Profiling and benchmarking

Roman Khatko

Optimization Problem

- **Define Optimization Criteria**
  - Elapsed time
  - Throughput
  - Requests / sec
  - Latency
  - Frames per second
  - Power
  - …
- **Define Stop Criteria**
  - Optimization Criteria value when we're done
  - May depend on the tuning potential

Workload

- **System → Workload**
- Simple case – app itself
- In complex system – focus on something:
  - Specific data
  - Specific execution scenario
  - …

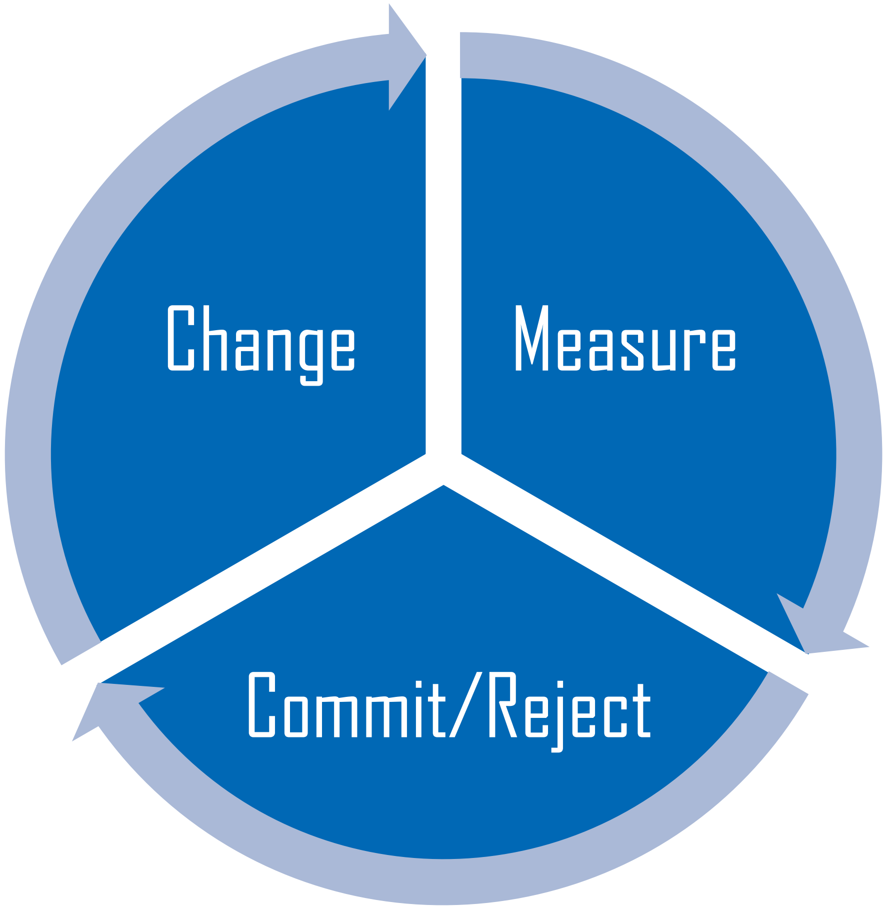
Workload must be:

- **Representative** – Workload improvement leads to system improvement.
- **Measurable** – It is possible to calculate Optimization Criteria after each run.
- **Reproducible** – Optimization Criteria value persists if the workload does not change.

Start with a solid baseline

- Use optimal compiler flags
  - Compile in **Release** mode
  - Compile with optimization flags (at least -O2)
- Use optimized libraries and runtimes
  - Intel® oneAPI libraries
  - Intel® Distribution for Python
- Measure on the system without unrelated activity
- Run multiple times to calculate deviations
- Check the correctness
- *Invest in the measurement automation!*

Optimization Process



- **Start with the baseline**
- **Iterate**
- **Finish**, when stopping condition is met
- For efficiency, this process should be guided by the Profiler

Software optimization directions

Compiler options

- **Release** build with **debug information** for profiling
- Optimization levels: -O0, **-O2, -O3, -Ofast**
  - Enabled optimizations depend on the compiler
  - Example – to allow auto-vectorization:
    - Intel Compiler: -O2
    - GCC: -O2 -ftree-vectorize
- Allowed Instructions
  - ICC: -xHost, -xCASCADELAKE – will run only on current machine; only on CascadeLake servers
  - GCC: -march=native, -march=cascadelake
  - Portable binaries with multiple code paths
    - ICC: -axCASCADELAKE,COMMON-AVX512,CORE-AVX2,SSE4.2
    - GCC: target_clones + flatten function attributes
- Fast Math
  - -Ofast: -O3 + fast math optimizations
  - ICC: -fp-model fast
  - GCC: -ffp-contract=fast

Example:

- X*X*X*X*X*X*X*X -> 7 MUL
- A=X*X, B=A*A, C=B*B -> 3 MUL

Check if options allow the compiler to optimize!

Ways to optimize performance and their impact

- **Performance increase is unknown and might be huge**
  - Algorithmic optimization
  - Design optimization

- **Limited performance increase**
  - Parallelization
  - Vectorization
  - Memory Access
  - Other microarchitecture optimizations
  - Offload

Optimization level – potential impact

Algorithmic & design optimization

∞

Parallelization & vectorization

Depends..

e.g. *N* times faster when having *N* cores – ideally

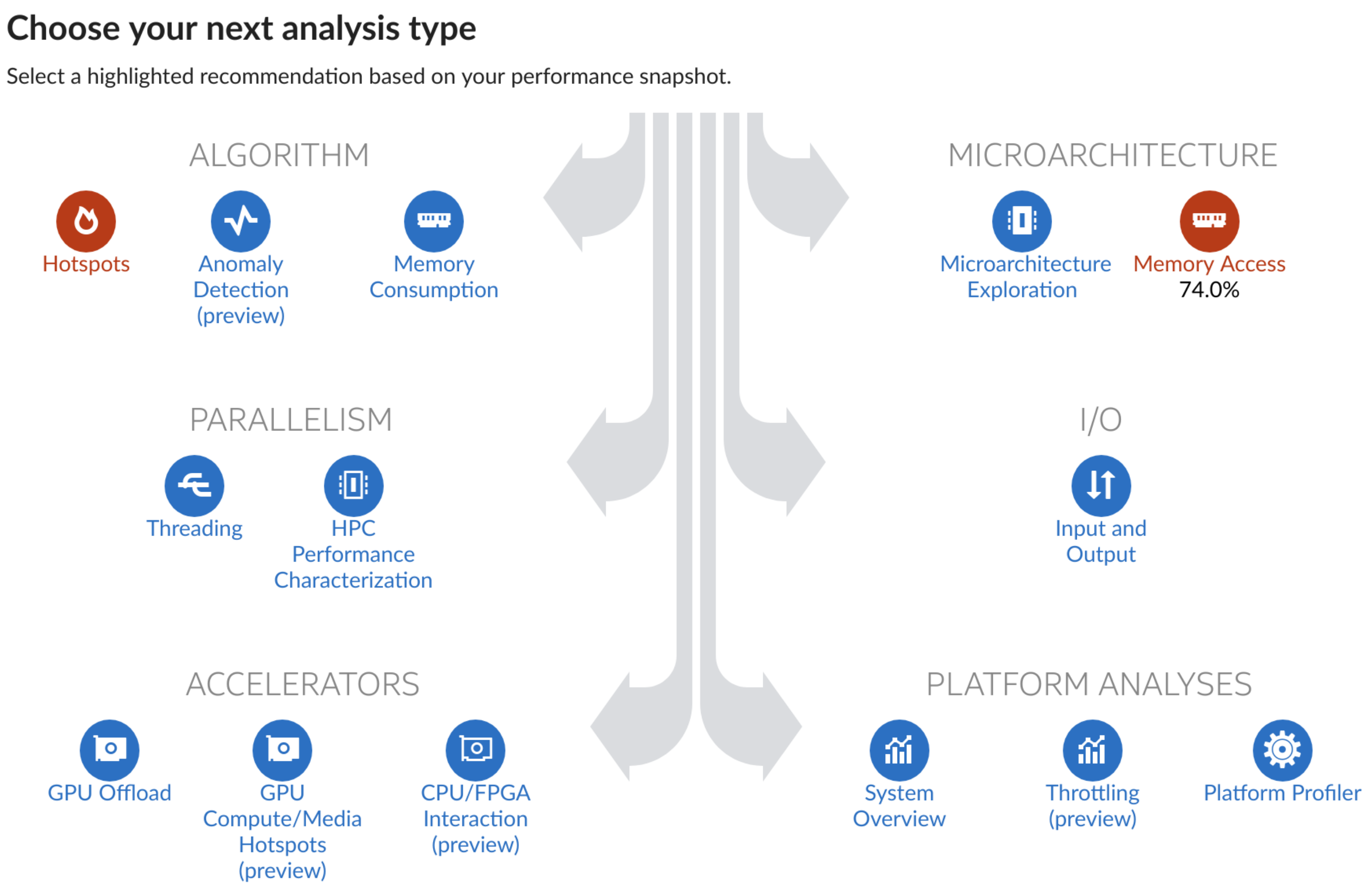
Microarchitecture optimization

?

Evaluate the directions – data driven approach

VTune Profiler – Performance Snapshot

- **Algorithmic and design optimizations**
- Parallelization
- Vectorization
- Offload
- Memory Access
- Other microarchitecture optimizations



Algorithmic optimization

VTune Profiler - Hotspots

- Focus on **CPU Time**
- Identify **Hotspots** – most time-consuming functions, loops
- Focus on Hotspots
- Choose better algorithms (complexity, constant)
- Use libraries optimized for target HW

Top Hotspots

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

Function	Module	CPU Time
multiply1	matrix	146.389s
init_arr	matrix	0.020s

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

Software design optimization

VTune Profiler - Hotspots

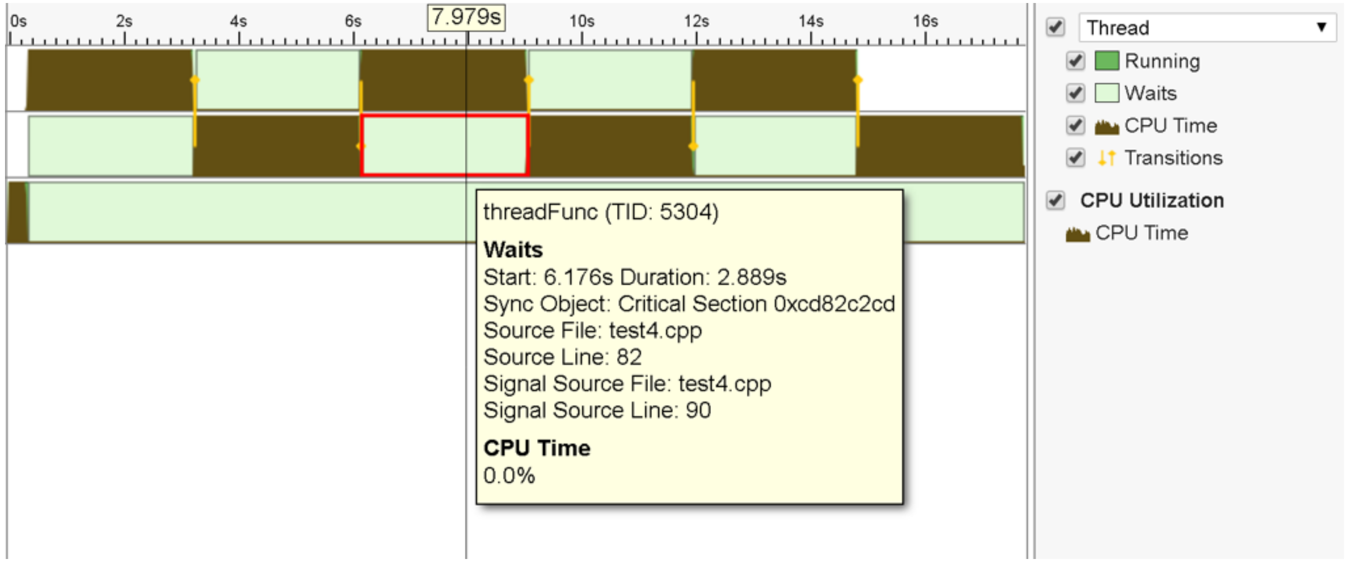
- Algorithm optimization – “calculate fast”
- Design optimization – “avoid calculations”
- Rework application architecture to reduce the calculations
  - Cache frequent requests
  - Avoid unnecessary copies

Parallel optimizations

VTune Profiler - Threading

Perspectives:

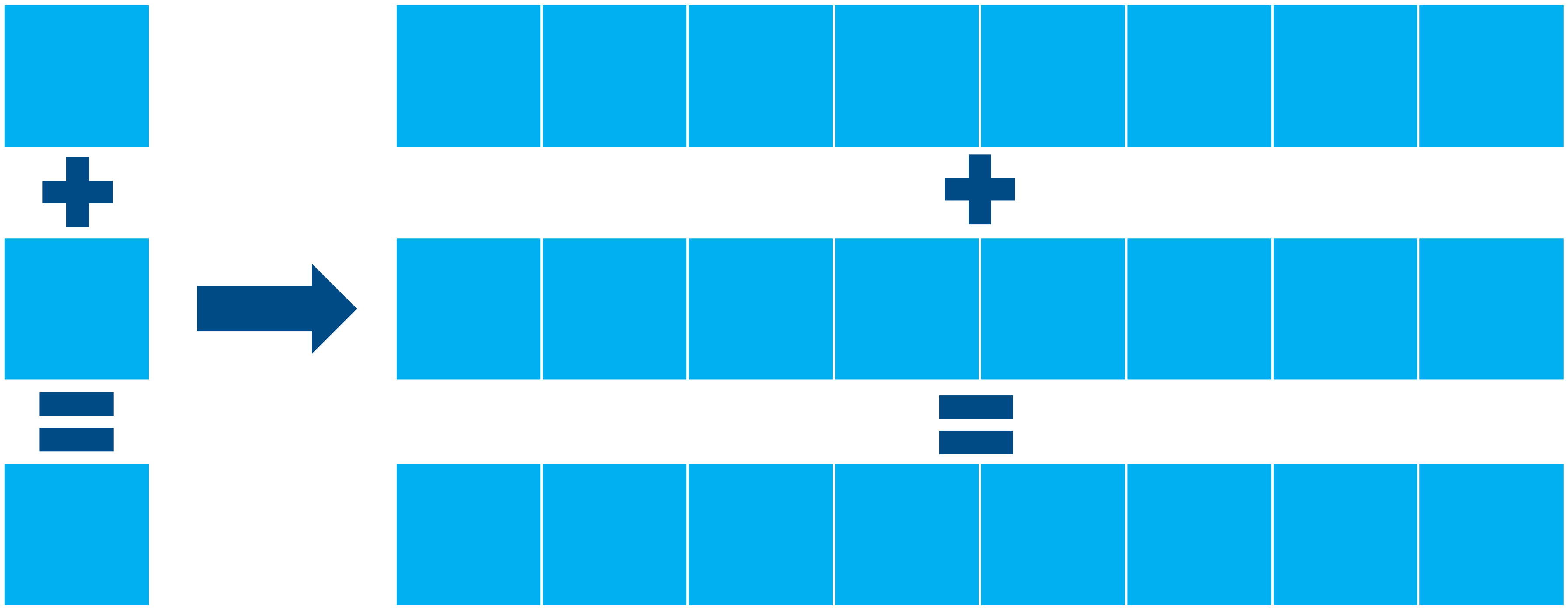
- CPUs
  - CPU Utilization
  - Logical cores, Physical cores
- Threads
  - Effective CPU time, Spin and Overhead Time
  - Inactive time, Sync Wait time, Preemption time
- Sync objects



Vectorization

Intel® Advisor

SIMD instructions:



Vectorization boundaries

AVX-512 zmm register size

int	int	int	int	int	int	int	int	int	int	int	int	int	int	int	int	16x 32-bit int

double	double	double	double	double	double	double	double	8x 64-bit double

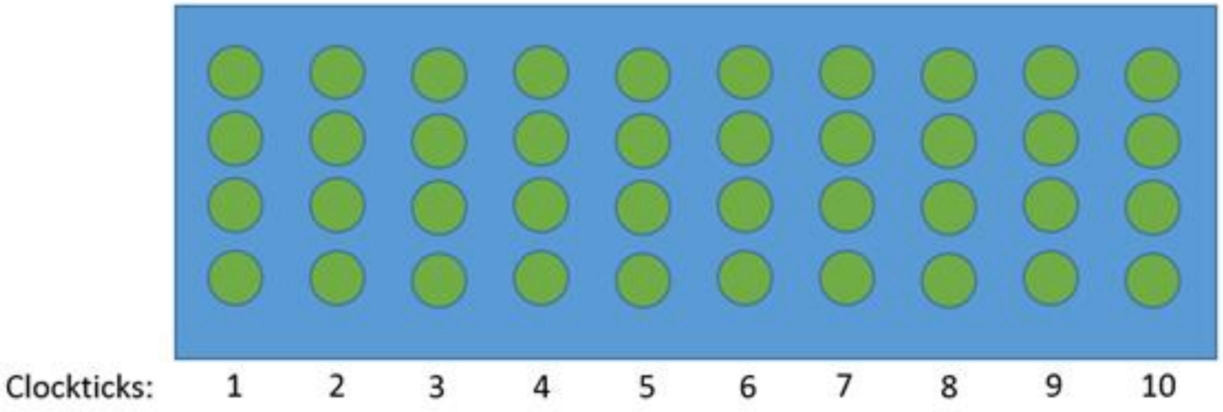
Vectorization

Intel® VTune Profiler – HPC Performance Characterization

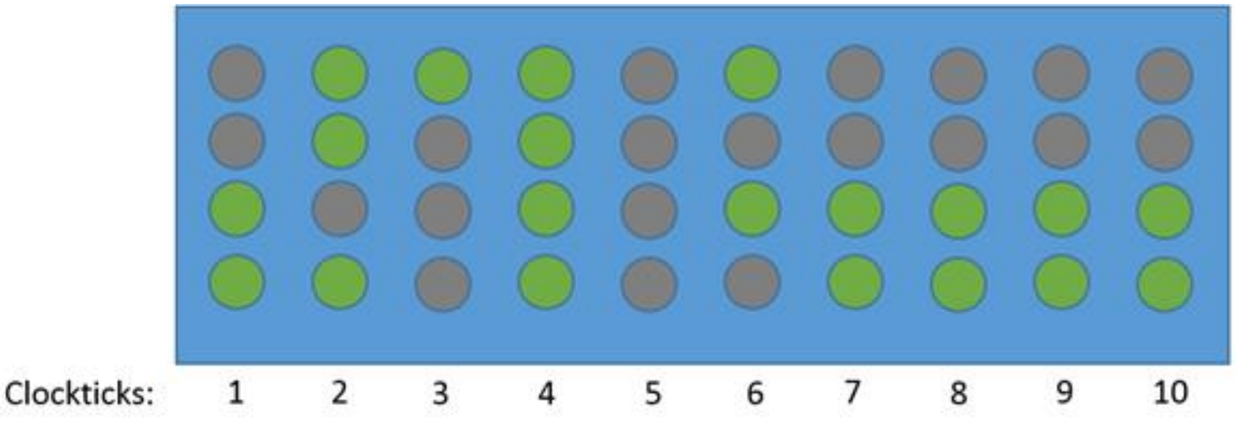
- New CPU’s support x86 ISA extensions: AVX, AVX2, AVX-512
- Optimized libraries
  - **Update** the library to benefit from the new ISA extension
- Pragmas – hints to the compiler
  - **Recompile** to benefit from the new ISA extension
  - Align, prefer SOA vs AOS
- Intrinsics & asm
  - **Rewrite** to benefit from the new ISA extension
  - Try to avoid with pragmas and compiler options
  - Limit to small “kernels”
  - Keep the reference code

uArch optimizations

- µArch performance gain is limited by ideal CPI (Clocks Per Instruction)
- CPI does not depend on CPU frequency which may change
- Why instructions might not retire:
  - Front-end bound
  - Back-end bound
  - Bad Speculation
- [Top-Down methodology]



Ideal CPU pipeline

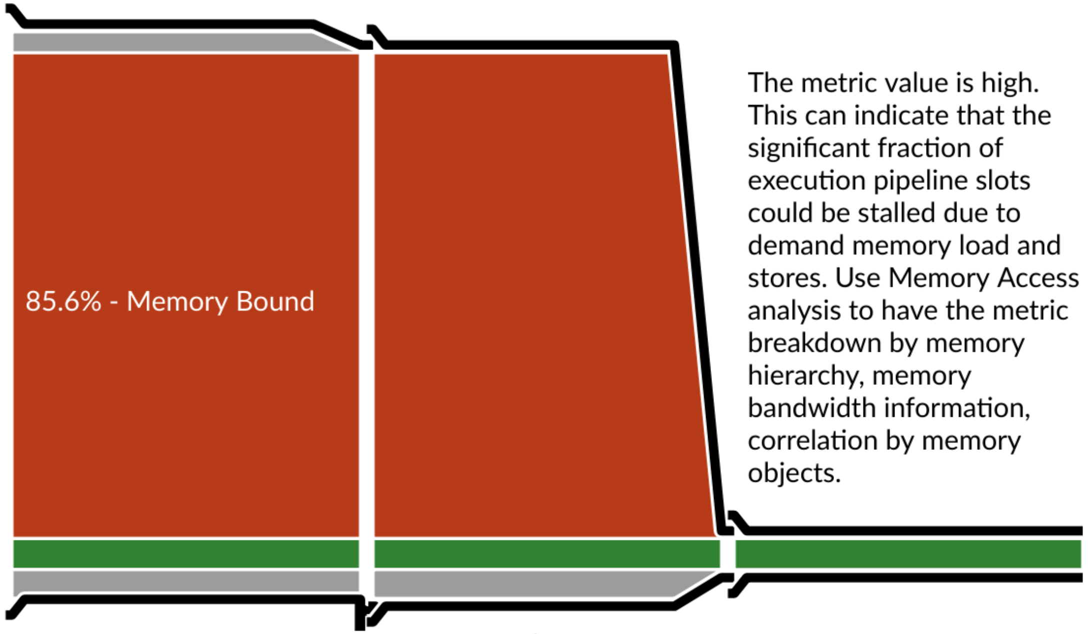


Real CPU pipeline

Microarchitecture optimization – Top down

Elapsed Time: 22.583s

Clockticks:	647,479,000,000	
Instructions Retired:	69,560,600,000	
CPI Rate:	9.308	
MUX Reliability:	0.998	
Retiring:	5.5%	of Pipeline Slots
Front-End Bound:	3.7%	of Pipeline Slots
Bad Speculation:	0.2%	of Pipeline Slots
Back-End Bound:	90.7%	of Pipeline Slots
Memory Bound:	85.6%	of Pipeline Slots
L1 Bound:	0.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	0.8%	of Clockticks
DRAM Bound:	89.4%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	5.1%	of Pipeline Slots
Average CPU Frequency:	3.7 GHz	
Total Thread Count:	9	
Paused Time:	0s	



The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Top-down metrics can be applied to the specific hotspot

Memory Access optimizations

VTune Profiler – Memory Access

- Start with Top-Down methodology to locate the problem
- Make your app NUMA-aware
- Reduce frequent DRAM accesses
- Fix false sharing issues
- Add hints for prefetchers

Summary

- Use optimized libraries
- Use optimal compiler flags
- Get a solid baseline
- Define optimization criteria for your workload
- Profile your code, measure
- Optimize your code design and algorithms first
- Parallelize
- Vectorize, help the compiler rather than play with assembly code
- Optimize memory access
- Optimize for microarchitecture efficiency
- Offload to the accelerators

Links

- Intel® VTune™ Profiler
- Intel® VTune™ Profiler Performance Analysis Cookbook
- Intel® oneAPI Toolkits
- Intel® Advisor

Q&A

VTune Profiler Demo

intel®