Outline

Motivation for Intel® Cilk™ Plus

SIMD notations

Fork-Join notations

Karatsuba multiplication example

GCC branch
Multi-Threading and Vectorization are Essential to Performance

Latest Intel® Xeon® chip:

- 8 cores
  - 2 independent threads per core
  - 8-lane (single prec.) vector unit per thread

= 128-fold potential for single socket

Intel® Many Integrated Core Architecture

- >50 cores (KNC)
  - ? independent threads per core
  - 16-lane (single prec.) vector unit per thread

= parallel heaven
Importance of Abstraction

Software outlives hardware.

Recompiling is easier than rewriting.

Coding too closely to hardware du jour makes moving to new hardware expensive.

C++ philosophy: abstraction with minimal penalty

Do not expect compiler to be clever. But let it do tedious bookkeeping.
“Three Layer Cake” Abstraction

**Message Passing**
exploit multiple nodes

**Fork-Join**
exploit multiple cores
exploit parallelism at multiple algorithmic levels

**SIMD**
exploit vector hardware
Composition

Message Driven
compose via send/receive

Fork-Join
compose via call/return

SIMD
compose sequentially
Implementing the Cake

Message Driven
MPI, tbb::flow

Fork-Join
OpenMP, TBB, or Cilk

SIMD
Array Notation or #pragma SIMD

Intel® Cilk™ Plus
Intel® Cilk™ Plus

Introduced in 2010 by Intel 12.0 compiler

Open specification

Covers both fork-join and SIMD

Fork-Join
- MIT Cilk
- Cilk Arts Cilk++

SIMD
- APL
- F90 Array Expr.
- Intel® Cilk™ Plus
- OpenMP parallel for
Background

Vectorization of C/C++ is hard...

```c
void saxpy( float a, float x[], float y[], size_t n ) {
    for( size_t i=0; i<n; ++i )
        y[i] += a*x[i];
}
```

... because aliasing is allowed!

```c
// Set z to frequencies of piano keys
float z[88];
std::fill_n( z, 88, 0 );
z[0] = 27.5;
saxpy( 1.059463f, z, z+1, 87 );
```
SSE intrinsics freeze vector length.

Painful to write.
Array Notation

Let programmer specify parallel intent

- Give license to the compiler to vectorize

// Set $y[i] \leftarrow y[i] + a \cdot x[i]$ for $i \in [0..n)$

```c
void saxpy(float a, float x[], float y[], size_t n ) {
    y[0:n] += a*x[0:n];
}
```
Array Section Notation

Pointer or array.

\[ \text{base[first:length:stride]} \]

First index

Optional stride

Number of elements (different than F90!)

Rules for \( \text{section}_1 \ op \ \text{section}_2 \)

- Elementwise application of \( op \)
- Also works for \( \text{func(section}_1, \text{section}_2) \)
- Sections must be the same length
- Scalar arguments implicitly extended
More Examples

• Rank 2 Example – Update $m \times n$ tile with corner $[i][j]$.

  \[ Vx[i:m][j:n] += a \cdot (U[i:m][j+1:n]-U[i:m][j:n]); \]

  Scalar implicitly extended

• Function call

  \[ \text{theta}[0:n] = \text{atan2}(y[0:n],1.0); \]

• Gather/scatter

  \[ w[0:n] = x[i[0:n]]; \]
  \[ y[i[0:n]] = z[0:n]; \]
Improvement on Fortran 90

Compiler does *not* generate temporary arrays.

- Would cause unpredictable space demands and performance issues.
- Want abstraction with minimal penalty.
- Partial overlap of left and right sides is undefined.

Exact overlap still allowed for updates.

- Just like for structures in C/C++.

```
x[0:n] = 2*x[1:n];        // Undefined – partial overlap*
x[0:n] = 2*x[0:n];        // Okay – exact overlap
x[0:n:2] = 2*x[1:n:2];     // Okay – interleaved
```

*unless n≤1.
Reductions

Build-in reduction operation for common cases +, *, min, index of min, etc.

User-defined reductions allowed too.

float dot( float x[], float y[], size_t n ) {
    return __sec_reduce_add( x[0:n]*y[0:n] );
}

sum reduction

elementwise multiplication
#pragma simd

Another way to grant permission to vectorize.

• Programmer responsible for correct use.
• Ignorable by compilers that do not understand it.
• Similar in style to OpenMP “#pragma parallel for”

```c
void saxpy( float a, float x[], float y[], size_t n ) {
    #pragma simd
    for( size_t i=0; i<n; ++i )
        y[i] += a*x[i];
}
```
Clauses for Trickier Cases

linear clause for induction variables

reduction clause for reduction variables

private, firstprivate, lastprivate à la OpenMP

```c
float dot( float *x, float *y, size_t n ) {
    float sum = 0;
    #pragma simd linear(x,y), reduction(+:sum)
    for( size_t i=0; i<n; ++i )
        sum += (*x++) * (*y++);
    return sum;
}
```
Conditionals/Masking

Array sections can be used to control “if”:

```c
if( a[0:n] < b[0:n] )
    c[0:n] += 1;
```

- Can use `#pragma simd` on loops with conditionals

```c
#pragma simd
for( int i=0; i<n; ++i )
    if( a[i]<b[i] )
        c[i] += 1;
```
Elemental Functions

Enables vectorization of separately compiled scalar callee.

In file with definition.

```cpp
__declspec(vector)
float add(float x, float y) {
    return x + y;
}
```

In file with call site.

```cpp
__declspec(vector) float add(float x, float y);

void saxpy( float a, float x[], float y[], size_t n ) {
    #pragma simd
    for( size_t i=0; i<n; ++i )
        y[i] = add(y[i], a*x[i]);
}
```
Final Comment on Array Notation and #pragma SIMD.

No magic – just does tedious bookkeeping.

Use “structure of array” (SoA) instead of “array of structure” (AoS) to get SIMD benefit.
Cilk for Fork-Join Parallelism

Cilk is result of ~15 years of MIT research
• Extension to C/C++

Three keywords
• cilk_for
• cilk_spawn
• cilk_join

Reducers for reductions.
Why Yet Another Idiom for Thread Parallelism?

Simple for non-experts

Predictable performance and space demands

Automatic load balancing

Efficient nested parallelism
Parallel Loops

```cilk
  cilk_for( size_t i=1; i<n; ++i )
    DoBorder(i);
  cilk_for( size_t i=0; i<n; ++i )
    DoInterior(i);
```

Rules for loop test and increment are similar to OpenMP. Implementation is radically different.
Fork-Join

cilk_spawn makes a call asynchronous
• Caller does not have to wait for callee to return.
• But only if idle processor is available.

cilk_sync waits for spawned calls to complete.

cilk_spawn f();
cilk_spawn g();
h();
cilk_sync;
...

...
**Argument/Result for Spawned Function**

`cilk_spawn` forks *after* evaluating arguments.

```
x = cilk_spawn f(++i);
y = g(++i);
cilk_sync;
z = x+y;
```

```
t = ++i  
x = f(t)  
z = x+y
```

```
y = g(++i)
```
cilk_sync Has Function Scope
Scope of cilk_sync is entire function.

End of function has implicit cilk_sync.

```c
void bar() {
    for( int i=0; i<3; ++i ) {
        cilk_spawn f(i);
        if( i&1 ) cilk_sync;
    }
    // implicit cilk_sync
}
```

Serial call/return property
All Cilk Plus parallelism created in function completes before it returns.
Reducers

Race-free shared global variables without locks!
• Work for any associative reduction operation.
• Work for fork-join, not just loops.

cilk::reducer_string r;

void f( const string& s ) {
   r += s;
}

cilk_spawn f("ABC");
f("DEF");
cilk_sync;
cout << r.get_value() << endl;
cilk::reducer_string r;

void f( const string& s ) {
    r += s;
}
cilk_spawn f("ABC");
f("DEF");
cilk_sync;
cout << r.get_value() << endl;

r1+="ABC";
r1+=r2;

r2="";
r2+="DEF";

r1+="ABC";
r1+="DEF";
Polynomial Multiplication

Example: \( c = a \cdot b \)

\[
\begin{array}{ccc}
  x^2 & + & 2x & + & 3 \\
  x^2 & + & 4x & + & 5 \\
  \hline
  5x^2 & + & 10x & + & 15 \\
  4x^3 & + & 8x^2 & + & 12x \\
  x^4 & + & 2x^3 & + & 3x^2 \\
  \hline
  x^4 & + & 6x^3 & + & 16x^2 & + & 22x & + & 15
\end{array}
\]

\( b \)

\( a \)

\( c \)
Storage Scheme for Coefficients

---  ---  ---  
a[2]  a[1]  a[0]  
---  ---  ---  
---  ---  ---  ---  ---  

b  a  c
Karatsuba Trick: Divide and Conquer

Suppose polynomials \( a \) and \( b \) have degree \( n \)

- let \( K = x^\lfloor n/2 \rfloor \)
  \[
  a = a_1 K + a_0 \\
  b = b_1 K + b_0
  \]

Compute:
  \[
  t_0 = a_0 \cdot b_0 \\
  t_1 = (a_0 + a_1) \cdot (b_0 + b_1) \\
  t_2 = a_1 \cdot b_1
  \]

Then
  \[
  a \cdot b = t_2 K^2 + (t_1 - t_0 - t_2) K + t_0
  \]

Partition coefficients.

3 half-sized multiplications. Do recursively in parallel.

Sum products, shifted by multiples of \( K \).
Multithreaded Karatsuba in Cilk Plus

```c
void karatsuba( T c[], const T a[], const T b[], size_t n ) {
    if( n<=CutOff ) {
        simple_mul( c, a, b, n );
    } else {
        size_t m = n/2;
        cilkr_spawn karatsuba( c, a, b, m ); // t_0 = a_0 × b_0
        cilkr_spawn karatsuba( c+2*m, a+m, b+m, n-m ); // t_2 = a_1 × b_1
        temp_space<T> s(4*(n-m));
        T *a_=s.data(), *b_=a_+(n-m), *t=b_+(n-m);
        a_[0:m] = a[0:m]+a[m:m]; // a_ = (a_0+a_1)
        b_[0:m] = b[0:m]+b[m:m]; // b_ = (b_0+b_1)
        karatsuba( t, a_, b_, n-m ); // t_1 = (a_0+a_1)×(b_0+b_1)
        cilkr_sync;
        t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1]; // t = t_1-t_0-t_2
        c[2*m-1] = 0;
        c[m:2*m-1] += t[0:2*m-1]; // c = t_2K^2+(t_1-t_0-t_2)K+t_0
    }
}
```
Recap – Three Layer Cake

Message Driven
MPI, tbb::flow

Fork-Join
OpenMP, TBB, or Cilk

Intel® Cilk™ Plus

SIMD
Array Notation or #pragma SIMD
GCC Port

Experimental branch of GCC 4.7

• Implements Cilk Plus specification

• Balaji V. Iyer

• We welcome contributions.
Summary

Intel® Cilk™ Plus abstracts SIMD and thread parallelism.

• SIMD portion works well with OpenMP too.

Parallel programming for non-experts.

• Simple composition rules enable building software from components.

Open specification of language and ABI

• GCC experimental branch.

• Ask your compiler vendor to implement it.
URLs

Cilk Plus Forum

Cilk Plus Specifications

Intel® Cilk™ Plus Software Development Kit
  - Cilk Screen Race Detector
  - Cilk View Scalability Analyzer

GCC 4.7 Branch
• http://gcc.gnu.org/svn/gcc/branches/cilkplus/

“Three Layer Cake for Shared-Memory Programming”
• http://portal.acm.org/citation.cfm?id=1953616
Danke!
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY
ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS
DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR
IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES
RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY
PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on
Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using
specific computer systems, components, software, operations and functions. Any change to any of
those factors may cause the results to vary. You should consult other information and performance
tests to assist you in fully evaluating your contemplated purchases, including the performance of that
product when combined with other products.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk
are trademarks of Intel Corporation in the U.S. and other countries.

**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