Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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

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

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
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)

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

Table of Contents:
Foreword by Bronis de Supinski (CTO LLNL)
Preface
Chapter 1: No Time to Read this Book?
Chapter 2: Overview of Platform Architectures
Chapter 3: Top-Down Software Optimization
Chapter 4: Addressing System Bottlenecks
Chapter 5: Addressing Application Bottlenecks:
  Distributed Memory
Chapter 6: Addressing Application Bottlenecks:
  Shared Memory
Chapter 7: Addressing Microarchitecture Bottlenecks
Chapter 8: Application Design Implications


Order now at http://www.clusterbook.info
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.
Driving Innovation and Integration
Enabled by Leading Edge Process Technologies

Integrated Today

Coming in the Future
The Magic of Integration
Moore’s Law at Work & Architecture Innovations

1970s
150 MFLOPS
CRAY-1

2015
1000000 MFLOPS
2S Intel® Xeon® Processor

66666x
Intel “Tick-Tock” Roadmap – Part I
## Intel “Tick-Tock” Roadmap – Part II

**Future Release Dates & Features subject to Change without Notice!**

<table>
<thead>
<tr>
<th>Haswell</th>
<th>4th Generation Intel® Core™ Micro Architecture</th>
<th>TBD</th>
<th>TBD</th>
<th>TBD</th>
<th>TBD</th>
<th>TBD</th>
<th>TBD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NEW Micro architecture 22nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>TICK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2013</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AVX-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW Process Technology</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Broadwell</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW Micro architecture 14nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>TOCK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>September 2014!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 new Inst.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Skylake</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW Micro architecture 14nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>TICK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2015/6 (?)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AVX-512 (?)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW Process Technology 10nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>TOCK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>???</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>???</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>???</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW Process Technology 10nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>TICK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>???</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>???</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW Process Technology 7nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>TOCK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>???</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>???</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Other brands and names are the property of their respective owners.*

Copyright © 2015, Intel Corporation. All rights reserved.
Recap: Sandy Bridge and Ivy Bridge Execution Units
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
Haswell Execution Unit Overview

Unified Reservation Station

- Port 0: Integer ALU & Shift, FMA FP Multiply
- Port 1: Integer ALU & LEA, FMA FP Mult FP Add
- Port 2: Load & Store Address
- Port 3: Store Data
- Port 4: Integer ALU & LEA
- Port 5: Integer ALU & Shift
- Port 6: Store Address

- 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

<table>
<thead>
<tr>
<th></th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Haswell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-order Window</td>
<td>128</td>
<td>168</td>
<td>192</td>
</tr>
<tr>
<td>In-flight Loads</td>
<td>48</td>
<td>64</td>
<td>72</td>
</tr>
<tr>
<td>In-flight Stores</td>
<td>32</td>
<td>36</td>
<td>42</td>
</tr>
<tr>
<td>Scheduler Entries</td>
<td>36</td>
<td>54</td>
<td>60</td>
</tr>
<tr>
<td>Integer Register File</td>
<td>N/A</td>
<td>160</td>
<td>168</td>
</tr>
<tr>
<td>FP Register File</td>
<td>N/A</td>
<td>144</td>
<td>168</td>
</tr>
<tr>
<td>Allocation Queue</td>
<td>28/thread</td>
<td>28/thread</td>
<td>56</td>
</tr>
</tbody>
</table>
## Core Cache Size/Latency/Bandwidth

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

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

* Total instructions / different mnemonics
FMA: Fused Multiply Add Instruction
Improves accuracy and performance for commonly used class of algorithms

***FMA: Polynomial Evaluation***

\[ a x^2 + b x + c = x(ax + b) + c \]

\[ \begin{align*}
    & a \times x \times +5 \\
    & x \times +3 \\
    & +5 \times +3 \\
\end{align*} \]

16 cycle latency
2 cycle throughput

\[ \begin{align*}
    & a \times x \times \text{FMA} \times +5 \\
    & x \times +3 \\
    & +5 \times +3 \\
\end{align*} \]

10 cycle latency
1 cycle throughput

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Instruction Set</th>
<th>SP FLOPs per cycle</th>
<th>DP FLOPs per cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>SSE (128-bits)</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>AVX (256-bits)</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Haswell</td>
<td>AVX2 (FMA) (256-bits)</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

**2x peak FLOPs/cycle (throughput)**

<table>
<thead>
<tr>
<th>Latency (clocks)</th>
<th>Xeon E5 v2</th>
<th>Xeon E5 v3</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>MulPS, PD</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>AddPS, PD</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Mul+Add /FMA</td>
<td>8</td>
<td>5</td>
<td>0.625</td>
</tr>
</tbody>
</table>

>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, ...
Intel® Xeon® Processors

Intel® Xeon® E3

Intel® Xeon® E5

Intel® Xeon® E7

Copyright © 2015, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
Intel® Xeon® E5-2600v3 Processor Overview

- **New 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

- **Existing Feature**
  - Intel® QuickPath Interface (x2) 9.6GT/s
  - 9.6GT/s
  - Intel® AVX 2.0 / Haswell New Instruction (HNI)
  - Intel® QuickPath Interface
  - ~2.5 MB Last Level Cache/Core
  - Up to 45 MB total LLC
  - Memory Technology: Socket R3
  - 4xDDR4 channels
  - 1333, 1600, 1866, 2133 MTS
  - Power Management
  - Per Core P-State (PCPS)
  - Uncore Frequency Scaling (UFS)
  - Energy Efficient Turbo (EET)

- System Architectures:
  - Core
  - LLC
  - System Agent
  - DMI
  - Integrated Voltage Regulator
  - PCI Express 3.0
# Key Differences Between E5-2600 v2 & E5-2600 v3

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

E5-2600 v2

- PCIe
- IVB
- IVB
- IVB
- IVB
- Memory Controller

Shared L3 Cache (30MB)

QPI

Shared L3 Cache

E5-2600 v3

- PCIe
- IVB
- IVB
- IVB
- IVB
- Memory Controller

Shared L3 Cache (45MB)

QPI

Shared L3 Cache

- HSW
- HSW
- HSW
- HSW
- HSW
- Memory Controller

-buffered switch

- HSW
- HSW
- HSW
- HSW
- HSW
- Memory Controller

Optimization Notice

Copyright © 2015, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
Haswell EP Die Configurations

<table>
<thead>
<tr>
<th>Chop</th>
<th>Columns</th>
<th>Home Agents</th>
<th>Cores</th>
<th>Power (W)</th>
<th>Transitors (B)</th>
<th>Die Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCC</td>
<td>4</td>
<td>2</td>
<td>14-18</td>
<td>110-145</td>
<td>5.69</td>
<td>662</td>
</tr>
<tr>
<td>MCC</td>
<td>3</td>
<td>2</td>
<td>6-12</td>
<td>65-160</td>
<td>3.84</td>
<td>492</td>
</tr>
<tr>
<td>LCC</td>
<td>1</td>
<td>4-8</td>
<td>55-140</td>
<td>2.60</td>
<td></td>
<td>354</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Area</th>
<th>Change</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>On-die interconnect</strong></td>
<td>• Two Fully Buffered Rings</td>
<td>• Enables higher core counts and provides higher bandwidth per core.</td>
</tr>
</tbody>
</table>
| **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

<table>
<thead>
<tr>
<th>Dimms / Ch</th>
<th>DDR3 1.5v</th>
<th>DDR3 1.35v</th>
<th>DDR4 RDIMM</th>
<th>DDR4 LRDIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1866</td>
<td>1600</td>
<td>2133</td>
<td>2133</td>
</tr>
<tr>
<td>2</td>
<td>1600</td>
<td>1333</td>
<td>1866</td>
<td>2133</td>
</tr>
<tr>
<td>3</td>
<td>1066</td>
<td>800</td>
<td>1600</td>
<td>1600</td>
</tr>
</tbody>
</table>
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
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.
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

Efficient vectorization, threading, and parallel execution drives higher performance for suitable scalable applications.
Parallel Programming for Intel® Architecture

<table>
<thead>
<tr>
<th>NODES</th>
<th>Use Intel® MPI, Co-Array Fortran</th>
</tr>
</thead>
</table>
| 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 Processors

Parallelization

- Intel® Math Kernel Library
- 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: Change to SP
  - No

Precision is important: impacts the SIMD width.

- 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
### Data Layout – Common Layouts

<table>
<thead>
<tr>
<th>Array-of-Structs (AoS)</th>
<th>Struct-of-Arrays (SoA)</th>
<th>Hybrid (AoSoA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x \ y \ z \ x \ y \ z]</td>
<td>[x \ x \ x \ x \ x \ x]</td>
<td>[x \ x \ y \ y \ z \ z]</td>
</tr>
<tr>
<td>[x \ y \ z \ x \ y \ z]</td>
<td>[y \ y \ y \ y \ y \ y]</td>
<td>[x \ x \ y \ y \ z \ z]</td>
</tr>
<tr>
<td>[x \ y \ z \ x \ y \ z]</td>
<td>[z \ z \ z \ z \ z \ z]</td>
<td>[x \ x \ y \ y \ z \ z]</td>
</tr>
</tbody>
</table>

- **Pros:**
  - Good locality of \{x, y, z\}.
  - 1 memory stream.

- **Cons:**
  - Potential for gather/scatter.

- **Pros:**
  - Contiguous load/store.

- **Cons:**
  - Poor locality of \{x, y, z\}.
  - 3 memory streams.

- **Pros:**
  - Contiguous load/store.
  - 1 memory stream.

- **Cons:**
  - Not a “normal” layout.
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

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
Data Alignment – Real-life Applications

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data
Data Alignment – Real-life Applications

Data Halo
Data Alignment – Real-life Applications
Data Alignment – Real-life Applications

Not strictly necessary...

Data
Halo
Padding
Data Alignment – Real-life Applications

Data Halo Padding

Not strictly necessary...
Implicit Vectorization

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

```c
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
Implicit Vectorization

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

```c
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.
Explicit Vectorization – Motivating Example 2

```c
#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.
Explicit Vectorization – Performance Impact

Controlling FP Arithmetic with Intel® Composer XE
# Standard Compiler Switches

<table>
<thead>
<tr>
<th>GCC</th>
<th>ICC</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>-O0</td>
<td>Disable (almost all) optimization.</td>
</tr>
<tr>
<td>-O1</td>
<td>-O1</td>
<td>Optimize for speed (no code size increase for ICC)</td>
</tr>
<tr>
<td>-O2</td>
<td>-O2</td>
<td>Optimize for speed and enable vectorization (default for ICC)</td>
</tr>
<tr>
<td>-O3</td>
<td>-O3</td>
<td>Turn on high-level optimizations</td>
</tr>
<tr>
<td>-ftlo</td>
<td>-ipo</td>
<td>Enable interprocedural optimization</td>
</tr>
<tr>
<td>-ftrve-vectorize</td>
<td>-vec</td>
<td>Enable auto-vectorization (auto-enabled with -O2 and -O3)</td>
</tr>
<tr>
<td>-fprofile-generate</td>
<td>-prof-gen</td>
<td>Generate runtime profile for optimization</td>
</tr>
<tr>
<td>-fprofile-use</td>
<td>-prof-use</td>
<td>Use runtime profile for optimization</td>
</tr>
<tr>
<td></td>
<td>-parallel</td>
<td>Enable auto-parallelization</td>
</tr>
<tr>
<td>-fopenmp</td>
<td>-qopenmp</td>
<td>Enable OpenMP</td>
</tr>
<tr>
<td>-g</td>
<td>-g</td>
<td>Emit debugging symbols</td>
</tr>
<tr>
<td></td>
<td>-qopt-report</td>
<td>Generate the optimization report</td>
</tr>
<tr>
<td></td>
<td>-ansi-alias</td>
<td>Enable ANSI aliasing rules for C/C++</td>
</tr>
<tr>
<td>-mcorei7</td>
<td>-xSSE4.1</td>
<td>Generate code for Intel processors with SSE 4.1 instructions</td>
</tr>
<tr>
<td>-mcorei7-avx</td>
<td>-xCORE-AVX</td>
<td>Generate code for Intel processors with AVX1 instructions</td>
</tr>
<tr>
<td>-mcorei7-avx2</td>
<td>-xCORE-AVX2</td>
<td>Generate code for Intel processors with AVX2 instructions</td>
</tr>
<tr>
<td>-mnative</td>
<td>-xHOST</td>
<td>Generate code for the current machine used for compilation</td>
</tr>
</tbody>
</table>
Frequently Users want Consistent FP Results (which is not necessarily the “most accurate” result)

Root cause for variations in results
- floating-point numbers \(\rightarrow\) order of computation matters!
- Single precision arithmetic example \((a+b)+c \neq a+(b+c)\)
  - \(226 - 226 + 1 = 1\) (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
- \(\text{repeatable} = \text{results the same as last run (same conditions)}\)
- \(\text{reproducible} = \text{results the same as results in other environments}\)
- \(\text{environments} = \text{OS / CPU / architecture / # threads / # processes / BIOS / pinning}\)
Example

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

- **Favor Accuracy**
  ```
  fld  DWORD PTR _t1
  fadd DWORD PTR _t2
  fadd DWORD PTR _Cnst4.0
  fadd DWORD PTR _Cnst0.1
  fstp DWORD PTR _t0
  ```

- **Favor Portability**
  ```
  movss xmm0, DWORD PTR _t1
  addss xmm0, DWORD PTR _t2
  addss xmm0, DWORD PTR _Cnst4.0
  movss DWORD PTR _t0, xmm0
  ```

- **Favor Performance**
  ```
  movss xmm0, DWORD PTR _Cnst4.1
  addss xmm0, DWORD PTR _t1
  addss xmm0, DWORD PTR _Cnst4.0
  movss DWORD PTR _t0, xmm0
  ```
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**
  - 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

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

* 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.
Value Safety

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

  - $x / x \leftrightarrow 1.0$  
    - $x$ could be 0.0, $\infty$, or NaN
  - $x - y \leftrightarrow -(y - x)$  
    - If $x$ equals $y$, $x - y$ is +0.0 while $-(y - x)$ is -0.0
  - $x - x \leftrightarrow 0.0$  
    - $x$ could be $\infty$ or NaN
  - $x * 0.0 \leftrightarrow 0.0$  
    - $x$ could be -0.0, $\infty$, or NaN
  - $x + 0.0 \leftrightarrow x$  
    - $x$ could be -0.0
  - $(x + y) + z \leftrightarrow x + (y + z)$  
    - General reassociation is not value safe
  - $(x == x) \leftrightarrow true$  
    - $x$ could be NaN

- UNSAFE mode is the default
- VERY UNSAFE mode enables riskier transformations
Reassociation Example: Reductions

- 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

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

<table>
<thead>
<tr>
<th>Key</th>
<th>Value Safety</th>
<th>Expression Evaluation</th>
<th>FPU Environ. Access</th>
<th>Precise FP Exceptions</th>
<th>FMA Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>precise source double extended</td>
<td>Safe</td>
<td>Varies</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>strict</td>
<td>Safe</td>
<td>Varies</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>fast=1 (default)</td>
<td>Unsafe</td>
<td>Unknown</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>fast=2</td>
<td>Very Unsafe</td>
<td>Unknown</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>except</td>
<td><em>/</em>* *</td>
<td>*</td>
<td>*</td>
<td>Yes</td>
<td>*</td>
</tr>
<tr>
<td>except-</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>No</td>
<td>*</td>
</tr>
</tbody>
</table>

* These modes are unaffected. 
** It is illegal to specify `–fp-model except[-]` in an unsafe value safety mode.
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!

<table>
<thead>
<tr>
<th>Library</th>
<th>Used for</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>libimf</td>
<td>Library routines for single elements - libm replacement</td>
<td></td>
</tr>
<tr>
<td>libsvml</td>
<td>Small vector math library: Used by vectorizer to replace math calls in vectorized loops</td>
<td>Optimizations still on-going</td>
</tr>
<tr>
<td>MKL-VML</td>
<td>Vector math library component of MKL</td>
<td></td>
</tr>
</tbody>
</table>
Sample Performance Data for SVML
Double Precision – Cycles per Element

<table>
<thead>
<tr>
<th>Routine</th>
<th>Sandy Bridge</th>
<th>Haswell</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HA</td>
<td>LA</td>
</tr>
<tr>
<td>sqrt</td>
<td>10.51</td>
<td>10.51</td>
</tr>
<tr>
<td>exp</td>
<td>11.20</td>
<td>8.48</td>
</tr>
<tr>
<td>sin</td>
<td>16.91</td>
<td>16.11</td>
</tr>
</tbody>
</table>

See here for complete data of MKL 11.2 comparing VML execution on Haswells (desktop processor), Westmere and SandyBridge EP
cell.all.html

Code of VML similar to SVML but loop unrolling etc accelerate computation by working on multiple (vector-) computations simultaneously
Using Intel MPI for Performance
Use Best Possible Communication Fabric

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

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
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:

```bash
$ 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:

```bash
$ export I_MPI_STATS=3
$ export I_MPI_STATS_SCOPE=coll
```
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 #> ...
```
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
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>
Using the MPI Performance Snapshot Tool

1. Install Intel® Trace Analyzer and Collector
2. Setup your environment
   
   ```bash
   $ source /opt/intel/itac/9.1/bin/mpi_perf_snapshot_vars.sh
   ```
3. Run with the MPI Performance Snapshot enabled
   
   ```bash
   $ mpirun -mps -n 1024 ./exe
   ```
4. Analyze your results
   
   ```bash
   $ 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

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
<th>Percent</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI Imbalance</td>
<td>207.847</td>
<td>73.12%</td>
<td>All processes</td>
</tr>
<tr>
<td>MIN</td>
<td>23.044</td>
<td>64.85%</td>
<td>6</td>
</tr>
<tr>
<td>MAX</td>
<td>30.113</td>
<td>88.57%</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
<th>Percent</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP Regions</td>
<td>228.631</td>
<td>80.43%</td>
<td>56 region(s) (All processes)</td>
</tr>
<tr>
<td>MIN</td>
<td>25.348</td>
<td>71.33%</td>
<td>7 region(s) (rank 7)</td>
</tr>
<tr>
<td>MAX</td>
<td>33.124</td>
<td>97.42%</td>
<td>7 region(s) (rank 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
<th>Percent</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP Imbalance</td>
<td>103.924</td>
<td>36.56%</td>
<td>All processes</td>
</tr>
<tr>
<td>MIN</td>
<td>11.522</td>
<td>32.43%</td>
<td>3</td>
</tr>
<tr>
<td>MAX</td>
<td>15.057</td>
<td>44.29%</td>
<td>2</td>
</tr>
</tbody>
</table>
Easy-to-read HTML output helps you categorize performance issues
Full MPI Profiling via Intel® Trace Analyzer and Collector

- Summary page
- Time interval shown
- Aggregation of shown data
- Tagging & Filtering
- Compare
- Idealizer
- Perf Assistant
- Settings

Compare 2 communication profiles - focus on bottlenecks

Shows how MPI processes interact
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