HPC codes **modernization** using **vector** and threading **parallelism**

Zakhar A. Matveev, PhD,

Intel Russia, Intel Software and Services Group

July‘ 2015, CERN OpenLab

Acknowledgments

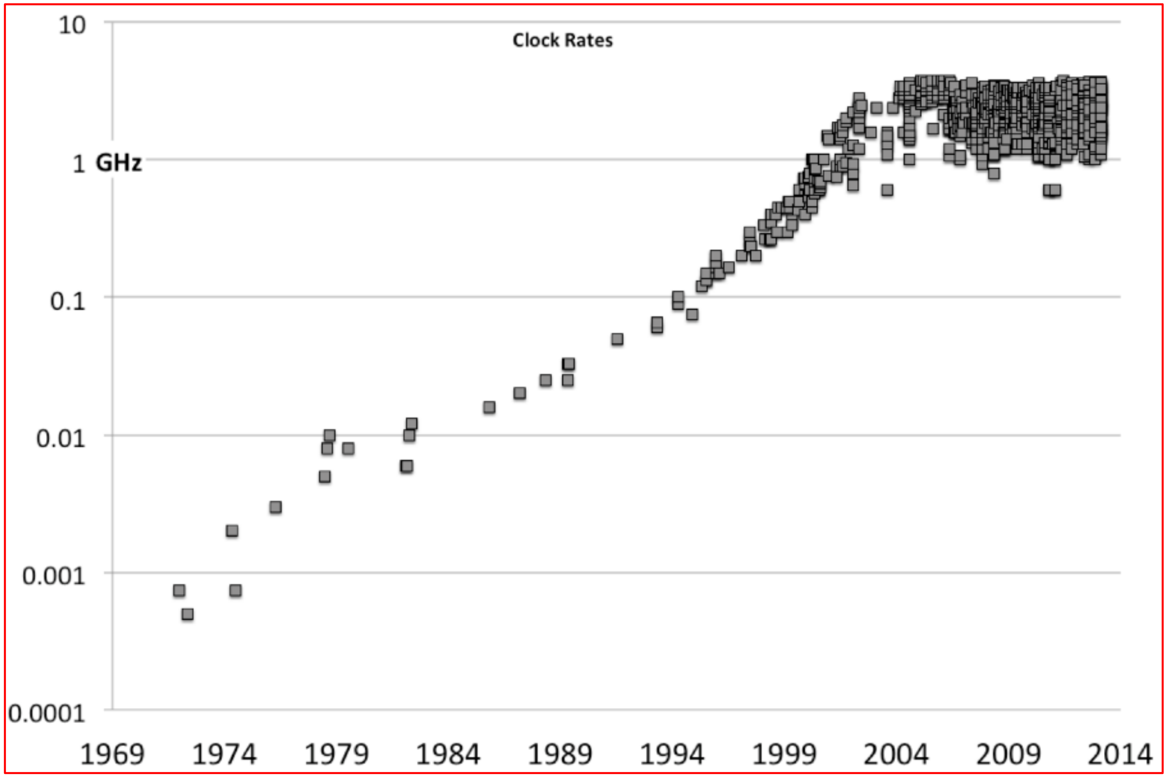
This foildeck re-uses some of content created by:

- Kevin O’Leary, Dick Kaiser, Stephen Blair-Chapel
- James Reinders and Arch D. Robison
- Intel® Compiler architects
- Geoff Lowney and Victor Lee (SIMD conference keynotes)

Motivation

The “Free Lunch” is over, really

Processor clock rate growth halted around 2005



Source: © 2014, James Reinders, Intel, used with permission

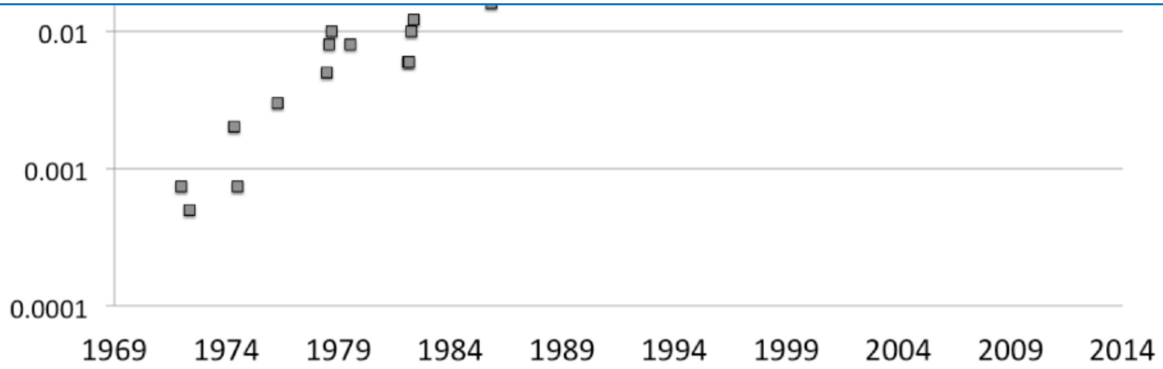
The “Free Lunch” is over, really

Processor clock rate growth halted before 2005

Real message:

Software has to be changed

To keep performance growth curve and to effectively exploit hardware.



Source: © 2014, James Reinders, Intel, used with permission

The “Free Lunch” is over, really

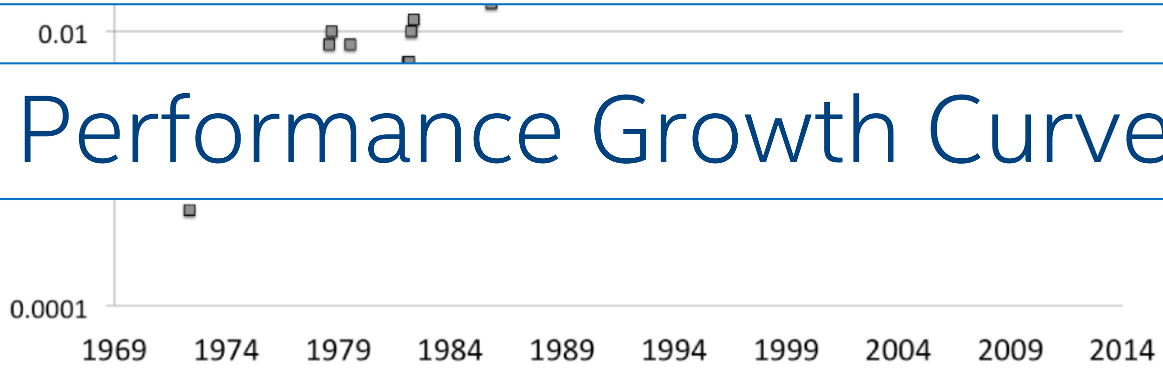
Processor clock rate growth halted before 2005

Real message:

Software has to be changed

To keep performance growth curve and to effectively exploit hardware.

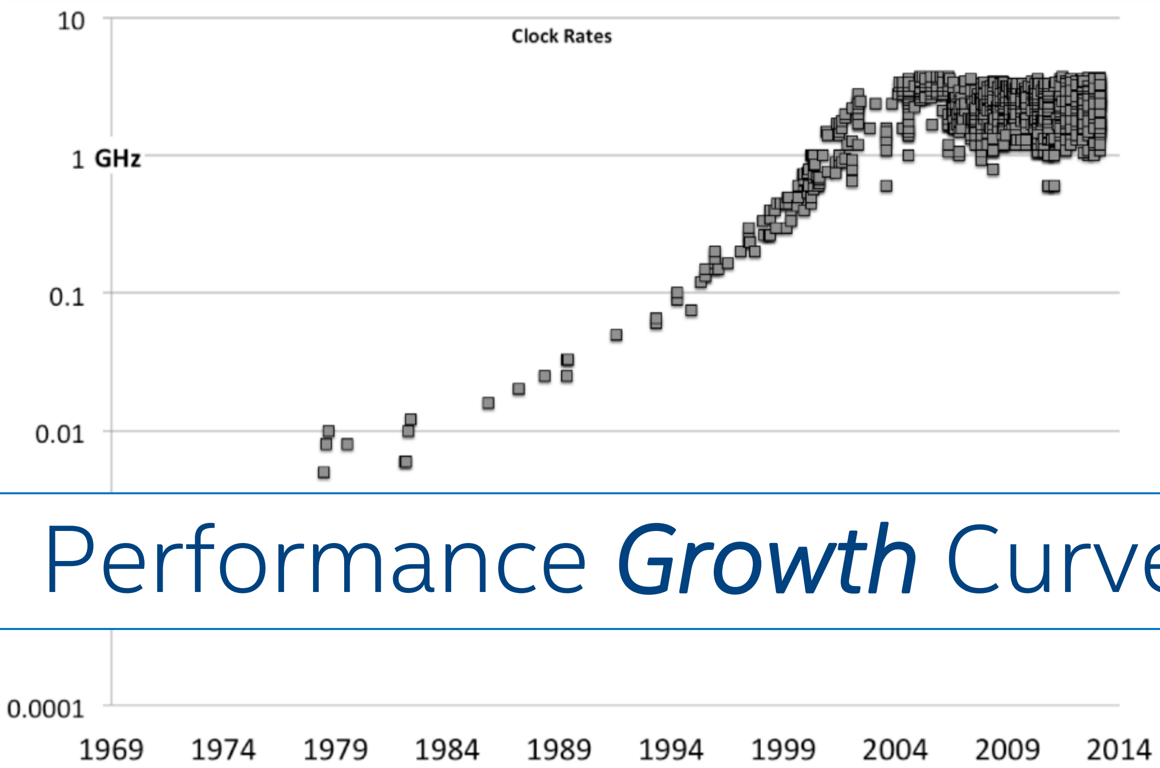
Performance Growth Curve?



Source: © 2014, James Reinders, Intel, used with permission

The “Free Lunch” is over, really

Processor clock rate growth halted around 2005



Performance ***Growth*** Curve???

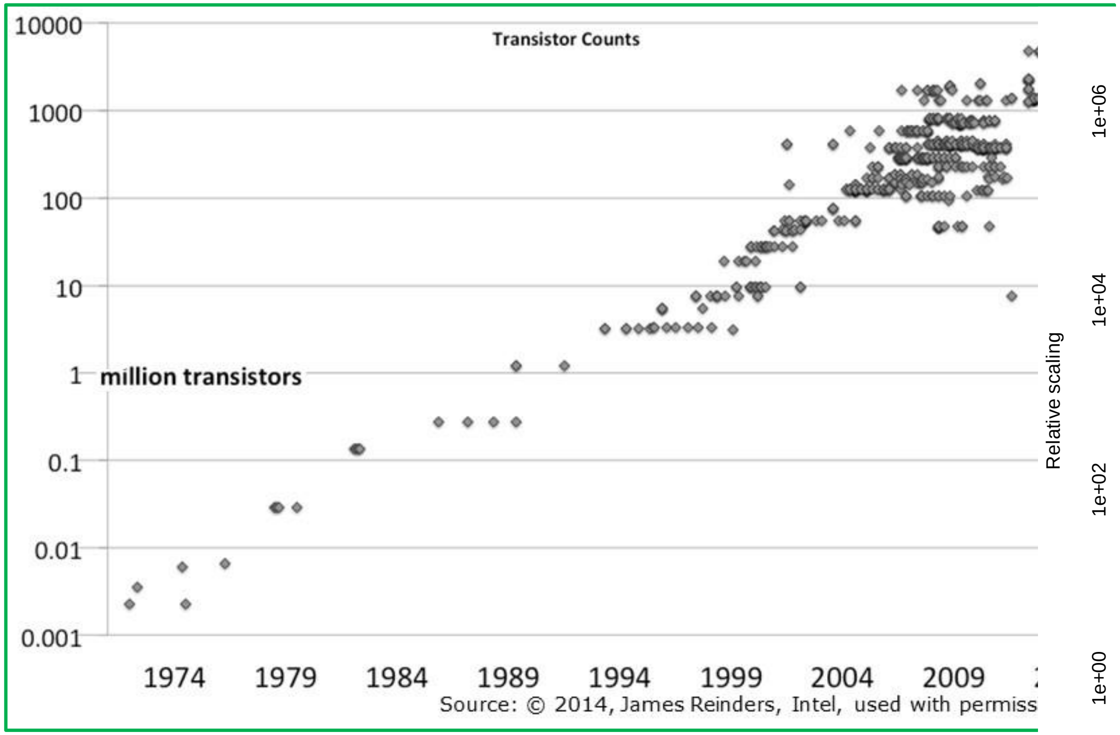
Source: © 2014, James Reinders, Intel, used with permission

Moore's Law Is STILL Going Strong

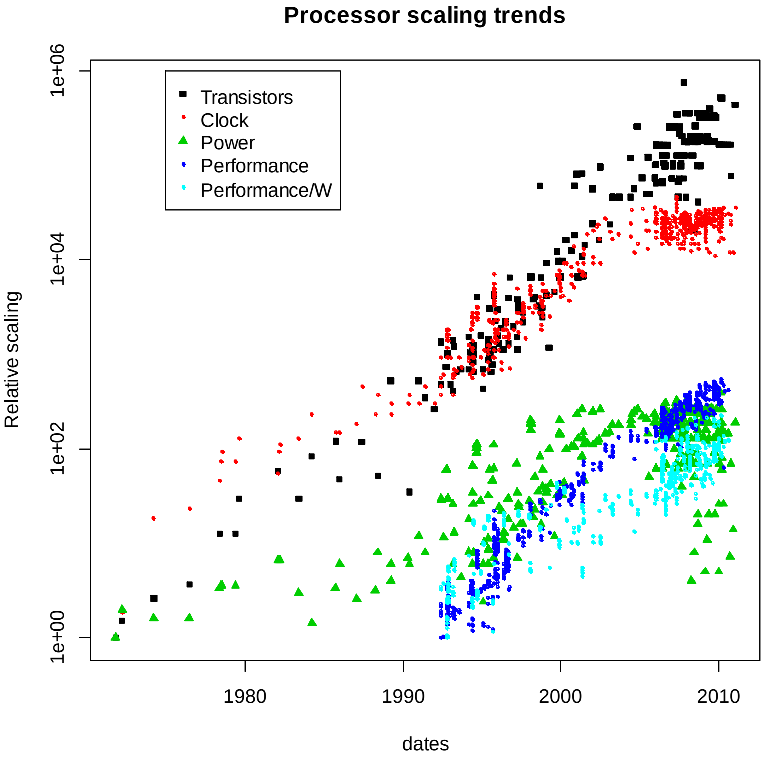
Hardware performance **continues to grow** exponentially

> *"We think we can continue Moore's Law for at least another 10 years."*

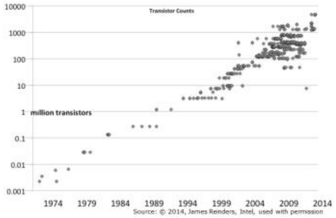
Intel Senior Fellow Mark Bohr, **2015**



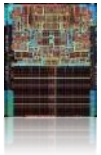
Source: © 2014, James Reinders, Intel, used with permiss

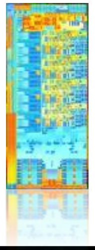
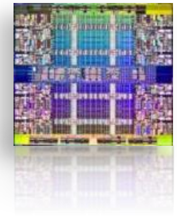


More cores . More Threads . Wider vectors



Source: © 2014, James Reinders, Intel, used with permission



	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge EP	Intel® Xeon® processor code-named Ivy Bridge EP	Intel® Xeon® processor code-named Haswell EP	Intel® Xeon Phi™ coprocessor Knights Corner	Intel® Xeon Phi™ processor & coprocessor Knights Landing[1]
Core(s)	1	2	4	6	8	12	18	61	60+
Threads	2	2	8	12	16	24	36	244	
SIMD Width	128	128	128	128	256	256	256	512	

*Product specification for launched and shipped products available on ark.intel.com.

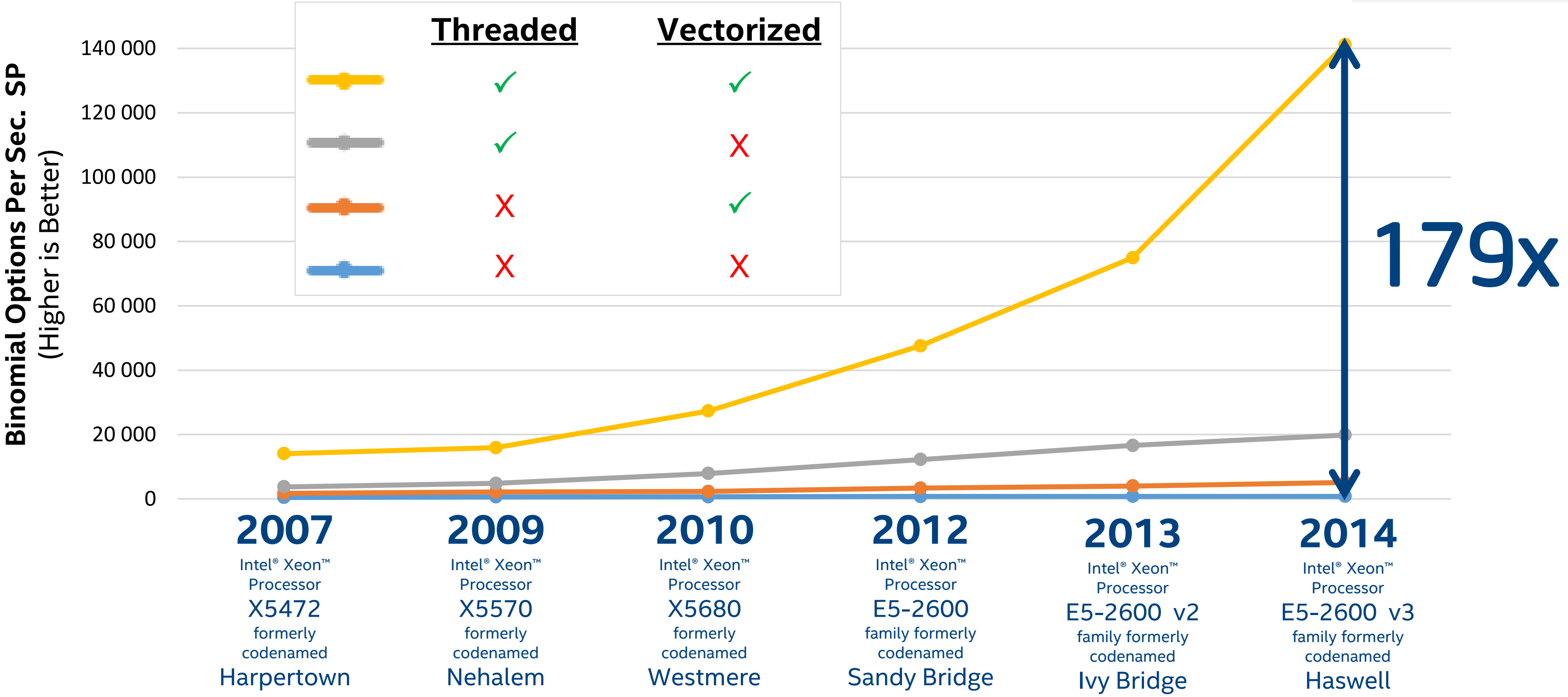
[1] Not launched or in planning.

High Performance Software has to be changed to exploit both:

- Threading parallelism
- Vector data parallelism

Untapped Potential Can Be Huge!

Configurations for Binomial Options SP
at the end
of this presentation



4C	4C	6C	8C	12C	14C	60C+
4 * SP	4*SP	4*SP	8*SP	8*SP	16*SP	32*SP

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. For more information go to http://www.intel.com/performance

Multi-Threading and Vectorization = Huge Potential

Let's do some accounting..

Current Intel Xeon processor

12 cores

x 2 hyper-threads

x 8 lane (SP) vector unit per thread (another x2 for FMA)

= **384**-folds parallelism for single socket

Intel Many Integrated Core architecture

> 60 cores

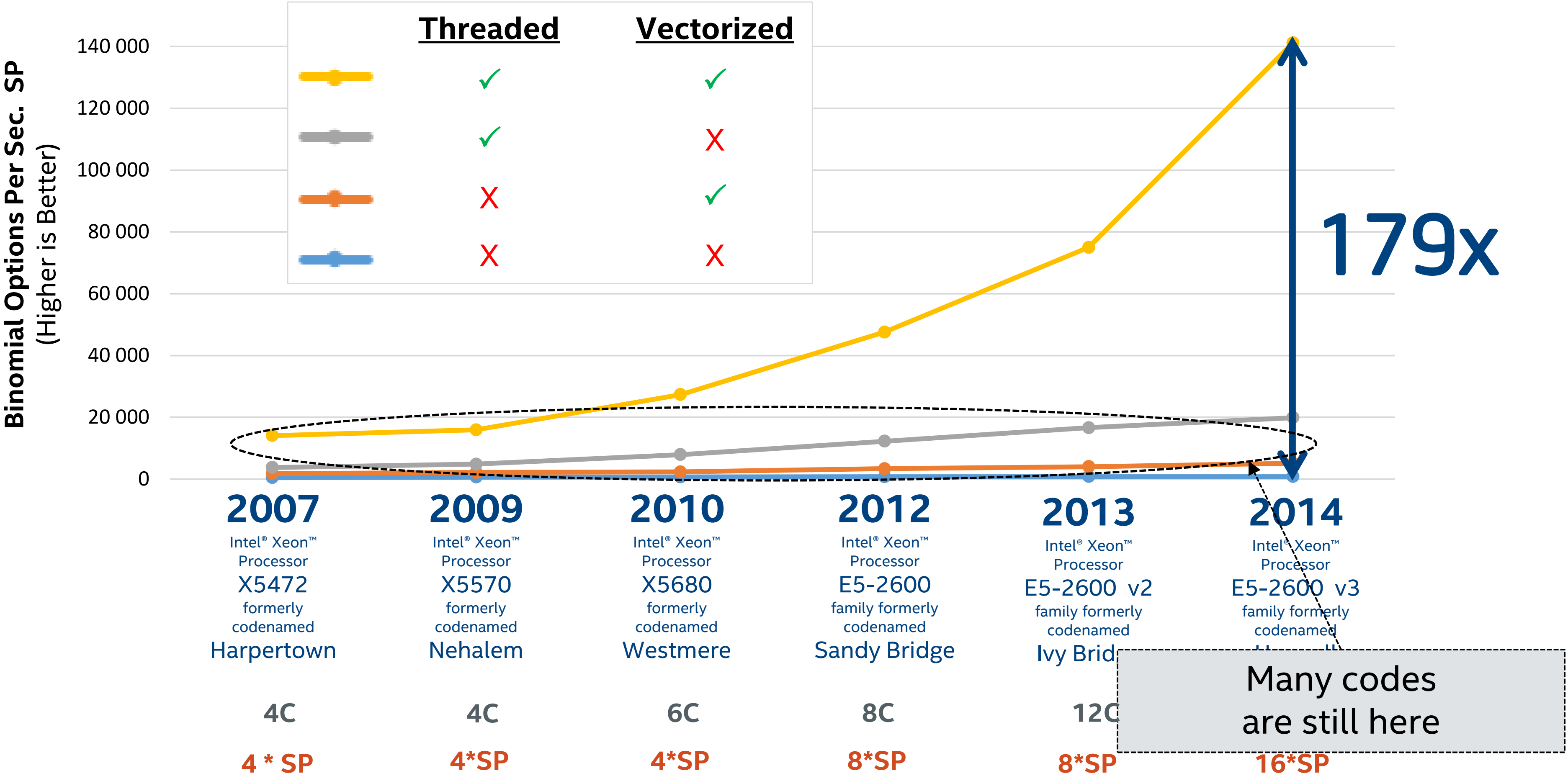
x ?? independent threads per core

x 16 lane (SP) vector unit per thread (x2 for FMA)

= parallel heaven

The Gap ~~Untapped Potential~~ Can Be Huge!

Threaded + Vectorized can be much faster than either one alone



	Threaded	Vectorized
🟨	✓	✓
⬜	✓	X
🟧	X	✓
🟦	X	X

Many codes are still here

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. For more information go to http://www.intel.com/performance

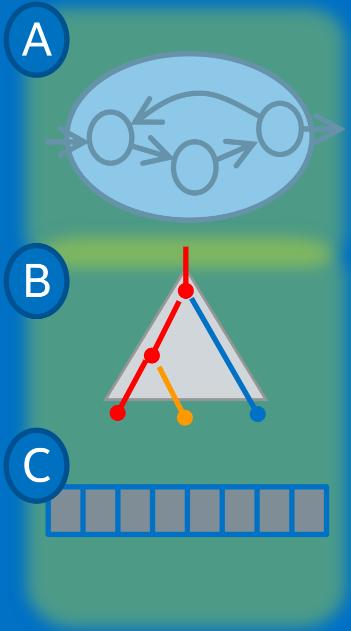
Don’t use a single Vector lane/thread!

Un-vectorized and un-threaded software will under perform

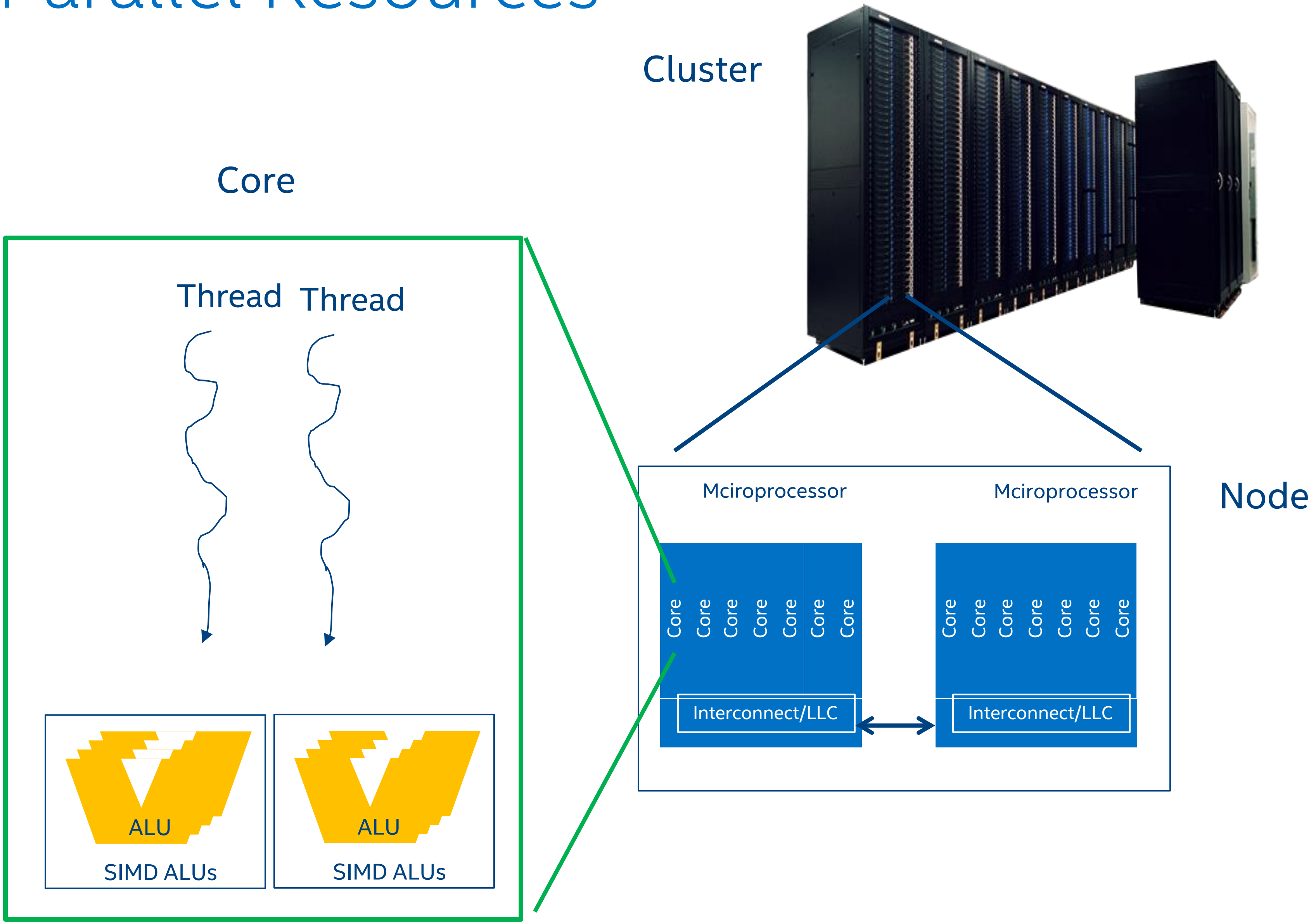
Permission to Design for All Lanes

Threading and Vectorization needed to fully utilize modern hardware

Parallel Programming
for multi-core and manycore processors



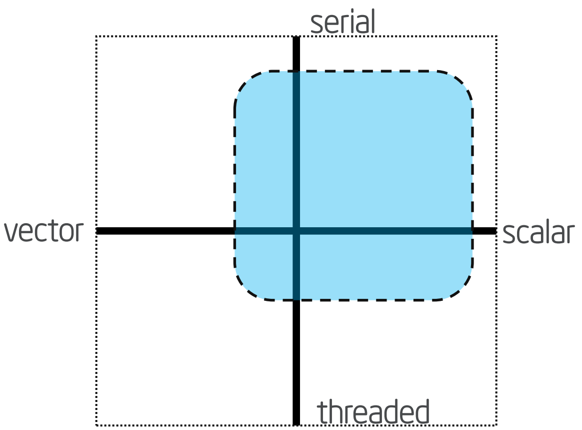
Parallel Resources



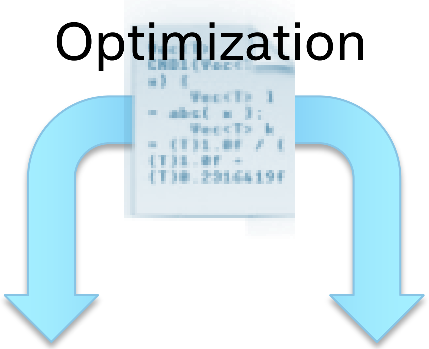
Cluster → Node → Sockets → Processor/Co-processor → Core → Thread → SIMD (Vector)

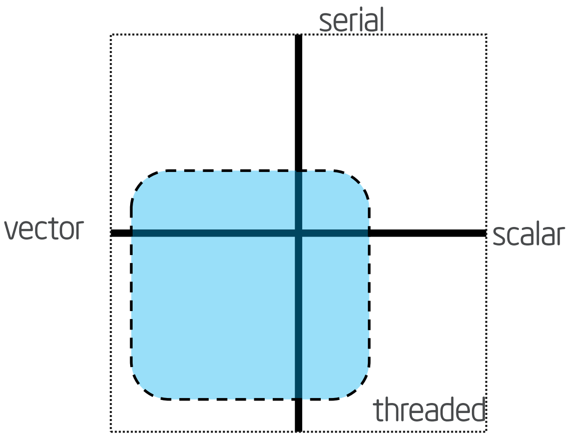
Next generation Intel Xeon Phi (Knights Landing)

Targeted for Highly-Vectorizable, Parallel Apps



Single Source Code Optimization





Most Commonly Used Parallel Processor*

Parallel, Fast Serial

Multicore + Vector

Leadership Today and Tomorrow

intel inside XEON + intel inside XEON PHI

Optimized for Highly-Vectorizable Parallel Apps

Many Core

Support for 512 bit vectors

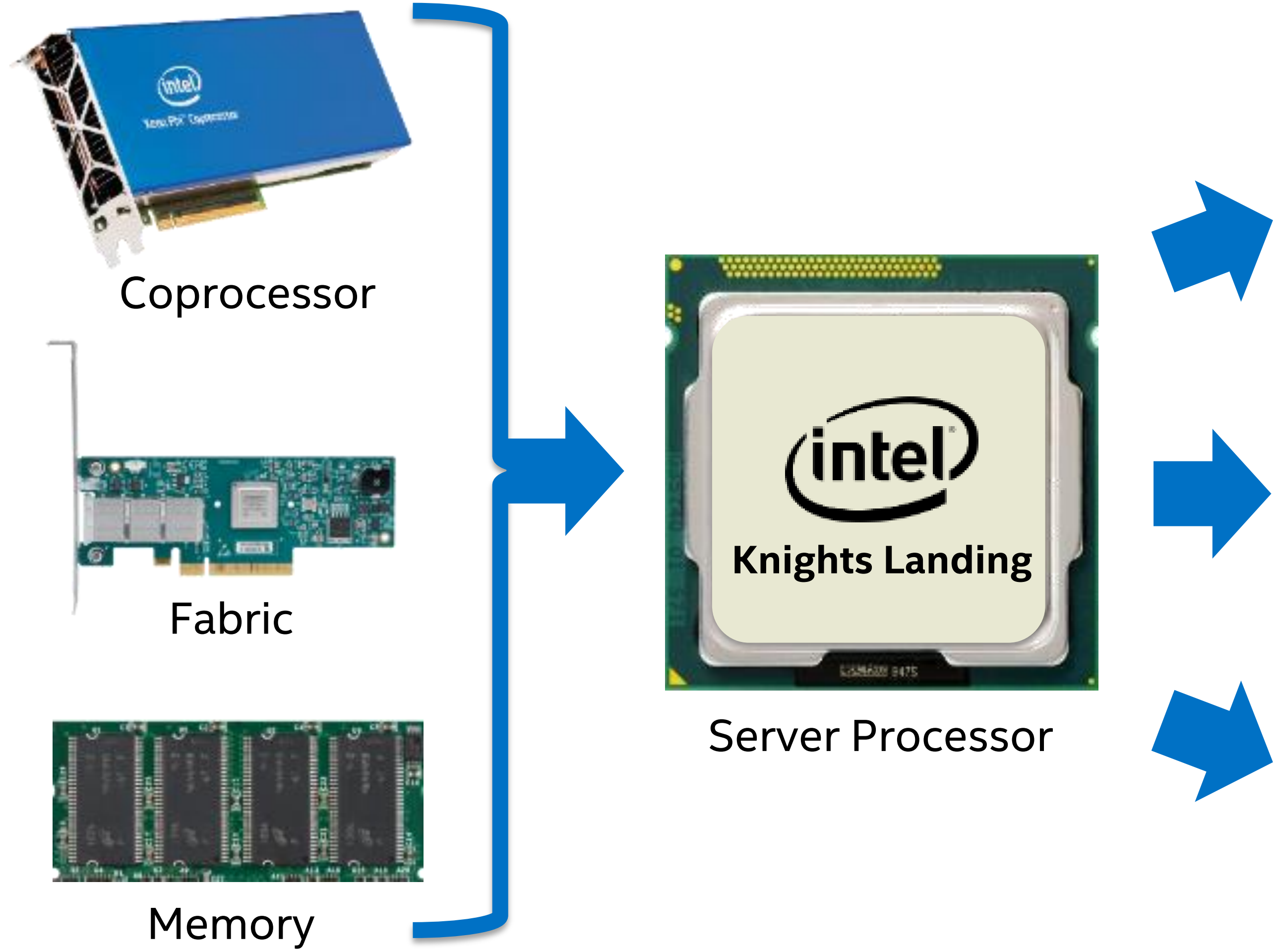
Higher memory bandwidth

Common SW programming

*Based on highest volume CPU in the IDC HPC Qview Q1'13

A Paradigm Shift for Highly-Parallel

Server Processor and **Integration** are Keys to Future



Memory Bandwidth
~500 GB/s STREAM

Memory Capacity
Over 25x KNC*

Resiliency
Systems scalable to >100 PF

Power Efficiency
Over 25% better than card[1]

I/O
Up to 100 GB/s with int fabric

Cost
Less costly than discrete parts[1]

Flexibility
Limitless configurations

Density
3+ KNL with fabric in 1U[3]

**Comparison to 1st Generation Intel® Xeon Phi™ 7120P Coprocessor (formerly codenamed Knights Corner)*

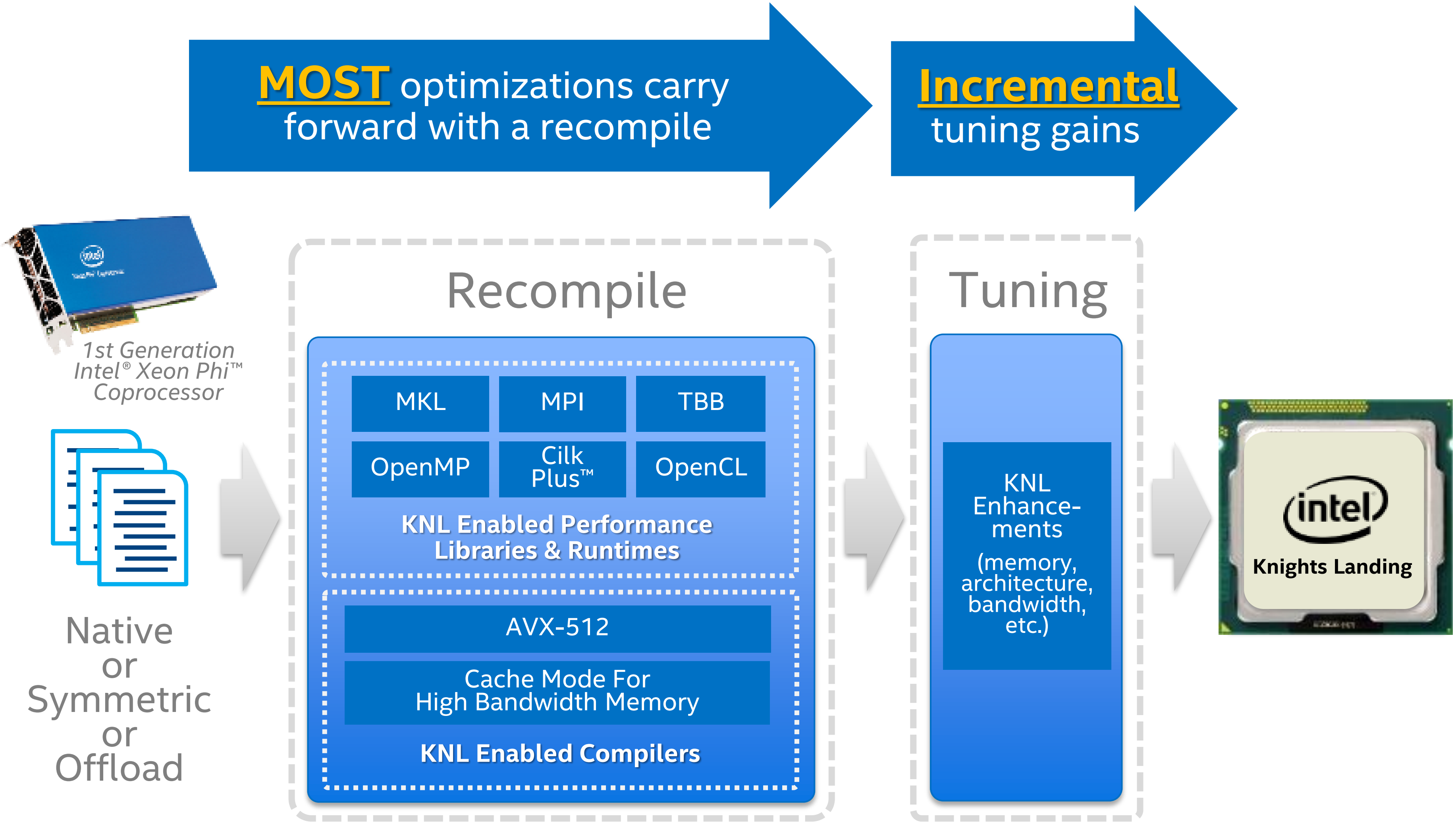
[1] Results based on internal Intel analysis using estimated power consumption and projected component pricing in the 2015 timeframe. This analysis is provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

[2] Comparison to a discrete Knights Landing processor and discrete fabric component.

[3] Theoretical density for air-cooled system; other cooling solutions and configurations will enable lower or higher density.

Today's Parallel Investment Carries Forward

Sustained threading, vectorization, cache-blocking and more



MOST optimizations carry forward with a recompile

Incremental tuning gains

1st Generation Intel® Xeon Phi™ Coprocessor

Native or Symmetric or Offload

Recompile

MKL	MPI	TBB
OpenMP	Cilk Plus™	OpenCL

KNL Enabled Performance Libraries & Runtimes

AVX-512

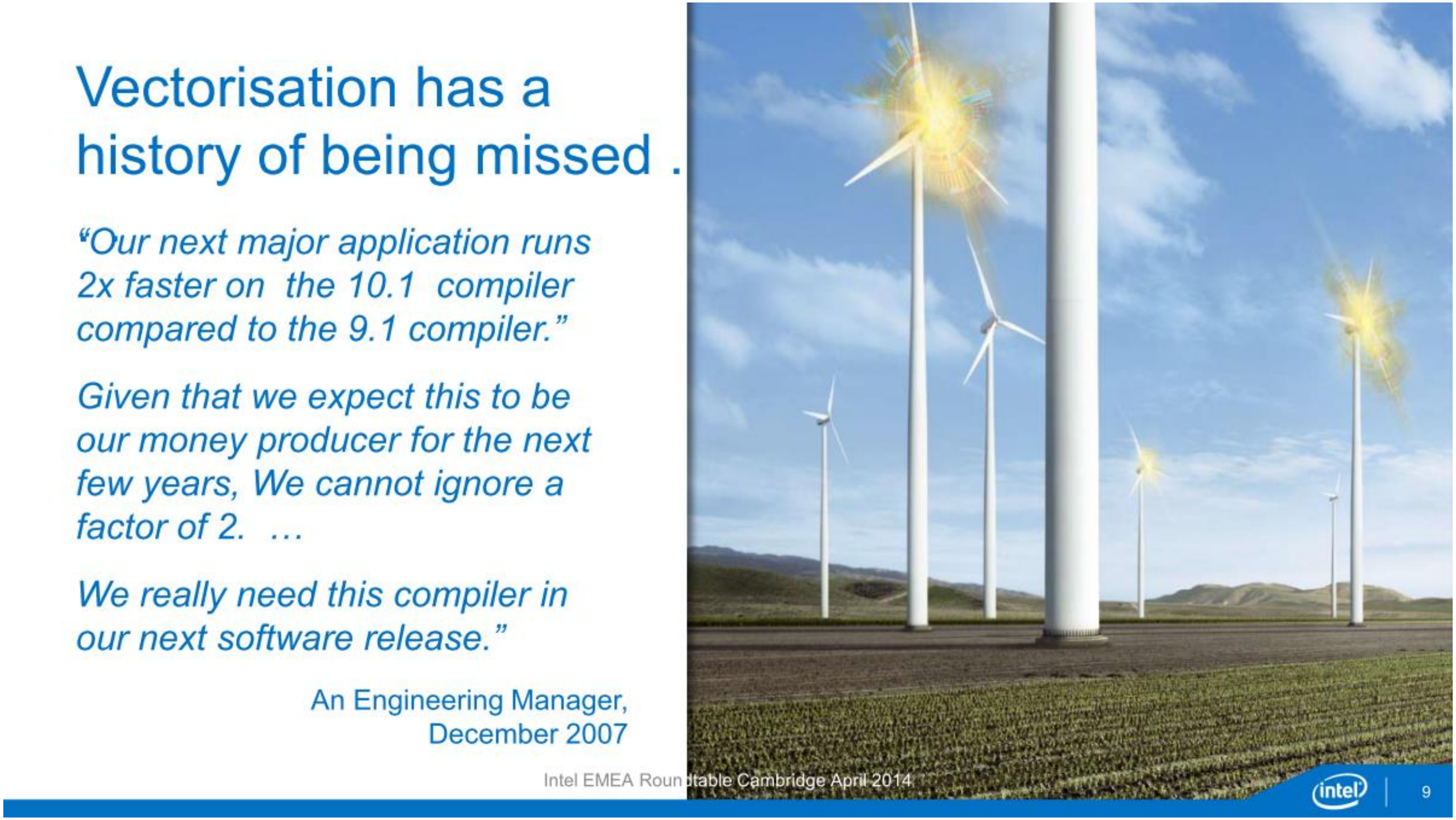
Cache Mode For High Bandwidth Memory

KNL Enabled Compilers

Tuning

KNL Enhancements (memory, architecture, bandwidth, etc.)

Recompile?



Vectorisation has a history of being missed .

“Our next major application runs 2x faster on the 10.1 compiler compared to the 9.1 compiler.”

Given that we expect this to be our money producer for the next few years, We cannot ignore a factor of 2. …

We really need this compiler in our next software release.”

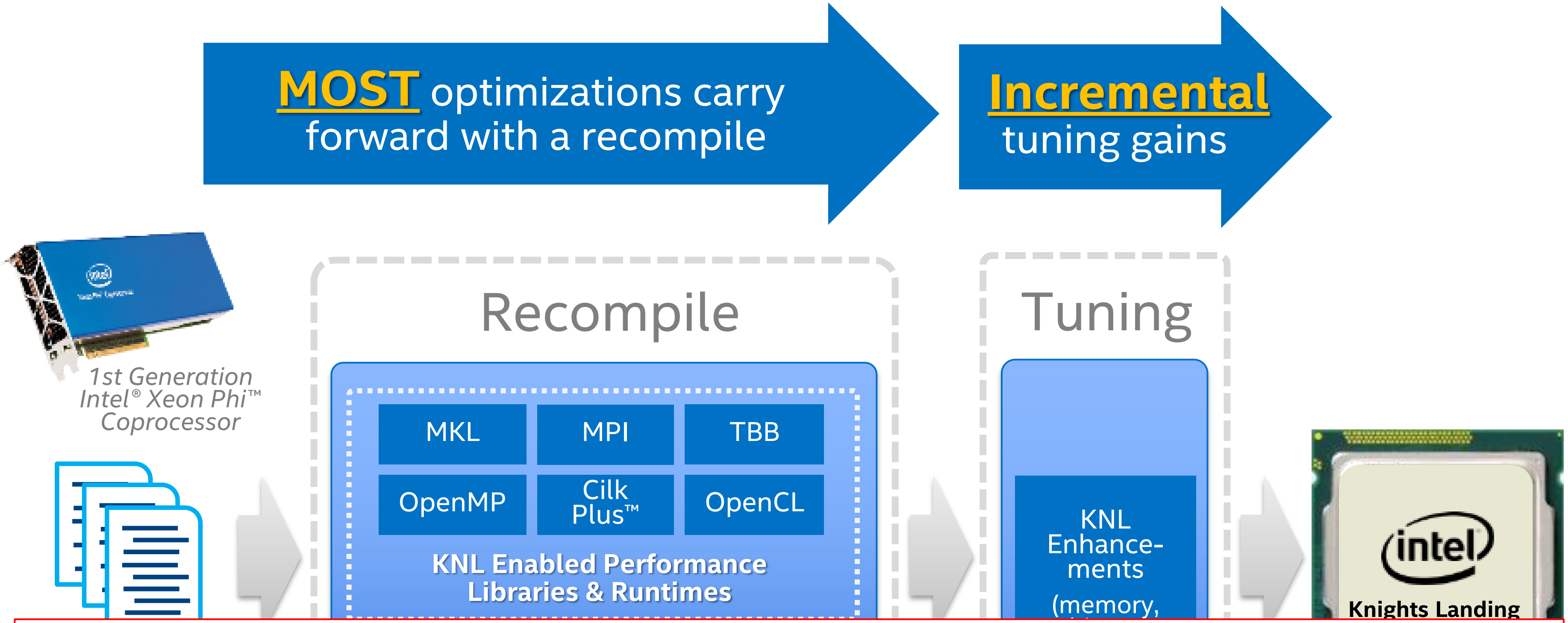
An Engineering Manager,
December 2007

Intel EMEA Roundtable Cambridge April 2014

9

Today's Parallel Investment Carries Forward

Sustained threading, vectorization, cache-blocking and more

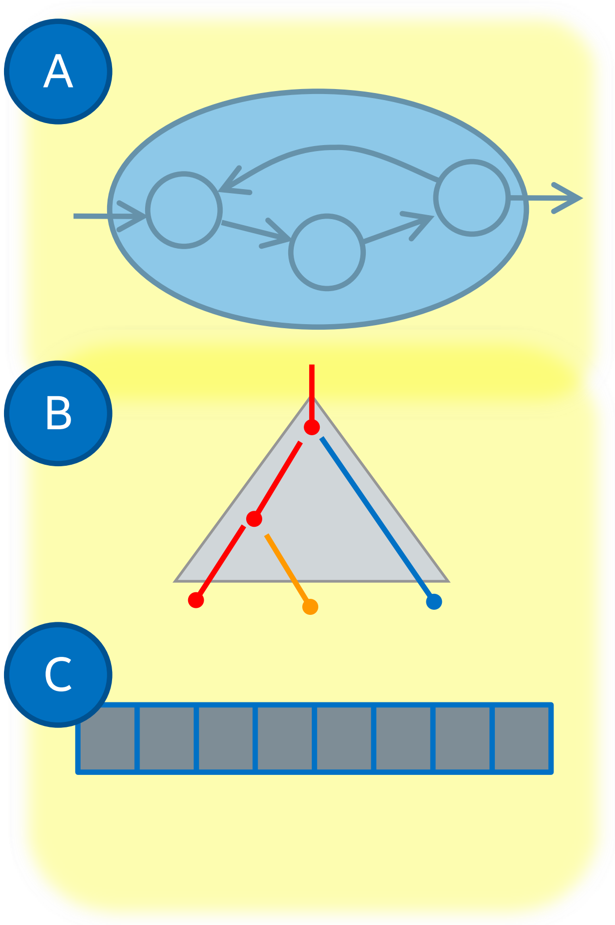


MOST optimizations carry forward with a recompile
Incremental tuning gains
1st Generation Intel® Xeon Phi™ Coprocessor
Recompile
MKL
MPI
TBB
OpenMP
Cilk Plus™
OpenCL
KNL Enabled Performance Libraries & Runtimes
Tuning
KNL Enhance-ments
(memory,
intel
Knights Landing

Recompile and tune "recipe" will only work if "parallel investment" has been made ***already.***

Software has to be changed.

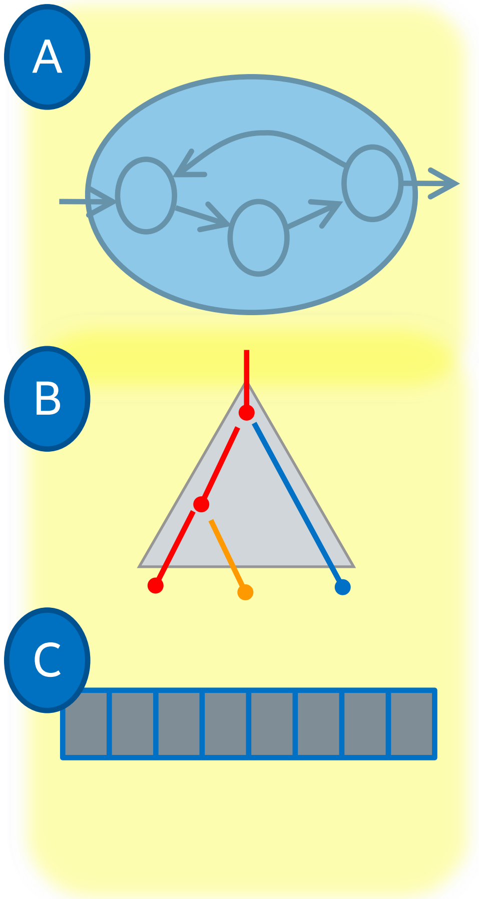
How could we program these parallel machines?



“Three Layer Cake”

“abstracts” common hybrid parallelism programming approaches

How could we program these parallel machines?



Parallelism type

A – Message Passing

B – Fork-Join

C- SIMD

Exploiting hardware* :

A: exploit multiple **nodes**, distributed memory

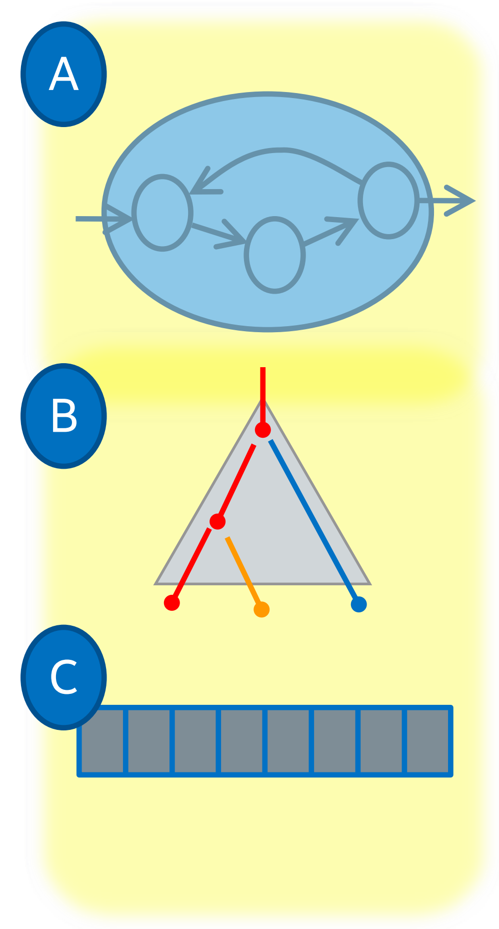
B – exploit multiple **cores**, hardware threads

C- exploit **vector units**

* - alternate hardware mappings also possible

How could we program these parallel machines?

Implementing *the Cake*



Programming models

A – MPI, tbb::flow, PGAS

B – **OpenMP4.x,** Cilk Plus, TBB

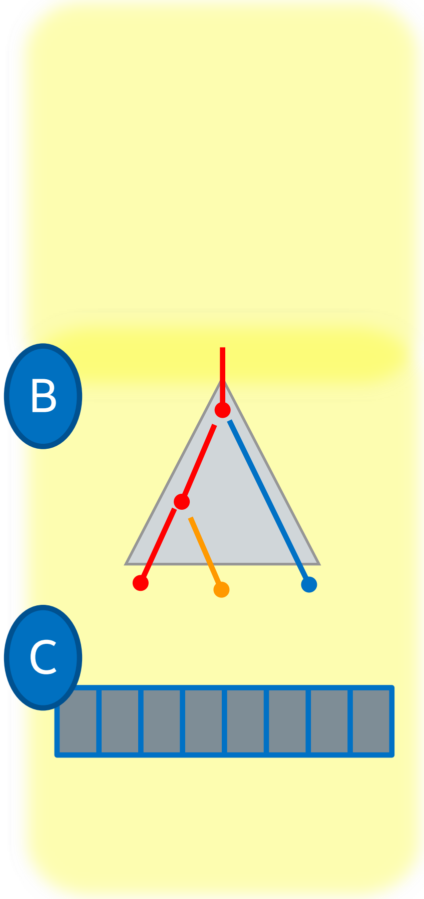
C - **OpenMP4.x,** Cilk Plus

Software tools

Cluster Edition

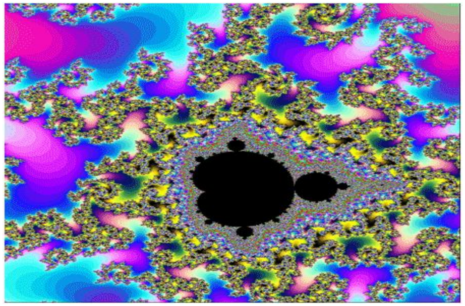
Professional Edition

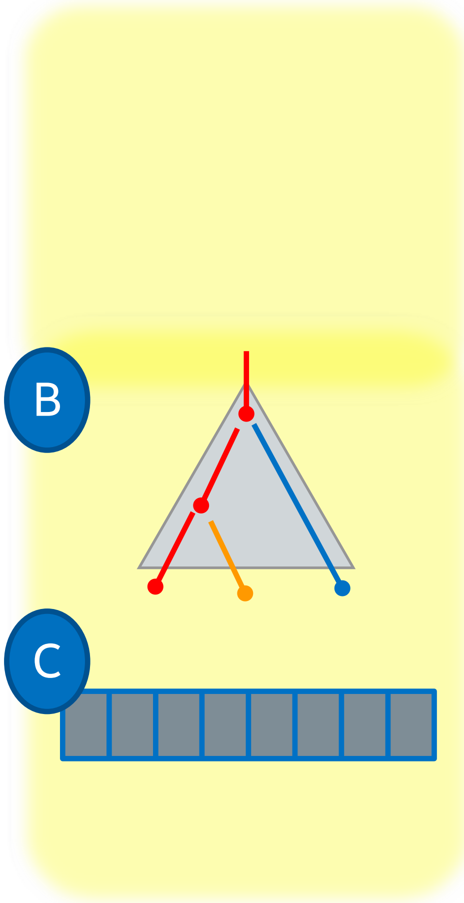
How could we program these parallel machines?



- Different methods exist
- OpenMP4.x:
  - Industry standard
  - C/C++ and Fortran
  - Supported by Intel Compiler (14, 15, 16), GCC 4.9, …
  - Both levels of microprocessor parallelism

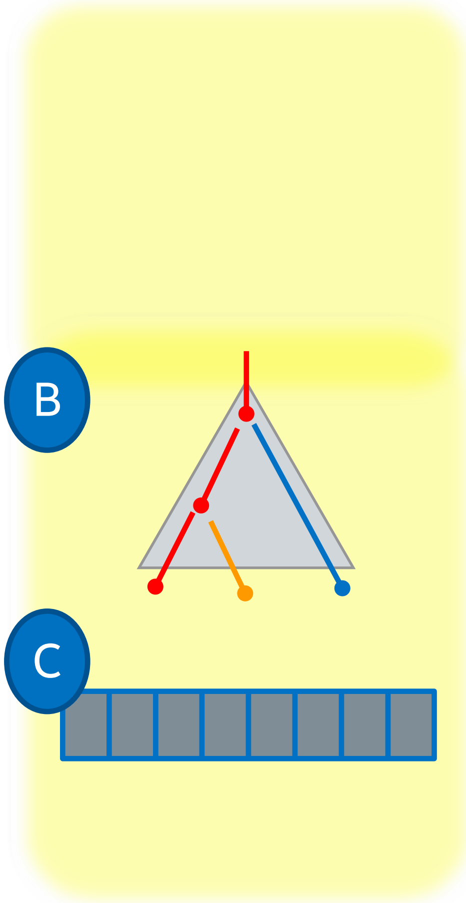
2 level parallelism decomposition with **OpenMP4.x**: image processing example

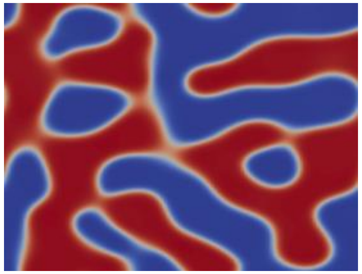




```
#pragma omp parallel for
 for (int y = 0; y < ImageHeight; ++y){
#pragma omp simd
    for (int x = 0; x < ImageWidth; ++x){
       count[y][x] = mandel(in_vals[y][x]);
    }
 }
```

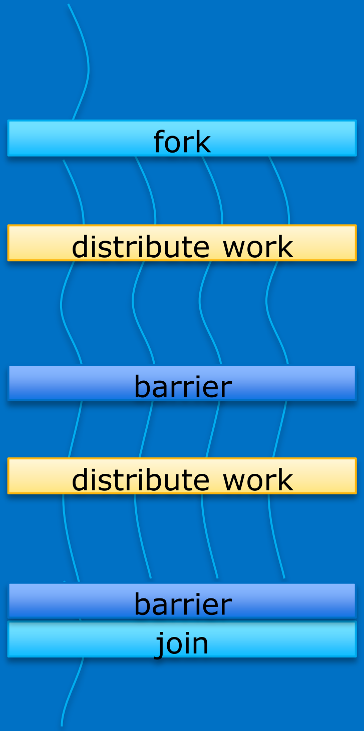
2L parallelism decomposition with **OpenMP4.x**: fluid dynamics example





```
#pragma omp parallel for
for (int i = 0; i < X_Dim; ++i){
#pragma omp simd
    for (int m = 0; x < n_velocities; ++m){
        next_i = f(i, velocities(m));
        X[i] = next_i;
    }
}
```

Programming for threading parallelism



Knights Landing Architectural Diagram

Over 3 TF DP peak

Full Xeon ISA compatibility through AVX-512

~3x single-thread vs. compared to Knights Corner

Up to **16GB high-bandwidth on-package memory** (MCDRAM)

Exposed as **NUMA** node

~500 GB/s sustained BW

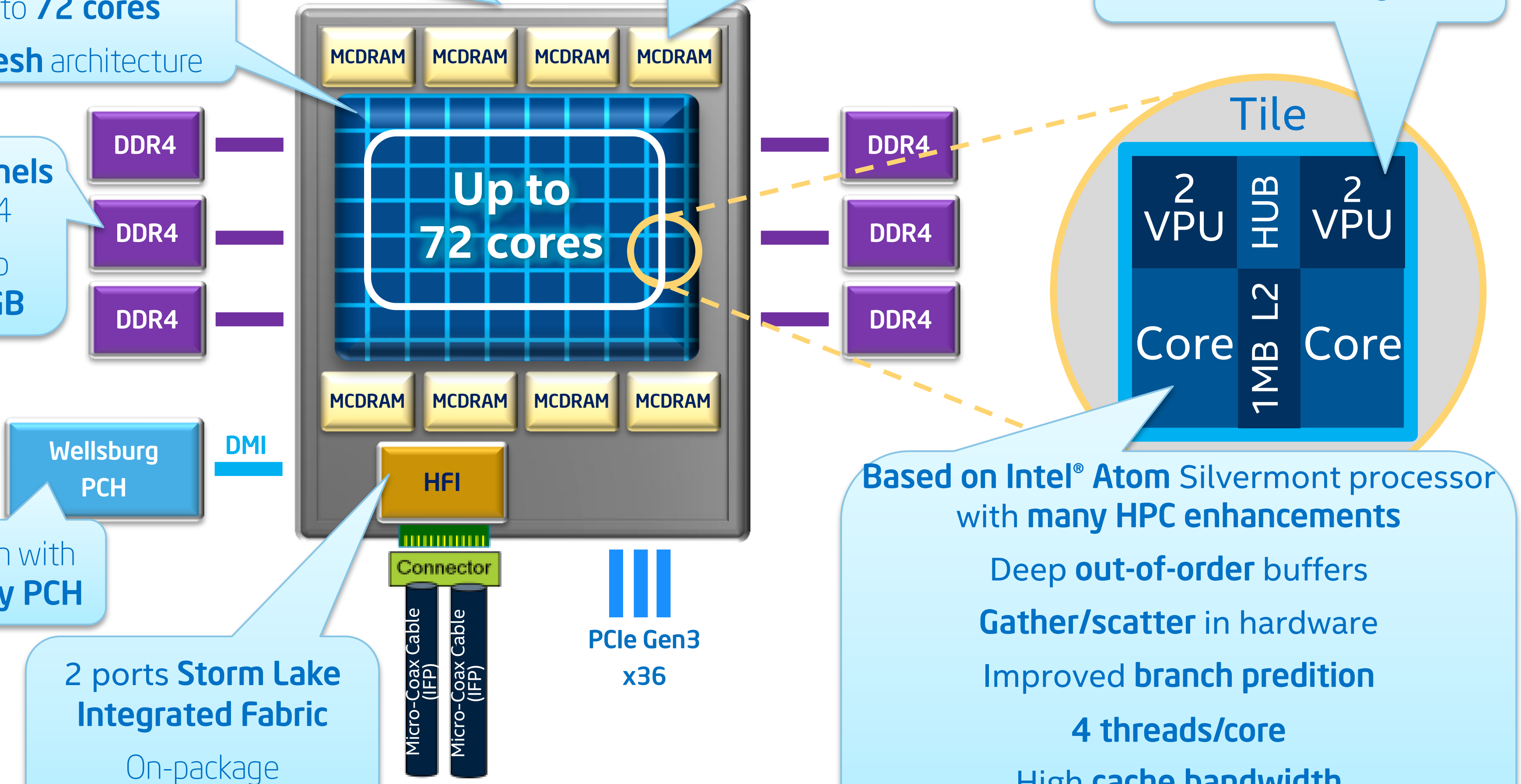
2x 512b VPU per core
(Vector Processing Units)

Up to **72 cores**

2D mesh architecture

6 channels
DDR4

Up to
384GB



Common with
Grantley PCH

2 ports **Storm Lake**
Integrated Fabric

On-package
50 GB/s bi-directional

Based on Intel® Atom Silvermont processor with **many HPC enhancements**

Deep **out-of-order** buffers

Gather/scatter in hardware

Improved **branch predition**

4 threads/core

High **cache bandwidth**

& more

Threading Recommendation **#1**: *Pick a threading model. Don’t use raw threads*

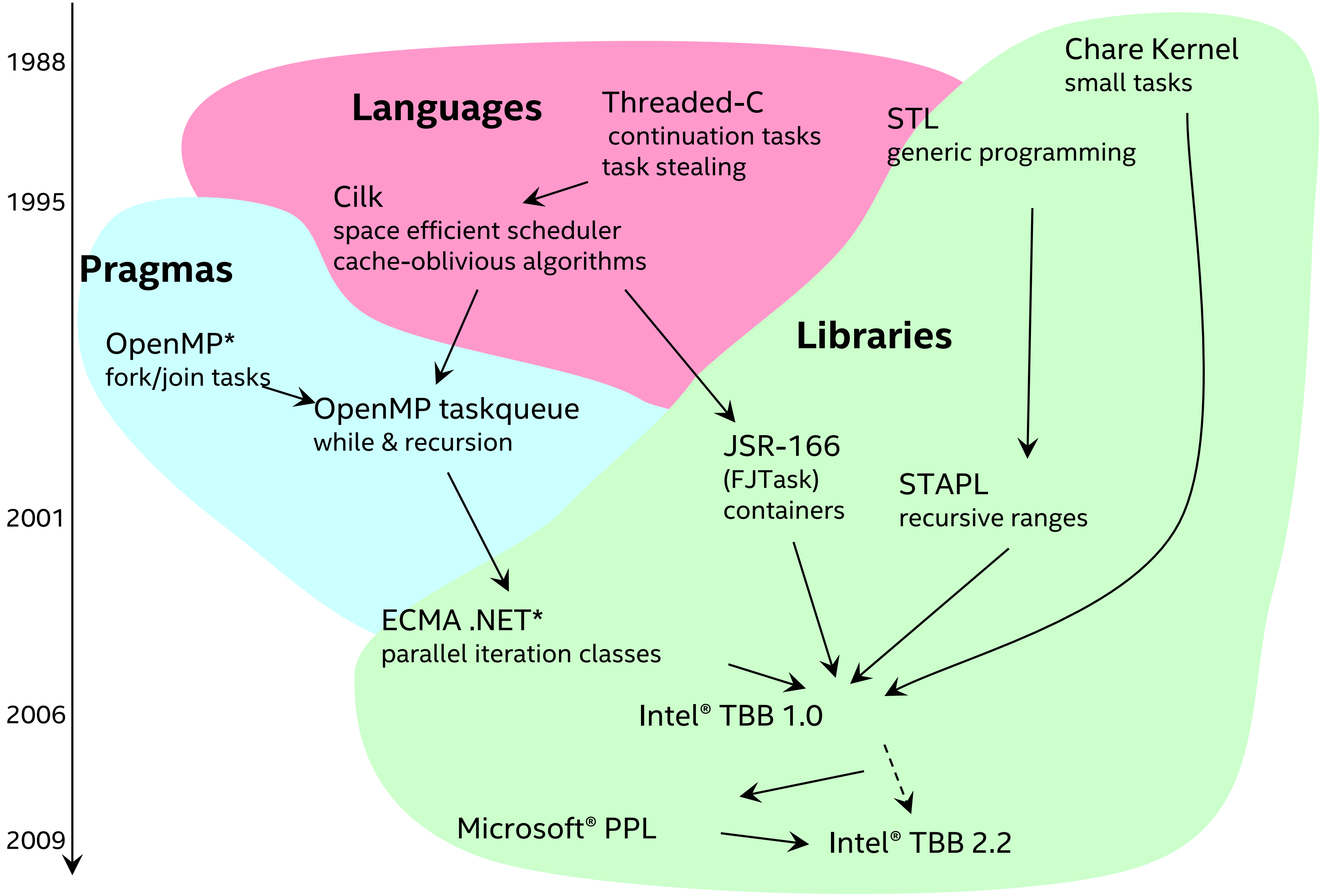
Don’t use raw threads.

- Just trouble on almost all counts: no scalability, no ease of programming, no composability.
- Usually no portability and hardware awareness
- Exception: use raw threads if their purpose is “to wait for things to happen” as opposed to “accelerate a computation”.

Use threading parallel programming models.

- Simpler to use and support
- Future-proof scalability
- Minimize threading overheads
- Portable
  - Threading models implementations are optimized on low-level and could be hardware-aware.

Family tree (not so HPC-centric)



Threading Recommendation **#2**: *Pick a threading model.* ***OpenMP***

If you have *loopy* HPC code, want single standardized model for threads and vector, Fortran and C++, and don't care about threads "composability"

.. **Then Use OpenMP**.

- Industry **Standard**
- Will cover Threading **and** Vector parallelism for you.
- It is widely portable and often "**easy**" to use.
- Has some inherent composability problems for nested parallelism (OpenMP3.x, 4.x is improving)

What is OpenMP?

HPC industry standard:

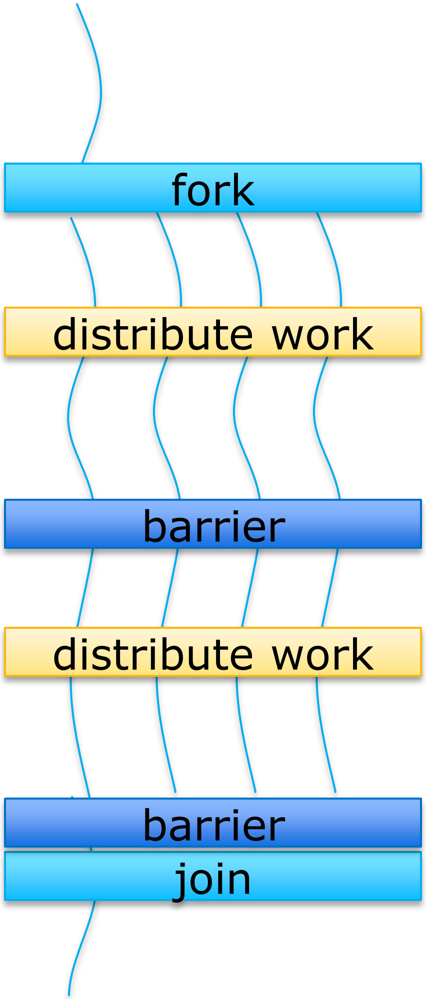
- Portable across systems and vendors
- Maintained by the OpenMP ARB
  - a consortium of industry (Intel, IBM, Cray, ..) and academic institutions (LLNL, ANL, Aachen, BSC, ...)

API for C/C++/Fortran for programming shared-memory systems

- Directive based
- Provides support for
  - (Threading) Data parallelism
  - (SIMD vector) Data parallelism
  - (Threading) Task parallelism
  - Synchronizations

OpenMP in a nutshell: threading data parallelism

```
#pragma omp parallel
{
    #pragma omp for
    for ( i = 0; i <N ; i++)
    {…}

    #pragma omp for
    for ( i = 0; i < N; i++)
    {…}
}
```



Threading Recommendation **#3**: Pick a threading model. *TBB or Cilk™Plus*

… **Else** Use TBB or Cilk™Plus

- From technology standpoint could be seen as similar
  - Both are based on work-stealing.
  - Both do nested threading parallelism well and compose cleanly
  - Both have an exception-handling model

Up close there are some significant differences between TBB and Cilk™Plus

TBB vs. Cilk™Plus:

Cilk

Cilk:

- Will cover Threading **and** Vector parallelism for you
  - but as of right now only for x86 C, C++
- Cilk syntax is easier than TBB, particularly if you are not comfortable with C++ lambda functions.
- Cilk semantics are cleaner than TBB. E.g. serial elision properties and hyperobjects.
- Cilk can be used directly in C code.
- Cilk requires compiling with Intel compiler or experimental gcc branch

TBB vs. Cilk™Plus:
TBB

TBB:

- TBB is more portable – e.g. you can use Microsoft's compiler or any version of g++ if you insist.
- TBB supports more flexible forms of parallelism such as pipelines and flowgraphs.
  - This is either a feature or hanging rope depending on the programmer.
  - TBB exposes lots of low-level hooks for writing your own forms of parallelism
- TBB is directly callable only from C++.
- For Vector Parallelism you will have to use something else (because TBB is NOT a compiler technology)

Threading Recommendation #4: *Pick a Concurrent container*

If you need concurrent containers, scalable memory allocator, or atomic operations:

- Use the corresponding TBB components.
- You can mix these with any of the threading models.

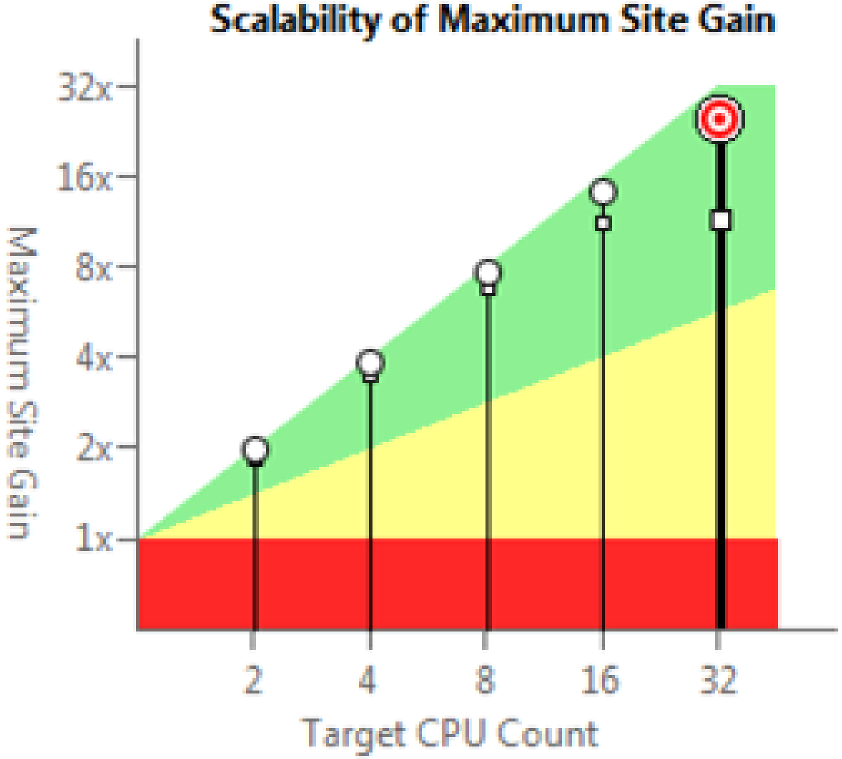
Predict parallel behavior:

Scalability graph

Ideal scalability : linear

- The speedup increases linear

  to the number of cores



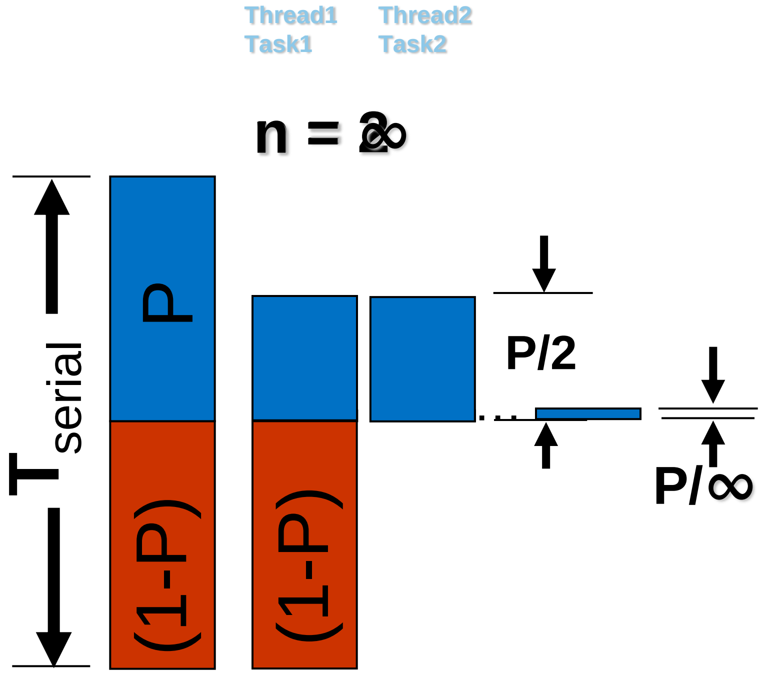
Scalability can be limited by:

- Serial execution (Amdahl's law)
- Load balancing
- Dataset size (Gustafson's law)
- Task granularity

  vs. runtime scheduler overheadeads
- Lock contention
- Other hardware limits (memory-bound, uarch,...)

Predict parallel behavior:

Amdahl's law



0.5 + 0.05

$$T_{parallel} = \{\underline{(1-P)} + \underline{P/n\}\, T_{serial}} + O$$

n = number of processors

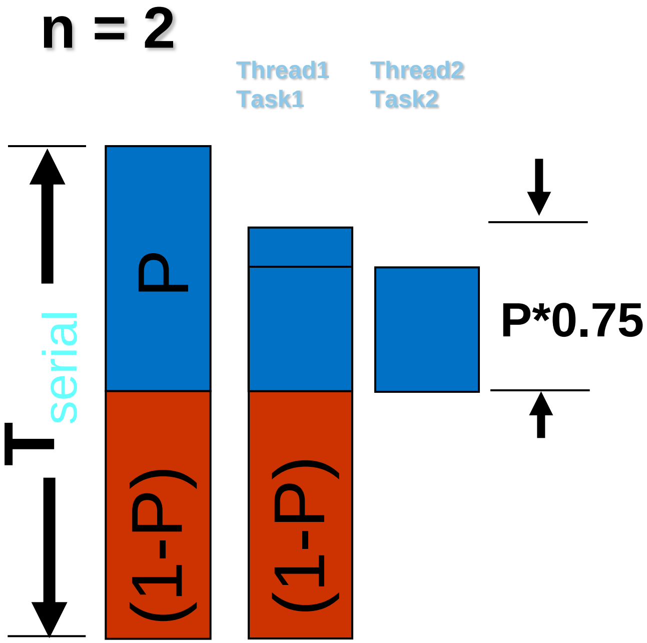
Scaling = T_{serial} / $T_{parallel}$ 1.0/0.55==21033

Serial code limits scaling

Describes the upper bound of parallel speedup

Predict parallel behavior:
Load balancing

Loop iterations (tasks or “task chunks” in general) may **not be distributed evenly**

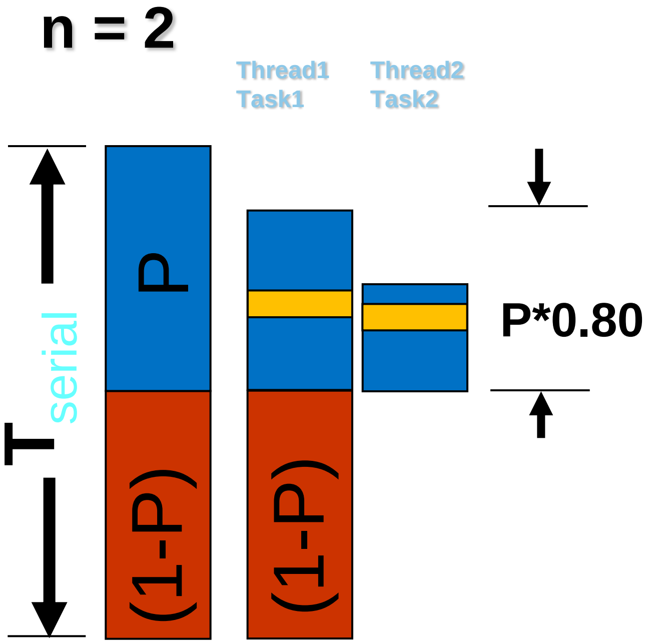


1.0/0.75 = **1.33**
1.0/0.86 = **1.16**

Load balancing limits scaling

Predict parallel behavior: Lock contention

Each task alternates between unlocked execution and locked execution



1.0/0.90 = 1.11x

Lock contention limits scaling

Parallel behavior:

IS LIMITED / DEFINED BY:

CPU bound work, parallelizable (serial code impact and Amdahl's law)

Dataset size (Gustafson' law)

Task Granularity (& chunking)

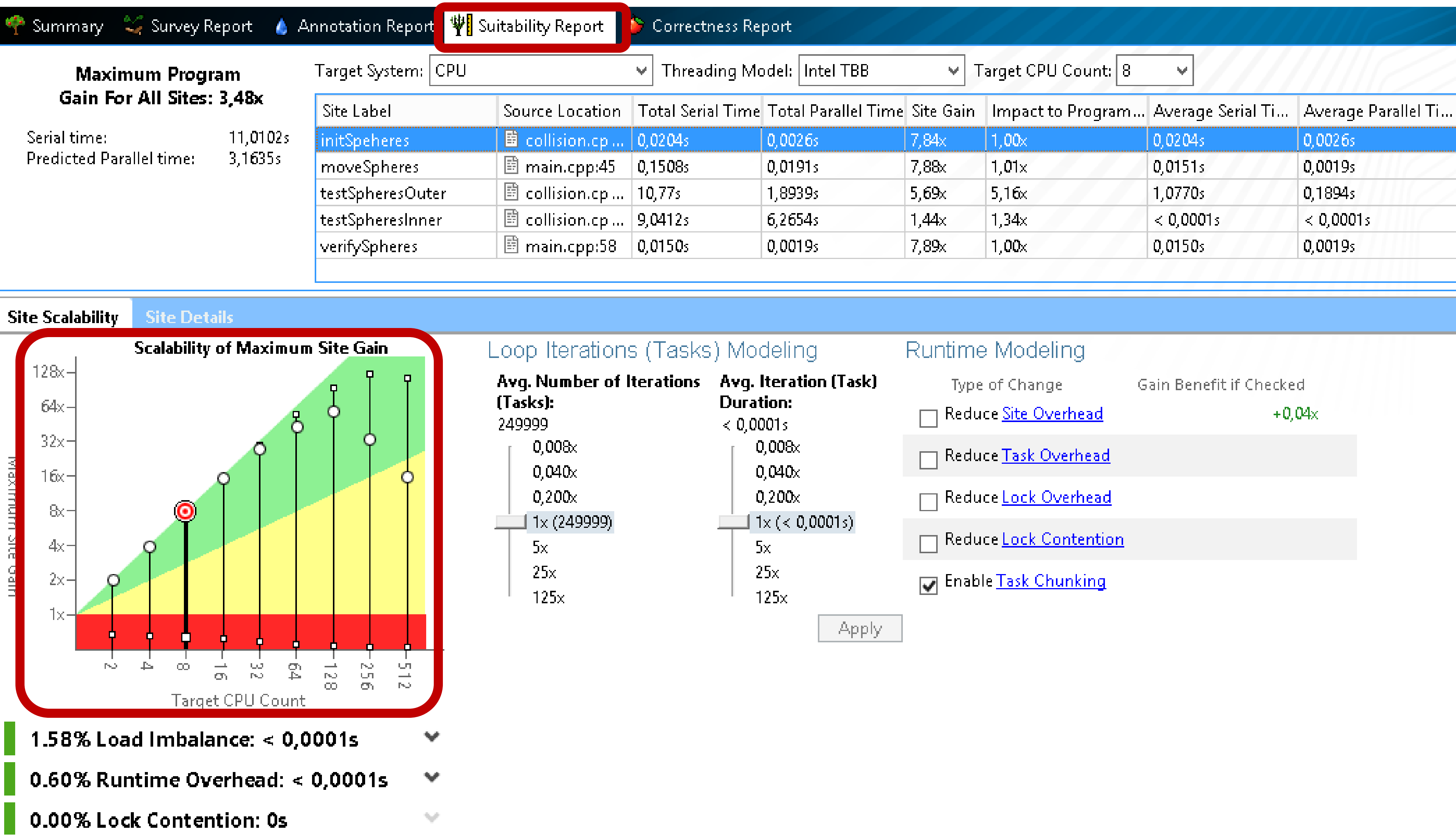
Load balancing

Lock contention

Parallel Runtime Overheads

Other hardware limits (memory-bound, uarch,...)

Intel® Advisor Suitability



Maximum Program Gain For All Sites: 3,48x

Serial time:	11,0102s
Predicted Parallel time:	3,1635s

Target System: CPU | Threading Model: Intel TBB | Target CPU Count: 8

Site Label	Source Location	Total Serial Time	Total Parallel Time	Site Gain	Impact to Program...	Average Serial Ti...	Average Parallel Ti...
initSpeheres	collision.cp ...	0,0204s	0,0026s	7,84x	1,00x	0,0204s	0,0026s
moveSpheres	main.cpp:45	0,1508s	0,0191s	7,88x	1,01x	0,0151s	0,0019s
testSpheresOuter	collision.cp ...	10,77s	1,8939s	5,69x	5,16x	1,0770s	0,1894s
testSpheresInner	collision.cp ...	9,0412s	6,2654s	1,44x	1,34x	< 0,0001s	< 0,0001s
verifySpheres	main.cpp:58	0,0150s	0,0019s	7,89x	1,00x	0,0150s	0,0019s

1.58% Load Imbalance: < 0,0001s

0.60% Runtime Overhead: < 0,0001s

0.00% Lock Contention: 0s

Loop Iterations (Tasks) Modeling

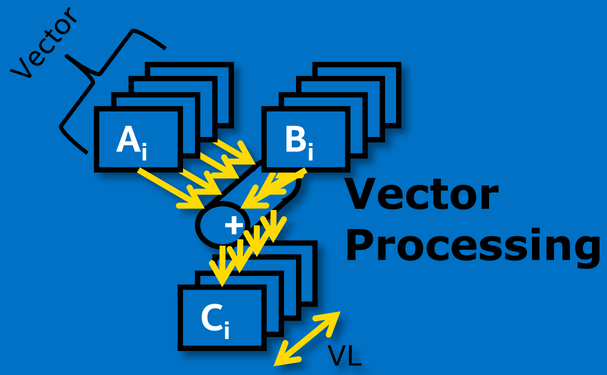
Avg. Number of Iterations (Tasks):	Avg. Iteration (Task) Duration:
249999	< 0,0001s
0,008x	0,008x
0,040x	0,040x
0,200x	0,200x
1x (249999)	1x (< 0,0001s)
5x	5x
25x	25x
125x	125x

Runtime Modeling

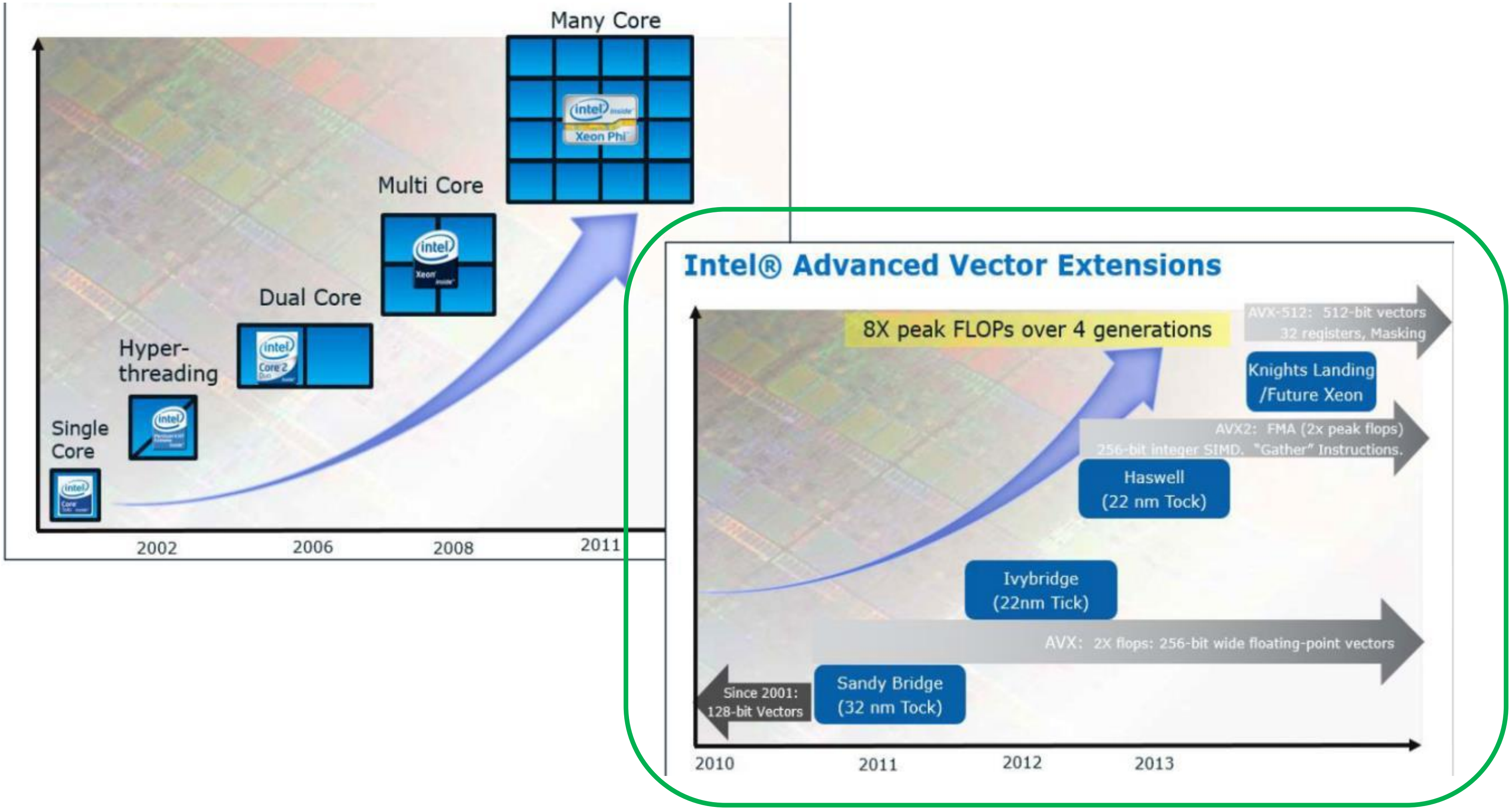
Type of Change	Gain Benefit if Checked
□ Reduce Site Overhead	+0,04x
□ Reduce Task Overhead	
□ Reduce Lock Overhead	
□ Reduce Lock Contention	
☑ Enable Task Chunking	

Analyze the potential benefit of your proposal

Programming
for vector SIMD parallelism

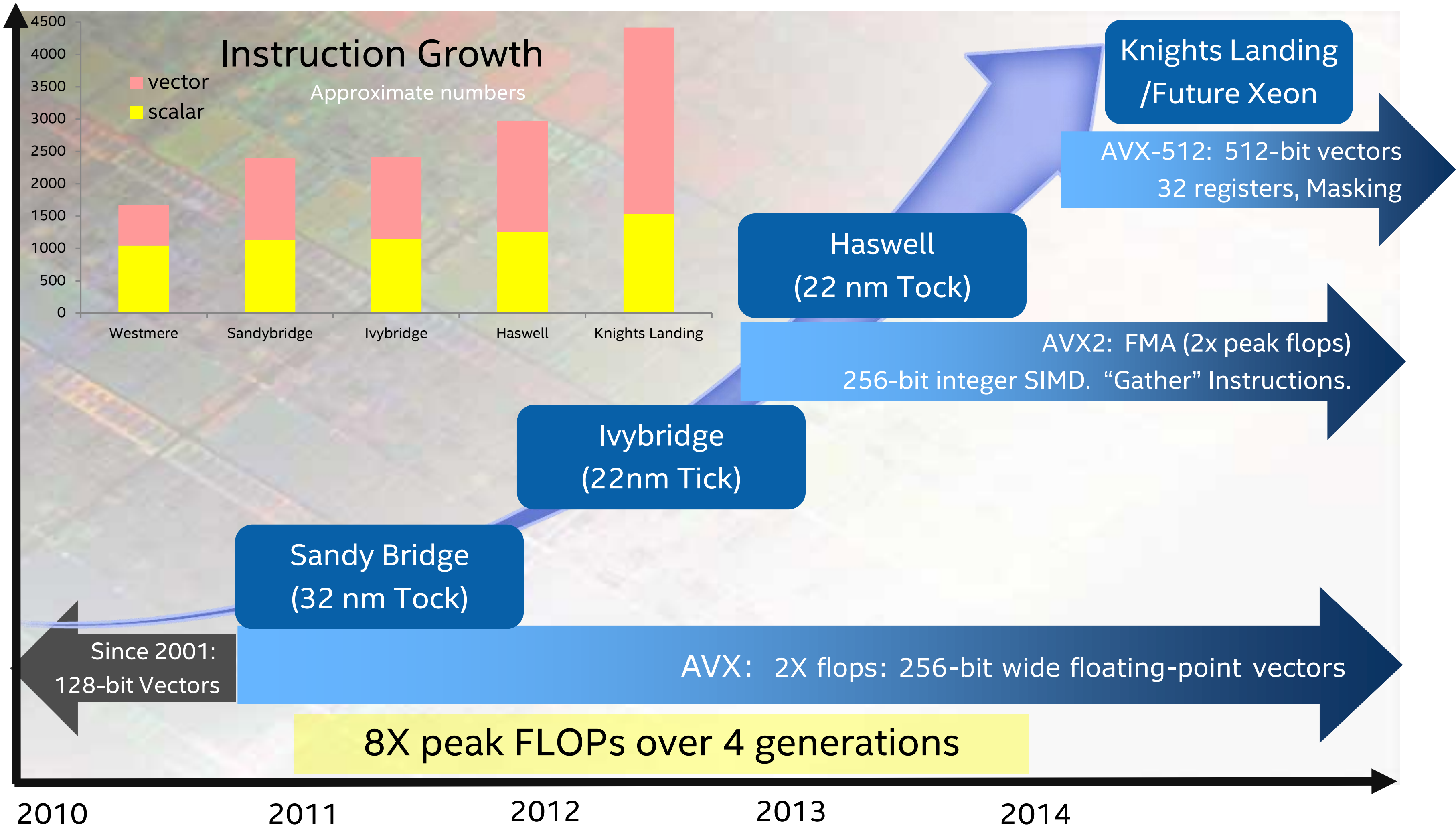


Why should we care about Vector SIMD parallelism at all?



4C	4C	6C	8C	12C	14C	60C+
4 * SP	4*SP	4*SP	8*SP	8*SP	16*SP	32*SP

Intel® Advanced Vector Extensions



Since 2001:
128-bit Vectors

Sandy Bridge
(32 nm Tock)

AVX: 2X flops: 256-bit wide floating-point vectors

Ivybridge
(22nm Tick)

Haswell
(22 nm Tock)

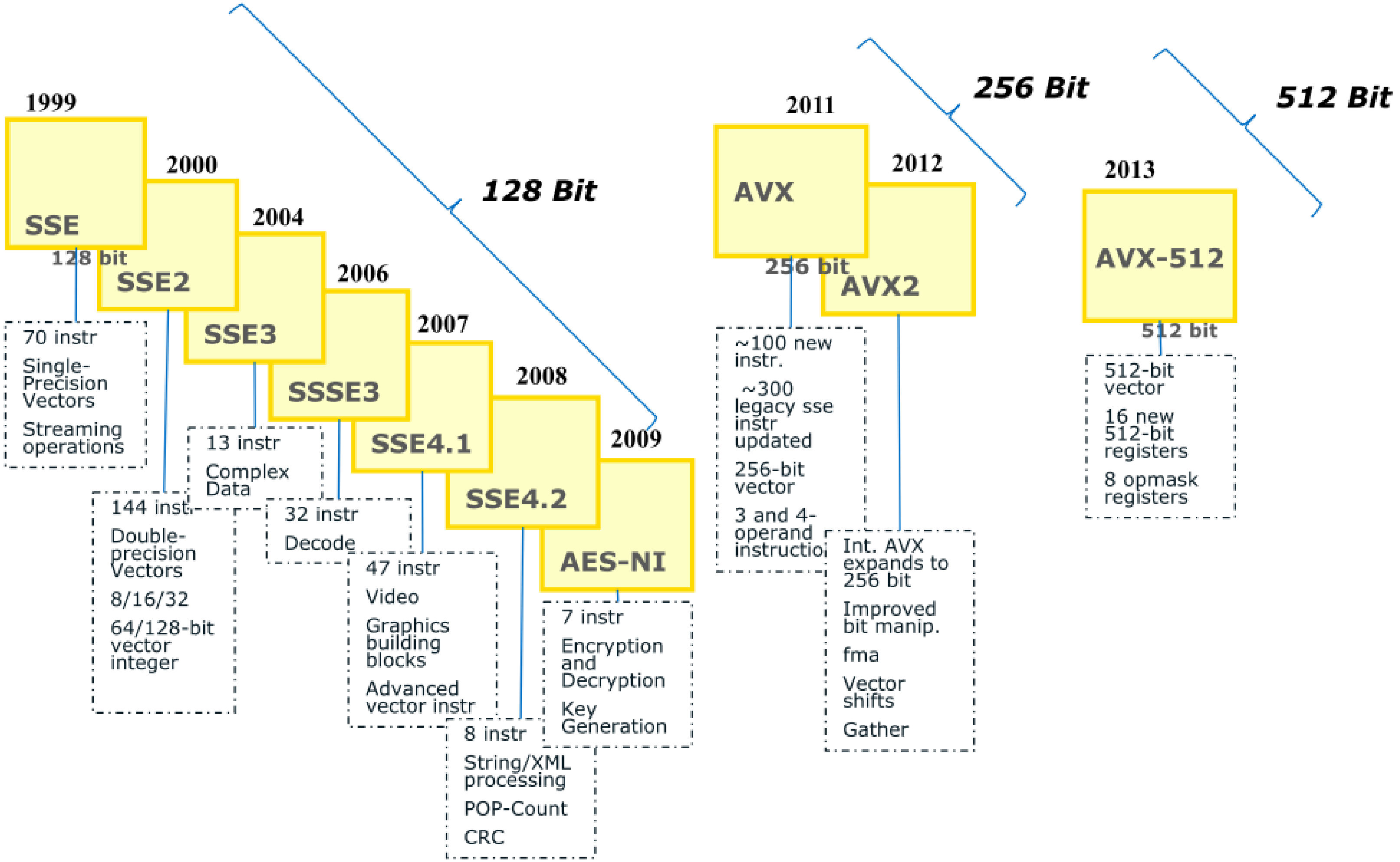
AVX2: FMA (2x peak flops)
256-bit integer SIMD. "Gather" Instructions.

Knights Landing
/Future Xeon

AVX-512: 512-bit vectors
32 registers, Masking

8X peak FLOPs over 4 generations

This is old story. Even for x86.



128 Bit

256 Bit

512 Bit

1999

SSE

128 bit

70 instr

Single-Precision Vectors

Streaming operations

2000

SSE2

144 instr

Double-precision Vectors

8/16/32

64/128-bit vector integer

2004

SSE3

13 instr

Complex Data

2006

SSSE3

32 instr

Decode

2007

SSE4.1

47 instr

Video

Graphics building blocks

Advanced vector instr

2008

SSE4.2

8 instr

String/XML processing

POP-Count

CRC

2009

AES-NI

7 instr

Encryption and Decryption

Key Generation

2011

AVX

256 bit

~100 new instr.

~300 legacy sse instr updated

256-bit vector

3 and 4-operand instructio

2012

AVX2

Int. AVX expands to 256 bit

Improved bit manip.

fma

Vector shifts

Gather

2013

AVX-512

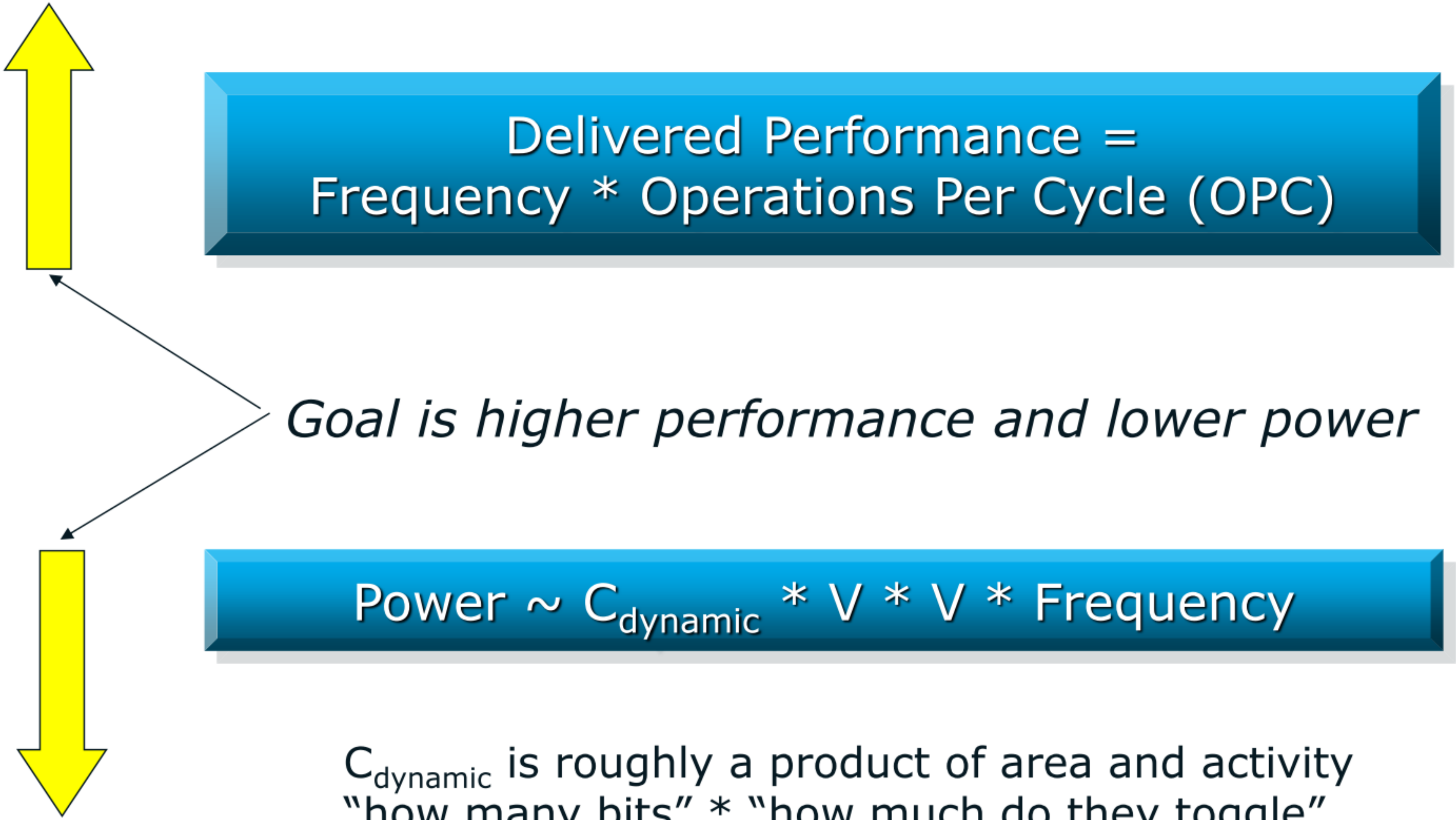
512 bit

512-bit vector

16 new 512-bit registers

8 opmask registers

Why SIMD vector parallelism?



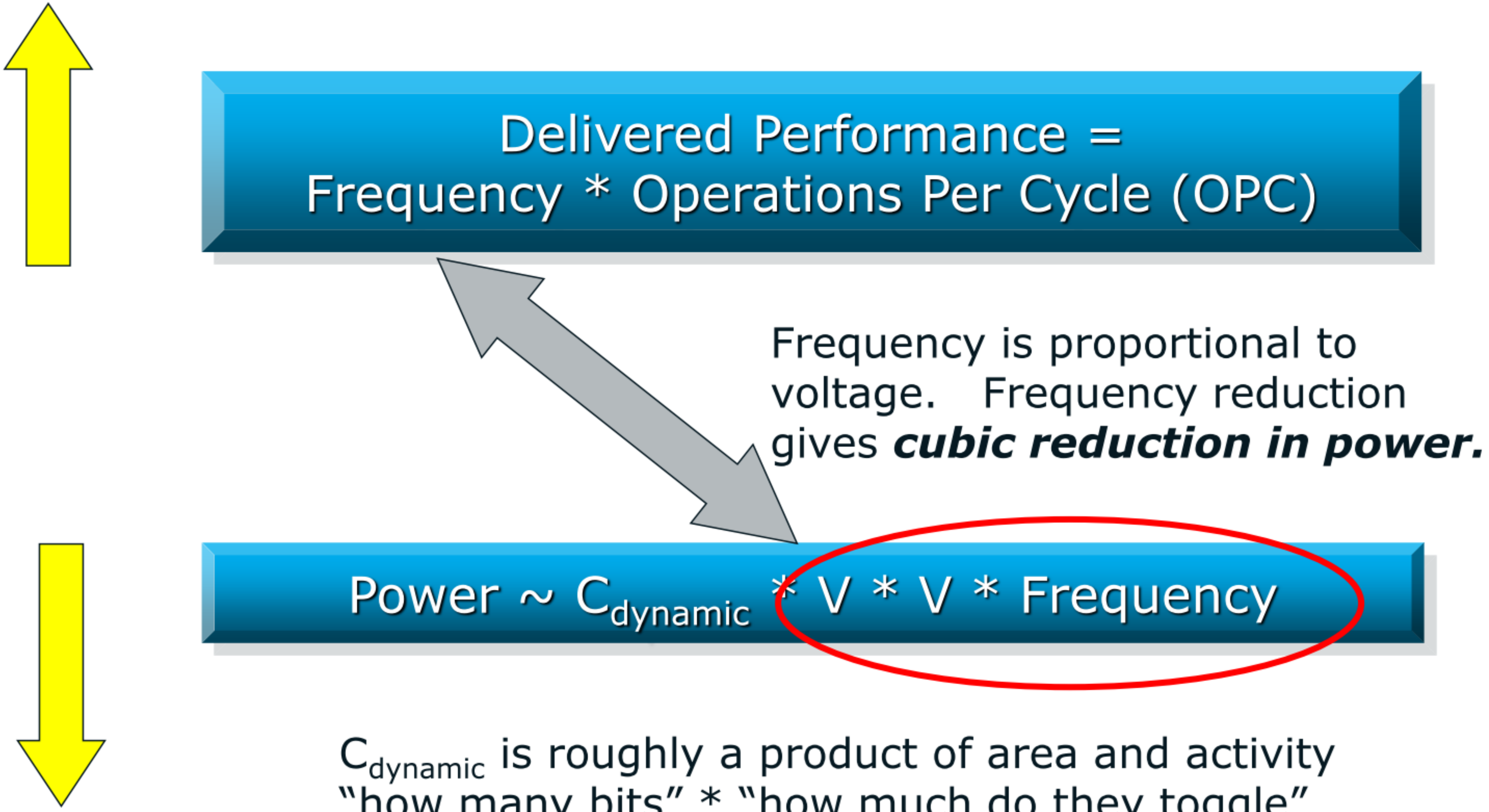
Delivered Performance = Frequency * Operations Per Cycle (OPC)

Goal is higher performance and lower power

Power ~ $C_{dynamic}$ * V * V * Frequency

$C_{dynamic}$ is roughly a product of area and activity
“how many bits” * “how much do they toggle”

Why SIMD vector parallelism?



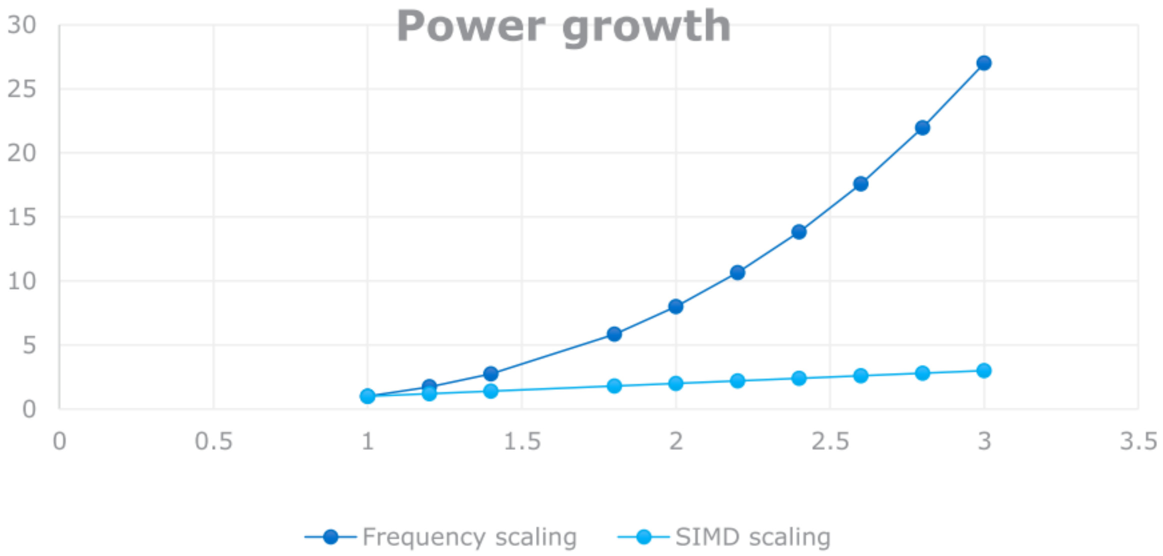
Delivered Performance = Frequency * Operations Per Cycle (OPC)

Frequency is proportional to voltage. Frequency reduction gives ***cubic reduction in power.***

Power ~ $C_{dynamic}$ * V * V * Frequency

$C_{dynamic}$ is roughly a product of area and activity
“how many bits” * “how much do they toggle”

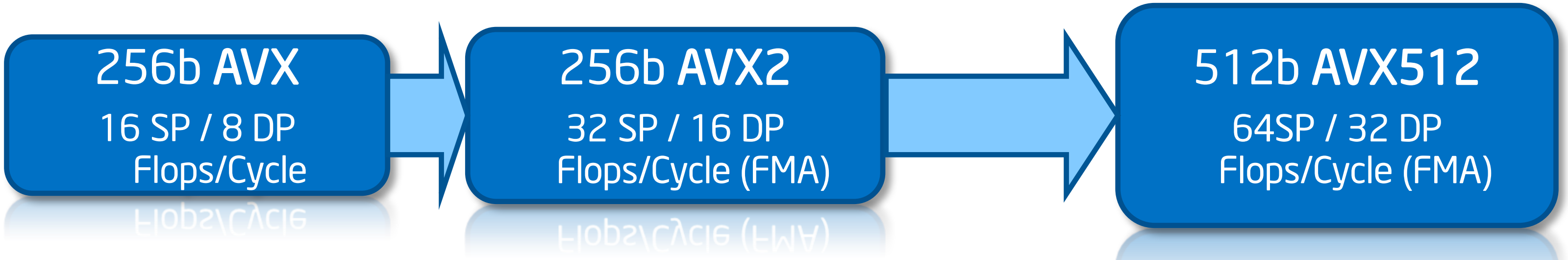
Why SIMD vector parallelism?



Wider SIMD -- Linear increase in area and power
Wider superscalar – Quadratic increase in area and power
Higher frequency – Cubic increase in power

With SIMD we can go faster with less power

Intel® AVX Technology



256b AVX
16 SP / 8 DP Flops/Cycle

256b AVX2
32 SP / 16 DP Flops/Cycle (FMA)

512b AVX512
64SP / 32 DP Flops/Cycle (FMA)

AVX	AVX2
256-bit basic FP	Float16 (IVB 2012)
16 registers	256-bit FP FMA
NDS (and AVX128)	256-bit integer
Improved blend	PERMD
MASKMOV	Gather
Implicit unaligned	

AVX512

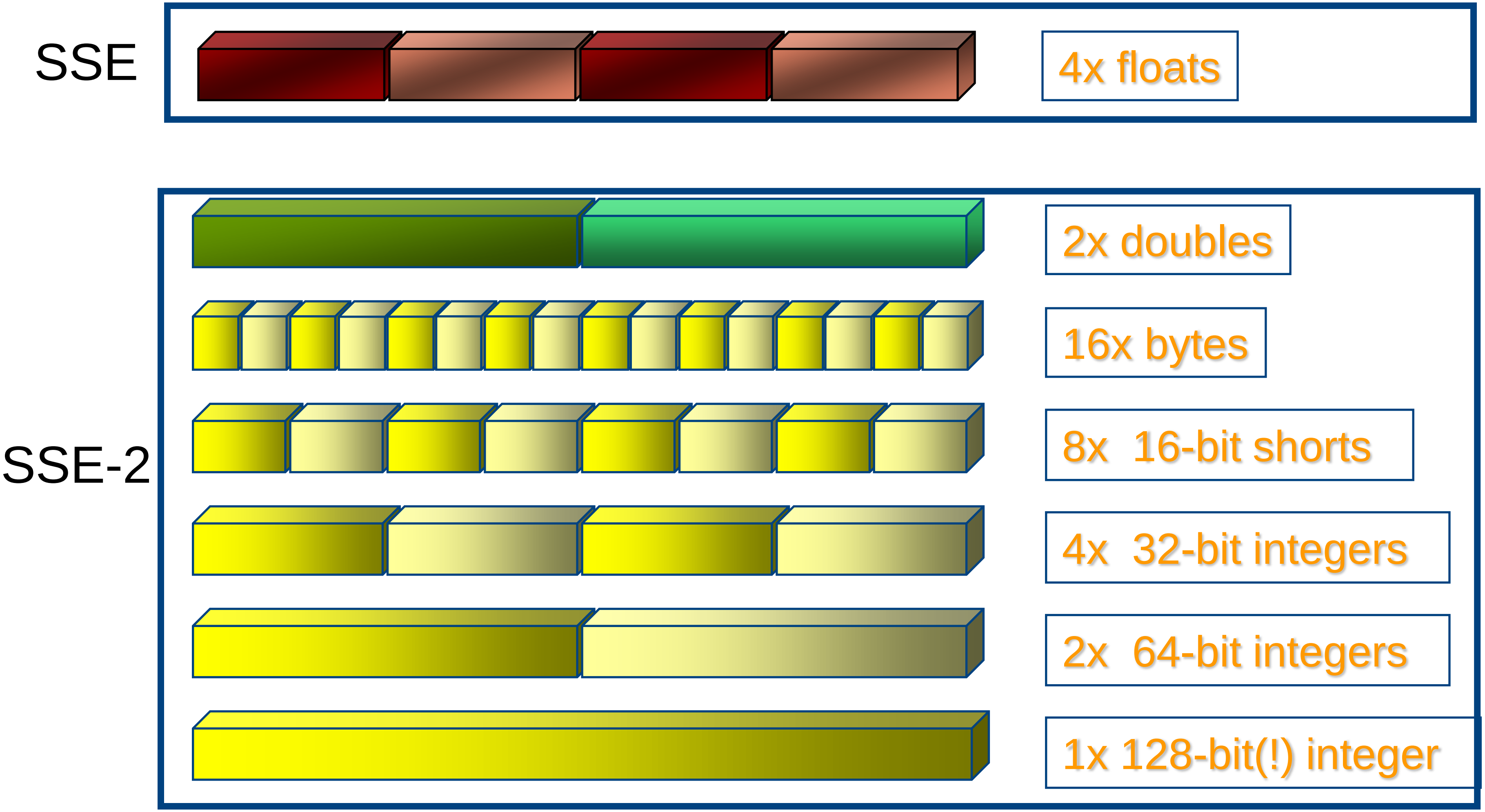
512-bit FP/Integer
32 registers
8 mask registers
Embedded rounding
Embedded broadcast
Scalar/SSE/AVX "promotions"
HPC additions
Transcendental support
Gather/Scatter

SNB
2011

HSW
2013

Future Processors (KNL & future Xeon)

Intel® SSE and AVX-128 Data Types



SSE

4x floats

SSE-2

2x doubles

16x bytes

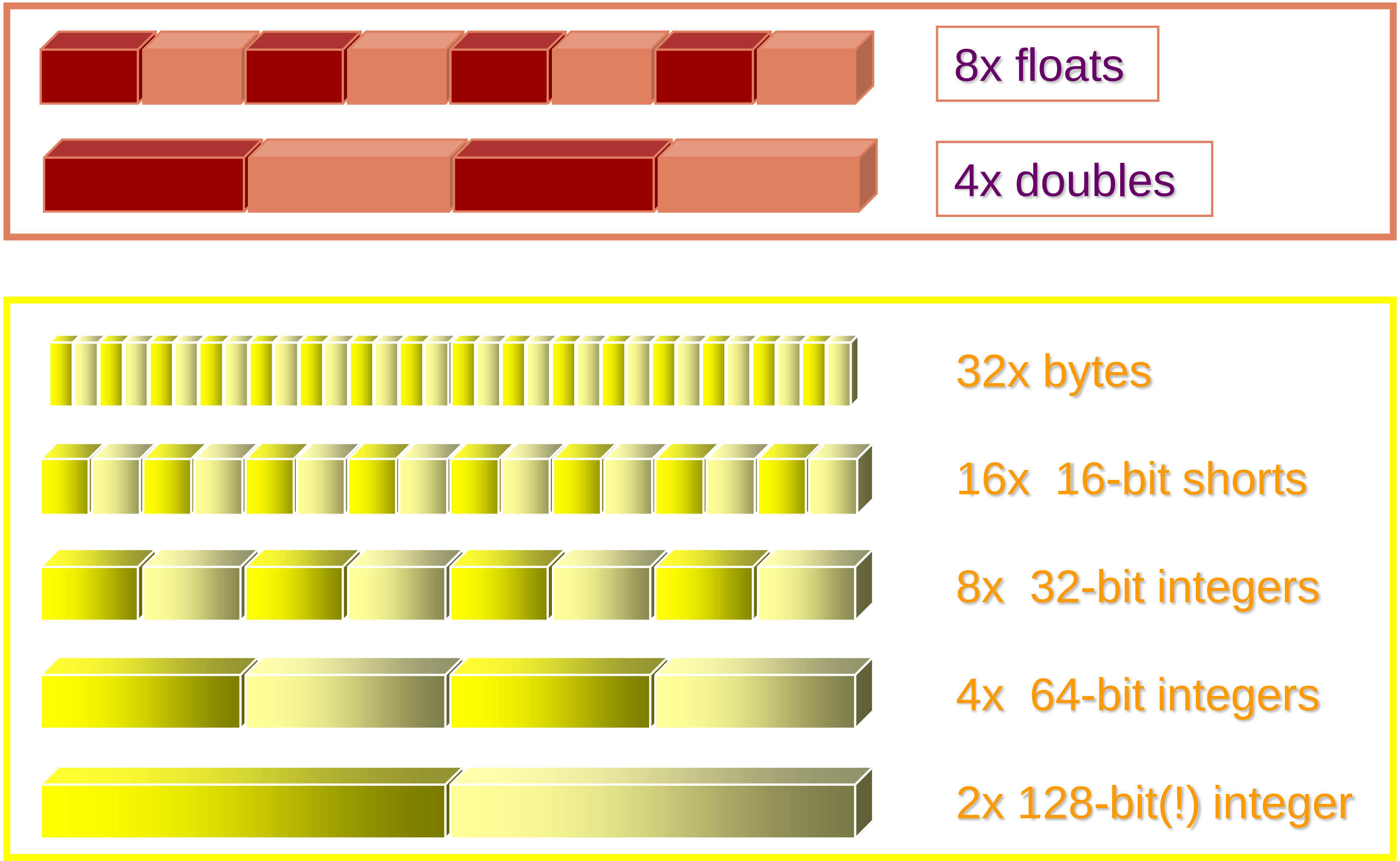
8x 16-bit shorts

4x 32-bit integers

2x 64-bit integers

1x 128-bit(!) integer

AVX-256 Data Types



Intel® AVX

8x floats

4x doubles

Intel® AVX2

32x bytes

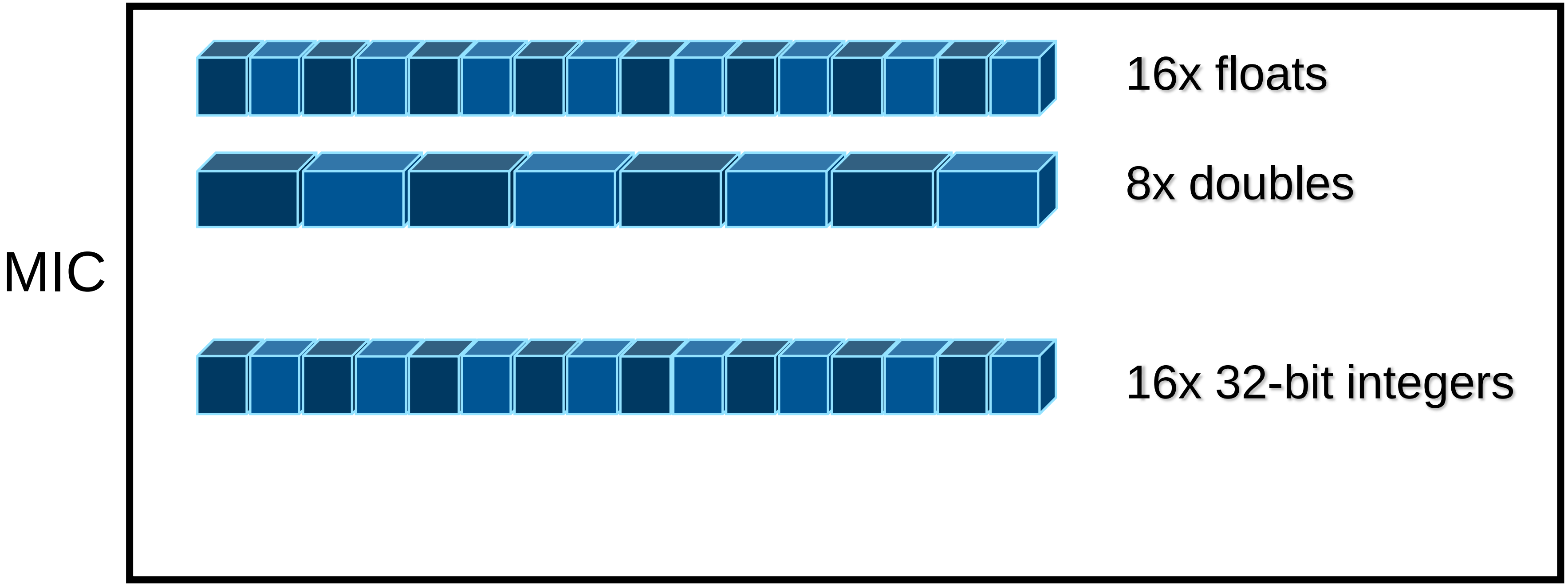
16x 16-bit shorts

8x 32-bit integers

4x 64-bit integers

2x 128-bit(!) integer

Data Types for Intel® MIC Architecture



MIC

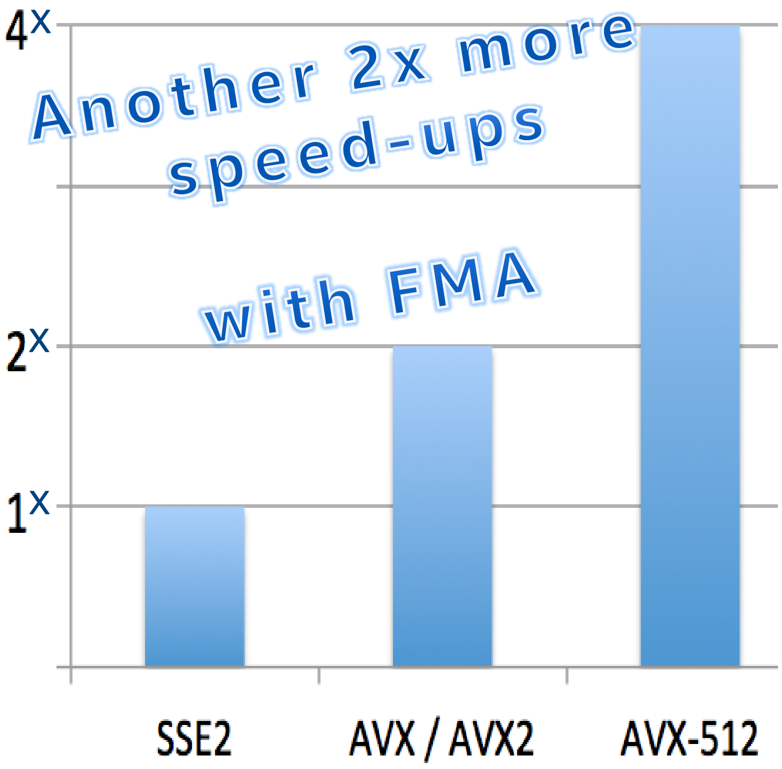
16x floats

8x doubles

16x 32-bit integers

8x Double-Precision speed-up over SSE

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

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

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

We call this “vectorization”

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

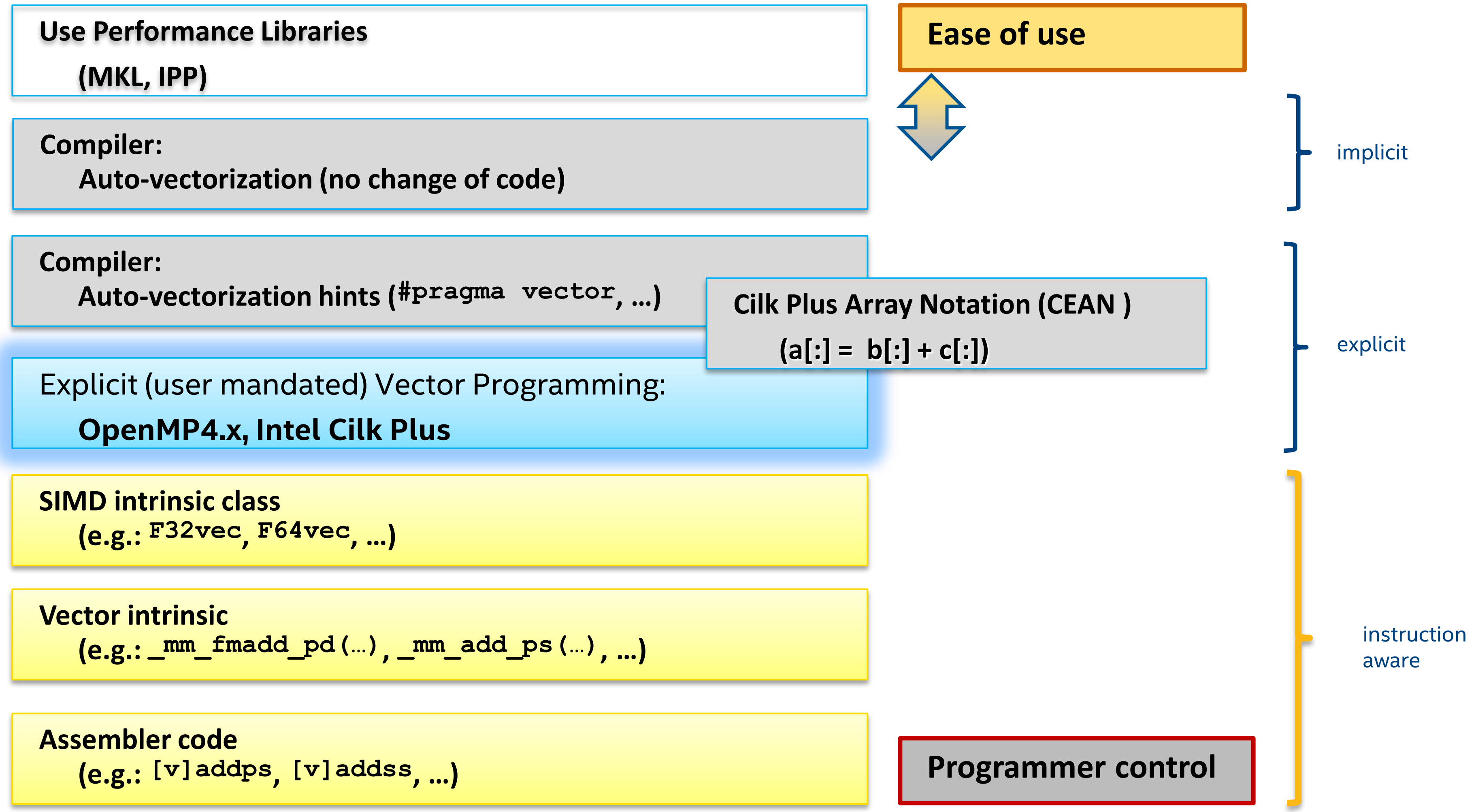
Loop:

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

Many Ways to Vectorize



Use Performance Libraries
(MKL, IPP)

Compiler:
Auto-vectorization (no change of code)

Compiler:
Auto-vectorization hints (`#pragma vector`, …)

Cilk Plus Array Notation (CEAN)
(a[:] = b[:] + c[:])

Explicit (user mandated) Vector Programming:
OpenMP4.x, Intel Cilk Plus

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

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

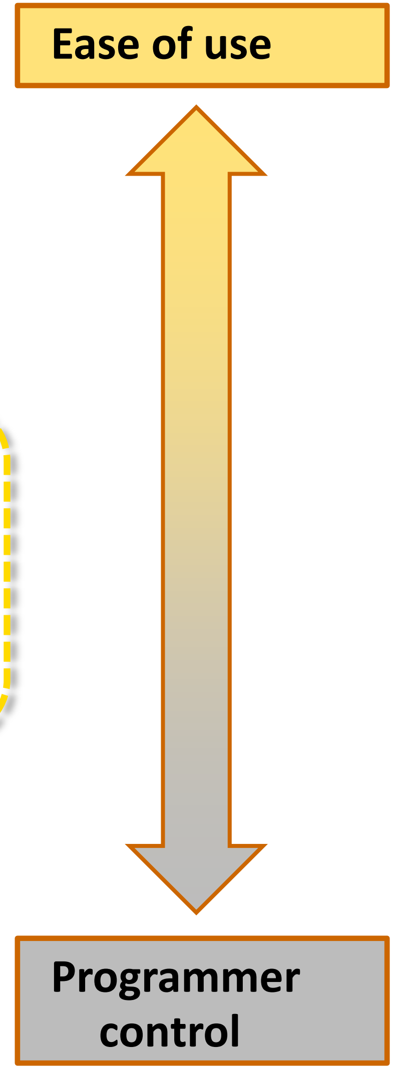
Assembler code
(e.g.: `[v]addps`, `[v]addss`, …)

Ease of use

Programmer control

OpenMP4.x: threading and vectors

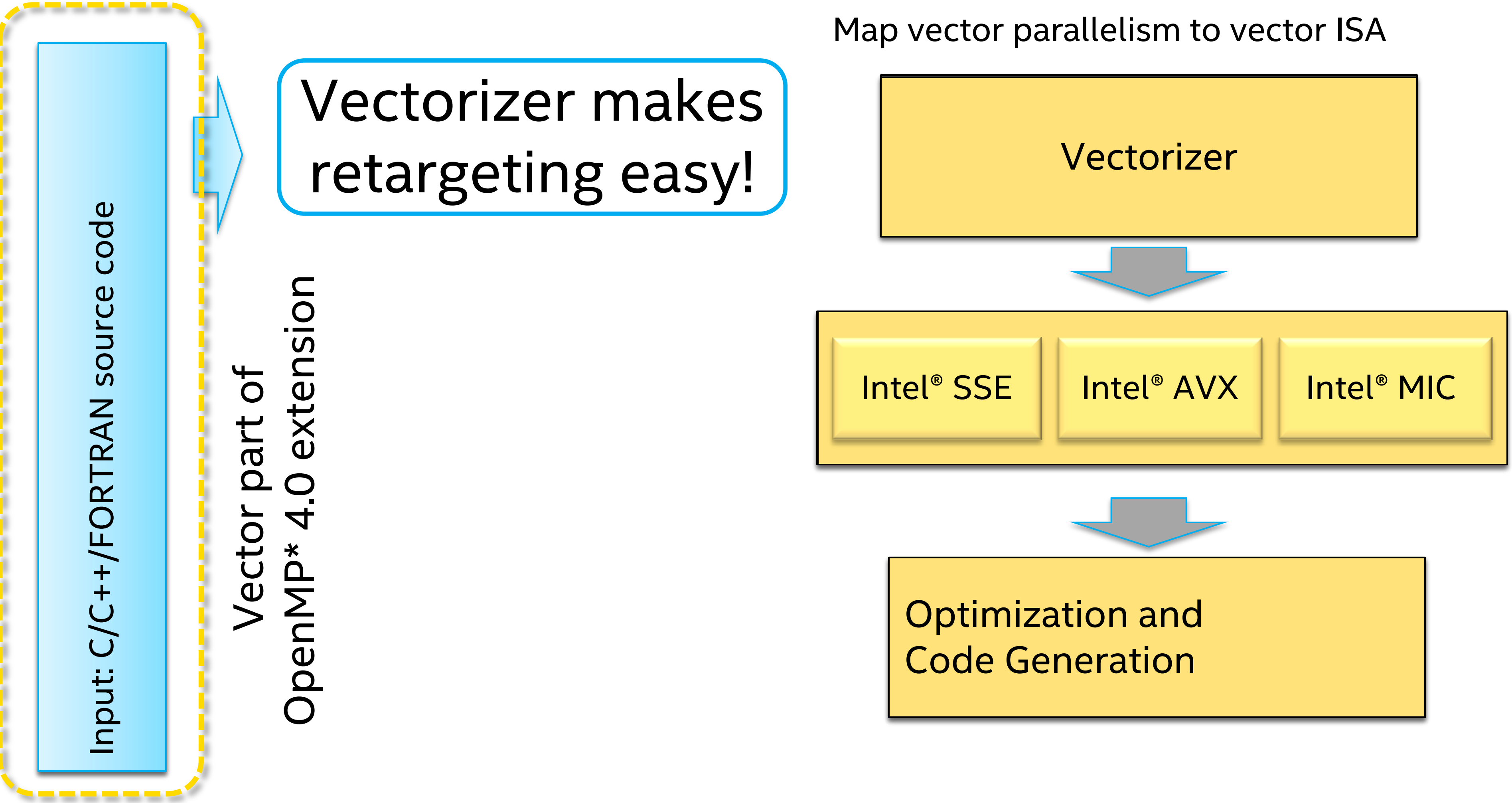
Thread Level Parallelism	SIMD Parallelism
Auto-Parallel invoked by compiler switch, some loops parallelized automatically by compiler `	**Auto-Vectorization** invoked at O2, some loops vectorized automatically by compiler, developer can provide a few hints to the compiler
Parallelization using OpenMP* threading Developer guides parallelization via statements and lexicon of clauses	**Vectorization using OpenMP* 4.0 simd** Developer guides vectorization via statements and lexicon of clauses
Parallelization using Posix* or Windows* Threads	**Vectorization using Intrinsics**



Ease of use

Programmer control

Explicit Vector Programming with OpenMP 4.0



Input: C/C++/FORTRAN source code

Vector part of OpenMP* 4.0 extension

Vectorizer makes retargeting easy!

Map vector parallelism to vector ISA

Vectorizer

Intel® SSE

Intel® AVX

Intel® MIC

Optimization and Code Generation

Compiling for Intel® AVX(2)

Compile with –xavx (Intel® AVX; Sandy Bridge etc)

Compile with –xcore-avx2 (Intel® AVX2; Haswell)

- Intel processors only (Use -mavx, -march=core-avx2 for non-Intel)
- Vectorization works just as for SSE
  - Best if 32 byte aligned
- More loops can be vectorized than with SSE
  - Individually masked data elements
  - More powerful data rearrangement instructions

-axavx (-axcore-avx2) gives both SSE2 and newer ISA code paths

- (!) but use –x or –m switches to modify the default SSE2 code path
  - Eg –axcore-avx2 –xavx to target both Haswell and Sandy Bridge
    (/Qaxcore-avx2 /Qxavx on Windows*)

Math libraries may target AVX and/or AVX2 automatically at runtime

SIMD Pragma Notation

OpenMP 4.0: #pragma omp simd [clause [,clause] …]

- **Targets loops**
  - Can target inner or outer canonical loops

- **Developer asserts loop is suitable for SIMD**
  - The Intel Compiler will vectorize if possible (will ignore dependency or efficiency concerns)
  - Use when you **KNOW** that a given loop is safe to vectorize
  - Can choose from lexicon of clauses to modify behavior of SIMD directive
- **Developer should validate results (correctness)**
  - Just like for race conditions in OpenMP* threading loops

- **Minimizes source code changes needed to enforce vectorization**

OMP SIMD Pragma Clauses

`reduction(operator:v1, v2, …)`

- v1 etc are reduction variables for operation “operator”
- Examples include computing averages or sums of arrays into a single scalar value : *reduction (+:sum)*

`linear(v1:step1, v2:step2, …)`

- declares one or more list items to be private to a SIMD lane and to have a linear relationship with respect to the iteration space of a loop : *linear (i:2)*

`safelen (`*`length`*`)`

- no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than this value
- *Typical values are* 2, 4, 8, 16

Refer to OpenMP 4.0 Specification.
http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

OMP SIMD Pragma Clauses cont…

`aligned(v1:alignment, v2:alignment)`

- declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the aligned clause.

`collapse(number of loops)`

- Nested loop iterations are collapsed into one loop with a larger iteration space.

`private(v1, v2, …), lastprivate (v1, v2, …)`

- declares one or more list items to be private to an implicit task or to a SIMD lane, lastprivate causes the corresponding original list item to be updated after the end of the region..

SIMD-enabled functions

Write a function for one element and add **pragma** as follows

```
#pragma omp declare simd
float foo(float a, float b, float c, float d)
{
  return a * b + c * d;
}
```

Call the scalar version:

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

Call vector version via SIMD loop:

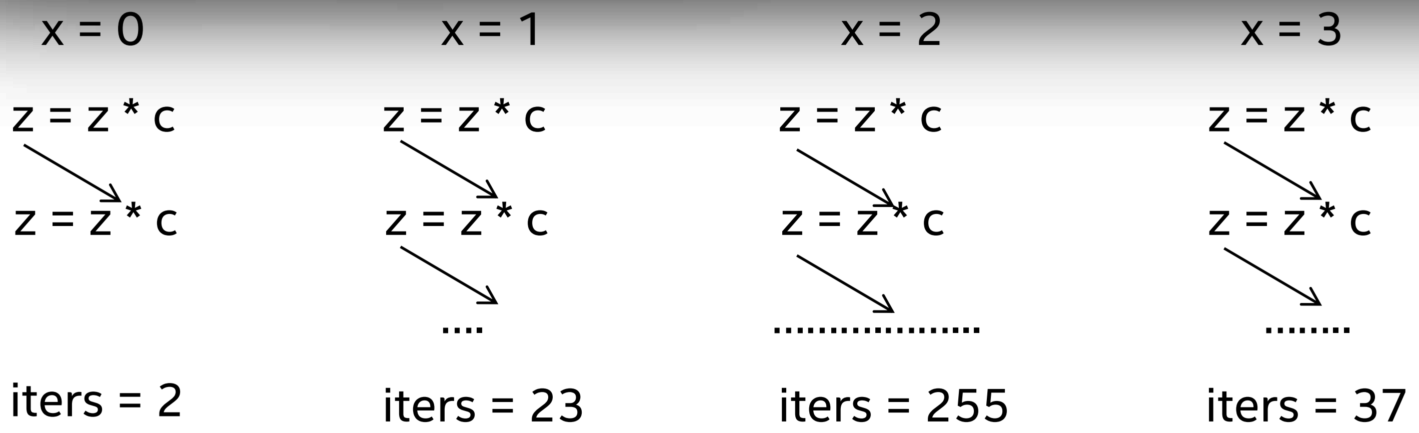
```
#pragma omp simd
for(i = 0; i < n; i++) {
  A[i] = foo(B[i], C[i], D[i], E[i]);
}
```

```
A[:] = foo(B[:], C[:], D[:], E[:]);
```

Example of Outer Loop Vectorization

```
#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;
}
```

```
float in_vals[];
#pragma omp simd
for(int x = 0; x < Width; ++x) {
    count[x] = lednam(in_vals[x]);
}
```



x = 0

z = z * c

z = z * c

iters = 2

x = 1

z = z * c

z = z * c

....

iters = 23

x = 2

z = z * c

z = z * c

....

iters = 255

x = 3

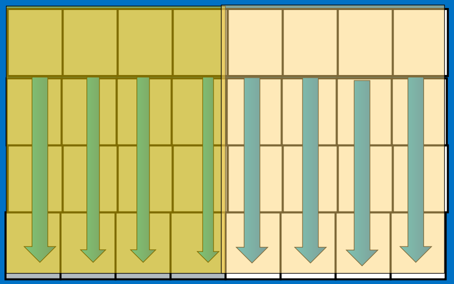
z = z * c

z = z * c

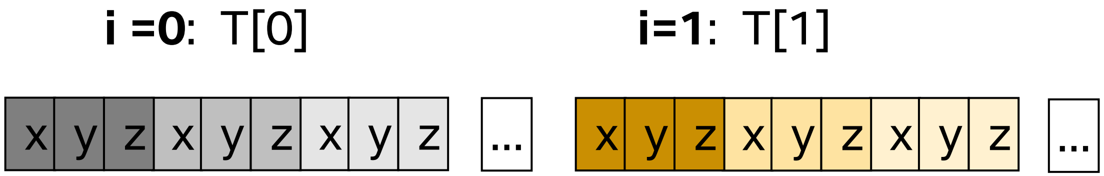
....

iters = 37

Parallelism vs. Memory



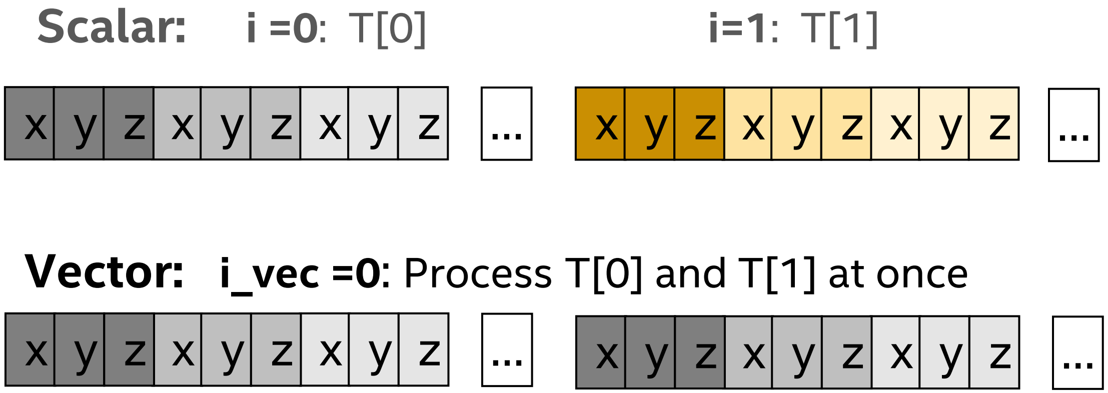
Memory Access



```
Class Point
    {
        float x,y,z;
        //some weights/colors..
    }

Class Triangle
//could be Figure, Vector,
Particle..
    {Point a,b,c;}

Triangle T[N];

void TraverseTriangles
{
    for (int i=0; i<N; i++)
        //do something with T[i]
}
```

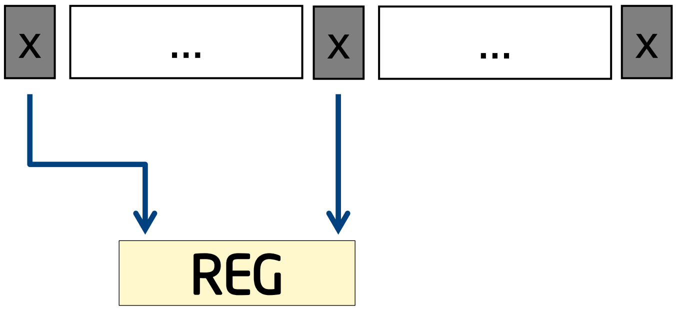
Problem #1:
Memory Access Pattern

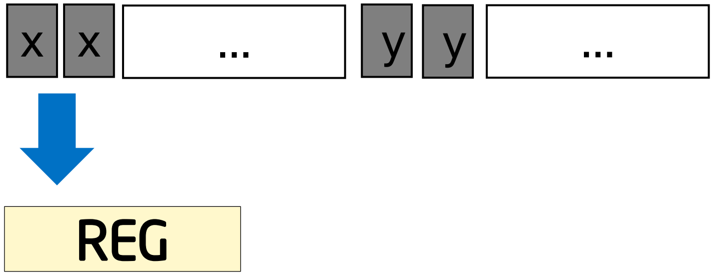
```
void TraverseTriangles
{
     #pragma omp simd simdlen(2)
     for (int i=0; i<N; i++)
         //do something with T[i]
}
```



Scalar: **i =0**: T[0] **i=1**: T[1]

Vector: **i_vec =0**: Process T[0] and T[1] at once





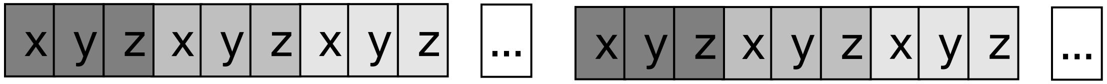
Problem #1:
non-contiguous memory access (non-unit-stride)

- Two sequential (scalar) loads into vector register. Instead of single packed load
- All memory operations (could easily be >50% of time) are **serialized**, not parallelized. **Bottleneck**.

Solution: AoS -> SoA to introduce unit stride (contiguous) access pattern

- Two values loaded in once
- No serialization, no bottleneck
- And could be more cache-friendly

Problem #1:
Memory Access Pattern. Locality.



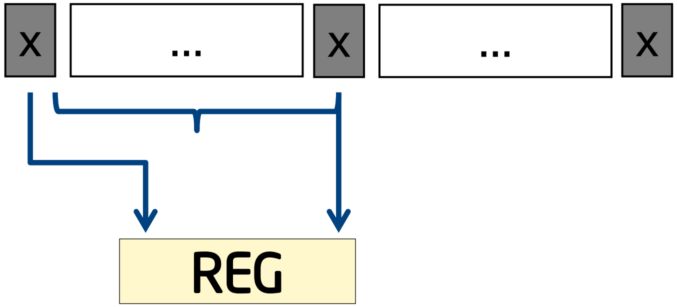
i =0: Process T[0] and T[1] at once

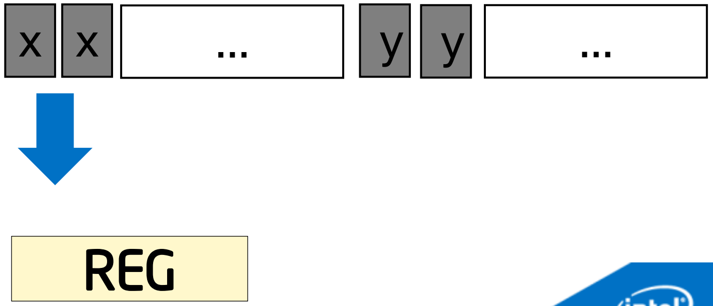
Problem #1:
non-contiguous memory access
(non-unit-stride)

- Two sequential (scalar) loads into vector register. Instead of single packed load
- All memory operations are serialized, not parallelized. Bottleneck.
- Distance between T[0].a.x and T[0].a.y

Solution: Array of Structures -> Structure of Arrays (AoS -> SoA): unit stride (linear, contiguous) Two values loaded at once

- No serialization, no bottleneck
- *Could* be more cache-friendly





Problem #2:
Locality and Bandwidth.

Assume we solved Problem #1
Problem #2:1 : What if D> L1 size, D > L2 size?
D >> L2 size (streaming..)

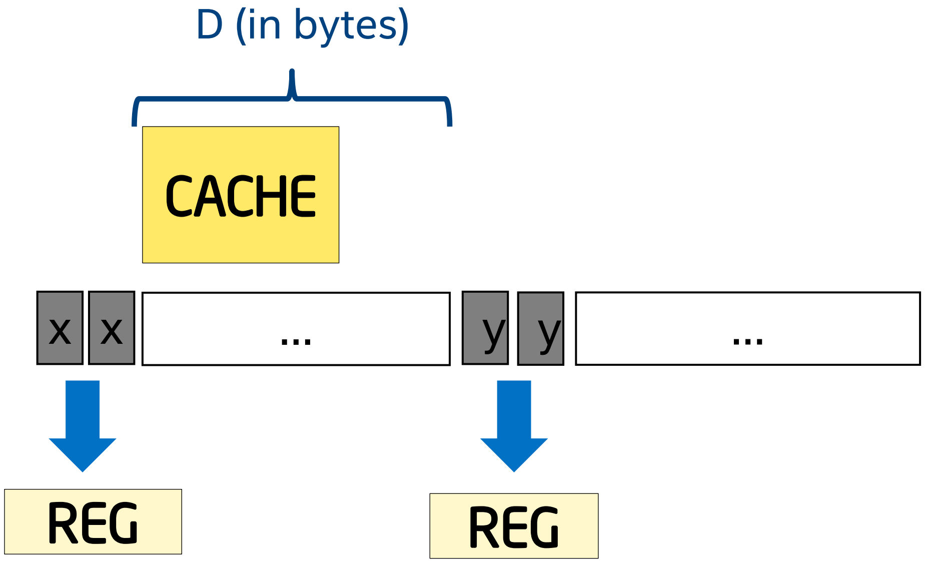
Very "expensive" memory accesses
Every next instruction leads to cache miss

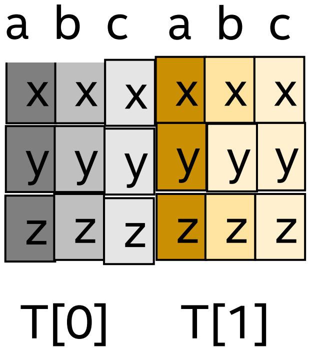
Problem #2:2 :

- Not enough computations to "amortize" bigger memory latency :
- SIMD benefits will be smaller and limited by DRAM/L3..

Possible solutions

- Array Of Structure of Arrays
- Tiling
- Pre-fetching..
- Merge kernels to make them compute-intensive, unrolling





```
Struct {
     float x[100];
     float y[100];
     float z[100];
} T;
```

Problem #3: Latency bound codes

Problem #3 : What if D varies unpredictably:

Variable (random) stride.

Every access could be cache miss

Data divergence -> serialization on AVX(1)

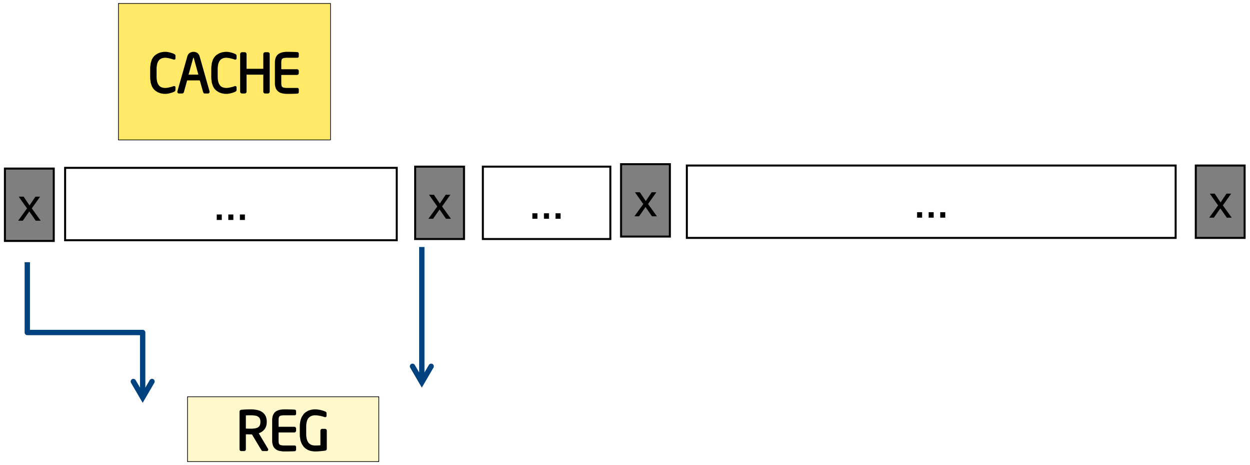
For newer ISA (AVX2): vgather (but mov* will anyway be faster)

Possible problem #3:1 :

- Substantially not enough computations to "amortize" bigger memory latency

Possible solutions:

- Know your access patterns!
- Consider vectorizing along different iteration space..
- Consider newer architectures with better data divergence support

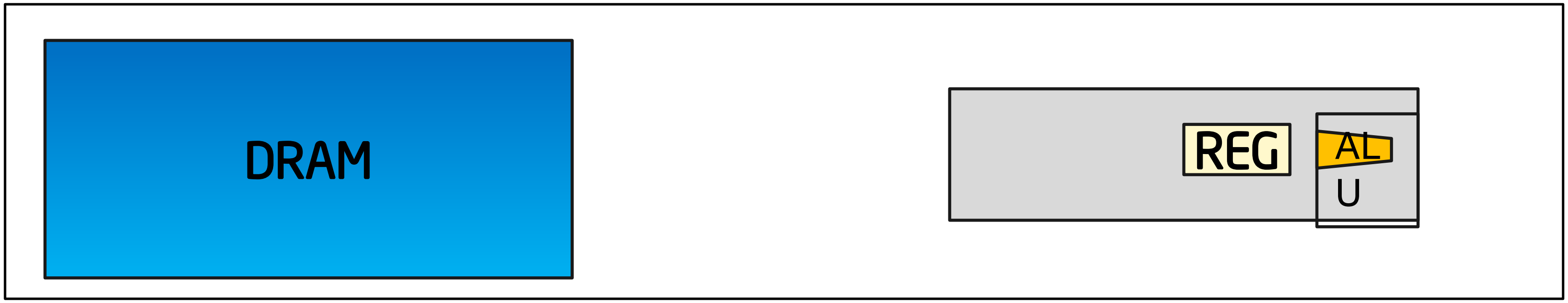


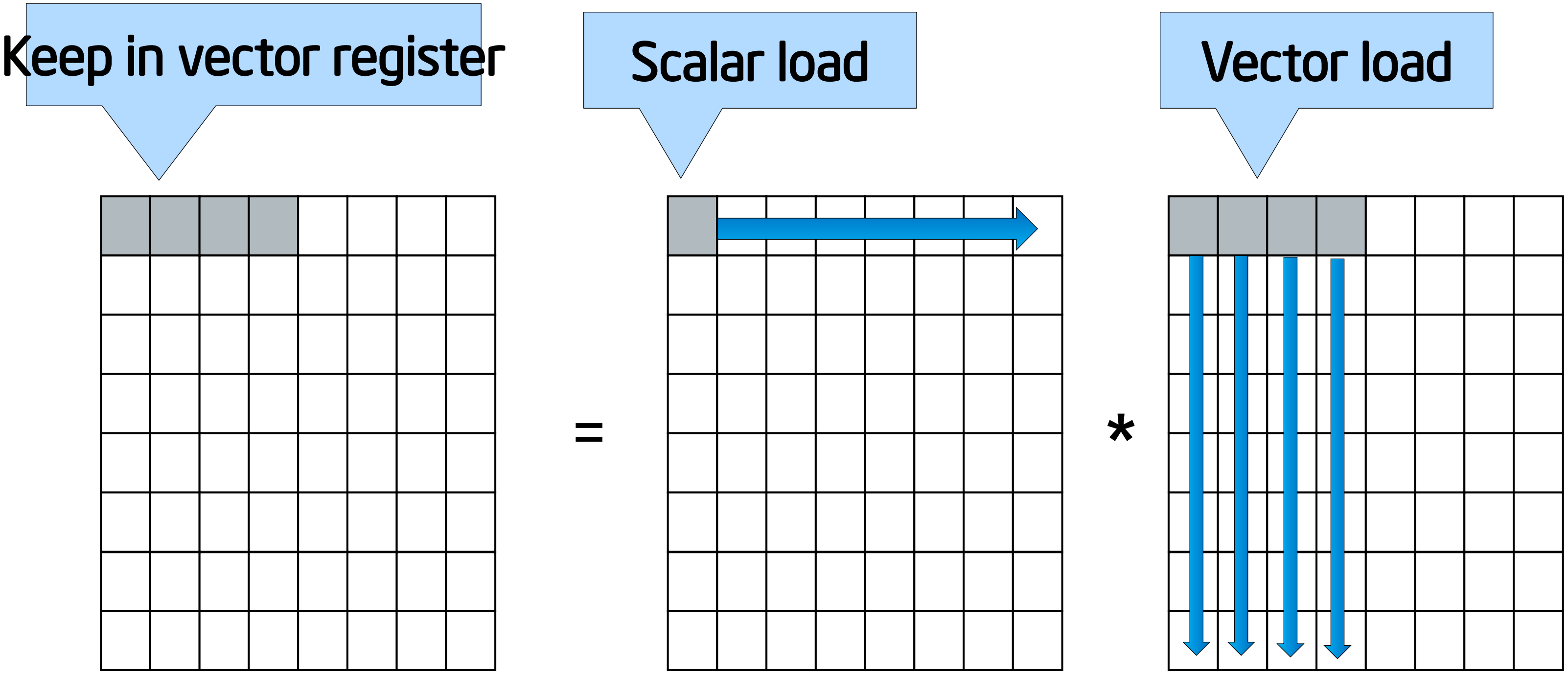
To confuse it slightly more..

And bring multi-core , NUMA on the table

Geoff Lowney, Intel Fellow:

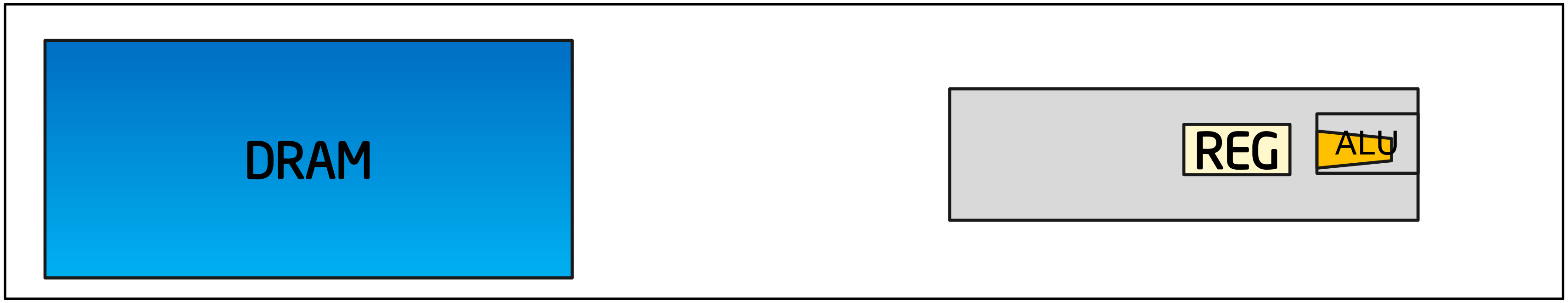
SIMD workshop keynote examples

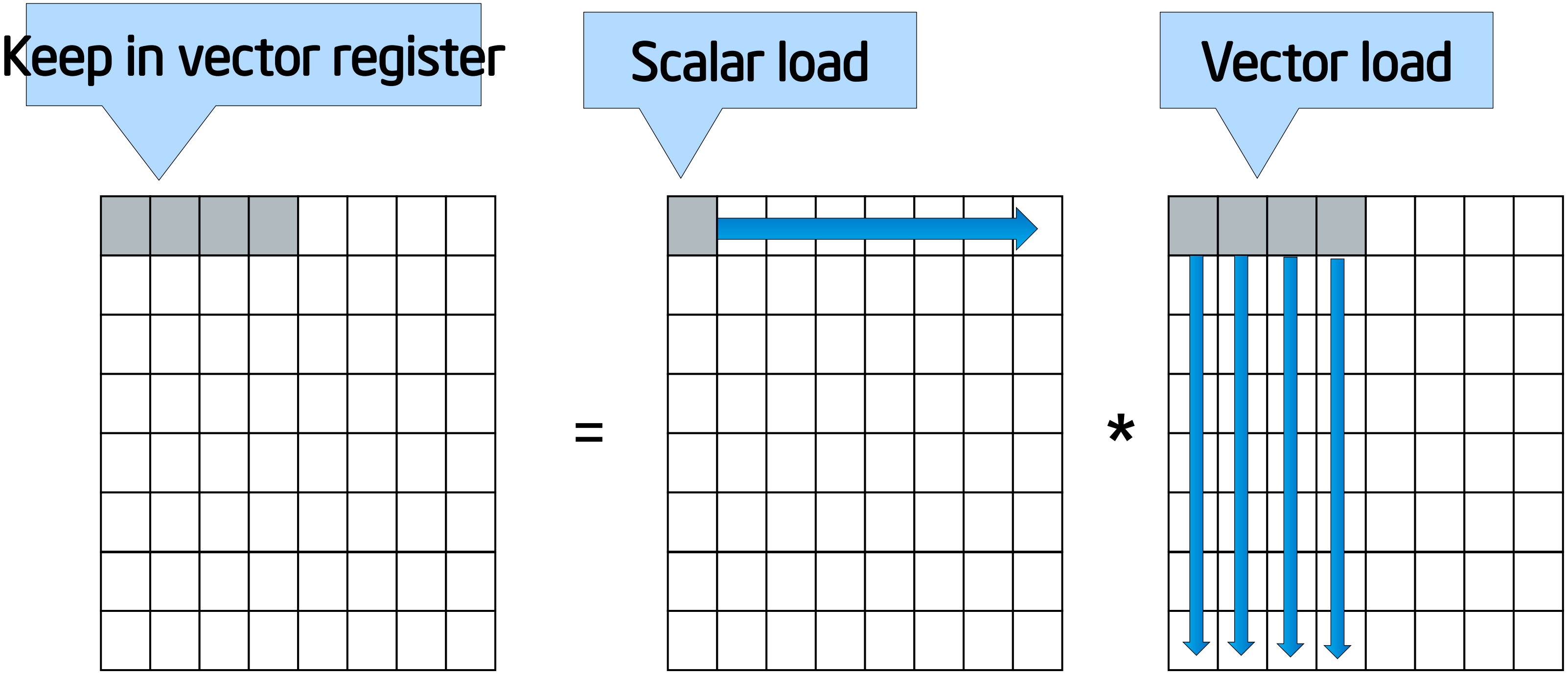




Performance limited by memory bandwidth

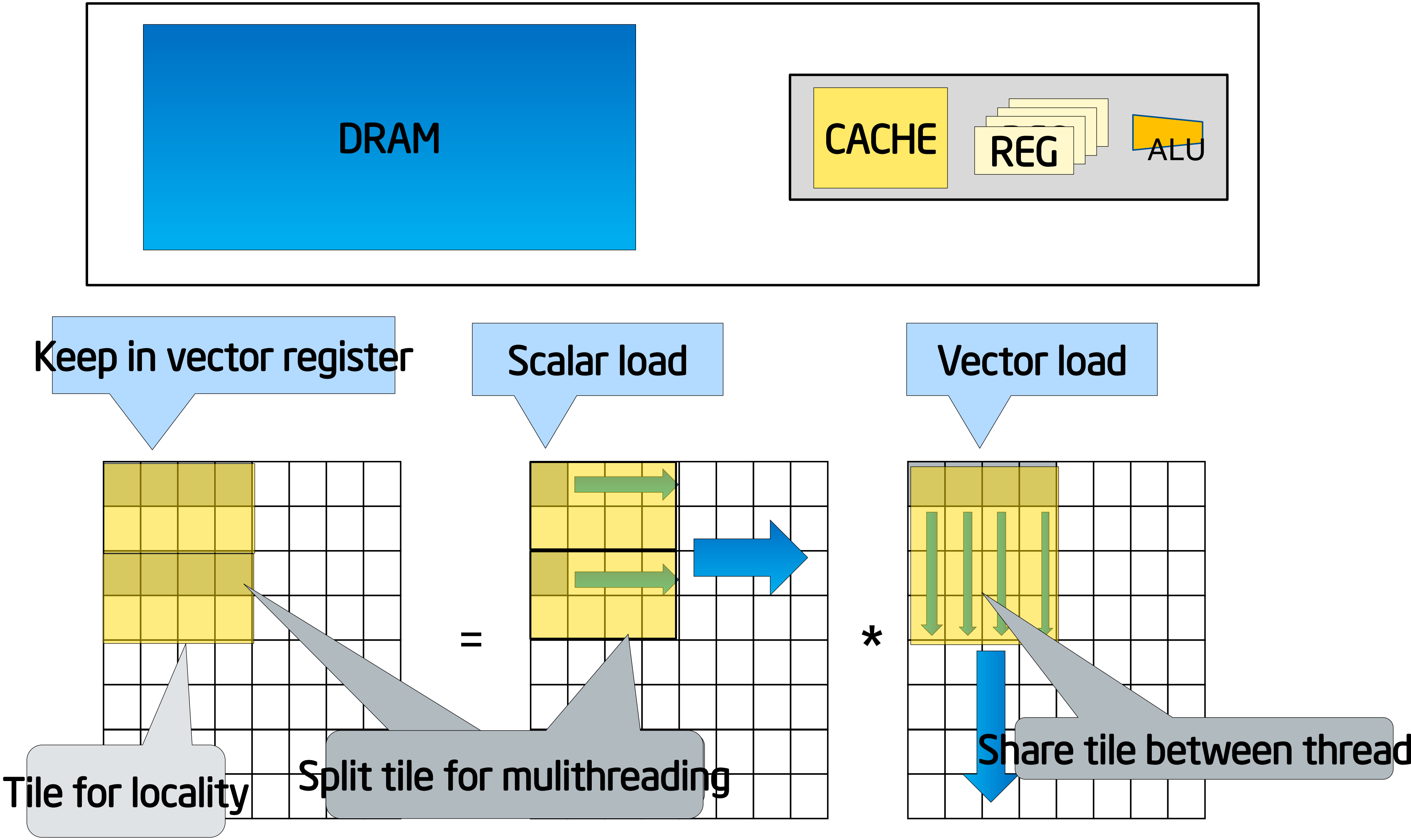
A SIMD code generation strategy





Performance limited by memory bandwidth

A SIMD code generation strategy



DRAM

CACHE

REG

ALU

Keep in vector register

Scalar load

Vector load

=

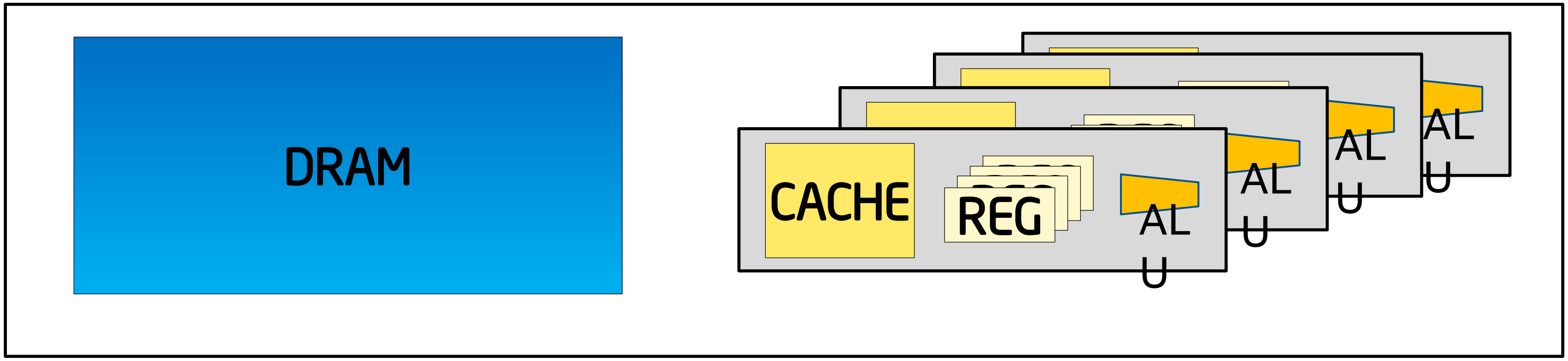
*

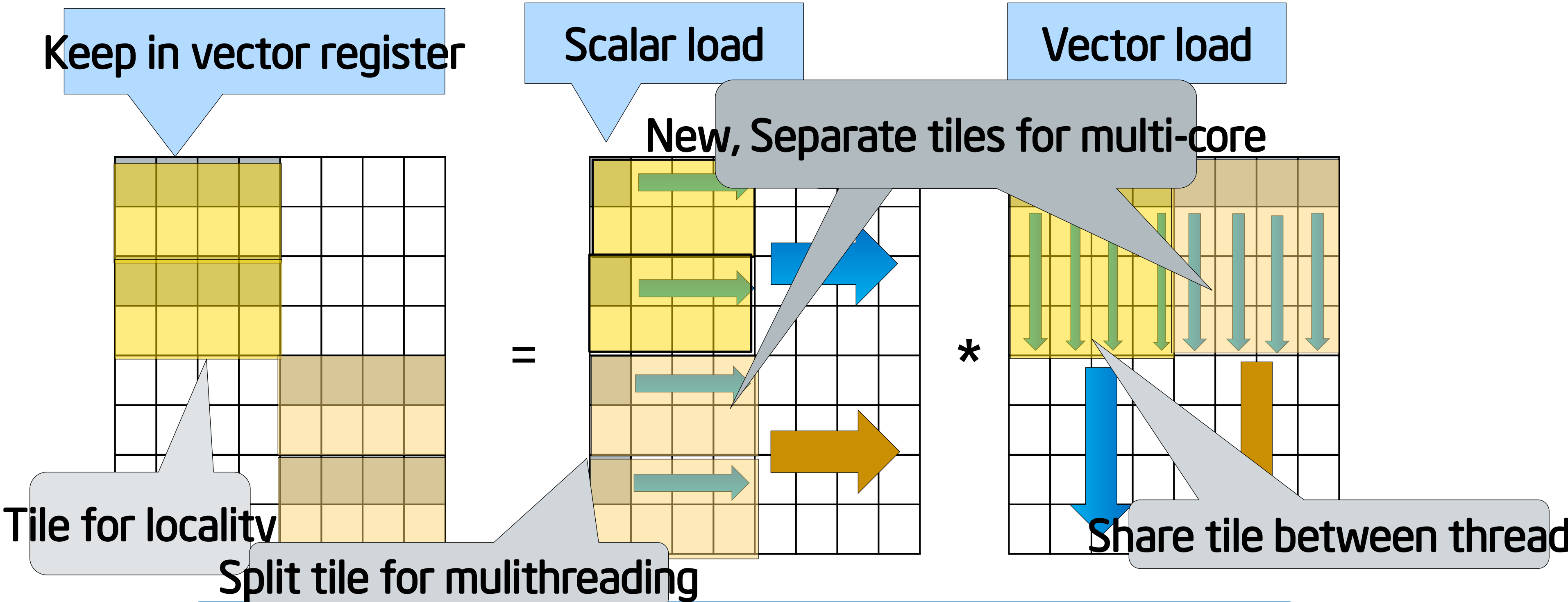
Tile for locality

Split tile for mulithreading

Share tile between thread

A hardware multi-threading, cache and SIMD code generation strategy





Keep in vector register

Scalar load

Vector load

New, Separate tiles for multi-core

Tile for locality

Split tile for mulithreading

Share tile between thread

A multi-core, hardware multi-threading, cache and SIMD code generation strategy

To confuse it slightly more..

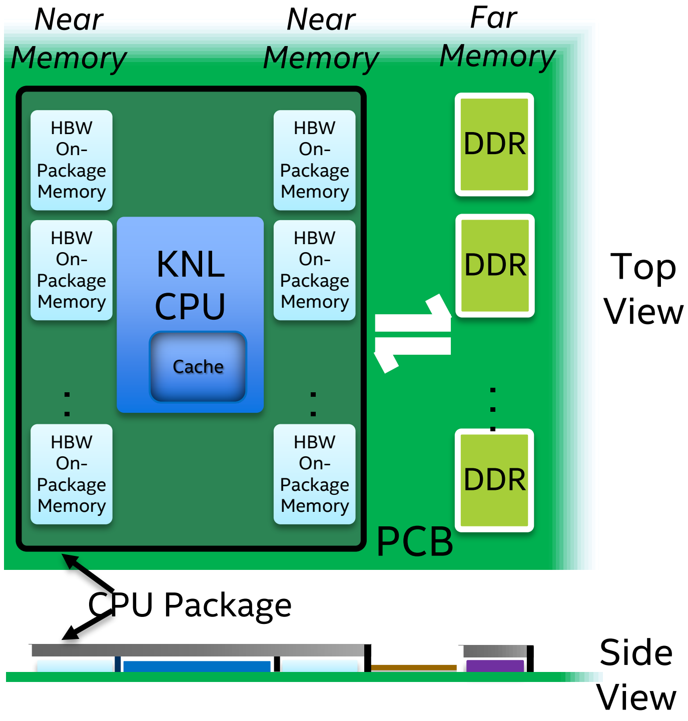
<table>
<tr><th colspan="2">High quality SIMD and threading code generation requires optimizing at least for 4 hardware features</th></tr>
<tr><td>SIMD functional units</td><td>Linear data access</td></tr>
<tr><td>Caches</td><td>Tiled data access</td></tr>
<tr><td>Hardware multi-threading</td><td>Shared tiles</td></tr>
<tr><td>Multi-core</td><td>Disjoint tiles</td></tr>
</table>

Knights Landing Integrated On-Package Memory

Cache Model Let the hardware automatically manage the integrated on-package memory as an "L3" cache between KNL CPU and external DDR

Flat Model Manually manage how your application uses the integrated on-package memory and external DDR for peak performance

Hybrid Model Harness the benefits of both cache and flat models by segmenting the integrated on-package memory



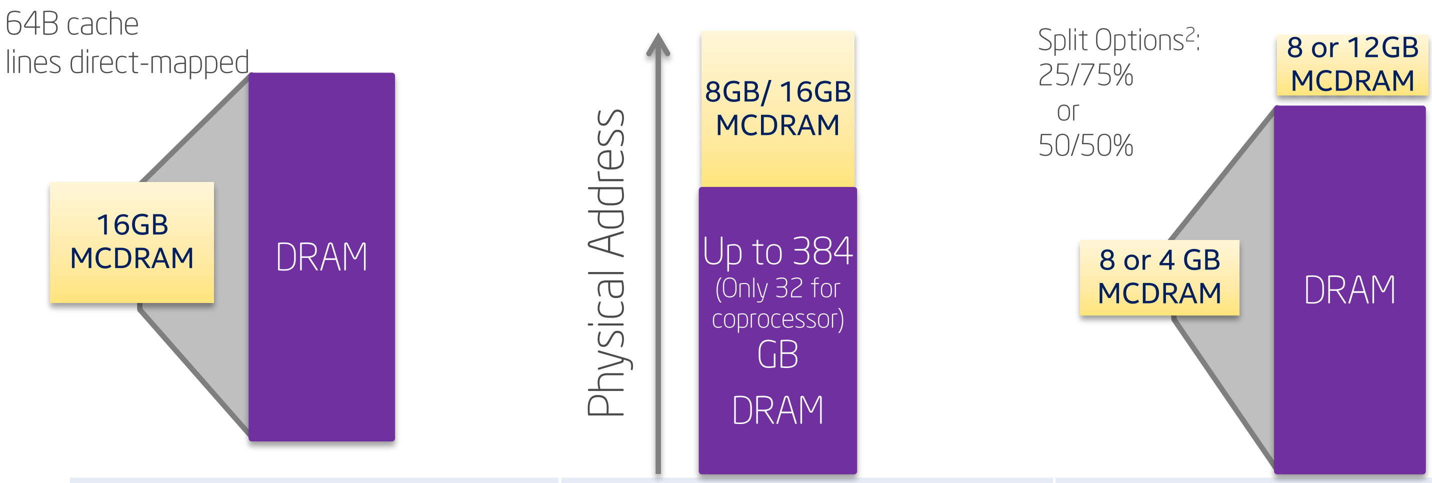
Maximizes performance through higher memory bandwidth and flexibility[1]

[1] As compared with Intel® Xeon Phi™ x100 Coprocessor Family

Diagram is for conceptual purposes only and only illustrates a CPU and memory – it is not to scale, and is not representative of actual component layout.

Integrated On-Package Memory Usage Models

Model configurable at boot time and software exposed through NUMA[1]


64B cache lines direct-mapped
16GB MCDRAM
DRAM
Physical Address
8GB/ 16GB MCDRAM
Up to 384 (Only 32 for coprocessor) GB
DRAM
Split Options[2]: 25/75% or 50/50%
8 or 12GB MCDRAM
8 or 4 GB MCDRAM
DRAM

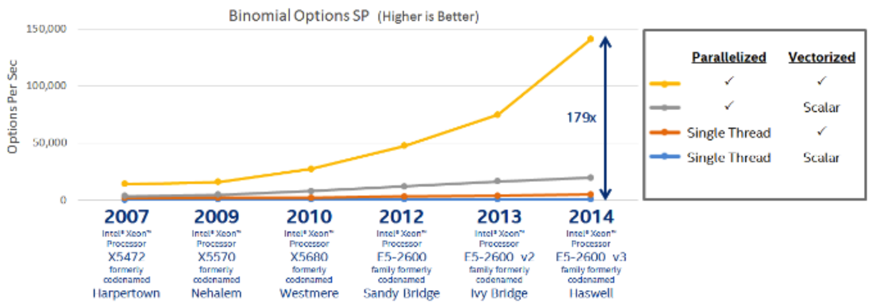
	Cache Model	Flat Model	Hybrid Model
Description	Hardware automatically manages the MCDRAM as a “L3 cache” between CPU and ext DDR memory	Manually manage how the app uses the integrated on-package memory and external DDR for peak perf	Harness the benefits of both Cache and Flat models by segmenting the integrated on-package memory
Usage Model	▪ App and/or data set is very large and will not fit into MCDRAM ▪ Unknown or unstructured memory access behavior	▪ App or portion of an app or data set that can be, or is needed to be “locked” into MCDRAM so it doesn’t get flushed out	▪ Need to “lock” in a relatively small portion of an app or data set via the Flat model ▪ Remaining MCDRAM can then be configured as Cache

Maximum flexibility for maximum performance

1. NUMA = non-uniform memory access
2. As projected based on early product definition

Back-up

Configurations for Binomial Options SP



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

Performance measured in Intel Labs by Intel employees

Platform Hardware and Software Configuration

Platform	Unscaled Core Frequency	Cores/ Socket	Num Sockets	L1 Data Cache	L1 I Cache	L2 Cache	L3 Cache	Memory	Memory Frequency	Memory Access	H/W Prefetchers Enabled	HT Enabled	Turbo Enabled	C States	O/S Name	Operating System	Compiler Version
Intel® Xeon™ 5472 Processor	3.0 GHZ	4	2	32K	32K	12 MB	None	32 GB	800 MHZ	UMA	Y	N	N	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
Intel® Xeon™ X5570 Processor	2.93 GHZ	4	2	32K	32K	256K	8 MB	48 GB	1333 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
Intel® Xeon™ X5680 Processor	3.33 GHZ	6	2	32K	32K	256K	12 MB	48 MB	1333 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
Intel® Xeon™ E5 2690 Processor	2.9 GHZ	8	2	32K	32K	256K	20 MB	64 GB	1600 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
Intel® Xeon™ E5 2697v2 Processor	2.7 GHZ	12	2	32K	32K	256K	30 MB	64 GB	1867 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.11.10-301.fc20	icc version 14.0.1
Codename Haswell	2.2 GHz	14	2	32K	32K	256K	35 MB	64 GB	2133 MHZ	NUMA	Y	Y	Y	Disabled	Fedora 20	3.13.5-202.fc20	icc version 14.0.1

TBB General Limits

Does not require compile-time analysis.

- Does not support fine-grained parallelism

“Help first” tasking

- Can simulate “work first” using explicit continuation passing.

Limited to direct use from C++.

- Consider doing Java, C#, and Managed C++ versions later.

Distributed memory is not supported.

- Target is desktop.

Requires more work than just sprinkling in pragmas a la OpenMP.

No support for mandatory parallelism.

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.

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