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
1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. Phi Programming Models
8. Phi Offload-model Details
9. Vectorization for Phi
10. Phi Performance Tuning
Terminology

• CPU, Host, Intel Xeon Processor
  • Traditional x86 CPU’s such as SSEx/AVX capable processors
  • Drives servers, desktops, laptops etc.

• Phi, MIC, Target, Native, Intel Xeon Phi Coprocessor
  • Coprocessor based on MIC architecture
  • Coprocessor card plugged into host over PCIe
  • A host may have more than one target/Phi plugged in

• Notes:
  • Most topics presented here apply to both host-processors and target-coprocessors
  • Most features are also available for C/C++ & Fortran
  • Differences will be pointed out
Agenda

1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. Phi Programming Models
8. Phi Offload-model Details
9. Vectorization for Phi
10. Phi Performance Tuning
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-CPUs
      - 512-bit for Phi coprocessors

- **Thread-Level Parallelism (TLP)**
  - Multi-core architecture w/ & w/o Hyper-Thread (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
Agenda

1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. Phi Programming Models
8. Phi Offload-model Details
9. Vectorization for Phi
10. Phi Performance Tuning
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

- Industry-leading performance from advanced compilers
- Comprehensive libraries
- Parallel programming models
- Insightful analysis tools
Enabling & Advancing Parallelism
High Performance Parallel Programming

Intel tools, libraries and parallel models extend to multicore, many-core and heterogeneous computing

Use One Software Architecture Today. Scale Forward Tomorrow.
Code the Future
Top new features in 2013 release

<table>
<thead>
<tr>
<th>Performance</th>
<th>Performance Profiling</th>
<th>Reliability</th>
<th>Reproducibility</th>
<th>Standards</th>
<th>Parallelism Assistance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved compiler and library performance</td>
<td>A dozen new analysis features</td>
<td>Pointer checker</td>
<td>Conditional numerical reproducibility</td>
<td>Expanded C++ 11</td>
<td>Analysis extended to include Linux*, Fortran and C# (in addition to Windows* and C/C++)</td>
</tr>
<tr>
<td>+ Ivy Bridge microarchitecture</td>
<td>Low overhead Java* profiling</td>
<td>Heap growth analysis</td>
<td></td>
<td>Expanded Fortran 2008</td>
<td></td>
</tr>
<tr>
<td>+ Haswell microarchitecture</td>
<td>CPU Power Analysis</td>
<td>Improved MPI fault tolerance†</td>
<td></td>
<td>MPI 2.2†</td>
<td></td>
</tr>
<tr>
<td>+ Intel® Xeon Phi™ coprocessor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Efficiently produce fast, scalable and reliable applications, Linux* Windows*
Agenda

1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. **Why Intel Compiler?**
4. Intel compiler’s key features (Host CPU context)
   - Vectorization – auto, semi-auto, and explicit
   - IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. Phi Programming Models
8. Phi Offload-model Details
9. Vectorization for Phi
10. Phi Performance Tuning
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
      - Ever improving SIMD capabilities of each core (MMX ➔ SSE ➔ AVX, Phi)
      - Auto-Vectorization, Array Notations, SIMD-pragma, Elemental Functions
    - Thread Level Parallelism
      - 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 for simple gains
  - 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
Agenda

1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. Phi Programming Models
8. Phi Offload-model Details
9. Vectorization for Phi
10. Phi Performance Tuning
**Vectorization** is the process of transforming a scalar operation that acts on single data elements at a time (Single Instruction Single Data – SISD), to an operation that acts on multiple data elements at once (Single Instruction Multiple Data – SIMD).

- **Scalar mode**
  - one instruction produces one result

- **SIMD processing**
  - with SSE or AVX or MIC instructions
  - one instruction can produce multiple results

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

For example:

```
```
SIMD Potential Speedup

Double Precision FP vector width vs speedup potential

- 128 bit: 2x potential for SSE2
- 256 bit: 4x potential for AVX
- 512 bit: 8x potential for MIC

- Default at O2 or O3 is SSE2 – you can do better!
  - Wider vectors allow for higher potential performance gains
  - For DP, gains of up to 4x on AVX and up to 8x on MIC
  - For SP, gains of up to 8x on AVX and up to 16x on MIC

Compiler “safe” default for O2 or O3 is SSE2.
VLP / SIMD: Role of Intel Compilers & Libraries

- **Intel Compilers**
  - 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
  - **Auto-Vectorization** (details coming up!)
    - The fastest & easiest way; recommended for most cases
  - **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 exploitation of latest ( = best performing) SSE/AVX extension used
  - Critical routines implemented via manually written SSE/AVX-code - different versions for each SSE/AVX level
Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

Compiler:
Semi-Auto-vectorization using hints (#pragma ivdep, vector, ...)

Explicit Vector Programming

SIMD intrinsic class
(e.g.: F32vec, F64vec, ...)

Vector intrinsic
(e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

Assembler code
(e.g.: [v]addps, [v]addss, ...)

Ease of use

Programmer control
OpenMP & Vectorization paradigms compared

<table>
<thead>
<tr>
<th>Thread Level Parallelism</th>
<th>SIMD Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Auto-Parallel</strong></td>
<td><strong>Auto-Vectorization</strong></td>
</tr>
<tr>
<td>invoked by compiler switch, some</td>
<td>invoked at 02, some loops</td>
</tr>
<tr>
<td>loops parallelized automatically by</td>
<td>vectorized automatically by</td>
</tr>
<tr>
<td>compiler</td>
<td>compiler, developer can provide a few</td>
</tr>
<tr>
<td></td>
<td>hints to the compiler</td>
</tr>
<tr>
<td><strong>Explicit</strong> Thread-level</td>
<td><strong>Explicit</strong> SIMD-level</td>
</tr>
<tr>
<td>Parallelization using OpenMP</td>
<td>Vectorization using Intel® Cilk™ Plus</td>
</tr>
<tr>
<td>Developer guides parallelization</td>
<td>Developer guides vectorization via</td>
</tr>
<tr>
<td>via statements / pragmas / clauses</td>
<td>statements / pragmas / clauses</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>*<em>Parallelization using Posix</em> or</td>
<td><strong>Vectorization using Intrinsics</strong></td>
</tr>
<tr>
<td>Windows* Threads</td>
<td></td>
</tr>
</tbody>
</table>

Ease of use

Programmer control
Explicit Vector Programming with Cilk Plus, OpenMP 4.0 SIMD, Fortran

Map vector parallelism to vector ISA

- Fully Automatic Analysis
- Vectorization Hints (ivdep/vector pragmas)
- Array Notation
- SIMD pragma/directive
- Elemental Function
- Vectorizer makes retargeting easy!

Input: C/C++/FORTRAN source code

Vector part of Intel® Cilk™ Plus extension

Optimize and Code Generation

- Intel® SSE
- Intel® AVX
- Intel® MIC

Express/expose vector parallelism
Auto-Vectorization (Host)

- Auto-vectorizer exploits SIMD/VLP opportunities
  - Auto-vectorizes sequential operations using SSE and/or AVX instructions
  - No significant changes to source-code
  - Much easier to learn, debug, maintain, ...
  - 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)
Semi*-Auto-Vectorization Example
Using -fargument-noalias to help auto-vectorize

```c
void work( float* a, float *b, float *c, int MAX) {
    for (int I=0; I<=MAX; I++)
        c[I] = a[I] + b[I];
}
```

$ icpc -c work.cpp -xAVX -vec-report2 -fargument-noalias
work.cpp(2): (col. 6) remark: LOOP WAS VECTORIZED.
Semi*-Auto-Vectorization – Black Scholes

Using hint #pragma ivdep to help auto-vectorize

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

```cpp
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.
```
### Vectorizable Math functions (SVML)

<table>
<thead>
<tr>
<th>acos</th>
<th>ceil</th>
<th>fabs</th>
<th>round</th>
</tr>
</thead>
<tbody>
<tr>
<td>acosh</td>
<td>cos</td>
<td>floor</td>
<td>sin</td>
</tr>
<tr>
<td>asin</td>
<td>cosh</td>
<td>fmax</td>
<td>sinh</td>
</tr>
<tr>
<td>asinh</td>
<td>erf</td>
<td>fmin</td>
<td>sqrt</td>
</tr>
<tr>
<td>atan</td>
<td>erfc</td>
<td>log</td>
<td>tan</td>
</tr>
<tr>
<td>atan2</td>
<td>erfinv</td>
<td>log10</td>
<td>tanh</td>
</tr>
<tr>
<td>atanh</td>
<td>exp</td>
<td>log2</td>
<td>trunc</td>
</tr>
<tr>
<td>cbrt</td>
<td>exp2</td>
<td>pow</td>
<td></td>
</tr>
</tbody>
</table>

Also float versions, such as `sinf()`

Uses short vector math library, `libsvml`

New entry points for AVX, e.g.  
`__svml_pow4`  
`__svml_powf8`  
`cf for SSE`  
`__svml_pow2`  
`__svml_powf4`

Many routines in the `libsvml` math library are more highly optimized for Intel microprocessors than for non-Intel microprocessors.
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.
  - 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)`  
    - `#pragma simd`
  - 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.
**Why Didn’t My Loop Vectorize?**

- Get vectorization-report using
  - `/Qvec-report<n>` (Windows) or `–vec-report<n>` (Linux)

- “Loop was not vectorized” because:
  - “Not Inner Loop”
  - “Low trip count”
  - “Existence of vector dependence”
  - “Non-unit stride used”
  - “Mixed Data Types”
  - “Condition too Complex”
  - “Condition may protect exception”
  - “Top test could not be found”
  - “Subscript too complex”
  - “Unsupported Loop Structure”
  - “Contains unvectorizable statement at line XX”
  - “vectorization possible but seems inefficient”
  - “Operator unsuited for vectorization”
  - … (some more)
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)
Problems with C/C++ Pointers

- Hard for compiler to determine aliasing (pointers pointing to the same memory location)
  - Aliases may hide dependencies that make vectorization unsafe
- In simple cases, compiler may generate vectorized and unvectorized loop versions, and test for aliasing at runtime
- Otherwise, compiler may need help:
  - `-fargument-noalias` & similar switches
  - Use Intel® Cilk™ Plus array notation
  - “restrict” keyword with `-restrict` or `--std=c99` or by inlining
    - and now `__restrict__`
  - `#pragma ivdep` asserts no potential dependencies
    - Compiler still checks for proven dependencies
  - `#pragma simd` asserts no dependencies

```c
void saxpy(float *x, float *y, float* restrict z, float *a, int n) {
  #pragma ivdep
  for (int i=0; i<n; i++) z[i] = *a*x[i] + y[i];
}
```
Ways to Write Vector Code

(Auto-)Vectorization

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

```fortran
do i = 1, N
    A(i) = B(i) + C(i)
end do
```

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

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

```fortran
A = B + C
```

SIMD Pragma/Directive

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

```fortran
!DIR$ SIMD
do i = 1, N
    A(i) = B(i) + C(i)
end do
```

Elemental Function

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

Explicit Data Parallelism Based on C/C++ Arrays
Accessing a section of an array:

```c
float a[10], *b;
...
// allocate *b
...
* b[0:6] = a[2:6];
...```

**Example:**

<table>
<thead>
<tr>
<th>a: 0 1 2 3 4 5 6 7 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
</tr>
<tr>
<td>2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b: 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 4 5 6 7</td>
</tr>
</tbody>
</table>
Section of 2D array:

```c
float a[10][10], *b;
...
// allocate *b
...
b[0:10] = a[:,5];
...
```

Diagram: (Illustration of a 2D array with a slice highlighted and mapped to a linear array `b`.)
Strided section of an array:

```c
float a[10], *b;
...
// allocate *b
...
    b[0:3] = a[0:3:2];
...```

- **a:** 0 1 2 3 4 5 6 7 8 9
- **b:** 0 2 4
Intel® Cilk™ Plus Array Notation Operator Maps

• Most C/C++ operators are available for array sections:
  • +, -, *, /, %, <, ==, !=, >, |, &, ^, &&, ||, !, - (unary), + (unary),
    ++, --, +++, =, *=, /=, * (pointer de-referencing)

• Examples:

```
a[:,] * b[:,]  // element-wise multiplication
a[0:4][1:2] + b[1:2][0:4] // error, different rank sizes
a[0:4][1:2] + c    // adds scalar c to array section
```

• Operators are implicitly mapped to all elements of the array section operands.
• Operations on different elements can be executed in parallel without any ordering constraints.
• Array operands must have the same rank and size.
• Scalar operands are automatically expanded.
Intel® Cilk™ Plus Array Notation Assignment Maps

- Assignment operator applies in parallel to every element of array section on left hand side (LHS):

\[
\begin{align*}
  a[::][::] &= b[][2][::] + c; \\
  e[] &= d; \\
  e[] &= b[][1][::]; \quad // \text{error, different rank} \\
  a[::][::] &= e[]; \quad // \text{error, different rank}
\end{align*}
\]

- Recent change with 12.1: Overlapping RHS & LHS results in undefined behavior:

\[
a[1:s] = a[0:s] + 1; \quad // \text{undefined because of overlap}
\]

**Hint:** Size & Shape must be preserved
**Intel® Cilk™ Plus Array Notation**

**Gather & Scatter**

- When an array section occurs directly under a subscript expression, it designates a set of elements indexed by the values of the array section.

  ![Scatter]
  
  **Scatter:**
  
  \[a[b[0:s]] = c[:] \rightarrow \text{for}(i=0; i<s; i++) a[b[i]] = c[i];\]

  ![Gather]
  
  **Gather:**
  
  \[c[0:s] = a[b[:]] \rightarrow \text{for}(i=0; i<s; i++) c[i] = a[b[i]];\]

- Compiler can generate scatter and gather instructions on supported hardware (like Intel® MIC) for irregular vector access.

- Compiler can also generate AVX2 which provides support for a gather instruction only
Intel® Cilk™ Plus Array Notation

Reductions

Combine array section elements using a predefined operator, or a user function:

```
int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[:]); // sum is 10
res = __sec_reduce(func, a[:], 0);
  // apply function func to all
  // elements in a[], identity value is 0
int func(int arg1, int arg2);
```

Other reductions:

```
__sec_reduce_mul, __sec_reduce_all_zero,
__sec_reduce_all_nonzero,
__sec_reduce_any_nonzero, __sec_reduce_max,
__sec_reduce_min, __sec_reduce_max_ind,
__sec_reduce_min_ind
```

Intel® Cilk™ Plus Array Notation
Shift/Rotate

Create expressions containing the array index value:

```c
b[:] = __sec_shift (a[:], signed shift_val, fill_val)
b[:] = __sec_rotate (a[:], signed rotate_val)
```

- Shift elements in `a[:]` to the right/left by `shift_val`
- The leftmost/rightmost element will get `fill_val` assigned
- Rotate will circular-shift elements in `a[:]` to the right/left by `rotate_val`
- Result is assigned to `b[:]`
- Argument `a[:]` is not modified
Array Notations in conditions:

If \( a[:,] < b[:,] \) {
    mask[:,] = -1; // all elements of \( a \) are smaller than \( b \)
} else {
    mask[:,] = 1;
}
// mask[n] contains -1 if \( a[n] < b[n] \), 1 if \( a[n] \geq b[n] \)

// alternative:
mask[:,] = a[:,] < b[:,] ? -1 : 1;
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
  - Annotates it with `__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):

\[
\text{declspec(vector)} \quad \text{float foo(float a, float b, float c, float d)} \\
\text{\{} \\
\text{\quad return a * b + c * d;} \\
\text{\}}
\]

- Call the scalar version:

\[
e = \text{foo}(a, b, c, d);
\]

- Call scalar version via SIMD loop:

\[
\text{\#pragma simd} \\
\text{for(i = 0; i < n; i++) \{} \\
\text{\quad A[i] = foo(B[i], C[i], D[i], E[i]);} \\
\text{\}}
\]

- Call it with array notations:

\[
\text{A[:]} = \text{foo}(\text{B[:]}, \text{C[:]}, \text{D[:]}, \text{E[:]});
\]
Elemental Functions Syntax

• C/C++
  
  __declspec(vector(clauses)) Windows*
  __attribute__((vector(clauses))) Linux*

• OpenMP 4.0
  
  #pragma omp declare simd

• Add thedeclspec clause to both the function definition as well as the function prototype or header

• Add optional optimization clauses:
  
  – uniform(param1[, param2]...):
    Scalar parameters are broadcasted to all iterations
  – linear(param1:step1[, param2:step2]...):
    In serial execution parameters are incremented by steps
  – processor(cpuid):
  – vectorlength(num):
    If not set vector length is selected by arguments
Elemental Functions

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

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

- Call the scalar version:

```c
e = foo(a, b, c, d);
```

- Call scalar version via SIMD loop:

```c
#pragma simd
for(i = 0; i < n; i++) {
    A[i] = foo(B[i], C[i], D[i], E[i]);
}
```

- Call it with array notations:

```c
A[:] = foo(B[:], C[:], D[:], E[:]);
```
Elemental Functions: Linear/ Uniform I

• Why do we need them? Because vector loads and stores of IA are optimized for accessing consequent elements in memory (e.g., \([v]\text{movups}\)).
• They are most useful when consumed in the address computation.

```c
__declspec(vector(uniform(a), linear(i:1))) void foo(float *a, int i):
    a is a pointer
    i is a sequence of integers \([i, i+1, i+2, ...]\)
    \(a[i]\) is a unit-stride load/store (\([v]\text{movups}\))
```

```c
__declspec(vector) void foo(float *a, int i):
    a is a vector of pointers
    i is a vector of integers
    \(a[i]\) becomes gather/scatter.
```
## 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
SIMD Motivation

• Provide 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
for(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 SIMD Pragma/Directive

Programmer asserts:

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

```c
#pragma simd reduction(+:sum)
for(i = 0; i < \*p; i++) {
    A[i] = B[i] * C[i];
    sum = sum + A[i];
}
```

Explicit vector programming lets you express what you mean!
SIMD Pragma/Directive Notation

C/C++:  #pragma simd [clause [,clause] ...]

Fortran:  !$OMP SIMD [clause [,clause] ...]

• Primarily targets loops
  • Can target inner or outer loops
  • Provides a lexicon of clauses to modify behavior of SIMD directive
  • Developer informs the compiler that a given loop can be vectorized, and clarifies data usage patterns
SIMD Pragma/Directive Clauses

reduction(operator:v1, v2, …)
- v1 etc are reduction variables for operation “operator”

private(v1, v2, …)
- variables private to each iteration; initial value is broadcast to all private instances, and the last value is copied out from the last iteration instance.

linear(v1:step1, v2:step2, …)
- for every iteration of original scalar loop, v1 is incremented by step1, … etc. Therefore it is incremented by step1 *(vector length) for the vectorized loop.
- n1, n2, … must be 2, 4, 8 or 16: The compiler can assume a vectorization for a vector length of n1, n2, … to be safe

Others: {Intel} [no]assert, vectorlength(n1, [,n2] …)
{OpenMP} [no]assert, vectorlength(n1, [,n2] …)
Solution:
If, for example, offsets are at least 4 elements, vectorization is still possible as vector length can be controlled via `#pragma simd`:

```cpp
#pragma simd
C++ Example:

Pi Using Monte Carlo

#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++;
    }
}
```

This program results in good utilization of vector level parallelism and provides measurable speedups.
// 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 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. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. Phi Programming Models
8. Phi Offload-model Details
9. Vectorization for Phi
10. Phi Performance Tuning
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 ➔ single file IPO
    - Inlining functions defined within same file
  - icpc -ipo ➔ multi-file IPO
    - Inlining functions defined across multiple files

**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:
  ```cpp
  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.
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.
The -fp-model (/fp:) switch

- Lets you choose the floating point semantics at a coarse granularity

- **fp-model:**
  - fast \([=1]\) allows value-unsafe optimizations (default)
  - fast=2 allows additional approximations
  - precise value-safe optimizations only
    (also source, double, extended)
  - except enable floating point exception semantics
  - strict precise + except + disable fma

- Replaces -mp, -float-consistency, etc.

- Lets you control tradeoffs and achieve these conflicting goals

- **-fp-model precise -fp-model source**
  - recommended for ANSI/IEEE standards compliance C++/Fortran

- For details, please refer to:
A Family of Parallel Programming Models
Developer Choice

<table>
<thead>
<tr>
<th>Intel® Cilk™ Plus</th>
<th>Intel® Threading Building Blocks</th>
<th>Domain-Specific Libraries</th>
<th>Established Standards</th>
<th>Research and Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ language extensions to simplify parallelism</td>
<td>Widely used C++ template library for parallelism</td>
<td>Intel® Integrated Performance Primitives</td>
<td>Message Passing Interface (MPI)</td>
<td>Intel® Concurrent Collections</td>
</tr>
<tr>
<td>Open sourced</td>
<td>Open sourced</td>
<td>Intel® Math Kernel Library</td>
<td>OpenMP*</td>
<td>Offload Extensions</td>
</tr>
<tr>
<td>Also an Intel product</td>
<td>Also an Intel product</td>
<td></td>
<td>Coarray Fortran</td>
<td>Intel® Array Building Blocks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OpenCL*</td>
<td>Intel® SPMD Parallel Compiler</td>
</tr>
</tbody>
</table>

**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.

*Other brands and names are the property of their respective owners.
Parallel Models: Few Recommendations

- Cilk Plus Array Notation for Vector Parallelism
  - Easily apply to new & existing apps for predictable vectorization
  - Interoperate with other threading models – TBB, Cilk, native-threads...

- Cilk Plus for Task Parallelism
  - Simplest & most debuggable parallel code
  - Shared non-local variables

- TBB for Task Parallelism
  - General task creation & scheduling
  - Portability to non-Intel compilers and CPUs

- OpenMP
Intel® Cilk™ Plus

Key Benefits

- Compiler supported solution offering a tasking system via 3 simple keywords
- Includes array notation to specify vector code
- Fork/join tasking system is simple to understand and mimics serial behavior
- Low overhead tasks offer scalability to high core counts
- Reducers give better performance than mutex locks and maintain serial semantics
- Mixes with Intel® TBB for a complete task and vector parallel solution

What is it?

- Compiler supported solution offering a tasking system via 3 simple keywords
- Includes array notation to specify vector code
- Reducers - powerful parallel data structures to efficiently prevent races
- Based on 15 years of research at MIT
- Pragmas to force vectorization of loops and attributes to specify functions that can be applied to all elements of arrays

Simple syntax which is very easy to learn and use
Array notation guarantees fast vector code
Fork/join tasking system is simple to understand and mimics serial behavior
Low overhead tasks offer scalability to high core counts
Reducers give better performance than mutex locks and maintain serial semantics
Mixes with Intel® TBB for a complete task and vector parallel solution
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:
  ```cpp
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:
  ```cpp
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]);
    }
}
Vector and Thread Parallelism Together

- Write function in array notation that handles a size-$m$ “chunk” of data
- Call the function on multiple chunks of data, in parallel, using multi-threading

```c
void saxpy_vec(int m, float a, float x[m], float y[m]) {
    y[:] += a * x[:];    // Vector code for size m data
}

void main(void) {
    int a[2048], b[2048];
    cilk_for (int i = 0; i < 2048; i +=256) {
        // Call function on size-256 chunks in parallel
        saxpy_vec(256, 2.0, &a[i], &b[i]);
    }
}
```
#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 Execution of Black Scholes

- 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 and with vectorization

$ icpc -no-vec -no-simd -vec-report3 BlackScholes.cpp -o bs-no_vec$
$
$ 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

$ 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. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO, Parallel programming models

5. **Phi Hardware Overview**
6. Phi Software Stack Overview
7. Phi Programming Models
8. Phi Offload-model Details
9. Vectorization for Phi
10. Phi Performance Tuning
Intel® Xeon Phi™ Coprocessor core
Fully functional multi-thread execution unit

- >50 in-order cores
  - Ring interconnect

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
- Round-robin execution hides scalar unit latency
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, L2 = 30.5MB (= 512KB/core x 61 cores)
    - if all code and data is shared among all cores, L2 = 512KB
**Phi Card Hardware Overview**

- **Highly Parallel** device!!
- **SMP on-a-chip** best describes Intel Xeon Phi Coprocessor
- First product that’s based on Intel MIC architecture
- Individual cores are tied together via fully coherent caches into a bidirectional ring

---

**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

---

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

1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. **Phi Software Stack Overview**
7. Phi Programming Models
8. Phi Offload-model Details
9. Vectorization for Phi
10. Phi Performance Tuning
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
- User-level code
- System-level code
- Intel® Xeon Phi™ Coprocessor
  - Support libraries, tools, and drivers

Intel® Xeon Phi™ Coprocessor
- Target-side offload application
  - User code
  - Offload libraries, user-accessible APIs and libraries
- User-level code
- System-level code
- Intel® Xeon Phi™ Coprocessor communication and application-launch support

Linux* OS
- PCI-E Bus

Copyright © 2013, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
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

Virtual terminal session

Intel® Xeon Phi™ Coprocessor

**Use if**
- Not serial
- Modest memory
- Complex code
- No hot spots

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

Intel® Xeon Phi™ Coprocessor communication and application-launch support

User-level code

System-level code

**User code**

System-level code

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

Linux* OS

PCI-E Bus

IB fabric

**System-level code**

Intel® Xeon Phi™ Coprocessor

Linux* OS

PCI-E Bus
Agenda

1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. **Phi Programming Models**
8. Phi Offload-model Details
9. Vectorization for Phi
10. Phi Performance Tuning
Spectrum of Programming Models and Mindsets

Multi-Core Centric

Multi-Core Hosted
General purpose serial and parallel computing

Symmetric
Codes with balanced needs

Many-Core Centric

Many Core Hosted
Highly-parallel codes

Offload
Codes with highly-parallel phases

Main( )
Foo( )
MPI_*( )

Main( )
Foo( )
MPI_*( )

Main( )
Foo( )
MPI_*( )

Main( )
Foo( )
MPI_*( )

Range of models to meet application needs
IA Benefit: Wide Range of Development Options

Breadth

Multi-Core Centric

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

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)

Ease of use

Depth

Fine control

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
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
    
    `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.
  
  ```
  ssh mic0 ‘mkdir /tmp/mydir’    (or mic1:, mic2:, etc)
  ```

- Copy executable, any dependencies and small data files onto coprocessor using `scp`, e.g.:
  
  ```
  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
Running Native Applications

• Use `ssh` to get console access (as root)
  
  `ssh mic0`
  
  `cd /tmp/mydir`

• Set permissions (if necessary) and environment
  
  • `export LD_LIBRARY_PATH=./LD_LIBRARY_PATH`
  
  • May need to increase stack limit with “`ulimit -s`”
  
  • Run app just like on host using “`./a.out <args>`”

• Alternatively, remotely submit a shell script that sets environment and runs app with `ssh mic0 './myscript.sh'`
  
  • Useful for performance analysis with Intel® Vtune™ Amplifier XE

• The Intel® Xeon Phi™ Coprocessor runs a reduced form of Linux*
  
  • Many familiar commands are available
  
  • `top`, `time`, ...

• Can NFS mount file systems from host & elsewhere
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 and with vectorization

```
$ icpc -mmic -no-vec -no-simd -vec-report3 BlackScholes.cpp -o bs-mic-no_vec
$
```
```
$ 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.
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

# ./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

# ./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

# 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

# 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
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>

**Execution Flow**

- **Intel Host Processor**
- **Your Application**
  - With identified Compute Intensive Kernels
- **Intel® MIC Co-processor(s)**
- **Host Offload Library**
- **Target Offload Library**
- **Message Library**
- **Multicore**
- **Many-core**

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

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

_f_part1_() {
    a = b + g();
}

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

__attribute__((target(mic)))
h() {
    ...
}
```

**Host**
Intel® Xeon® processor

**Target**
Intel® Xeon Xeon Phi™ coprocessor
Agenda

1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   • Vectorization – auto, semi-auto, and explicit
   • IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. Phi Programming Models
8. **Phi Offload-model Details**
   • Synchronous & Asynchronous offload
9. Vectorization for Phi
10. Phi Performance Tuning
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 offload target (mic)
    #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\n",pi/N);
}
```

One Line Change to Offload to MIC Co-Processor

OpenMP* is Applicable to Multicore and Many-core Programming

Intel® Xeon® processor

Intel® MIC co-processor
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 → Allocating memory for parts of C/C++ array
  - Moving data from one variable into the another

• Data persistence
  - Avoid unneeded allocation/de-allocation of data-buffers
  - alloc_if/free_if → ALLOC/RETAIN/REUSE/FREE
Language Extensions for Offload (pragmas/directives)

- **Offload** **pragma/directive** for data marshalling
  - 
    \[
    \#\text{pragma offload} \text{<clauses>}\]
    in C/C++
    Offloads the following OpenMP block or Intel® Cilk™ Plus construct or function call or compound statement
  - 
    \[
    !\text{dir$ offload} \text{<clauses>}\]
    in Fortran
    Offloads the following OpenMP block or subroutine/function call
    \[
    \text{RESULT} = \text{FUNC}(A,B) \quad \text{! but not RESULT} = \text{SCALE} \times \text{FUNC}(A,B)
    \]
  - 
    \[
    !\text{dir$ offload begin} \text{<clauses>...}
    !\text{dir$ end offload}
    \]
    to offload other block of code

- 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`

```c
// Traditional "Hello World" from Phi

#pragma offload target(mic)
printf("Hello World\n");

// Offloading your function

__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

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

    g_count++; // accessing on host processor
```

```cpp
#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!

```cpp
int count = 0;

    count++; // accessing on host processor
```

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

<table>
<thead>
<tr>
<th></th>
<th>Fortran Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offload directive</td>
<td><code>!dir$ omp offload &lt;clauses&gt; &lt;statement&gt;</code></td>
</tr>
<tr>
<td></td>
<td>Execute OpenMP* parallel block on coprocessor</td>
</tr>
<tr>
<td></td>
<td><code>!dir$ offload &lt;clauses&gt; &lt;statement&gt;</code></td>
</tr>
<tr>
<td></td>
<td>Execute next statement or function on coproc.</td>
</tr>
<tr>
<td>Variable/function</td>
<td><code>!dir$ attributes offload:&lt;mic&gt; :: &lt;ret-name&gt; OR &lt;var1,var2,...&gt;</code></td>
</tr>
<tr>
<td>offload properties</td>
<td>Compile function or variable for CPU and coprocessor</td>
</tr>
<tr>
<td>Entire code blocks</td>
<td><code>!dir$ offload begin &lt;clauses&gt;</code> !dir$ end offload</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;</td>
<td><code>alloc ( array-slice )</code></td>
<td>Enables partial array allocation and data copy into other vars &amp; ranges</td>
</tr>
<tr>
<td>variable relocation</td>
<td><code>into ( var-expr )</code></td>
<td></td>
</tr>
</tbody>
</table>
Offload Capabilities

- Offload anything (even kernels)
- Synchronous offloads
- Asynchronous Offloads
- Multiple targets
## Offloading “a kernel”

### Elemental Function for SIMD

```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);
}
```

Each vector lane can execute an instance of the function
Offloading “a kernel”
Elemental Function for SIMD using 1-core

```c
#pragma spec (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
```c
#pragma spec 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
- Phi coprocessor’s compute capabilities are also unused during
data transfers before and after the compute is completed
Synchronous Offload mode

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 WASVECTORIZED.

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

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
$
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
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_ofDevices() (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
Asynchronous Offload

• New synchronization clauses SIGNAL(&x) and WAIT(&x)
  • Argument is a unique address (usually of the data being transferred)

• Asynchronous Data Transfer:
  • #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

• There is also a 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>0</td>
<td>Allocate input &amp; output data buffers</td>
</tr>
<tr>
<td>1</td>
<td>Start sending Input data</td>
</tr>
<tr>
<td></td>
<td>Start sending Input data</td>
</tr>
<tr>
<td>2</td>
<td>Wait for Input data transfer completion</td>
</tr>
<tr>
<td>3</td>
<td>Start Compute</td>
</tr>
<tr>
<td></td>
<td>Start Compute</td>
</tr>
<tr>
<td>4</td>
<td>Wait for Compute completion</td>
</tr>
<tr>
<td>5</td>
<td>Start receiving Output data</td>
</tr>
<tr>
<td></td>
<td>Start receiving Output data</td>
</tr>
<tr>
<td></td>
<td>Wait for Compute completion</td>
</tr>
<tr>
<td></td>
<td>Wait for Output data transfer completion</td>
</tr>
<tr>
<td>6</td>
<td>Wait for Output data transfer completion</td>
</tr>
<tr>
<td>1</td>
<td>Start sending Input data</td>
</tr>
<tr>
<td></td>
<td>Start sending Input data</td>
</tr>
<tr>
<td>2</td>
<td>Wait for Input data transfer completion</td>
</tr>
<tr>
<td>3</td>
<td>Start Compute</td>
</tr>
<tr>
<td></td>
<td>Start Compute</td>
</tr>
<tr>
<td>4</td>
<td>Wait for Compute completion</td>
</tr>
<tr>
<td>5</td>
<td>Start receiving Output data</td>
</tr>
<tr>
<td></td>
<td>Start receiving Output data</td>
</tr>
<tr>
<td></td>
<td>Wait for Compute completion</td>
</tr>
<tr>
<td></td>
<td>Wait for Output data transfer completion</td>
</tr>
<tr>
<td>6</td>
<td>Wait for Output data transfer completion</td>
</tr>
</tbody>
</table>

- Ping
- Pong

**Notes:**

- Allocate input & output data buffers
- Start sending Input data
- Wait for Input data transfer completion
- Start Compute
- Start receiving Output data
- Wait for Compute completion
- Wait for Output data transfer completion
- Start sending Input data
- Free input & output data buffers
Asynchronous Offload
Allocate and De-allocate buffers on Phi

0. Allocating input/out data buffers for persistence

// *** 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

// *** 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)
3. Launch computation on the coprocessor

```c
#pragma offload target(mic:mic_dev_num) \ 
  nocopy(ping_in, ping_out : length(num_elements) \ 
  REUSE RETAIN) \ 
signal(&sig_ping_compute)

calculate(ping_in, ping_out, num_elements);
```

4. Wait for computation to complete

```c
#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

```c
// *** 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

```c
// *** 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 dealing with 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
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>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td><code>int _Cilk_shared f(int x){ return x+1; }</code></td>
</tr>
<tr>
<td></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>
</tr>
<tr>
<td></td>
<td>Datum is visible on both sides</td>
</tr>
<tr>
<td>File/Function</td>
<td><code>static _Cilk_shared int x;</code></td>
</tr>
<tr>
<td></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>
</tr>
<tr>
<td></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>
</tr>
<tr>
<td></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>
</tr>
<tr>
<td></td>
<td><code>p</code> is shared; should only point at shared data</td>
</tr>
<tr>
<td>Entire blocks of code</td>
<td>`#pragma offload_attribute( push, _Cilk_shared)</td>
</tr>
<tr>
<td></td>
<td><code>#pragma offload_attribute(pop)</code></td>
</tr>
<tr>
<td></td>
<td>Mark entire files or blocks of code _Cilk_shared using this pragma</td>
</tr>
</tbody>
</table>
Preprocessor Macros

• __INTEL_OFFLOAD
  • Set automatically unless disabled by –no-offload (or –mmic)
  • Set for the host compilation but not the target (coprocessor) compilation
  • Use to protect code on the host that is specific for offload
e.g. `omp_num_set_threads_target()` family of APIs
  but must remember to set –no-offload for host-only builds

• __MIC__
  • NOT set for host compilation in an offload build
  • Set automatically for target (coprocessor) compilation in offload build
  • Also set automatically when building native coprocessor application
  • Use to protect code that is compiled & executed only on coprocessor
e.g. `_mm512` intrinsics
Agenda

1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   - Vectorization – auto, semi-auto, and explicit
   - IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. Phi Programming Models
8. Phi Offload-model Details
9. **Vectorization for Phi**
10. Phi Performance Tuning
MIC SIMD support

For best performance, it’s not sufficient to use all the cores, you need to use the 512 bit SIMD registers and instructions

Vector classes and intrinsics are supported for C/C++
- See micvec.h and zmmmintrin.h in the include/mic directory
- Just include <immintrin.h>, the compiler takes care of the rest.

Semi/Auto-Vectorization for Intel® MIC architecture works just like for SSE or AVX on the host
- Data alignment should be to 64 bytes (512 bits)

Because of the greater SIMD width, vectorization is even more important on Intel® MIC architecture than on Intel® Xeon® processors. The Intel compiler now supports

Explicit Vector Programming
- Via Intel® Cilk™ Plus language extensions
- Via the SIMD constructs from OpenMP 4.0 RC1
Vectorization (MIC)

- The vectorizer for Intel® MIC architecture works just like for SSE or AVX on the host, for C, C++ and Fortran
  - Enabled at default optimization level (-O2)
  - Data alignment should be to 64 bytes, instead of 16 (see later)
  - More loops can be vectorized, because of masked vector instructions, gather/scatter instructions, fused multiply-add (FMA)
  - Avoid 64 bit integers where not essential

- Vectorized loops may be recognized by:
  - Vectorization and optimization reports (simplest), e.g.
    - -vec-report2 or -opt-report-phase hpo
  - Unmasked vector instructions (there are no separate scalar instructions; masked vector instructions are used instead)
  - Gather & scatter instructions
  - Math library calls to libsvml
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]);
    }
}
Auto-Vectorization Report – Black Scholes

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

$ icpc -c -vec-report3 BlackScholes.cpp

BlackScholes.cpp(16): (col. 34) remark: FUNCTION WAS VECTORIZED
BlackScholes.cpp(16): (col. 34) remark: FUNCTION WAS VECTORIZED
BlackScholes.cpp(16): (col. 34) remark: FUNCTION WAS VECTORIZED
BlackScholes.cpp(16): (col. 34) remark: FUNCTION WAS VECTORIZED
BlackScholes.cpp(16): (col. 34) remark: FUNCTION WAS VECTORIZED
BlackScholes.cpp(21): (col. 67) remark: FUNCTION WAS VECTORIZED
BlackScholes.cpp(21): (col. 67) remark: FUNCTION WAS VECTORIZED
BlackScholes.cpp(21): (col. 67) remark: FUNCTION WAS VECTORIZED
BlackScholes.cpp(21): (col. 67) remark: FUNCTION WAS VECTORIZED
BlackScholes.cpp(34): (col. 45) remark: LOOP WAS VECTORIZED
BlackScholes.cpp(16): (col. 34) remark: *MIC* FUNCTION WAS VECTORIZED
BlackScholes.cpp(16): (col. 34) remark: *MIC* FUNCTION WAS VECTORIZED
BlackScholes.cpp(34): (col. 45) remark: *MIC* LOOP WAS VECTORIZED
BlackScholes.cpp(34): (col. 45) remark: *MIC* PEEL LOOP WAS VECTORIZED
BlackScholes.cpp(34): (col. 45) remark: *MIC* REMAINDER LOOP WAS VECTORIZED
BlackScholes.cpp(21): (col. 67) remark: *MIC* FUNCTION WAS VECTORIZED
BlackScholes.cpp(21): (col. 67) remark: *MIC* FUNCTION WAS VECTORIZED
1. Rapidly Growing Parallelism
2. Enabling Advancing Parallelism
3. Why Intel Compiler?
4. Intel compiler’s key features (Host CPU context)
   - Vectorization – auto, semi-auto, and explicit
   - IPO, PGO, HLO, Parallel programming models
5. Phi Hardware Overview
6. Phi Software Stack Overview
7. Phi Programming Models
8. Phi Offload-model Details
9. Vectorization for Phi

10. Phi Performance Tuning (WIP)
Aligning Data in C/C++

- Allocate memory on heap aligned to n byte boundary:
  ```c
  void* _mm_malloc(int size, int n)
  int posix_memaligned(void **p, size_t n, size_t size)
  ```
- Alignment for variable declarations:
  ```c
  __attribute__((aligned(n))) var_name or __declspec(align(n)) var_name
  ```

**AND TELL the compiler at use...**

```c
#pragma vector aligned
```
- Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
  - May cause fault if data are not aligned
  ```c
  __assume_aligned(array, n)
  ```
- Compiler may assume array is aligned to n byte boundary

```
**n=64 for Intel® Xeon Phi™ coprocessors, n=32 for AVX, n=16 for SSE**
Prefetching – Automatic

• Compiler prefetching is on by default for the Intel® Xeon Phi™ coprocessor at –O2 and above
  • Prefetches issued for regular memory accesses inside loops
  • But not for indirect accesses \( a[\text{index}[i]] \)
  • More important for Intel Xeon Phi coprocessor (in-order) than for Intel® Xeon® processors (out-of-order)
  • Very important for apps with many L2 cache misses

• Use the compiler reporting options to see detailed diagnostics of prefetching per loop
  
  \(-\text{opt-report-phase hlo} -\text{opt-report} \ 3 \quad \text{e.g.}\)

  Total # of lines prefetched in main for loop at line 49=4
  Using noloc distance 8 for prefetching unconditional memory reference in stmt at line 49
  Using second-level distance 2 for prefetching spatial memory reference in stmt at line 50

  -opt-prefetch=n \ (4 = most aggressive) to control
  -opt-prefetch=0 \ or \ -\text{no-opt-prefetch} to disable
Prefetching – Manual

• Use intrinsics

```
_mm_prefetch((char *) &a[i], hint);
```

See xmmintrin.h for possible hints (for L1, L2, non-temporal, ...)

```
MM_PREFETCH(A, hint)    for Fortran
```

• But you have to figure out and code how far ahead to prefetch

• Also gather/scatter prefetch intrinsics, see zmmmintrin.h and compiler user guide, e.g. _mm512_prefetch_i32gather_ps

• Use a pragma / directive (easier):

```
#pragma prefetch a    [:hint[:distance]]
!DIR$ PREFETCH A, B, ...
```

• You specify what to prefetch, but can choose to let compiler figure out how far ahead to do it.

• Hardware L2 prefetcher is also enabled by default
  • If software prefetches are doing a good job, then hardware prefetching does not kick in
OpenMP defaults

- OMP_NUM_THREADS defaults to
  - 1 x ncore for host (or 2x if hyperthreading enabled)
  - 4 x ncore for native coprocessor applications
  - 4 x (ncore-1) for offload applications
    - one core is reserved for offload daemons and OS

- Defaults may be changed via environment variables or via API calls on either the host or the coprocessor
Target OpenMP environment (offload)

- Use target-specific APIs to set for coprocessor target only, e.g.
  
  ```
  omp_set_num_threads_target() (called from host)
  omp_set_nested_target() etc
  ```

- Protect with `#ifdef __INTEL_OFFLOAD`, undefined with `–no-offload`
- Fortran: `USE MIC_LIB` and `OMP_LIB` C: `#include offload.h`

- Or define MIC – specific versions of env vars using
  
  ```
  MIC_ENV_PREFIX=MIC (no underscore)
  ```

- Values on MIC **no longer default to values on host**
- Set values specific to MIC using
  
  ```
  export MIC_OMP_NUM_THREADS=120 (all cards)
  export MIC_2_OMP_NUM_THREADS=180 for card #2, etc
  export MIC_3_ENV="OMP_NUM_THREADS=102|KMP_AFFINITY=balanced"
  ```
OpenMP Thread Affinity

• On the coprocessor OS, the logical processor numbering is not the same as the numbering of the hardware thread contexts.
  • The last physical core is used for kernel & low level SCIF/COI threads
    – This is why OMP_NUM_THREADS defaults to 4*(ncore-1) for offloads
  • This corresponds to the first and the last three logical processors:
    – 0, 241,242,243 for 61 cores,
    – The offload runtime will usually try to avoid these
    – KMP_AFFINITY (and pthread_setaffinity calls and /proc/cpuinfo ) use this logical processor numbering
      • So be aware of this if you use explicit processor lists for KMP_AFFINITY
      • Easier to use “compact”, “scatter”, or “balanced” (new, coprocessor only)
  • Otherwise, the hardware threads map to logical processors in sequence:
    core 0 ≡ h/w threads 0,1,2,3 ≡ logical processors 1,2,3,4 etc
  • KMP_AFFINITY=balanced (new for coprocessor, not yet host) uses all cores like “scatter”, but keeps adjacent threads on the same core
  • Other options for MIC_AFFINITY are physical, scatter, and compact
Stack Sizes for Coprocessor Target

- For the main thread, (thread 0), the default stack limit is 12 MB
  - In offloaded functions, stack is used for local or automatic arrays and compiler temporaries
  - To increase limit, export MIC_STACKSIZE=100M
  - M (or B, K, G), default unit is K (Kbytes)
  - For native apps, use ulimit –s (default units are Kbytes)

- For other threads: default stack size is 4 MB
  - Space is only needed for those local variables or automatic arrays or compiler temporaries for which each thread has a private copy
  - To increase limit, export OMP_STACKSIZE=10M (or as needed)
  - Or use dynamic allocation (may be less efficient)

- Typical error messages if stack limits exceeded:
  - offload error: EventWait failed with error COI_PROCESS_DIED
  - offload error: process on the device 0 was terminated by SEGFAULT
Floating-Point Behavior on Intel® Xeon Phi™ Coprocessors

Trapping of floating-point exceptions in vector instructions is not supported

The bits of the SIMD floating-point control word that mask/unmask floating-point protections are protected

- If you try to unmask exceptions, your app will seg fault
- Unmasking by compiler switches such as –fp-trap or –fpe0 is disabled for native builds or for target part of an offload build
- The exception flags still get set, and you can test on these
- Otherwise, the computation just continues with QNaNs, infinities, etc
- -fp-model except or –fp-model strict preserves exception semantics
  - Generates x87 instead of vector instructions, big performance impact
  - May be useful for debugging

Denormals are supported

- Needs –no-ftz or –fp-model precise (like on host)

Refer to following link for differences in FP between CPUs and Phi:
Floating-Point Behavior on Intel® Xeon Phi™ Coprocessors

-fp-model fast=2 enables some more aggressive optimizations
• Faster inlined versions of some math functions
  – May not give standard behavior for extreme or exceptional arguments

Floating-point results on Intel® Xeon Phi™ may not be bit-for-bit identical to results obtained on Intel® Xeon processors
• Most common cause is fused multiply-add (FMA) instructions
  – Not disabled by -fp-model precise
  – Can disable for testing with -no-fma
    – With some impact on performance
• Implementation of math functions might also differ
• To get close, try -fp-model precise -no-fma -fimf-precision=high
  – But most parallel reductions will still cause differences
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**
Preserve Your Development Investment
Common Tools and Programming Models for Parallelism

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**


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

• Upcoming 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