Parallel Programming in Intel® Cilk™ Plus with Autotuning

William Leiserson, Chi-Keung (CK) Luk, Peng Tu
Tutorial Outline

- Introduction (15 mins) // CK
- Cilk (1 hr) // Will
- Array Notation-Part 1 (30 mins) // Peng
- Break (15 mins)
- Array Notation-Part 2 (30 mins) // Peng
- Autotuning (20 mins) // CK
- Put it All Together (40 mins) // CK
A New Era...

**THE OLD**

- Frequency Focus
- Unconstrained Power
- Voltage Scaling

**THE NEW**

- IPC Focus
- Multi-Core
- Power Efficiency
- Microarchitecture Advancements
Use Transistors, Not Cycle-time for Performance

Parallel performance: $P \sim \text{Area}$

Parallel technologies fill gap:
- Multi-core
- Threading
- SIMD

Sequential Performance: $P \sim \sqrt{\text{Area}}$
Multi-core: Multiple Cores on One Die

Cores
- Performance scales with number of cores

On-die Interconnect
- High bandwidth
- Low latency

Cache hierarchy
- Private and shared
- Inclusive and exclusive

Scalability
Intel® Hyper-Threading Technology: Fine-Grained Interleaving of Threads

Instruction Issue Slots

Two threads compete for issue slots

Intel® HTT in Nehalem.
Run 2 threads at the same time in a core.
Exploit 4-wide execution engine.

Hyperthreading is a power efficient performance feature.
SIMD Instructions Compute Multiple Operations per Instruction

Intel® Advanced Vector Extensions in Sandy Bridge
Intel® Sandy Bridge Processor:

- 4 IA cores + 1 GPU per die
- 2 256b SIMD pipelines per IA core
- 2 Intel® Hyper-Threading Technology threads per IA core
1st MIC implementation (KNF) has 32 cores, each with 4 threads, and a 512-bit SIMD unit

Many cores and many, many more threads

Standard IA programming and memory model
Intel Parallel Programming Model

Multiple cores
Hardware threads

Tasks
- Intel® Cilk
- Intel® Threading Building Blocks

SIMD instructions

Vectors
Array Notation
Intel® Array Building Block

Parallel tasks with SIMD kernels
Task Parallel Programming

Tasks are pieces of work at the application level
- Loop bodies, functions

Scalable. Specify potential parallelism. Scheduler maps tasks to threads

Task graph dynamically created and executed

Supported in Intel Cilk and TBB

Scalable parallelism at the application level
Intel® Cilk™ Plus:
One of the Intel® Parallel Building Blocks

**What is it?**
Language extensions to simplify task and vector parallelism

**Features**
- 3 Simple keywords and array notations for parallelism
- Support for Task and Vector parallelism
- Similar semantics as serial code
- Simple way to parallelize your code
- Sequentially consistent + low overhead = powerful solution
- Supports C & C++; Windows* and Linux*

**Reasons to Use**

**Intel® Cilk™ Plus**

**Intel® Threading Building Blocks**
Widely used C++ template library for task parallelism
- Parallel Algorithms and Data Structures
- Scalable Memory Allocation and Task Scheduling
- Synchronization Primitives
- Rich feature set for general purpose parallelism
- Available as open source or commercial
- Supports C++; Windows, Linux, Mac OS*, other OS's

**Intel® Array Building Blocks**
Sophisticated C++ template library for vector parallelism
- Automatically scales to future Intel platforms
- Use of cores, threads, SIMD determined by run time compiler
- Used for flexible vector parallelism
- JIT & VM technology = flexible and powerful
- Supports C++; Windows & Linux

**MIX AND MATCH TO OPTIMIZE YOUR APPLICATION’S PERFORMANCE**
Intel® Cilk™ Plus

- Cilk Plus = Cilk + Array Notation
- A new feature in the Intel Compiler version 12
- A simple and effective approach to exploiting both task and vector parallelism
- Work for both Intel mainstream processors and Many-Integrated-Core (MIC) accelerator
Outline

1. Cilk Syntax and Concepts
2. Scalability Analysis
3. Race Detection and Correction
Outline

1. Cilk Syntax and Concepts
2. Scalability Analysis
3. Race Detection and Correction
Cilk Plus

- Small set of *linguistic* extensions to C/C++ to support fork-join parallelism.
- Based on Cilk, developed at MIT, and Cilk++, developed at Cilk Arts.
- Uses a provably efficient work-stealing scheduler.
- Offers *hyperobjects* as a lock-free mechanism to resolve race conditions.
- The Cilkscreen race detector and Cilkview scalability analyzer are available as a free download from the Intel website.
Fibonacci Numbers

The Fibonacci numbers are the sequence \( \langle 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \rangle \), where each number is the sum of the previous two.

Recurrence:

\[
F_0 = 0,
F_1 = 1,
F_n = F_{n-1} + F_{n-2} \text{ for } n > 1.
\]

The sequence is named after Leonardo di Pisa (1170-1250 A.D.), also known as Fibonacci, a contraction of *filius Bonaccii* — “son of Bonaccio.” Fibonacci’s 1202 book *Liber Abaci* introduced the sequence to Western mathematics, although it had previously been discovered by Indian mathematicians.
Fibonacci Program

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

int fib(int n)
{
    if (n < 2) return n;
    int x = fib(n-1);
    int y = fib(n-2);
    return x + y;
}

int main(int argc, char *argv[])
{
    int n = atoi(argv[1]);
    int result = fib(n);
    printf("Fibonacci of %d is %d.\n", n, result);
    return 0;
}
```

Disclaimer
This recursive program is a poor way to compute the nth Fibonacci number, but it provides a good didactic example.
Fibonacci Execution

Key idea for parallelization
The calculations of \( \text{fib}(n-1) \) and \( \text{fib}(n-2) \) can be executed simultaneously without mutual interference.

```c
int fib(int n)
{
    if (n < 2) return n;
    int x = fib(n-1);
    int y = fib(n-2);
    return x + y;
}
```
int fib(int n) {
    if (n < 2) return n;
    int x = fib(n-1);
    int y = fib(n-2);
    return x+y;
}
int fib(int n)
{
    if (n < 2) return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x+y;
}

The named **child** function may execute in parallel with the **parent** caller.

Control cannot pass this point until all spawned children have returned.

Cilk keywords **grant permission** for parallel execution. They do not **command** parallel execution.
Work-stealing task scheduler

Each processor has a work queue of spawned tasks

When each processor has work to do, a spawn is roughly the cost of a function call.
Work-stealing task scheduler

Each processor has a work queue of spawned tasks

When each processor has work to do, a spawn is roughly the cost of a function call.
Work-stealing task scheduler

Each processor has a work queue of spawned tasks

When each processor has work to do, a spawn is roughly the cost of a function call.
Work-stealing task scheduler

Each processor has a work queue of spawned tasks

When each processor has work to do, a spawn is roughly the cost of a function call.
Work-stealing task scheduler

Each processor has a work queue of spawned tasks

When a processor has no work, it steals from another processor.
Work-stealing task scheduler

Each processor has a work queue of spawned tasks

With sufficient parallelism, the steals are rare, and we get *linear speedup* (ignoring memory effects)
Parallelizing Loops

```
for (int i = 0; i < 8; ++i)
{
    do_work(i);
}
```

```
for (int i = 0; i < 8; ++i)
{
    cilk_spawn do_work(i);
}
cilk_sync;
```

- A lot of overhead: spawning is cheap, **stealing** is expensive.
- Low parallelism: There is only one producer of parallel work at any time.
If work per iteration is small then there is no parallelism.
Worker 0

Worker 1

Alternative: Divide and Conquer

Divide and conquer results in fewer steals and more parallelism.
for (int i = 0; i < 8; ++i)
{
    do_work(i);
}

cilk_for (int i = 0; i < 8; ++i)
{
    do_work(i);
}

cilk_for performs divide-and-conquer over the iterations.
Outline

1. Cilk Syntax and Concepts
2. Scalability Analysis
3. Race Detection and Correction
template <typename T>
void qsort(T begin, T end) {
    if (begin != end) {
        T middle = partition(
            begin,
            end,
            bind2nd( less<typename
                iterator_traits<T>::value_type>(),
                    *begin )
        );
        cilk_spawn qsort(begin, middle);
        qsort(max(begin + 1, middle), end);
        cilk_sync;
    }
}
Quicksort Performance

How do we extrapolate scalability?

Like this?
Quick sort Performance

Or this?
Quicksort Performance

![Quicksort Performance Graph]

Maybe this?
Quicksort Performance

![Graph showing speedup vs. number of processors.]

Possibly this?
Quicksort Performance

Limiting Factors

- Insufficient parallelism
- Scheduling overhead
- Memory bandwidth
- Contention (locking and true/false sharing)

Cilkview extrapolates limits on scalability due to insufficient parallelism and scheduling overhead.
Cilkview Output for Quicksort

![Graph showing Cilkview Output for Quicksort]

- **Application Parallelism**
- **Burdened Parallelism**
- **Linear Speedup**
- **Measured Speedup**

The graph shows the comparison between measured speedup and the ideal speedup, indicating linear growth as the number of cores increases. The measured speedup is shown as a blue triangle line, while the ideal speedup is represented by a green line. The application parallelism is depicted by a black line, and the burdened parallelism is shown by a darker line.

The measurement indicates a speedup of 21.31 when all cores are utilized.
How Cilkview Works

• Cilkview reads from *metadata* embedded by the Cilk Plus compiler to perform its calculations.
• Cilkview generates rough (but repeatable) performance measures by counting instructions rather than reading from a clock.
• Despite the coarseness of measurements, Cilkview accurately estimates scalability.
A *strand* is a chain of serially executed instructions with no parallel control.

Strands are partially ordered by control *dependencies*.

A *spawn node* has two successors.

Control joins at a *sync node*. 
**Work/Span Analysis**

$T_P = \text{time to execute on } P \text{ processors.}$

Assuming unit-time strands:

- $T_1 = 18$
- $T_\infty = 9$

The **work** $T_1$ is the time to execute the program on 1 processor of an “ideal” parallel computer.
  - $T_1 = \text{total instructions in all nodes.}$

The **span** $T_\infty$ is the time to execute the program on an infinite number of processors, discounting scheduling.
  - $T_\infty = \text{longest path.}$
Parallelism

- Speedup = $T_1 / T_P$.
- Work Law: $T_P \geq T_1 / P$
  \[ \Rightarrow T_1 / T_P \leq P \] (no superlinear speedup).
- Span Law: $T_P \geq T_\infty$.

- Parallelism: $T_1 / T_\infty$
  \[ \Rightarrow T_1 / T_P \leq T_1 / T_\infty \] (parallelism bounds speedup).

$T_1 = 18$
$T_\infty = 9$
$T_1 / T_\infty = 2$
Parallel Slackness

If $T_1 / T_\infty \gg P$, then $T_1 / T_p \approx P$ (sufficient parallelism implies linear speedup).

We call this property \textit{parallel slackness}.
Both snippets have the same parallelism, but **Snippet A** significantly outperforms **Snippet B**.
Measured Performance of Snippets

![Graph showing the measured performance of Snippets A and B across different numbers of processors. The x-axis represents the number of processors (0 to 8), and the y-axis represents the speedup. Snippet A shows a consistent speedup of 4 across all processor counts, while Snippet B shows a speedup of 1 at the single-processor level, increasing to 2, 3, 4, 5, 6, 7, and 8, respectively, as the number of processors increases.]
Scheduling Overhead

// Snippet A
cilk_for (int i = 0; i < 4; ++i)
    for (int j = 0; j < 1000000; ++j)
        do_work();

// Snippet B
for (int j = 0; j < 1000000; ++j)
    cilk_for (int i = 0; i < 4; ++i)
        do_work();

**Snippet A**: Schedule 4 units of a million, once.

**Snippet B**: Schedule 4 units of one, a million times!
**The Burdened DAG**

*Burden* the continuation and return edges to reflect overheads due to scheduling.

*Burdened span* $= \hat{T}_\infty$, time on an infinite number of processors, taking burdens into account.
Series Composition

**Work:** \[ T_1(A \cup B) = T_1(A) + T_1(B) \]

**Span:** \[ T_\infty(A \cup B) = T_\infty(A) + T_\infty(B) \]
Parallel Composition

Work: \( T_1(A \cup B) = T_1(A) + T_1(B) \)

Span: \( T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\} \)
Cilkview output for Snippets

Graphs showing speedup vs. number of cores for Snippet_A and Snippet_B. The graphs illustrate measured speedup, lower performance bound, upper performance bound, and ideal speedup for each snippet.


Cilkview Output for Quicksort

\[ \frac{T_1}{T_0} \]

\[ P \]

\[ \frac{T_1}{P + kT_0} \]
Outline

1. Cilk Syntax and Concepts
2. Scalability Analysis
3. Race Detection and Correction
Race Bugs

Definition: A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

```c
int x = 0;
cilk_for (int i=0, i<2, ++i)
{
    x++;
}
assert(x == 2);
```
Types of Races

Suppose that instruction A and instruction B both access a location x, and suppose that $A \parallel B$ (A is parallel to B).

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Race Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
<td>none</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>read race</td>
</tr>
<tr>
<td>write</td>
<td>read</td>
<td>read race</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>write race</td>
</tr>
</tbody>
</table>

Two sections of code are *independent* if they have no determinacy races between them.
Non-Local Variables

1973 — *Historical perspective*

*Wulf & Shaw:* “We claim that the non-local variable is a major contributing factor in programs which are difficult to understand.”

2011 — *Today’s reality*

Non-local variables are used extensively, in part because they avoid *parameter proliferation* — long argument lists to functions for passing numerous, frequently used variables.

Global and other nonlocal variables can inhibit parallelism by inducing race bugs.
void increment(int& i) {
    ++i;
}

int main () {
    int x = 0;
    cilk_spawn increment(x);
    int y = x - 1;
    return y;
}
Cilkscreen Race Detector

- If a deterministic Cilk Plus program run on a given input could possibly behave any differently than its serialization, the race detector will report and localize at least two accesses participating in the race.
- Employs a regression-test methodology, where the programmer provides test inputs.
- Identifies filenames, lines, and variables involved in races, including stack traces.
- Runs off the binary executable using dynamic instrumentation.
Cilk Plus and Cilkscreen

• In *theory*, a race detector could be constructed to find races in any multithreaded program *but...*

• Cilk Plus allows race detection with *bounded memory and time overhead*, independent of the number of threads.
Avoiding Races

- Iterations of a `cilk_for` should be independent.
- Between a `cilk_spawn` and the corresponding `cilk_sync`, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children. *Note:* The arguments to a spawned function are evaluated in the parent before the spawn occurs.
- Machine word size matters. Watch out for races in packed data structures:

```c
struct{
    char a;
    char b;
} x;
```

Updating `x.a` and `x.b` in parallel may cause a race! Nasty, because it may depend on the compiler optimization level. (Safe on x86 and x86_64.)
Reducer Hyperobjects

- A variable \( x \) can be declared as a \textit{reducer} over an \textit{associative} operation, such as addition, multiplication, logical AND, list concatenation, etc.
- Strands can update \( x \) as if it were an ordinary nonlocal variable, but \( x \) is, in fact, maintained as a collection of different \textit{views}.
  - The Cilk Plus runtime system coordinates the views and combines them when appropriate.
  - When only one view of \( x \) remains, the underlying value is stable and can be extracted.

Example: summing reducer

\[
\begin{align*}
\text{x: 42} \\
\text{x: 14} \\
\text{x: 33}
\end{align*}
\]

\[
89
\]
Using Reducer Hyperobjects

```c++
#include <cilk/cilk.h>
#include <cilk/reducer_min.h>

template <typename T>
size_t IndexOfMin(T array[], size_t n) {
    cilk::reducer_min_index<size_t, T> r;

    cilk_for (int i = 0; i < n; ++i) 
        r.min_of(i, array[i]);

    return r.get_index();
}
```
Data Race Take-aways

- A data race occurs when two parallel strands access the same memory location and at least one performs a write.
- Access need not be simultaneous for the race to be harmful.
- Cilkscreen will find data races if they occur in a program execution.
- Non-local variables are a common source of data races.
- Cilk Plus hyperobjects can be used to eliminate many data races.
A Closer Look

```
int x = 0;

A

x++; x++;

B

assert(x == 2);

C

x = 0;

D

r1 = x;

2

r1++; r1++;

3

x = r1;

4

r2 = x;

5

r2++; r2++;

6

x = r2;

7

assert(x == 2);

8

1

x

1

r1

1

r2
```
Array Notation and Elemental Function

Peng Tu
Why Data Parallel Extension to C/C++

• Language must let the programmer get to hardware performance easily.
• Array operations provide a natural mapping between data parallel algorithms and parallel execution hardware.
• C/C++ is the dominating language for performance sensitive sequential applications because it’s constructs map easily to scalar ISA.
• Array extensions to C/C++ aim at achieving predictable parallelization for application on parallel hardware:
  • Express data parallelism that is syntactically and semantically consistent with the familiar language.
  • Simple compiler techniques map array constructs to underlying multi-threaded/SIMD hardware for predictable performance.
  • Work seamlessly with existing C/C++ parallel runtime systems such as Cilk.
Overview of Data Parallel Extensions to C/C++

• Array notations to express parallelism
  – \([<\text{lower bound}> : <\text{length}> [: <\text{stride}>]]\)
  – Compiler vectorizer maps to targeted CPU hardware

• Array reductions
  – __sec_reduce_<type>
  – Merge all array elements into a scalar result

• Elemental Functions
  – __declspec(vector) <function signature>
  – Maps scalar function across multiple array elements

• All this supported on C/C++ and Windows*, Linux* and Mac OS X*
Array Section Notation

• Definition:
  `<array base> [<lower> : <length> [: <stride>]]`
  `[<lower> : <length> [: <stride>]]...`

• We use a “:” in an array subscript to indicate multiple array elements (a section of an array)
• “:” can be used by itself to specify all array elements
• Note: the second operand is length, not upper bound

**Examples**

A[:]
    // All elements of vector A
B[2:6]
    // Elements 2 to 7 of vector B
C[:][5]
    // Column 5 of matrix C
D[0:3:2]
    // Elements 0,2,4 of vector D
E[0:3][0:4]
    // 12 elements from E[0][0] to E[2][3]
Operator Maps

- Most C/C++ arithmetic & logic operators can be performed on array sections
  
  +, -, *, /, %, <, ==, !=, >, |, &, ^, &&, ||, !,(unary), +(unary), ++, --, +=, -=, *=, /=, *(p)

- Operators are mapped to all elements of the array section operands
  
  a[:,] * b[:,] // element-wise multiplication

- Operations on different elements can be executed in parallel without any ordering constraints

- Array section operands must have the same rank & extent
  
  a[0:4][1:2] + b[1:2] // error, different ranks

- Scalar operand automatically expanded to fill the whole section
  
  a[:,][:] + b[0][1] // adds b[0][1] to all elements of a
Assignment Map

• Assignment operator applies in parallel to every element of the array section on LHS

\[ a[0:n] = b[0:n] + 1; \]

• RHS must be same rank as LHS, or scalar

\[ a[:,:] = c; \quad \text{ // } c \text{ fills the array } a \]
\[ e[:,:] = b[:][:]; \quad \text{ // } \text{error, different rank} \]

• Overlap between RHS and LHS?
  – RHS is evaluated before any element on LHS is stored
  – Compiler will insert temporary arrays, if needed

\[ a[1:s] = a[0:s] + 1; \quad \text{ // use old value of } a[1:s-1] \]
#include <iostream>

int main() {
    double a[4] = {1.,2.,3.,4.};
    double b[4] = {5.,7.,11.,13.};
    double c[4] = {0.,0.,0.,0.};

    std::cout << "Display a:\n" << a[:] << " ";
    std::cout << std::endl << std::endl;
    std::cout << "Display b:\n" << b[:] << " ";
    std::cout << std::endl << std::endl;
    std::cout << "Display c:\n" << c[:] << " ";
    std::cout << std::endl << std::endl;

    c[:] = a[:] + b[:];

    std::cout << "c = a + b:\n" << c[:] << " ";
    std::cout << std::endl << std::endl;  

    $ icc test-add.cpp
    $ ./a.out

    Display a:
    1 2 3 4

    Display b:
    5 7 11 13

    Display c:
    0 0 0 0

    c = a + b:
    6 9 14 17
Using Conditional Select

```cpp
$ cat test-sel.cpp
#include <iostream>

int main() {
    bool x[4] = {0, 0, 1, 1};
    bool y[4] = {0, 1, 1, 0};
    double a[4] = {1.,2.,3.,4.};
    double b[4] = {5.,7.,11.,13.};
    double c[4] = {0.,0.,0.,0.};
    std::cout << "Display a:\n" << a[:] << " ";
    std::cout << std::endl << std::endl;
    std::cout << "Display b:\n" << b[:] << " ";
    std::cout << std::endl << std::endl;
    std::cout << "Display c:\n" << c[:] << " ";
    std::cout << std::endl << std::endl;
    c[:] = x[:] && y[:] ? a[:] : b[:];
}

$ ./a.out
Display a:
1 2 3 4
Display b:
5 7 11 13
Display c:
0 0 0 0
Display updated c:
5 7 3 13
```
Compare to Implementation in Intrinsics

```cpp
$ cat test-intrinsic.cpp
#include <pmmintrin.h>

void foo(float* dest, short* src, long len, float a) {
    __m128 xmmMul = _mm_set1_ps(a);
    for(long i = 0; i < len; i+=8) {
        __m128i xmmSrc1i = _mm_loadl_epi64((__m128i*) &src[i]);
        __m128i xmmSrc2i = _mm_loadl_epi64((__m128i*) &src[i+4]);

        xmmSrc1i = _mm_cvtepi16_epi32(xmmSrc1i);
        xmmSrc2i = _mm_cvtepi16_epi32(xmmSrc2i);

        __m128 xmmSrc1f = _mm_cvtepi32_ps(xmmSrc1i);
        __m128 xmmSrc2f = _mm_cvtepi32_ps(xmmSrc2i);

        xmmSrc1f = _mm_mul_ps(xmmSrc1f, xmmMul);
        xmmSrc2f = _mm_mul_ps(xmmSrc2f, xmmMul);

        _mm_store_ps(&dest[i], xmmSrc1f);
        _mm_store_ps(&dest[i+4], xmmSrc2f);
    }
}
```
Implementation in Array Notation

$ cat test-cilkplus.cpp

void foo(float *restrict dest, short *restrict src, long len, float a) {
    __assume_aligned(dest, 16);
    __assume_aligned(src, 16);

dest[0:len] = ((float) src[0:len]) * a;
}

• The restrict qualifier and alignment assertions are added to avoid checks for aliasing and alignment.
Comparable Quality

**Intrinsics**

..B1.3: # Preds ..B1.1 ..B1.3

```
movq (%rsi,%rax,2), %xmm1  #9.18
movq 8(%rsi,%rax,2), %xmm2  #10.18
pmovsxwd %xmm1, %xmm1  #9.18
pmovsxwd %xmm2, %xmm2  #10.18
cvtdq2ps %xmm1, %xmm3  #12.25
cvtdq2ps %xmm2, %xmm4  #13.25
mulps %xmm0, %xmm3  #15.18
mulps %xmm0, %xmm4  #16.18
movaps %xmm3, (%rdi,%rax,4)  #18.21
movaps %xmm4, 16(%rdi,%rax,4)  #19.21
addq $8, %rax  #5.34
cmpq %rdx, %rax  #5.24
jl ..B1.3  Prob 82%  #5.24
```

**Array Notations**

..B1.15: # Preds ..B1.15 ..B1.14

```
movq (%rsi,%rcx,2), %xmm2  #2.45
pmovsxwd %xmm2, %xmm2  #2.45
cvtdq2ps %xmm2, %xmm3  #2.45
mulps %xmm1, %xmm3  #2.58
movaps %xmm3, (%rdi,%rcx,4)  #2.15
movq 8(%rsi,%rcx,2), %xmm4  #2.45
pmovsxwd %xmm4, %xmm4  #2.45
cvtdq2ps %xmm4, %xmm5  #2.45
mulps %xmm1, %xmm5  #2.58
movaps %xmm5, 16(%rdi,%rcx,4)  #2.15
addq $8, %rcx  #2.15
cmpq %r8, %rcx  #2.15
jb ..B1.15  Prob 44%  #2.15
```

•Portable vector program with similar performance as intrinsics.
Specify Array Shape

• Compiler must know the shape and size of an array to identify multidimensional array sections.

• C/C++ has 3 ways to specify the shape and size of arrays

  • Fixed length array and array parameter
    - float a[100][50]; void foo(float b[50][100]);

  • Variable length local array or parameter (C99 VLA)
    - int bar(int m, int n) { float a[m][n]; ... }
    - int fred(int m, int n, float b[m][n]);

  • Pointer to heap memory array
    - float (*p2d)[100], (*q2d)[m]
C99 Multidimensional Array Parameters

• User can pass a pointer to array to a function with VLA parameter, and section it inside the function body.

• Compiler uses the C99 VLA declaration to find out the shape and extent of array sections.

```c
int add3(int rows, int cols, int B[rows][cols]) {
    B[:][:] += 3; // add 3 to each element of
    // B[0:rows][0:cols]
}
```
Heap Allocated Arrays

• Specify dynamic multi-dimensional arrays in C/C++:

```c
typedef int (*a2d)[10]; // pointer to int vector of size 10
a2d *p;

p = (a2d) malloc(sizeof(int)*rows*10);
p[4][::] = 42;  // set all of row 4 to 42
p[0:rows][::] = 42;  // set the whole array to 42
p[::][::] = 42;  // error
```

• User needs to explicitly include the row size for heap allocated arrays.
Reductions

- Reduction combines array section elements to generate a scalar result
  
  ```c
  int a[] = {1,2,3,4};
  sum = __sec_reduce_add(a[:]); // sum is 10
  ```

- Nine built-in reduction functions supporting basic C data-types:
  - add, mul, max, max_ind, min, min_ind, all_zero, all_non_zero, any_nonzero

- Supports user-defined reduction function
  
  ```c
  type fn(type in1, type in2); // scalar reduction function
  out = __sec_reduce(fn, identity_value, in[x:y:z]);
  ```
Custom Reduction

$ cat test.cpp
#include <iostream>

unsigned int bitwise_and(unsigned int x, unsigned int y) {
    return (x & y);
}

int main() {
    unsigned int a[4] = {5,7,13,15};
    unsigned int b = 0;

    std::cout << "Display a:\n" << a[:] << "\n";
    std::cout << std::endl << std::endl;

    b = __sec_reduce(bitwise_and, 0xffffffff, a[:]);

    std::cout << "b:\n" << b << std::endl;
    return(0);
}

$ ./a.out
Display a:
5 7 13 15 // (i.e. 0101, 0111, 1101, 1111)

b:
5 // (i.e. 0101)
Gather/Scatter Support

• When array section occurs in array subscript, it represents a set of elements indexed by values of the array section:

\[
a[b[0:s]] = c[:] \quad // \quad a[b[0]]=c[0], a[b[1]]=c[1], \ldots
\]

\[
c[0:s] = a[b[:]] \quad // \quad c[0]=a[b[0]], c[1]=a[b[1]], \ldots
\]

• Compiler generates scatter and gather instructions on supported hardware.
Gather/Scatter Example

$ cat gather_scatter.cpp
void foo(float *dest, float *src, unsigned int *ind_dest, unsigned int *ind_src, int len) {
    dest[ind_dest[0:len]] = src[ind_src[0:len]];
}
$ cat main.cpp
#include <iostream>
void foo(float *dest, float *src, unsigned int *ind_dest, unsigned int *ind_src, int len);
int main() {
    float x[5] = {1., 2., 3., 4., 5.};
    float y[5] = {0., 0., 0., 0., 0.};
    unsigned int y_ind[5] = {4,3,2,1,0};
    unsigned int x_ind[5] = {1, 3, 0, 2, 4}; // i.e.
    return {2,4,1,3,5}
}

$ cat main.cpp
#include <iostream>

void foo(float *dest, float *src, unsigned int *ind_dest, unsigned int *ind_src, int len);
int main() {
    float x[5] = {1., 2., 3., 4., 5.};
    float y[5] = {0., 0., 0., 0., 0.};
    unsigned int y_ind[5] = {4,3,2,1,0};
    unsigned int x_ind[5] = {1, 3, 0, 2, 4}; // i.e.
    return {2,4,1,3,5}
}

$ ./a.out
x: 1 2 3 4 5
y: 0 0 0 0 0
y: 5 3 1 4 2 // 2,4,1,3,5 backwards
FIR Filter

Scalar code:

```c
for (i=0; i < M-K; i++) {
    s = 0;
    for (j = 0; j < K; j++) {
        s += x[i+j] * c[j]
    }
    y[i] = s;
}
```

Inner (j) loop vectorized:

```c
for (i=0; i < M-K; i++) {
    y[i] = __sec_reduce_add(x[i:K] * c[0:K]); // perform K multiplications in parallel
    // and add up the products
}
```

Outer (i) loop vectorized:

```c
y[0:M-K] = 0; // calculate all the y[i] results in parallel
for (j = 0; j < K; j++) {
    y[0:M-K] += x[j:M-K] * c[j]
}
```
Parameter Passing

• An array section can be passed to a fixed or variable length array parameter as argument:

  • Programmer directly passes the address of its first element.

  • A section argument must be consecutive in order to match the layout of a formal parameter.

```c
void saxpy_vec(int m, float a, float restrict x[m], float restrict y[m])
{
    y[:] += a * x[:];
}

cilk_for(int i = 0; i < n; i += 256)
    saxpy_vec(256, 2.0, &x[i], &y[i]);
```
Function Maps

A scalar function call is mapped to the elements of array section parameters by default:

\[
\begin{align*}
    a[:] &= \sin(b[:]); \\
    a[:] &= \text{pow}(b[:], c); \quad // \quad b[:]**c \\
    a[:] &= \text{pow}(c, b[:]); \quad // \quad c**b[:] \\
    a[:] &= \text{foo}(b[:]) \quad // \quad \text{user defined function}
\end{align*}
\]

- Functions are mapped in parallel
- No specific order on side effects
- Compiler generates calls to vectorized library functions
- Compiler may generate parallel threads
- Compiler may generate vectorized function body for function declared as ‘elemental’
User Defined Elemental Vector Function

$ cat main.cpp
float saxpy(float a, float *x, float *y);
void foo(float *x, float *y, float a, int len) {
    for(int i = 0; i < len; i++)
        saxpy(a, x[i], y[i]);
}
$ cat saxpy1.cpp
void saxpy(float a, float *x, float *y) {
    *y += a * (*x);
}

$ icc -vec-report3 -c main.cpp
main.cpp(3): (col. 58) remark: routine skipped: no vectorization candidates.
Declare an Elemental Function

$ cat main2.cpp
__declspec(vector(scalar(a),linear(x),linear(y)))
void saxpy(float a, float *x, float *y);
void foo(float *x, float *y, float a, int len) {
    for(int i = 0; i < len; i++)
        saxpy(a, x[i], y[i]);
}
$ cat saxpy2.cpp
__declspec(vector(scalar(a),linear(x),linear(y)))
void saxpy(float a, float *x, float *y) {
    *y += a * (*x);
}
$ icc -vec-report3 -restrict -c main2.cpp
main2.cpp(4): (col. 4) remark: LOOP WAS VECTORIZED.
Invoke Elemental Function with Array Sections

__declspec(vector(scalar(a),linear(x),linear(y)))
void saxpy(float a, float *x, float *y);
void foo(float *restrict x, float *restrict y, float a, int len) {
    saxpy(a, x[0:len], y[0:len]);
}

Compiler effectively generates:

void saxpy_4(float a, float x[4], float y[4])
{
    y[:] += a * x[:];
}
for(i = 0; i < len-3; i += 4) {
    saxpy_4(a, &x[i], &y[i]);
}
for(; i < len; i++) {
    saxpy(a, &x[i], &y[i]);
}
Current Limitations on Elemental Functions

- Only certain parameter types allowed
  - signed/unsigned 8/16/32/64 bit ints
  - 32 or 64 bit floating point
  - 64 or 128 bit complex
  - pointer or C++ reference
- No for, while, do or goto keywords
- No switch statements
- No Class or Struct Operations other than member selection
- No Function calls to non-elemental functions
- No threading, OpenMP*, or cilk_spawn / cilk_for
- No expressions with array notations
- No setjmp/longjmp or exception handling
- No inline assembly
- No virtual functions or function pointer calls
- Any of these will result in a syntax error
Array Notation Summary

- Parallel array operations
- Parallel elemental function map
- Parallel reduction across array section
- Make vectorization obvious and straightforward to the compiler
- Work with existing C/C++ array data types

\[ a[::] + b[::] \]

Function mapping

\[ f(b[::]) \]

__sec_reduce(f, a[::]);

Element-wise vector operations

Reductions
Intel® Software Autotuning Tool (ISAT)

Chi-Keung (CK) Luk
What is Autotuning?

• Definition in Software Community (from Berkeley ParLab):

  – Autotuners optimize the performance of a set of library kernels by generating many variants of a given kernel and benchmarking each variant by running on the target platform. The search process effectively tries many or all optimizations and hence may take hours to complete on the target platform.
Autotuning Usage Models

Our focus

Applications
(Consumer HPC, video, graphics)

Autotuning

Libraries
(e.g., MKL, IPP, ATLAS, FFTW)
Intel Software Autotuning Tool (ISAT)

• Supports automatic searching of the near-optimal values of important program parameters
• Built-in support for tuning parameters in commonly used APIs:
  – OpenMP
  – TBB
• Provides supplementary tools for:
  – Visualizing tuning results
  – Identifying regions that need tuning
• Easy to use:
  – No compiler dependence

Freely available at:
Example: Tuning General Program Parameters

- Tuning the block size in blocked matrix multiply

```c
void BlockedMatrixMultiply(float* A, float* B, float* C, int n) {
    #pragma isat tuning variable(blk_i, range(32, 256, 16))
    variable(blk_j, range(32, 256, 16)) variable(blk_k, range(32, 256, 16))
    const int blk_i = 64;
    const int blk_j = 64;
    const int blk_k = 64;
    for (int i=0; i<n; i+=blk_i)
        for (int j=0; j<n; j += blk_j)
            for (int k=0; k<n; k += blk_k)
                for (int ii=i; ii<MIN(i+blk_i, n); ii++)
                    for (int jj=j; jj<MIN(j+blk_j, n); jj++)
                        for (int kk=k; kk<MIN(k+blk_k, n); kk++)
                        C[ii*n+jj] += A[ii*n+kk]*B[kk*n+jj];
```
Example: Tuning OpenMP Parameters

```c
#pragma isat tuning
variable(@omp_schedule_type,[static, dynamic, guided])
variable(@omp_schedule_chunk, range(1000,10000,200))
search(dependent)

#pragma omp parallel for
for (i=0; i<N; i++)
    C[i] = Compute(A[i], B[i]);
```

*ISAT defined variables*

Instruct ISAT to try all possible combinations of the two variables
Example: Tuning TBB Parameters

```
#pragma isat tuning variable(@tbb_gain_size, range(100,1000,50))

parallel_for(blocked_range<size_t>(0, N), Compute())
```

Automatically instantiated as the 3rd argument of the blocked_range
Case Study: Tuning Game-of-Life

Game-of-Life:

- A cellular automation algorithm invented by John Conway in 1970
- The game board evolves from current generation to the next according to a set of rules
- Parallelism can be exploited by computing independent cells on the board in parallel
Game-of-Life in OpenMP

- An OMP implementation of Game-of-Life:

```c
int main() {
    ...
#pragma isat tuning measure(M1_begin, M1_end) scope(M1_begin, M1_end)
variable(x_blksize, range(200, 600, 10))
variable(y_blksize, range(200, 600, 10))
search(dependent)
#pragma isat marker M1_begin
    const int x_blksize = 100;
    const int y_blksize = 100;

    run_benchmark( new TiledOMPBoard(
        xsizes, ysizes,
        x_blksize, y_blksize,
        "tiledomp.log" ),
        count );
#pragma isat marker M1_end
}
```

```c
void TiledOMPBoard::next_gen() {
    int n = m_tiles.size();
#pragma isat tuning measure(M2_begin, M2_end)
variable(@omp_schedule_type, [static,dynamic,guided])
variable(@omp_schedule_chunk, range(1, 1000, 10))
search(dependent)
#pragma isat marker M2_begin
    // first update the border/halo information in each tile
#pragma isat marker M2_begin
    for( int i = 0; i < n; ++i ) {
        int west = getWest( i );
        const uch* border_west = west >= 0 ?
            m_tiles[west]->getEast() : 0;
        int ne = getNE( i );
        uch corner_ne = ne >= 0 ?
            m_tiles[ne]->getSW() : 0;
        int north = getNorth( i );
        const uch* border_north = north >= 0 ?
            m_tiles[north]->getSouth():0;
        int nw = getNW( i );
        uch corner_nw = nw >= 0 ? m_tiles[nw]->getSE() : 0;
        int east = getEast( i );
        const uch* border_east = east >= 0 ?
            m_tiles[east]->getWest() : 0;
        int sw = getSW( i );
        uch corner_sw = sw >= 0 ? m_tiles[sw]->getNE() : 0;
        int south = getSouth( i );
        const uch* border_south = south >= 0 ?
            m_tiles[south]->getNorth(): 0;
        int se = getSE( i );
        uch corner_se = se >= 0 ? m_tiles[se]->getNW() : 0;
        m_tiles[i]->updateBorders( border_north, border_east,
            border_south, border_west, corner_nw,
            corner_ne, corner_se, corner_sw );
    }
#pragma isat marker M2_end
}
```
Visualization of Tuning Results (1)

Tuning \((x_{blksize}, y_{blksize})\)

Tuned region=tr2

Execution time (secs) (lower is better)

View: 60,0000, 30,0000  scale: 1,000000, 1,000000
Visualization of Tuning Results (2)

Tuning the schedule for `parallel_for`
A Bit of Implementation Details

- **Implemented in Python**
  - Source-to-Source translation

- **Work with any C/C++ compilers**
  - Extendable to other languages: (Fortran, Java ...)

- **Thread-safe implementation**
Potential Features in Future Releases

• Tune for Energy:
  – Feasible on SandyBridge with its energy counters

• Support for tuning MPI programs
  – Attractive since MPI programs are highly latency sensitive

• Advanced searching strategies:
  – Machine learning

• Search for the best permutations of compiler optimizations
A Synergetic Approach to Parallel Programming on Multicores

Chi-Keung (CK) Luk
Multicore Architectures

Observations:
1. Multiple levels of hardware parallelism
2. Multiple levels of caches

=> Software must exploit both parallelism and locality
Our Approach

Basic Idea: *Divide-And-Conquer*

Steps:

1. Use *Cache-oblivious Techniques* to divide a large problem into smaller subproblems

2. Perform *Compiler-based Simdization* on the basecase

3. Use *Autotuning* to tune the parameters in Steps (1) and (2)
Step 1: Cache-Oblivious Parallelization

Decompose problem ...

.. recursively...

...until $\leq$ grainsize.

- **Parallelism:**
  - Independent subproblems are computed in parallel
  - Work-stealing achieves good load balancing

- **Choice of grainsize:**
  - Small enough to fit in the smallest cache
  - Big enough to amortize the recursion overhead
Example: **Strassen’s Matrix-Multiplication Algorithm**

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} =
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\times
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

**Strassen’s Algorithm:**

\[
P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})
\]

\[
P_2 = (A_{21} + A_{22}) \times B_{11}
\]

\[
P_3 = A_{11} \times (B_{12} - B_{22})
\]

\[
P_4 = A_{22} \times (B_{21} - B_{11})
\]

\[
P_5 = (A_{11} + A_{12}) \times B_{22}
\]

\[
P_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})
\]

\[
P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})
\]

\[
C_{11} = P_1 + P_4 - P_5 + P_7
\]

\[
C_{12} = P_3 + P_5
\]

\[
C_{21} = P_2 + P_4
\]

\[
C_{22} = P_1 - P_2 + P_3 + P_6
\]

**P_1 to P_7 can be computed in parallel**

**C_{11} to C_{22} can be computed in parallel once P_1 to P_7 are available**
Step 2: Compiler-based Simdization

- Simdization should be best done by compiler instead of programmer
- Simdization support in Intel Compiler:
  1. Auto-simdization
  2. #pragmas
  3. Array Notation
  4. SIMD Intrinsics (We don’t encourage)
Step 3: Autotuning

• To maximize performance, we can autotune the parameters in Steps (1) and (2):

  – Basecase size in a cache-oblivious algorithm

  – Degree of parallelism:
    – E.g., How many threads to use?

  – Level of parallelism:
    – Distribution of work over TLP, ILP, and SIMD

  – Scheduling policy and granularity
Case Study: Stencil Computations

• Stencil computations are very common:
  – Scientific computing, image processing, geometric modeling

• A $n$-dimension stencil:
  – The computation of an element in an $n$-dimensional spatial grid at time $t$ as a function of neighboring grid elements at times $t-1, \ldots , t-k$
Lattice Boltzmann Method (LBM)

• The stencil problem used in this case study:
  – Lattice Boltzmann Method (LBM) from SPEC’06
    – Performs numerical simulation in computational fluid dynamics in the 3D space

• LBM is a 19-point stencil:

Grid of cells

1 cell (with 19 velocity fields)
Original LBM Code

typedef enum {
    C=0, N, S, E, W, T, B,
    NE, NW, SE, SW, NT, NB,
    ST, SB, ET, EB, WT, WB,
    FLAGS, N_CELL_ENTRIES
} CELL_ENTRIES;

typedef LBM_Grid[(PADDING + SIZE_Z * SIZE_Y * SIZE_X) * N_CELL_ENTRIES];

void LBM_performStreamCollide(LBM_Grid* src, LBM_Grid* dst) {
    for (z=0; z<SIZE_Z; z++)
        for (y=0; y<SIZE_Y; y++)
            for (x=0, x<SIZE_X; x++) {
                // Compute dst[]'s as functions of src[]'s
            }
}

void main() {
    LBM_Grid* srcGrid, *dstGrid;
    srcGrid = AllocateGrid();
    dstGrid = AllocateGrid();
    ...
    for (int t=1; t<= nTimeSteps; t++) {
        LBM_performStreamCollide(*srcGrid, *dstGrid);
        LBM_swapGrids(&srcGrid, &dstGrid);
    }
}
Our LBM Optimization Strategy

• Subdivide the 4D space \((x, y, z, t)\) using Frigo and Strumpen’s cache-oblivious algorithm:
  – Maximize data reuse in all 4 dimensions

• Simdize the cache-oblivious basecase with `#pragma simd`

• Autotune the parameters in their algorithm
Frigo and Strumpen’s Stencil Algorithm

Maximize parallelism & locality in the 4D space
LBM: Cache-Oblivious + Autotuned

int NPIECES=2; int dx_threshold=32; int dy_threshold=2;
int dz_threshold=2; int dt_threshold=3;
#pragma isat tuning measure(start_timing, end_timing) scope(start_scope, end_scope)
variable(NPIECES, range(2, 8, 1)) variable(dx_threshold, range(2, 128, 1))
variable(dy_threshold, range(2, 128, 1)) variable(dz_threshold, range(2, 128, 1))
variable(dt_threshold, range(2, 128, 1))
#pragma isat marker(start_scope)
void CO(int t0, int t1, int x0, int dx0, int x1, int dx1,
        int y0, int dy0, int y1, int dy1, int z0, int dz0, int z1, int dz1) {
    int dt = t1-t0; int dx = x1-x0,
    int dy = y1-y0; int dz = z1-z0;
    if (dx >= dx_threshold && dx >= dy && dx >= dz &&
        dt >= 1 && dx >= 2 * dt * NPIECES) {
        int chunk = dx / NPIECES; int i;
        for (i=0; i<NPIECES-1; ++i)
            cilk_spawn CO(t0, t1, x0+i*chunk, 1, x0+(i+1)*chunk, -1,
                          y0, dy0, y1, dy1, z0, dz0, z1, dz1);
        CO(t0, t1, x0+i*chunk, 1, x1, -1, y0, dy0, y1, dy, z0, dz0, z1, dz1);
        cilk_sync;
    } else if (… /* Subdivide in y dimension? */)
    …
} else if (… /* Subdivide in z dimension? */)
    …
} else if (… /* Subdivide in t dimension? */)
    …
} else { /* call the basecase */
    BaseCase(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1, dy1, z0, dz0, z1, dz1);
}
#pragma isat marker(end_scope)
Case Study 2: Binary-Tree Search

- Search a query based on its key in a database organized as a packed binary tree

**Original Breath-first layout**

The number shown in each node is the key.

Number in [] is the memory location of the node.

**Corresponding Search Code**

```c
int Keys[numNodes]; // keys organized as a binary tree
int Queries[numQueries]; // input queries
int Answers[numQueries]; // output if the query is found

void ParallelSearchForBreadthFirstLayout() {
    // Search the queries in parallel
    cilk_for (int q=0; q<numQueries; q++) {
        const int searchKey = Queries[q];

        // Look for searchKey in the binary tree
        for (int i=0; i<numNodes; ) {
            const int currKey = Key[i];

            if (searchKey == currKey) {
                Answers[q] = 1;
                break; // found
            } else if (searchKey < currKey)
                i = 2*i + 1;
            else
                i = 2*i + 2;
        }
    }
}
```
Optimization Opportunities

- Two optimization opportunities:
  1. Reducing cache misses
  2. Simdizing the search

- Our strategy:
  - Use cache-oblivious tree layout
    - A variant of the Van Emde Boas layout that also facilitates simdization
  - Use array notation to simdize
    - 1 SIMD lane processes 1 query
  - Use autotuning to distribute work over TLP, ILP, and SIMD

Cache-oblivious layout

Number in each node is the key
Number in [ ] is the memory location of the node

= a subtree
#pragma isat tuning scope(start_scope, end_scope) measure(start_timing, end_timing)
  variable(SUBTREE_HEIGHT, [4,6,8,12])
#pragma isat tuning scope(start_scope, end_scope) measure(start_timing, end_timing)
  variable(BUNDLE_SIZE, range(8,64,1)) variable(VLEN, range(4,64,4)) search(dependent)

void ParallelSearchForCacheOblivious() {
  int numNodesInSubTree = (1 << SUBTREE_HEIGHT) - 1;
  int bundleSize = BUNDLE_WIDTH * VLEN; int remainder = numQueries % bundleSize;
  int quotient = numQueries / bundleSize; int numBundles = ((remainder==0)? quotient : (quotient+1));

cilk_for (int b=0; b < numBundles; b++) {
  int q_begin = b * bundleSize; int q_end = MIN(q_begin+bundleSize, numQueries);
  for (int q = q_begin; q < q_end; q += VLEN) {
    int searchKey[VLEN] = Queries[q:VLEN]; int* array[VLEN] = Keys;
    int subTreeIndexInLayout[VLEN] = 0; int localAnswers[VLEN] = 0;
    for (int hTreeLevel=0; hTreeLevel < HierTreeHeight; ++hTreeLevel) {
      int i[VLEN] = 0;
      for (int levelWithSubTree = 0; levelWithSubTree < SUBTREE_HEIGHT; ++levelWithSubTree) {
        int currKey[VLEN];
        for (int k=0; k<VLEN; k++)
          currKey[k] = (array[k])[i[k]];
        bool eq[:] = (searchKey[:] == currKey[:]);
        bool lt[:] = (searchKey[:] < currKey[:]);
        localAnswers[:] = eq[:]? 1: localAnswers[:];
        i[:] = localAnswers[:]? i[:]: ((lt[:])? (2*i[:]+1): (2*i[:]+2));
      }
      int whichChild[VLEN] = i[:]-numNodesInSubTree;
      subTreeIndexInLayout[:]= localAnswers[:]? subTreeIndexInLayout[:]:
      (subTreeIndexInLayout[:]<<SUBTREE_HEIGHT + whichChild[:]+1);
      array[:] = localAnswers[:]? array[:]:
      (Keys + subTreeIndexInLayout[:]* numNodesInSubTree);
    }
    Answers[q:VLEN] = localAnswers[:];
  }
}
# Experimental Framework

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Intel Nehalem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Clock</td>
<td>2.27 GHz</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>8 cores (on 2 sockets)</td>
</tr>
<tr>
<td>Memory Size</td>
<td>12 GB</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>22.6 GB/s</td>
</tr>
<tr>
<td>Compiler</td>
<td>ICC v12 -fast</td>
</tr>
<tr>
<td>OS</td>
<td>64bit CentOS 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3dfd</td>
<td>3dfd finite difference computation</td>
</tr>
<tr>
<td>Bilateral</td>
<td>Bilateral image filtering</td>
</tr>
<tr>
<td>LBM</td>
<td>Lattice Boltzeman Method</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>Dense matrix multiplication</td>
</tr>
<tr>
<td>Search</td>
<td>Searching a binary tree</td>
</tr>
<tr>
<td>Sort</td>
<td>Sorting</td>
</tr>
</tbody>
</table>
Overall Performance Results

Our approach is 19x faster than best serial or 4x faster than loop-based parallel
LBM Detailed Results

• Other researchers have applied the AOS-to-SOA optimization onto LBM
• So, how does our approach perform against the SOA optimization?

![Graph showing speedup over serial for different configurations]

- LoopParallel+SIMD with AOS (default): 1.15
- LoopParallel+SIMD with SOA: 2.7
- Cache-oblivious Parallel + SIMD: 3.8
How Autotuning Improved TreeSearch

Best configuration found: VLEN=48, BUNDLE_SIZE=32
Summary

• Intel® Cilk™ Plus is a simple way to write efficient parallel programs on multicores

• A synergetic approach based on Cilk Plus with autotuning achieves 19x speedup over best serial

• Call for actions:
  – Try Cilk Plus in the Intel® Compiler:
  – Try CilkView and CilkScreen:
  – Try Intel® Software Autotuning Tool:
  – Download our IEEE Software paper:
Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101
Legal Disclaimer

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.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.


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

Copyright © 2010. Intel Corporation.

http://intel.com/software/products