

ATPESC

We resume @ **10:45am**

(Argonne Training Program on Extreme-Scale Computing)

Vectorization (SIMD), and scaling (TBB and OpenMP*)

James Reinders, Intel

August 3, 2015, Pheasant Run, St Charles, IL

10:45 – 12:00



ATPESC

(Argonne Training Program on Extreme-Scale Computing)

Vectorization (SIMD), and scaling (TBB and OpenMP*)

James Reinders, Intel

August 3, 2015, Pheasant Run, St Charles, IL

10:45 – 12:00





Use abstractions !!!



Choosing a non-proprietary *parallel abstraction*

non-proprietary	BLAS, FFTW	MPI	OpenMP*	TBB	Cilk™ Plus
prog. lang.	Fortran, C, C++	Fortran, C, C++	Fortran or C	C++	C++

Use abstractions !!!

Avoid direct programming to the low level interfaces (like pthreads).

PROGRAM IN TASKS, NOT THREADS

Is OpenCL* low level? For HPC - YES.

Choosing a non-proprietary *parallel abstraction*

non-proprietary	BLAS, FFTW	MPI	OpenMP*	TBB	Cilk™ Plus
prog. lang.	Fortran, C, C++	Fortran, C, C++	Fortran or C	C++	C++



Choose First
(limited functions)

Choosing a non-proprietary *parallel abstraction*

non-proprietary	BLAS, FFTW	MPI	OpenMP*	TBB	Cilk™ Plus
prog. lang.	Fortran, C, C++	Fortran, C, C++	Fortran or C	C++	C++



Choose First
(limited functions)



Cluster
(distributed memory)

Choosing a non-proprietary *parallel abstraction*

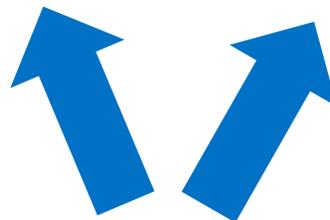
non-proprietary	BLAS, FFTW	MPI	OpenMP*	TBB	Cilk™ Plus
prog. lang.	Fortran, C, C++	Fortran, C, C++	Fortran or C	C++	C++



Choose First
(limited functions)



Cluster
(distributed
memory)



Node
(shared
memory)

Choosing a non-proprietary *parallel abstraction*

non-proprietary	BLAS, FFTW	MPI	OpenMP*	TBB	Cilk™ Plus
prog. lang.	Fortran, C, C++	Fortran, C, C++	Fortran or C	C++	C++



Choose First
(limited functions)



Cluster
(distributed
memory)



Node
(shared
memory)

Up and coming
for C++
(keywords,
compilers)

Because... you
just have to
expect "more"

Affect future
C++ standards?
(2021?)

Choosing a non-proprietary *parallel abstraction*

non-proprietary	BLAS, FFTW	MPI	OpenMP*	TBB	Cilk™ Plus
prog. lang.	Fortran, C, C++	Fortran, C, C++	Fortran or C	C++	C++
implemented	vendor libraries	many	in compiler	portable	in compiler
standard	open interfaces	open interfaces	OpenMP standard (1997-)	open source (2007, Intel)	open interfaces (MIT, Intel)
supported by	most vendors	open src & vendors	most compilers	ported most everywhere	gcc and Intel (llvm future)

Compare...

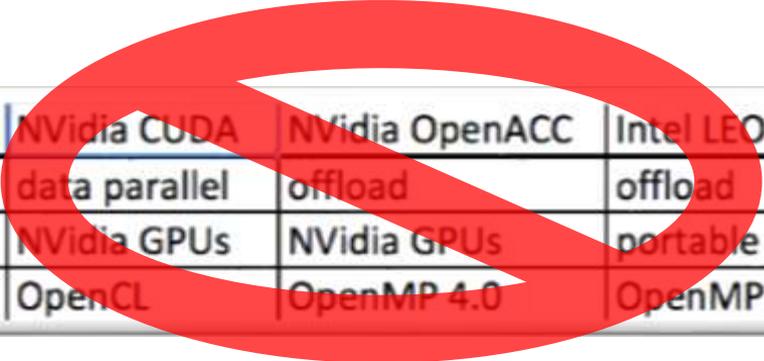
proprietary	NVidia* CUDA	NVidia OpenACC	Intel LEO
purpose	data parallel	offload	offload
target (perf.)	NVidia GPUs	NVidia GPUs	portable
alternative	OpenCL*	OpenMP 4.0	OpenMP 4.0

Choosing a non-proprietary *parallel abstraction*

non-proprietary	BLAS, FFTW	MPI	OpenMP*	TBB	Cilk™ Plus
prog. lang.	Fortran, C, C++	Fortran, C, C++	Fortran or C	C++	C++
implemented	vendor libraries	many	in compiler	portable	in compiler
standard	open interfaces	open interfaces	OpenMP standard (1997-)	open source (2007, Intel)	open interfaces (MIT, Intel)
supported by	most vendors	open src & vendors	most compilers	ported most everywhere	gcc and Intel (llvm future)

Compare...

proprietary	NVidia CUDA	NVidia OpenACC	Intel LEO
purpose	data parallel	offload	offload
target (perf.)	NVidia GPUs	NVidia GPUs	portable
alternative	OpenCL	OpenMP 4.0	OpenMP 4.0



Choosing a non-proprietary *parallel abstraction*

non-proprietary	BLAS, FFTW	MPI	OpenMP*	TBB	Cilk™ Plus
prog. lang.	Fortran, C, C++	Fortran, C, C++	Fortran or C	C++	C++
implemented	vendor libraries	many	in compiler	portable	in compiler
standard	open interfaces	open interfaces	OpenMP standard (1997-)	open source (2007, Intel)	open interfaces (MIT, Intel)
supported by	most vendors	open src & vendors	most compilers	ported most everywhere	gcc and Intel (Illum future)
composable?	usually	YES	NO	YES	YES
memory	shared/distributed	distributed	shared (in implementations)	shared memory	shared memory
tasks	yes	n/a	YES	YES	limited keywords, TBB
explicit SIMD	internal	n/a	YES (OpenMP 4.0: SIMD)	use compiler options, OpenMP directives, or Cilk Plus keywords	keywords
offload	some	n/a	YES (OpenMP 4.0: SIMD)	use Cilk Plus or OpenMP	keywords

Best options for Performance *and* Performance Portability



Intel Threading Building Blocks

We asked ourselves:

- How should C++ be extended?
 - “templates / generic programming”
- What do we want to solve?
 - Abstraction with good performance (scalability)
 - Abstraction that steers toward easier (less) debugging
 - Abstraction that is readable

Intel® Threading Building Blocks (Intel® TBB)

C++ Library for parallel programming

- Takes care of managing multitasking

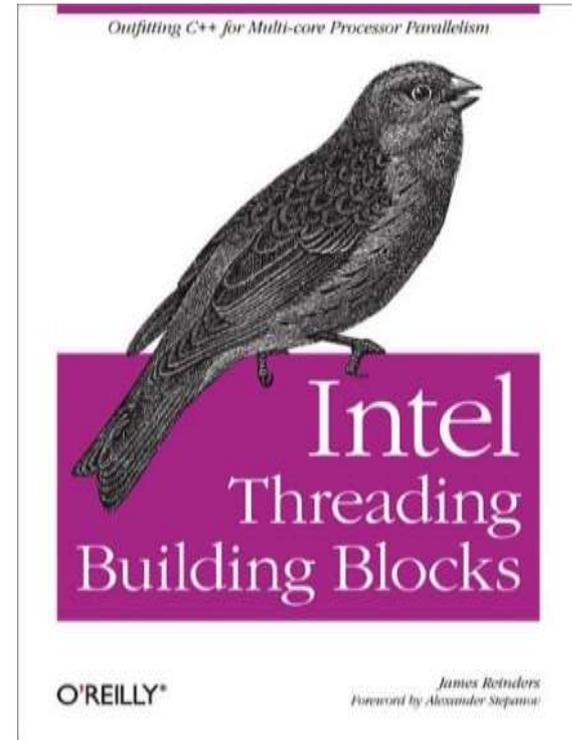
Runtime library

- Scalability to available number of threads

Cross-platform

- Windows*, Linux*, Mac OS* and others

<http://threadingbuildingblocks.org/>

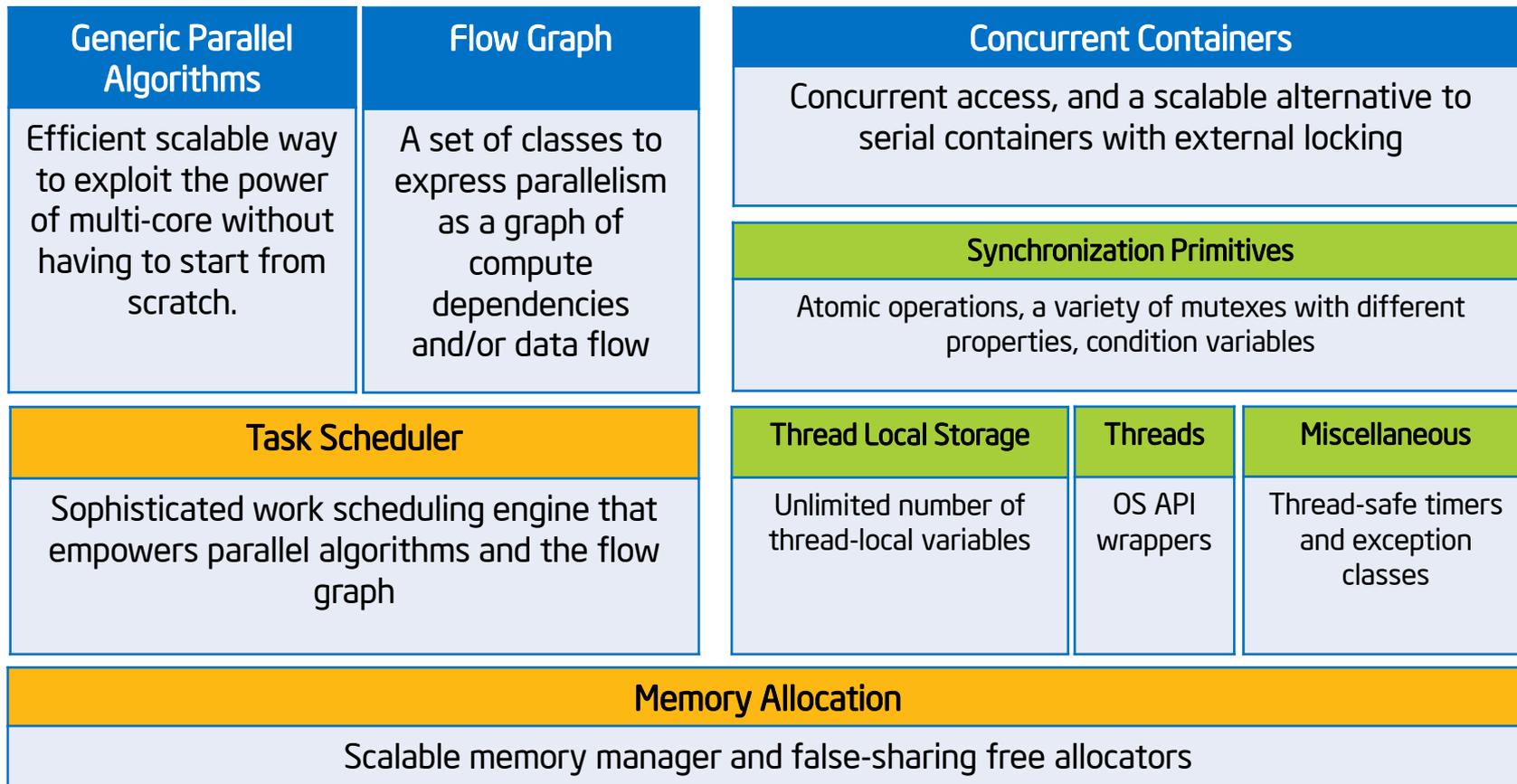


Rich Feature Set for Parallelism

Parallel algorithms and data structures

Threads and synchronization

Memory allocation and task scheduling



Rich Feature Set for Parallelism

Parallel algorithms and data structures

Threads and synchronization

Memory allocation and task scheduling

Generic Parallel Algorithms

Efficient scalable way to exploit the power of multi-core without having to start from scratch.

Flow Graph

A set of classes to express parallelism as a graph of compute dependencies and/or data flow

Concurrent Containers

Concurrent access, and a scalable alternative to serial containers with external locking

Synchronization Primitives

Atomic operations, a variety of mutexes with different properties, condition variables

Task Scheduler

Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph

Thread Local Storage

Unlimited number of thread-local variables

Threads

OS API wrappers

Miscellaneous

Thread-safe timers and exception classes

Memory Allocation

Scalable memory manager and false-sharing free allocators

Generic Algorithms

Loop parallelization

`parallel_for`

`parallel_reduce`

- load balanced parallel execution
- fixed number of independent iterations

`parallel_scan`

- computes parallel prefix

$$y[i] = y[i-1] \text{ op } x[i]$$

Parallel sorting

`parallel_sort`

Parallel function invocation

`parallel_invoke`

- Parallel execution of a number of user-specified functions

Parallel Algorithms for Streams

`parallel_do`

- Use for unstructured stream or pile of work
- Can add additional work to pile while running

`parallel_for_each`

- `parallel_do` without an additional work feeder

`pipeline / parallel_pipeline`

- Linear pipeline of stages
- Each stage can be parallel or serial in-order or serial out-of-order.
- Uses cache efficiently

Computational graph

`flow::graph`

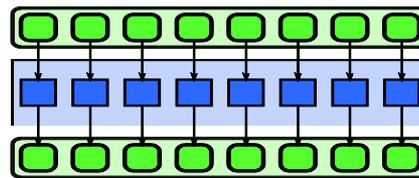
- Implements dependencies between nodes
- Pass messages between nodes

Parallel For

tbb::parallel_for

Has several forms.

Map



Execute $functor(i)$ for all $i \in [lower, upper)$

```
parallel_for( lower, upper, functor );
```

Execute $functor(i)$ for all $i \in \{lower, lower+stride, lower+2*stride, \dots\}$

```
parallel_for( lower, upper, stride, functor );
```

Execute $functor(subrange)$ for all $subrange$ in $range$

```
parallel_for( range, functor );
```

tbb::parallel_for

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#define N 10

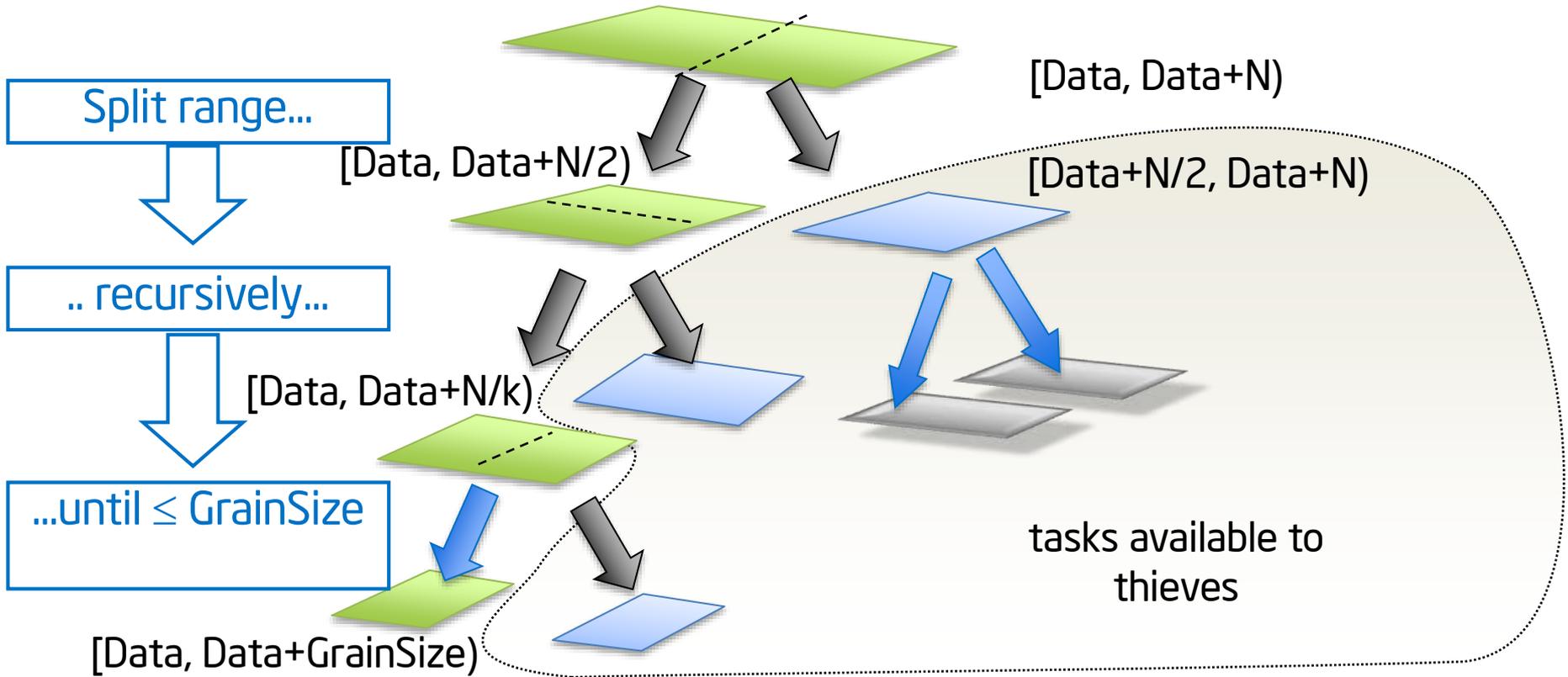
inline int Prime(int & x) {
    int limit, factor = 3;
    limit = (long)(sqrtf((float)x)+0.5f);
    while( (factor <= limit) && (x % factor))
        factor ++;
    x = (factor > limit ? x : 0);
}

int main (){
    int a[N];
    // initialize array here...
    tbb::parallel_for (0, N, 1,
        [&](int i){
            Prime (a[i]);
        });
    return 0;
}
```

A call to a template function
`parallel_for` (lower, upper, stride, functor)

Task: loop body as C++ lambda expression

Recursive parallelism

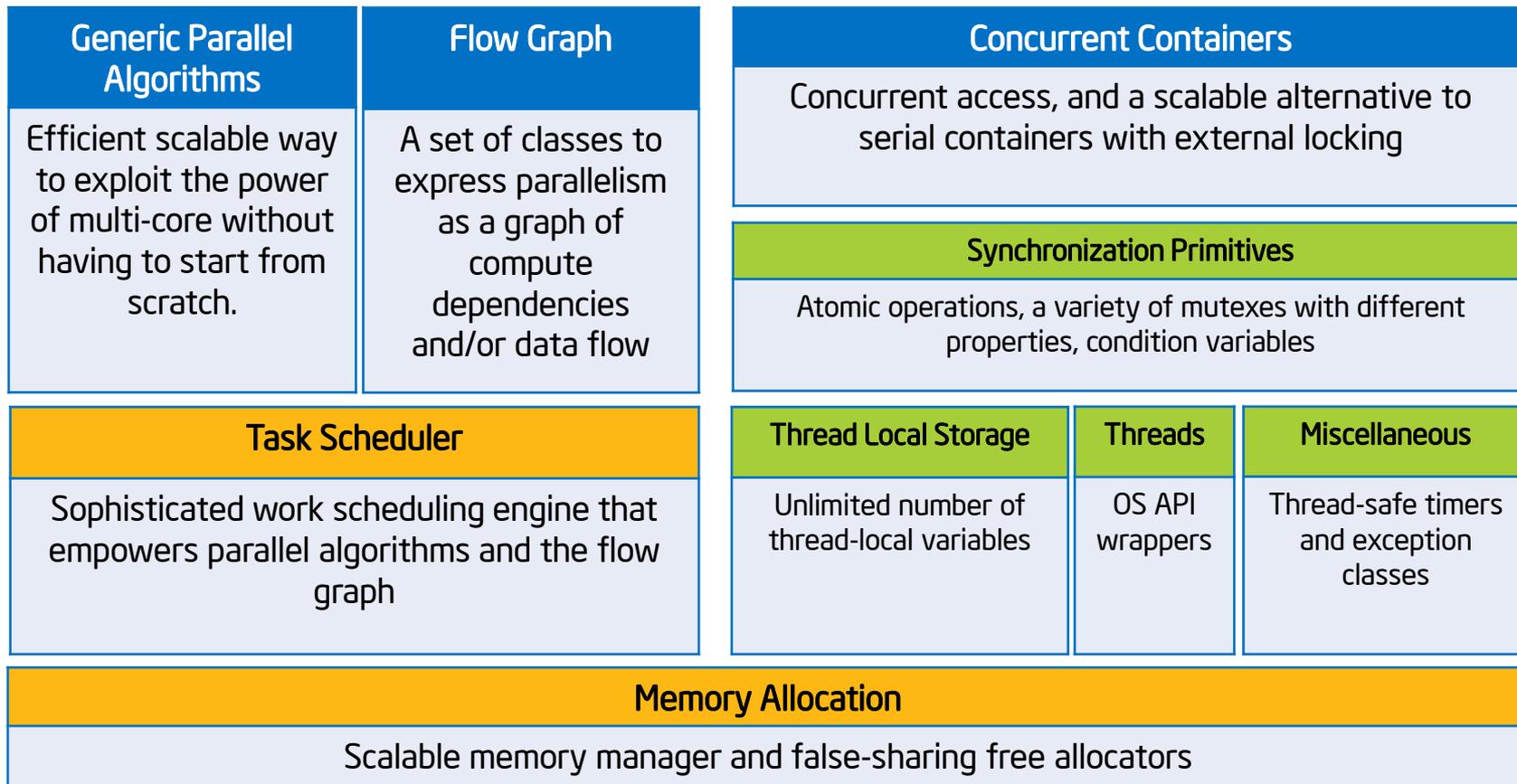


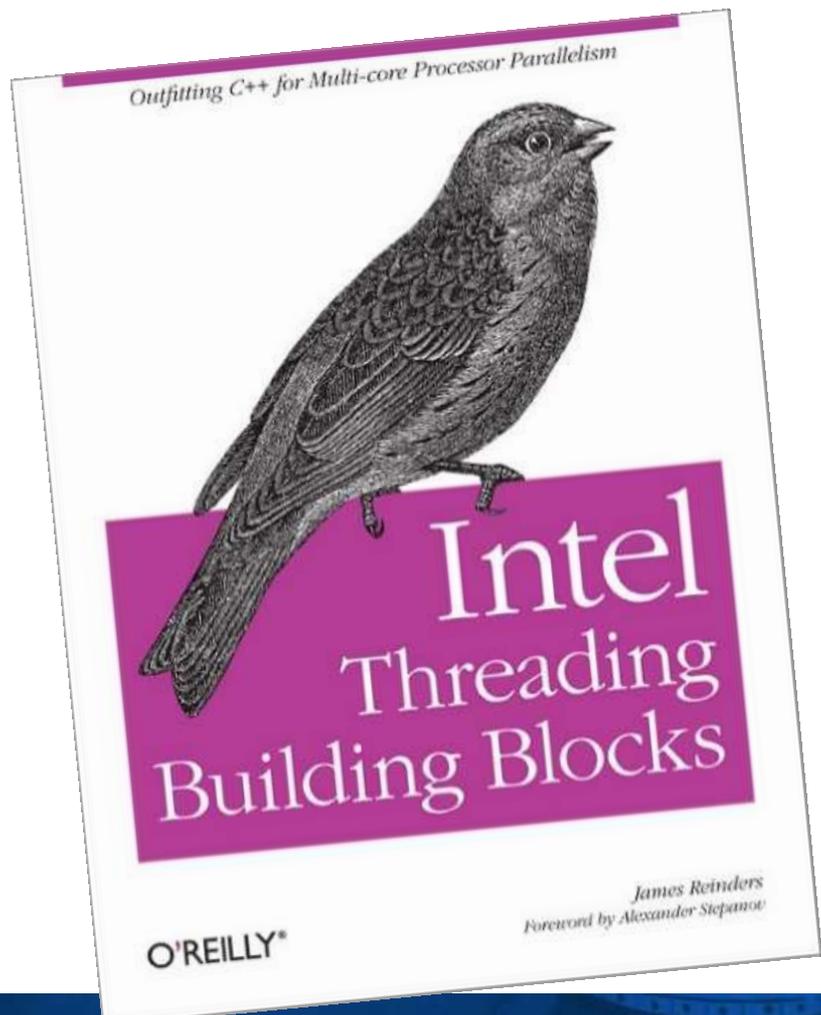
Rich Feature Set for Parallelism

Parallel algorithms and data structures

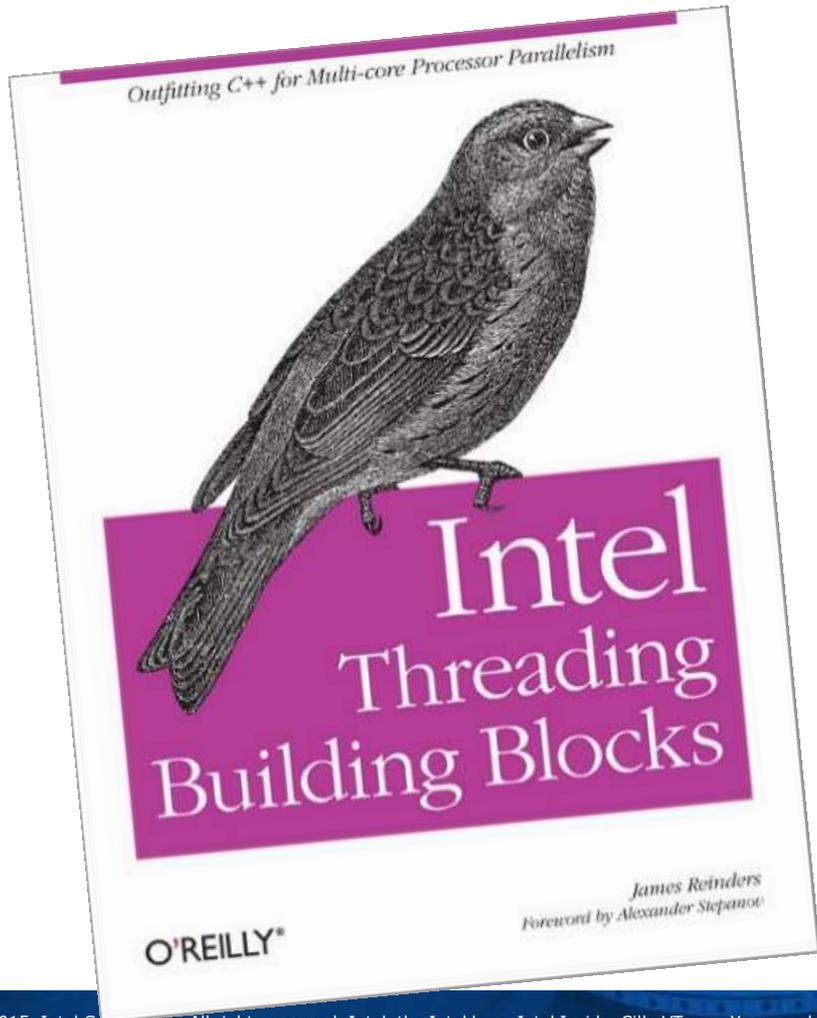
Threads and synchronization

Memory allocation and task scheduling





The MOST popular
abstract parallelism
model for C++



The MOST popular
abstract parallelism
model for C++

Down with
OpenMP!



Sorry OpenMP

You just do not cut it.

(for C++)



Sorry ~~OpenMP~~

You just do not cut it.

**Down with
OpenMP!**

(for C++)

The next few slides are based on following paper from WHPCF'14:



STAC-A2 on Intel Architecture: From Scalar Code to Heterogeneous Application

Evgeny Fiksman
evgeny.fiksman@intel.com

Sania Salahuddin
sania.salahudin@intel.com

SC'14, New Orleans, November 16th, 2014

STAC-A2 overview (<https://stacresearch.com/>)

- A vendor independent market risk analysis benchmark
- Defined by Securities Technology Analysis Center (STAC*)
- Calculate “Greeks” – sensitivity of the option price to changes in parameters of the underlying market
- Heston option pricing model & Least Squares Monte Carlo of Longstaff & Schwartz
- Benchmark Metrics
 - Speed (GREEKS.TIME.COLD/WARM)
 - Workload scalability (MAX_ASSETS, MAX_PATHS)
 - Power & Space efficiency
 - Quality

TBB used on STAC-A2 Benchmark – beat OpenMP

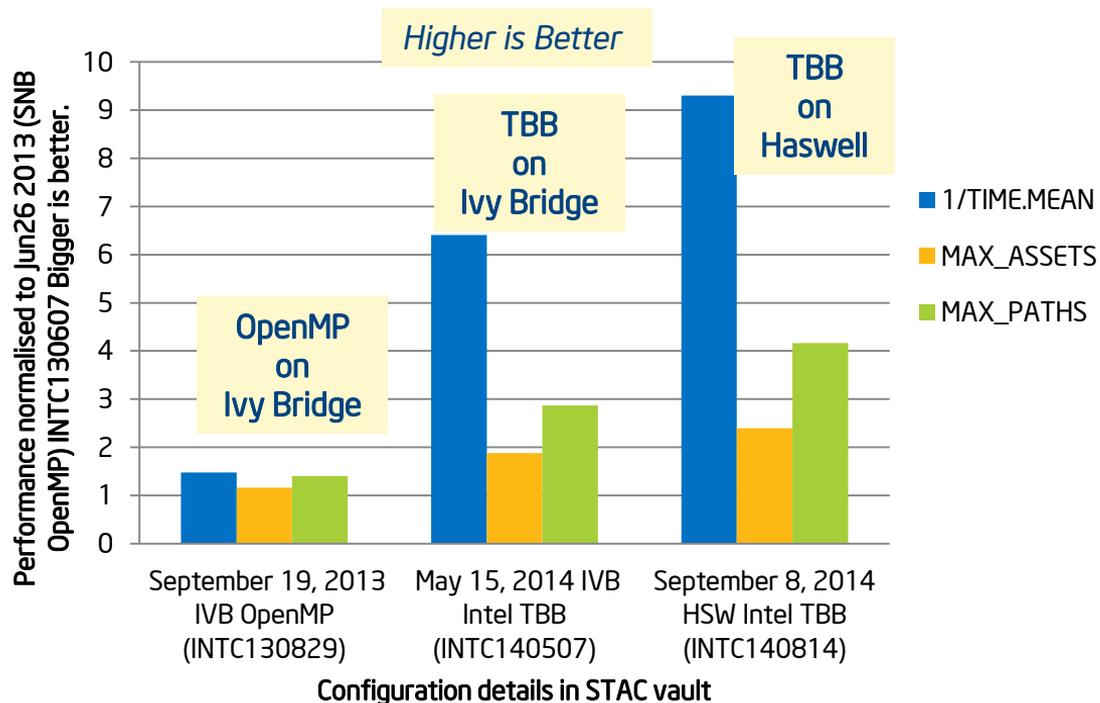


130829 and 140507 use identical hardware

140507 and 140814 use identical source code

This is portable code: no “intrinsic”

~1.45x from each HW generation, SW change worth at least 2 HW generations



Parallelization choices matter

Hold on!!!

Who is the invited
keynote speaker
for OpenMP conference
in September 2015?



How did Intel TBB beat OpenMP annotations on STAC-A2?

OpenMP annotations work well when

- You control the whole machine
- You have one level of parallelism
- You want to take low level control of scheduling, placement,...

Intel TBB tends to out perform OpenMP when...

- You don't know about the machine you'll run on
- You have many levels of parallelism (recursive, or in libraries)
- You're happy to let the runtime handle things

Both are portable: Intel TBB does not require compiler support. Both are reasonably performance portable in practice, although TBB is composable – which can be a significant advantage in perf. port.

OpenMP is very popular – and works very well on technical applications (like HPC) with C and Fortran.

But, for C++... TBB is better.

I was having a little fun... to make a point.

**We love
TBB!!!**

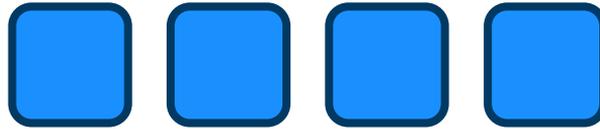
**Long live
OpenMP!**



Nested parallelism is
important to exploit.

Trending: more and more so.

OpenMP Nested Parallelism: HOT TEAMS



OpenMP worker threads -
created ONCE PER PROGRAM

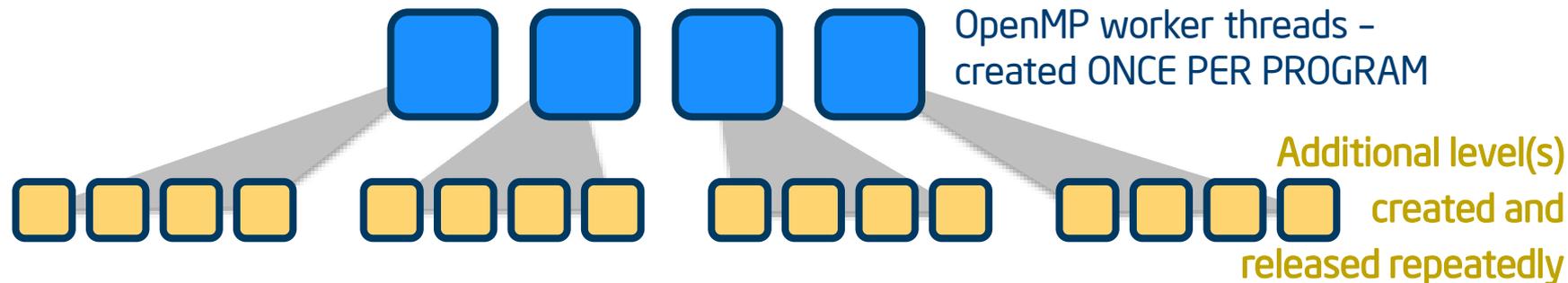
NESTED PARALLEL:

By DEFAULT, any parallel worker that executes a parallel construct does that work inside the same worker thread.

PRO: controlled memory footprint (including stack space)

CON: no load balancing

OpenMP Nested Parallelism: HOT TEAMS



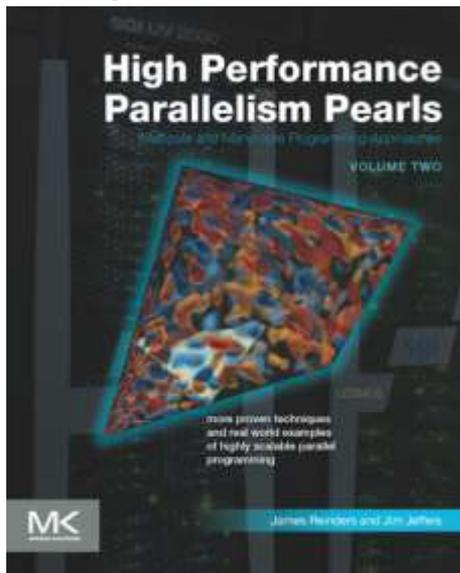
NESTED PARALLEL:

TURN ON NESTING (no code changes - done with
environment variables)

PRO: load balancing

CON: high overhead, potential oversubscription (runaway
memory/stack usage being the key issue)

<http://lotsofcores.com>



“High Performance Parallelism Pearls Volume 2” - available now!

73 expert contributors
23 affiliations
10 countries
24 contributed chapters

Volume 2: August 2015

Foreword
Introduction
Numerical Weather Prediction Optimization
WRF Goddard Microphysics Scheme Optimization
Pairwise DNA Sequence Alignment Optimization
Accelerated Structural Bioinformatics for Drug Discovery
Amber PME Molecular Dynamics Optimization
Low Latency Solutions for Financial Services
Parallel Numerical Methods in Finance
Wilson Dslash Kernel From Lattice QCD Optimization

Cosmic Microwave Background Analysis: Nested Parallelism In Practice
Visual Search Optimization
Radio Frequency Ray Tracing
Exploring Use of the Reserved Core
High Performance Python Offloading
Fast Matrix Computations on Asynchronous Streams
MPI-3 Shared Memory Programming Introduction
Coarse-Grain OpenMP for Scalable Hybrid Parallelism
Exploiting Multilevel Parallelism with OpenMP
OpenCL: There and Back Again
OpenMP vs. OpenCL: Difference in Performance?
Prefetch Tuning Optimizations
SIMD functions via OpenMP
Vectorization Advice
Portable Explicit Vectorization Intrinsics
Power Analysis for Applications and Data Centers

Nested OpenMP is an optional feature of the OpenMP standard. Its support is subject to the compilers and runtime libraries. The default is to ignore OpenMP parallel regions within a running parallel region; in OpenMP parlance, the nested regions are serialized. This can be overridden by setting `OMP_NESTED=true`. The Intel OpenMP runtime has greatly improved performance for nested OpenMP since releasing Intel Composer XE 15.1 with so-called `HOT_TEAMS`. They are enabled in our experiments by setting these environment variables:

```
export KMP_HOT_TEAMS_MODE=1
export KMP_HOT_TEAMS_MAX_LEVEL=7
export MKL_DYNAMIC=false
```

Note that we set `MKL_DYNAMIC=false` for DGEMM or FFT when they are used.

HOT TEAMS MOTIVATION

“Hot teams” is an extension to OpenMP supported by the Intel runtime. It reduces the overhead of OpenMP parallelism. It works with standard OpenMP code but enables nested parallelism. It is a logical extension that may inspire similar capabilities in other implementations.

To understand “hot teams,” it is important to know that any modern implementation of OpenMP, in order to avoid the cost of creating and destroying pthreads, has the OpenMP runtime maintain a pool of OS threads (pthreads on Linux) that it has already created. This is standard practice in OpenMP runtimes because OS thread creation is normally quite expensive.

However, OpenMP also has a concept of a thread team, which is the set of pthreads that will execute

OpenMP 4.0 AFFINITY AND HOT TEAMS OF INTEL OpenMP RUNTIME

A node contains multiple parallel units—multiple cores, multiple sockets, multiple hardware threads, and optionally coprocessors. The ability to bind OpenMP threads to physical processing units has become increasingly important to achieve high performance on these modern CPUs. OpenMP 4.0 affinity features provide standard ways to control thread affinity that can have a dramatic performance effect. This impact is especially true on current generation Intel Xeon Phi coprocessors: four hardware threads share the L1/L2 cache of an in-order core. We use OpenMP runtime environments to optimally bind MPI tasks and OpenMP threads. For instance, when using 5 MPI and 12 OpenMP threads for the band loop and 4 OpenMP threads for compute, they are set as

```
export OMP_NESTED=true
export OMP_NUM_THREADS=12,4
export OMP_PLACES=threads
export OMP_PROC_BIND=spread,close
mpirun -np 5 ./
```

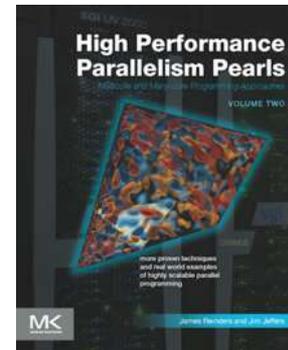
```
export OMP_NESTED=true
export OMP_NUM_THREADS=12,4
export OMP_PLACES=threads
export OMP_PROC_BIND=spread,close
mpirun -np 5 ./myapp
```

CHAPTER 10 COSMIC MICROWAVE BACKGROUND ANALYSIS

costs are prohibitively expensive when the nested regions are encountered often, such as when the threads are spawned for an inner-most loop.

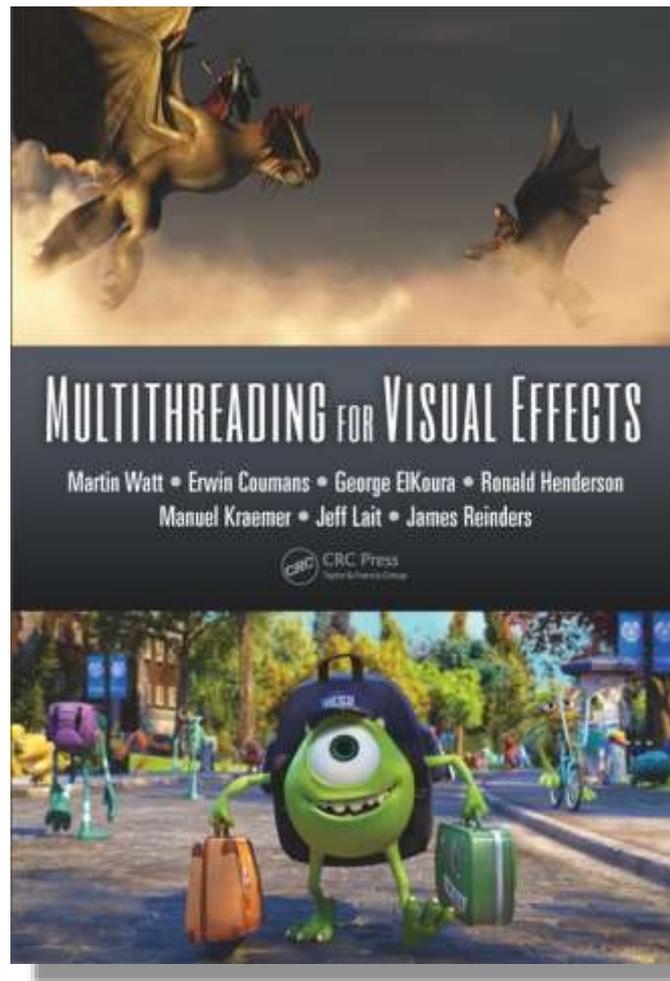
There is, however, support for an experimental feature in the Intel® OpenMP runtime (Version 15 Update 1 or later) known as “hot teams” that is able to reduce these overheads, by keeping a pool of threads alive (but idle) during the execution of the non-nested parallel code. The use of hot teams is controlled by two environment variables: `KMP_HOT_TEAMS_MODE` and `KMP_HOT_TEAMS_MAX_LEVEL`. To keep unused team members alive when team sizes change we set `KMP_HOT_TEAMS_MODE=1`, and because we have two levels of parallelism we set `KMP_HOT_TEAMS_MAX_LEVEL=2`.

Care must also be taken with thread affinity settings. OpenMP 4.0 provides new environment variables for handling the physical placement of threads, `OMP_PROC_BIND` and `OMP_PLACES`, and these are compatible with nested parallel regions. To place team leaders on separate cores, and team members on the same core, we set `OMP_PROC_BIND=spread,close` and `OMP_PLACES=threads`.



Parallel first

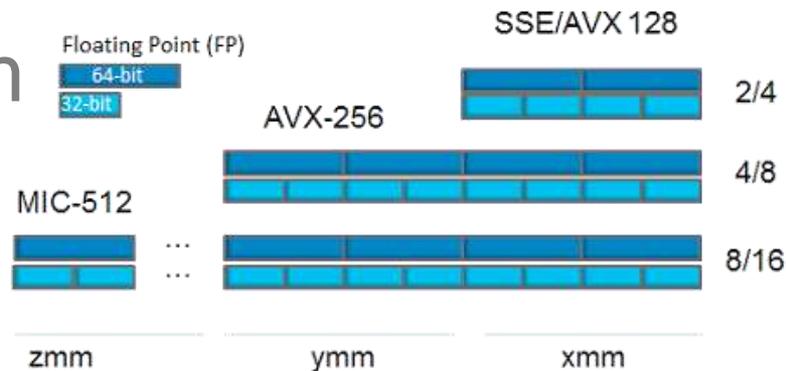
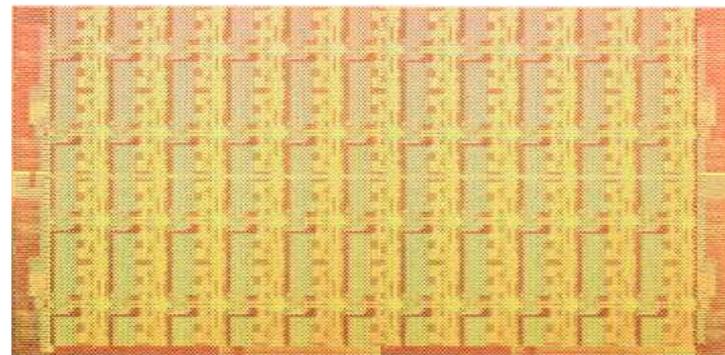
Vectorize second



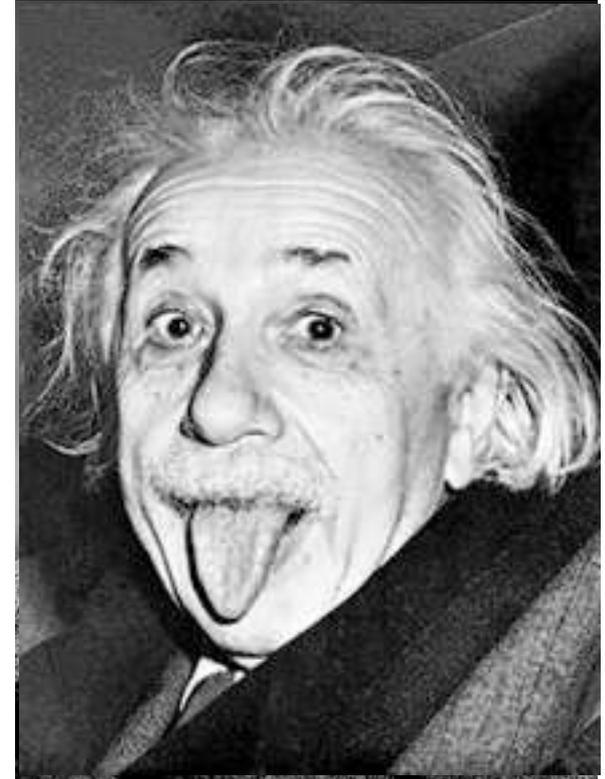
Multithreading is more powerful than vectorization – by simple math:

16 way from vectorization

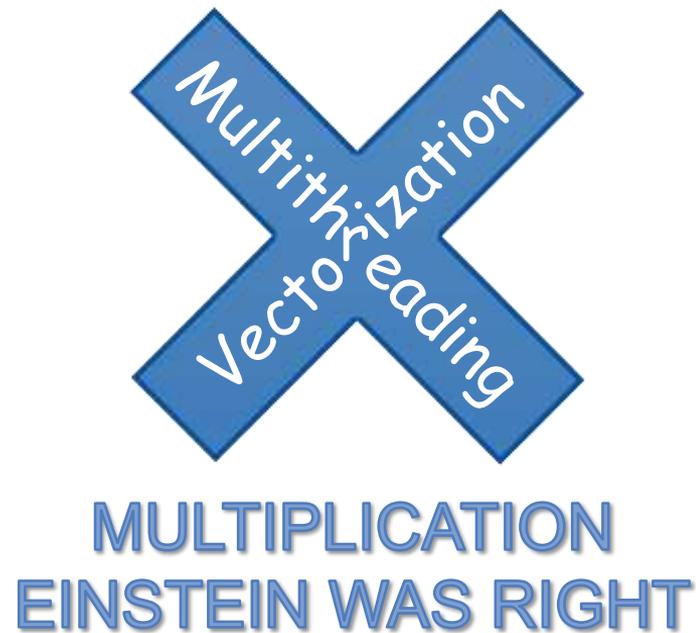
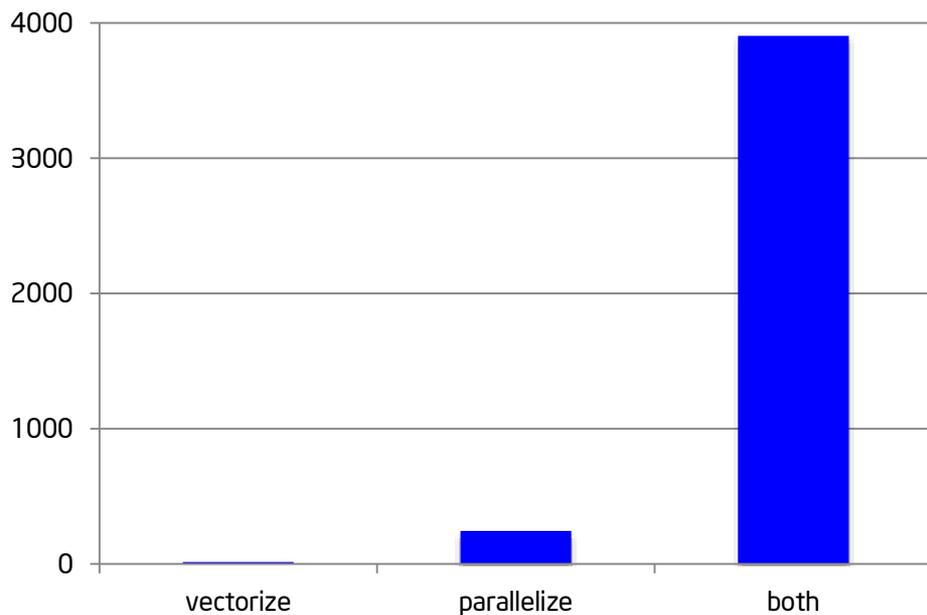
244 way from thread parallelism



There is an urban legend that Albert Einstein once said that compounding interest is the most powerful force in the universe.



$$16 \times 244 = 3904$$



How many of us here today...
have ever worried about vectorization for
your application?

Assertion:

We need to embrace *explicit* vectorization
in our programming.

Shouldn't we solve with better tools?

What is vectorization?

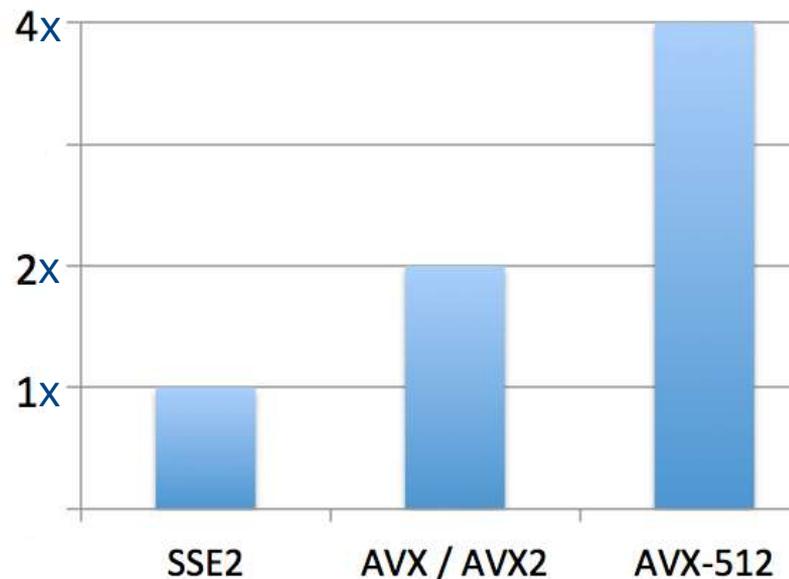
Could we just ignore it?

Vectors Instructions (SIMD instructions) Make things Faster

(that's the premise)

Up to 4x Performance

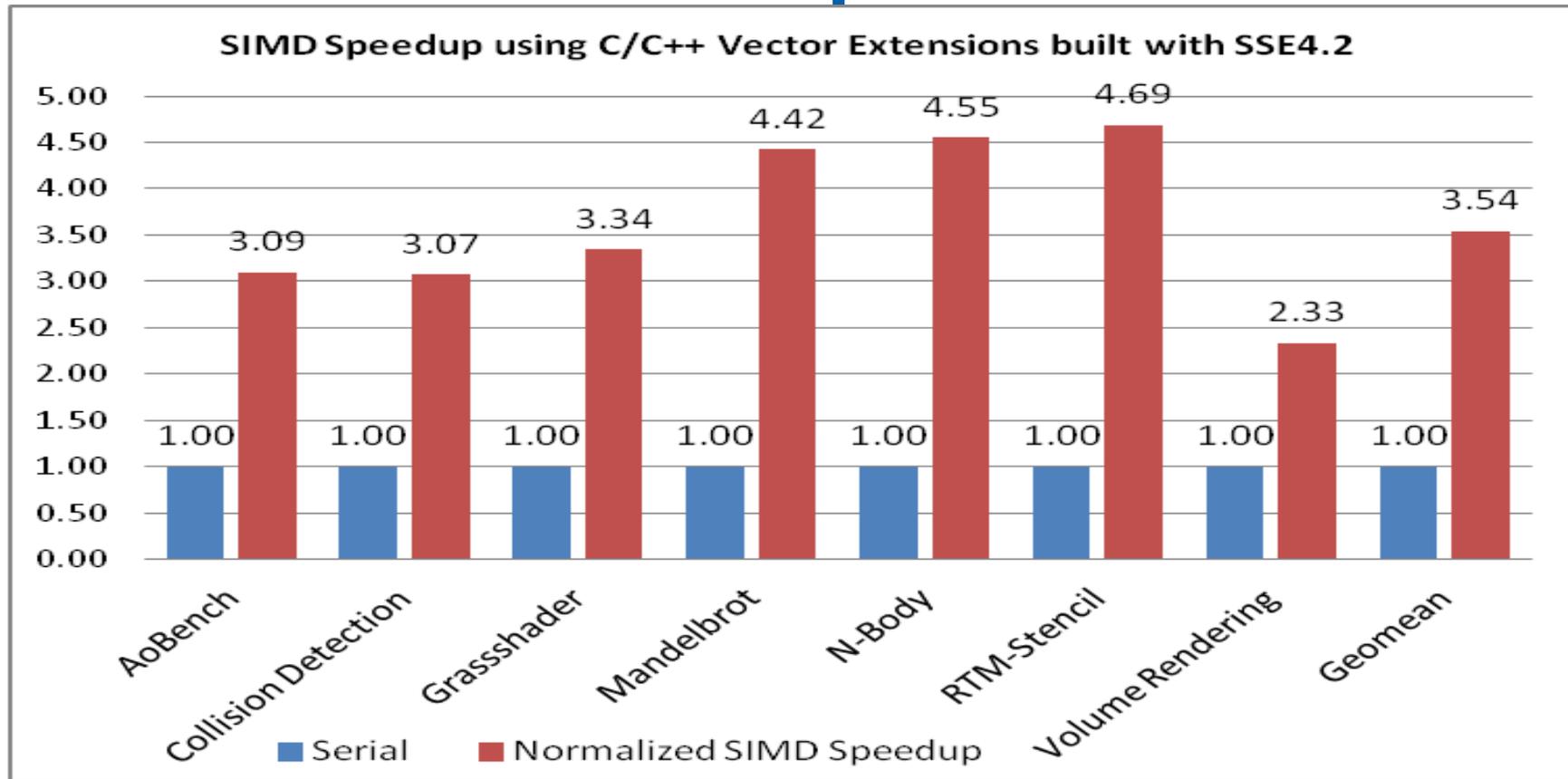
with Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Support



- Significant leap to 512-bit SIMD support for processors
- Intel® Compilers and Intel® Math Kernel Library include AVX-512 support
- Strong compatibility with AVX
- Added EVEX prefix enables additional functionality
- Appears first in future Intel® Xeon Phi™ coprocessor, code named Knights Landing

Higher performance for the most demanding computational tasks

Performance with Explicit Vectorization



Configuration: Intel® Core™ i7 CPU X980 system (6 cores with Hyper-Threading On), running at 3.33GHz, with 4.0GB RAM, 12M smart cache, 64-bit Windows Server 2008 R2 Enterprise SP1. For more information go to <http://www.intel.com/performance>

What is a Vector?

Vector of numbers

[4.4 | 1.1 | 3.1 | -8.5 | -1.3 | 1.7 | 7.5 | 5.6 | -3.2 | 3.6 | 4.8]

Vector addition

$$\begin{array}{r} + \\ = \end{array} \begin{bmatrix} 4.4 & 1.1 & 3.1 & -8.5 & -1.3 & 1.7 & 7.5 & 5.6 & -3.2 & 3.6 & 4.8 \\ -0.3 & -0.5 & 0.5 & 0 & 0.1 & 0.8 & 0.9 & 0.7 & 1 & 0.6 & -0.5 \\ 4.1 & 0.6 & 3.6 & -8.5 & -1.2 & 2.5 & 8.4 & 6.3 & -2.2 & 4.2 & 4.3 \end{bmatrix}$$

...and Vector multiplication

$$\begin{array}{l} + \\ = \end{array} \begin{bmatrix} 4.4 & 1.1 & 3.1 & -8.5 & -1.3 & 1.7 & 7.5 & 5.6 & -3.2 & 3.6 & 4.8 \\ -0.3 & -0.5 & 0.5 & 0 & 0.1 & 0.8 & 0.9 & 0.7 & 1 & 0.6 & -0.5 \\ 4.1 & 0.6 & 3.6 & -8.5 & -1.2 & 2.5 & 8.4 & 6.3 & -2.2 & 4.2 & 4.3 \end{bmatrix}$$
$$\begin{array}{l} \times \\ = \end{array} \begin{bmatrix} 4.4 & 1.1 & 3.1 & -8.5 & -1.3 & 1.7 & 7.5 & 5.6 & -3.2 & 3.6 & 4.8 \\ -0.3 & -0.5 & 0.5 & 0 & 0.1 & 0.8 & 0.9 & 0.7 & 1 & 0.6 & -0.5 \\ -1.32 & -0.55 & 1.55 & 0 & -0.13 & 1.36 & 6.75 & 3.92 & -3.2 & 2.16 & -2.4 \end{bmatrix}$$

An example

vector data operations: data operations done in parallel

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

vector data operations: data operations done in parallel

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

Loop:

1. LOAD a[i] -> Ra
2. LOAD b[i] -> Rb
3. ADD Ra, Rb -> Rc
4. STORE Rc -> c[i]
5. ADD i + 1 -> i

vector data operations: data operations done in parallel

```
void v add (float *c,
```

Loop:

1. LOADv4 a[i:i+3] -> Rva
2. LOADv4 b[i:i+3] -> Rvb
3. ADDv4 Rva, Rvb -> Rvc
4. STOREv4 Rvc -> c[i:i+3]
5. ADD i + 4 -> i

Loop:

1. LOAD a[i] -> Ra
2. LOAD b[i] -> Rb
3. ADD Ra, Rb -> Rc
4. STORE Rc -> c[i]
5. ADD i + 1 -> i

vector data operations:

data **We call this "vectorization"**

```
void v add (float *c,
```

Loop:

1. LOADv4 a[i:i+3] -> Rva
2. LOADv4 b[i:i+3] -> Rvb
3. ADDv4 Rva, Rvb -> Rvc
4. STOREv4 Rvc -> c[i:i+3]
5. ADD i + 4 -> i

Loop:

1. LOAD a[i] -> Ra
2. LOAD b[i] -> Rb
3. ADD Ra, Rb -> Rc
4. STORE Rc -> c[i]
5. ADD i + 1 -> i

vector data operations: data operations done in parallel

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

vector data operations: data operations done in parallel

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

PROBLEM:

This LOOP is NOT LEGAL to (automatically) VECTORIZE in C / C++ (without more information).

Arrays *not* really in the language

Pointers are, evil pointers!



Choice 1:
use a compiler switch for
auto-vectorization

(and *hope* it vectorizes)



Choice 2:

give your compiler hints

(and *hope* it vectorizes)

C99 *restrict* keyword

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

IVDEP (ignore assumed vector dependencies)

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



Choice 3:
code explicitly for vectors
(mandatory vectorization)

OpenMP* 4.0: #pragma omp simd

```
void v_add (float *c,  
           float *a,  
           float *b)  
{  
#pragma omp simd  
    for (int i=0; i<= MAX; i++)  
        c[i]=a[i]+b[i];  
}
```

OpenMP* 4.0: #pragma omp declare simd

```
#pragma omp declare simd
void v1_add (float *c,
            float *a,
            float *b)
{
    *c=*a+*b;
}
```

SIMD instruction intrinsics

```
void v_add (float *c,  
           float *a,  
           float *b)  
{  
    __m128* pSrc1 = (__m128*) a;  
    __m128* pSrc2 = (__m128*) b;  
    __m128* pDest = (__m128*) c;  
    for (int i=0; i<= MAX/4; i++)  
        *pDest++ = _mm_add_ps(*pSrc1++, *pSrc2++);  
}
```

Hard coded to 4 wide !

array operations (Cilk™ Plus)

```
void v_add (float *c,  
           float *a,  
           float *b)  
{  
    c [0:MAX] = a [0:MAX] + b [0:MAX] ;  
}
```

*Challenge: long vector slices
can cause cache issues; fix is to
keep vector slices short.*

Cilk™ Plus is supported
in Intel compilers, and
gcc (4.9).

vectorization solutions

1. auto-vectorization (use a compiler switch and hope it vectorizes)
 - sequential languages and practices gets in the way
2. give your compiler hints and hope it vectorizes
 - C99 restrict (implied in FORTRAN since 1956)
 - `#pragma ivdep`
3. code explicitly
 - OpenMP 4.0 `#pragma omp simd`
 - Cilk™ Plus array notations
 - SIMD instruction intrinsics
 - Kernels: OpenMP 4.0 `#pragma omp declare simd`; OpenCL; CUDA kernel functions

vectorization solutions

1. auto-vectorization (use a compiler switch and hope it vectorizes)
 - sequential languages and practices gets in the way
2. give your compiler hints and hope it vectorizes
 - C99 restrict (implied in FORTRAN since 1956)
 - #pragma ivdep
3. code explicitly
 - OpenMP 4.0 #pragma omp simd
 - Cilk™ Plus array notations
 - SIMD instruction intrinsics
 - Kernels: OpenMP 4.0 #pragma omp declare simd; OpenCL; CUDA kernel functions

Best at being
Reliable, predictable and portable

Explicit parallelism

parallelization

Try auto-parallel capability

- parallel (Linux* or OS X*)
- Qparallel (Windows*)

```
1 PROGRAM TEST
2 PARAMETER (N=10000000)
3 REAL A, C(N)
4 DO I = 1, N
5   A = 2 * I - 1
6   C(I) = SQRT(A)
7 ENDDO
8 PRINT*, N, C(1), C(N)
9 END
```

Or explicitly use...

Fortran directive (!DIR\$ PARALLEL)

C pragma (#pragma parallel)

Intel® Threading Building Blocks (TBB)

parallelization

Try auto-parallel capability:

- parallel (Linux or OS X*)
- Qparallel (Windows)

**Best at being
Reliable, predictable and portable**

Or explicitly use...

OpenMP

Intel® Threading Building Blocks (TBB)

```
c$OMP PARALLEL DO
  DO I=1,N B(I) = (A(I) + A(I-1)) / 2.0
  END DO
c$OMP END PARALLEL DO
```

OpenMP 4.0

Based on a proposal from Intel based on customer success with the Intel® Cilk™ Plus features in Intel compilers.

simd construct

Summary

The **simd** construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

OpenMP 4.0

Based on a proposal from Intel based on customer success with the Intel® Cilk™ Plus features in Intel compilers.

`simd construct`

```
#pragma omp simd reduction(+:val) reduction(+:val2)
for(int pos = 0; pos < RAND_N; pos++) {
    float callValue=
        expectedCall(Sval,Xval,MuByT,VBySqrtT,l_Random[pos]);
    val += callValue;
    val2 += callValue * callValue;
}
```

simd construct

(OpenMP 4.0)

YES - VECTORIZE THIS !!!

Summary

The **simd** construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

C/C++

```
#pragma omp simd [clause[[:] clause] ...] new-line  
for-loops
```

where *clause* is one of the following:

- safelen** (*length*)
- linear** (*list[:linear-step]*)
- aligned** (*list[:alignment]*)
- private** (*list*)
- lastprivate** (*list*)
- reduction** (*reduction-identifier: list*)
- collapse** (*n*)

The **simd** directive places restrictions on the structure of the associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (Section 2.6 on page 51).

C/C++

Fortran

```
!$omp simd [clause[[:] clause ...]  
do-loops  
[!$omp end simd]
```

where *clause* is one of the following:

- safelen** (*length*)
- linear** (*list[:linear-step]*)
- aligned** (*list[:alignment]*)
- private** (*list*)
- lastprivate** (*list*)
- reduction** (*reduction-identifier: list*)
- collapse** (*n*)

If an **end simd** directive is not specified, an **end simd** directive is assumed at the end of the *do-loops*.

All associated *do-loops* must be *do-constructs* as defined by the Fortran standard. If an **end simd** directive follows a *do-construct* in which several loop statements share a DO termination statement, then the directive can only be specified for the outermost of these DO statements.

Note: per the OpenMP standard, the "for-loop" must have canonical loop form.

Fortran

declare simd construct

(OpenMP 4.0)

Make VECTOR versions of this function.

Summary

The `declare simd` construct can be applied to a function (C, C++ and Fortran) or a subroutine (Fortran) to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop. The `declare simd` directive is a declarative directive. There may be multiple `declare simd` directives for a function (C, C++, Fortran) or subroutine (Fortran).

C/C++

```
#pragma omp declare simd [clause[[:] clause] ...] new-line  
[#pragma omp declare simd [clause[[:] clause] ...] new-line  
[...]  
    function definition or declaration
```

where *clause* is one of the following:

- `simdlen (length)`
- `linear (argument-list[:constant-linear-step])`
- `aligned (argument-list[:alignment])`
- `uniform (argument-list)`
- `inbranch`
- `notinbranch`

C/C++

Fortran

```
!$omp declare simd (proc-name) [clause[[:] clause] ...]
```

where *clause* is one of the following::

- `simdlen (length)`
- `linear (argument-list[:constant-linear-step])`
- `aligned (argument-list[:alignment])`
- `uniform (argument-list)`
- `inbranch`
- `notinbranch`

Fortran

Loop SIMD construct

(OpenMP 4.0)

Summary

The loop SIMD construct specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel by threads in the team.

Syntax

C/C++

```
#pragma omp for simd [clause[.] clause] ...] new-line
for-loops
```

where *clause* can be any of the clauses accepted by the **for** or **simd** directives with identical meanings and restrictions.

C/C++

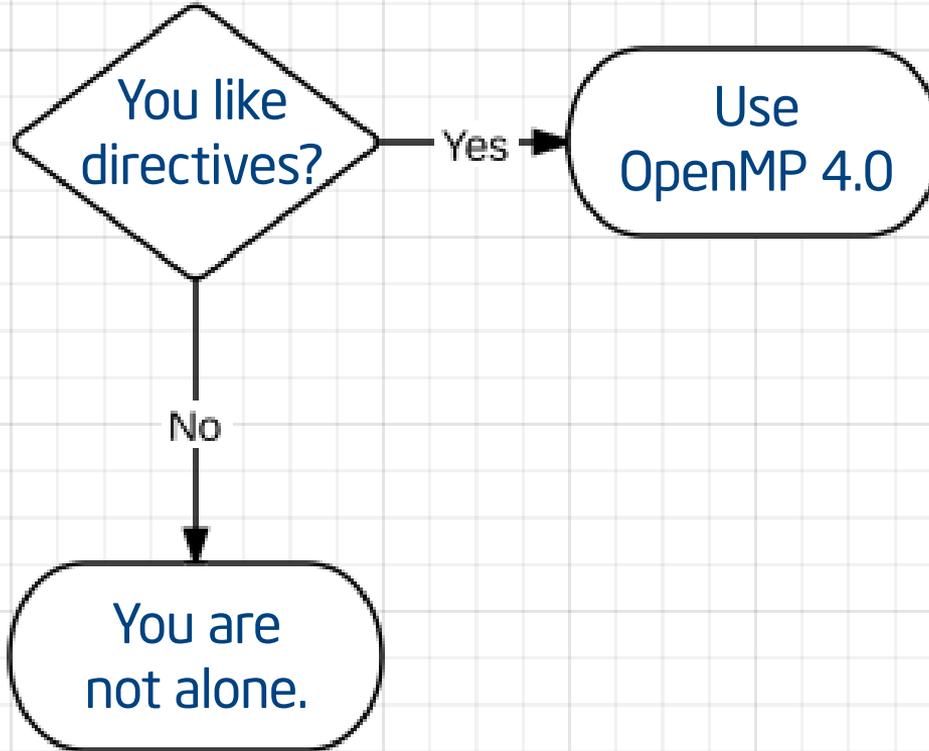
Fortran

```
!$omp do simd [clause[.] clause] ...]
do-loops
[!$omp end do simd [nowait]]
```

where *clause* can be any of the clauses accepted by the **simd** or **do** directives, with identical meanings and restrictions.

If an **end do simd** directive is not specified, an **end do simd** directive is assumed at the end of the do-loop.

Fortran



for your consideration:

Intel 15.0 Compilers (in beta now) support **keywords** as an alternative

- Keyword versions of SIMD pragmas added:
 `_Simd, _Safelen, _Reduction`
- `__intel_simd_lane()` intrinsic for SIMD enabled functions

Keywords / library interfaces being discussed for SIMD constructs in C and C++ standards

History of Intel vector instructions

Intel Instruction Set Vector Extensions from 1997-2008

1997	1998	1999	2004	2006	2007	2008
Intel® MMX™ technology	Intel® SSE	Intel® SSE2	Intel® SSE3	Intel® SSSE3	Intel® SSE4.1	Intel® SSE4.2
57 new instructions 64 bits Overload FP stack Integer only media extensions	70 new instructions 128 bits 4 single-precision vector FP scalar FP instructions cacheability instructions control & conversion instructions media extensions	144 new instructions 128 bits 2 double-precision vector FP 8/16/32/64 vector integer 128-bit integer memory & power management	13 new instructions 128 bits FP vector calculation x87 integer conversion 128-bit integer unaligned load thread sync.	32 new instructions 128 bits enhanced packed integer calculation	47 new instructions 128 bits packed integer calculation & conversion better vectorization by compiler load with streaming hint	7 new instructions 128 bits string (XML) processing POP-Count CRC32

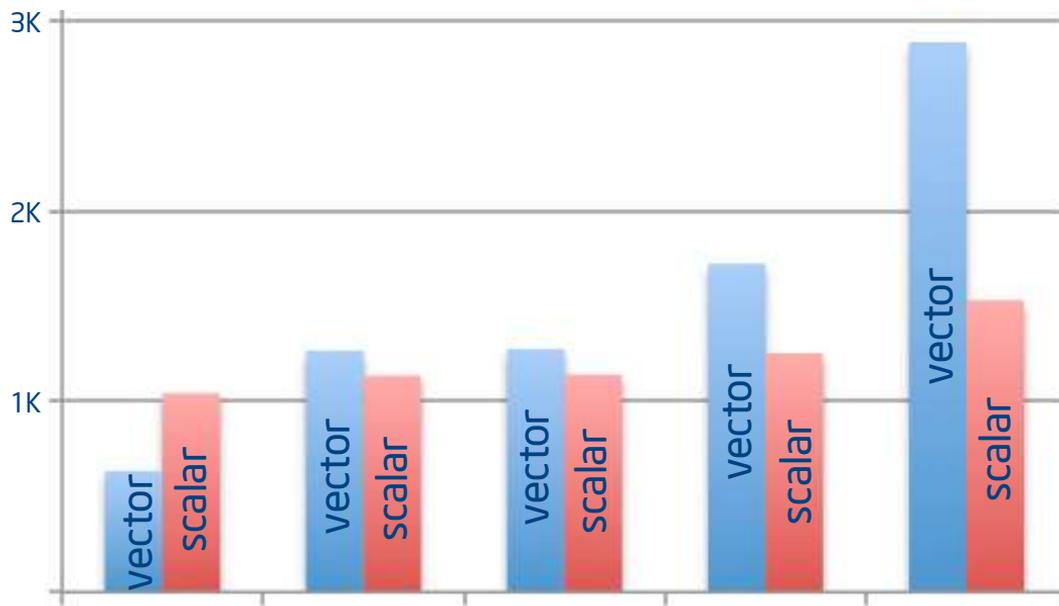
Intel Instruction Set Vector Extensions since 2011

2011	2011	2012	2013	TBD
Intel® AVX	Co-processor only 512	"AVX-1.5"	Intel® AVX-2	Intel® AVX-512
Promotion of 128 bit FP vector instructions to 256 bit	Coprocessor predecessor to AVX-512. New 512 bit vector instructions for MIC architecture, binary compt. not supported by processors - mostly source compatible with AVX-512	7 new instructions 16 bit FP support RDRAND ...	Promotion of integer instruction to 256 bit - FMA - Gather - TSX/RTM	Promotion of vector instructions to 512 bits Xeon Phi: FI, CDI, ERI, PFI Xeon: FI, CDI, BWI, DQI, VLE

Reinders blogs announced -
July 2013, and June 2014.

		width	Int.	SP	DP
1997	MMX	64	✓		
1999	SSE	128	✓	✓(x4)	
2001	SSE2	128	✓	✓	✓(x2)
2004	SSE3	128	✓	✓	✓
2006	SSSE 3	128	✓	✓	✓
2006	SSE 4.1	128	✓	✓	✓
2008	SSE 4.2	128	✓	✓	✓
2011	AVX	256	✓	✓(x8)	✓(x4)
2013	AVX2	256	✓	✓	✓
<i>future</i>	AVX-512	512	✓	✓(x16)	✓(x8)

Growth is in vector instructions



Disclaimer: Counting/attributing instructions is in inexact science. The exact numbers are easily debated, the trend is quite real regardless.

Motivation for AVX-512 Conflict Detection

Sparse computations are common in HPC, but hard to vectorize due to race conditions

```
for(i=0; i<16; i++) { A[B[i]]++; }
```

Consider the “histogram” problem:

```
index = vload &B[i]           // Load 16 B[i]
old_val = vgather A, index     // Grab A[B[i]]
new_val = vadd old_val, +1.0   // Compute new values
vscatter A, index, new_val     // Update A[B[i]]
```

Motivation for AVX-512 Conflict Detection

Sparse computations are common in HPC, but hard to vectorize due to race conditions

```
for(i=0; i<16; i++) { A[B[i]]++; }
```

Consider the “histogram” problem:

```
index = vload &B[i]           // Load 16 B[i]
old_val = vgather A, index     // Grab A[B[i]]
new_val = vadd old_val, +1.0   // Compute new values
vscatter A, index, new_val     // Update A[B[i]]
```

- Code above is wrong if any values within B[i] are duplicated
 - Only one update from the repeated index would be registered!
- A solution to the problem would be to avoid executing the sequence gather-op-scatter with vector of indexes that contain conflicts

Conflict Detection Instructions in AVX-512

improve vectorization!

VPCONFLICT instruction detects elements with previous conflicts in a vector of indexes

- Allows to generate a mask with a subset of elements that are guaranteed to be conflict free
- The computation loop can be re-executed with the remaining elements until all the indexes have been operated upon

CDI instr.

VPCONFLICT{D,Q} zmm1{k1}, zmm2/mem

VPBROADCASTM{W2D,B2Q} zmm1, k2

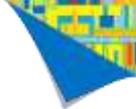
VPTSTNM{D,Q} k2{k1}, zmm2, zmm3/mem

VPLZCNT{D,Q} zmm1 {k1}, zmm2/mem

```
index = vload &B[i] // Load 16 B[i]
pending_elem = 0xFFFF; // all still remaining
do {
    curr_elem = get_conflict_free_subset(index, pending_elem)
    old_val = vgather {curr_elem} A, index // Grab A[B[i]]
    new_val = vadd old_val, +1.0 // Compute new values
    vscatter A {curr_elem}, index, new_val // Update A[B[i]]
    pending_elem = pending_elem ^ curr_elem // remove done idx
} while (pending_elem)
```

for illustration: this not even the fastest version

-vec-report



“Dear compiler, did you vectorize my loop?”

We heard your feedback.....

-**vec-report** output was hard to understand;

Messages were too cryptic to understand;

Information about one loop showing up at many places of report;

Was easy to be confused about multiple versions of one loop created by the compiler

**We couldn't do everything you asked,
but here are the
improvements made for 15.0 compiler (in 2014).**

**Expect more changes to come,
during beta and in future versions.**

Optimization Reports (since 2014)

- Old functionality implemented under **-opt-report**, **-vec-report**, **-openmp-report**, **-par-report** replaced by unified **-opt-report** compiler options
 - `[vec,openmp,par]-report` options deprecated and map to equivalent `opt-report-phase`
- Can still select phase with **-opt-report-phase** option. For example, to only get vectorization reports, use **-opt-report-phase=vec**
- Output now defaults to a `<name>.optrpt` file where `<name>` corresponds to the output object name. This can be changed with **-opt-report-file=[<name>|stdout|stderr]**
- Windows*: **/Qopt-report**, **/Qopt-report-phase=<phase>** etc.
 - Optimization report integration with Microsoft* Visual Studio planned to appear in beta update 1

Summary

We need to embrace explicit vectorization in our programming.

It's messy today

Vectorization yesterday

DO 1 k = 1,n
1 A(k) = B(k) + C(k)

K=1

Ld C(1)

Ld B(1)

Add

St A(1)

K=2

Ld C(2)

Ld B(2)

Add

St A(2)

K=1..2

Ld C(1)

Ld C(2)

Ld B(1)

Ld B(2)

Add

Add

St A(1)

St A(2)

Scalar code

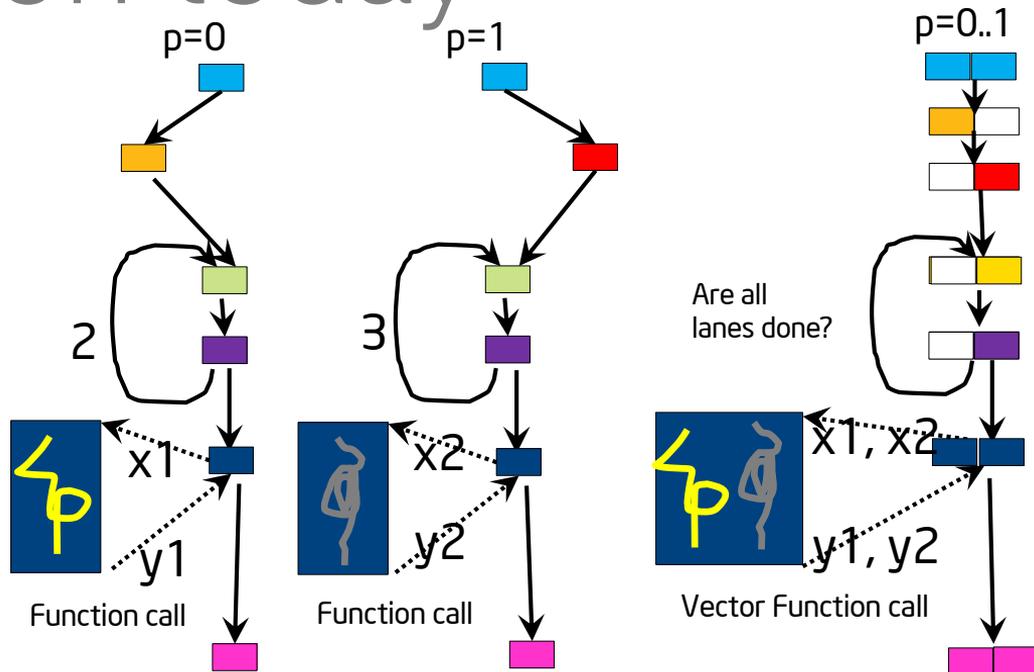
Vector code

Vector code generation was straightforward
Emphasis on analysis and disambiguation

Vectorization today

```
#pragma omp simd reduction(+:....)
for(p=0; p<N; p++) {
  // Blue work
  if(...) {
    // Green work
  } else {
    // Red work
  }
  while(...) {
    // Gold work
    // Purple work
  }
  y = foo(x);
  // Pink work
}
```

Two fundamental problems
Data divergence
Control divergence



Vector code generation has become a more difficult problem
Increasing need for user guided explicit vectorization
Explicit vectorization maps threaded execution to simd hardware



```
#pragma omp simd
```

```
for (x = 0; x < w; x++) {
```

```
  for (v = 0; v < nsubsamples; v++) {
```

```
    for (u = 0; u < nsubsamples; u++) {
```

```
      float px = (x + (u / (float)nsubsamples) - (w / 2.0f)) / (w / 2.0f);
```

```
      Ray ray; Isect isect;
```

```
      ....
```

```
      ray.dir.x = px;
```

```
      ....
```

```
      vnormalize(&(ray.dir));
```

```
      .....
```

```
      ray_sphere_intersect(&isect, &ray, &spheres[0]);
```

```
      .....
```

```
      ray_plane_intersect (&isect, &ray, &plane);
```

```
      if (isect.hit) {
```

```
        vec col;
```

```
        ambient_occlusion_simd(&col, &isect);
```

```
        fimg[3 * (y * w + x) + 0] += col.x;
```

```
      .....
```

```
    }
```

```
  }
```

```
}
```

```
}
```

Loops

Function calls

Conditionals

Conditional Function calls

Motivational Example

```
//foo.c
float in_vals[];
for(int x = 0; x < Width; ++x) {
    count[x] = lednam(in_vals[x]);
}
```

```
//bar.c
int lednam(float c)
{    // Compute n >= 0 such that c^n > LIMIT
    float z = 1.0f;
    int iters = 0;
    while (z < LIMIT) {
        z = z * c; iters++;
    }
    return iters;
}
```

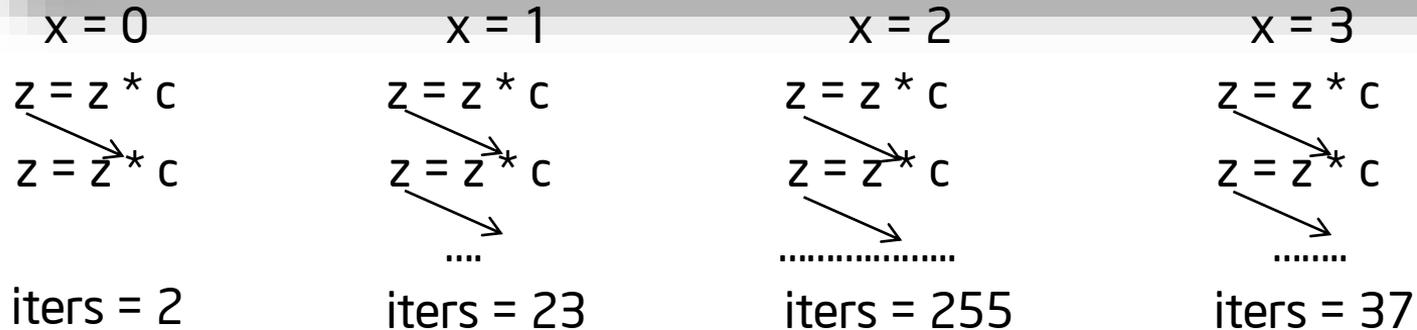
What are the simplest changes required for the program to utilize today's multicore and simd hardware?

```
float in_vals[];
```

```
for(int x = 0; x < Width; ++x) {  
    count[x] = lednam(in_vals[x]);  
}
```

```
#pragma omp declare simd
```

```
int lednam(float c)  
{  
    // Compute n >= 0 such that c^n > LIMIT  
    float z = 1.0f; int iters = 0;  
    while (z < LIMIT) {  
        z = z * c; iters++;  
    }  
    return iters;  
}
```

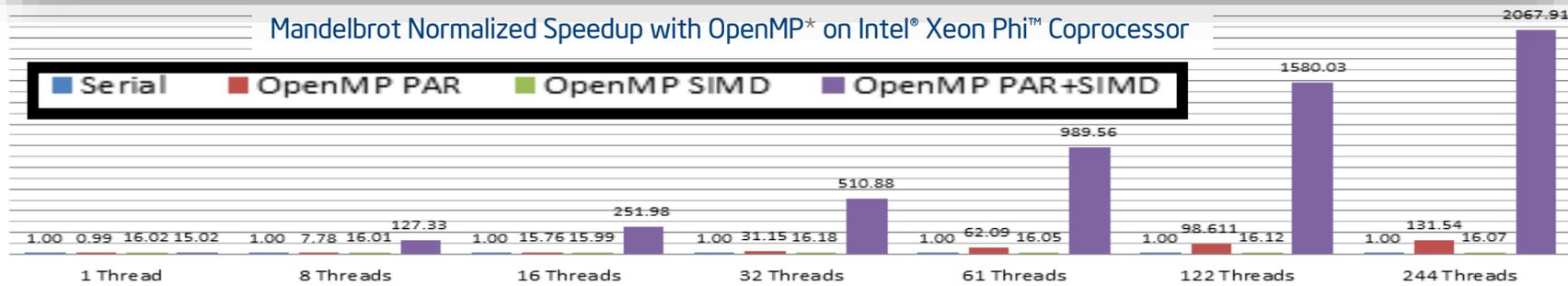


Mandelbrot

```
#pragma omp parallel for
for (int y = 0; y < ImageHeight; ++y) {
    #pragma omp simd
    for (int x = 0; x < ImageWidth; ++x) {
        count[y][x] = mandel(in_vals[y][x]);
    }
}
```

```
#pragma omp declare simd
int mandel(fcomplex c)
{ // Computes number of iterations for c to escape
  fcomplex z = c;
  for (int iters=0; (cabsf(z) < 2.0f) && (iters < LIMIT); iters++) {
      z = z * z + c;
  }
  return iters;
}
```

Mandelbrot Normalized Speedup with OpenMP* on Intel® Xeon Phi™ Coprocessor

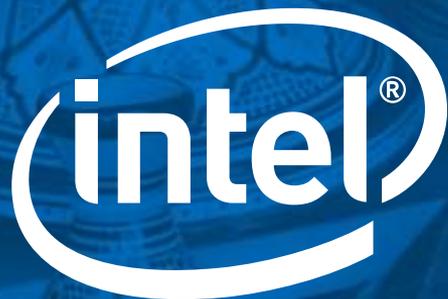


Summary

We need to embrace explicit vectorization in our programming.

But, generally use parallelism first (tasks, threads, MPI, etc.)

Questions?



james.r.reinders@intel.com

James Reinders. Parallel Programming Evangelist. Intel.

James is involved in multiple engineering, research and educational efforts to increase use of parallel programming throughout the industry. He joined Intel Corporation in 1989, and has contributed to numerous projects including the world's first TeraFLOP/s supercomputer (ASCI Red) and the world's first TeraFLOP/s microprocessor (Intel® Xeon Phi™ coprocessor). James been an author on numerous technical books, including VTune™ Performance Analyzer Essentials (Intel Press, 2005), Intel® Threading Building Blocks (O'Reilly Media, 2007), Structured Parallel Programming (Morgan Kaufmann, 2012), Intel® Xeon Phi™ Coprocessor High Performance Programming (Morgan Kaufmann, 2013), Multithreading for Visual Effects (A K Peters/CRC Press, 2014), High Performance Parallelism Pearls Volume 1 (Morgan Kaufmann, Nov. 2014), and High Performance Parallelism Pearls Volume 2 (Morgan Kaufmann, Aug. 2015). James is working on a refresh of both the Xeon Phi™ book (original Feb. 2013, revised with KNL information by mid-2016) and a refresh of the TBB book (original June 2007, revised by 2017).



Legal Disclaimer & Optimization Notice

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 © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo 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