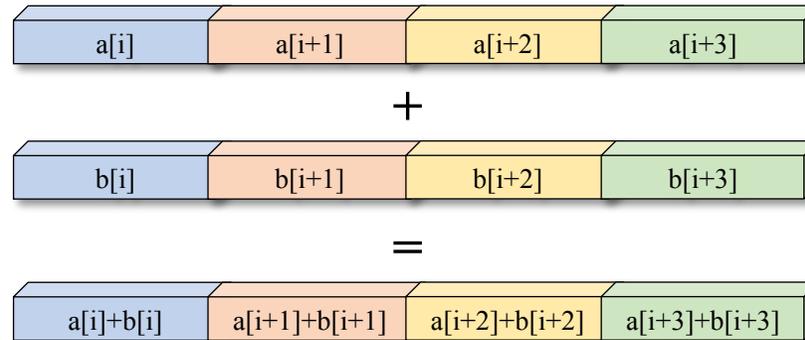


Introduction to Performance Tuning & Optimization Tools



Ian A. Cosden, Ph.D.

Manager, HPC Software Engineering and Performance Tuning
Research Computing, OIT, Princeton University

icosden@princeton.edu

CoDaS-HEP Summer School
July 26, 2018

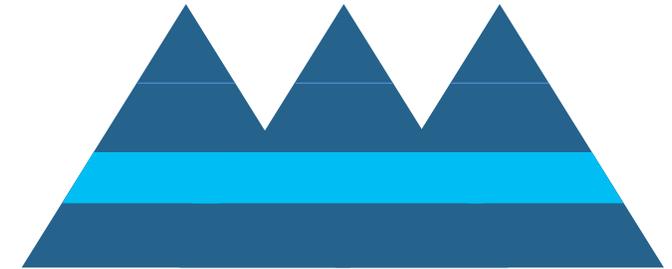
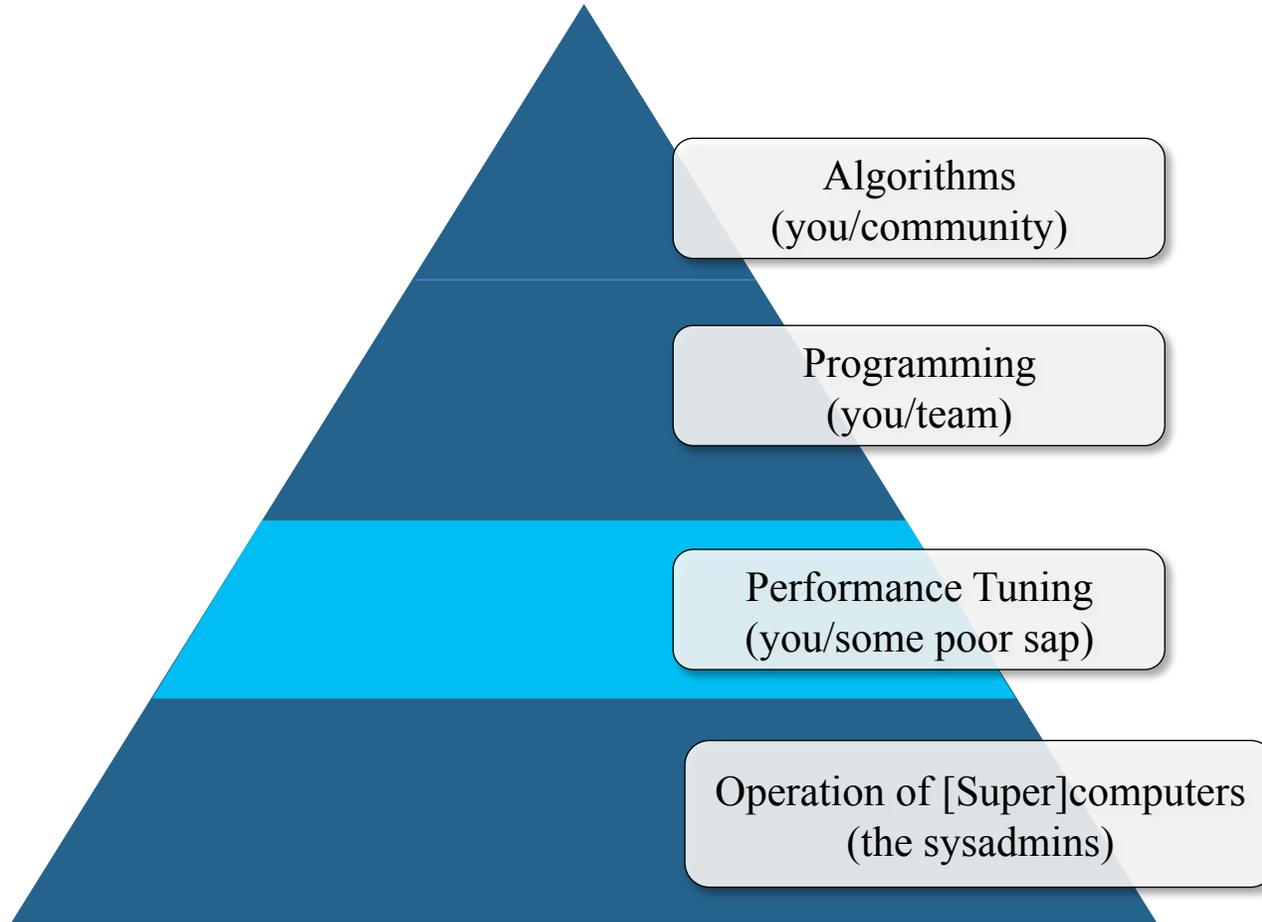
Outline

- What is Performance Tuning?
- Performance Tuning Workflow
- Software Tools
 - Hotspots: Intel VTune
 - OpenMP Scaling: Intel VTune
 - Vectorization: Intel Advisor
- Hands-on VTune Exercise
- Hardware Counters: VTune

What is Performance?

- “Performance is the degree to which a computing system meets expectations of the persons involved in it.” Doherty (1970)
 - Is it doing what I want?
- “Performance...is the effectiveness with which the resources of the host computer system are utilized toward meeting the objectives of the software system.” Graham (1973)
 - Is it utilizing the resources well?

What is Tuning?



Applicable to all domains

Where does Tuning fit?

- Performance should not be ignored during development
 - Algorithm selection
 - Vectorization and Parallelization
 - Data structures
 - Hardware awareness
- *Tuning* typically happens near the end
 - What if you invest time tuning code you never use?
- Principles and techniques applicable at any stage
- We'll assume we are tuning an underperforming correct code

Tuning Tradeoff

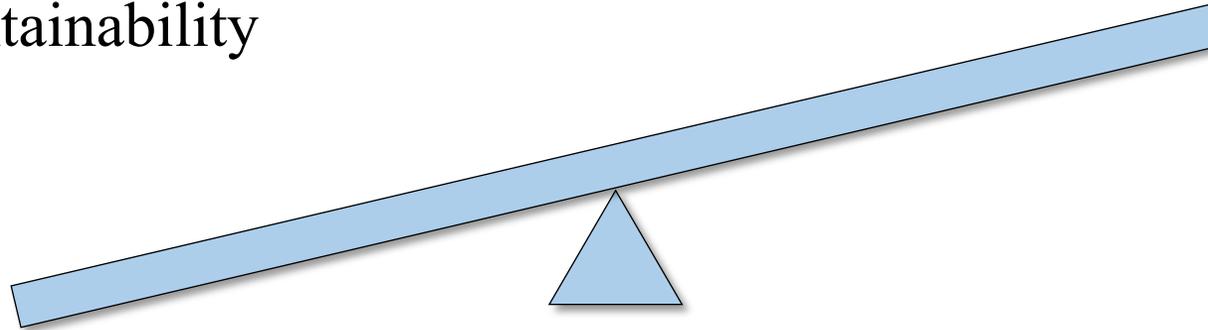
Readability

Simplicity

Maintainability

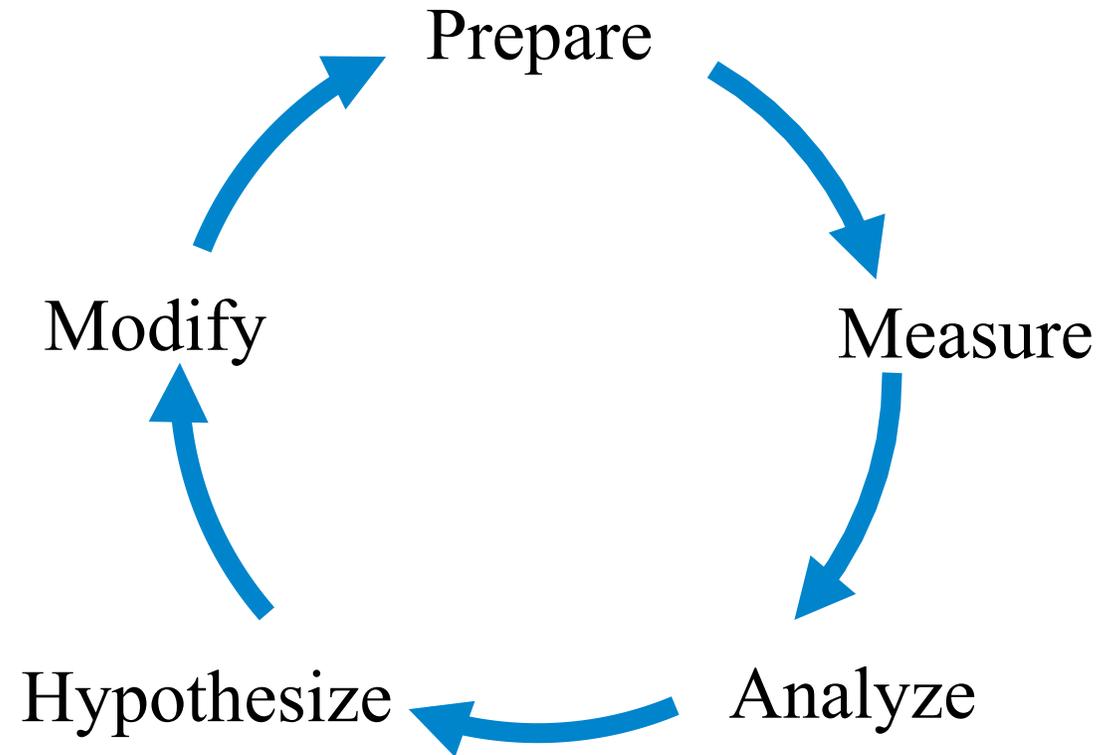
Time

Performance



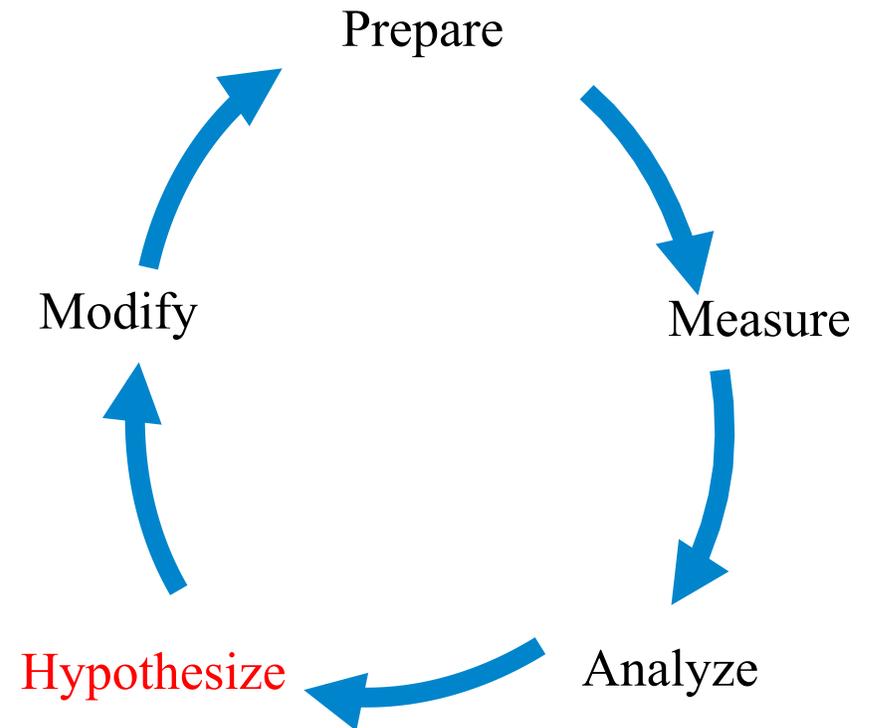
- Set goals - *Faster*
 - It could always be faster.
 - How much faster? Be specific.
 - Set an upper limit on your time investment

Performance Tuning Workflow



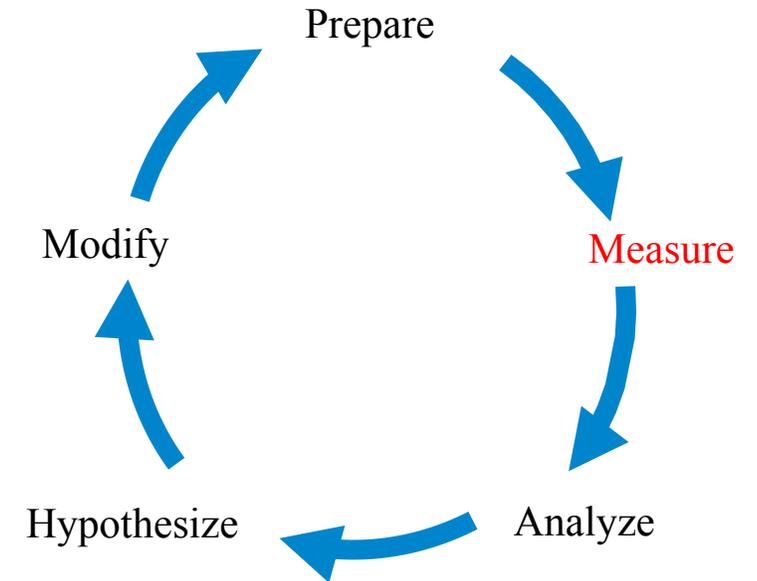
Hypothesis

- Why is my code slow?
 - CPU bound
 - Memory bound
 - I/O bound
 - Network bound
 - Unbalanced Workload (Parallel)
- What is the best I can expect?
 - CPU
 - Memory/Cache
 - I/O
 - Network
 - Parallel Scaling



Measure

- Need some kind of monitor
- Direct Measurement a.k.a. *Instrumentation*
 - Event is what we want to measure
 - Accurate
 - High overhead for frequent events
 - Ex. Tracing
- Indirect Measurement a.k.a. *Sampling*
 - Record system state at regular interval
 - Typically low overhead
 - Not every event recorded
 - Ex. Profiling
- Measurement can influence result



Measuring

- Measuring the performance of your code
 - You can't fix what you can't see
 - Find the “hotspots”
 - How much time is spent in each function
 - **Not always where you think it is**
 - Identify regions to optimize/parallelize
 - Hardware Performance
 - Vectorization, cache misses, branch misprediction, etc.
- Do it yourself
 - Put time calls around loops/functions
 - Only works if:
 - Done carefully during development
 - On a small code base
- Use a tool

Easy Measures - Linux

- Total Runtime (“time”)
 - Likely not enough
 - Critically important number

```
$ time ./app-ICC 65536 > output.log

real    0m47.560s
user    0m47.441s
sys     0m0.000s
```

Un-tuned

```
$ time ./app-ICC 65536 > output.log

real    0m0.643s
user    0m19.810s
sys     0m0.044s
```

Tuned

- CPU usage (“top”)
 - $N * 100\%$ cpu = N threads running full-tilt
 - $100\% \neq 100\%$ peak theoretical performance
 - Useful for threading efficiency only

```
top - 08:32:06 up 8 days, 19:47, 7 users, load average: 0.72, 0.16, 0.09
Tasks: 771 total, 2 running, 769 sleeping, 0 stopped, 0 zombie
%Cpu(s): 42.8 us, 0.1 sy, 0.0 ni, 57.1 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 26372923+total, 25595838+free, 2061328 used, 5709524 buff/cache
KiB Swap: 8191996 total, 8191996 free, 0 used. 26107923+avail Mem
```

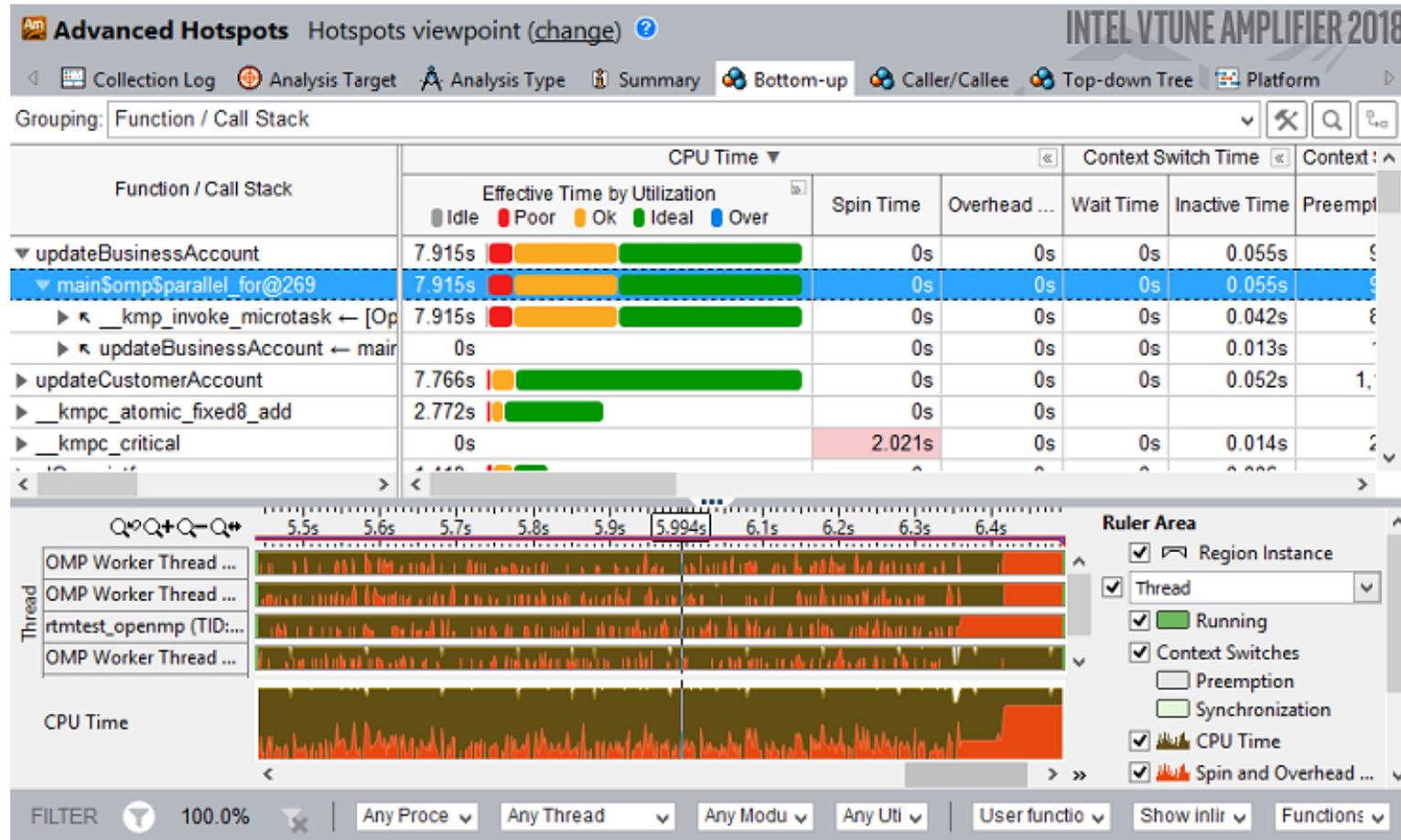
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3863	icosden	20	0	857096	5788	1620	R	1201	0.0	1:04.35	app-ICC

Performance Tools

- Many free and commercial products
- Site specific – but most places have the major products
- Intel Parallel Studio
 - VTune Amplifier XE
 - Advisor
- Many others: Allinea Forge (MAP), Tau, Intel Trace Analyzer and Collector, HPCToolkit, gprof, perf, gperftools, ...

Intel VTune

- Intel VTune Amplifier XE
 - Commercial Profiler
 - Extraordinarily powerful (and complicated)
 - Nice GUI
- Source code profiling
- Shared memory only
 - Serial
 - OpenMP
 - MPI on single node



Intel Advisor

Where should I add vectorization and/or threading parallelism? Intel Advisor XE 2016

Summary Survey Report Refinement Reports Annotation Report Suitability Report

Elapsed time: 54.44s Vectorized Not Vectorized FILTER: All Modules All Sources

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Trip Counts	Loop Type	Why No Vectorization?	Vectorized Loops	
							Vecto...	Efficiency
[loop at stl_algo.h:4740 in std::tr ...]		0.170s	0.170s		Scalar	non-vectorizable loop ins ...		
[loop at loopstl.cpp:2449 in s234_]	2 Ineffective peeled/rem ..	0.170s	0.170s	12; 4	Collapse	Collapse	AVX	~100%
[loop at loopstl.cpp:2449 in s ...]		0.150s	0.150s	12	Vectorized (B		AVX	
[loop at loopstl.cpp:2449 in s ...]		0.020s	0.020s	4	Remainder			
[loop at loopstl.cpp:7900 in vas_]		0.170s	0.170s	500	Scalar	vectorization possible but ...		
[loop at loopstl.cpp:3509 in s2 ...]	1 High vector register ...	0.160s	0.160s	12	Expand	Expand	AVX	~69%
[loop at loopstl.cpp:3891 in s279_]	2 Ineffective peeled/rem ..	0.150s	0.150s	125; 4	Expand	Expand	AVX	~96%
[loop at loopstl.cpp:6249 in s414_]		0.150s	0.150s	12	Expand	Expand	AVX	~100%
[loop at stl_numeric.h:247 in std ...]	1 Assumed dependency...	0.150s	0.150s	49	Scalar	vector dependence preve ...		

- Intel Advisor
 - Vectorization (and threading) advisor
- Identifies loops to target for vectorization
 - Provides efficiency statistics and tips for improvement
- Roofline Analysis
 - In many ways it's a simple profiler + GUI vec-report

Using Profilers

- Strength of all profiling tools revolves around ability to trace performance back to source code
 - Need to include debug symbols in executable
 - -g flag
- Use release-build optimization flags
 - Ex:
 - O3, -xhost (Intel)
 - Don't waste time optimizing code the compiler can do automatically!
- Sometimes the compiler will optimize out useful regions
 - Recommendations:
 - debug inline-debug-info, -debug full (Intel)
- For difficult problems use more than one profiler

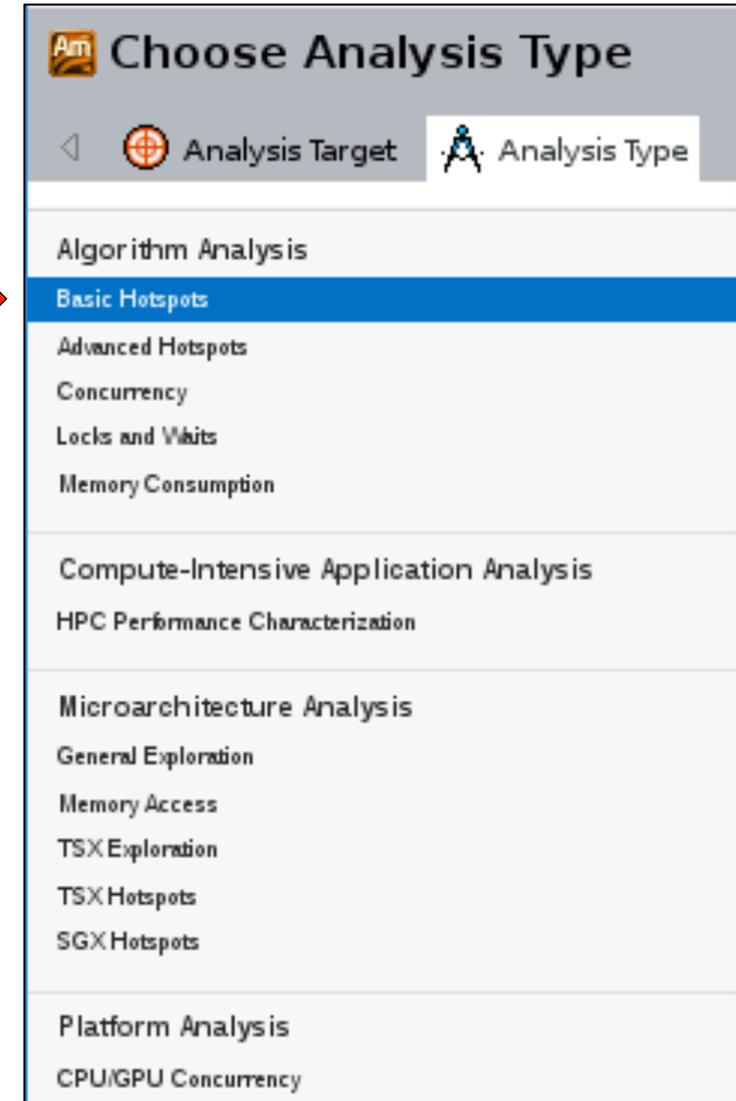
Examples

A handful of real(ish) codes to demonstrate profiling:

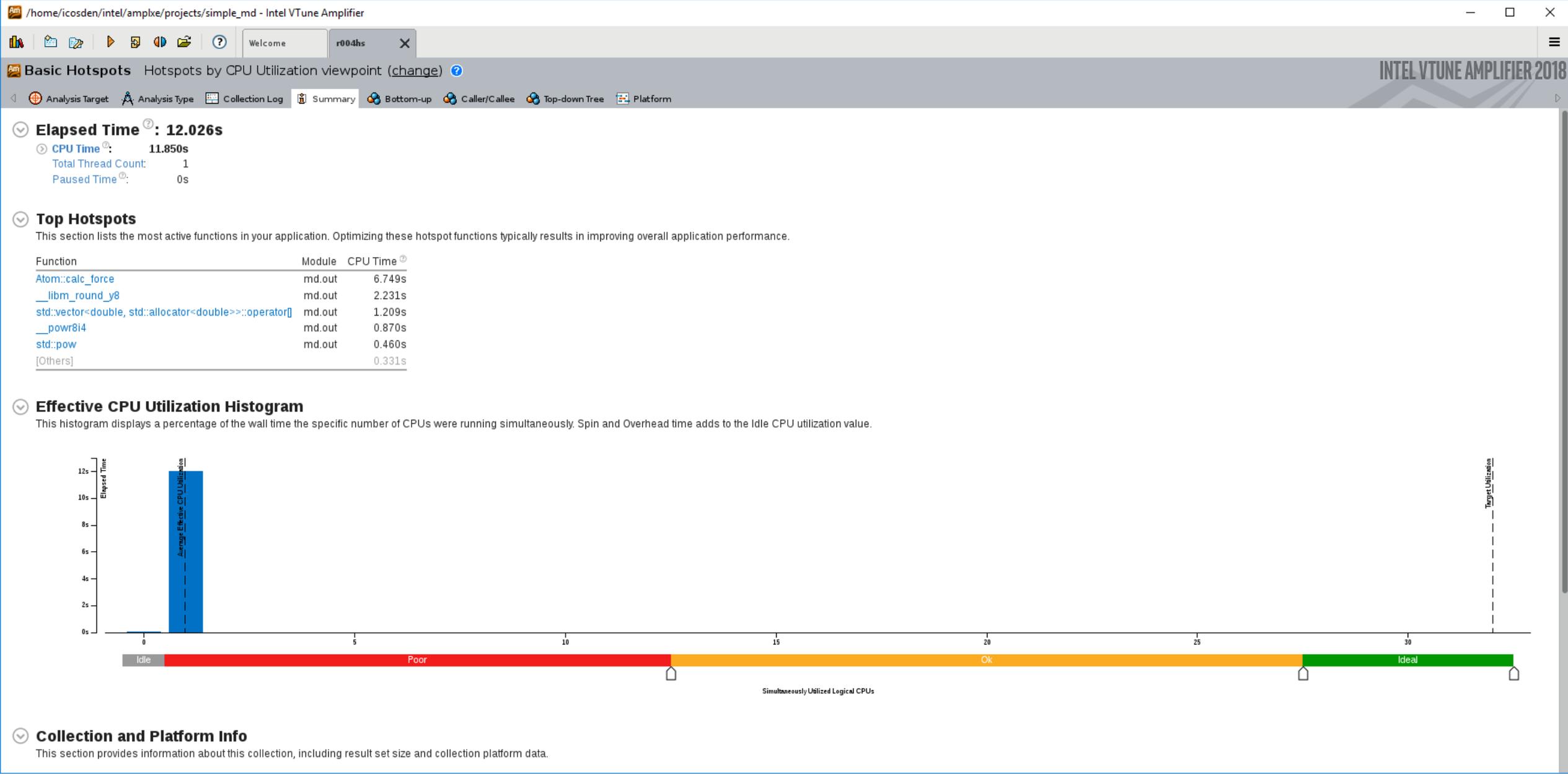
1. Hotspots
 - VTune
2. OpenMP Scaling
 - VTune
3. Vectorization
 - Advisor
4. Cache performance & Hardware Metrics
 - VTune

VTune Example

- Code compiled with -g to enable source code profiling
- Example serial C++ code with:
 - No multithreading
 - No MPI
 - Some I/O
- Use “Basic Hotspots” analysis
 - This will be subject of hands-on exercise later



VTune – Profile Result



Bottom-up Tab (Functions)

Intel VTune Amplifier 2018 interface showing the Bottom-up tab for CPU Utilization. A red arrow points to the 'Bottom-up' tab in the navigation bar.

Grouping: Function / Call Stack

Function / Call Stack	CPU	Module	Function (Full)	Source File	Start Ad
Atom::calc_force	6.749s	md.out	Atom::calc_force(void)	atom.cpp	0x401dae
__libm_round_y8	2.231s	md.out	__libm_round_y8		0x40a890
std::vector<double, std::allocator<double>>::operator[]	1.209s	md.out	std::vector<double, std::allocator<double>>::operator...	std_vector.h	0x402f78
__powr8i4	0.870s	md.out	__powr8i4		0x408240
std::pow	0.460s	md.out	std::pow(double, int)	cmath	0x404718
_IO_fprintf	0.231s	libc.so.6	_IO_fprintf		0x531a0
round	0.070s	md.out	round		0x407960
__fopen_internal	0.020s	libc.so.6	__fopen_internal		0x6e9c0
_IO_fclose	0.010s	libc.so.6	_IO_fclose		0x6de60

Viewing 1 of 2 selected stack(s)
99.7% (6.729s of 6.749s)

md.out!Atom::calc_force - atom.cpp
md.out!Md_sim::run+0xd1 - md_sim.cpp:80
md.out!main+0x97 - main.cpp:12
libc.so.6!__libc_start_main+0xf4 - [unknown so...
md.out!_start+0x28 - [unknown source file]

Thread: md.out (TID: 167468)

CPU Utilization

Thread: Thread

- Running
- CPU Time
- Spin and Ov...
- CPU Sample

CPU Utilization

- CPU Time
- Spin and Ov...

FILTER 100.0% | Process: Any Process | Thread: Any Thread | Module: Any Module | Utilization: Any Utilization | Call Stack Mode: User functions + 1 | Inline Mode: Show inline function | Loop Mode: Functions only

Bottom-up (Loops)

Intel VTune Amplifier 2018 interface showing a bottom-up view of CPU hotspots. The main table displays the following data:

Function / Call Stack	CPU Time	Module	Function (Full)	Source File	Start Address
[Loop at line 80 in Atom::calc_force]	11.329s	md.out	[Loop at line 80 in Atom::calc_force(v...	atom.cpp	0x401f78
[Loop at line 228 in Md_sim::write]	0.231s	md.out	[Loop at line 228 in Md_sim::write(voi...	md_sim.cpp	0x40740d
[Loop@0x40829c in __powr8i4]	0.230s	md.out	[Loop@0x40829c in __powr8i4]		0x40829c
[Loop at line 76 in Md_sim::run]	0.030s	md.out	[Loop at line 76 in Md_sim::run(void)]	md_sim.cpp	0x405c34
[Loop at line 114 in Md_sim::vel_B]	0.020s	md.out	[Loop at line 114 in Md_sim::vel_B(v...	md_sim.cpp	0x407063
[Loop at line 96 in Md_sim::vel_A]	0.010s	md.out	[Loop at line 96 in Md_sim::vel_A(void)]	md_sim.cpp	0x406bb8

The interface also shows a 'Thread' view at the bottom left and a 'CPU Utilization' view at the bottom right. A red arrow points to the 'Loops only' filter in the bottom right corner.

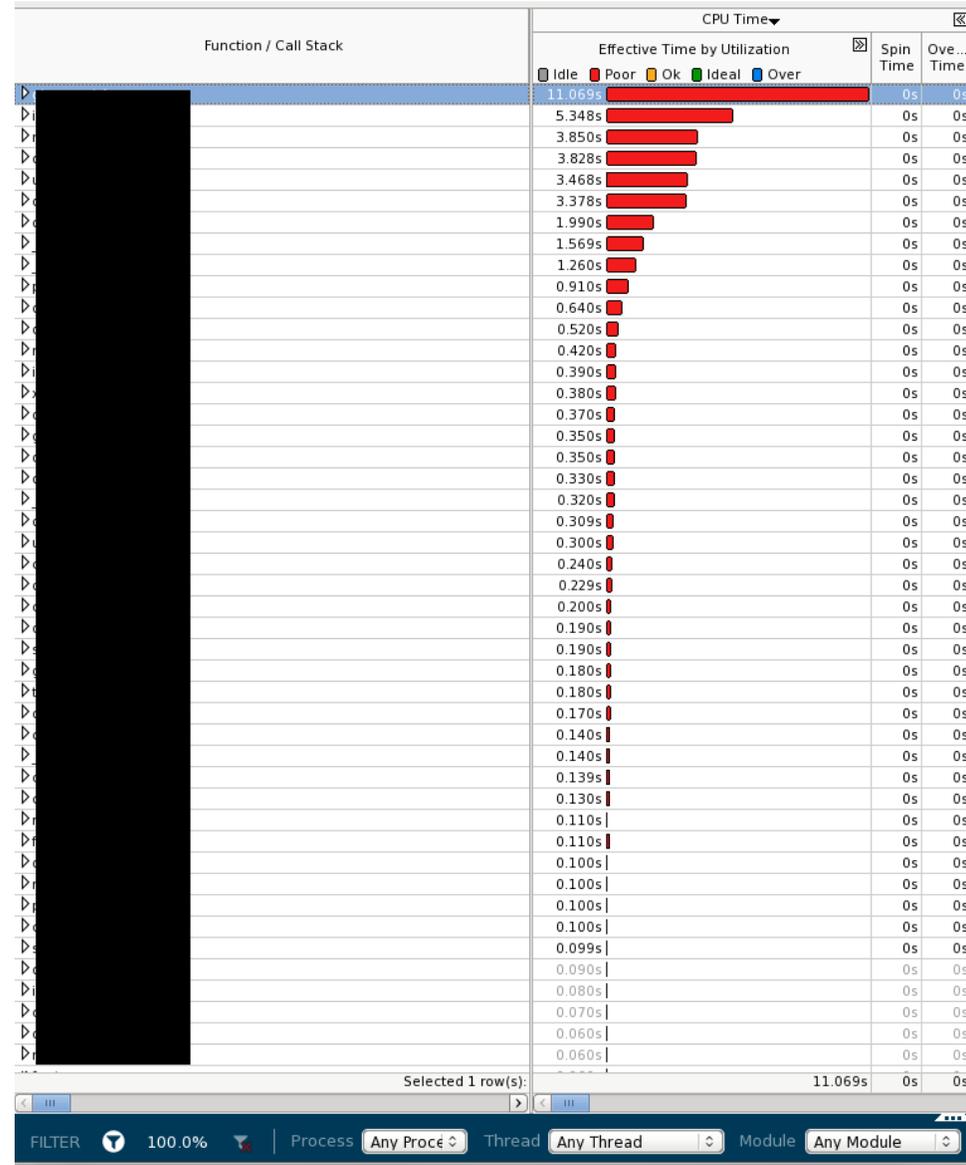
Source Code Hotspots (line-by-line)

The screenshot shows the Intel VTune Amplifier 2018 interface. The main window displays a table of CPU time utilization for various lines of code in the file 'atom.cpp'. A red arrow points to the 'atom.cpp' tab. The table is organized into columns for 'CPU Time: Total' and 'CPU Time: Self', each with sub-columns for 'Effective Time by Utilization' (Idle, Poor, Ok, Ideal, Over) and 'Spin Time' and 'Over Time'. The table shows that line 103 is the most significant hotspot, with a CPU time of 1.919s and 16.2% utilization. Other hotspots include lines 87, 88, 109, 110, 111, 115, 116, 119, 120, and 121.

So. Lin.	Source	CPU Time: Total			CPU Time: Self		
		Effective Time by Utilization	Spin Time	Over Time	Effective Time by Utilization	Spin Time	Over Time
77	for(int i=0; i<(natoms-1); i++) {						
78							
79							
80	for(int j=i+1; j<natoms; j++) {	0.9%	0.0%	0.0%	0.111s	0s	0s
81							
82	//Calculate distance between atoms						
83	Xr = rx[i] - rx[j];	4.9%	0.0%	0.0%	0.581s	0s	0s
84	Yr = ry[i] - ry[j];	3.6%	0.0%	0.0%	0.429s	0s	0s
85	Zr = rz[i] - rz[j];	3.5%	0.0%	0.0%	0.410s	0s	0s
86	Xr = Xr - box_x*round(Xr/box_x); //periodic boundary condition	12.8%	0.0%	0.0%	1.519s	0s	0s
87	Yr = Yr - box_y*round(Yr/box_y);	16.0%	0.0%	0.0%	1.891s	0s	0s
88	Zr = Zr - box_z*round(Zr/box_z);	14.3%	0.0%	0.0%	1.699s	0s	0s
89							
90	Xr2 = Xr*Xr; //square of distance	0.9%	0.0%	0.0%	0.110s	0s	0s
91	Yr2 = Yr*Yr;	1.4%	0.0%	0.0%	0.170s	0s	0s
92	Zr2 = Zr*Zr;	0.6%	0.0%	0.0%	0.070s	0s	0s
93							
94	RijSQ = Xr2 + Yr2 + Zr2;	3.0%	0.0%	0.0%	0.350s	0s	0s
95							
96	if(RijSQ<=RadiusSQ) {	2.1%	0.0%	0.0%	0.250s	0s	0s
97							
98	r2i = 1.0/RijSQ; //Intermediate Calcs for the LJ (6-12) potential	2.4%	0.0%	0.0%	0.290s	0s	0s
99	r6i = r2i*r2i*r2i;	1.9%	0.0%	0.0%	0.220s	0s	0s
100							
101	//Fij = 48.0*r2i*r6i*(r6i-0.5); //LJ Force in reduced units						
102							
103	force = 48.0*pow(1.0/RijSQ,6)-24*pow(1.0/RijSQ,3);	16.2%	0.0%	0.0%	1.919s	0s	0s
104	Fij = force*r2i;	1.4%	0.0%	0.0%	0.170s	0s	0s
105							
106	PE=PE+r6i*(r6i-1.0);	0.9%	0.0%	0.0%	0.111s	0s	0s
107							
108	//Component Forces						
109	Fxij = Xr*Fij;	0.8%	0.0%	0.0%	0.100s	0s	0s
110	Fyij = Yr*Fij;	0.4%	0.0%	0.0%	0.051s	0s	0s
111	Fzij = Zr*Fij;	0.7%	0.0%	0.0%	0.080s	0s	0s
112							
113	//Sum forces on atom i						
114	fx[i] +=Fxij;	0.8%	0.0%	0.0%	0.100s	0s	0s
115	fy[i] +=Fyij;	1.6%	0.0%	0.0%	0.189s	0s	0s
116	fz[i] +=Fzij;	0.8%	0.0%	0.0%	0.090s	0s	0s
117							
118	//Sum forces on atom j						
119	fx[j] -= Fxij;	1.1%	0.0%	0.0%	0.130s	0s	0s
120	fy[j] -= Fyij;	0.7%	0.0%	0.0%	0.080s	0s	0s
121	fz[j] -= Fzij;	1.3%	0.0%	0.0%	0.149s	0s	0s
122							
123	p_PE -= Fxij*Xr-Fyij*Yr-Fzij*Zr;	0.5%	0.0%	0.0%	0.060s	0s	0s
124							
Selec...							

CPU Time
Viewing 1 of 2 selected stack(s)
99.6% (11.279s of 11.329s)
md.out[Loop at line 80 in Atom::calc_forc...
md.out[Loop at line 78 in Atom::calc_forc...
md.out[Loop at line 76 in Md_sim::run]+0...
md.out[Outside any loop] - [unknown sour...

Bottom-up – Real Code



Same Real Code – Top-down

Function Stack	CPU Time: Total				CPU Time: Self			
	Effective Time by Utilization			Spin Time	Overhead Time	Effective Time by Utilization		
	Idle	Poor	Ok			Idle	Poor	Ok
Total	100.0%			0.0%	0.0%			0s
_start	100.0%			0.0%	0.0%			0s
_libc_start_main	100.0%			0.0%	0.0%			0s
main	100.0%			0.0%	0.0%			0.010s
g	99.9%			0.0%	0.0%			0.010s
g	72.4%			0.0%	0.0%			0s
ts	70.9%			0.0%	0.0%			0s
g	56.2%			0.0%	0.0%			0s
g	56.2%			0.0%	0.0%			0.350s
g	39.3%			0.0%	0.0%			11.069s
g	5.7%			0.0%	0.0%			2.200s
g	4.1%			0.0%	0.0%			1.910s
g	1.9%			0.0%	0.0%			0.190s
g	1.4%			0.0%	0.0%			0.390s
g	0.8%			0.0%	0.0%			0.120s
g	0.6%			0.0%	0.0%			0.100s
g	0.3%			0.0%	0.0%			0.150s
g	0.3%			0.0%	0.0%			0.120s
g	0.1%			0.0%	0.0%			0.050s
g	0.1%			0.0%	0.0%			0.030s
g	0.0%			0.0%	0.0%			0s
g	0.0%			0.0%	0.0%			0.010s
g	8.4%			0.0%	0.0%			0.100s
g	3.9%			0.0%	0.0%			0.240s
g	3.6%			0.0%	0.0%			0.060s
g	0.1%			0.0%	0.0%			0s
g	0.0%			0.0%	0.0%			0s
g	0.0%			0.0%	0.0%			0s
g	0.0%			0.0%	0.0%			0.010s
g	8.4%			0.0%	0.0%			0s
g	5.6%			0.0%	0.0%			0.099s
g	0.7%			0.0%	0.0%			0.020s
g	0.0%			0.0%	0.0%			0s
g	0.0%			0.0%	0.0%			0.010s
g	1.3%			0.0%	0.0%			0s
g	0.2%			0.0%	0.0%			0.010s
g	0.1%			0.0%	0.0%			0s
g	0.0%			0.0%	0.0%			0s
g	0.0%			0.0%	0.0%			0.010s
g	25.2%			0.0%	0.0%			0s
g	24.9%			0.0%	0.0%			0s
g	0.3%			0.0%	0.0%			0s
g	1.0%			0.0%	0.0%			0s
g	0.6%			0.0%	0.0%			0s
g	0.6%			0.0%	0.0%			0.060s
g	0.1%			0.0%	0.0%			0s
Highlighted 1 row(s):	39.3%			0.0%	0.0%			11.069s

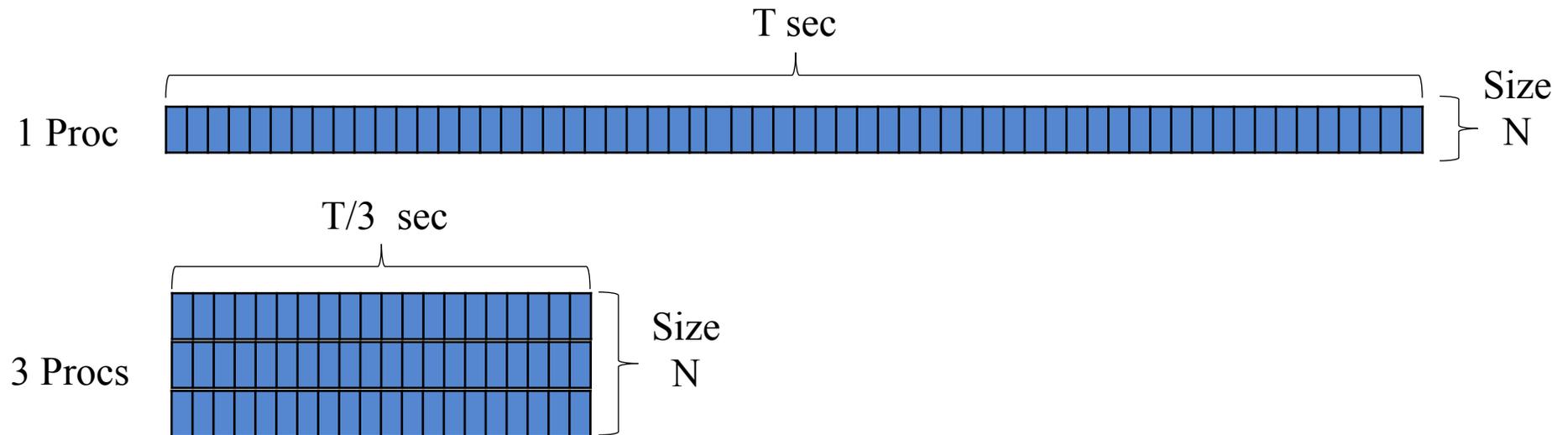
Examples

A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
 - VTune
2. OpenMP Scaling
 - VTune
3. Vectorization
 - Advisor
4. Cache performance & Hardware Metrics
 - VTune

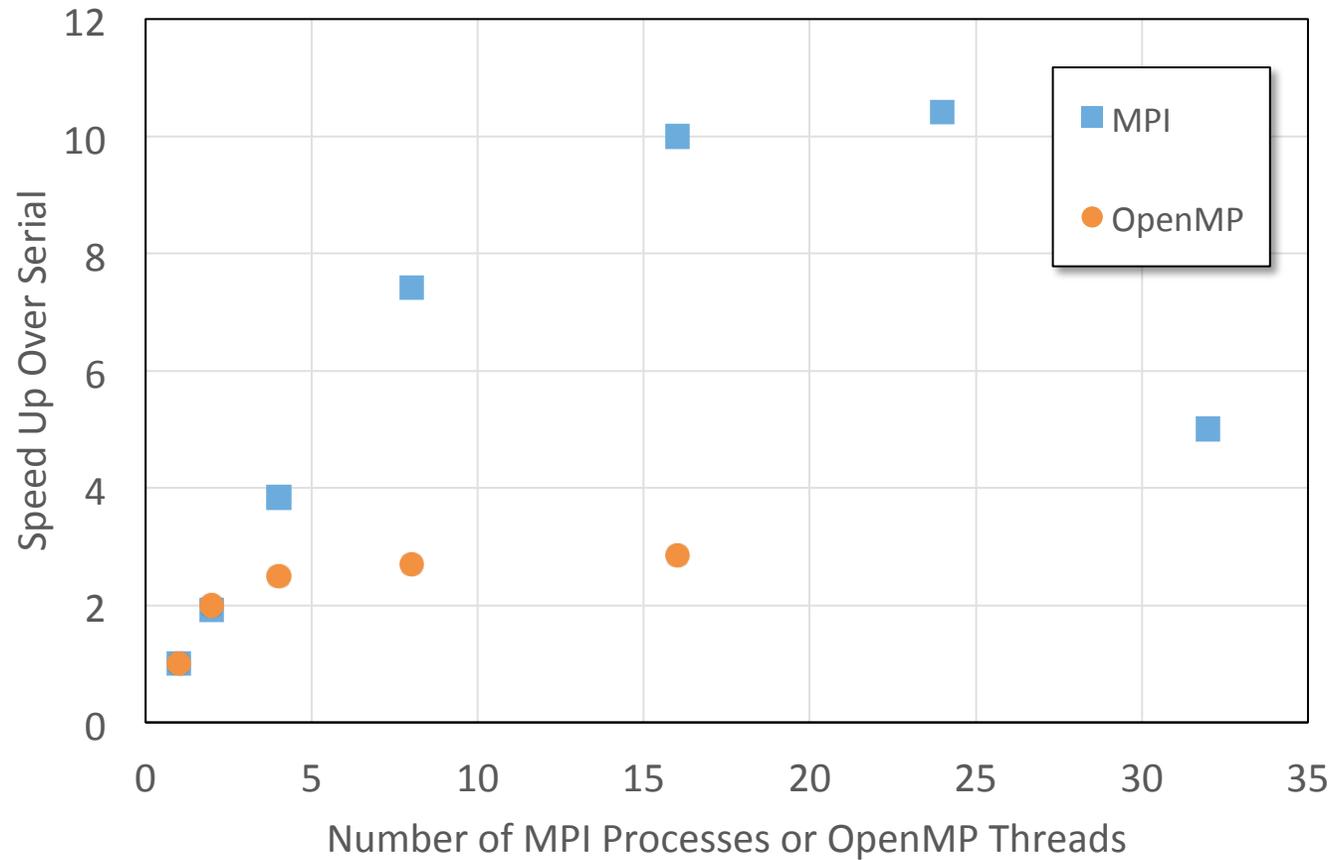
Strong Scaling

- Strong Scaling
 - Fixed problem size
 - Measure how solution time decreases with more processors



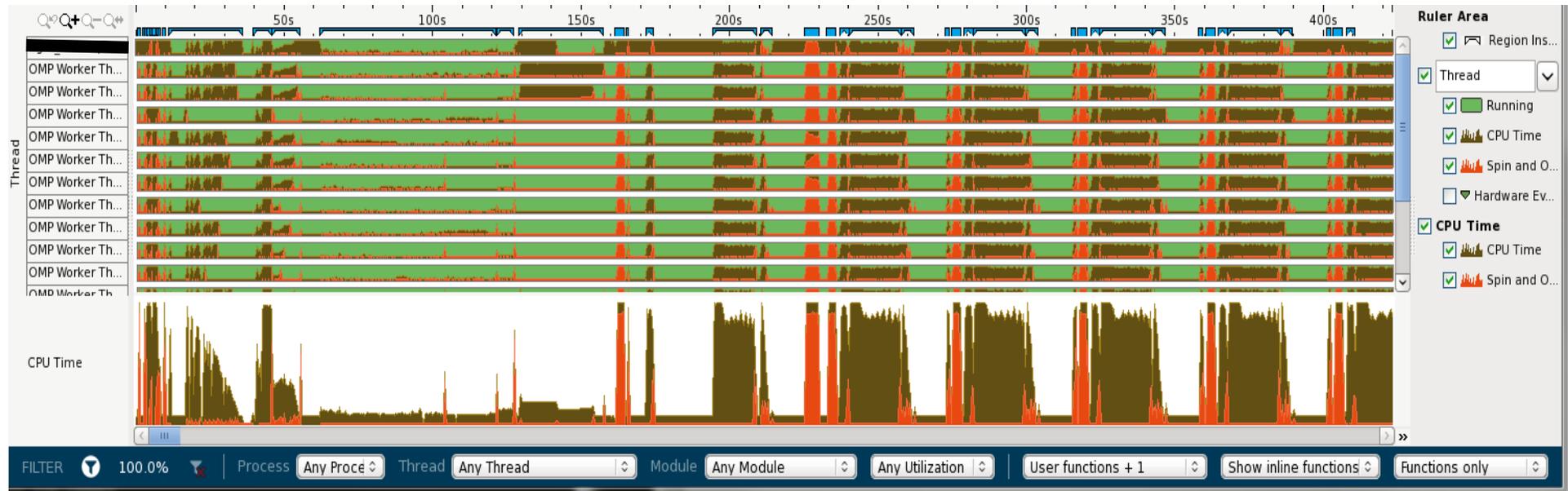
Scaling and Performance

Example Strong Scaling of MPI and OpenMP code



OpenMP Threading Efficiency

- Example code was seeing ~3x speed up with 16 threads
 - Why? How to fix?
- Zoom in on threading timeline in VTune:
 - Brown = good
 - Green (idle) and Orange (overhead) = bad



Examples

A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
 - VTune
2. OpenMP Scaling
 - VTune
3. Vectorization
 - Advisor
4. Cache performance & Hardware Metrics
 - VTune

Intel Advisor Example

- Example code compiled with intel compiler with flags: `-no-vec` and `-no-simd`
 - Purposefully turns *off* auto-vectorization
 - Useful with debugging or performance benchmarking only
 - Or faking poorly vectorized code
- Then built normally
 - `-g` (debugging symbols) and all release flags
- Ran on Broadwell node
 - AVX2 instruction set (FMA)
 - Two, 256-bit vector registers
 - Single precision (32 bit) floating point numbers: 8 per register possible

Poorly Vectorized - Advisor

Where should I add vectorization and/or threading parallelism?

Elapsed time: 10.19s Vectorized Not Vectorized FILTER: All Module All Sources Loop All Threads

Summary Survey Report Refinement Reports Annotation Report

Higher instruction set architecture (ISA) available
Consider recompiling your application using a higher ISA.

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops			Instruction Set
						Vec...	Gai...	VL ...	Traits
[loop in MoveParticles at nbody.cc:27]	1 Potential...	231.026s	231.026s	Scalar	auto-vectorization is disabled with -no-vec flag				Square Roots
[loop in MoveParticles at nbody.cc:20]	2 Assume ...	0.010s	231.036s	Threaded (Op...	vector dependence prevents vectorization				
[loop in start_thread]		0.000s	222.980s	Scalar					
[loop in main at nbody.cc:101]	1 Assume ...	0.000s	8.056s	Scalar	vector dependence prevents vectorization				Divisions
[loop in [OpenMP worker at z_Linux_util.c:769]		0.000s	222.980s	Scalar					
[loop in __kmp_launch_thread at kmp_runtime...		0.000s	222.980s	Scalar					

Source Top Down Loop Analytics Loop Assembly Recommendations Compiler Diagnostic Details

File: nbody.cc:27 MoveParticles

Line	Source	To
21		
22	// Components of the gravity force on particle i	
23	float Fx = 0, Fy = 0, Fz = 0;	
24		
25	// Loop over particles that exert force: vectorization expected here	
26	#pragma vector aligned	
27	for (int i = 0; i < nParticles; i++) {	
28		
29	// Avoid singularity and interaction with self	
30	const float softening = 1e-20;	
31		
32	// Newton's law of universal gravity	
33	const float dx = particle.x[i] - particle.x[i];	
34	const float dy = particle.y[i] - particle.y[i];	
35	const float dz = particle.z[i] - particle.z[i];	
36		
37	const float drSquared = dx*dx + dy*dy + dz*dz + softening;	
38	const float drSqrt = sqrtf(drSquared);	
39	const float drSqrtRecip = 1.0f/drSqrt;	

Selected (Total Time):

Vectorized - Advisor

Where should I add vectorization and/or threading parallelism?

Elapsed time: 1.47s Vectorized Not Vectorized FILTER: All Module All Sources Loop: All Threads

Summary Survey Report Refinement Reports Annotation Report

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops				Instruction Set Analysis
						Vec...	Efficiency	Gai...	VL ...	
[loop in MoveParticles at nbody.cc:27]		32.896s	32.896s	Vectorized (B...		AVX2	~96%	7.71x	8	Traits
[loop in MoveParticles at nbody.cc:20]	1 Potential...	0.060s	32.956s	Threaded (Op...	inner loop was already vectorized					Extracts; FMA; Shu...
[loop in MoveParticles at nbody.cc:57]	2 Assume...	0.020s	0.020s	Threaded (Op...	vector dependence prevents vectorization					FMA
[loop in start_thread]		0.000s	31.616s	Scalar						
[loop in main at nbody.cc:101]	1 Assume...	0.000s	1.360s	Scalar	vector dependence prevents vectorization					Divisions
[loop in [OpenMP worker at z_Linux_u...		0.000s	31.616s	Scalar						
[loop in __kmp_launch_thread at kmp_...		0.000s	31.616s	Scalar						

Source Top Down Loop Analytics Loop Assembly Recommendations Compiler Diagnostic Details

File: nbody.cc:27 MoveParticles

Line	Source	Total Ti
17	void moveParticles(const int nParticles, ParticleArrays &particle, const float G)	
18	// Loop over particles that experience force	
19	#pragma omp parallel for	1.4
20	for (int i = 0; i < nParticles; i++) {	
21	// Components of the gravity force on particle i	
22	float Fx = 0, Fy = 0, Fz = 0;	0.0
23	// Loop over particles that exert force: vectorization expected here	
24	#pragma vector aligned	
25	for (int j = 0; j < nParticles; j++) {	1.1
26	// Avoid singularity and interaction with self	
27	const float softening = 1e-20;	
28	// Newton's law of universal gravity	
29	const float dx = particle.x[j] - particle.x[i];	0.0
30	const float dy = particle.y[j] - particle.y[i];	1.0
31	const float dz = particle.z[j] - particle.z[i];	1.0
32	const float dSquared = dx*dx + dy*dy + dz*dz + softening	1.0
33	const float invD = 1.0 / dSquared;	1.0
34	const float invD3 = invD*invD*invD;	1.0
35	const float Fx = -G*particle.mass[j]*dx*invD3;	1.0
36	const float Fy = -G*particle.mass[j]*dy*invD3;	1.0
37	const float Fz = -G*particle.mass[j]*dz*invD3;	1.0
38	Fx += particle.mass[i]*invD3*dy*dz;	1.0
39	Fy += particle.mass[i]*invD3*dz*dx;	1.0
40	Fz += particle.mass[i]*invD3*dx*dy;	1.0
41	particle.Fx[i] += Fx;	1.0
42	particle.Fy[i] += Fy;	1.0
43	particle.Fz[i] += Fz;	1.0
44	}	1.1
45	}	1.4

Selected (Total Time): 1.1

Vectorized - Advisor

Where should I add vectorization and/or threading parallelism? INTEL ADVISOR XE 2016

Elapsed time: 1.47s Vectorized Not Vectorized FILTER: All Module All Sources Loop All Threads

Summary Survey Report Refinement Reports Annotation Report

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops				Instruction Set Analysis			Advanced	Location
						Vec...	Efficiency	Gai...	VL ...	Traits	Data ...	Num.		
[loop in MoveParticles at nbody.cc:27]		32.896s	32.896s	Vectorized ...		AVX	~96%	7.7	8	FMA; Square Roots	Float...	14	nbody.cc:27	
[loop in MoveParticles at nbody.cc:20]	1 Potential ...	0.060s	32.956s	Threaded (Op...	inner loop was already vectorized					Extracts; FMA; Shu...	Float3...	16	nbody.cc:20	
[loop in MoveParticles at nbody.cc:57]	2 Assume ...	0.020s	0.020s	Threaded (Op...	vector dependence prevents vectorization					FMA	Float32	4	nbody.cc:57	
[loop in start_thread]		0.000s	31.616s	Scalar								0		
[loop in main at nbody.cc:101]	1 Assume ...	0.000s	1.360s	Scalar	vector dependence prevents vectorization					Divisions	Float64	8	nbody.cc:101	
[loop in [OpenMP worker at z_Linux_u...		0.000s	31.616s	Scalar								0	z_Linux_util.c:769	
[loop in __kmp_launch_thread at kmp_...		0.000s	31.616s	Scalar								0	kmp_runtime.c:5606	

Source Top Down **Loop Analytics** Loop Assembly Recommendations Compiler Diagnostic Details

32.896s
Vectorized (Body) Total time

AVX; FMA 32.896s
Instruction Set Self time

Memory 17% (4)
Compute 74% (17)
Other 9% (2)
Instruction Mix Summary

7.71x
Vectorization Gain

~96%
Vectorization Efficiency

Traits
FMA, Square Roots

Instruction Mix
Memory: 4 Compute: 17 Other: 2 Number of Vector Registers: 14

Hands-on Exercise

- Goal: Identify hotspots in sample code
 - Targets for parallelization and/or optimization
- Test code has 4 functions: mm[1-4]
 - Each does a different version of matrix-matrix multiplication $C=A \times B$
- Each function is **called** a different number of times
 - Approximate call frequency:
 - mm1: 10%
 - mm2: 20%
 - mm3: 30%
 - mm4: 40%
 - Where should we optimize?

Adroit Test Set Up

- Enable X11 forwarding
 - “ssh -Y -C <user>@adroit.princeton.edu
 - Will need local xserver (XQuartz for OSX, Xming for Windows)
- Clone the repo

<https://github.com/cosden/CoDaS-HEP-Perf-Tuning>

- Follow instructions in repo Readme.md
- What functions are most/least expensive?
- Change threshold values to select only the fastest function

Hands-on Discussion

- During a break feel free to try
 - GUI on head node
 - Other analyses on compute node
 - advanced-hotspots
 - general-exploration
 - Your code?

Examples

A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
 - VTune
2. OpenMP Scaling
 - VTune
3. Vectorization
 - Advisor
4. Cache performance & Hardware Metrics
 - VTune

Hardware Counters

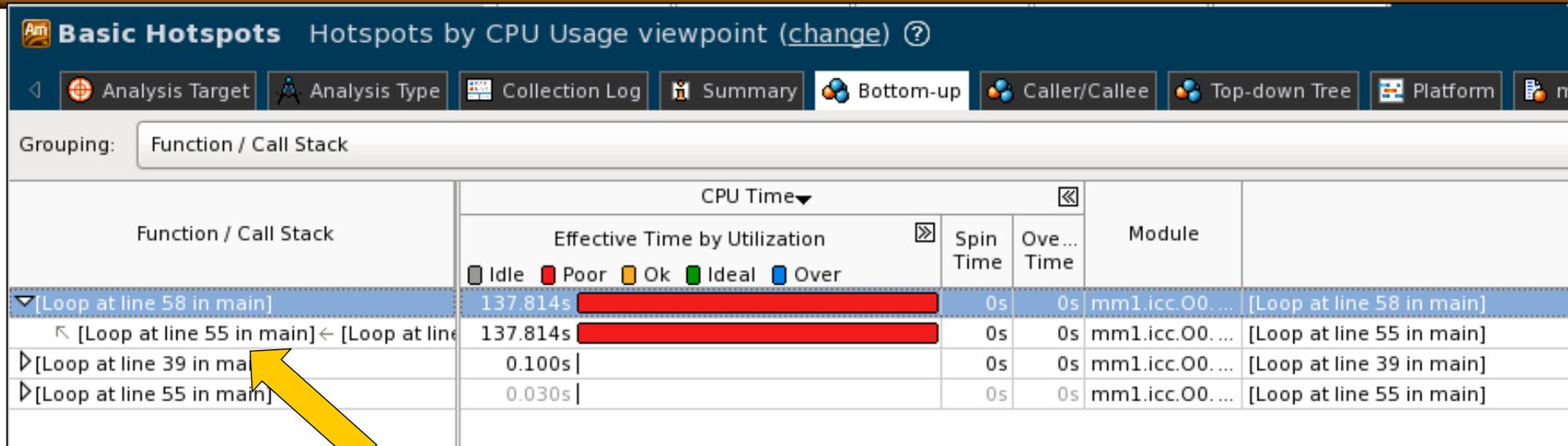
- Basic profilers “sample” the current program counter to see what instruction the cpu is currently executing.
- Then, after collecting enough samples, can give a good approximation as to what the program was doing.

- Performance Hardware Counters
 - Special purpose registers to track hardware events
- VTune can access hardware counters
 - Other profiling tools can access too, many build on top of open-source PAPI
- Literally hundreds of possible events to measure
 - Different in name and meaning for different chips

Hardware Counter Code Example

- 2D Matrix multiplication from before – mm1
 - 3 nested for loops
 - Let's assume we don't know why it is slow or even if it is slow

VTune Hotspots – Bottom-up View



Basic Hotspots Hotspots by CPU Usage viewpoint (change) ?

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

Grouping: Function / Call Stack

Function / Call Stack	CPU Time		Spin Time	Ove... Time	Module
	Effective Time by Utilization				
	Idle	Poor	Ok	Ideal	Over
[Loop at line 58 in main]	137.814s	<div style="width: 100%; height: 10px; background-color: red;"></div>	0s	0s	mm1.icc.O0. ... [Loop at line 58 in main]
[Loop at line 55 in main]	137.814s	<div style="width: 100%; height: 10px; background-color: red;"></div>	0s	0s	mm1.icc.O0. ... [Loop at line 55 in main]
[Loop at line 39 in main]	0.100s		0s	0s	mm1.icc.O0. ... [Loop at line 39 in main]
[Loop at line 55 in main]	0.030s		0s	0s	mm1.icc.O0. ... [Loop at line 55 in main]

Double click here

opens source below:

58	for (int k = 0; k < matrix_size; k++)	2.6%	<div style="width: 2.6%; height: 10px; background-color: red;"></div>	0.0%	0.0%
59	{				
60	C[i][j] += A[i][k] * B[k][j];	97.3%	<div style="width: 97.3%; height: 10px; background-color: red;"></div>	0.0%	0.0%
61	}				
62	}				
63	}				

Clearly there is one very expensive line inside nested for loops. *But why?*

General Exploration Analysis - VTune

General Exploration General Exploration viewpoint (change) ?

Analysis Target Analysis Type Collection Log Summary Bottom-up Event Count Platform matrix_multip...

Elapsed Time: 113.478s

Clockticks:	325,550,488,325
Instructions Retired:	369,072,553,608
CPI Rate:	0.882
MUX Reliability:	0.987
Front-End Bound:	2.1%
Bad Speculation:	0.2%
Back-End Bound:	69.5%

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps. This metric describes a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their retirement slots. uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the over-saturation of the pipeline are highlighted.

Memory Bound:	49.8%
---------------	-------

The metric value is high. This can indicate that the significant fraction of execution pipeline slots are stalled due to memory access issues. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth, and memory latency.

L1 Bound:	0.075
-----------	-------

This metric shows how often machine was stalled without missing the L1 data cache. The L1 Bound metric is highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead:	0.503
----------------	-------

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.005
L2 Bound:	0.016
L3 Bound:	0.213

This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance.

Contested Accesses:	0.000
Data Sharing:	0.000
L3 Latency:	0.327

This metric shows a fraction of cycles with demand load accesses that hit the L3 cache under unloaded scenarios (possibly L3 latency limited). Avoiding private cache misses (i.e. L2 misses/L3 hits) will improve the latency, reduce contention with sibling physical cores and increase performance. Note the value of this node may overlap with its siblings.

SQ Full:	0.000
DRAM Bound:	0.105

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth:	0.089
Memory Latency:	0.654

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Store Bound:	0.000
Core Bound:	19.7%

All flagged in pink:

Memory Bound: 49.8%
DRAM Bound: 0.105
Memory Latency: 0.654

Hardware Counters

- Remember every generation of chips has different counters
- VTune developers have calculated *derived metrics* that they feel best represent real problems
 - For example the Haswell/Broadwell hardware counter: MEM_LOAD_UOPS_RETIRED.L3_MISS_PS may not truly indicate if the cpu was stalled waiting for the data.
 - Colored in pink when they represent a value that *might* warrant investigation.
- My experience with cache misses:
 - Memory Bound, DRAM Bound, & Memory Latency together are very good indicators that cache is not being used well.
- How to fix cache misses?
 - In this case: Reorder nested loops (mm2)

After Loop Reordering

General Exploration General Exploration viewpoint (change) ?

Analysis Target Analysis Type Collection Log Summary Bottom-up Event Count Platform matrix_multip...

Elapsed Time: 2.602s

Clockticks:	7,446,011,169
Instructions Retired:	13,616,020,424
CPI Rate:	0.547
MUX Reliability:	0.981
Front-End Bound:	8.9%
Bad Speculation:	0.3%
Back-End Bound:	42.2%

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the pipeline. This metric describes a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units. uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded dispatch units.

Memory Bound: 24.4%
The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to memory access. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information.

L1 Bound: 0.051
This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache type is shared. Cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied. High values are highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead:	0.046
Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.120

A significant proportion of cycles is spent dealing with false 4k aliasing between loads and stores. Use the source/assembly view to identify the aliasing between loads and stores, and then adjust your data layout so that the loads and stores no longer alias.

L2 Bound: 0.102
This metric shows how often machine was stalled on L2 cache. Avoiding cache misses (L1 misses/L2 hits) will improve the latency and increase performance.

L3 Bound: 0.109
This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance.

Contested Accesses:	0.000
Data Sharing:	0.000
L3 Latency:	0.389

This metric shows a fraction of cycles with demand load accesses that hit the L3 cache under unloaded scenarios (possibly L3 latency limited). Avoiding private cache misses (i.e. L2 misses/L3 hits) will improve the latency, reduce contention with sibling physical cores and increase performance. Note the value of this node may overlap with its siblings.

SQ Full: 0.333

This metric measures fraction of cycles where the Super Queue (SQ) was full taking into account all request-types and both hardware SMT threads. The Super Queue is used for requests to access the L2 cache or to go out to the Uncore.

DRAM Bound:	0.032
Memory Bandwidth:	0.011
Memory Latency:	0.880
Store Bound:	0.000

Core Bound:	17.8%
Retiring:	48.7%
Total Thread Count:	1
Paused Time:	0s

Change in metrics:

Memory Bound: 49.8% → 24.4%

DRAM Bound: 0.105 → 0.032

Memory Latency: 0.654 → 0.880

Elapsed Time: 132s → 2.6s

Results

- Reordering nested loops resulted in a better use of cache and a 50x speedup
- Clues in the VTune metrics (hardware counters)
- Note that “optimized” case still showed MANY pink highlighted metrics
 - Something will always be limiting performance
 - Always requires interpretation and consideration

Function / Call Stack	CPU Time			Module
	Effective Time by Utilization	Spin Time	Ove...	
[Loop at line 59 in main]	2.499s	0s	0s	mm1a.icc.O3 ... [Loop at line 59 in main]
↳ [Loop at line 56 in main] ← [Loop at line 59 in main]	2.499s	0s	0s	mm1a.icc.O3 ... [Loop at line 56 in main]
↳ [Loop at line 39 in main]	0.080s	0s	0s	mm1a.icc.O3 ... [Loop at line 39 in main]
↳ [Loop at line 56 in main]	0.050s	0s	0s	mm1a.icc.O3 ... [Loop at line 56 in main]
↳ [Loop at line 30 in main]	0.010s	0s	0s	mm1a.icc.O3 ... [Loop at line 30 in main]
↳ [Loop at line 122 in main]	0.010s	0s	0s	mm1a.icc.O3 ... [Loop at line 122 in main]

Final Remarks

- Measurement is key
- Tools are helpful
 - Free Intel tools for students: <https://software.intel.com/en-us/qualify-for-free-software/student>
- Fast computing of wrong results is completely irrelevant!
 - Have correctness test(s)
 - Test after each modification
- Don't fall into the “tuning trap”
 - Remember what really matters: **Total Runtime**