Asynchronous Offload
Host mode – Black Scholes example app

• App is built for host processors that are AVX/SSE4.2 capable
• Objective here is to demonstrate the performance gain achieved by exploiting both wider vectors (SIMD capability) and many-cores (threads) of Phi coprocessor
• App is built for following 3 different scenarios:
  1. Single-Thread no-vec
  2. Single-Thread Vectorized
  3. Multi-Thread Vectorized
Host mode (contd)

- App is built for host processors that are AVX/SSE4.2 capable
- Vectorization is disabled to create performance baseline
- Non-vectorized code is then run on single & multiple-threads
- Similarly, vectorized code is generated and run on single and multiple-threads

- Building the app **without** vectorization
  
  ```
  $ icpc -no-vec -no-simd -vec-report3 BlackScholes.cpp -o bs-no_vec
  $
  ```

- Building the app **with** vectorization
  
  ```
  $ icpc -xSSE4.2 -vec-report3 BlackScholes.cpp -o bs-SSE4.2
  BlackScholes.cpp(95): (col. 22) remark: loop was not vectorized: statement cannot be vectorized.
  BlackScholes.cpp(110): (col. 19) remark: LOOP WAS VECTORIZED.
  BlackScholes.cpp(105): (col. 4) remark: loop was not vectorized: not inner loop.
  BlackScholes.cpp(55): (col. 45) remark: LOOP WAS VECTORIZED.
  $
  ```
Host mode (contd)

Running the app with single-thread – looking at vectorization gain
DP (64-bit) calculations on SSE4.2 (128-bit) arch ➔ 2x gain expected

$ export CILK_NWORKERS=1
$

$ time ./bs-no_vec
num_options = 1048576, num_iterations = 256, chunk_size = 65536
...
real 0m36.305s
user 0m36.290s
sys 0m0.005s
$

$ time ./bs-SSE4.2
num_options = 1048576, num_iterations = 256, chunk_size = 65536
...
real 0m17.713s
user 0m17.701s
sys 0m0.006s
$
Host mode (contd)

Running the app with 4-threads (on quad-core CPU) - looking at multi-thread gain

$ export CILK_NWORKERS=4
$ 
$ time ./bs-no_vec
num_options = 1048576, num_iterations = 256, chunk_size = 65536
...
real 0m9.520s
user 0m36.356s
sys 0m0.256s
$
$ 
$ time ./bs-SSE4.2
num_options = 1048576, num_iterations = 256, chunk_size = 65536
...
real 0m4.813s
user 0m17.787s
sys 0m0.161s
$
Agenda

1. Enabling Rapidly Growing Parallelism
2. Intel Compiler Key Optimization Features
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO
   • Parallel programming models
3. Phi Hardware & Software-stack Overview
4. Phi Programming Models
   • Native mode
   • Offload mode – Synchronous & Asynchronous
5. Other Important Stuff
   • Data alignment
   • Numeric string conversion library
   • FP accuracy, reproducibility, performance
Intel® Xeon Phi™ Coprocessor core

Fully functional multi-thread execution unit

- >60 in-order cores interconnected with high-speed bidirectional ring
- 64-bit addressing
- Scalar unit based on Intel® Pentium® processor family
- Two pipelines
- Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
- 4 clock latency from issue to resolution
- 4 hardware threads per core
  - Each thread issues instructions in turn
  - Each thread can issue instructions every other clock, so at least 2 threads must be run per core!
  - Highly tuned kernels may saturate a core with just 2 threads, otherwise 3 to 4 threads/core may be necessary
  - Round-robin execution helps mask latencies on individual thread’s in-order execution
  - Not same as Hyper threading (HT) on CPU’s
Intel® Xeon Phi™ Coprocessor core

Fully functional multi-thread execution unit

Optimized for single and double precision

All new vector unit
- 512-bit SIMD Instructions – not Intel® SSE, MMX™, or Intel® AVX
- 32 512-bit wide vector registers
  - Hold 16 singles or 8 doubles per register

Cache organization
- L1 cache
  - L1-D 32KB
  - L1-I 32KB
- L2 cache
  - 512KB per core
  - inclusive of L1-D & L1-I
  - shared across all cores over ODI
  - If neither code nor data is shared among all cores, then L2 = 30.5MB (= 512KB/core x 61 cores)
  - If all code + data is shared among all cores, then L2 = 512KB

Other brands and names are the property of their respective owners.
Phi Card Hardware Overview

- **Highly Parallel** device!!
- **SMP on-a-chip** best describes Intel Xeon Phi Coprocessor
- >60 cores, 4 threads/core, 512-bit SIMD capable
- Individual cores are tied together via fully coherent caches into a bidirectional ring

Bidirectional ring 115 GB/sec
Distributed Tag Directory (DTD) reduces ring snoop traffic
PCIe port has its own ring stop

GDDR5 Memory
16 memory channels
- Up to 5.5 Gb/sec
8 GB 300ns access

L1 32K I & D-cache per core
3 cycle access
Up to 8 concurrent accesses

L2 512K cache per core
11 cycle best access
Up to 32 concurrent accesses
Phi Coprocessor Microarchitecture
Picture worth many words

Intel Xeon Phi coprocessor Peak

Intel Xeon processors Peak

Performance

Threads

Code the Future

2013, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.
Running as a **Native** or **MPI** compute node via IP

### Host Processor

**Advantages**
- Simpler model
- No directives
- Easier port
- Good kernel test

- ssh or telnet connection to coprocessor IP address

- User-level code
- System-level code

- **Intel® Xeon Phi™ Coprocessor**
  - Architecture support libraries, tools, and drivers

- **Linux* OS**
- **PCI-E Bus**

### Intel® Xeon Phi™ Coprocessor

**Target-side “native” application**
- User code
  - Standard OS libraries plus any 3rd-party or Intel libraries

- **Virtual terminal session**

**Use if**
- Highly parallel
- No serial
- Modest memory
- Complex code
- No hot spots

- **Intel® Xeon Phi™ Coprocessor**
  - Communication and application-launch support

- **Linux* OS**
- **PCI-E Bus**
Running as an accelerator for Offloaded host computation

**Advantages**
- More memory available
- Better file access
- Host better on serial code
- Better uses resources

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

**Intel® Xeon Phi™ Coprocessor**
- Target-side offload application
  - User code
  - Offload libraries, user-accessible APIs and libraries

- Intel® Xeon Phi™ Coprocessor support libraries, tools, and drivers
- L**inux* OS**
- PCI-E Bus

- Intel® Xeon Phi™ Coprocessor communication and application-launch support
- Linux* OS
- PCI-E Bus
Agenda

1. Enabling Rapidly Growing Parallelism
2. Intel Compiler Key Optimization Features
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO
   • Parallel programming models
3. Phi Hardware & Software-stack Overview
4. **Phi Programming Models**
   • Native mode
   • Offload mode – Synchronous & Asynchronous
5. Other Important Stuff
   • Data alignment
   • Numeric string conversion library
   • FP accuracy, reproducibility, performance
Spectrum of Programming Models and Mindsets

Multi-Core Centric

Many-Core Centric

Multi-Core Hosted
General purpose serial and parallel computing

Symmetric
Codes with balanced needs

Many Core Hosted
Highly-parallel codes

Offload
Codes with highly-parallel phases

Main( )
Foo( )
MPI_*()
IA Benefit: Wide Range of Development Options

*Breadth*

**Multi-Core Centric**

- Offload: Code with highly-parallel phases

**Many-Core Centric**

- Symmetric: Codes with balanced needs

---

**Threading Options**

- Intel® Math Kernel Library
- Intel® Threading Building Blocks
- Intel® Cilk™ Plus
- OpenMP®
- Pthreads*

---

**Vector Options**

- Intel® Math Kernel Library
- Array Notation: Intel® Cilk™ Plus
- Auto vectorization
- Semi-auto vectorization: #pragma (vector, ivdep, simd)
- OpenCL*
- C/C++ Vector Classes (F32vec16, F64vec8)

---

*Fine control*

*Ease of use*

Breadth, depth, familiar models meet varied application needs
Phi Programming Models

• An Intel® Xeon Phi™ coprocessor is accessed via the host system, but may be programmed either as a coprocessor(s) or as an autonomous processor.
• The appropriate model may depend on application and context.  
• Data parallelism, use of parallel algorithms and application scalability are criteria for targeting Intel® MIC Architecture, but not for distinguishing between native or offload modes.

**Offload (Coprocessor)**
- Pragma/directives based
- Better serial processing
- More memory
- Better file access
- Makes fuller use of available resources

**Native (Autonomous)**
- Simpler programming model
  - Easier or no code porting
- Maybe quicker route for initial testing of key kernels
- Some constraints
  - Memory availability
  - File I/O access
Agenda

1. Enabling Rapidly Growing Parallelism
2. Intel Compiler Key Optimization Features
   - Vectorization – auto, semi-auto, and explicit
   - IPO, PGO, HLO
   - Parallel programming models
3. Phi Hardware & Software-stack Overview
4. Phi Programming Models
   - Native mode
   - Offload mode – Synchronous & Asynchronous
5. Other Important Stuff
   - Data alignment
   - Numeric string conversion library
   - FP accuracy, reproducibility, performance
Native Execution Model

• Appropriate if application
  • Contains very little serial processing
  • Has a modest memory footprint
  • Has a very complex code structure and/or does not have well-identified hot kernels than can be offloaded without substantial data transfer overhead
  • Does not perform extensive I/O

• Simplest programming model for Phi
  • Simple or no code porting required!
  • Simple to build
  • Simple to run
  • Simple to tune
Building Native Applications

- **Cross** compiler only, (same one as used for offload)
  - Set environment in the usual way
    
    ```bash
    source /opt/intel/compilerxe/bin/compilervars.sh intel64
    ```
  - Build on host with **–mmic**
    - This sets the `__MIC__` macro
  - Remotely, create a directory on the targeted coprocessor, e.g.
    ```bash
    ssh mic0 ‘mkdir /tmp/mydir’ (or mic1:, mic2:, etc)
    ```
  - Copy executable, any dependencies and small data files onto coprocessor using `scp`, e.g.:
    ```bash
    scp ./a.out mic0:/tmp/mydir/.
    scp /opt/intel/compiler_xe_2013/lib/mic/libiomp5.so mic0:/tmp/mydir/.
    ```
    - or copy to `/lib64` if no other users...
  - Files are not permanent (in RAM) – recopy after reboot
Building Native Libraries

• Shared Libraries
• Use the standard method for creating shared objects and also include –mmic
  • $icc -mmic -c -fpic mylib.c  // Creates mylib.o by default
  • $icc -mmic -shared -o libmylib.so mylib.o  // Creates the shared object
  • $icc -mmic main.c libmylib.so  // Link the application

• Static Libraries
• Use xiar to create native static libraries
  • $icc -mmic -c -fpic mylib.c  // Creates mylib.o by default
  • $xiar crs libmylib.a mylib.o  // Creates the static library
  • $icc -mmic main.c libmylib.a  // Link the application
Native mode – Black Scholes example app

1. Host Mode
   i. Single-Thread no-vec
   ii. Single-Thread Vectorized
   iii. Multi-Thread Vectorized

2. Native Mode
   i. Single-Thread no-vec
   ii. Single-Thread Vectorized
   iii. Multi-Thread Vectorized

3. Offload Mode
   i. Synchronous Offload
   ii. Asynchronous Offload
Native mode (contd.)

- App is built for Phi coprocessors
- Vectorization is disabled to create performance baseline
- Non-vectorized code is then run on single & multiple-threads
- Similarly, vectorized code is generated and run on single and multiple-threads

- Building the app without vectorization
  
  $ \text{icpc -mmic -no-vec -no-simd -vec-report3 BlackScholes.cpp -o bs-mic-no_vec}$

  $
  
  \$

- Building the app with vectorization
  
  $ \text{icpc -mmic -vec-report3 BlackScholes.cpp -o bs-mic-vec}$

  BlackScholes.cpp(95): (col. 22) remark: loop was not vectorized: statement cannot be vectorized.
  BlackScholes.cpp(110): (col. 19) remark: LOOP WAS VECTORIZED.
  BlackScholes.cpp(110): (col. 19) remark: REMAINDER LOOP WAS VECTORIZED.
  BlackScholes.cpp(105): (col. 4) remark: loop was not vectorized: not inner loop.
  BlackScholes.cpp(55): (col. 45) remark: LOOP WAS VECTORIZED.
  BlackScholes.cpp(55): (col. 45) remark: PEEL LOOP WAS VECTORIZED.
  BlackScholes.cpp(55): (col. 45) remark: REMAINDER LOOP WAS VECTORIZED.
Native mode (contd)

Preparing to run the native app on Phi coprocessor

```bash
$ sudo ssh mic0 'mkdir /tmp/JD'
$ sudo scp bs-mic-* mic0:/tmp/JD/.
bs-mic-no_vec 100% 84KB 84.2KB/s 00:00
bs-mic-vec 100% 131KB 131.2KB/s 00:00
```

```bash
# ./bs-mic-no_vec
./bs-mic-no_vec: error while loading shared libraries: libcilkrts.so.5: cannot open shared object file: No such file or directory

$ sudo scp /.../13.1/163/composer_xe_2013.3.163/compiler/lib/mic/libcilkrts.so.5 mic0:/tmp/JD/.
libcilkrts.so.5 100% 269KB 269.4KB/s 00:00
```

```bash
# ./bs-mic-no_vec
./bs-mic-no_vec: error while loading shared libraries: libcilkrts.so.5: cannot open shared object file: No such file or directory
# export LD_LIBRARY_PATH=/tmp/JD
```
Native mode (contd)

Running the app with single-thread – looking at vectorization gain
DP (64-bit) calculations on Phi (512-bit) arch ➔ 8x gain expected

# export CILK_NWORKERS=1
#
# time ./bs-mic-no_vec
num_options = 1048576, num_iterations = 256, chunk_size = 65536
...
real 5m 30.21s
user 5m 28.23s
sys 0m 0.39s
#
# time ./bs-mic-vec
num_options = 1048576, num_iterations = 256, chunk_size = 65536
...
real 0m 53.86s
user 0m 53.42s
sys 0m 0.14s
#
Native mode (contd)

Running the app with multiple-threads

```bash
# export CILK_NWORKERS=120
#
# time ./bs-mic-no_vec
num_options = 1048576, num_iterations = 256, chunk_size = 65536
...
real  0m 10.86s
user   8m 0.76s
sys    0m 3.84s
#
# time ./bs-mic-vec
num_options = 1048576, num_iterations = 256, chunk_size = 65536
...
real  0m 3.10s
user   1m 52.69s
sys    0m 3.44s
```
1. Enabling Rapidly Growing Parallelism
2. Intel Compiler Key Optimization Features
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO
   • Parallel programming models
3. Phi Hardware & Software-stack Overview
4. **Phi Programming Models**
   • Native mode
   • **Offload mode – Synchronous** & Asynchronous
5. Other Important Stuff
   • Data alignment
   • Numeric string conversion library
   • FP accuracy, reproducibility, performance
Offload Execution Model

• Appropriate if application
  • Cannot be made highly parallel throughout its execution
  • Is large/complex and requires much more memory
  • Performs a lot of I/O
  • Needs frequent access to special device(s)
  • Has well-identified fewer hotspots (compute-kernels type)
• Not-so-simple programming model compared to native-mode
• Requires two-level of memory-blocking to manage within available limited coprocessor memory
  1. Fit the input/output data
  2. Fit the offload code
**Offload Programming Model**

- *Explicit* programming model
- App-developer identifies compute-intensive app-sections and uses pragmas/directives to offload it to run on target coprocessor
- Very different than say MKL’s automatic-offload model
- Good fit when offloaded-code is compute-intensive and can exploit both wider-vectors and cores/threads on Phi coprocessor, *without* performing a lot of I/O
- Execution begins and ends on host/processor
- If Phi coprocessor is available, offload-sections are executed on Phi coprocessor
- If Phi coprocessor is not present in the system, the app continues to run to-be-offloaded app-sections on the host
- Simple to use offload options but better performance may be extracted by controlling the data transfers to/from Phi
## Running your Hybrid Application

**Execution on the host and Intel® MIC Co-processor(s)**

<table>
<thead>
<tr>
<th>Without: Intel® MIC Co-processor(s) are absent</th>
<th>With: Intel® MIC Co-processor(s) are present</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application starts and executes on host</td>
<td>Application starts on host and executes portions on Intel MIC Co-processor(s)</td>
</tr>
<tr>
<td>At runtime, if Intel® MIC Co-processor(s) are available, the target binary is loaded</td>
<td></td>
</tr>
<tr>
<td>At each offload, the construct runs on host cores/threads</td>
<td>At each offload, the construct runs on the Intel MIC® Co-processor(s)</td>
</tr>
<tr>
<td>Normal program termination on host</td>
<td>At program termination, target binary is unloaded</td>
</tr>
</tbody>
</table>

**Intel MIC Co-processor(s)**

- Host Offload Library
- Target Offload Library
- Message Library

**Execution Flow**

Your Application

With identified Compute Intensive Kernels

```
Your Application

With identified Compute Intensive Kernels
```

**Intel® MIC Co-processor(s)**

- Multicore
- Many-core

Intel Confidential - Use under NDA only
**Offload Model Program Flow**

**Execution**
- If at first offload the target is available, the target program is loaded.
- At each offload if the target is available, statement is run on target, else it is run on the host.
- At program termination the target program is unloaded.

---

**Host**
Intel® Xeon® processor

```cpp
f() {
    #pragma offload
    a = b + g();
    h();
}
```

```cpp
__attribute__((target(mic)))
g() {
    ...
}
```

```cpp
h() {
    ...
}
```

**Target**
Intel® Xeon Phi™ coprocessor

```cpp
f_part1() {
    a = b + g();
}
```

```cpp
__attribute__((target(mic)))
g() {
    ...
}
```
Parallel programming is the same on coprocessor and host.
Go Parallel with OpenMP*
Intel® C/C++ and Fortran Compilers
(C Example)

```c
main()
{   double pi = 0.0f; long i;

#pragma omp parallel for reduction(+:pi)
for (i=0; i<N; i++)
{   double t = (double)((i+0.5)/N); pi += 4.0/(1.0+t*t);
}
printf("pi = %f
",pi/N);
}
```

OpenMP* is Applicable to Multicore and Many-core Programming

Intel® Xeon® processor

Intel® MIC co-processor

Intel Confidential - Use under NDA only

*Other brands and names are the property of their respective owners.
Two Offload Programming Models

- No shared common system memory between host and target
- Data gets transferred/copied back and forth as specified
- Two programming models to bridge these two separate spaces
  1. Non-shared memory model
  2. Virtual-shared memory model
Offload using pragmas/directives

Non-shared Memory Model

• No physical shared memory & no shared VM
• No coherence maintained between VMs of processor & coprocessors’

• Appropriate for bitwise copyable data (no pointers)
• Using offload pragmas/directives
• Data transfer/copy
  - Avoid unneeded copy/transfers of data
  - in/out/inout/nocpy
  - alloc $\rightarrow$ Allocating memory for parts of C/C++ array
  - Moving data from one variable into the another

• Data persistence
  - Avoid unneeded allocation/de-allocation of dataBuffers
  - alloc_if/free_if $\rightarrow$ ALLOC/RETAIN/REUSE/FREE
Language Extensions for Offload ( pragmas/directives )

- Offload **pragma/directive** for data marshalling
  - `#pragma offload <clauses>` in C/C++
    - Offloads the following OpenMP block or Intel® Cilk™ Plus construct or function call or compound statement
- Offloaded data must be scalars, arrays, bit-wise copyable structs (C/C++) or derived types (Fortran)
  - no embedded pointers or allocatable arrays
  - Excludes all but simplest C++ classes
  - Excludes most Fortran 2003 object-oriented constructs
  - All data types can be used within the target code
  - Data copy is explicit
Offload examples using `#pragma offload`

// Traditional “Hello World” from Phi

```c
#pragma offload target(mic)
printf(“Hello World\n”);
```

// Offloading your function

```c
__declspec(target(mic)) void do_something();

do_something(); // invoke on host processor
#pragma offload target(mic)
do_something(); // offloaded invocation
```

// All functions available for processor
// Only those declared as above available for both
Offload examples using \#pragma offload

// Global variable access

__declspec(target(mic)) int g_count = 0;

g_count++; // accessing on host processor

#pragma offload target(mic)
g_count++; // accessing in offloaded section

// All global variables accessible on host processor
// Only those declared as above accessible on both

// Local variable access - nothing special to be done!

int count = 0;

count++; // accessing on host processor

#pragma offload target(mic)
count++; // accessing in offloaded section
## Pragmas/Directives Mark Data and Code to be Offloaded and Executed on Coprocessor

<table>
<thead>
<tr>
<th>C/C++ Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Offload pragma</strong></td>
</tr>
<tr>
<td><code>#pragma offload &lt;clauses&gt; &lt;statement&gt;</code></td>
</tr>
<tr>
<td>Allow next statement to execute on coprocessor or host CPU</td>
</tr>
<tr>
<td><strong>Variable/function offload properties</strong></td>
</tr>
<tr>
<td><code>__attribute__((target(mic)))</code></td>
</tr>
<tr>
<td>Compile function for, or allocate variable on, both host CPU and coprocessor</td>
</tr>
<tr>
<td><strong>Entire blocks of data/code defs</strong></td>
</tr>
<tr>
<td><code>#pragma offload_attribute(push, target(mic))</code></td>
</tr>
<tr>
<td><code>#pragma offload_attribute(pop)</code></td>
</tr>
<tr>
<td>Mark entire files or large blocks of code to compile for both host CPU and coprocessor</td>
</tr>
</tbody>
</table>
Offload options to control data copying and managing memory alloc/free on coprocessor

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple coprocessors</td>
<td><code>target(mic[:unit] )</code></td>
<td>Select specific coprocessors</td>
</tr>
<tr>
<td>Conditional offload</td>
<td><code>if (condition) / manadatory</code></td>
<td>Select coprocessor or host compute</td>
</tr>
<tr>
<td>Inputs</td>
<td><code>in(var-list modifiers$_{opt}$)</code></td>
<td>Copy from host to coprocessor</td>
</tr>
<tr>
<td>Outputs</td>
<td><code>out(var-list modifiers$_{opt}$)</code></td>
<td>Copy from coprocessor to host</td>
</tr>
<tr>
<td>Inputs &amp; outputs</td>
<td><code>inout(var-list modifiers$_{opt}$)</code></td>
<td>Copy host to coprocessor and back when offload completes</td>
</tr>
<tr>
<td>Non-copied data</td>
<td><code>nocopy(var-list modifiers$_{opt}$)</code></td>
<td>Data is local to target</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifiers</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify copy length</td>
<td><code>length(N)</code></td>
<td>Copy N elements of pointer’s type</td>
</tr>
<tr>
<td>Coprocessor memory allocation</td>
<td><code>alloc_if ( bool )</code></td>
<td>Allocate coprocessor space on this offload (default: TRUE)</td>
</tr>
<tr>
<td>Coprocessor memory release</td>
<td><code>free_if ( bool )</code></td>
<td>Free coprocessor space at the end of this offload (default: TRUE)</td>
</tr>
<tr>
<td>Control target data alignment</td>
<td><code>align ( N bytes )</code></td>
<td>Specify minimum memory alignment on coprocessor</td>
</tr>
<tr>
<td>Array partial allocation &amp; variable relocation</td>
<td><code>alloc ( array-slice ) into ( var-expr )</code></td>
<td>Enables partial array allocation and data copy into other vars &amp; ranges</td>
</tr>
</tbody>
</table>
Offload Capabilities

- Offload anything (even kernels)
- Synchronous offloads
- Asynchronous Offloads
- Multiple targets
Offloading “a kernel”
Elemental Function for SIMD using 1-core

```c
__declspec(vector)
double option_price_call_black_scholes(
    double S,       // spot (underlying) price
    double K,       // strike (exercise) price,
    double r,       // interest rate
    double sigma,   // volatility
    double time)    // time to maturity
{
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}
```

// execute on a single core
#pragma simd
for (int i=0; i<num_options; i++) {
    call[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
}
Offloading “a kernel”
Using OpenMP to use SIMD on all-cores

```c
__declspec (vector)
double option_price_call_black_scholes(
    double S,       // spot (underlying) price
    double K,       // strike (exercise) price,
    double r,       // interest rate
    double sigma,   // volatility
    double time)    // time to maturity

{
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}

// execute on a all cores
#pragma omp parallel for
#pragma simd
for (int i=0; i<num_options; i++) {
    call[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
}
```
Offloading “a kernel”
Offloading full kernel to Phi using all-cores & SIMD

```c
__declspec(target(mic))
__declspec (vector)
double option_price_call_black_scholes(
    double S,       // spot (underlying) price
    double K,       // strike (exercise) price,
    double r,       // interest rate
    double sigma,   // volatility
    double time)    // time to maturity
{
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}
```

// offload to Phi – running on all its cores and using SIMD!
#pragma offload target(mic) \
    in(S,K,time:length(num_options)) in(r,sigma) \
    out(call:length(num_options))
#pragma omp parallel for
#pragma simd
for (int i=0; i<num_options; i++) {
    call[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
}
Synchronous Offload mode

One pragma offload! Easy... but be careful...

// offload to Phi - running on all its cores and using SIMD!
#pragma offload target(mic) \
    in(S,K,time:length(num_options)) in(r,sigma) \
    out(call:length(num_options))
#pragma omp parallel for
#pragma simd
for (int i=0; i<num_options; i++) {
    call[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
}

Note that all necessary tasks of transferring data in both directions and computation are completed by just pragma here!

Easy to use but may not be performant!

- Host thread is blocked for the entire offload, including:
  - Data transfers in both directions
  - Computations on Phi
- Phi coprocessor’s compute capabilities are also unused during data transfers before and after the compute is completed
Black Scholes w/ Cilk Plus Offloaded to MIC

// This sample is derived from code published by Bernt Arne Odegaard http://finance.bi.no/~bernt/gcc_prog/recipes/recipes/
__declspec(target(mic)) __declspec(vector)
static double N(const double& z) {
    return (1.0/sqrt(2.0*PI))*exp(-0.5*z*z);
}

__declspec(target(mic)) __declspec(vector(uniform(r,sigma)))
double option_price_call_black_scholes( double S, double K, double r, double sigma, double time) {
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}

void test_option_price_call_black_scholes( double S[], double K[], double r, double sigma, double time[], double call[], int num_options) {
    #pragma offload target(mic)
    in(S, K, time : length(num_options))
    in(r, sigma)
    out(call : length(num_options))
    cilk_for (int i=0; i < num_options; i++) {
        call[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
    }
}
Synchronous Offload mode – Black Scholes

Building app

$ icpc -vec-report3 BlackScholes-SynchronousOffload.cpp -o bs-so-vec

BlackScholes-SynchronousOffload.cpp(102): (col. 22) remark: loop was not vectorized: statement cannot be vectorized.

BlackScholes-SynchronousOffload.cpp(121): (col. 19) remark: LOOP WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(112): (col. 4) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate.

BlackScholes-SynchronousOffload.cpp(34): (col. 1) remark: FUNCTION WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(34): (col. 1) remark: FUNCTION WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(49): (col. 1) remark: FUNCTION WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(49): (col. 1) remark: FUNCTION WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(62): (col. 45) remark: LOOP WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(62): (col. 45) remark: *MIC* LOOP WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(62): (col. 45) remark: *MIC* PEEL LOOP WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(62): (col. 45) remark: *MIC* REMAINDER LOOP WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(49): (col. 1) remark: *MIC* FUNCTION WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(49): (col. 1) remark: *MIC* FUNCTION WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(34): (col. 1) remark: *MIC* FUNCTION WAS VECTORIZED.

BlackScholes-SynchronousOffload.cpp(34): (col. 1) remark: *MIC* FUNCTION WAS VECTORIZED.
Synchronous Offload mode

Running app

$ export MIC_ENV_PREFIX=MIC
$
$ export MIC_CILK_NWORKERS=120
$
$ time ./bs-so-vec
num_options = 1048576, num_iterations = 256, chunk_size = 65536
...
real  0m12.350s
user  0m5.051s
sys  0m3.484s
$

Quite **slow** compared to native-mode due to overhead of data copy in both directions and synchronously waiting for all offload related operations!
Synchronous Offload mode

Looking at runtime report

$ export H_TRACE=1
$
$
$ time ./bs-so-vec 1000 1
num_options = 1000, num_iterations = 1, chunk_size = 65536
HOST: Offload function __offload_entry_BlackScholes_SynchronousOffload_cpp_114main, is_empty=0, #varDescs=8, #waits=0, signal=(nil)
HOST: Total pointer data sent to target: [24000] bytes
HOST: Total copyin data sent to target: [20] bytes
HOST: Total pointer data received from target: [16000] bytes
MIC0: Total copyin data received from host: [20] bytes
MIC0: Total copyout data sent to host: [0] bytes
HOST: Total copyout data received from target: [0] bytes
...
real  0m1.781s
user  0m0.301s
sys   0m0.100s
$
Support for Multiple Coprocessors

```c
#pragma offload target(mic [ :<expr> ] ) ...

coprocessor # = <expr> % number_of_devices
```

- Code must run on coprocessor #, aborts if not available (counts from 0)
- If -1, runtime chooses coprocessor, aborts if not available
- If not present, runtime chooses coprocessor or runs on host if none available

- APIs:    #include offload.h (C/C++);    USE MIC_LIB (Fortran)
  ```c
  int _Offload_number_of_devices() (C/C++)
  result = OFFLOAD_NUMBER_OF_DEVICES() (Fortran)
  ```
  - Returns # of coprocessors installed, or 0 if none

  ```c
  int _Offload_get_device_number() (C/C++)
  result = OFFLOAD_GET_DEVICE_NUMBER() (Fortran)
  ```
  - Returns coprocessor number where executed, (-1 for CPU)
  - Can use to share work explicitly by card number
Agenda

1. Enabling Rapidly Growing Parallelism
2. Intel Compiler Key Optimization Features
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO
   • Parallel programming models
3. Phi Hardware & Software-stack Overview
4. Phi Programming Models
   • Native mode
   • Offload mode – Synchronous & Asynchronous
5. Other Important Stuff
   • Data alignment
   • Numeric string conversion library
   • FP accuracy, reproducibility, performance
Asynchronous Offload mode

Splitting one synchronous offload into Five asynchronous smaller tasks

• An offload task really consists of 5 steps under-the-hood:
  1. Coprocessor data space allocation
  2. Input data copies to the coprocessor memory
  3. Offloaded execution on the coprocessor
  4. Coping of results back to the host processor memory
  5. De-allocation of data space allocated on the coprocessor

• All these steps were performed in single offload pragma during synchronous offload

• Intel compiler provides control to perform these tasks separately and creates opportunities for:
  • Asynchronous overlapped data-transfers
  • Asynchronous overlapped computation
  • Persistent data residing on coprocessor across multiple offload computations
Asynchronous Offload

• Asynchronous Data Transfers:
  • `#pragma offload_transfer target(mic:n) IN(....) signal(&s1)`
    – Standalone data offload
  • `#pragma offload_wait target(mic:n) wait(&s1)`
    – Standalone synchronization, host waits for transfer completion (blocking)

• Asynchronous Offload Computation:
  • `#pragma offload target(mic:n) wait(&s1) signal(&s2)`
    – Offload computation when data transfer has completed
    – Computation on host then continues in parallel
  • `#pragma offload_wait target(mic:n) wait(&s2)`
    – Host waits for signal that offload computation completed

• Non-blocking API to test signal value
## Asynchronous Offload mode

### Overlapped data-transfers & computations

<table>
<thead>
<tr>
<th>Ping</th>
<th>Pong</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong> Allocate input &amp; output data buffers</td>
<td>Allocate input &amp; output data buffers</td>
</tr>
<tr>
<td><strong>1</strong> Start sending Input data</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td><strong>2</strong> Wait for Input data transfer completion</td>
<td></td>
</tr>
<tr>
<td><strong>3</strong> Start Compute</td>
<td>Start sending Input data</td>
</tr>
<tr>
<td>...</td>
<td>A</td>
</tr>
<tr>
<td><strong>4</strong> Wait for Compute completion</td>
<td>...</td>
</tr>
<tr>
<td><strong>5</strong> Start receiving Output data</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td><strong>6</strong> Wait for Output data transfer completion</td>
<td>...</td>
</tr>
<tr>
<td><strong>1</strong> Start sending Input data</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td><strong>2</strong> Wait for Input data transfer completion</td>
<td></td>
</tr>
<tr>
<td><strong>3</strong> Start Compute</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td><strong>4</strong> Wait for Compute completion</td>
<td></td>
</tr>
<tr>
<td><strong>5</strong> Start receiving Output data</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td><strong>6</strong> Wait for Output data transfer completion</td>
<td></td>
</tr>
</tbody>
</table>

- Free input & output data buffers

- Free input & output data buffers
Asynchronous Offload
Allocate and De-allocate buffers on Phi

0. Allocating input/out data buffers for persistence

```c
// *** 0 ***
// ALLOCATE input & output PING-PONG memory buffers on Target
// _and_ RETAIN them till all calculations are completed
//
#pragma offload_transfer target(mic:mic_dev_num) \\
in(num_elements) \\
  nocopy(ping_in, ping_out, pong_in, pong_out \\
    : length(num_elements) ALLOC RETAIN)
```

7. De-allocating input/out data buffers after persistence

```c
// *** 7 ***
// DEALLOCATE input & output PING-PONG memory buffers on Target
//
#pragma offload_transfer target(mic:mic_dev_num) \\
in(num_elements) \\
  nocopy(ping_in, ping_out, pong_in, pong_out \\
    : length(num_elements) FREE)
```
Asynchronous Offload
Initiate send-to-Phi and wait-for-completion

1. Start sending input data from host to coprocessor

// *** 1 ***
// PING - Start sending Input data
//
#pragma offload_transfer target(mic:mic_dev_num) \\ in(ping_in : length(num_elements) REUSE RETAIN) \\ signal(&sig_ping_in)

2. Wait for data transfer completion

// *** 2 ***
// PING - Wait for Input data transfer completion
//
#pragma offload_wait target(mic:mic_dev_num) \\ wait(&sig_ping_in)
Asynchronous Offload
Initiate compute-on-Phi & wait-for-completion

3. Launch computation on the coprocessor

```c
// *** 3 ***
// PING - Start Compute
//
#pragma offload target(mic:mic_dev_num) \ 
nocopy(ping_in, ping_out : length(num_elements) \ 
        REUSE RETAIN) \ 
signal(&sig_ping_compute)
compute(ping_in, ping_out, num_elements);
```

4. Wait for computation to complete

```c
// *** 4 ***
// PING - Wait for Compute completion
//
#pragma offload_wait target(mic:mic_dev_num) \ 
wait(&sig_ping_compute)
```
Asynchronous Offload
Initiate send-to-Host & wait-for-completion

5. Start sending output data from coprocessor to host

// *** 5 ***
// PING - Start receiving Output data
//
#pragma offload_transfer target(mic:mic_dev_num) \ out(ping_out : length(num_elements) REUSE RETAIN) \ signal(&sig_ping_out)

6. Wait for data transfer completion

// *** 6 ***
// PING - Wait for Output data transfer completion
//
#pragma offload_wait target(mic:mic_dev_num) \ wait(&sig_ping_out)
Offload using Cilk Plus keywords
Shared VM Memory Model

• Coherence *simulated* & maintained between VMs of processor & coprocessor’s

• **Appropriate for complex pointer-based data structures**
  - Linked-lists, tree, etc.

• `__Cilk_shared` keyword used for
  - Sharing variables and Sharing functions

• `__Cilk_offload` keyword used for
  - Synchronous function offload
  - Asynchronous function offload

• Shared dynamic memory management
  - `__Offload_shared_malloc() / __Offload_shared_free()`
  - `__Offload_shared_aligned_malloc() / __Offload_shared_aligned_free()`

• Synchronization of data between processor and coprocessor
  - Compiler-runtime automatically maintains coherence at the beginning and end of the offload statements
  - Only modified data is transferred, of course
To handle more complex data structures on the coprocessor, use Virtual Shared Memory

An identical range of virtual addresses is reserved on both host and coprocessor: changes are shared at offload points, allowing:

- Seamless sharing of complex data structures, including linked lists
- Elimination of manual data marshaling and shared array management
- Freer use of new C++ features and standard classes

![Diagram showing shared virtual address space between Host VM and coproc VM](image)
Virtual Shared Memory uses special allocation to manage data sharing at offload boundaries

Declare virtual shared data using _Cilk_shared allocation specifier

Allocate virtual dynamic shared data using these special functions:

`_Offload_shared_malloc()`, `_Offload_shared_aligned_malloc()`,
`_Offload_shared_free()`, `_Offload_shared_aligned_free()`

Shared data copying occurs automatically around offload sections

- Memory is only synchronized on entry to or exit from an offload call
- Only modified data blocks are transferred between host and coprocessor

Allows transfer of C++ objects

- Pointers are transportable when they point to “shared” data addresses

Well-known methods can be used to synchronize access to shared data and prevent data races within offloaded code

- E.g., locks, critical sections, etc.

This model is integrated with the Intel® Cilk™ Plus parallel extensions

Note: Not supported on Fortran - available for C/C++ only
Data sharing between host and coprocessor can be enabled using this Intel® Cilk™ Plus syntax

<table>
<thead>
<tr>
<th>What</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td><code>int _Cilk_shared f(int x){ return x+1; }</code></td>
<td>Code emitted for host and target; may be called from either side.</td>
</tr>
<tr>
<td>Global</td>
<td><code>_Cilk_shared int x = 0;</code></td>
<td>Datum is visible on both sides.</td>
</tr>
<tr>
<td>File/Function static</td>
<td><code>static _Cilk_shared int x;</code></td>
<td>Datum visible on both sides, only to code within the file/function.</td>
</tr>
<tr>
<td>Class</td>
<td><code>class _Cilk_shared x {...};</code></td>
<td>Class methods, members and operators available on both sides.</td>
</tr>
<tr>
<td>Pointer to shared data</td>
<td><code>int _Cilk_shared *p;</code></td>
<td><code>p</code> is local (not shared), can point to shared data.</td>
</tr>
<tr>
<td>A shared pointer</td>
<td><code>int *_Cilk_shared p;</code></td>
<td><code>p</code> is shared; should only point at shared data.</td>
</tr>
<tr>
<td>Entire blocks of code</td>
<td><code>#pragma offload_attribute(push, _Cilk_shared)</code></td>
<td>Mark entire files or blocks of code <code>_Cilk_shared</code> using this pragma.</td>
</tr>
</tbody>
</table>
Agenda

1. Enabling Rapidly Growing Parallelism
2. Intel Compiler Key Optimization Features
   - Vectorization – auto, semi-auto, and explicit
   - IPO, PGO, HLO
   - Parallel programming models
3. Phi Hardware & Software-stack Overview
4. Phi Programming Models
   - Native mode
   - Offload mode – Synchronous & Asynchronous
5. **Other Important Stuff**
   - Data alignment
   - Numeric string conversion library
   - FP accuracy, reproducibility, performance
Aligning Data in C/C++

● Aligning memory allocated on heap
  ```c
  void* _mm_malloc(int size, int n)
  int posix_memaligned(void **p, size_t n, size_t size)
  ```

● Aligning memory on stack
  ```c
  __attribute__((aligned(n))) var_name or
  __declspec(align(n)) var_name
  ```

AND TELL the compiler at use...

#pragma vector aligned

• Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor

  ```c
  __assume_aligned(array, n)
  ```

• Compiler may assume array is aligned to n byte boundary

• May cause fault if data are not aligned

n=64 for Phi coprocessors, n=32 for AVX, n=16 for SSE

New attribute for assumed data alignment

__declspec(alignment_size)
__attribute__((align_value(alignment_size)))

- Ability to declare pointer alignment
- Data aligned by the designated pointer is aligned by the specified value
- Compiler now may generate optimized code based on this data-alignment assumption
- Example of adding this to a pointer typedef:
  
  typedef double* __restrict__ __attribute__((align_value (64))) RealPtr;
  RealPtr rp;

- If used incorrectly (data not aligned as promised), behavior/results would be undefined
New attribute to avoid false sharing

__declspec(avoid_false_share)
__declspec(avoid_false_share(identifier))
__attribute__((avoid_false_share))
__attribute__((avoid_false_share(identifier)))

• Indicates to compiler to allocate the specified variable through padding and/or alignment such that it won’t share the cache-line with other variables unless they share the same identifier

• Scope may be within a function or global/namespace

• Supported for scalar and arrays only (refer to docs)
Intel’s Numeric String Conversion Library

libistrconv

• Collection of routines for converting between ASCII strings and C data-types

• Rationale ➔ better performance!

• Need istrconv.h header and libistrconv library

• Floating point numbers to ASCII string
  • __IML_float_to_string()
  • __IML_double_to_string()

• ASCII string to floating point numbers
  • __IML_string_to_float()
  • __IML_string_to_double()
Floating Point (FP) Programming Objectives

- **Accuracy**
  - Produce results that are “close” to the correct value
  - Measured in relative error, possibly in ulp

- **Reproducibility**
  - Produce consistent results
    - From one run to the next
    - From one set of build options to another
    - From one compiler to another
    - From one platform to another

- **Performance**
  - Produce the most efficient code possible

These options usually conflict!
Judicious use of compiler options lets you control the tradeoffs.
FP Consistency – starting point

• The finite precision of floating-point operations leads to an inherent uncertainty in the results of a floating-point computation
  – Results may vary within this uncertainty
  – Usually, this is not a concern

• For some purposes, reproducibility beyond this uncertainty may be desired
  – Typically for reasons related to Quality Assurance

• “reproducible” is not necessarily more “accurate”
- **fp-model switch for icc**

- **-fp-model**
  - fast [=1] allows value-unsafe optimizations (default)
  - fast=2 allows additional approximations
  - precise value-safe optimizations only
  - source | double | extended imply “precise” unless overridden
    - see “FP Expression Evaluation” for more detail
  - except enable floating point exception semantics
  - strict precise + except + disable fma +
    - don’t assume default floating-point environment

- **-fp-model precise -fp-model source**
  - recommended for best reproducibility
  - also for ANSI/IEEE standards compliance, C++ & Fortran
  - “source” is default with “precise” on Intel 64 Linux
Comparing Floating-Point Results between Intel® Xeon processors and the Intel® Xeon Phi™ Coprocessor

• Different architectures – expect some differences
  – Different optimizations
  – Use of fused multiply-add (FMA)
  – Different implementations of math functions

• To minimize differences (e.g. for debugging)
  – Build with –fp-model precise (both architectures)
  – Build with –no-fma (Intel® MIC architecture)
  – Select high accuracy math functions
    – (e.g. -fimf-precision=high; default with –fp-model precise )
  – Choose reproducible parallel reductions (slides 19,33,34)
    – Or run sequentially, if you have the patience...
  – Remember, the true uncertainty of your result is probably much greater!
FP Consistency – Further Information

- Microsoft Visual C++* Floating-Point Optimization

- The Intel® C++ and Fortran Compiler Documentation, “Floating Point Operations”

- “Consistency of Floating-Point Results using the Intel® Compiler”

- “Differences in Floating-Point Arithmetic between Intel® Xeon® Processors and the Intel® Xeon Phi™ Coprocessor”

Summary

- To take full advantage of modern Intel® Architectures ...

**VECTORIZE + PARALLELIZE**

- Intel’s optimizing compilers (and libraries) take full advantage of the host CPU and Intel® Xeon Phi™ coprocessors
  => **Performance**

- The offload compiler makes parallel programming on Intel® MIC Architecture as easy as programming the host CPU
  => **Productivity**
Quick Steps For Tuning Using Intel Compilers

0. Before optimization, validate for code-correctness using /Od (-O0)

1. Use General Optimization options
   - Start with O2 and then move on to O3 to take advantage of HLO

2. Add Inter-Procedural Optimization (IPO)
   - Use ipo for multi-file or ip for single-file IPO

3. Add Profile Guided Optimization (PGO)
   - Use prof-gen to instrument binary, gather runtime data, use prof-use for optimized binary

4. Tune using vectorization for your target for processor(s)
   - Specific target xAVX for host or mmic for Phi native-mode
   - Auto-dispatch using axAVX
   - Align your data to appropriate boundary based on targeted processors/coprocessor

5. Use VTune to identify performance hotspots and then focus on those with compiler vectorization/optimization reports
   - Vectorization reports using vec-reportN
   - Tune up by guiding Intel compiler using explicit vectorization (pragmas/hints), if needed

6. Exploit thread-level parallelism (TLP) in many & multi-core architectures using Cilk Plus, TBB, OpenMP, MKL...
   - Use Intel Inspector XE for uncovering threading and memory errors and then
   - Use Intel VTune Amplifier XE to profile for efficient parallel scaling
Preserve Your Development Investment
Common Tools and Programming Models for Parallelism

C/C++
- Intel® Cilk Plus
- OpenCL*
- OpenMP*
- Intel® TBB
- Offload Pragmas
- Intel® C/C++ Compiler
- Intel® MKL

Fortran
- Intel® Fortran Compiler
- Coarray
- Offload Directives
- OpenMP*

Develop Using Parallel Models that Support Heterogeneous Computing
Resources

- http://software.intel.com/mic-developer
  - Developer’s Quick Start Guide
  - Programming Overview

- **Book** - Intel Xeon Phi Coprocessor High Performance Programming
  http://www.amazon.com/Intel-Xeon-Coprocessor-Performance-Programming/dp/0124104142


- Intel® Composer XE 2013 for Linux* User and Reference Guides
- Intel Premier Support https://premier.intel.com
Resources (2)

• Webinars:

• Recordings of Spring Webinars:
Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

All products, platforms, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice. All dates specified are target dates, are provided for planning purposes only and are subject to change.

This document contains information on products in the design phase of development. Do not finalize a design with this information. Revised information will be published when the product is available. Verify with your local sales office that you have the latest datasheet before finalizing a design.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

Code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel’s internal code names is at the sole risk of the user.

•Intel, the Intel logo, Intel Xeon, Intel VTune Intel Cilk and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2012, Intel Corporation. All rights reserved.
Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Thank you