

EFFECTIVE USAGE OF VTUNE™ AMPLIFIER XE ON THETA

Larry Meadows, Intel DCG/E&G/HEAT

ATPESC, St. Charles, IL, August 8, 2017

The Processor

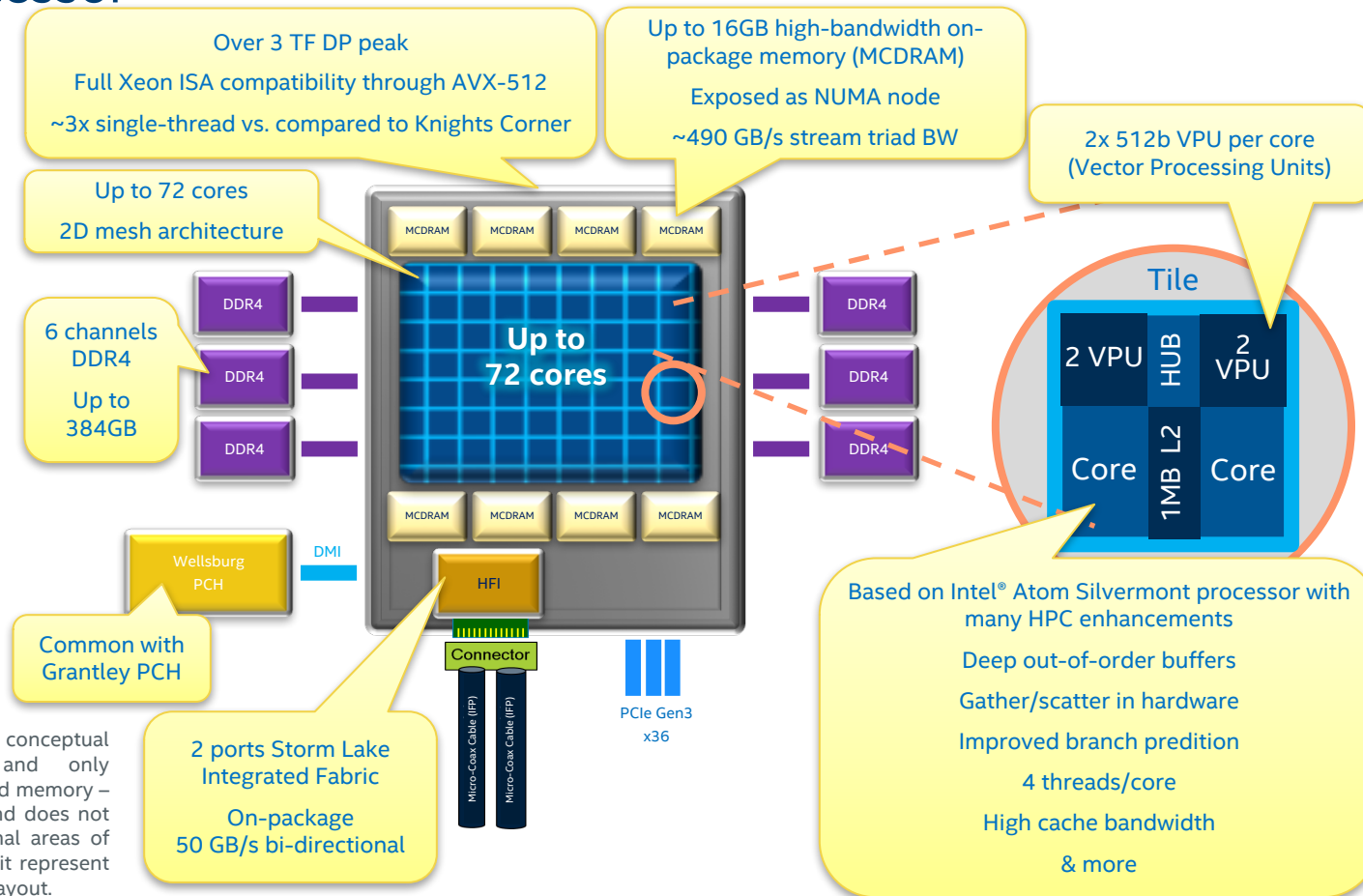


Diagram is for conceptual purposes only and only illustrates a CPU and memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

Core Features

Hyperthreading	4 SMT threads
Out-of-order	72 entry reorder buffer
VPU	2x AVX-512
High Bandwidth memory	16GiB on-package MCDRAM
L1 Data Cache	32KiB, 8-way, 2 lines read, 1 line write/cycle
L1 Instruction Cache	32KiB, 8-way
L2 Shared Cache	1024KiB (per 2-core tile), 16-way, 1 line read, ½ line write/cycle
L2 TLB	256 x 4K, 128 x 2M, 16 x 1G pages
Retirement width	2 per cycle

Throughput, Optimization and Analysis

Peak instruction throughput occurs when every core is retiring two full-width SIMD operations per clock.

Peak instruction throughput requires parallelization (use all the cores) and vectorization (use all the SIMD lanes).

Peak instruction throughput may not be achievable due to some other limiter, typically memory bandwidth.

Performance optimization attempts to achieve peak throughput.

Performance analysis attempts to discover why peak throughput is not reached.

This talk is mostly about performance analysis.

Impediments to Peak Throughput

Imperfect parallel scalability

- Algorithmic: synchronization and load imbalance (including serial time)
- Architectural: cache contention, memory BW limited
- On- and off-chip interconnect limited (on-chip == Mesh, off-chip == Fabric)

Imperfect SIMD usage

- Non-vectorizable algorithms. Rewrite your code to make it vectorizable.
- Other limitations
 - ISA limitations (e.g., data type support)
 - Compiler limitations (always improving)
 - Sparse masks due to branchy code (less work per instruction)

Finding Impediments with Vtune

This is the way I use vtune. Other people use it differently. That's fine.

Use only the hardware event types, primarily advanced hotspots.

Execution model has two concepts:

1. Threads are user application threads, usually bound to hardware threads.
2. Code segments are shared libraries (MPI, OpenMP) or functions of interest

Threads execute code segments and vtune gives us time per code segment per thread.

Comparing segment times across threads tells us about parallel execution.

Analyzing code and hardware events within a thread and segment tells us about processor performance.

Case Study

Application: miniFE <https://github.com/Mantevo/miniFE>

Hybrid OpenMP+MPI sparse solver

Run on 1 node*, 8 MPI ranks

- Run and collect vtune results
- Preliminary overview
- Parallel performance and bottlenecks
- Processor performance characteristics and analysis
- When am I done?

* Vtune installation available on only 1 node of theta at this writing

Run and Collect Vtune Results

Run vtune with hardware counter sampling on only one MPI rank on one node, but collect data for the whole node. Don't resolve symbols (finalize).

```
aprun -cc depth -j 4 -d 32 -n 8 -N 8 sh ./run.sh
```

run.sh:

```
export PE_RANK=$ALPS_APP_PE
export PMI_NO_FORK=1
if [ "$PE_RANK" == "0" ];then
    ampxe-cl -collect advanced-hotspots -analyze-system \
        -finalization-mode=none \
        ./miniFE.x nx=300
else
    ./miniFE.x nx=300
fi
```


Preparing and Viewing Results

Switch to login node and finalize:

```
source /opt/intel/vtune_amplifier_xe/amplxe-vars.sh
```

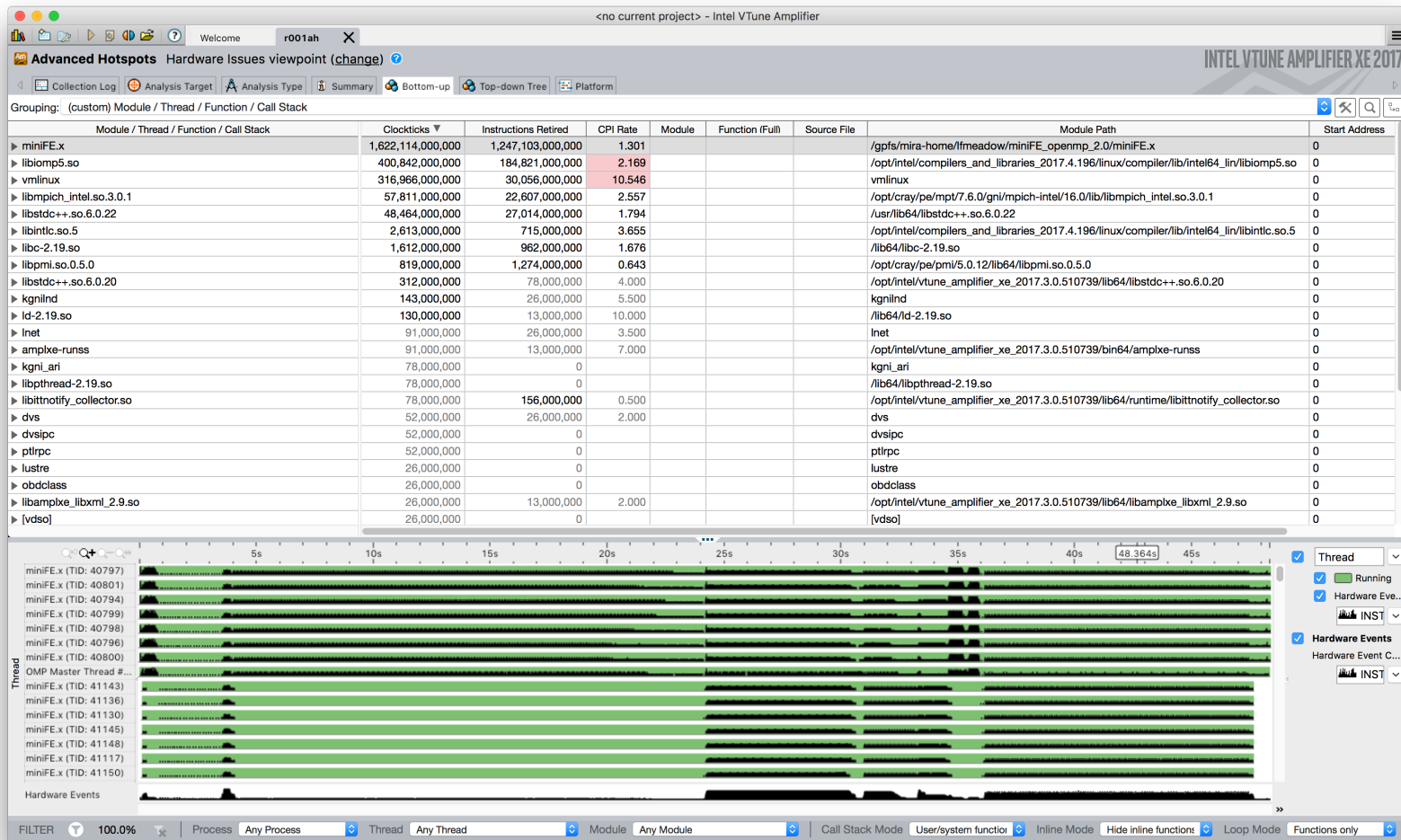
```
amplxe-cl -finalize -search-dir . -r r000ah/
```

- You can name the result with `-r` on the collection line, or vtune names it for you; use the same name here
- Semi-advanced: add `-search-dir` for runtime libraries, e.g.:
`-search-dir /opt/cray/pe/pmi/5.0.12/lib64`

```
amplxe-gui r000ah
```

- Needs X connection
- I copied the result to my local system and ran the GUI there.

Full GUI screenshot



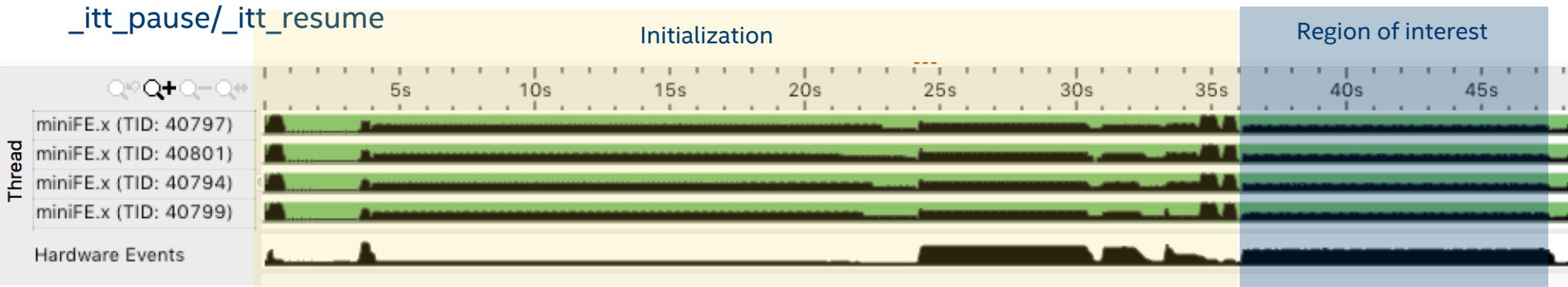
Initial Grid and Timeline

Link with `-dynamic` and
automatically bucket time by
`app/MPI/OpenMP/vmlinux`

Initialization can skew
results. Zoom in or use
`_itt_pause/_itt_resume`

Grouping: (custom) Module / Thread / Function / Call Stack

Module / Thread / Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▶ miniFE.x	1,622,114,000,000	1,247,103,000,000	1.301
▶ libiomp5.so	400,842,000,000	184,821,000,000	2.169
▶ vmlinux	316,966,000,000	30,056,000,000	10.546
▶ libmpich_intel.so.3.0.1	57,811,000,000	22,607,000,000	2.557
▶ libstdc++.so.6.0.22	48,464,000,000	27,014,000,000	1.794
▶ libintlc.so.5	2,613,000,000	715,000,000	3.655
▶ libc-2.19.so	1,612,000,000	962,000,000	1.676
▶ libpmi.so.0.5.0	819,000,000	1,274,000,000	0.643
▶ libstdc++.so.6.0.20	312,000,000	78,000,000	4.000



Vtune Data Collection

Theory of operation

- `-collect advanced-hotspots` uses 3 fixed counters: instructions retired, thread cycles (at actual frequency), and reference cycles (at nominal fixed frequency, 1.3GHz on Theta)
- Each event is programmed to interrupt every N occurrences and a sample with the timestamp, instruction pointer, process/thread id, and hardware thread is recorded.
- Useful metrics: elapsed time, time running vs. halted, frequency ratio, instructions/cycle
- Yields a statistical profile. View with GUI or command-line reports

Avoid pain points

- You don't need data for every MPI node. Node 0 is usually enough.
- Collect for the whole system with one `amplxe-cl` instance, the driver does it anyway and the overhead is low with SEP driver.
- Collect only the interesting bits. Use `__itt_resume()/__itt_pause()` and `-start-paused`. You probably don't need all the iterations.
- Finalize outside of job run.

Parallel Performance and Imbalance

Grouping: (custom) Module / Thread / Function / Call Stack

Module / Thread / Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▼ miniFE.x	22,061,000,000	14,950,000,000	1.476
▶ miniFE.x (TID: 40801)	11,050,000,000	7,553,000,000	1.463
▶ miniFE.x (TID: 41103)	11,011,000,000	7,397,000,000	1.489
▼ libiomp5.so	2,847,000,000	1,482,000,000	1.921
▶ miniFE.x (TID: 41103)	2,665,000,000	1,417,000,000	1.881
▶ miniFE.x (TID: 40801)	182,000,000	65,000,000	2.800
▼ libmpich_intel.so.3.0.1	2,574,000,000	1,027,000,000	2.506
▶ miniFE.x (TID: 40801)	2,574,000,000	1,027,000,000	2.506
▶ vmlinux	351,000,000	39,000,000	9.000

- Select grouping with thread on top, choose one master and one slave thread, filter in by selection, then switch grouping.
- Time spent by the slave in libiomp5.so is equal to time spent by the master in libmpich.

Load Imbalance

- Elapsed time is determined by longest executing thread
- Threads that finish work early spin-wait in OpenMP.
- Threads doing MPI may spin wait waiting for remote sender
- We can see this by comparing times for each thread in each module
- Vtune grouping lets you do this in the GUI but it is hard to compare many threads
- One good technique is a pivot table: threads vs. modules.
- Generate a csv file with `amplxe-cl -report` and import into spreadsheet or use python+pandas.
- May need other tools for global MPI imbalance, then zero in on slowest node with vtune

Pivot Table Example

```
amplxe-cl -report hw-events -r result -group-by=package,cpuuid,core,thread,function \  
-format=csv -csv-delimiter=comma -inline-mode=off -time-filter 37:47 >file.csv
```

Thread	miniFE.x	libiomp5.so	libmpich_intel.so.3.0.1	vmlinux
miniFE.x (TID: 40799)	9,854,000,000	169,000,000	2,782,000,000	117,000,000
miniFE.x (TID: 40794)	9,893,000,000	234,000,000	2,730,000,000	65,000,000
miniFE.x (TID: 40798)	10,127,000,000	26,000,000	2,626,000,000	130,000,000
miniFE.x (TID: 40797)	10,088,000,000	156,000,000	2,587,000,000	78,000,000
miniFE.x (TID: 40800)	10,023,000,000	169,000,000	2,535,000,000	182,000,000
miniFE.x (TID: 40796)	10,205,000,000	78,000,000	2,509,000,000	130,000,000
miniFE.x (TID: 40801)	10,101,000,000	247,000,000	2,431,000,000	130,000,000
OMP Master Thread #0 (TID: 40828)	12,454,000,000	65,000,000	78,000,000	91,000,000
miniFE.x (TID: 41125)	9,828,000,000	2,938,000,000		156,000,000
OMP Worker Thread #6 (TID: 41158)	9,828,000,000	2,912,000,000		182,000,000
miniFE.x (TID: 41117)	10,179,000,000	2,496,000,000		247,000,000
miniFE.x (TID: 41124)	10,114,000,000	2,483,000,000		325,000,000
miniFE.x (TID: 41114)	10,231,000,000	2,457,000,000		234,000,000

- 13 out of 64 threads shown, others similar. Entries are reference clocks (elapsed time). Sorted by MPI clocks then by OMP clocks. MPI threads on the top. Small load imbalance in app, lots of spin time in slave threads due to MPI time in master (except 0).
- Master thread 0 is a major laggard, see next slide

What's up with master thread 0?

Slow performance of master thread 0 was a surprise (Honest!). Perfect opportunity to teach and learn!

Diagnose using vtune filtering and grouping. We want to compare two hardware contexts that are running master (MPI) threads.

1. Add custom grouping thread/hw-context/function and find which core master thread 0 is on (cpu 0) and another master thread (cpu 8)*
2. Switch to package/hw-context grouping and select 0 and 8 and filter in.
3. Add custom grouping hw-context/Module and switch to that
4. Filter range on timeline (only if not using pause/resume API).

* We already knew this from aprun but good practice

CPU 0 vs. CPU 8

Grouping: (custom) H/W Context / Module / Thread / Function

H/W Context / Module / Thread / Function	Clockticks ▼	Instructions Retired	CPI Rate
▼ cpu_0	13,923,000,000	7,384,000,000	1.886
▶ miniFE.x	13,468,000,000	7,345,000,000	1.834
▼ vmlinux	260,000,000	13,000,000	20.000
▶ OMP Master Thread #0 (TID: 40828)	130,000,000	0	
▶ ampxe-runss (TID: 41102)	130,000,000	13,000,000	10.000
▶ libiomp5.so	91,000,000	0	
▶ ld-2.19.so	26,000,000	0	
▶ libmpich_intel.so.3.0.1	26,000,000	26,000,000	1.000
▶ dvsipc	13,000,000	0	
▶ dvs	13,000,000	0	
▶ libc-2.19.so	13,000,000	0	
▶ libampxe_libxml_2.9.so	13,000,000	0	
▼ cpu_8	13,910,000,000	8,567,000,000	1.624
▶ miniFE.x	10,530,000,000	7,423,000,000	1.419
▶ libmpich_intel.so.3.0.1	2,990,000,000	1,053,000,000	2.840
▶ libiomp5.so	260,000,000	78,000,000	3.333
▶ vmlinux	104,000,000	13,000,000	8.000
▶ ld-2.19.so	26,000,000	0	

cpu_0 and cpu_8 execute same number of instructions in miniFE.x but cpu_0 uses 3.5e9 more clockticks. No other activity on cpu_0. Why is it so slow?

I finally remember:
There are other threads on the same core as CPU 0. Let's look at them.

Switch to core/hw context/module grouping and look at core 0

Core 0

Grouping: (custom) Core / H/W Context / Module / Function

Core / H/W Context / Module / Function	Clockticks ▼	Instructions Retired	CPI Rate
▼ core_0	18,603,000,000	8,541,000,000	2.178
▼ cpu_0	13,923,000,000	7,384,000,000	1.886
▶ miniFE.x	13,468,000,000	7,345,000,000	1.834
▶ vmlinux	260,000,000	13,000,000	20.000
▶ libiomp5.so	91,000,000	0	
▶ ld-2.19.so	26,000,000	0	
▶ libmpich_intel.so.3.0.1	26,000,000	26,000,000	1.000
▶ dvsipc	13,000,000	0	
▶ dvs	13,000,000	0	
▶ libc-2.19.so	13,000,000	0	
▶ libamplxe_libxml_2.9.so	13,000,000	0	
▼ cpu_64	4,641,000,000	1,157,000,000	4.011
▶ vmlinux	4,641,000,000	1,157,000,000	4.011
▶ cpu_192	26,000,000	0	
▶ cpu_128	13,000,000	0	

OS CPU 64 on core 0 executes for 4.7e9 clockticks in the OS, greatly slowing down OS CPU 0, also on core 0.

Possible remedy: avoid core 0 (or even tile 0):
`aprun -r 1 (?)`

With automated pivot tables we would have seen this immediately.

Summary of Parallel Analysis

Look at time spent per-thread, that determines elapsed time.

Vtune filtering and grouping is very powerful for ad-hoc analysis.

Getting threads side-by-side to compare in the vtune gui is hard, using the trick of selecting two threads with different behaviors helps.

Pivot tables summarize thread vs. module/function behavior well for large numbers of threads, make it easier to spot groups of threads with unique behavior (serial time, MPI time, systemic load imbalance, misbehaving cores).

Pivot tables are useful for lots of other tasks such as checking for correct affinity to HW contexts and unexpected noise; automating it with python+pandas is a promising approach.

Processor Performance

Standard metric is retired Instructions Per Cycle (IPC)

- Intel® Xeon Phi™ processor max is 2 IPC per core
- vtune displays the reciprocal CPI
- Computed as $\frac{\sum_{hwthreads} instructions}{\sum_{hwthreads} cycles}$
 - *instructions* == INST_RETIRED.ANY *cycles* == CPU_CLK_UNHALTED.THREAD

Note: IPC per core depends on how many HW threads per core (nHT) are running:

$$IPC_{core} = IPC_{thread} * nHT$$

$$CPI_{core} = CPI_{thread} / nHT$$

Vtune always displays CPI_{thread}

This is why all scaling graphs should be in terms of cores, not threads, and should show the number of threads per core used.

Analyzing CPI

There are two approaches to analyzing CPI:

1. Top-down analysis. This attempts to break the instruction time into categories reflecting the hardware pipeline. Vtune supports this with `-collect general-exploration` .
2. Looking for typical contributors to CPI by inspection and by computing specific metrics.

In HPC, memory stalls are usually the biggest contributor to CPI. Here we look specifically at L2 input bandwidth for KNL and compare to peak values (~380 GB/sec in flat mode for 7250 68-core part).

```
amplxe-cl -collect-with runsa -knob event-config=\
CPU_CLK_UNHALTED.REF_TSC:sa=13000000,CPU_CLK_UNHALTED.THREAD:sa=13000000,\
INST_RETIRED.ANY:sa=13000000,L2_REQUESTS.MISS,L2_PREFETCHER.ALLOC_XQ
```

L2 BW Vtune View

Module / Function / Call Stack	Hardware	
	INST RETIRED.ANY	CPU CLK UNHALTED.THREAD
▼ miniFE.x	473,252,000,000	692,354,000,000
▶ miniFE::cg_solve<miniFE::CSRMatrix<double>	461,318,000,000	630,448,000,000
▶ miniFE::daxpby<miniFE::Vector<double, int,	7,839,000,000	39,351,000,000
▶ miniFE::daxpby<miniFE::Vector<double, int,	3,549,000,000	21,099,000,000

(cont)

are Event Count by Hardware Event Type		
CPU CLK UNHALTED.REF TSC ▼	L2 REQUESTS.MISS	L2 PREFETCHER.ALLOC XQ
642,876,000,000	2,336,235,043	30,399,727,832
584,493,000,000	1,896,828,452	27,135,899,380
37,570,000,000	273,404,101	2,152,050,633
19,539,000,000	127,001,905	1,075,175,257

(cont)

Paste it into a spreadsheet...

L2 Input Bandwidth

Event or metric	Loop		
	matvec	daxbpy #1	daxbpy #2
INST_RETIRED.ANY	461,318,000,000	7,839,000,000	3,549,000,000
CPU_CLK_UNHALTED.THREAD	630,448,000,000	39,351,000,000	21,099,000,000
CPU_CLK_UNHALTED.REF_TSC	584,493,000,000	37,570,000,000	19,539,000,000
L2_REQUESTS.MISS	1,896,828,452	273,404,101	127,001,905
L2_PREFETCHER.ALLOC_XQ	27,135,899,380	2,152,050,633	1,075,175,257
cpi	1.367	5.020	5.945
L2 lines in	29,032,727,832	2,425,454,734	1,202,177,162
Elapsed cycles	9,132,703,125	587,031,250	305,296,875
bytes/cycle	203.5	264.4	252.0
GB/sec	264.5	343.8	327.6

L2 lines in = demand misses plus hardware prefetches

1 line = 64 bytes

Elapsed cycles = reference clocks / active threads

Multiply bytes/cycle by 1.3GHz to get GB/sec

Last two loops are close to peak bandwidth
Let's look at the assembly

matvec

```

543         for(MINIFE_GLOBAL_ORDINAL i = row_start; i < row_end; ++i) {
544             sum += Acoefs[i] * xcoefs[Acols[i]];
545         }

```

0x414796		Block 271:
0x414796	236	vmovaps %zmm0, %zmm8
0x41479c	236	vpcmpgtd %zmm4, %zmm3, %k3{%k1}
0x4147a2	236	vpadd %ymm1, %ymm4, %ymm4
0x4147a6	236	vmovdqu32z (%rax,%rcx,1), %zmm7{%k3}{z}
0x4147ad	236	kmovw %k3, %k2
0x4147b1	236	vmovupd (%r11,%rsi,8), %zmm9{%k3}{z}
0x4147b8	236	add \$0x8, %rsi
0x4147bc	236	add \$0x20, %rax
0x4147c0	236	vgatherdpdz (%r12,%ymm7,8), %k2, %zmm8
0x4147c7	236	vfmadd231pd %zmm9, %zmm8, %zmm6{%k3}
0x4147cd	236	cmp %rdx, %rsi
0x4147d0	236	jb 0x414796 <Block 271>

Indirection resulting in vgatherdpdz limits bandwidth

daxpby

220	for(int i = 0; i < n; ++i) {
221	ycoefs[i] += alpha * xcoefs[i];
222	}

0x418188		Block 8:
0x418188	221	vmovupsz (%r14,%r15,8), %zmm2
0x41818f	221	vfmadd213pdz (%r8,%r15,8), %zmm0, %zmm2
0x418196	221	vmovupdz %zmm2, (%r8,%r15,8)
0x41819d	218	add \$0x8, %r15
0x4181a1	218	cmp %rdx, %r15
0x4181a4	218	<u>jb 0x418188 <Block 8></u>

contiguous memory streams
result in near peak
bandwidth

Other Metrics

Use and understand the vtune hardware collection types. They are easy to use and have many valuable metrics and displays. You can learn a lot about the hardware events and top-down cycle breakdown. Use the tooltips.

Be very aware of overheads and quantization errors due to multiplexing (only two hardware events are available so not all events run all the time with the complex collection types). Watch for large increases in runtime when using vtune.

Don't be afraid to roll your own analysis. You can directly sample 3 fixed events plus two programmable event. Adjust the sample after value to avoid undue overhead.

When am I done?

Achieve high percentage of bandwidth

- Beware of BW bound code that doesn't need to be (e.g., fails to optimize for reuse in L2 cache)
- Lack of vectorization, lack of streaming stores, missing SW prefetches may limit BW

Achieve high IPC/low CPI

- BW bound code will have lower IPC
- Code may be BW bound from L1 or L2 cache as well as from memory.
- Micro-architectural decisions may limit IPC (e.g., some VPU instructions issue on only one port)
- Real dependences may limit IPC (one instruction needs to wait for another's result)
- Code may be front-end bound (instruction fetch and decode, branch prediction)

Ultimately you need excellent understanding of your algorithm's BW requirements and careful inspection of generated code to understand and eliminate micro-architectural bottlenecks.

References

Vtune reference:

- https://software.intel.com/en-us/amplifier_help_linux

Hardware counters:

- <https://software.intel.com/en-us/articles/intel-sdm> (Volume 3B Section 19.4)
- <https://software.intel.com/en-us/articles/intel-xeon-phi-x200-family-processor-performance-monitoring-reference-manual>

Recommended collection types:

```
advanced-hotspots, general-exploration, memory-access, hpc-performance  
collect-with runsa -knob event-config=hw-event-list  
sep -el to get the hw-event-list
```

Pause/resume:

```
#include <ittnotify.h>, use ittnotify  
-I/opt/intel/vtune_amplifier_xe/include -L/opt/intel/vtune_amplifier_xe/lib64 -littnotify  
__itt_resume()/__itt_pause()/-start-paused
```

BACKUP

Optimization example: CCS-QCD

Recompilation alone is
insufficient

* <https://conference.ippp.dur.ac.uk/event/470/session/14/contribution/44>

QCD code written by Dr. Ken-Ichi Ishikawa, Hiroshima University

Highly optimized* for Intel® Xeon Phi™ gen 1 (formerly known as KNC) on
Tsukuba COMA, tuned for gen 2 for Oakforest-PACS

Optimizations: MCDRAM, tiling for cache, manual prefetching, intrinsics, aligned
memory allocation, cooperative threading

Optimization	Dslash operator GF/sec	Full Solver GF/sec
DDR only	126	96
MCDRAM, no SW prefetch	393	326
MCDRAM, SW prefetch	542	424

24³x96 problem size
68 cores, 1.4GHz
All values in
Gflops/Second
nohz_full boot

Optimizations for Intel® Xeon Phi™ processor

MCDRAM for bandwidth-bound problems

Latency hiding:

- OOO provides some latency hiding (72 uops deep)
- HT (multiple threads per core); 2-level cooperative threading for many problems
- Manual prefetching for irregular or semi-regular access patterns (e.g. indirection, tiled loop nests)

Vectorization:

- Instruction selection: AVX-512 ERI for faster exp, recip; limited precision compiler flags; #pragma nontemporal for non-temporal stores
- AVX-512 CDI for various idioms using vector scatter

Performance analysis: use HW events

Classes of hardware performance events

Uncore

- MCDRAM and DDR reads and writes
- MCDRAM-as-cache HIT, MISS

Core/Tile

- Retired loads and stores
- Retired L1, L2, uTLB, and dTLB misses
- Instructions, reference and thread cycles (architectural PMU)
- Branch prediction, I\$ and iTLB misses
- Front-end stall cycles (instruction decode)
- Retired packed and scalar SIMD operations (proxy for vectorization quality)
- L2 cache line traffic

Module-level HW profile

Advanced Hotspots Hardware Issues viewpoint ([change](#)) ?

Collection Log Analysis Target Analysis Type Summary Bottom-up Top-down Tre

Grouping: Module / Function / Call Stack

Module / Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▶ ccs_qcd_solver_bench	473,662,000,000	191,632,000,000	2.472
▶ libiomp5.so	286,510,000,000	86,506,000,000	3.312
▶ vmlinux	896,000,000	126,000,000	7.111
▶ ld-2.17.so	364,000,000	42,000,000	8.667
▶ igb	70,000,000	14,000,000	5.000

Lots of spin time

Low kernel time (good)

Seems high;
1.0 is min (best possible) for
2 threads/core

Vtune tips

Vtune is a quick way to get most of the data you need.

Limit your vtune collection time to 20s or so on one node during program steady-state: either use `__itt_pause/__itt_resume` or collect for 20s while program is running:

```
amplxe-cl -collect advanced-hotspots -analyze-system -d 20
```

`-collect general-exploration` and `-collect memory-access` are both useful for seeing how well the hardware is being used.

Vtune has lower overhead than perf record, especially on many small cores. Finalization can be slow; limiting collection time helps a lot.

Non-sampling approaches

Instrument your code

- Collect core counters for all threads
 - Use `sep -count` and `RDPMC`
 - `Jevents`
 - Add them all together
- Collect uncore events separately with `linux perf`
- Consider normalizing per something meaningful to your code, e.g., per solver iteration

Use `emon` to collect counters over time

- Lower overhead than sampling
- Correlation to application is over time, not to source code

Fine-grained analysis example: CCS-QCD

Instrument only the Dslash operator
Use RDPMC to read the counters

Metric	Formula	Value	
L1 hit rate	$\text{MEM_UOPS_RETIRED.L1_MISS_LOADS} / \text{MEM_UOPS_RETIRED.ALL_LOADS}$	96.85%	
L2 hit rate	$\text{MEM_UOPS_RETIRED.L2_HIT_LOADS} / (\text{MEM_UOPS_RETIRED.L2_HIT_LOADS} + \text{MEM_UOPS_RETIRED.L2_MISS_LOADS})$	96.72%	
IPC/core	$2 * \text{INST_RETIRED.ANY} / \text{CPU_CLK_UNHALTED.THREAD}$	1.16	With 2 threads/core
MCDRAM Read BW	*Memory_reads/Time	317GB/Sec	Rule of thumb: 360 GB/sec aggregate max L2 input BW Implies this code is bandwidth bound
Per-core L2 input BW	$64 * (\text{L2_PREFETCHER.ALLOC_XQ} + \text{L2_REQUESTS.MISS}) / \text{Time}$	341GB/Sec	

* Elapsed time can be computed several ways, e.g., $\text{CPU_CLK_UNHALTED.REF_TSC} / \text{\#threads}$, assuming threads are all active during the measured region.

Legal Disclaimers

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer. No computer system can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit <http://www.intel.com/performance>.

Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Statements in this document that refer to Intel's plans and expectations for the quarter, the year, and the future, are forward-looking statements that involve a number of risks and uncertainties. A detailed discussion of the factors that could affect Intel's results and plans is included in Intel's SEC filings, including the annual report on Form 10-K.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Intel, the Intel logo, Atom, Xeon, Xeon Phi, 3D Xpoint, Iris Pro and others are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

Legal Disclaimers

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 SSE3 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

