Legal Disclaimer & Optimization Notice

Performance results are based on testing as of August 2017 to September 2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.

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

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.

Copyright © 2019, Intel Corporation. All rights reserved. Intel, Xeon, Core, VTune, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

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

Notice revision #20110804
ASPECTS OF HPC/THROUGHPUT APPLICATION PERFORMANCE

Intel Hardware Features

- **Distributed memory**
  - Message size
  - Rank placement
  - Rank Imbalance
  - RTL Overhead
  - Network Bandwidth

- **Memory**
  - False Sharing
  - Access with strides
  - Latency
  - Bandwidth
  - NUMA

- **I/O**
  - File I/O
  - I/O latency
  - I/O waits
  - System-wide I/O

- **Threading**
  - Threaded/serial ratio
  - Thread Imbalance
  - RTL overhead (scheduling, forking)
  - Synchronization

- **CPU Core**
  - uArch issues (IPC)
  - Vectorization
  - FPU usage efficiency

*Other names and brands may be claimed as the property of others.*
What are the Aspects of Performance

- False Sharing
- Access with strides
- Latency
- Bandwidth
- NUMA

- File I/O
- I/O latency
- I/O waits
- System-wide I/O

Threaded/serial ratio
Thread Imbalance
RTL overhead (scheduling, forking)
Synchronization

Optimization Notice
Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
INTEL® VTUNE™ AMPLIFIER’S APPLICATION PERFORMANCE SNAPSHOT
APS workflow

Setup Environment
- `source <VTune_install_dir>/amplxe-vars.sh` – integrated to VTune
- `source <APS_Install_dir>/apsvars.sh` – standalone APS package

Run Application
- `>aps <application and args>`
- MPI: `mpirun <mpi options> aps <application and args>`

Generate summary report for result_folder
- `>aps –report <result_folder>`

Generate CL reports with detailed MPI statistics for a result_folder
- `aps-report –<option> <result folder>`
**APS HTML Report**

**Application Performance Snapshot**

| Report examples (press the links to play): **MPI Bound, Memory Bound, OpenMP imbalance**

### MPI Time

<table>
<thead>
<tr>
<th>Label</th>
<th>Time (Max 2.0%, Min 0.0%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>2.05s</td>
</tr>
<tr>
<td>Total</td>
<td>44.40s</td>
</tr>
</tbody>
</table>

### OpenMP Imbalance

<table>
<thead>
<tr>
<th>Label</th>
<th>Time (Max 2.0%, Min 0.0%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0.00s</td>
</tr>
<tr>
<td>Total</td>
<td>35.09s</td>
</tr>
</tbody>
</table>

### Memory Footprint

<table>
<thead>
<tr>
<th>Label</th>
<th>Time (Max 2.0%, Min 0.0%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0.00%</td>
</tr>
<tr>
<td>Total</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

---

*Other names and brands may be claimed as the property of others.*
APS HTML Report Breakdown – Parallel Runtimes

- **MPI Time**
  - How much time was spent in MPI calls
  - Averaged by ranks with % of Elapsed time
  - Available for MPICH-based MPI and OpenMPI

- **MPI Imbalance**
  - Unproductive time spent in MPI library waiting for data
    - Switched off by default
    - Available for Intel MPI with APS_IMBALANCE_TYPE=1
    - Over supported MPISs with APS_IMBALANCE_TYPE=2

- **OpenMP Imbalance**
  - Time spent at OpenMP Synchronization Barriers normalized by number of threads
  - Available for Intel OpenMP

- **Serial time**
  - Time spend outside OpenMP regions
  - Available for Intel OpenMP, shared memory applications only

---

**MPI Time**
- 1.33s
- 10.75% of Elapsed Time

**MPI Imbalance**
- 1.23s
- 9.19% of Elapsed Time

**TOP 5 MPI Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waitall</td>
<td>10.24</td>
</tr>
<tr>
<td>Irecv</td>
<td>0.18</td>
</tr>
<tr>
<td>Isend</td>
<td>0.06</td>
</tr>
<tr>
<td>Barrier</td>
<td>0.03</td>
</tr>
<tr>
<td>Reduce</td>
<td>0.02</td>
</tr>
</tbody>
</table>

**OpenMP Imbalance**
- 3.44s
- 42.25% of Elapsed Time

**Serial Time**
- 4.45s
- 31.11% of Elapsed Time
APS HTML Report Breakdown – Intel omni-Path interconnect

- Average Bandwidth and Packet Rate
  - Available only with Intel driver enabled on a system

<table>
<thead>
<tr>
<th>Intel Omni-Path Fabric Usage</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interconnect Bandwidth</td>
<td>AVG, GB/sec</td>
<td></td>
</tr>
<tr>
<td>Outgoing</td>
<td>0.87</td>
<td></td>
</tr>
<tr>
<td>Incoming</td>
<td>0.87</td>
<td></td>
</tr>
<tr>
<td>Interconnect Packet Rate</td>
<td>AVG, Million Packets/sec</td>
<td></td>
</tr>
<tr>
<td>Outgoing</td>
<td>3.58</td>
<td></td>
</tr>
<tr>
<td>Incoming</td>
<td>3.58</td>
<td></td>
</tr>
</tbody>
</table>
APS HTML Report Breakdown – Memory Access

- Memory stalls measurement with breakdown by cache and DRAM
- Average DRAM Bandwidth*
- NUMA ratio

*Average DRAM bandwidth collection is available with Intel driver or Linux Perf system wide monitoring enabled on a system
APS HTML Report Breakdown – vectorization

- Vectorization efficiency based on HW-event statistics with
  - Breakdown by vector/scalar instructions
  - Floating point vs memory instruction ratio

**Vectorization**

41.40% of Packed FP Operations

Instruction Mix:

<table>
<thead>
<tr>
<th>Type</th>
<th>% of uOps</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP FLOPs</td>
<td>0.00%</td>
</tr>
<tr>
<td>DP FLOPs</td>
<td>17.40%</td>
</tr>
</tbody>
</table>

- Packed: 41.40% from DP FP
  - 128-bit: 41.40%
  - 256-bit: 0.00%
- Scalar: 58.60% from DP FP

Non-FP:

82.60% of uOps

FP Arith/Mem Rd Instr. Ratio: 0.50x

FP Arith/Mem Wr Instr. Ratio: 4.14x
APS Command Line Reports – Detailed MPI statistics

aps-report [keys] [options] <result>

[keys] – what to show
--functions
--mpi_time_per_rank
--collop_time_per_rank
--message_sizes
--transfers_per_communication
--transfers_per_rank
--node_to_node
--transfers_per_function
--communicators_list

[options] – how to show
--rank
--comm_id
--details
--communicators
--volume_threshold
--time_threshold
--number_of_lines
--no_filters
--communicators_list
--format

See descriptions with >aps-report command
Please note: some reports are available with non-default MPS_STAT_LEVEL = 1
APS Command Line Reports – Detailed MPI statistics

Data Transfers for Rank-to-Rank Communication – UI representation

>aps-report --transfers_per_communication --format=html <result>

use “-v” to generate the chart by volume

Requires setting MPS_STAT_LEVEL=4 (5) before collection
“Automatic” Vectorization Often Not Enough
A good compiler can still benefit greatly from vectorization optimization

Compiler will not always vectorize
- Check for Loop Carried Dependencies using Intel® Advisor
- All clear? Force vectorization. C++ use: pragma simd, Fortran use: SIMD directive

Not all vectorization is efficient vectorization
- Stride of 1 is more cache efficient than stride of 2 and greater. Analyze with Intel® Advisor.
- Consider data layout changes Intel® SIMD Data Layout Templates can help

Arrays of structures are great for intuitively organizing data, but are much less efficient than structures of arrays. Use the Intel® SIMD Data Layout Templates (Intel® SDLT) to map data into a more efficient layout for vectorization.

Benchmarks on prior slides did not all “auto vectorize.” Compiler directives were used to force vectorization and get more performance.
Intel® Advisor – Vectorization Advisor
Get breakthrough vectorization performance

Faster Vectorization Optimization:
▪ Vectorize where it will pay off most
▪ Quickly ID what is blocking vectorization
▪ Tips for effective vectorization
▪ Safely force compiler vectorization
▪ Optimize memory stride

The data and guidance you need:
▪ Compiler diagnostics + Performance Data + SIMD efficiency
▪ Detect problems & recommend fixes
▪ Loop-Carried Dependency Analysis
▪ Memory Access Patterns Analysis

Optimize for AVX-512 with or without access to AVX-512 hardware

Part of Intel® Parallel Studio XE

Optimization Notice
Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

http://intel.ly/advisor-xe
Find Effective Optimization Strategies
Intel® Advisor: Cache-aware Roofline Analysis

Roofline Performance Insights
- Highlights poor performing loops
- Shows performance “headroom” for each loop
  - Which can be improved
  - Which are worth improving
- Shows likely causes of bottlenecks
- Suggests next optimization steps
What’s New for 2019?

Intel® Advisor - Optimize Vectorization, Prototype Threading & Analyze Flow Graphs

Improved Roofline Analysis
- Hierarchical roofline find small changes that add up to a large gain
- Optimize integer and floating point calculations

More Ways to Share and Analyze Results
- Export a dynamic HTML roofline from the command line or UI
- Use macOS* to Analyze Windows* and Linux* results

Faster and Better Analysis
- Cache simulation – Get an accurate memory footprint for multiple configurations
- Selective profiling – Reduce overhead for faster analysis results
- Memory footprint, filtering and compare operations simplify analysis

Flow Graph Analyzer
- Rapid interactive prototyping and analysis
- Build, validate, and visualize algorithms
Integrated Roofline model

In the Intel® Advisor Integrated Roofline chart the Arithmetic Intensity and memory traffic for each level of the memory hierarchy is represented separately.

You can visualize the levels that need further optimization.
Demo: Nbody kernel implementation

GSimulation.cpp:
...

for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        real_type distance, dx, dy, dz;
        real_type distanceSqr = 0.0;
        real_type distanceInv = 0.0;

        dx = particles[j].pos[0] - particles[i].pos[0];
        dy = particles[j].pos[1] - particles[i].pos[1];

        distSqr = dx*dx + dy*dy + dz*dz + softeningSquared;
        distInv = 1.0 / sqrt(distanceSqr);
        particles[i].acc[0] += dx * G * particles[j].mass * distInv * distInv * distInv;
        particles[i].acc[1] += ...
        particles[i].acc[2] += ...
    }
}
...
// update acceleration

Optimization Notice
Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Demo – nbody/ver-avx512: Advisor Summary
Demo – nbody/ver-avx512: Code Analytics
Demo – vectorization_tutorial/ver-avx512: Recommendations
Demo – vectorization_tutorial/ver-avx512

- Go to the folder vectorization_tutorial/ver-avx512
- Type `make clean-results` to delete the previous data.
- Generate a new Survey Analysis of Advisor and Roofline:
  - `make roofline`
- To run the MAP Analysis: `make map`
Demo – vectorization_tutorial/ver-avx512: Map Analysis
Memory access pattern

How should I access data?

- **Unit stride access are faster**
  
  ```
  for (i=0; i<N; i++)
  A[i] = B[i]*d
  ```

- **Constant stride are more complex**
  
  ```
  for (i=0; i<N; i+=2)
  A[i] = B[i]*d
  ```

- **Non predictable access are usually bad**
  
  ```
  for (i=0; i<N; i++)
  A[i] = B[C[i]]*d
  ```

For B, 1 cache line load computes 4 DP

For B, 2 cache line loads compute 4 DP with reconstructions

For B, 4 cache line loads compute 4 DP with reconstructions, prefetching might not work
Non-unit stride load: AoS vs SoA

The compiler might generate gather/scatter instructions for loops automatically vectorized where memory locations are not contiguous.

```cpp
struct Particle
{
  public:
  ...
  real_type pos[3];
  real_type vel[3];
  real_type acc[3];
  real_type mass;
};

struct ParticleSoA
{
  public:
  ...
  real_type *pos_x,*pos_y,*pos_z;
  real_type *vel_x,*vel_y,*vel_z;
  real_type *acc_x,*acc_y,*acc_z
  real_type *mass;
};
```
Demo – vectorization_tutorial/ver-soa: Report
Demo – nbody/ver-soa: Code Analytics
Demo – Roofline Comparison

- **no-fp-converts**
- **avx-512**
- **soa**
- **baseline**
Compiler optimization report: last result

LOOP BEGIN at GSimulation.cpp(145,6)

remark #15389: vectorization support: reference this->particles->pos_x[j] has unaligned access [ GSimulation.cpp(151,8) ]

remark #15389: vectorization support: reference this->particles->mass[j] has unaligned access [ GSimulation.cpp(160,20) ]

remark #15381: vectorization support: unaligned access used inside loop body
remark #15305: vectorization support: vector length 8
remark #15309: vectorization support: normalized vectorization overhead 1.055

remark #15450: unmasked unaligned unit stride loads: 4
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 118
remark #15477: vector cost: 13.620
remark #15478: estimated potential speedup: 7.910
remark #15488: --- end vector cost summary ---

LOOP END
Memory alignment

The compiler cannot know if your data is aligned to a multiple of the vector register width. This could effect the performance.

A pointer $p$ is aligned to a memory location on $n$-byte boundary if: $((\text{size}_t)p \% n == 0)$

For AVX, alignment to 32bytes boundary (4DP words) allows a single reference to a cache line for moving 4 DP words into the registers.

![Diagram showing memory alignment]

- **Single Cache access for 4 DP words**:
  - cache line 0: Load 4DP words
  - cache line 1: Load 4DP words
  - cache line 2: Load 4DP words

- **Across Cache line access for 4 DP words**:
  - cache line 0: 2 Loads 4DP words
  - cache line 1: 2 Loads 4DP words
  - cache line 2: 2 Loads 4DP words

32byte Aligned | Non-Aligned
Memory alignment

On the Stack: for declared variables the Intel® C/C++ compiler aligned the data naturally:

```c
float f; //4-byte aligned    double d; //8-byte aligned
```

For array data an attribute is necessary:

```c
float array[N] __attribute__((aligned(32))); //32byte aligned
```

On the Heap: the array can be allocated/deallocate with special functions:

```c
#include <malloc.h>
...
float *array = (float*) _mm_malloc(N*sizeof(float), 32);
...
_mm_free(array);
```
Peel and reminder loop

The compiler can generate a Peel and Reminder loop in case where:

- The loop trip count is known only during runtime
- The alignment is not known during compilation

Then the compiler generates a check in code at the beginning of the loop to verify its assumptions. This could cause inefficiency, since every time enters the loops, it does these checks.

```c
for(j = 0; j < N; j++) array[j] = ...
```
Analyze & Tune Application Performance
Intel® VTune™ Amplifier—Performance Profiler

Save Time Optimizing Code
- Accurately profile C, C++, Fortran*, Python*, Go*, Java*, or any mix
- Optimize CPU, threading, memory, cache, storage & more
- Save time: rich analysis leads to insight
- Take advantage of Priority Support
  - Connects customers to Intel engineers for confidential inquiries (paid versions)

What’s New in 2019 Release (partial list)
- New Platform Profiler! - Longer Data Collection
- A more accessible user interface provides a simplified profiling workflow
- Smarter, faster Application Performance Snapshot: Analyze CPU utilization of physical cores, pause/resume, more... (Linux*)
- Improved JIT profiling for server-side/cloud applications

Learn More: software.intel.com/intel-vtune-amplifier-xe
# Two Great Ways to Collect Data

**Intel® VTune™ Amplifier**

<table>
<thead>
<tr>
<th>Software Collector</th>
<th>Hardware Collector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses OS interrupts</td>
<td>Uses the on chip Performance Monitoring Unit (PMU)</td>
</tr>
<tr>
<td>Collects from a single process tree</td>
<td>Collect system wide or from a single process tree.</td>
</tr>
<tr>
<td>~5ms default resolution</td>
<td>~1ms default resolution (finer granularity - finds small functions)</td>
</tr>
<tr>
<td>Either an Intel® or a compatible processor</td>
<td>Requires a genuine Intel® processor for collection</td>
</tr>
<tr>
<td>Call stacks show calling sequence</td>
<td>Optionally collect call stacks</td>
</tr>
<tr>
<td>Works in virtual environments</td>
<td>Works in a VM only when supported by the VM (e.g., vSphere*, KVM)</td>
</tr>
<tr>
<td>No driver required</td>
<td>Uses Intel driver or perf if driver not installed</td>
</tr>
</tbody>
</table>

No special recompiles - C, C++, C#, Fortran, Java, Python, Assembly
The Stream Benchmark
John D. McCalpin (TACC)

```
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
c[j] = a[j];

#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
b[j] = scalar*c[j];

#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
c[j] = a[j]+b[j];

#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
a[j] = b[j]+scalar*c[j];
```

![Diagram showing Copy, Scale, Add, and Triad]

Arithmetic intensity (>9 for peak DP DRAM)

- Copy: 0 Flop / 2 * 8 Bytes = 0
- Scale: 1 Flop / 3 * 8 Bytes = 0.042
- Add: 1 Flop / 3 * 8 Bytes = 0.042
- Triad: 2 Flop / 4 * 8 Bytes = 0.0625

Note that Stream is reporting the best Bandwidth rate out of 10 iterations per default.
Performance Note

Xeon Scalable 2\textsuperscript{nd} Generation 8260

Name: Intel(R) Xeon(R) Processor code named Cascade Lake

Frequency: 2.4 GHz

Logical CPU Count: 96

Max DRAM Single-Package Bandwidth: 140 GB/s

(DDR4-2933 * 6 Channels * 8 Bytes)

Performance figures are reported OOB without further optimizations like hugepages
We modified Stream

```plaintext
$ diff ./stream.c ./stream_mod.c | wc -l
4
$ icc -qopenmp -DSTREAM_ARRAY_SIZE=1000000000 -mcmodel large ../stream.c -O2 -g -xHost -o stream.x
$ icc -qopenmp -DSTREAM_ARRAY_SIZE=1000000000 -mcmodel large ../stream_mod.c -O2 -g -xHost -o stream_mod.x
$ export OMP_PLACES=threads
```
### STREAM Baseline vs Modified

**stream.x**

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>179386.1</td>
<td>0.089317</td>
<td>0.089193</td>
<td>0.089479</td>
</tr>
<tr>
<td>Scale:</td>
<td>187742.8</td>
<td>0.085394</td>
<td>0.085223</td>
<td>0.085636</td>
</tr>
<tr>
<td>Add:</td>
<td>201301.6</td>
<td>0.119386</td>
<td>0.119224</td>
<td>0.119830</td>
</tr>
<tr>
<td>Triad:</td>
<td>200889.1</td>
<td>0.119528</td>
<td>0.119469</td>
<td>0.119599</td>
</tr>
</tbody>
</table>

Solution Validates: avg error less than 1.000000e-13 on all three arrays

---

**stream_mod.x**

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>95325.4</td>
<td>0.205046</td>
<td>0.167846</td>
<td>0.230016</td>
</tr>
<tr>
<td>Scale:</td>
<td>84390.4</td>
<td>0.214661</td>
<td>0.189595</td>
<td>0.275163</td>
</tr>
<tr>
<td>Add:</td>
<td>107456.1</td>
<td>0.263129</td>
<td>0.223347</td>
<td>0.317161</td>
</tr>
<tr>
<td>Triad:</td>
<td>100594.8</td>
<td>0.289895</td>
<td>0.238581</td>
<td>0.347999</td>
</tr>
</tbody>
</table>

Solution validates: avg error less than 1.000000e-13 on all three arrays
Problem Investigation

What causes the memory bandwidth drop?
Collecting Hotspots

amplxe-cl -c hotspots -r r_hs_mod -- ./stream_mod.x
**Optimization Notice**

Intel Internal Only

---

**Elapsed Time**: 22.803s

- **CPU Time**: 1116.020s
- **Effective Time**: 979.855s
- **Spin Time**: 136.065s
- **Imbalance or Serial Spinning**: 123.754s
- **Lock Contention**: 0s
- **Other**: 12.314s
- **Overhead Time**: 0.068s
- **Total Thread Count**: 96
- **Paused Time**: 0s

---

**Top Hotspots**

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>main$omp$parallel_fors434</td>
<td>stream_mod.x</td>
<td>273.620s</td>
</tr>
<tr>
<td>main$omp$parallel_fors333</td>
<td>stream_mod.x</td>
<td>257.123s</td>
</tr>
<tr>
<td>_intel_avx_rep_memory</td>
<td>libibm.so.5</td>
<td>210.412s</td>
</tr>
<tr>
<td>main$omp$parallel_fors323</td>
<td>stream_mod.x</td>
<td>206.630s</td>
</tr>
<tr>
<td>_kmp_fork_barrier</td>
<td>libibmp5.so</td>
<td>132.014s</td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td>36.221s</td>
</tr>
</tbody>
</table>

*N/A is applied to non-summable metrics.

---

**Effective CPU Utilization Histogram**

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.
Intel VTune Amplifier

Hotsps Hotsps by CPU Utilization

Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform

Grouping: Function / Call Stack

CPU Time

Function / Call Stack

Effective Time Spin Time Overhead Time Module Function (Full) Source File Start Addr

main$omp$parallel_for@343 273.620s 0s 0s stream_mod.x stream_mod.x main$omp$parallel_for@343 0x401a

CPU Time

Thread

O MP Worker Thread #83 (Tl...
O MP Worker Thread #84 (Tl...
O MP Worker Thread #80 (Tl...
O MP Worker Thread #85 (Tl...
O MP Worker Thread #76 (Tl...
O MP Worker Thread #82 (Tl...
O MP Worker Thread #59 (Tl...
O MP Worker Thread #75 (Tl...

CPU Utilization

FILTER 24.5%

Process Any Process Thread Any Thread Module Any Module Any Utilization Call Stack Mode User Functions Loop Mode Functions on inline Mode Show inline fu
<table>
<thead>
<tr>
<th>Address</th>
<th>Source</th>
<th>CPU Time</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40100</td>
<td></td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>0x40104</td>
<td></td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>0x40108</td>
<td></td>
<td>0.2%</td>
<td></td>
</tr>
<tr>
<td>0x4010c</td>
<td></td>
<td>14.7%</td>
<td></td>
</tr>
<tr>
<td>0x40110</td>
<td></td>
<td>10.3%</td>
<td></td>
</tr>
<tr>
<td>0x40114</td>
<td></td>
<td>0.1%</td>
<td></td>
</tr>
<tr>
<td>0x40118</td>
<td></td>
<td>jav 0x401e0 &lt;block 8&gt;</td>
<td></td>
</tr>
<tr>
<td>0x40160</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40164</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40168</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4016c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40170</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40174</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40178</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4017c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40180</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40184</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40188</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40190</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40194</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40198</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
327     times1[k] = mysecond() - times1[k];
329     times2[k] = mysecond();
330     #ifdef TUNED
331     tuned_STREAM_Add();
332     #else
333     #pragma omp parallel for
334     for (j=0; j<STREAM_ARRAY_SIZE; j++)
335     c[j] = a[i]*b[j];
336     #endif
337     times2[k] = mysecond() - times2[k];
338     times3[k] = mysecond();
340     #ifdef TUNED
341     tuned_STREAM_Triad(vector);
342     #else
343     #pragma omp parallel for
344     for (j=0; j<STREAM_ARRAY_SIZE; j++)
345     ...
346     #endif
350     /* --- SIMARAY --- */
352     for (k=1; k<NTIMES; k++) /* note -- skip first iteration
353     {
354     for (j=0; j<4; j++)
355     { avgtime3[j] = avgtime3[j] + times3[k];
356     minitime3[j] = MIN(minitime3[j], times3[k]);
357     maxtime3[j] = MAX(maxtime3[j], times3[k]);
359     }
```
Collecting Hotspots via EBS

amplxe-cl -c hotspots -knob sampling-mode=hw -r r_hshw_mod -- ./stream_mod.x
### Function / Call Stack

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>CPU Time</th>
<th>Instructions Retired</th>
<th>Microarchitecture Usage</th>
<th>Module</th>
<th>Function (Full)</th>
<th>Source File</th>
<th>Start Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>main$omp$parallel_for@343</td>
<td>255.234s</td>
<td>17,844,000,000</td>
<td>1.4%</td>
<td>stream_mod.x</td>
<td>main$omp$parallel...</td>
<td>stream_mod.c</td>
<td>0x401ada</td>
</tr>
<tr>
<td>main$omp$parallel_for@333</td>
<td>230.836s</td>
<td>17,400,000,000</td>
<td>1.6%</td>
<td>stream_mod.x</td>
<td>main$omp$parallel...</td>
<td>stream_mod.c</td>
<td>0x401cb5</td>
</tr>
<tr>
<td>intel_avx_rep_memcpy</td>
<td>186.121s</td>
<td>15,012,000,000</td>
<td>1.7%</td>
<td>stream_mod.x</td>
<td>intel_avx_rep_memcpy</td>
<td>stream_mod.c</td>
<td>0x401e70</td>
</tr>
<tr>
<td>INTERNAL_25__src_kmp_barrier.cpp_38a19466__kmp_wait_template__kmp</td>
<td>185.084s</td>
<td>7,728,000,000</td>
<td>0.8%</td>
<td>ibmmp.so</td>
<td>bool_INTERNAL_2__kmp_wait_template__kmp</td>
<td>ibmmp.so</td>
<td>0x48530</td>
</tr>
</tbody>
</table>

### Thread Utilization

- **Thread**
  - OMP Master Thread #0 (T...)
  - OMP Worker Thread #46 (T...)
  - OMP Worker Thread #47 (T...)
  - OMP Worker Thread #27 (T...)
  - OMP Worker Thread #26 (T...)
  - OMP Worker Thread #25 (T...)
  - OMP Worker Thread #24 (T...)
  - OMP Worker Thread #23 (T...)
  - OMP Worker Thread #22 (T...)
  - OMP Worker Thread #21 (T...)
  - CPU Time

**Filter**
- Running
- CPU Time
- Spin and Overhead
- Clocktick Sample

**Function**
- Any Function
- Any Module
- Any Utilization
- Call Stack Mode
- User function
- Loop Mode
- Functions or
- Inline Mode
- Show inline

**Values**
- 97.2%
Collecting HPC Performance

amplxe-cl -c hpc-performance -r r_hpc_mod -- ./stream_mod.x
Elapsed Time: 21.772s

SP GFLOPS: 0.000
DP GFLOPS: 0.990
x87 GFLOPS: 0.000
CPI Rate: 9.293
Average Frequency: 2.4 GHz
Total Thread Count: 96

Effective Physical Core Utilization: 44.3% (21.274 out of 48)

Effective Logical Core Utilization: 43.9% (42.101 out of 96)

Serial Time (outside parallel regions): 11.448s (62.8%)

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the Vast time of the master thread, it may significantly exceed the aggregated CPU time in the table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Serial CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[vmlinux]</td>
<td>vmlinux</td>
<td>4.962s</td>
</tr>
<tr>
<td>[Loop at line 258 in main]</td>
<td>stream_mod.x</td>
<td>4.210s</td>
</tr>
<tr>
<td>[Loop at line 462 in checkSTREAMresults]</td>
<td>stream_mod.x</td>
<td>2.225s</td>
</tr>
<tr>
<td>func@0x7f3d40</td>
<td>libtrim_collector.so</td>
<td>0.005s</td>
</tr>
<tr>
<td>[sep0]</td>
<td>sep0</td>
<td>0.005s</td>
</tr>
</tbody>
</table>

* N/A is applied to non-summable metrics.

Parallel Region Time: 10.323s (47.4%)

Effective CPU Utilization Histogram

Memory Bound: 86.7% of Pipeline Slots

Cache Bound: 32.5% of Clockticks

DRAM Bound: 51.1% of Clockticks

DRAM Bandwidth Bound: 43.9% of Elapsed Time

NUMA % of Remote Accesses: 32.8%

Bandwidth Utilization Histogram

Vectorization: 100.0% of Packed FP Operations
Memory Bound: 86.7% of Pipeline Slots

- Cache Bound: 32.5% of Clockticks
- DRAM Bound: 51.1% of Clockticks
- DRAM Bandwidth Bound: 43.9% of Elapsed Time
- NUMA: % of Remote Accesses: 32.8%

Bandwidth Utilization Histogram

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: DRAM, GB/sec

Bandwidth Utilization Histogram

This histogram displays the time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium, and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and interconnect bandwidth.
### Optimization Notice

Intel Internal Only

### Elapsed Time: 21.772s

- **SP GFLOPS:** 0.000
- **DP GFLOPS:** 0.090
- **X87 GFLOPS:** 0.000
- **CPU Rate:** 9.233%
  - Average CPU Frequency: 2.4 GHz
  - Total Thread Count: 96

**Effective Physical Core Utilization:** 44.3% (21.274 out of 48)

**Effective Logical Core Utilization:** 43.9% (42.101 out of 96)

**Serial Time (outside parallel regions):** 10.445s (52.6%)

**Effective CPU Utilization Histogram**

**Memory Bound:** 86.7% of Pipeline Slots

- Cache Bound: 32.5% of Clockticks
- DRAM Bound: 51.1% of Clockticks
  - DRAM Bandwidth Bound: 43.9% of Elapsed Time
  - Bandwidth Utilization Histogram

**NUMA % of Remote Accesses:** 32.8% of Elapsed Time

**Vectorization:** 100.0% of Packed FP Operations

### CPU Time

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>Effective Time by Utilization</th>
<th>Spin &amp; Overhead Time</th>
<th>Serial CPU Time</th>
<th>Memory Bound</th>
<th>NUMA % of Remote Accesses</th>
<th>% of FP Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop at line 343 in main$momp$parallel_for@343</td>
<td>249.314s</td>
<td>0s</td>
<td>0s</td>
<td>96.5%</td>
<td>30.3%</td>
<td>22.7%</td>
</tr>
<tr>
<td>Loop at line 333 in main$momp$parallel_for@333</td>
<td>228.390s</td>
<td>0s</td>
<td>0s</td>
<td>96.2%</td>
<td>36.0%</td>
<td>8.3%</td>
</tr>
<tr>
<td>Loop at line 323 in main$momp$parallel_for@323</td>
<td>163.500s</td>
<td>0s</td>
<td>0s</td>
<td>96.5%</td>
<td>16.7%</td>
<td>0.5%</td>
</tr>
<tr>
<td>Loop at 0x468d0 in __intel_avx4_rep_memory</td>
<td>179.688s</td>
<td>0s</td>
<td>0s</td>
<td>98.2%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Loop at line 361 in INTERNAL_25_src_k</td>
<td>63.419s</td>
<td>0s</td>
<td>0s</td>
<td>96.0%</td>
<td>4.0%</td>
<td>0.0%</td>
</tr>
<tr>
<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>
</tr>
</tbody>
</table>
```c
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
    c[j] = a[j];
```
```c
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
c[j] = a[j];
```
```
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
    b[j] = scalar*c[j];
```
```c
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
    c[j] = a[j]+b[j];
```
343 #pragma omp parallel for
344 for (j=0; j<STREAM_ARRAY_SIZE; j++)
345     a[j] = b[j]+scalar*c[j];
Collecting Memory Access

amplxe-cl -c memory-access -r ./r_ma_mod -- ./stream_mod.x
Elapsed Time: 21.897s

CPU Time: 1028.961s

Memory Bound:
  - L1 Bound: 87.0% of Pipeline Slots
  - L2 Bound: 21.2% of Clockticks
  - L3 Bound: 0.2% of Clockticks

DRAM Bound:
  - DRAM Bandwidth Bound: 10.9% of Clockticks
  - UPI Bandwidth Bound: 45.5% of Elapsed Time

Memory Latency:
  - Local DRAM: 24.5% of Clockticks
  - Remote DRAM: 19.9% of Clockticks
  - Remote Cache: 0.0% of Clockticks

NUMA, % of Remote Accesses: 34.4%

Loads: 89,315,679,390
Stores: 19,964,596,020

LLC Miss Count: 2,121,148,470

Average Latency (cycles): 219
Total Thread Count: 96
Paused Time: 0s

*NA is applied to metrics with undefined value. There is no data to calculate the metric.

Bandwidth Utilization Histogram

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: DRAM, GB/sec

Bandwidth Utilization Histogram

This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and Interconnect bandwidth.
Collecting Memory Access with Objects

amplxe-cl -c memory-access -knob analyze-mem-objects=true -r ./r_mao_mod -- ./stream_mod.x
printf("Number of Threads requested = %d\n", k);
}
#endif

if (_OPENMP)
  k = 0;
#pragma omp parallel
#pragma omp atomic
  k++;
  printf("Number of Threads counted = %d\n", k);
#endif

/* Get initial value for system clock. */
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++) {
[i] = [j];
  b[j] = 2.0;
  c[j] = 0.0;
}

printf(MLINE);

if ( (quantum = checktick()) >= 1)
  printf("Your clock granularity/precision appears to be ", quantum);
else {
  printf("Your clock granularity appears to be ",
  "less than one microsecond.\n”);
  quantum = 1;
}

  t = mysecond();
#pragma omp parallel for
Linux first touch policy

- Memory is assigned to NUMA domains
  - not during the (default) allocation
  - but when the memory is being touched by the first time
- The NUMA domain that will get the memory assigned as local memory, is therefore the domain from where the corresponding thread touched the memory for the first time
Fine, but what about the mem BW increase on socket #2?
Linux Kernel page migration

- New in RHEL 7 / SLES 12
- Default configuration is ON
- Introduces background noise, bad for benchmarking

$ cat /proc/sys/kernel/numa_balancing

What if we would increase the runtime from 10 iterations to 100?
Let’s fix it!
**Elapsed Time**: 7.490s  
- SP GFLOPS: 0.000  
- DP GFLOPS: 3.818  
- x87 GFLOPS: 0.000  
- CPI Rate: 10.039  
- Average CPU Frequency: 2.4 GHz  
- Total Thread Count: 96

**Effective Physical Core Utilization**: 61.1% (29.339 out of 48)

- Effective Logical Core Utilization: 60.8% (56.327 out of 96)
- Serial Time (outside parallel regions): 2.872s (38.3%)

**Top Serial Hotspots (outside parallel regions)**

- This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the Vtime of the master thread, it may significantly exceed the aggregated CPU time in the table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Serial CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Loop at line 462 in checkSTREAMresults]</td>
<td>stream.x</td>
<td>2.140s</td>
</tr>
<tr>
<td>[vmlinux]</td>
<td>vmlinux</td>
<td>0.692s</td>
</tr>
<tr>
<td>func@0x7f3d40</td>
<td>libthtnotify_collector.so</td>
<td>0.005s</td>
</tr>
<tr>
<td>strcmp</td>
<td>libc-2.17.so</td>
<td>0.005s</td>
</tr>
</tbody>
</table>

* N/A is applied to non-summable metrics

- **Parallel Region Time**: 4.618s (61.7%)
- **Effective CPU Utilization Histogram**

**Memory Bound**: 88.7% of Pipeline Slots

- Cache Bound: 29.0% of Clockticks
- DRAM Bound: 50.2% of Clockticks

- **DRAM Bandwidth Bound**: 56.2% of Elapsed Time
- NUMA: % of Remote Accesses: 0.3%

- **Bandwidth Utilization Histogram**

**Vectorization**: 100.0% of Packed FP Operations

- Instruction Mix
Some earlier problem indicators we missed?

Besides being fooled by filters and zoom
Elapsed Time: 21.772s
- SP GFLOPS: 0.000
- DP GFLOPS: 0.090
- x87 GFLOPS: 0.000
- CPI Rate: 9.293
- Average CPU Frequency: 2.4 GHz
- Total Thread Count: 96

Effective Physical Core Utilization: 44.3% (21.274 out of 48)

Serial Time (outside parallel regions): 11.449s (62.6%)

Top Serial Hotspots (outside parallel regions)
This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the VWait time of the master thread, it may significantly exceed the aggregated CPU time in the table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Serial CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Loop at line 256 in main]</td>
<td>stream_mod.x</td>
<td>4.210s</td>
</tr>
<tr>
<td>[Loop at line 256 in main]</td>
<td>stream_mod.x</td>
<td></td>
</tr>
<tr>
<td>func@0x7d340</td>
<td>libbitcoin_runner.so</td>
<td>0.005s</td>
</tr>
<tr>
<td>[septo]</td>
<td>sep5</td>
<td>0.005s</td>
</tr>
</tbody>
</table>

*N/A is applied to non-summable metrics.

Parallel Region Time: 10.323s (47.4%)

Effective CPU Utilization Histogram

Memory Bound: 86.7% of Pipeline Slots
- Cache Bound: 32.5% of Clockticks
- DRAM Bound: 51.1% of Clockticks
- DRAM Bandwidth Bound: 43.9% of Elapsed Time
- NUMA % of Remote Accesses: 32.8%

Bandwidth Utilization Histogram

Vectorization: 100.0% of Packed FP Operations
Memory Bound: 86.7% of Pipeline Slots

- Cache Bound: 32.5% of Clockticks
- DRAM Bound: 51.1% of Clockticks
- DRAM Bandwidth Bound: 43.9% of Elapsed Time
- NUMA: % of Remote Accesses: 32.8%

Bandwidth Utilization Histogram

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: DRAM, GB/sec

This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and interconnect bandwidth.
How to Use Intel® Trace Analyzer and Collector
It’s Easy…

Run your binary and create a tracefile:

$ mpirun -trace -n 2 ./test

View the results:

$ traceanalyzer &
Views and Charts

- Helps navigating through the trace data and keep orientation.
- Every View can contain several Charts.
- All Charts in a View are linked to a single:
  - Time-span
  - Set of threads
  - Set of functions
- All Charts follow changes to View (e.g., zooming)
### Event Timeline

- Get detailed impression of program structure.
- Display functions, messages, and collective operations for each process/thread along time axis.
- Retrieve detailed event information.
Flat Function Profile
Statistics About Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>TSelf</th>
<th>TSelf /</th>
<th>TTotal</th>
<th>#Calls</th>
<th>TSelf /Call</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>FREC0</em></td>
<td>678,747</td>
<td>4,463</td>
<td>682,767</td>
<td>4,463</td>
<td>0,013,703</td>
</tr>
<tr>
<td><em>KMP_SREC</em></td>
<td>524,341</td>
<td>39,208</td>
<td>563,549</td>
<td>39,208</td>
<td>0,014,225</td>
</tr>
<tr>
<td><em>MATHVUL</em></td>
<td>410,131</td>
<td>31,260</td>
<td>441,391</td>
<td>31,260</td>
<td>0,010,598</td>
</tr>
<tr>
<td><em>SOLVER</em></td>
<td>329,918</td>
<td>25,605</td>
<td>355,523</td>
<td>25,605</td>
<td>0,010,762</td>
</tr>
<tr>
<td><em>User_Code</em></td>
<td>165,164</td>
<td>12,988</td>
<td>178,152</td>
<td>12,988</td>
<td>0,103,717</td>
</tr>
<tr>
<td><em>MPI_Bcast</em></td>
<td>94,227</td>
<td>7,414</td>
<td>101,641</td>
<td>7,414</td>
<td>0,098,950</td>
</tr>
<tr>
<td><em>ASSEMBLY</em></td>
<td>43,822</td>
<td>3,383</td>
<td>47,205</td>
<td>3,383</td>
<td>0,100,865</td>
</tr>
<tr>
<td><em>MPI_BARRIER</em></td>
<td>24,826</td>
<td>1,913</td>
<td>26,749</td>
<td>1,913</td>
<td>0,099,911</td>
</tr>
<tr>
<td><em>MI_P_Reduce</em></td>
<td>23,067</td>
<td>1,846</td>
<td>24,913</td>
<td>1,846</td>
<td>0,098,168</td>
</tr>
<tr>
<td><em>MPI_Waitall</em></td>
<td>17,067</td>
<td>1,330</td>
<td>18,397</td>
<td>1,330</td>
<td>0,098,245</td>
</tr>
<tr>
<td><em>MPI_Comm_dup</em></td>
<td>11,756</td>
<td>954</td>
<td>12,710</td>
<td>954</td>
<td>0,100,696</td>
</tr>
<tr>
<td><em>MPI_received</em></td>
<td>7,590</td>
<td>613</td>
<td>8,203</td>
<td>613</td>
<td>0,100,696</td>
</tr>
<tr>
<td><em>MPI_Wtime</em></td>
<td>7,470</td>
<td>600</td>
<td>8,070</td>
<td>600</td>
<td>0,100,696</td>
</tr>
<tr>
<td><em>MPI_Proc</em></td>
<td>4,017</td>
<td>324</td>
<td>4,341</td>
<td>324</td>
<td>0,100,696</td>
</tr>
<tr>
<td><em>MPI_Finalize</em></td>
<td>0,006</td>
<td>0,006</td>
<td>0,006</td>
<td>0,006</td>
<td>0,006,006</td>
</tr>
<tr>
<td><em>KMP_C tries</em></td>
<td>0,121</td>
<td>0,121</td>
<td>0,121</td>
<td>0,121</td>
<td>0,121,012</td>
</tr>
<tr>
<td><em>KMP_C_comm reimb</em></td>
<td>0,000</td>
<td>0,000</td>
<td>0,000</td>
<td>0,000</td>
<td>0,000,000</td>
</tr>
</tbody>
</table>

Optimization Notice
Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Communication Profiles

- Statistics about point-to-point or collective communication
- Generic matrix supports grouping by several attributes in each dimension: Sender, receiver, data volume per message, tag, communicator, type
- Available attributes: Count, bytes transferred, time, transfer rate
MPI Performance Analysis
Initial MPI-3.x Support

Automatic Performance Assistant

- Detects common MPI performance issues
- Automated tips on potential solutions

Automatically detect performance issues and their impact on runtime
Ideal Interconnect Simulator (Idealizer)
Helps to Figure Out Application's Imbalance, Simulating its Behavior in the ‘Ideal’ Communication Environment

Real trace

Ideal trace

Hot Spot

Easy Way to Identify Application Bottlenecks
Application Imbalance Diagram
Intuitive Diagram for Simplified Application Analysis

Combined information in one location:
- Load imbalance
- MPI overall time
- MPI Interconnect time
- Different breakdowns, etc.

Basic building block: Breakdown of a single run time into three colors:
- Physical MPI (library, interconnect)
- Imbalance (idling in MPI; application intrinsic)
- ‘Pure’ calculation

Simplified Application Analysis Helps to Identify Performance Issues
MPI Correctness Checking
Automatically Checks MPI Correctness

- Solves two problems:
  1. Finding MPI programming mistakes
  2. Detecting errors in execution environment
- Integrates with a debugger
- No manual instrumentation needed
- Errors reported to screen or embedded trace file
COMMAND LINE INTERFACES & BATCH JOB SUBMISSION
VTune CLI: syntax

- VTune command line application amplxe-cl
  amplxe-cl <action> [-action-option] [-global-option] [[--] <target> [target-options]]

- **-action**: usually either *collect* or *report*
- **-action-option**: modifies the behaviour of an action
- **-global-option**: adjusts global settings
- **<target>**: denotes the target application to profile

> amplxe-cl -collect advanced-hotspots -r result_dir -- ./app
VTune CLI: collect

• Syntax:
  -c[ollect] <analysis type> [-analysis-option]

  • The type of analysis defined with analysis type

  • Analysis type defines the set of available analysis-option modifiers or ”knob”s

• Command line help with -help on each analysis type and available knobs

> amplxe-cl -help -collect # List analysis types available
> amplxe-cl -help -collect hotspots # List knobs for “hotspots”
$ amplxe-cl -help

Intel(R) VTune(TM) Amplifier Command Line Tool
Copyright (C) 2009-2019 Intel Corporation. All rights reserved.


Type 'amplxe-cl -help <action>' for help on a specific action.

Available Actions:

  collect          Choose an analysis type.
  collect-with     Choose a collector.
  command          Issue a command to a running collection.
  finalize         Re-finalize the result.
  help             Display help text.
  import           Create a result directory by importing one or more legacy data files/directories.
  report           Generate a report with the specified name.
  version          Display product version.

... $ amplxe-cl -help collect
...
Vtune 2/2

-r, -result-dir=<string> (r@@@{at})
-target-duration-type=veryshort | short | medium | long (short)

Available Analysis Types:

Want to find out where your application spends time and optimize your algorithms?

hotspots
  Identify the most time consuming functions and lines of source code.

memory-consumption
  Analyze memory consumption by your application, its distinct memory objects and their allocation stacks.

Want to see how efficiently your code is using the underlying hardware?

uarch-exploration
  Analyze CPU microarchitecture bottlenecks affecting the performance of your application.

memory-access
  Measure a set of metrics to identify memory access related issues.

Want to assess the compute efficiency of your multi-threaded application?

threading
  Discover how well your application is using parallelism to take advantage of all available CPU cores.

hpc-performance
  Analyze important aspects of your application performance, including CPU utilization with details on OpenMP efficiency analysis, memory access, and vectorization information.
Advisor

$ advixe-cl -help collect

Examples:

1) Survey the application to determine hotspots.
   advixe-cl --collect survey --project-dir ./advi --search-dir src:r=./src
   --./bin/myApplication

2) Collect memory access patterns data with specified loops for analysis.
   advixe-cl --collect map --mark-up-list=5,10,12 --project-dir ./advi
   --search-dir src:r=./src --./bin/myApplication

3) Collect survey data on 4 nodes of MPI cluster into the shared ./advi project directory.
   mpirun -n 4 advixe-cl --project-dir ./advi --collect survey
   -- <PATH>/mpi-sample/1_mpi_sample_serial

4) Collect dependencies data for all loops that are both innermost and hold above 2% of the total CPU time.
   advixe-cl --collect dependencies --project-dir ./advi --loops="loop-height=0,total-time>2"
   --./bin/myApplication

5) Collect roofline for the application to reveal performance bounds and bottlenecks with a graphical model.
   advixe-cl --collect roofline --project-dir ./advi -- ./bin/myApplication
How can I apply VTune / Advisor to a subset of MPI Ranks only?

#!/bin/bash
#SBATCH ...
...

module ...

cp $I_MPI_ROOT/test/test.c .
mpiicc ./test.c -g -O2 -o ./test.x

cat << EOF > ./multiprog.conf
0-39  ./test.x
40    amplxe-cl -c hotspots -r rh001_srun -- ./test.x
41-79  ./test.x
EOF

#80 ranks
srun --multi-prog ./multiprog.conf

# alternatively gtool in mpirun
export I_MPI_PIN_RESPECT_CPUS=0
mpirun -gtool "amplxe-cl -c hotspots -r rh001_mpirun:40" ./test.x
$ source /mpcdf/soft/SLE_12_SP3/packages/x86_64/intel_parallel_studio/2019.4/vtune_amplifier/apsvars.sh
$ aps -h

Intel(R) VTune(TM) Amplifier 2019 Update 4 (build 597835) Command Line Tool
Copyright (C) 2009-2019 Intel Corporation. All rights reserved.
Usage: 1. aps [--result-dir=<dir> | -r=<dir>] [-c=<mode>] <app>
   2. <mpi_launcher> <mpi_parameters> aps [-c=<mode>] <app>
   3. aps --report=<dir> | -R=<dir>
   4. aps <option>

1. Run analysis for an application or script <app> and store results in <dir>
2. Run analysis for an MPI application <app> and store results in <dir>
   --collection-mode=<mode> Specify a comma separated list of data to collect. Possible values: hwc - hardware
counters, omp - OpenMP statistics, mpi - MPI statistics, all - all possible data (default).
   --start-paused Start data collection in the paused mode. Collection resumes when resume API call is
   reached.
3. Show analysis report based on data from <dir>
   Tip: use
       aps-report <options> <result_dir>
   to review more MPI usage metrics. Additional details include statistics view by ranks, message sizes, collective
   operations, and communicators as well as the ability to explore rank-to-rank and node-to-node communication statistics.
4. Show additional product info. Where <option> can be:
   --help, -h show this help and exit
   --version show version information and exit

$ srun /mpirun ... aps ./my_application
ITAC

$ source /mpcdf/soft/SLE_12_SP3/packages/x86_64/intel_parallel_studio/2019.4/itac_latest/intel64/bin/itacvars.sh

Either

$ export LD_PRELOAD=libVT.so  #for the ITAC Tracing

Or

$ export LD_PRELOAD=libVTmc.so  #for the ITAC Message Checker

$ srun ./my_application

Optional:

```
VT_LOGFILE_FORMAT=STFSINGLE  #single tracefile
VT_PCTRACE=4  #for stack unwinding (requires symbols)
```

Consider:

```
MPI_Pcontrol(1/0)  #limit tracing to particular ranks or routines
```