Programming and Tuning for Intel® Xeon® Processors

Dr-Ing. Michael Klemm
Software and Services Group
Intel
(michael.klemm@intel.com)

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

Optimization Notice

My Personal Disclaimer

Take this presentation with a grain of salt.

Some recipes, switches, settings, or some advice may or may not work on your system.

Please consult the manual and ask the operations team of the machine before you shoot yourself. ☺

Agenda

Intel® Platforms for HPC

Intel® Xeon® E5-2600v3 Processor Series

Programming for Intel Architecture

Controlling FP Arithmetic with Intel® Composer XE

Using Intel MPI for Performance

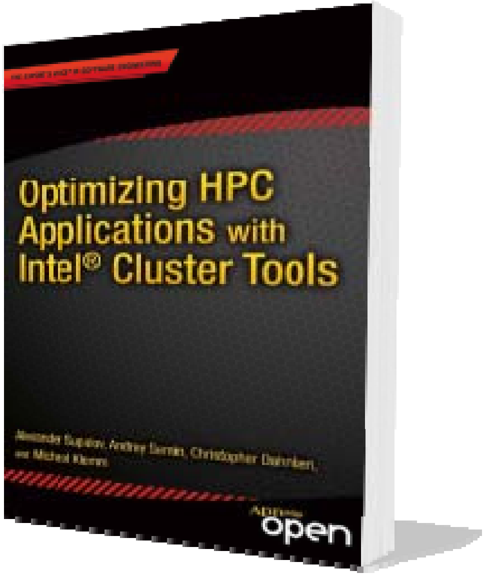
The Book of the Year… ☺ (Or: A Shameless Plug)

Apress open

Selected For Intel's Recommended Reading List

Authors: Alexander Supalov, Andrey Semin, Michael Klemm, Chris Dahnken

Table of Contents:



Optimizing HPC Applications with Intel® Cluster Tools

Alexander Supalov, Andrey Semin, Christopher Dahnken, and Michael Klemm

Apress open

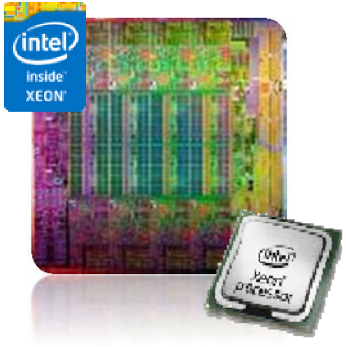
ISBN-13 (pbk): 978-1-4302-6496-5
ISBN-13 (electronic): 978-1-4302-6497-2

Order now at http://www.clusterbook.info

Intel Technologies for HPC

Processors

Intel® Xeon® Processor



Coprocessor

Intel® Many Integrated Core

Network & Fabric

I/O & Storage

Software & Services

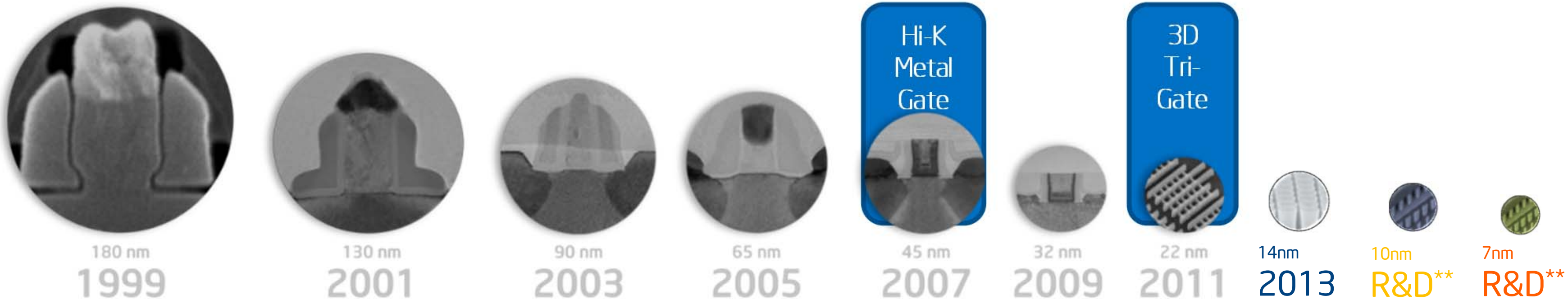
Intel Parallel Studio XE
Intel Cluster Studio XE
Intel® Cluster Ready
Intel Data Center Manager
Intel Cache Acceleration Software
Intel Enterprise Edition for Lustre* software
Intel Distribution for Apache Hadoop* software

Optimization Notice

Transforming the Economics of HPC

Executing to Moore's Law

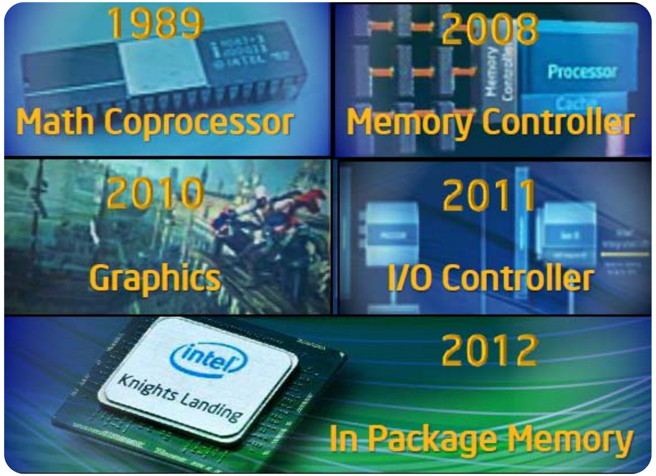
Predictable Silicon Track Record – well and alive at Intel.
Enabling new devices with higher performance and functionality while controlling power, cost, and size



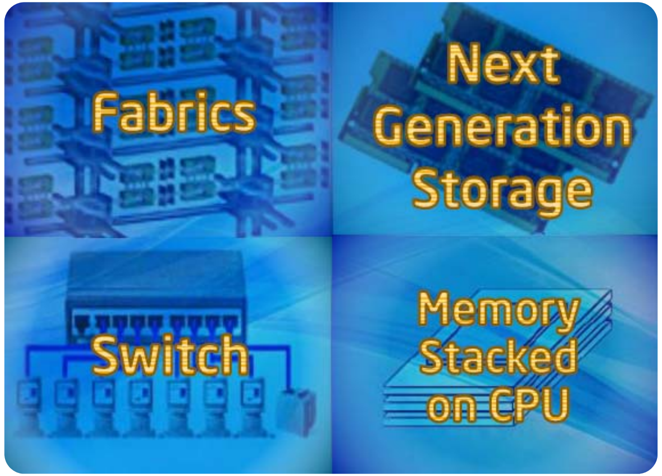
Optimization Notice

Driving Innovation and Integration

Enabled by Leading Edge Process Technologies



Integrated Today



Coming in the Future

Optimization Notice

The Magic of Integration

Moore‘s Law at Work & Architecture Innovations

Image: Rama, EPFL

6666x

1970s

150 MFLOPS

CRAY-1

2015

1000000 MFLOPS

2S Intel® Xeon® Processor

Optimization Notice

Intel® Xeon Processor Architecture

Intel "Tick-Tock" Roadmap – Part I

Intel® Core™ MicroArchitecture		Micro Architecture Codename "Nehalem"		2nd Generation Intel® Core™ Micro Architecture	3nd Generation Intel® Core™ Micro Architecture
Merom	**Penryn**	**Nehalem**	**Westmere**	**Sandy Bridge**	**Ivy Bridge**
NEW Micro architecture	**NEW Process Technology**	**NEW Micro architecture**	**NEW Process Technology**	**NEW Micro architecture**	**NEW Process Technology**
65nm	**45nm**	**45nm**	**32nm**	**32nm**	**22nm**
TOCK	***TICK***	***TOCK***	***TICK***	***TOCK***	***TICK***
2006	2007	2008	2009	2011	2012
SSSE-3	SSE4.1	SSE4.2	AES	AVX	RDRAND etc

Optimization Notice

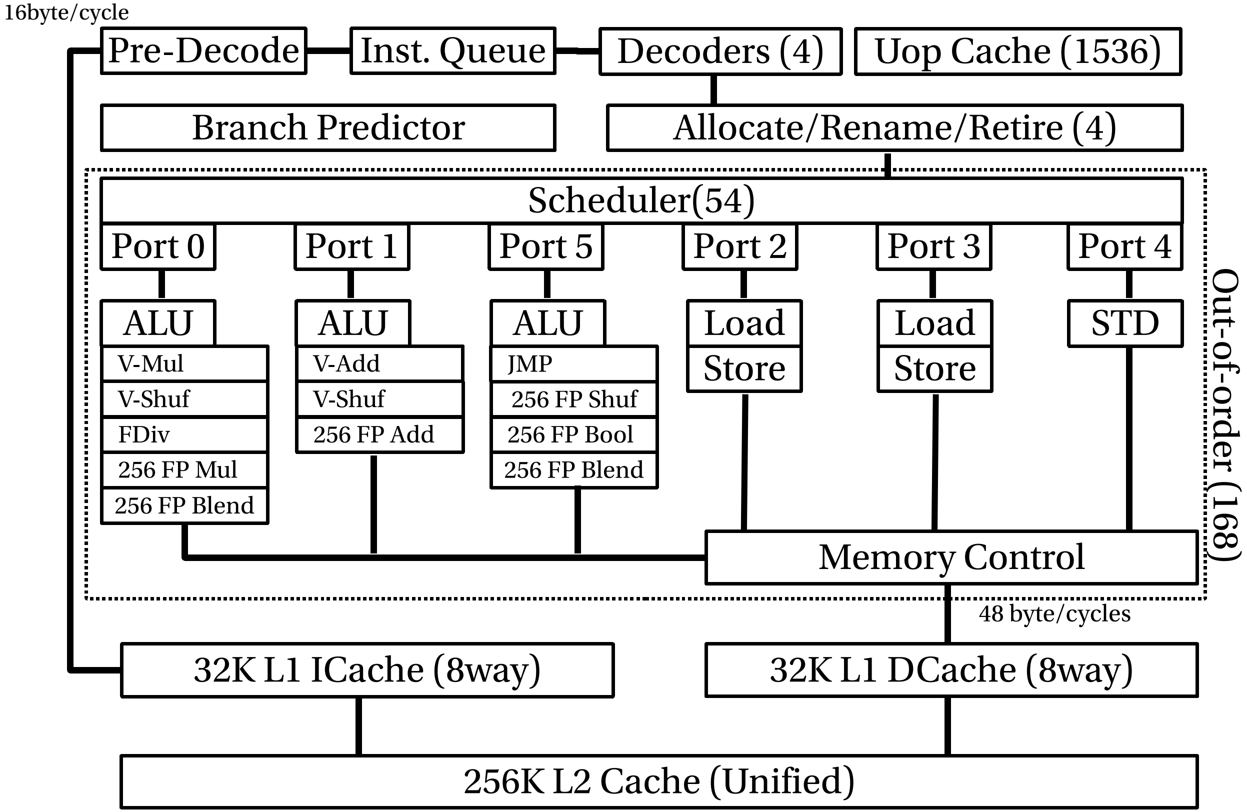
Intel "Tick-Tock" Roadmap – Part II

Future Release Dates & Features subject to Change without Notice !

4nth Generation Intel® Core™ Micro Architecture	TBD	TBD	TBD	TBD	TBD
Haswell	**Broadwell**	**Skylake**	**TBD**	**TBD**	**TBD**
NEW **Micro architecture**	**NEW** **Process Technology**	**NEW** **Micro architecture**	**NEW** **Process Technology**	**NEW** **Micro architecture**	**NEW** **Process Technology**
22nm	**14nm**	**14nm**	**10nm**	**10nm**	**7nm**
TICK	***TOCK***	***TICK***	***TOCK***	***TICK***	***TOCK***
2013	September 2014 !	2015/6 (?)	???	???	???
AVX-2	5 new Inst.	AVX-512 (?)			

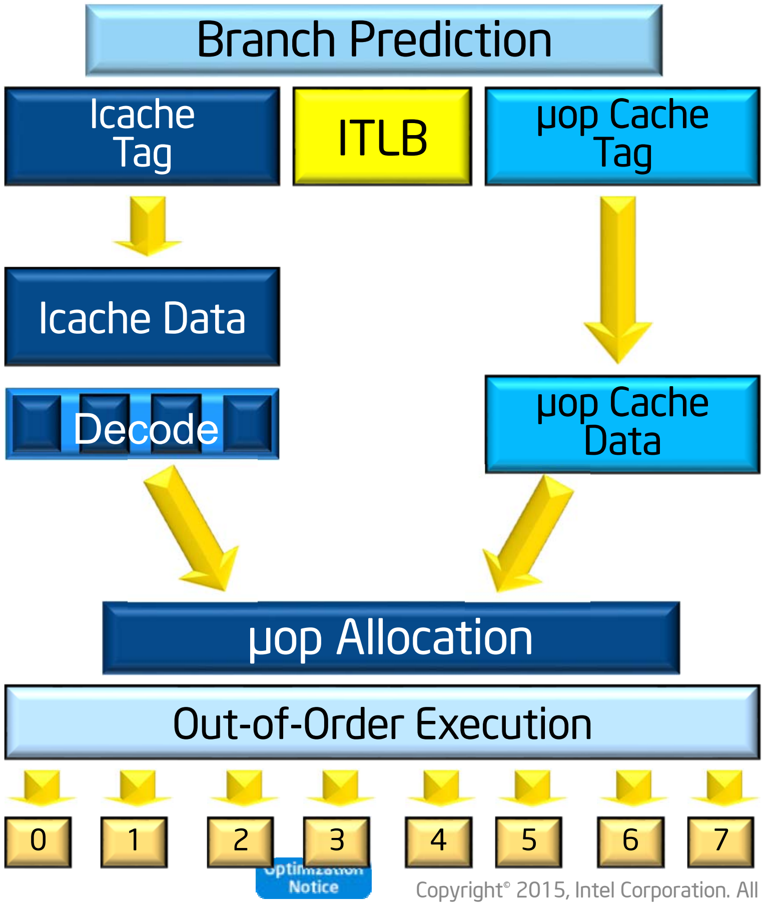
Optimization Notice

Recap: Sandy Bridge and Ivy Bridge Execution Units



Optimization Notice

Haswell Core at a Glance



Next generation branch prediction

- Improves performance *and* saves wasted work

Improved front-end

- Initiate TLB and cache misses speculatively
- Handle cache misses in parallel to hide latency
- Leverages improved branch prediction

Deeper buffers

- Extract more instruction parallelism
- More resources when running a single thread

More execution units, shorter latencies

- Power down when not in use

More load/store bandwidth

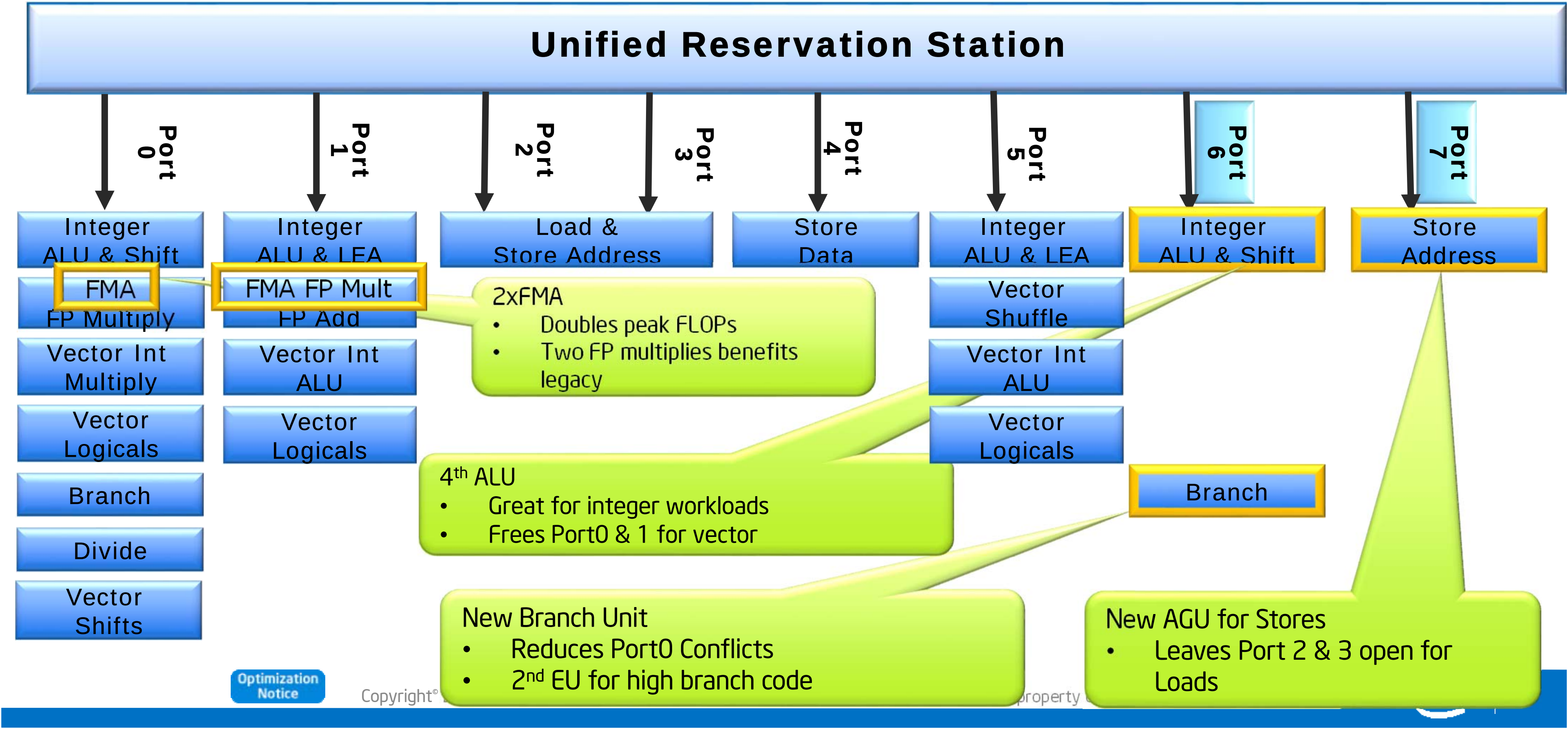
- Better prefetching, better cache line split latency & throughput, double L2 bandwidth

No pipeline growth

- Same branch misprediction latency
- Same L1/L2 cache latency

Optimization Notice

Haswell Execution Unit Overview



2xFMA

- Doubles peak FLOPs
- Two FP multiplies benefits legacy

4th ALU

- Great for integer workloads
- Frees Port0 & 1 for vector

New Branch Unit

- Reduces Port0 Conflicts
- 2nd EU for high branch code

New AGU for Stores

- Leaves Port 2 & 3 open for Loads

Haswell Buffer Sizes

Extract more parallelism in every generation

	Nehalem	Sandy Bridge	Haswell	
Out-of-order Window	128	168	192	⬆
In-flight Loads	48	64	72	⬆
In-flight Stores	32	36	42	⬆
Scheduler Entries	36	54	60	⬆
Integer Register File	N/A	160	168	⬆
FP Register File	N/A	144	168	⬆
Allocation Queue	28/thread	28/thread	56	⬆

Optimization Notice

Core Cache Size/Latency/Bandwidth

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	64 Bytes/cycle
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	32 Bytes/cycle
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	64 Bytes/cycle
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	4K+2M shared: 1024, 8-way

New Instructions in Haswell

Group		Description	Count *
AVX-2	SIMD Integer Instructions promoted to 256 bits	Adding vector integer operations to 256-bit	170 / 124
	Gather	Load elements using a vector of indices, vectorization enabler	
	Shuffling / Data Rearrangement	Blend, element shift and permute instructions	
FMA		Fused Multiply-Add operation forms (FMA-3)	96 / 60
Bit Manipulation and Cryptography		Improving performance of bit stream manipulation and decode, large integer arithmetic and hashes	15 / 15
TSX = RTM+HLE		Transactional Memory	4/4
Others		MOVBE: Load and Store of Big Endian forms INVPCID: Invalidate processor context ID	2 / 2

* Total instructions / different mnemonics

Optimization Notice

FMA: Fused Multiply Add Instruction

Improves accuracy and performance for commonly used class of algorithms

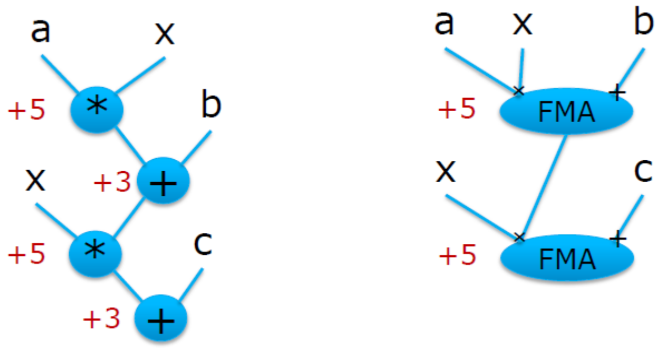
Example

FMA: Polynomial Evaluation

$$ax^2 + bx + c$$

$$=$$

$$x(ax + b) + c$$



16 cycle latency
2 cycle throughput

10 cycle latency
1 cycle throughput

Mirco-Architecture	Instruction Set	SP FLOPs per cycle	DP FLOPs per cycle
Nehalem	SSE (128-bits)	8	4
Sandy Bridge	AVX (256-bits)	16	8
Haswell	AVX2 (FMA) (256-bits)	32	16

2x peak FLOPs/cycle *(throughput)*

Latency (clocks)	Xeon E5 v2	Xeon E5 v3	Ratio *
MulPS, PD	5	5	
AddPS, PD	3	3	
Mul+Add /FMA	8	5	0.625

>37% reduced latency

(5-cycle FMA latency same as an FP multiply)

√ Increased performance potential for Technical Computing workloads like **Structural Analysis, CFD, EMF computation, Cosmology, ….** *

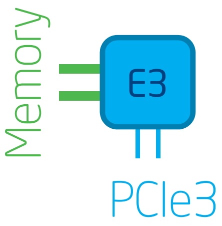
**Lower is better*

Optimization Notice

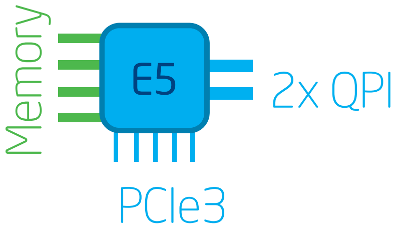
Intel® Xeon Processor Platforms

Intel® Xeon® Processors

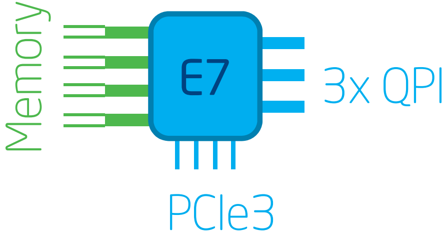
Intel® Xeon® E3



Intel® Xeon® E5

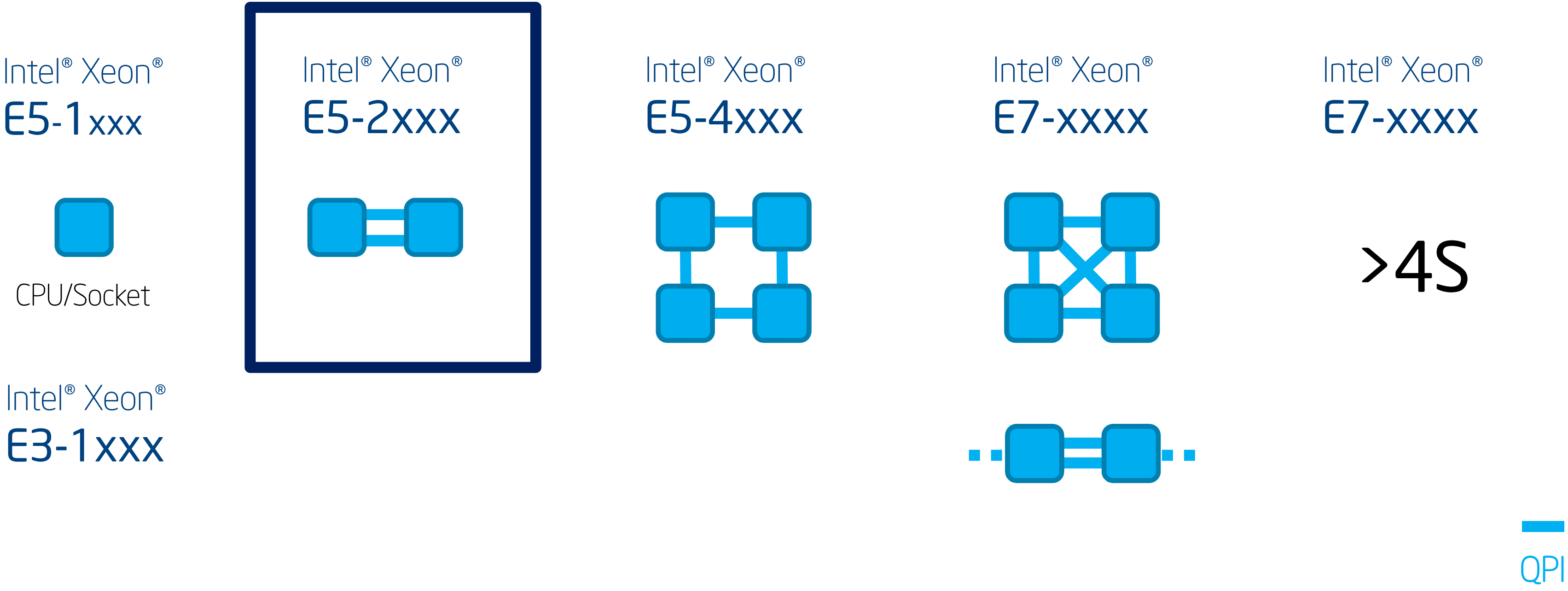


Intel® Xeon® E7



Optimization Notice

Intel® Xeon® Processors & Platforms



Intel® Xeon® E5-1xxx

CPU/Socket

Intel® Xeon® E3-1xxx

Intel® Xeon® E5-2xxx

Intel® Xeon® E5-4xxx

Intel® Xeon® E7-xxxx

Intel® Xeon® E7-xxxx

>4S

QPI

Optimization Notice

Intel® Xeon® E5-2600v3 Processor Overview

New Feature

Existing Feature

22nm Process (Tock)

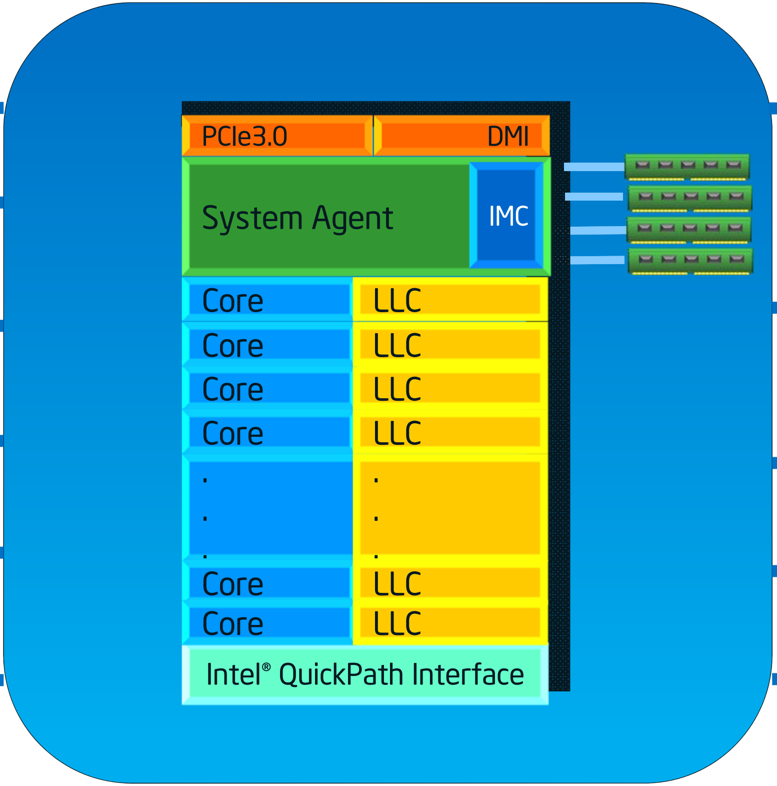
PCI Express 3.0
EP: 40 Lanes

Intel® Hyper-Threading Technology (2 threads/core)

Intel® Turbo Boost Technology

Up to 18 Cores

Integrated Voltage Regulator



Power Management
Per Core P-State (PCPS)
Uncore Frequency Scaling (UFS)
Energy Efficient Turbo (EET)

Memory Technology:
Socket R3
4xDDR4 channels
1333, 1600, 1866, 2133 MTS

~2.5 MB Last Level Cache/Core
Up to 45 MB total LLC

Intel® AVX 2.0 /
Haswell New Instruction (HNI)

Intel® QuickPath Interface (x2)
9.6GT/s

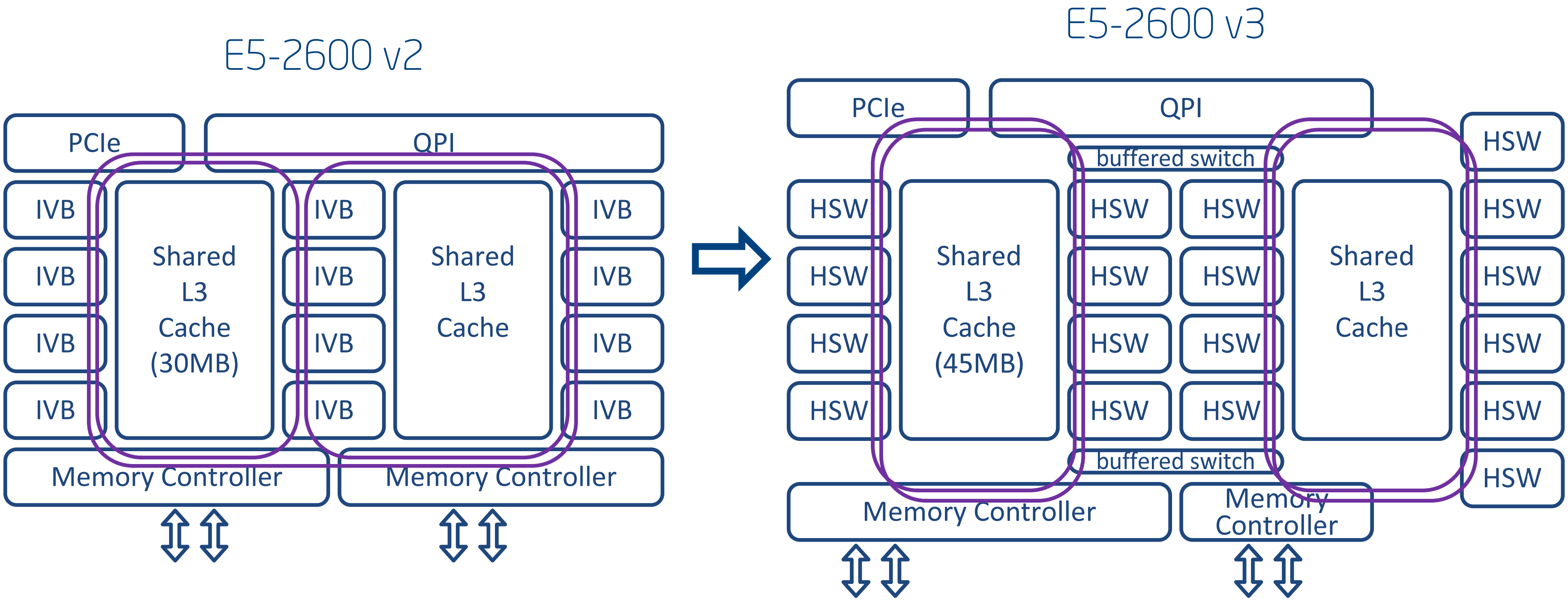
Optimization Notice

Key Differences Between E5-2600 v2 & E5-2600 v3

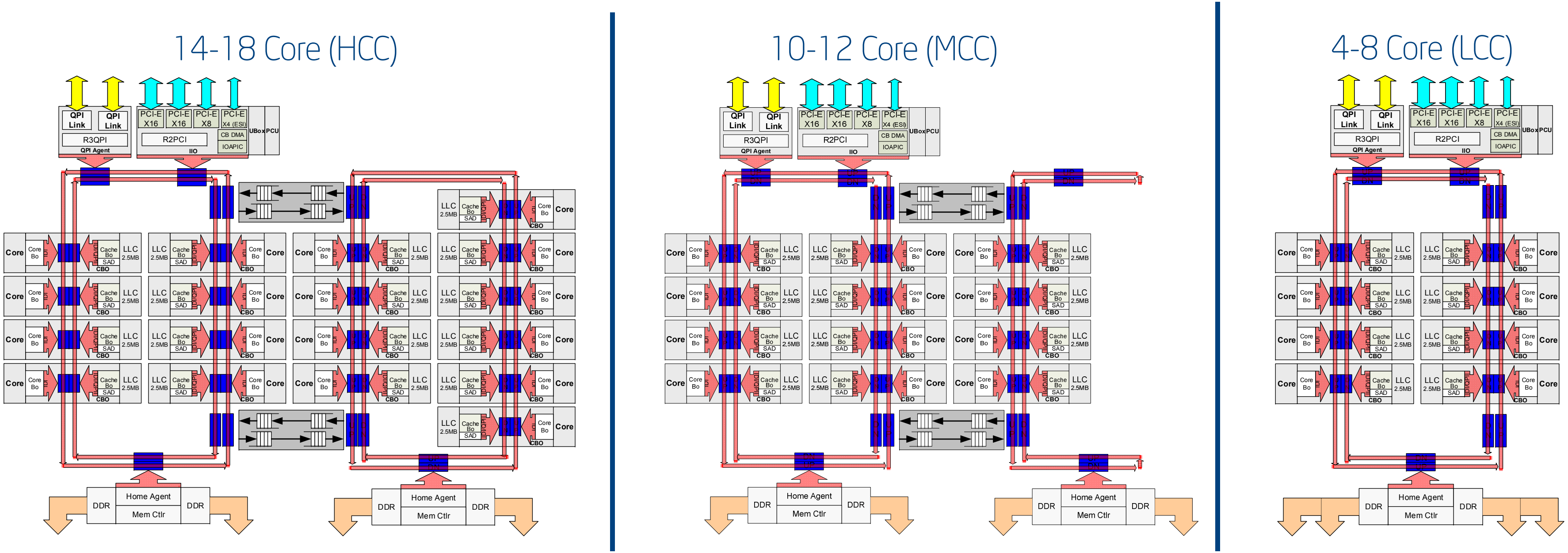
	Xeon E5-2600 v2	Xeon E5-2600 v3
Core Count	Up to 12 Cores	Up to 18 Cores
Frequency	TDP & Turbo Frequencies	TDP & Turbo Freq AVX & AVX Turbo Freq
AVX Support	AVX 1 8 DP Flops/Clock/Core	AVX 2 16 DP Flops/Clock/Core
Memory Type	4xDDR3 channels RDIMM, UDIMM, LRDIMM	4xDDR4 channels RDIMM, LRDIMM
Memory Frequency (MHz)	1866 (1DPC), 1600, 1333, 1066	RDIMM: 2133 (1DPC), 1866 (2DPC), 1600 LRDIMM: 2133 (1&2DPC), 1600
QPI Speed	Up to 8.0 GT/s	Up to 9.6 GT/s
TDP	Up to 130W Server, 150W Workstation	Up to 145W Server, 160W Workstation (Increase due to Integrated VR)
Power Management	Same P-states for all cores Same core & uncore frequency	Per-core P-states Independent uncore frequency scaling Energy Efficient Turbo

Optimization Notice

On-Die Interconnect Enhancements



Haswell EP Die Configurations



14-18 Core (HCC)

10-12 Core (MCC)

4-8 Core (LCC)

Not representative of actual die-sizes, orientation and layouts – for informational use only.

Chop	Columns	Home Agents	Cores	Power (W)	Transitors (B)	Die Area (mm^2)
HCC	4	2	14-18	110-145	5.69	662
MCC	3	2	6-12	65-160	3.84	492
LCC	2	1	4-8	55-140	2.60	354

Optimization Notice

Haswell Processor Improvements

Area	Change	Benefit
On-die interconnect	• Two Fully Buffered Rings	• Enables higher core counts and provides higher bandwidth per core.
Home Agent / Memory Controller	• DDR4 • Two Home Agents in more SKUs • Directory Cache	• Increased memory bandwidth and power efficiency • Greater socket BW with more outstanding requests • Lower average memory latency
LLC	• Cluster On Die (COD) mode • Improved LLC allocation policy • Cache Allocation Monitoring	• Increased performance, reduced latency • Enables improved performance by better application placement in a virtualized environment
Power Management	• Separate clock and voltage domains for each core and uncore (enables PCPS, UFS)	• Better performance per watt • Lower socket idle (package C6) power.
QPI 1.1	• Increase to 9.6GT/s	• Multi-socket coherence performance
Integrated IO-Hub (IIO)	• LLC cache tracks IIO cache line ownership • Increased PCIe buffers and credits	• Improves PCIe bandwidth under conflicts (concurrent accesses to the same cache line). • Increase PCIe bandwidth and latency tolerance
PCI Express 3.0	• DualCast - Allows a single write transaction to multiple targets. • Relaxed ordering	• Utilized to minimize memory channel bandwidth – data can be sent to memory and on the NTB port. Storage applications are typically memory bandwidth limited.

DDR4 Benefits

Lower Power

- Lower voltage (1.5v -> 1.2v) DIMMs
- Smaller page size (1024 -> 512) for x4 devices
- Initial results show savings of ~2W per DIMM at the wall.

Improved RAS

- Command/Address Parity error recovery

Higher bandwidth

- 14% higher STREAM results (DDR4-2133 vs. DDR3-1866)
- Increased DIMM frequency when multiple DIMMs per channel are installed

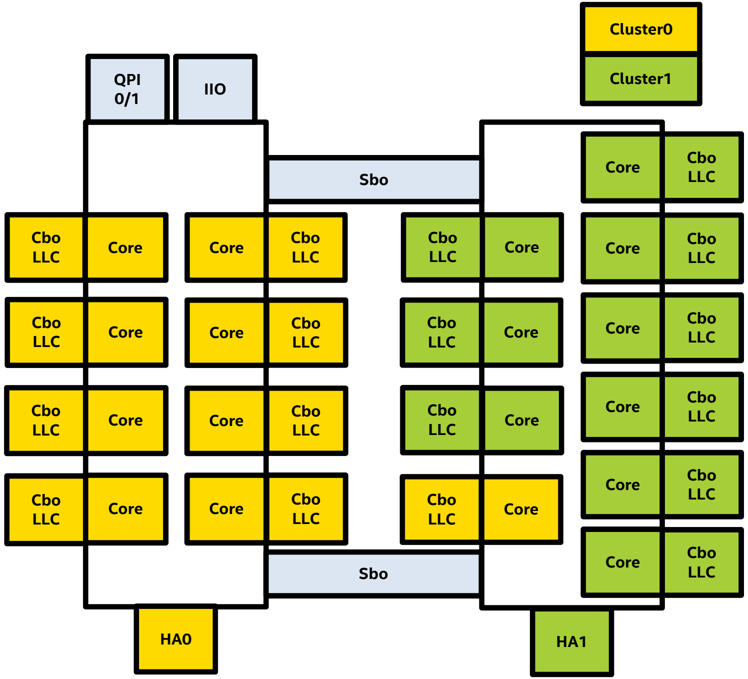
Dimms / Ch	DDR3 1.5v	DDR3 1.35v	DDR4 RDIMM	DDR4 LRDIMM
1	1866	1600	2133	2133
2	1600	1333	1866	2133
3	1066	800	1600	1600

Optimization Notice

Cluster on Die (COD) Mode

- Supported on 1S & 2S SKUs with 2 Home Agents (10+ cores)
- In memory directory bits & directory cache used on 2S to reduce coherence traffic and cache-to-cache transfer latencies
- Targeted at NUMA optimized workloads where latency is more important than sharing across Caching Agents
  - Reduces average LLC hit and local memory latencies
  - HA sees most requests from reduced set of threads potentially offering higher effective memory bandwidth
- OS/VMM own NUMA and process affinity decisions

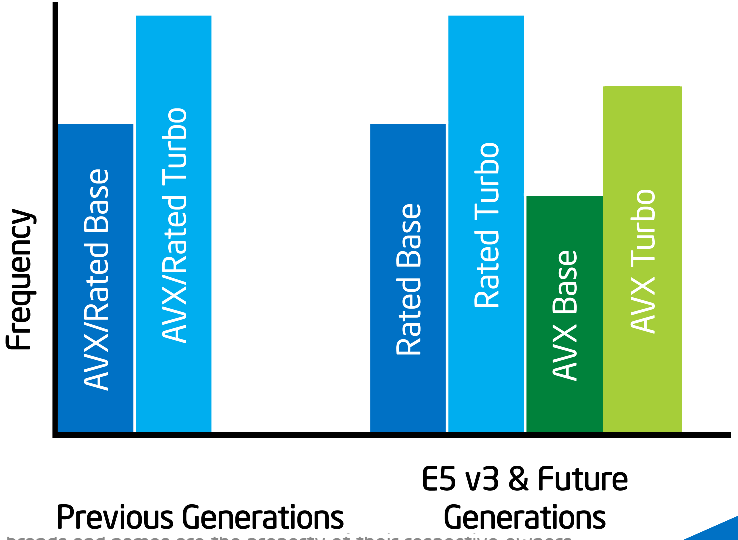
COD Mode for 18C E5-2600 v3



Optimization Notice

Intel® Turbo Boost Technology 2.0 and Intel® AVX*

- Intel® Turbo Boost Technology 2.0 automatically allows processor cores to run faster than the Rated and AVX base frequencies if they're operating below power, current, and temperature specification limits.
- Amount of turbo frequency achieved depends on the type of workload, number of active cores, estimated current & power consumption, and processor temperature
- Due to workload dependency, separate AVX base & turbo frequencies will be defined for Xeon® processors starting with E5 v3 product family



Optimization Notice

How does frequency change with AVX workloads?

Core detects presence of AVX instructions

- AVX instructions draw more current & higher voltage is needed to sustain operating conditions

Core signals to Power Control Unit (PCU) to provide additional voltage & core slows the execution of AVX instructions

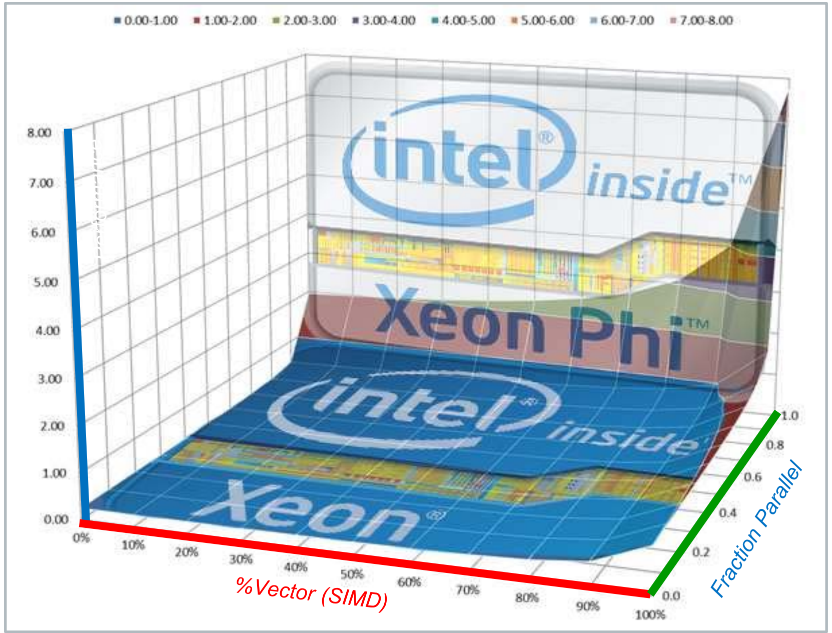
- Need to maintain TDP limits, so increasing voltage may cause frequency drop
- Amount of frequency drop will depend on workload power & AVX frequency limits

PCU signals that the voltage has been adjusted & core returns to full execution throughput

PCU returns to regular (non-AVX) operating mode 1ms after AVX instructions are completed

Programming for Intel Architecture

Highly Parallel Applications



Theoretical acceleration of a highly parallel processor over a Intel® Xeon® parallel processor (<1: Intel® Xeon® faster) – For illustration only

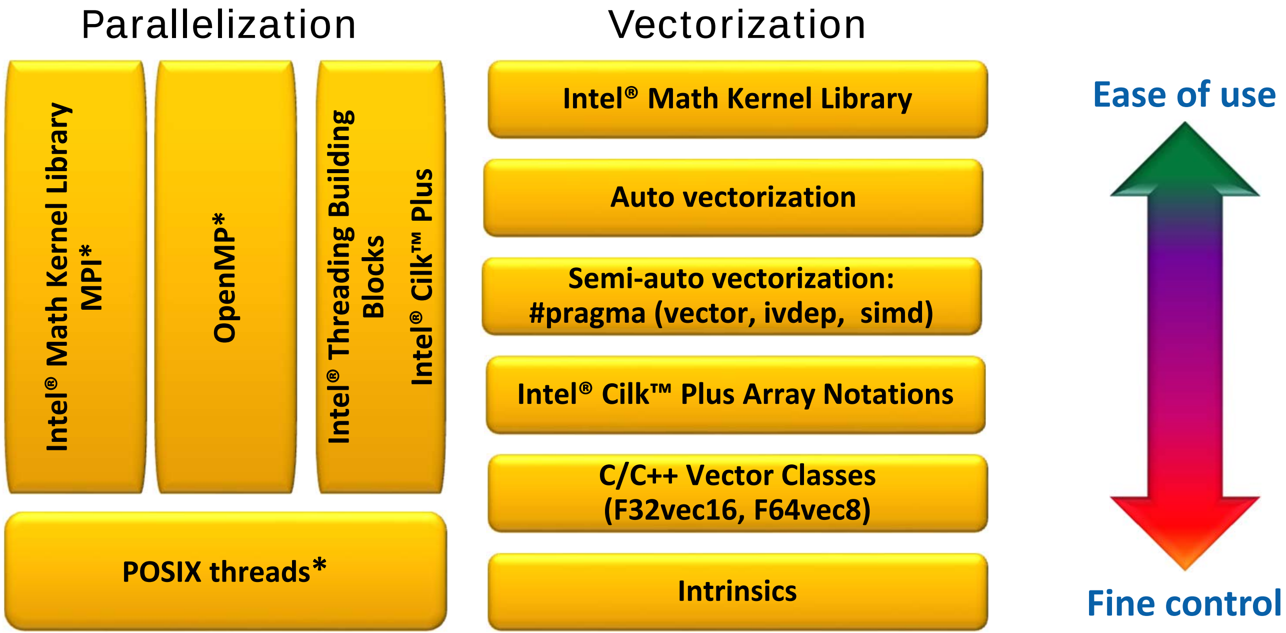
Efficient vectorization, threading, and parallel execution drives higher performance for suitable scalable applications

Parallel Programming for Intel® Architecture

NODES	Use Intel® MPI, Co-Array Fortran
CORES	Use threads directly (pthreads) or via OpenMP*, C++11 Use tasking, Intel® TBB / Cilk™ Plus
VECTORS	Intrinsics, auto-vectorization, vector-libraries Language extensions for vector programming (SIMD)
BLOCKING	Use caches to hide memory latency Organize memory access for data reuse
DATA LAYOUT	Structure of arrays facilitates vector loads / stores, unit stride Align data for vector accesses

Parallel programming to utilize the hardware resources, in an abstracted and portable way

Programming for Intel Procesors



Parallelization

Intel® Math Kernel Library MPI*

OpenMP*

Intel® Threading Building Blocks

Intel® Cilk™ Plus

POSIX threads*

Vectorization

Intel® Math Kernel Library

Auto vectorization

Semi-auto vectorization: #pragma (vector, ivdep, simd)

Intel® Cilk™ Plus Array Notations

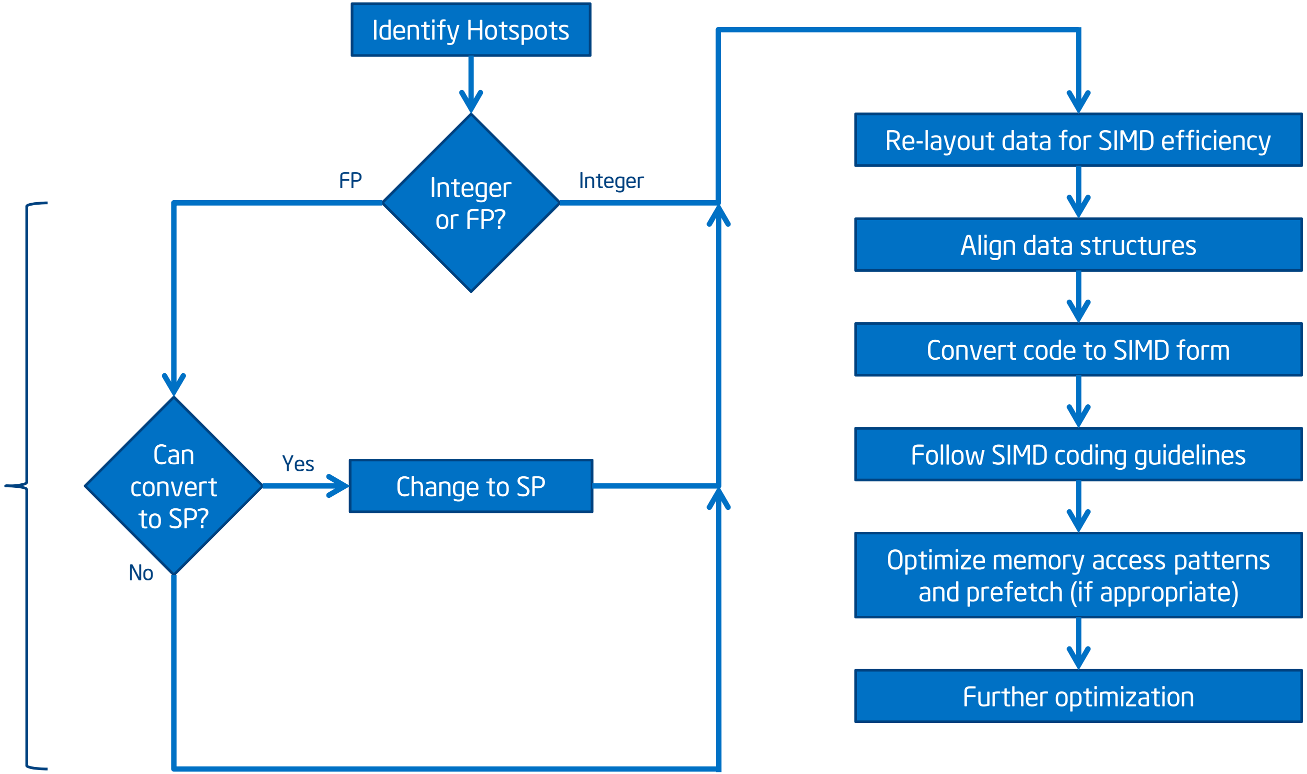
C/C++ Vector Classes (F32vec16, F64vec8)

Intrinsics

Ease of use

Fine control

Preparing Code for SIMD


Identify Hotspots
Integer or FP?
FP
Integer
Can convert to SP?
Yes
No
Change to SP
Re-layout data for SIMD efficiency
Align data structures
Convert code to SIMD form
Follow SIMD coding guidelines
Optimize memory access patterns and prefetch (if appropriate)
Further optimization

Precision is important: impacts the SIMD width.

Optimization Notice

Data Layout – Common Layouts

Array-of-Structs (AoS)

x	y	z	x	y	z
x	y	z	x	y	z
x	y	z	x	y	z

- Pros:
  Good locality of {x, y, z}.
  1 memory stream.
- Cons:
  Potential for gather/scatter.

Struct-of-Arrays (SoA)

x	x	x	x	x	x
y	y	y	y	y	y
z	z	z	z	z	z

- Pros:
  Contiguous load/store.
- Cons:
  Poor locality of {x, y, z}.
  3 memory streams.

Hybrid (AoSoA)

x	x	y	y	z	z
x	x	y	y	z	z
x	x	y	y	z	z

- Pros:
  Contiguous load/store.
  1 memory stream.
- Cons:
  Not a “normal” layout.

Optimization Notice

Data Layout – Why It's Important

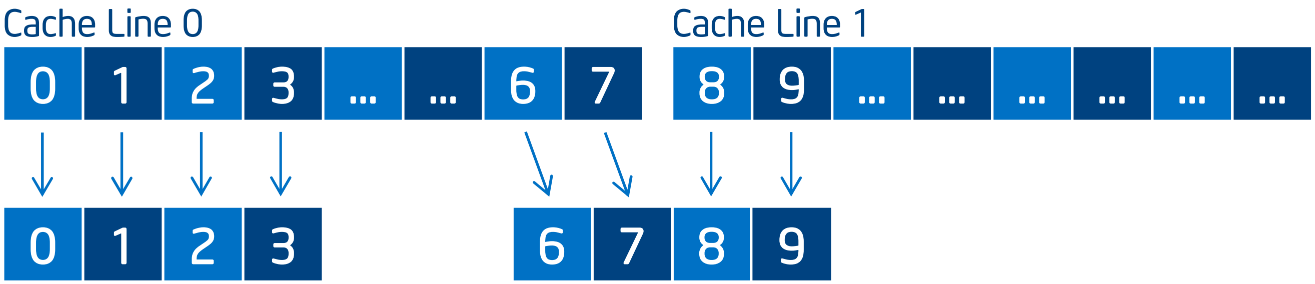
Instruction-Level

- Hardware is optimized for contiguous loads/stores.
- Support for non-contiguous accesses differs with hardware. (e.g., AVX2/KNC gather)

Memory-Level

- Contiguous memory accesses are cache-friendly.
- Number of memory streams can place pressure on prefetchers.

Data Alignment – Why It's Important



Cache Line 0

0	1	2	3	…	…	6	7

Cache Line 1

8	9	…	…	…	…	…	…

0	1	2	3

6	7	8	9

Aligned Load

- Address is aligned.
- One cache line.
- One instruction.

Unaligned Load

- Address is not aligned.
- Potentially multiple cache lines.
- Potentially multiple instructions.

Data Alignment – Sample Applications

1) Align Memory

- _mm_malloc(bytes, 64) / !dir$ attributes align:64

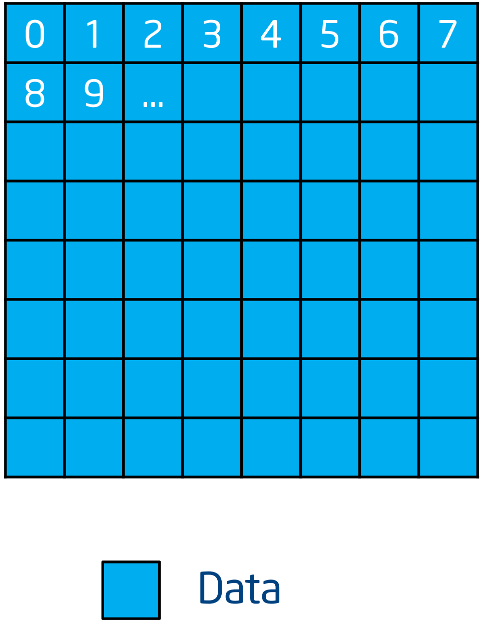
2) Access Memory in an Aligned Way

- for (i = 0; i < N; i++) { array[i] … }

3) Tell the Compiler

- #pragma vector aligned / !dir$ vector aligned
- __assume_aligned(p, 16) / !dir$ assume_aligned (p, 16)
- __assume(i % 16 == 0) / !dir$ assume (mod(i, 16) .eq. 0)

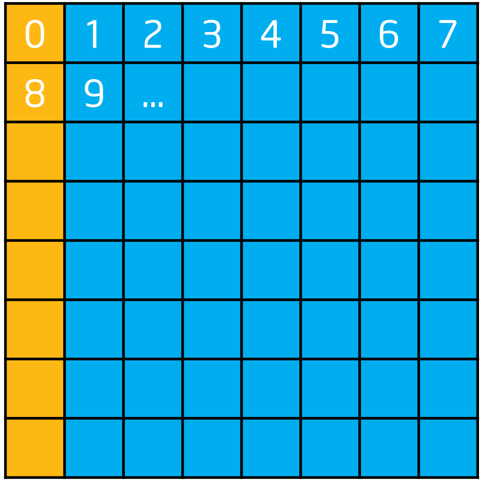
Data Alignment – Real-life Applications



0	1	2	3	4	5	6	7
8	9	…					

🟦 Data

Data Alignment – Real-life Applications

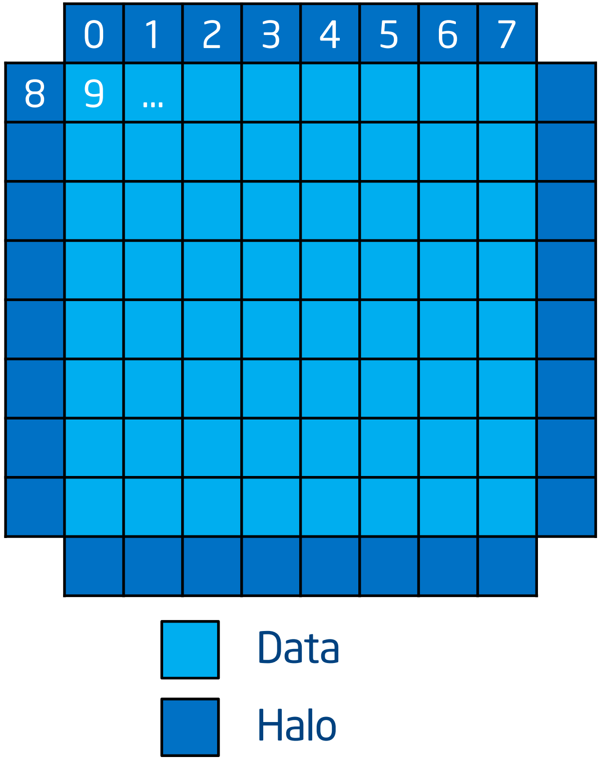


0	1	2	3	4	5	6	7
8	9	…					

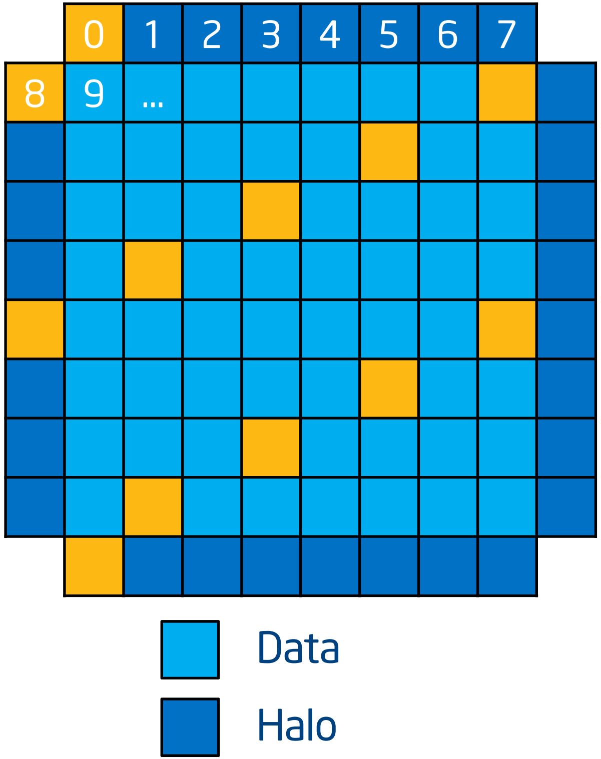
🟦 Data

Optimization Notice

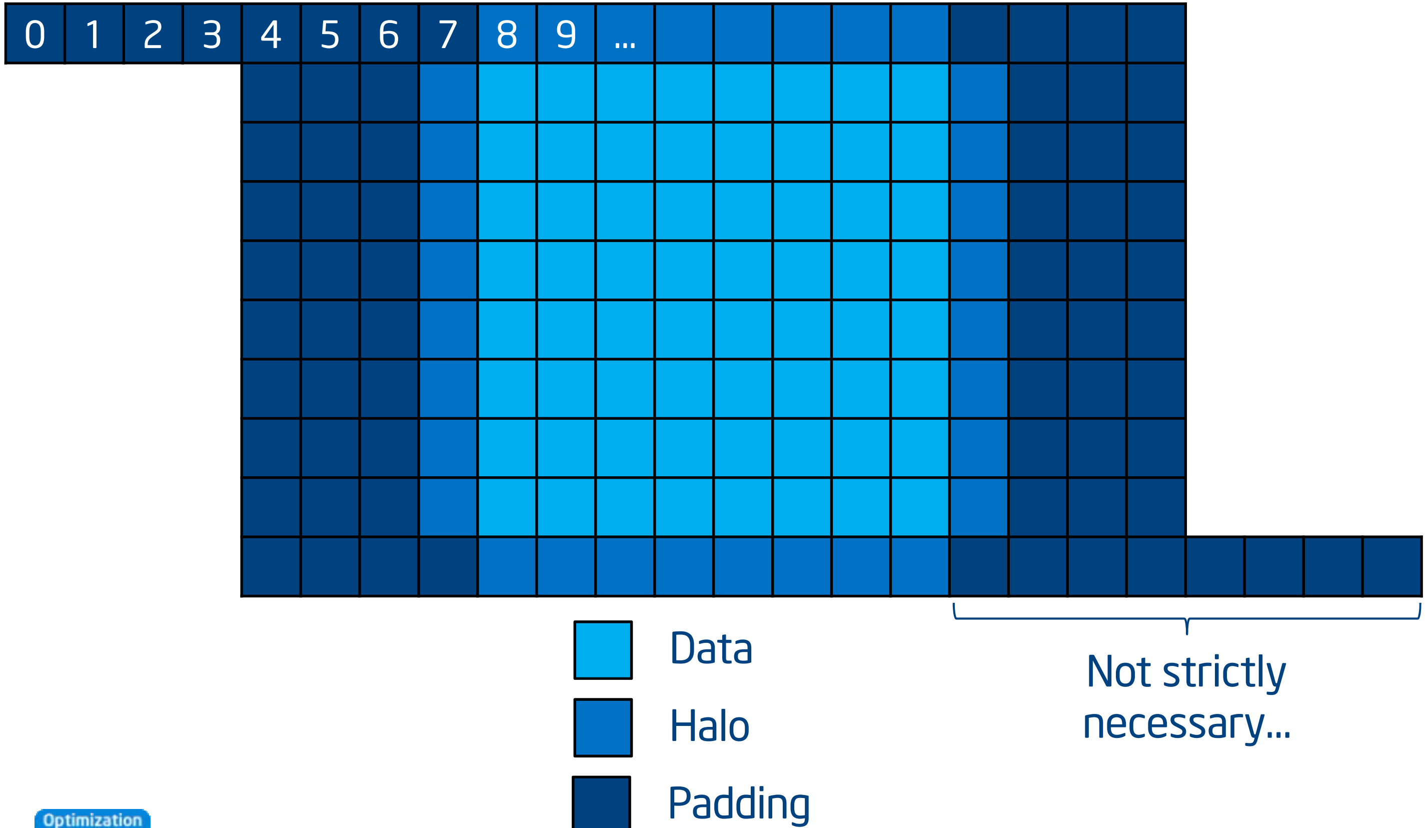
Data Alignment – Real-life Applications



Data Alignment – Real-life Applications

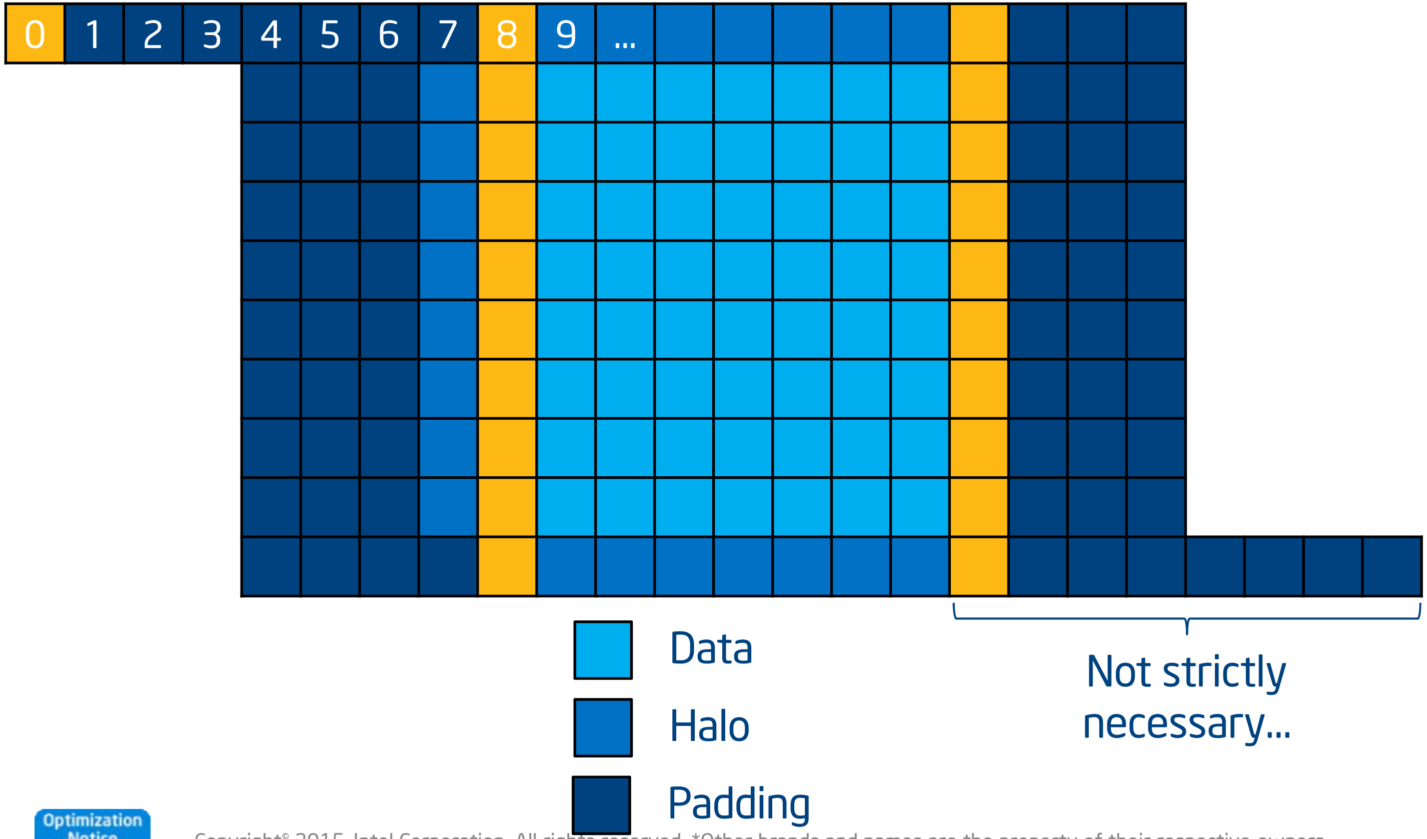


Data Alignment – Real-life Applications



Optimization Notice

Data Alignment – Real-life Applications



Not strictly necessary...

Implicit Vectorization

- Very powerful, but a compiler cannot make unsafe assumptions.

```
int* g_size;

void not_vectorizable
(float* a, float* b, float* c, int* ind) {
      for (int i = 0; i < *g_size; i++) {
            int j = ind[i];
            c[j] += a[i] + b[i];
      }
}
```

- Unsafe Assumptions:
  - a, b and c point to different arrays.
  - Value of global g_size is loop-invariant.
  - ind[i] is a one-to-one mapping.

Use the Compiler's Optimization Report

```
Begin optimization report for: not_vectorizable(float *, float *, float *, int *)

    Report from: Interprocedural optimizations [ipo]

INLINE REPORT: (not_vectorizable(float *, float *, float *, int *)) [1] vectorize.cc(4,63)


    Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]


LOOP BEGIN at vectorize.cc(5,9)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization. First
dependence is shown below. Use level 5 report for details
   remark #15346: vector dependence: assumed ANTI dependence between  line 7 and  line 7
   remark #25439: unrolled with remainder by 2
LOOP END

LOOP BEGIN at vectorize.cc(5,9)
<Remainder>
LOOP END
```

Optimization Notice

Implicit Vectorization

- Very powerful, but a compiler cannot make unsafe assumptions.

```
int* g_size;

void vectorizable
(float* restrict a, float* restrict b, float* restrict c, int* restrict ind) {
      int size = *g_size;
      #pragma ivdep
      for (int i = 0; i < size; i++) {
            int j = ind[i];
            c[j] += a[i] + b[i];
      }
}
```

- Safe Assumptions:
  - a, b and c point to different arrays. (restrict)
  - Value of global g_size is loop-invariant. (pointer dereference outside loop)
  - ind[i] is a one-to-one mapping. (#pragma ivdep)

Implicit Vectorization – Improving Performance

Getting code to vectorize is only half the battle

- “LOOP WAS VECTORIZED” != “the code is optimal”
- Vectorized code can be slower than the scalar equivalent.

Compiler will **always** choose correctness over performance

- “Hints” and pragmas can’t possibly cover all the situations…
- … but we can usually rewrite loop bodies to assist the compiler.

Explicit Vectorization

Compiler Responsibilities

- Allow programmer to declare that code **can** and **should** be run in SIMD.
- Generate the code the programmer asked for.

Programmer Responsibilities

- Correctness (e.g., no dependencies, no invalid memory accesses).
- Efficiency (e.g., alignment, loop order, masking).

Explicit Vectorization – Motivating Example 1

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma omp simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
        sum += *p;
        p += step;
}
```

- The two += operators have different meaning from each other.
- The programmer should be able to express those differently.
- The compiler has to generate different code.
- The variables *i*, *p* and *step* have different “meaning” from each other.

Optimization Notice

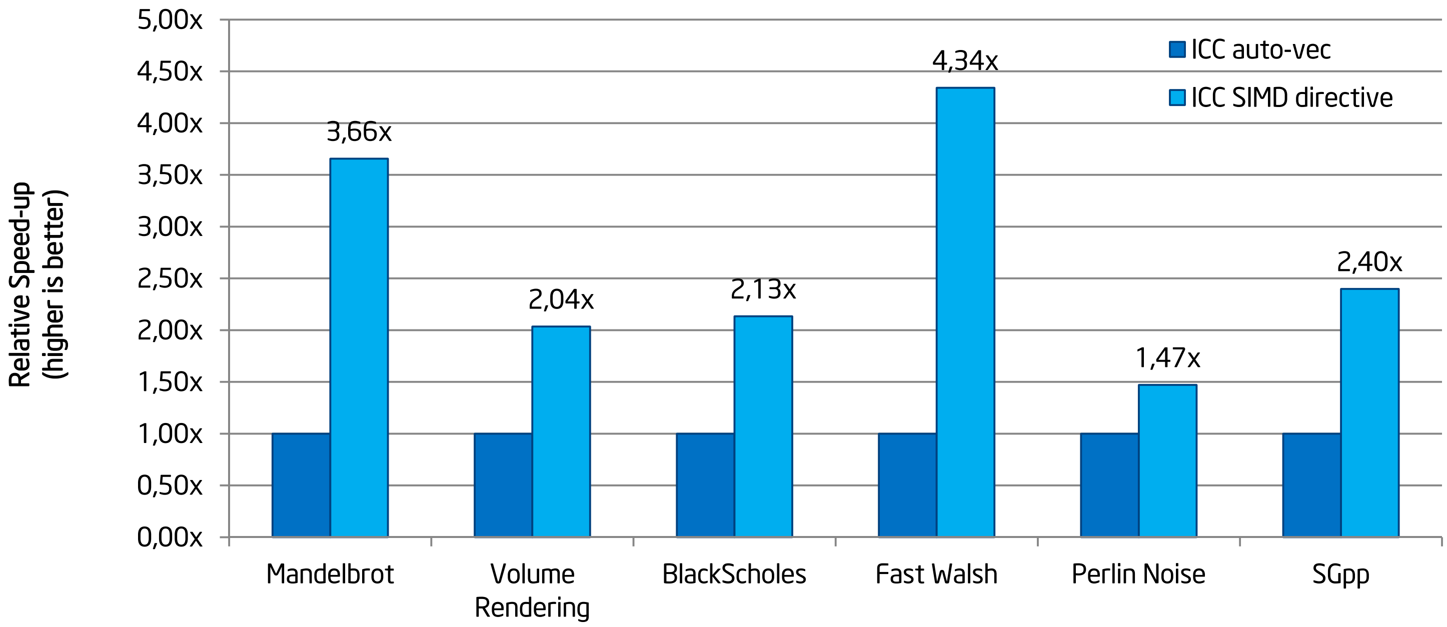
Explicit Vectorization – Motivating Example 2

```
#pragma omp declare simd simdlen(16)
uint32_t mandel(fcomplex c)
{
    uint32_t count = 1; fcomplex z = c;
    for (int32_t i = 0; i < max_iter; i += 1) {
        z = z * z + c;
        int t = cabsf(z) < 2.0f;
        count += t;
        if (!t) { break;}
    }
    return count;
}
```

- mandel() function is called from a loop over X/Y points.
- We would like to vectorize that outer loop.
- Compiler creates a vectorized function that acts on a vector of *N* values of *c*.

Optimization Notice

Explicit Vectorization – Performance Impact



M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell, "Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP", pages 59-72, Rome, Italy, June 2012. LNCS 7312.

Optimization Notice

Controlling FP Arithmetic with Intel® Composer XE

Standard Compiler Switches

GCC	ICC	Effect
-O0	-O0	Disable (almost all) optimization.
-O1	-O1	Optimize for speed (no code size increase for ICC)
-O2	-O2	Optimize for speed and enable vectorization (default for ICC)
-O3	-O3	Turn on high-level optimizations
-ftlo	-ipo	Enable interprocedural optimization
-ftree-vectorize	-vec	Enable auto-vectorization (auto-enabled with -O2 and -O3)
-fprofile-generate	-prof-gen	Generate runtime profile for optimization
-fprofile-use	-prof-use	Use runtime profile for optimization
	-parallel	Enable auto-parallelization
-fopenmp	-qopenmp	Enable OpenMP
-g	-g	Emit debugging symbols
	-qopt-report	Generate the optimization report
	-ansi-alias	Enable ANSI aliasing rules for C/C++
-mcorei7	-xSSE4.1	Generate code for Intel processors with SSE 4.1 instructions
-mcorei7-avx	-xCORE-AVX	Generate code for Intel processors with AVX1 instructions
-mcorei7-avx2	-xCORE-AVX2	Generate code for Intel processors with AVX2 instructions
-mnative	-xHOST	Generate code for the current machine used for compilation

Frequently Users want Consistent FP Results (which is not necessarily the “most accurate” result)

Root cause for variations in results

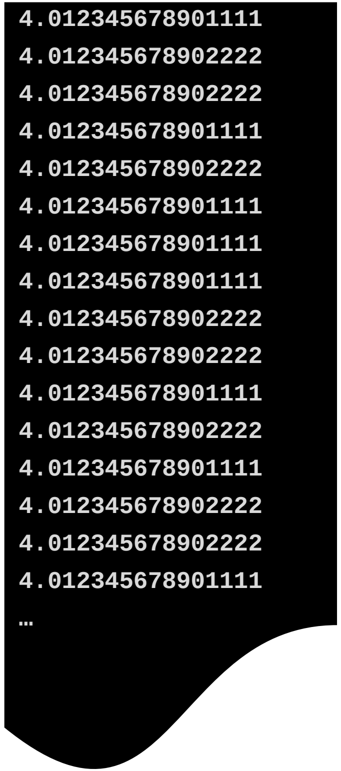
- floating-point numbers ➔ order of computation matters!
- Single precision arithmetic example (a+b)+c != a+(b+c)
  - 226 - 226 + 1 = 1 (infinitely precise result)
  - (226 - 226) + 1 = 1 (correct IEEE single precision result)
  - 226 - (226 - 1) = 0 (correct IEEE single precision result)

Conditions that affect the order of computations

- Different code branches (e.g., x87 versus SSE2 or AVX)
- Memory alignment (scalar or vector code)
- Dynamic parallel task/thread/rank scheduling

Bitwise repeatable/reproducible results

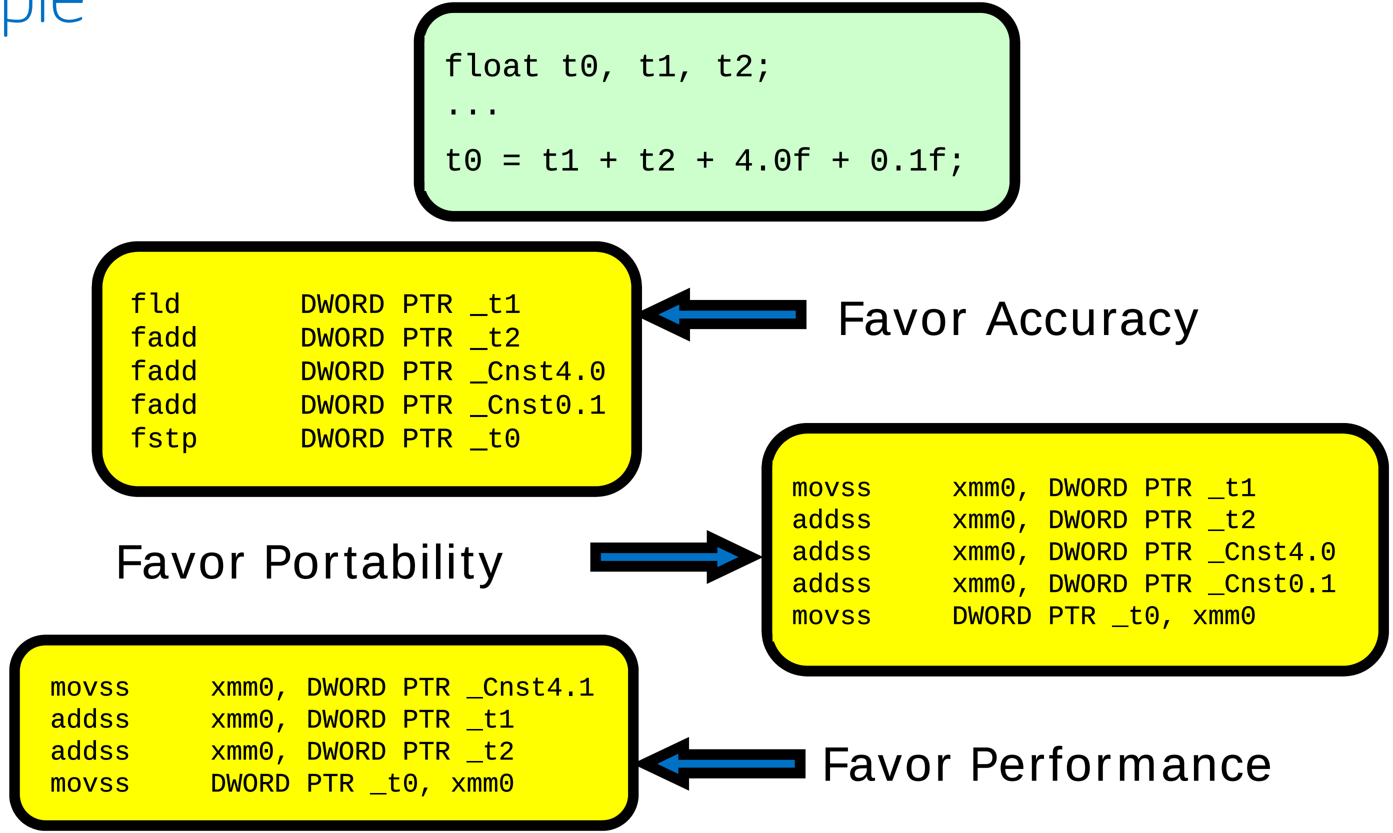
- repeatable = results the same as last run (same conditions)
- reproducible = results the same as results in other environments
- environments = OS / CPU / architecture / # threads / # processes / BIOS / pinning



```
4.012345678901111
4.012345678902222
4.012345678902222
4.012345678901111
4.012345678902222
4.012345678901111
4.012345678901111
4.012345678901111
4.012345678902222
4.012345678902222
4.012345678901111
4.012345678902222
4.012345678901111
4.012345678902222
4.012345678902222
4.012345678901111
…
```

Optimization Notice

Example



```
float t0, t1, t2;
...
t0 = t1 + t2 + 4.0f + 0.1f;
```

```
fld       DWORD PTR _t1
fadd      DWORD PTR _t2
fadd      DWORD PTR _Cnst4.0
fadd      DWORD PTR _Cnst0.1
fstp      DWORD PTR _t0
```

Favor Accuracy

Favor Portability

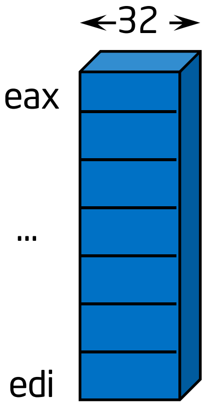
```
movss     xmm0, DWORD PTR _t1
addss     xmm0, DWORD PTR _t2
addss     xmm0, DWORD PTR _Cnst4.0
addss     xmm0, DWORD PTR _Cnst0.1
movss     DWORD PTR _t0, xmm0
```

```
movss     xmm0, DWORD PTR _Cnst4.1
addss     xmm0, DWORD PTR _t1
addss     xmm0, DWORD PTR _t2
movss     DWORD PTR _t0, xmm0
```

Favor Performance

Intel64 Register Set

IA32-INT Registers

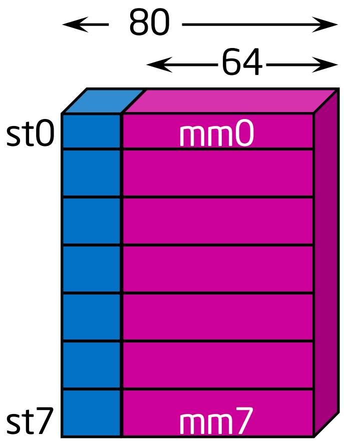


Fourteen 32-bit registers

Scalar data & addresses

Direct access to regs

MMX™ Technology / x87 Registers



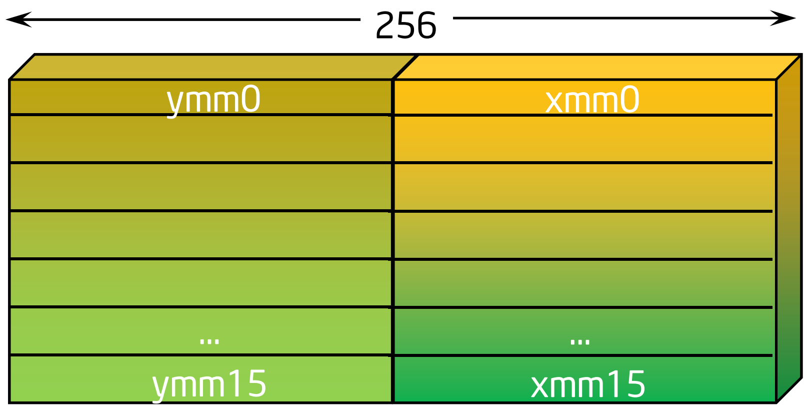
Eight 80/64-bit registers

Hold data only

Direct access to MM0..MM7

No MMX™ Technology / FP interoperability

AVX Registers SSE Registers



Sixteen 256-bit registers

Hold data only:

- 8 x single FP numbers
- 4 x double FP numbers
- overlap with 128-bit SSE registers

AVX-512 will extend ymm[0..15] to zmm[0..31] with 512-bit each.

FP Model Summary

Key	Value Safety	Expression Evaluation	FPU Environ. Access	Precise FP Exceptions
precise source double extended	Safe	Varies Source Double Extended	No	No
strict	Safe	Varies	Yes	Yes
fast=1 (default)	Unsafe	Unknown	No	No
fast=2	Very Unsafe	Unknown	No	No
except except-	*/** *	* *	* *	Yes No

*	These modes are unaffected. `–fp-model except[-]` only affects the precise FP exceptions mode.
**	It is illegal to specify `–fp-model except` in an unsafe value safety mode.

Optimization Notice

Value Safety

- In SAFE mode, the compiler may not make any transformations that could affect the result, e.g., all the following are prohibited.

x / x ⇔ 1.0	x could be 0.0, ∞, or NaN
x – y ⇔ - (y – x)	If x equals y, x – y is +0.0 while – (y – x) is -0.0
x – x ⇔ 0.0	x could be ∞ or NaN
x * 0.0 ⇔ 0.0	x could be -0.0, ∞, or NaN
x + 0.0 ⇔ x	x could be -0.0
(x + y) + z ⇔ x + (y + z)	General reassociation is not value safe
(x == x) ⇔ true	x could be NaN

- UNSAFE mode is the default
- VERY UNSAFE mode enables riskier transformations

Reassociation Example: Reductions

```
float Sum(const float A[], int n )
{
    float sum=0;
    for (int i=0; i<n; i++)
        sum = sum + A[i];
    return sum;
}
```

```
float Sum( const float A[], int n )
{
    int n4 = n-n%4; // or n4=n4&(~3)
    int i;
    float sum=0, sum1=0, sum2=0, sum3=0;
    for (i=0; i<n4; i+=4) {
        sum = sum + A[i];
        sum1 = sum1 + A[i+1];
        sum2 = sum2 + A[i+2];
        sum3 = sum3 + A[i+3];
    }
    sum = sum + sum1 + sum2 + sum3;
    for (; i<n; i++)
        sum = sum + A[i];
    return sum;
}
```

- Scalar reduction gives 7-8X perf gain for SSE – AVX even more !
- Invalid in SAFE modes
- Even in SAFE mode, OpenMP, MPI, TBB might do ‘unsafe’ reductions

Use of FMA Instructions [1]

Potential issue: Since execution of FMA does not round intermediate product result, final result may be different compared to older (non-FMA) CPUs

- For QA comparisons to older processors, FMAs in compiled code can be disabled explicitly by
  - -no-fma (/Qfma-)
  - -fp-model strict (disables much more besides)
- FMAs can be disabled at function level by

▪ #pragma fp_contract (off) (C/C++); !DIR$ NOFMA (Fortran)

Optimization Notice

Use of FMA Instructions [2]

- Putting multiply & add on separate lines does not disable FMA

```
  t = a*b;
  result = t + c;
// may still generate FMA
```

```
t = a*b;
 _mm_mfence();
 result = t + c;     // no FMA
```

- FMAs are not (completely) disabled by –fp-model precise
- None of the above disables FMA usage in math library
  - requires -fimf-arch-consistency=true
- Results may change on “Haswell” wrt “Sandy Bridge” even without recompilation!
  - math library may take different path at run-time
- For debugging interesting to know of: fma() and fmaf() intrinsics from math.h give FMA result with a single rounding via a libm call, **even for processors with no FMA instruction**

Optimization Notice

Sample of FMA Rounding Difference

```
double sub(double a, double b, double c, double d )
{
    c = -a;
    d =  b;
    return (a*b + c*d);
}
```

- Without FMA, should evaluate to zero
- With FMA, it may not evaluate to zero

  Returns   FMA(a, b, (c*d))   or   FMA (c, d, (a*b))

  Each has different rounding, unspecified which grouping the compiler will generate

  This behavior is not suppressed by ‘fp-model precise’ !

FP Model and FMA Summary

Key	Value Safety	Expression Evaluation	FPU Environ. Access	Precise FP Exceptions	FMA Use
precise source double extended	Safe	Varies Source Double Extended	No	No	Yes
strict	Safe	Varies	Yes	Yes	No
fast=1 (default)	Unsafe	Unknown	No	No	Yes
fast=2	Very Unsafe	Unknown	No	No	Yes
except except-	*/** *	* *	* *	Yes No	* *

* These modes are unaffected. `–fp-model except[-]` only affects the precise FP exceptions mode.

** It is illegal to specify `–fp-model except` in an unsafe value safety mode.

Optimization Notice

All Libraries Optimized for HSW

Math libraries detect target processor by their own – independent of code generated by compiler

- e.g., HSW-optimized version will be executed on HSW even in case binary is compiled for SandyBridge (-xavx)
- Can be disabled by switch -fimf-arch-consistency=true for libimf and libsvml and CBWR API (conditional bit wise reproducibility) for Intel® MKL-VML

- HSW optimization in some cases not necessarily implies use of FMA!

Library	Used for	Comment
libimf	Library routines for single elements – libm replacement	
libsvml	Small vector math library: Used by vectorizer to replace math calls in vectorized loops	Optimizations still on-going
MKL-VML	Vector math library component of MKL	

Optimization Notice

Sample Performance Data for SVML

Double Precision – Cycles per Element

Routine	Sandy Bridge			Haswell		
	HA	LA	EP	HA	LA	EP
sqrt	10.51	10.51	10.51	7.08	7.08	7.08
exp	11.20	8.48	8.06	7.12	5.13	4.62
sin	16.91	16.11	8.21	11.15	6.95	4.03

See here for complete data of MKL 11.2 comparing VML execution on Haswells (desktop processor), Westmere and SandyBridge EP https://software.intel.com/sites/products/documentation/doclib/mkl_sa/112/vml/functions/_performanceall.html

Code of VML similar to SVML but loop unrolling etc accelerate computation by working on multiple (vector-) computations simultaneously

Optimization Notice

Using Intel MPI for Performance

Use Best Possible Communication Fabric

Supported I_MPI_FABRICS	Description
shm	Shared-memory only; intra-node default
tcp	TCP/IP-capable network fabrics, such as Ethernet and InfiniBand* (through IPoIB*)
dapl	DAPL–capable network fabrics, such as InfiniBand*, iWarp*, and XPMEM* (through DAPL*)
ofa	OFA-capable network fabric including InfiniBand* (through OFED* verbs)
tmi	TMI-capable network fabrics including Qlogic*, Myrinet* (through Tag Matching Interface)

Intel MPI will select fastest available fabric by default (shared memory within a node and InfiniBand* across nodes – shm:dapl)

If using the OpenFabrics Enterprise Distribution (OFED*) software stack, select shm:ofa

Optimization Notice

Disable Fallback for Benchmarking (and Production)

Intel MPI Library falls back from the ‘dapl’ or ‘shm:dapl’ fabric to ‘tcp’ and/or ‘shm:tcp’ if DAPL provider initialization failed

Set I_MPI_FALLBACK to ‘disable’ to be sure that needed fast fabric is working

- Fallback is disabled by default if I_MPI_FABRICS is set

Same result is achieved with the command line option:

```
$ mpirun –genv I_MPI_FALLBACK 0 …
```

Use Connectionless Communication

The connectionless feature works for the ‘dapl’ and ‘tmi’ fabrics only

Provides better scalability

Significantly reduces memory requirements by reducing the number of receive queues

Generally advised for large jobs

```
$ export I_MPI_FABRICS=shm:dapl
$ export I_MPI_DAPL_UD=enable
```

Use lightweight statistics

- Set I_MPI_STATS to a non-zero integer value to gather MPI communication statistics (max value is 10)
- Manipulate the results with I_MPI_STATS_SCOPE to increase effectiveness of the analysis
- Example on the right – Gromacs rank 0 with suggested values
- Suggested values:

```
$ export I_MPI_STATS=3
$ export I_MPI_STATS_SCOPE=coll
```

```
Communication Activity by actual args
Collectives
Operation       Context  Algo  Comm size    Message size      Calls  Cost(%)
-------------------------------------------------------------------------------
--
Allreduce
1                    58     1          4              24          1    0.00
2                    58     1          4               4          8    0.00
3                    58     1          4               8         12    0.03
4                    58     1          4            1376        181    0.04
5                    58     1          4            1344         19    0.01
6                    58     1          4            1216          1    0.00
7                    58     1          4             224          1    0.00
8                     0     5        192               8          2    0.00
9                     0     5        192             968          1    0.00
10                    0     5        192             288          2    0.01
11                    0     5        192             768          2    0.00
Barrier
1                    62     5        160               0          1    0.00
2                     0     5        192               0          1    0.00
Bcast
...
Gather
1                    52     3          5              32         25    0.01
2                    54     3          4              36         25    0.00
3                    56     3          8              28         25    0.01
Reduce
1                    60     1         40              24          1    0.00
2                    60     1         40               4          8    0.00
3                    60     1         40               8         12    0.01
4                    60     1         40            1376        181    0.21
5                    60     1         40            1344         19    0.03
6                    60     1         40            1216          1    0.00
7                    60     1         40             224          1    0.00
Scatter
1                    62     1        160               8          1    0.00
Scatterv
1                    62     1        160          315840          2    0.03
2                    62     1        160           52640          1    0.08
===============================================================================
==
```

Optimization Notice

Choose the best collective algorithm

Use one of the I_MPI_ADJUST_<opname> knobs to change the algorithm

Recommendations:

- Focus on the most critical collective operations (see stats output)
- Run the Intel MPI Benchmarks by selecting various algorithms to find out the right protocol switchover points for hot collective operations
- … or use the mpitune tool

```
$ mpirun -genv I_MPI_ADJUST_REDUCE <algorithm #> …
```

Optimization Notice

Select Proper Process Layout

Default process layout is that all physical cores will be used

If running hybrid applications, you might want to reduce the number of ranks/node

Set I_MPI_PERHOST or use the –perhost (/-ppn) option to override the default process layout:

```
$ mpirun –ppn <#processes per node> -n <#processes> …
```

Same can be achieved using a “machinefile”

On batch scheduler environments, the Intel MPI library respects the scheduler settings

To overwrite the batch scheduler settings (at your own risk ☺):

```
$ export I_MPI_JOB_RESPECT_PROCESS_PLACEMENT=0
```

Use Proper Process Pinning

Default pinning options are suitable for most cases

Use I_MPI_PIN_PROCESSOR_LIST to define custom map of MPI processes to CPU cores pinning

The ‘cpuinfo’ utility of the Intel MPI Library shows the processor topology

Placing the processes on physical cores:

```
$ export I_MPI_PIN_PROCESSOR_LIST=allcores
```

Avoid sharing of common resources by adjacent MPI processes, use “map=scatter” setting:

```
$ export I_MPI_PIN_PROCESSOR_LIST=allcores,map=scatter
```

Choose to share resources by setting “map=bunch”:

```
$ export I_MPI_PIN_PROCESSOR_LIST=allcores,map=bunch
```

Optimization Notice

Use Proper Hybrid Process Pinning

Link with thread safe library (-qopenmp / -mt_mpi)

Choose MPI threading model (SINGLE / FUNNELED / SERIALIZED / MULTIPLE) – either using MPI_Init_thread(…) or env. var. I_MPI_THREAD_LEVEL_DEFAULT

```
$ export I_MPI_THREAD_LEVEL_DEFAULT=SINGLE
```

Choose distribution of MPI ranks & threads – (ranks x threads = cores)

```
$ mpirun –n <#ranks> -genv OMP_NUM_THREADS <#threads>
```

Pin MPI ranks using I_MPI_PIN_DOMAIN (e.g., „omp“ according to #OpenMP t.):

```
$ export I_MPI_PIN_DOMAIN=omp
```

Pin threads, e.g., KMP_AFFINITY

```
$ export KMP_AFFINITY=compact
```

If you want a nicer and more portable syntax, use OpenMP places introduced with OpenMP 4.0

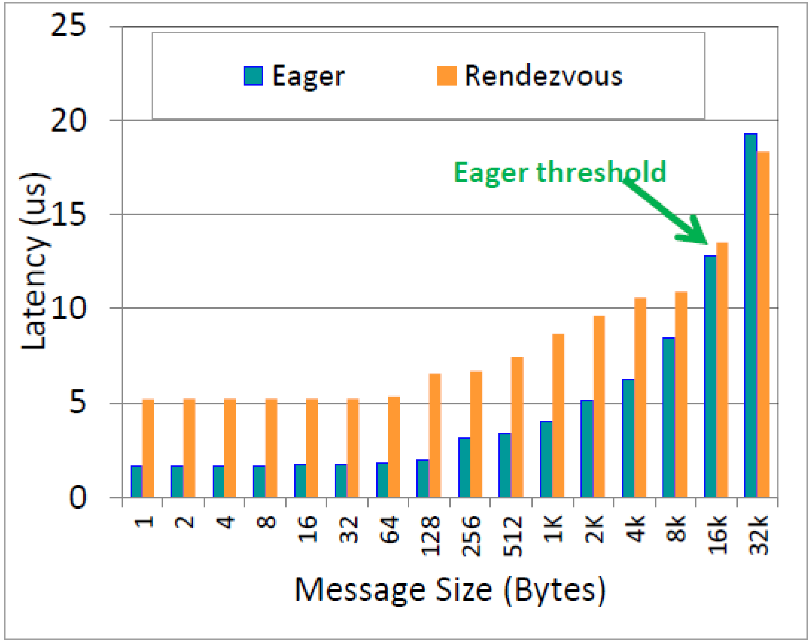
Adjust the eager / rendezvous protocol threshold

Two communication protocols:

“Eager” sends data immediately regardless of receive request availability and uses for short messages

“Rendezvous” notices receiving site on data pending and transfers when receive request is set

```
$ export I_MPI_EAGER_THRESHOLD=<#bytes>
```



Eager vs Rendezvous

Using the MPI Performance Snapshot Tool

1. Install Intel® Trace Analyzer and Collector
2. Setup your environment

```
$ source /opt/intel/itac/9.1/bin/mpi_perf_snapshot_vars.sh
```

3. Run with the MPI Performance Snapshot enabled

```
$ mpirun -mps -n 1024 ./exe
```

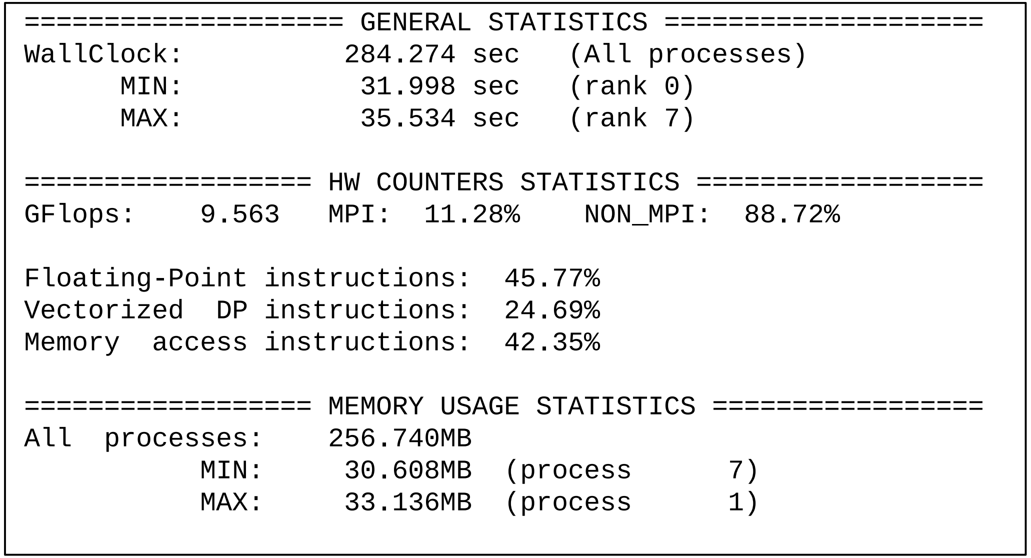
4. Analyze your results

```
$ mpi_perf_snapshot ./stats.txt ./app_stat.txt
```

Focus on Memory & Counter Usage

New collector displays summary info immediately after end of application run

HW counters & memory usage info:



```
==================== GENERAL STATISTICS ====================
WallClock:          284.274 sec   (All processes)
      MIN:           31.998 sec   (rank 0)
      MAX:           35.534 sec   (rank 7)

================== HW COUNTERS STATISTICS ==================
GFlops:    9.563   MPI:  11.28%    NON_MPI:  88.72%

Floating-Point instructions:  45.77%
Vectorized  DP instructions:  24.69%
Memory  access instructions:  42.35%

================== MEMORY USAGE STATISTICS =================
All  processes:    256.740MB
           MIN:     30.608MB  (process      7)
           MAX:     33.136MB  (process      1)
```

Find your MPI & OpenMP Imbalance hotspots

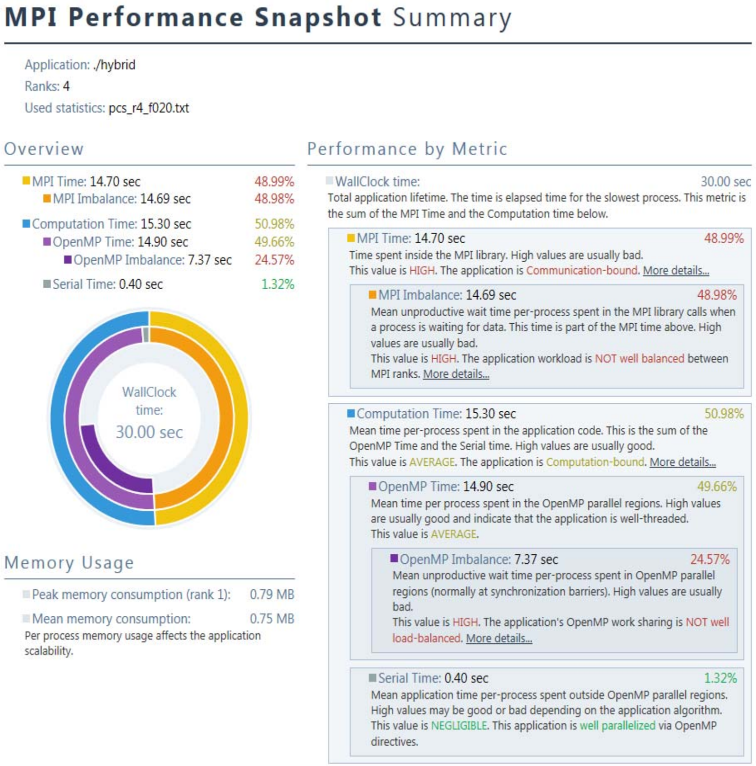
```
================= MPI IMBALANCE STATISTICS =================
MPI Imbalance:          207.847 sec            73,12% (All processes)
          MIN:           23.044 sec            64,85% (rank  6)
          MAX:           30.113 sec            88,57% (rank  1)

===================== OpenMP STATISTICS ====================
OpenMP Regions:        228.631 sec           80,43%           56 region(s) (All processes)
           MIN:         25.348 sec           71,33%            7 region(s) (rank 7)
           MAX:         33.124 sec           97,42%            7 region(s) (rank 1)

OpenMP Imbalance:        103.924 sec            36,56%  (All processes)
             MIN:         11.522 sec            32,43%  (rank 3)
             MAX:         15.057 sec            44,29%  (rank 2)
```

Optimization Notice

Easy-to-read HTML output helps you categorize performance issues



MPI Performance Snapshot Summary

Application: ./hybrid
Ranks: 4
Used statistics: pcs_r4_f020.txt

Overview

■ MPI Time: 14.70 sec	48.99%
■ MPI Imbalance: 14.69 sec	48.98%
■ Computation Time: 15.30 sec	50.98%
■ OpenMP Time: 14.90 sec	49.66%
■ OpenMP Imbalance: 7.37 sec	24.57%
■ Serial Time: 0.40 sec	1.32%

Memory Usage

■ Peak memory consumption (rank 1):	0.79 MB
■ Mean memory consumption:	0.75 MB

Per process memory usage affects the application scalability.

Performance by Metric

■ WallClock time: 30.00 sec
Total application lifetime. The time is elapsed time for the slowest process. This metric is the sum of the MPI Time and the Computation time below.

■ MPI Time: 14.70 sec 48.99%
Time spent inside the MPI library. High values are usually bad.
This value is HIGH. The application is Communication-bound. More details...

■ MPI Imbalance: 14.69 sec 48.98%
Mean unproductive wait time per-process spent in the MPI library calls when a process is waiting for data. This time is part of the MPI time above. High values are usually bad.
This value is HIGH. The application workload is NOT well balanced between MPI ranks. More details...

■ Computation Time: 15.30 sec 50.98%
Mean time per-process spent in the application code. This is the sum of the OpenMP Time and the Serial time. High values are usually good.
This value is AVERAGE. The application is Computation-bound. More details...

■ OpenMP Time: 14.90 sec 49.66%
Mean time per process spent in the OpenMP parallel regions. High values are usually good and indicate that the application is well-threaded.
This value is AVERAGE.

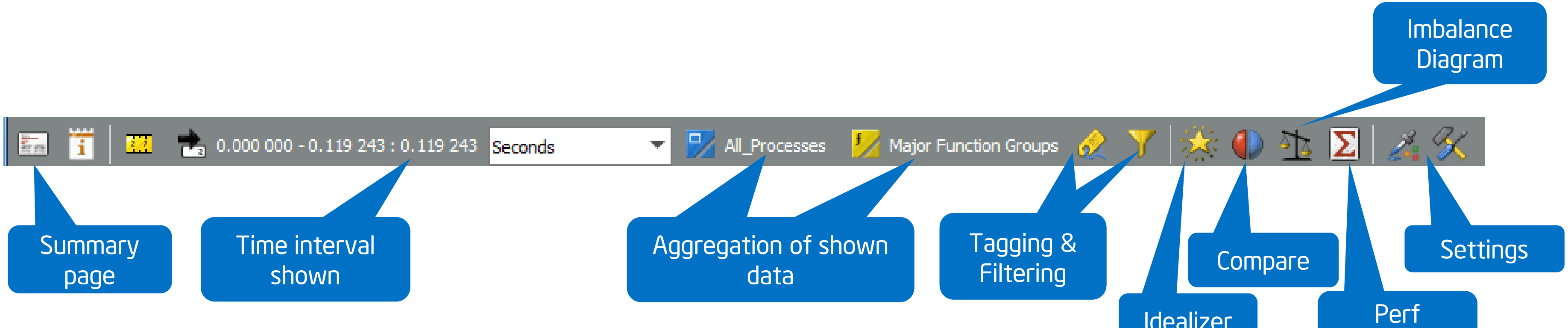
■ OpenMP Imbalance: 7.37 sec 24.57%
Mean unproductive wait time per-process spent in OpenMP parallel regions (normally at synchronization barriers). High values are usually bad.
This value is HIGH. The application's OpenMP work sharing is NOT well load-balanced. More details...

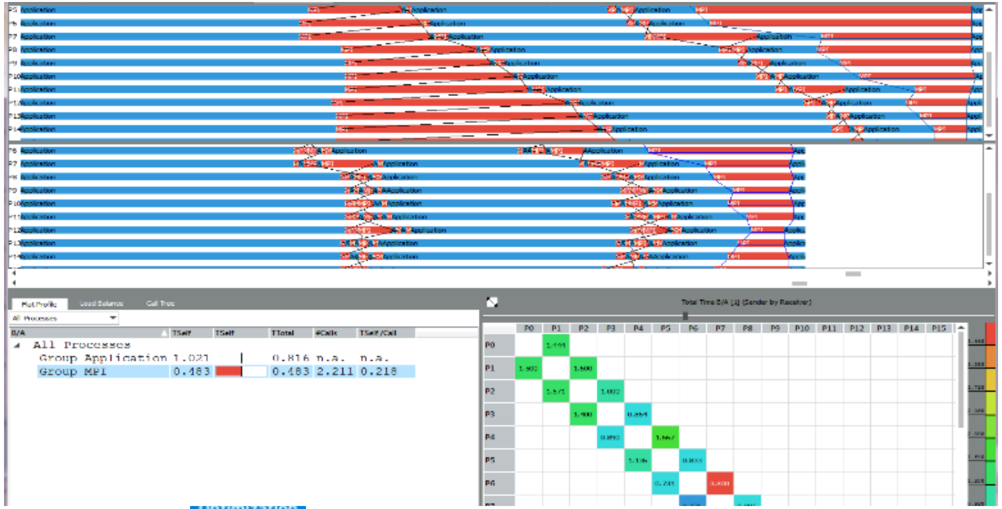
■ Serial Time: 0.40 sec 1.32%
Mean application time per-process spent outside OpenMP parallel regions. High values may be good or bad depending on the application algorithm.
This value is NEGLIGIBLE. This application is well parallelized via OpenMP directives.

Application Analysis to Guide Development Efforts

Optimization Notice

Full MPI Profiling via Intel® Trace Analyzer and Collector





```
Name                 TSelf   TSelf  TTotal  #Calls  TSelf/Call
All Processes
  Group Application  1.021          0.816   n.a.    n.a.
  Group MPI          0.483          0.483   2.211   0.218
```

Compare 2 communication profiles – focus on bottlenecks

Shows how MPI processes interact

Optimization Notice

The Last Slide…

The “Haswell” microarchitecture makes several performance improvements

SIMD-parallel programming is key to performance

Use implicit or explicit SIMD coding to exploit SIMD units

Tune MPI for optimal performance

Use MPI Performance Snapshot or ITAC to find MPI bottlenecks

Software