Computer Architecture *and* Organization

Lecture Series for VIT, Chennai

Jaiber John, Intel

2022

Module 6
Computer Performance Analysis

Contents from:

1. [Stallings2016] Chapter 2
2. [Denis2020] Chapters 2, 4, 5

Computer Performance Analysis

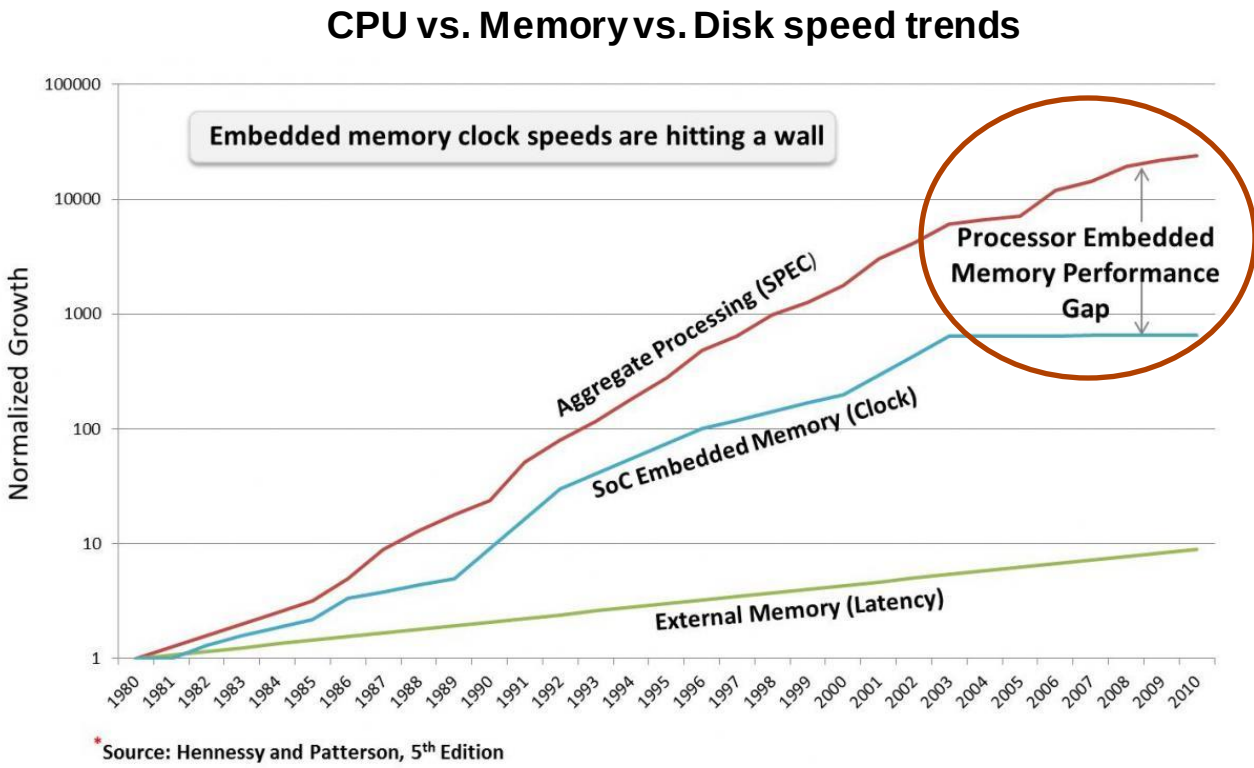
- Performance Metrics and Evaluation
  - Historic Performance measures
  - Benchmarks
- CPU Performance bottlenecks
- Profiling
  - Vtune
  - Performance Optimization

Computer Performance

- Computer Performance depends on:
  - CPU
  - Memory
  - IO

- Imbalance between growth of performance

- Goal of System Architect is to:
  - Identifying bottlenecks
  - Compensate mismatch
  - Reduce power consumption
  - Improve core performance



CPU Performance

- CPU Performance achieved by:
  - Increasing Clock speed
  - Increasing Logic density
  - Caching strategies
  - Exploiting parallelism (ILP, DLP, TLP)
  - Many Integrated Cores
  - Component Integration (GPU, IPU, various microcontrollers)

- How to evaluate performance across various CPU models in a standardized way?

Moore's Law and Dennard Scaling

- Moore's Law (1965)
  - **Number of transistors in a dense integrated circuit (IC) doubles about every two years**
  - Has slowed down since 2010

- Dennard Scaling (1974)
  - Known as MOSFET scaling, is a scaling law which states roughly that, as transistors get smaller, their power density stays constant, so that the power use stays in proportion with area; both voltage and current scale with length.
  - Ignored leakage current and threshold voltage
  - "Power wall" – occurs around 4 GHz

Amdahl's Law (1967)

- The speedup using a parallel processor with N processors that fully exploits the parallel portion of the program is :

$$\text{Speedup} = \frac{\text{Time to execute program on a single processor}}{\text{Time to execute program on } N \text{ parallel processors}}$$

$$= \frac{T(1-f) + Tf}{T(1-f) + \frac{Tf}{N}} = \frac{1}{(1-f) + \frac{f}{N}}$$

Where, f -> fraction of time taken by parallel part of a program
(1 – f) -> fraction of time taken by sequential part of a program
N -> number of processors

Measures of CPU Performance

- Cycle time $\tau = 1/f.$

  f -> clock frequency

- Cycles per Instruction $CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{I_c}$

  I_i -> no. of executed instructions of type i

  I_c -> total number of executions

  CPI_i -> CPI of instruction type *i*

- Time taken to execute program:

$$T = I_c \times CPI \times \tau$$

- MIPS (Million Instructions per second)

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

Problem #1 (CPI, T, MIPS)

Given

- Total instructions (Ic) = 2 million = 2,000,000
- Frequency = 400 MHz = 400,000,000 Hz

Instruction Type	*CPI*	Instruction Mix (%)
Arithmetic and logic	1	60
Load/store with cache hit	2	18
Branch	4	12
Memory reference with cache miss	8	10

Calculate avg. CPI, MIPS rate, program execution time T ?

CPI = 2.24 MIPS rate ~ 178 Program time T = 0.0139s

Arithmetic, Geometric, Harmonic Mean

Arithmetic mean

$$AM = \frac{x_1 + \cdots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \textbf{(2.4)}$$

For time-based variables

Geometric mean

$$GM = \sqrt[n]{x_1 \times \cdots \times x_n} = \left(\prod_{i=1}^{n} x_i\right)^{1/n} = e^{\left(\frac{1}{n}\sum_{i=1}^{n}\ln(x_i)\right)} \qquad \textbf{(2.5)}$$

For comparing benchmarks across systems

Harmonic mean

$$HM = \frac{n}{\left(\frac{1}{x_1}\right) + \cdots + \left(\frac{1}{x_n}\right)} = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{x_i}\right)} \qquad x_i > 0 \qquad \textbf{(2.6)}$$

For averaging rates (MIPS, MFLOPS)

It can be shown that the following inequality holds:

$$AM \leq GM \leq HM$$

Performance Evaluation

- Theoretical approach – Performance Modeling and Simulation
  - While designing new systems
  - Queuing theory, discrete event simulation, Markov models, etc
  - Trace and event simulation, performance prediction

- Practical approach – Benchmarking
  - Run programs called “Benchmarks” on a computer and measure various metrics
  - MIPS and MFLOPS cannot be directly compared across systems
  - SPEC, LINPACK, etc.. Popular benchmarks
  - VTune – collect and analyze “micro-architectural” events

Benchmarks

- SPEC
  - Standard Performance Evaluation Corporation (SPEC)
  - https://www.spec.org/benchmarks.html

- LINPACK/HPL (High Performance LINPACK) – For Supercomputer performance
  - www.Top500.org

- MLBench – For Machine Learning performance

SPEC Flowchart

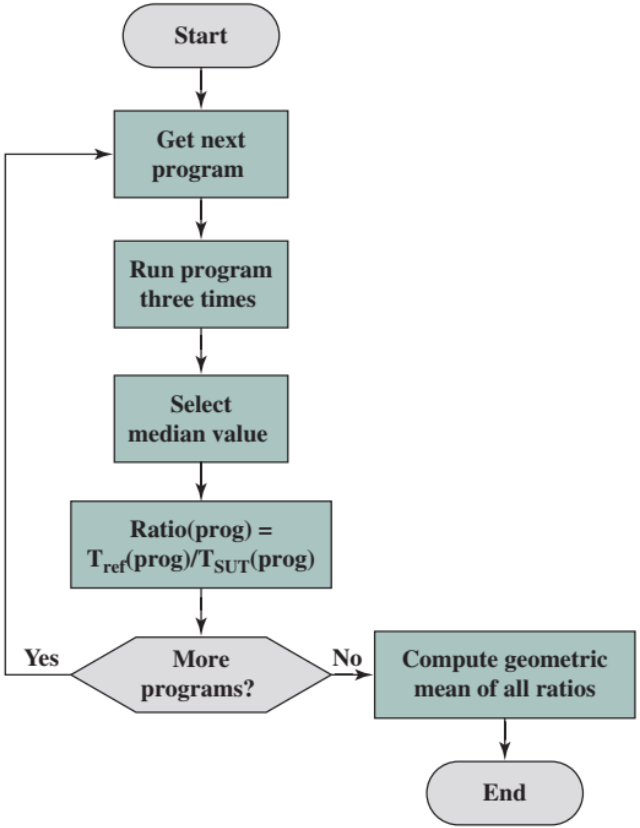


Figure 2.7 SPEC Evaluation Flowchart

Normalized runtime ratio $r_i = \frac{Tref_i}{Tsut_i}$

Final metric (GM of ratios) $r_G = \left(\prod_{i=1}^{12} r_i\right)^{1/12}$

Performance Analysis using VTune

- Free Book: Performance Analysis and Tuning on Modern CPUs, Denis Bakhvalov

Why Performance Engineering?

Version	Implementation	Absolute speedup	Relative speedup
1	Python	1	—
2	Java	11	10.8
3	C	47	4.4
4	Parallel loops	366	7.8
5	Parallel divide and conquer	6,727	18.4
6	plus vectorization	23,224	3.5
7	plus AVX intrinsics	62,806	2.7

Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices

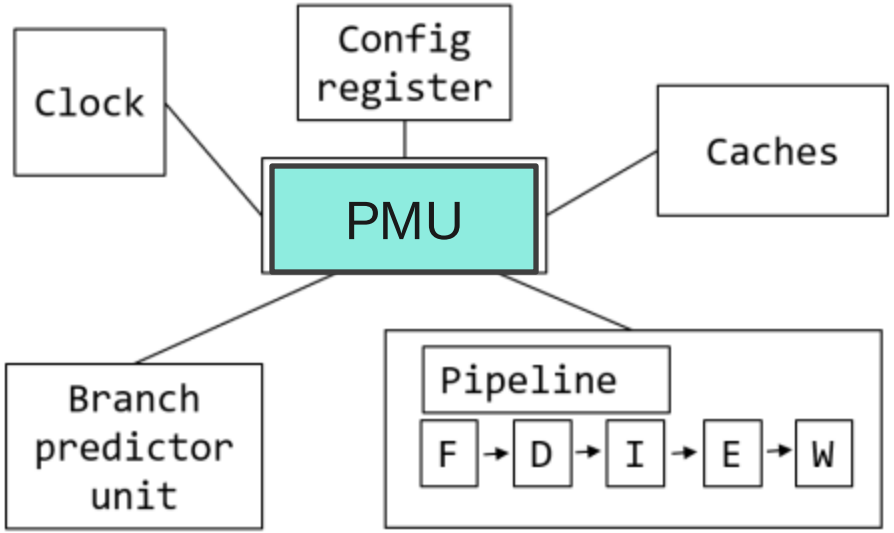
Performance Monitoring

- Software or Hardware Level

- Usages
  - Performance Tuning
  - Workload Characterization
  - Capacity Planning

Performance Evaluation with PMU

- PM - Critical step for Performance Evaluation
- Different ways to improve performance
  - Better Hardware
  - Better Algorithms
  - Optimizations in SW (Loop vectorization, unrolling, etc)
  - **Software Tuning**
- Performance Monitoring Unit (PMU)
  - Built-in processor core



Software Tuning by Performance Monitoring

- Frequent DRAM Access
- Cache Misses
- Page Faults
- Scheduling Overheads
- Core Underutilization
- Inefficient Synchronization
- .. Many more

PMU Events, Counters

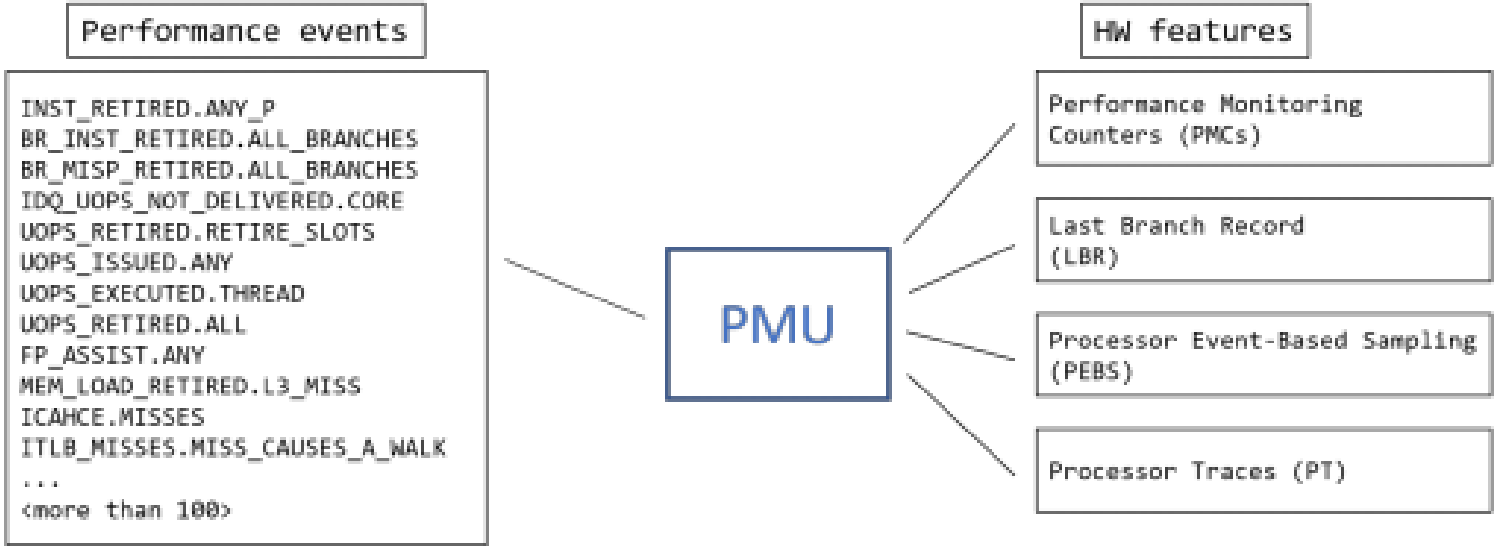


Figure 15: Performance Monitoring Unit of a modern Intel CPU.

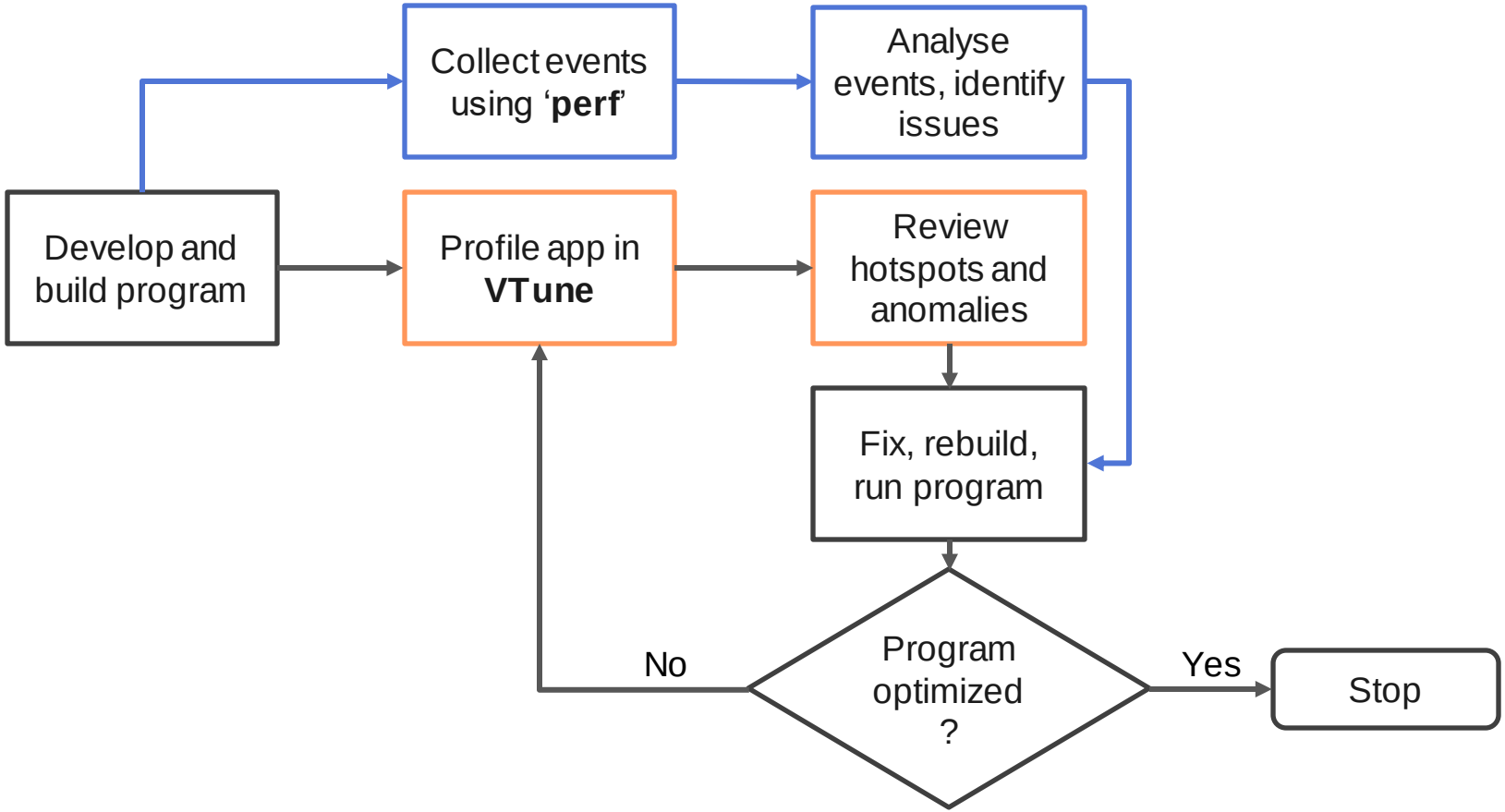
Tools

- Intel ‘VTune Profiler’
  - Fix Performance Bottlenecks Quickly and Realize All the Value of Hardware
- Intel ‘Advisor’
  - Efficient Vectorization, Threading, Memory Usage, and Accelerator Offloading
- Linux perf

VTune – Full System Profiler

Analysis Group	Analysis Types
Algorithm analysis	Hotspots Anomaly Detection Memory Consumption
Microarchitecture analysis	Microarchitecture Exploration Memory Access
Parallelism analysis	Threading HPC Performance Characterization
I/O analysis	Input and Output
Accelerators analysis	GPU Offload GPU Compute/Media Hotspots CPU/FPGA Interaction
Platform Analyses	System Overview Platform Profiler

Software Tuning and Optimization with VTune



Basic Analysis

- CPU Utilization
- Memory utilization
- CPI calculation
- Microarchitectural analysis
- Pipeline slots
- Cache Misses
- Branch mis-prediction

CPU Utilization

- Clock
  - Core clock – varies with frequency scaling
  - Ref clock – monotonic, counts independent of scaling (TSC)

$$CPU\ Utilization = \frac{CPU\_CLK\_UNHALTED.REF\_TSC}{TSC},$$

- If CPU Utilization low, app performance is low
- If CPU utilization too high, maybe it's wasting CPU cycles in busy loop

Cycles Per Instruction (CPI)

- Cycles Per Instruction (CPI) - how many cycles it took to retire one instruction on
  average.
- Instructions Per Cycle (IPC) - how many instructions were retired per one cycle on
  average.

$$IPC = \frac{INST\_RETIRED.ANY}{CPU\_CLK\_UNHALTED.THREAD}$$

$$CPI = \frac{1}{IPC},$$

Software Tuning with VTune

VTune – Hotspot Analysis

Hotspots | Hotspots by CPU Utilization | INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform | multiply.c

Source | Assembly

S...	Source	CPU Time: Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Spin Time	Overhead Time	Instructions Retired
37	c[i][j] = c[i][j] + a[i][k] * b[k][j];				
38	}				
39	}				
40	}				
41	}				
42					
43	void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], T				
44	{				
45	int i,j,k;				
46					
47	// This naive implementation of matrix multiply contains an in				
48	// Each iteration of the inner loop strides across the full wi				
49	// because the iterator 'k' is used in the first dimension of				
50	// This leads to bad cache reuse and significant memory stalls				
51	// Use Microarchitecture and Memory access analysis to estimat				
52	// See the 'multiply2' function implementation to overcome the				
53	for(i=tidx; i<msize; i=i+numt) {				
54	for(j=0; j<msize; j++) {	0.001s	0s	0s	1,900,000
55	for(k=0; k<msize; k++) {	2.738s	0s	0s	741,000,000
56	c[i][j] = c[i][j] + a[i][k] * b[k][j];	636.355s	0s	0s	313,458,200,000
57	}	2.769s	0s	0s	742,900,000
58	}				
59	}				
60	}				

VTune – Analysing Memory Access

Memory Access | Memory Usage | INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up | Platform

Elapsed Time: 102.308s

CPU Time:	634.334s	
Memory Bound:	**84.5%**	**of Pipeline Slots**
L1 Bound:	1.9%	of Clockticks
L2 Bound:	0.5%	of Clockticks
L3 Bound:	3.9%	of Clockticks
DRAM Bound:	**82.3%**	**of Clockticks**
DRAM Bandwidth Bound:	12.7%	of Elapsed Time
Store Bound:	0.0%	of Clockticks
Loads:	155,107,053,072	
Stores:	17,872,136,148	
LLC Miss Count:	7,876,151,292	
Average Latency (cycles):	39	
Total Thread Count:	10	
Paused Time:	0s	

VTune – Microarchitectural Exploration

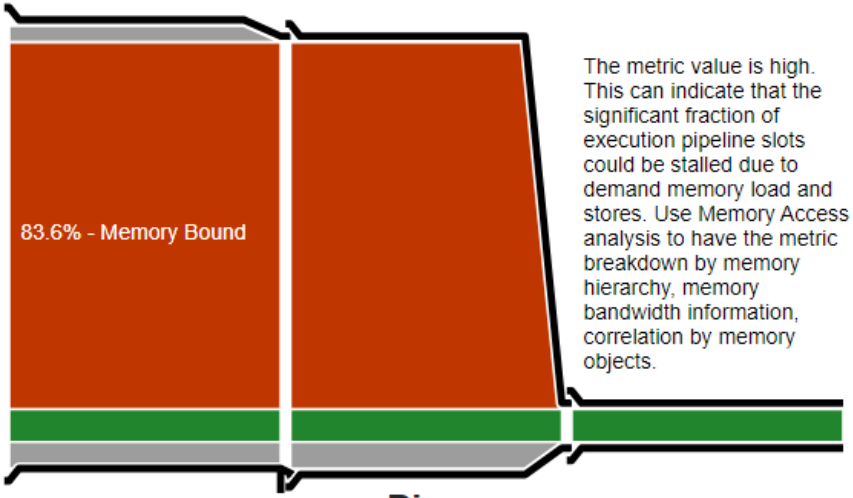
Microarchitecture Exploration Microarchitecture Exploration

INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

Elapsed Time: 2.731s

Clockticks:	37,686,500,000	
Instructions Retired:	3,773,400,000	
CPI Rate:	9.987	
MUX Reliability:	0.923	
Retiring:	7.5%	of Pipeline Slots
Front-End Bound:	3.7%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	89.6%	of Pipeline Slots
Memory Bound:	83.6%	of Pipeline Slots
L1 Bound:	0.0%	of Clockticks
L2 Bound:	1.0%	of Clockticks
L3 Bound:	19.1%	of Clockticks
DRAM Bound:	66.2%	of Clockticks
Memory Bandwidth:	81.8%	of Clockticks
Memory Latency:	11.5%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	6.0%	of Pipeline Slots
Average CPU Frequency:	2.7 GHz	
Total Thread Count:	11	
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.

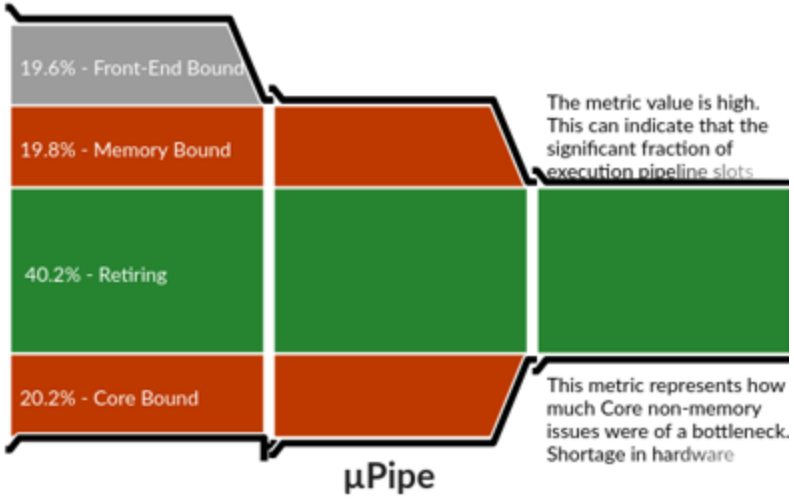
µPipe

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.

VTune – Analyse Page faults

Elapsed Time: 105.737s

Clockticks:	3,143,784,000,000	
Instructions Retired:	2,552,136,000,000	
CPI Rate:	1.232	
MUX Reliability:	0.999	
Retiring:	40.2%	of Pipeline Slots
Front-End Bound:	19.6%	of Pipeline Slots
Bad Speculation:	0.1%	of Pipeline Slots
Back-End Bound:	40.1%	of Pipeline Slots
Memory Bound:	19.8%	of Pipeline Slots
L1 Bound:	27.0%	of Clockticks
DTLB Overhead:	85.5%	of Clockticks
Load STLB Hit:	84.7%	of Clockticks
Load STLB Miss:	0.8%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	25.5%	of Clockticks
FB Full:	0.3%	of Clockticks
L2 Bound:	1.0%	of Clockticks
L3 Bound:	1.0%	of Clockticks
DRAM Bound:	1.2%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	20.2%	of Pipeline Slots
Average CPU Frequency:	3.8 GHz	
Total Thread Count:	9	
Paused Time:	0s	



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.

Analysis of various Matrix multiply functions and Optimization

Matrix Multiplication Sample code - VTune

- Summary: Multi-threaded sample code that implements several types of matrix multiplication functions for analysis
- Placed under: [Documents]\VTune\samples\matrix
- Configurations:
  - **NUM_THREADS** or **MAX_THREADS** : 16
  - **NUM** or **MSIZE** : 2048
  - **MATRIX_BLOCK_SIZE** : 64
  - **MULTIPLY** : “multiply0”, “multiply1”, ... “multiply5”
- 3 Modes:
  - USE_THR - Use Threading
  - USE_OMP - Use OpenMP
  - USE_KML - Use Intel Kernel Math Library (Link)
- Runs on:
  - Windows and Linux

Program Flow

```
main()                - matrix.c  64
  ParallelMultiply()  - thrmodel.c 79 (USE_THR), 171 (USE_OMP), 210 (USE_MKL)
    ThreadFunction    - thrmodel.c 49
       MULTIPLY       - multiply.c (any one of multiply0, multiply1, ...)
```

Tutorial:
https://www.intel.com/content/www/us/en/develop/documentation/vtune-tutorial-common-bottlenecks-windows/top.html

#1. Basic serial Implementation (USE_THR, multiply0)

```
void multiply0(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
TYPE c[][NUM], TYPE t[][NUM])
{
    int i, j, k;

// Basic serial implementation
    for(i=0; i<msize; i++) {
        for(j=0; j<msize; j++) {
            for(k=0; k<msize; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

1. Doesn't use thread index, each loop multiples full matrix
2. Matrix is multiplied NUM_THREADS times
3. Very slow..

#1. Analysis (multiply0)

Elapsed Time: 387.730s

CPU

IPC:	0.705
SP GFLOPS:	0.004
DP GFLOPS:	0.044
x87 GFLOPS:	0.003
Average CPU Frequency:	1.5 GHz

GPU

Time:	1.5% (5.700s) of Elapsed time
IPC Rate:	1.311

Logical Core Utilization:

53.0% (4.239 out of 8)

Physical Core Utilization: 73.6% (2.944 out of 4)

Microarchitecture Usage: 26.8%

of Pipeline Slots

Memory Bound: 18.4%

of Pipeline Slots

Vectorization: 0.7%

of Packed FP Operations

GPU Active Time: 1.5%

#2. Parallel multiply (multiply1, USE_THR)

```
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM],
TYPE b[][NUM], TYPE c[][NUM], TYPE t[][NUM])
{
    int i,j,k;

    for(i=tidx; i<msize; i=i+numt) {
        for(j=0; j<msize; j++) {
            for(k=0; k<msize; k++) {
                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

1. Better than multiply0, since task is divided equally by threads
2. 1st thread processes 0th, 16th, 32nd .. element,
3. 2nd thread processes 1st, 17th, 33r, .. element, so on
4. Big performance problem: Accesses columns in inner loop

#2. Analysis of multiply1

Elapsed Time: 126.668s

CPU

IPC:	0.511
SP GFLOPS:	0.006
DP GFLOPS:	0.137
x87 GFLOPS:	0.002
Average CPU Frequency:	1.5 GHz

GPU

Time:	2.1% (2.661s) of Elapsed time
IPC Rate:	1.299

Logical Core Utilization: 95.1% (7.610 out of 8)

Microarchitecture Usage: 25.4% of Pipeline Slots

Memory Bound: 31.2% of Pipeline Slots

Vectorization: 0.7% of Packed FP Operations

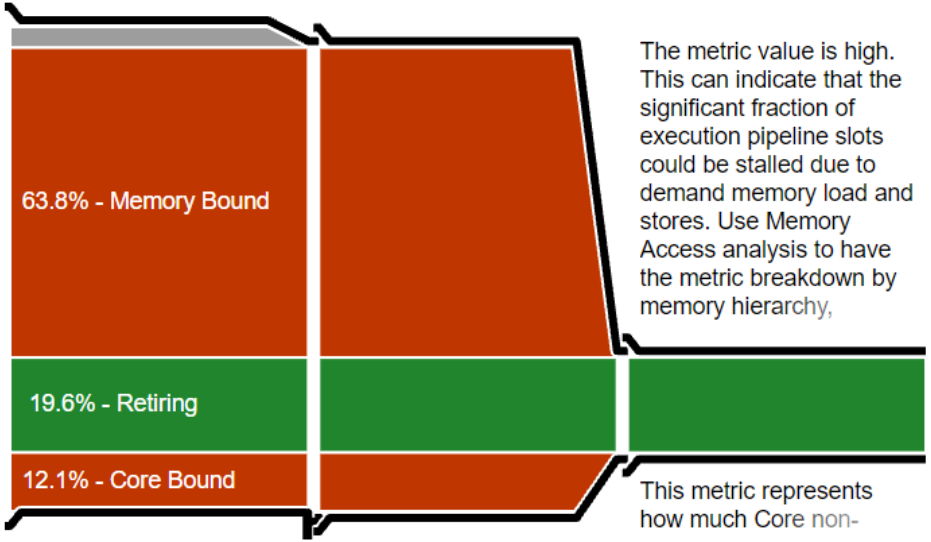
GPU Active Time: 2.1%

- Sequentially accessing a column causes cache thrashing and increases memory bandwidth
- Code is memory bound, has low IPC (conversely high CPI rate)

#2. Analysis of multiply1 (Microarchitectural exploration)

Elapsed Time: 61.004s

Clockticks:	379,994,300,000	
Instructions Retired:	147,238,600,000	
CPI Rate:	2.581	
MUX Reliability:	0.979	
Retiring:	19.6%	of Pipeline Slots
Front-End Bound:	4.3%	of Pipeline Slots
Bad Speculation:	0.3%	of Pipeline Slots
Back-End Bound:	75.9%	of Pipeline Slots
Memory Bound:	63.8%	of Pipeline Slots
L1 Bound:	7.2%	of Clockticks
L2 Bound:	2.6%	of Clockticks
L3 Bound:	16.6%	of Clockticks
Contested Accesses:	0.2%	of Clockticks
Data Sharing:	0.2%	of Clockticks
L3 Latency:	3.6%	of Clockticks
SQ Full:	0.0%	of Clockticks
DRAM Bound:	49.8%	of Clockticks
Memory Bandwidth:	22.8%	of Clockticks
Memory Latency:	63.8%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	12.1%	of Pipeline Slots



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,

This metric represents how much Core non-

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.

#3. Loop interchange (multiply2, USE_THR)

```
void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE
c[][NUM], TYPE t[][NUM])
{
    int i,j,k;

// This implementation interchanges the 'j' and 'k' loop iterations.
// The loop interchange technique removes the bottleneck caused by the
// inefficient memory access pattern in the 'multiply1' function.
    for(i=tidx; i<msize; i=i+numt) {
        for(k=0; k<msize; k++) {
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

1. Loop interchange – row wise access
2. Avoids stride access, cache misses and memory pressure
3. Improves performance

#3. Analysis multiply2

Elapsed Time: 20.816s

CPU

IPC:	1.265
SP GFLOPS:	0.002
DP GFLOPS:	0.829
x87 GFLOPS:	0.002
Average CPU Frequency:	1.9 GHz

GPU

Time:	1.0% (0.214s) of Elapsed time
IPC Rate:	1.212

Logical Core Utilization: 97.5% (7.803 out of 8)

Microarchitecture Usage: 61.2% of Pipeline Slots

Memory Bound: 1.6% of Pipeline Slots

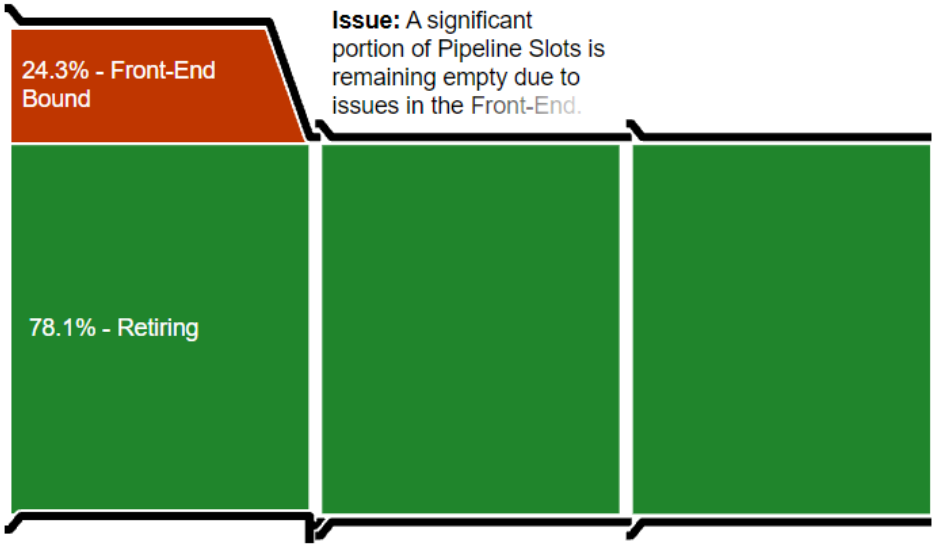
Vectorization: 0.0% of Packed FP Operations

GPU Active Time: 1.0%

#3. Microarchitectural analysis of multiply2

Elapsed Time: 15.238s

Clockticks:	196,007,800,000	
Instructions Retired:	318,844,700,000	
CPI Rate:	0.615	
MUX Reliability:	0.985	
Retiring:	78.1%	of Pipeline Slots
Light Operations:	77.5%	of Pipeline Slots
FP Arithmetic:	5.4%	of uOps
Memory Operations:	42.1%	of Pipeline Slots
Fused Instructions:	2.2%	of Pipeline Slots
Non Fused Branches:	2.1%	of Pipeline Slots
Nop Instructions:	0.0%	of Pipeline Slots
Other:	25.7%	of Pipeline Slots
Heavy Operations:	0.6%	of Pipeline Slots
Front-End Bound:	24.3%	of Pipeline Slots
Front-End Latency:	3.0%	of Pipeline Slots
Front-End Bandwidth:	21.2%	of Pipeline Slots
Front-End Bandwidth MITE:	26.5%	of Pipeline Slots
Decoder-0 Alone:	16.7%	of Clockticks
Front-End Bandwidth DSB:	0.0%	of Pipeline Slots
Front-End Bandwidth LSD:	0.0%	of Pipeline Slots
(Info) DSB Coverage:	0.0%	
(Info) LSD Coverage:	0.0%	
(Info) DSB Misses:	100.0%	of Pipeline Slots



Issue: A significant portion of Pipeline Slots is remaining empty due to issues in the Front-End.

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.

1. DSB (Decoded Streaming Buffer) miss is high

#3. Turn on compiler optimization

matrix Property Pages

Configuration: Active(Debug) Platform: Active(x64) Configuration Manager...

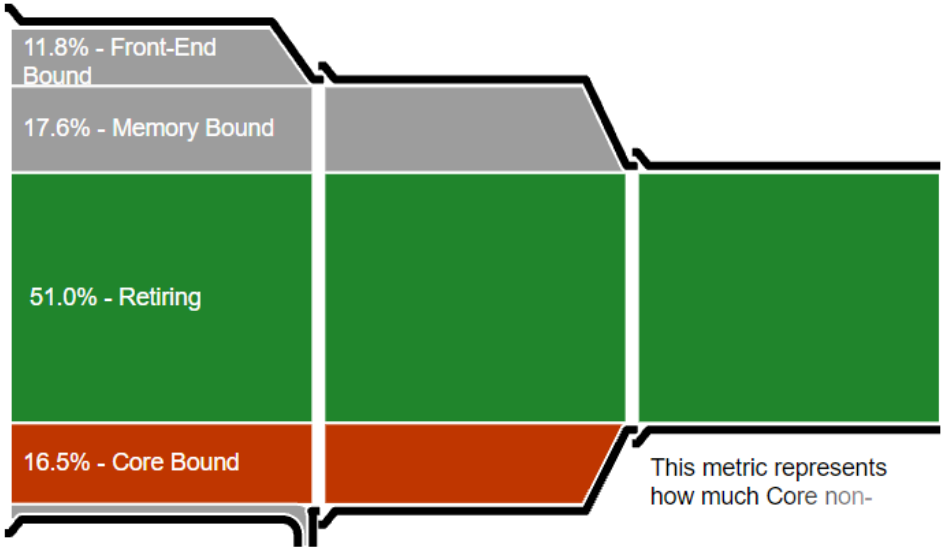
- Configuration Properties
  - General
  - Advanced
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation

Optimization	**Maximum Optimization (Favor Speed) (/O2)**
Inline Function Expansion	**Only __inline (/Ob1)**
Enable Intrinsic Functions	No
Favor Size Or Speed	Neither
Omit Frame Pointers	
Enable Fiber-Safe Optimizations	No
Whole Program Optimization	No

#3. Microarchitectural analysis of multiply2 (After compiler optimization)

Elapsed Time: 3.561s

Clockticks:	36,434,400,000	
Instructions Retired:	41,258,500,000	
CPI Rate:	0.883	
MUX Reliability:	0.999	
Retiring:	51.0%	of Pipeline Slots
Front-End Bound:	11.8%	of Pipeline Slots
Bad Speculation:	3.1%	of Pipeline Slots
Back-End Bound:	34.1%	of Pipeline Slots
Memory Bound:	17.6%	of Pipeline Slots
Core Bound:	16.5%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	24.3%	of Clockticks
Cycles of 0 Ports Utilized:	12.4%	of Clockticks
Cycles of 1 Port Utilized:	8.8%	of Clockticks
Cycles of 2 Ports Utilized:	12.7%	of Clockticks
Cycles of 3+ Ports Utilized:	64.7%	of Clockticks
Vector Capacity Usage (FPU):	25.0%	
Average CPU Frequency:	2.1 GHz	
Total Thread Count:	11	
Paused Time:	0s	



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.

#3. But.. No vectorization

Elapsed Time: 3.984s

CPU

IPC:	0.910
SP GFLOPS:	0.002
DP GFLOPS:	4.159
x87 GFLOPS:	0.005
Average CPU Frequency:	2.0 GHz

GPU

Time:	2.4% (0.096s)	of Elapsed time
IPC Rate:	1.546	

Logical Core Utilization: 107.4% (8.590 out of 8)

Microarchitecture Usage: 44.4% of Pipeline Slots

Retiring:	44.4%	of Pipeline Slots
Front-End Bound:	29.3%	of Pipeline Slots
Bad Speculation:	4.9%	of Pipeline Slots
Back-End Bound:	**21.4%**	**of Pipeline Slots**
Memory Bound:	**10.8%**	**of Pipeline Slots**

Memory Bound: 10.8% of Pipeline Slots

Vectorization: 0.0% of Packed FP Operations

Instruction Mix:

SP FLOPs:	0.0%	of uOps
Packed:	3.5%	from SP FP
Scalar:	96.5%	from SP FP
DP FLOPs:	26.8%	of uOps
Packed:	0.0%	from DP FP
Scalar:	100.0%	from DP FP
x87 FLOPs:	0.0%	of uOps
Non-FP:	73.2%	of uOps
FP Arith/Mem Rd Instr. Ratio:	0.533	
FP Arith/Mem Wr Instr. Ratio:	1.413	

#3. After Vectorization..

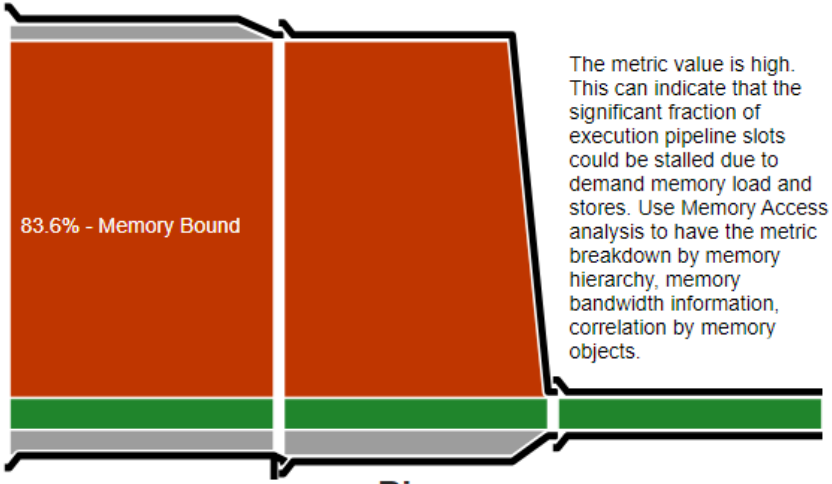
Microarchitecture Exploration Microarchitecture Exploration

INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

Elapsed Time: 2.731s

Clockticks:	37,686,500,000	
Instructions Retired:	3,773,400,000	
CPI Rate:	9.987	
MUX Reliability:	0.923	
Retiring:	7.5%	of Pipeline Slots
Front-End Bound:	3.7%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	89.6%	of Pipeline Slots
Memory Bound:	83.6%	of Pipeline Slots
L1 Bound:	0.0%	of Clockticks
L2 Bound:	1.0%	of Clockticks
L3 Bound:	19.1%	of Clockticks
DRAM Bound:	66.2%	of Clockticks
Memory Bandwidth:	81.8%	of Clockticks
Memory Latency:	11.5%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	6.0%	of Pipeline Slots
Average CPU Frequency:	2.7 GHz	
Total Thread Count:	11	
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.

#4. With USE_OMP (OpenMP)

Change from USE_THR to USE_OMP in the Project Properties page

matrix Property Pages

Configuration: Active(Debug) Platform: Active(x64) Configuration Manager...

- Intel Debugging
- Intel Libraries for oneAPI
- VC++ Directories
- C/C++
  - General
  - General [Intel C++]
  - Debug [Intel C++]
  - Optimization
  - Optimization [Intel C+
  - Preprocessor
  - Code Generation
  - Code Generation [Inte

Preprocessor Definitions	**WIN32;_DEBUG;_CONSOLE;ICC;USE_OMP;%(PreprocessorDe**
Undefine Preprocessor Definitions	
Undefine All Preprocessor Definitions	No
Ignore Standard Include Paths	No
Preprocess to a File	No
Preprocess Suppress Line Numbers	No
Keep Comments	No

#4. With USE_OMP (OpenMP, multiply3)

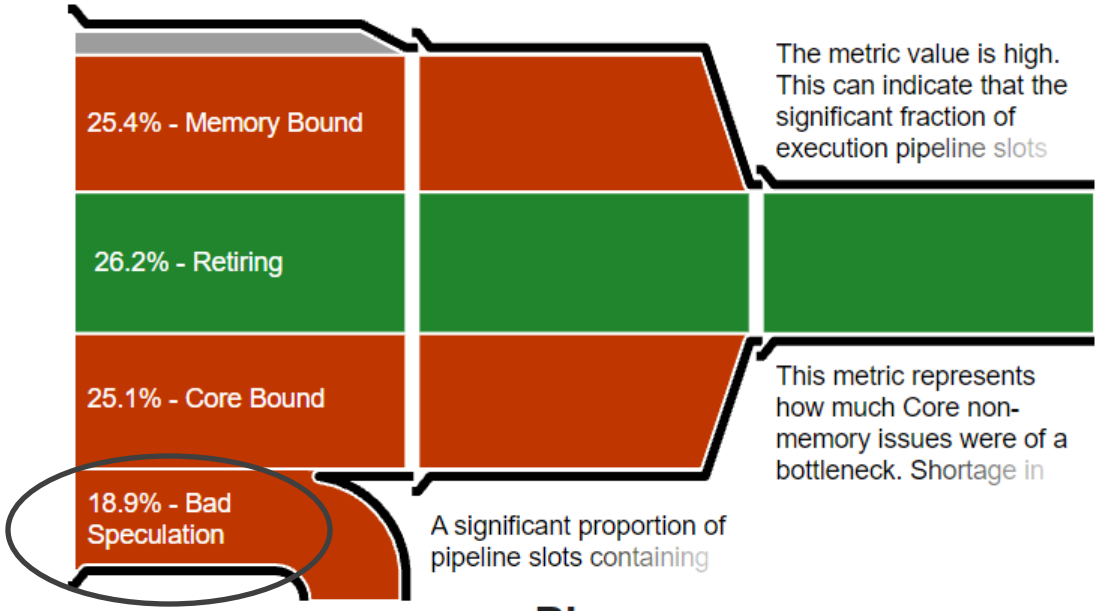
Microarchitecture Exploration Microarchitecture Exploration

Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

Elapsed Time: 4.525s

Clockticks:	42,185,700,000	
Instructions Retired:	22,667,000,000	
CPI Rate:	1.861	
MUX Reliability:	0.966	
Retiring:	26.2%	of Pipeline Slots
Front-End Bound:	4.4%	of Pipeline Slots
Bad Speculation:	18.9%	of Pipeline Slots
Branch Mispredict:	0.0%	of Pipeline Slots
Machine Clears:	18.9%	of Pipeline Slots
Back-End Bound:	50.5%	of Pipeline Slots
Memory Bound:	25.4%	of Pipeline Slots
L1 Bound:	25.0%	of Clockticks
DTLB Overhead:	0.2%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	0.4%	of Clockticks
FB Full:	0.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	9.6%	of Clockticks

#4. With USE_OMP (OpenMP, multiply3)



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

Intel Advisor

Vectorization, Offload modeling, etc

VTune vs. Advisor

VTune

- Hotspot Analysis
- Threading Analysis
- Microarchitecture Exploration
- Memory Access Analysis
- IO Analysis
- GPU Analysis

Advisor

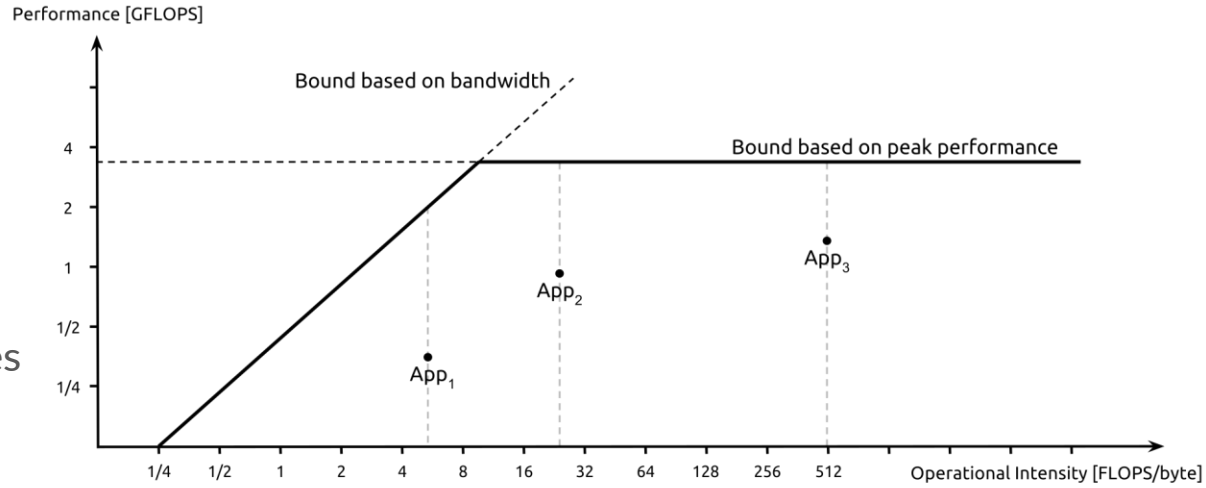
- Vectorization and Code Insights
- CPU/GPU Roofline Insights
- Offload Modeling
- Threading

Heterogeneous Computing

- Parts of program executes across various compute units
  - CPU, GPU, FPGA, ASIC, TPU, etc
- Commonly used frameworks: CUDA, OpenCL, SYCL
- Programmer's problems:
  - How to identify the heterogeneous compute units available
  - How to ascertain which parts of program to offload to compute units, to achieve the best performance?
- Intel Advisor has 'Offload Modeling'

Roofline Model Analysis

- Plots Machine Peak performance against Arithmetic intensity
- Arithmetic Intensity is ratio of no. of operations per byte read from memory (FLOPS/byte)
- Helps in
  - Identifying machine limitations
  - Optimization opportunities in the program
  - Compare performance across various architectures



Intel Advisor – Roofline Modeling

- Stride program
  - Scalar and vector loops
  - Row and column traversal

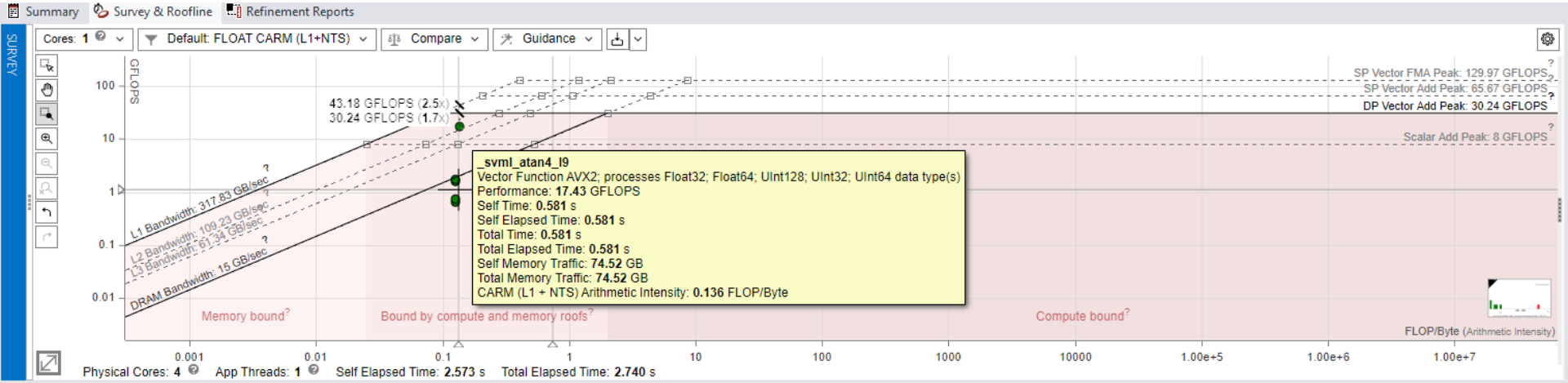
- Roofline Analysis
  - L1, L2, L3, DRAM Bandwidth
  - Loop location and analysis

Intel Advisor – Vectorization

- Before and after vectorization
  - Comparison of speed
  - Before: 27.48 s
  - GFLOPS 0.044

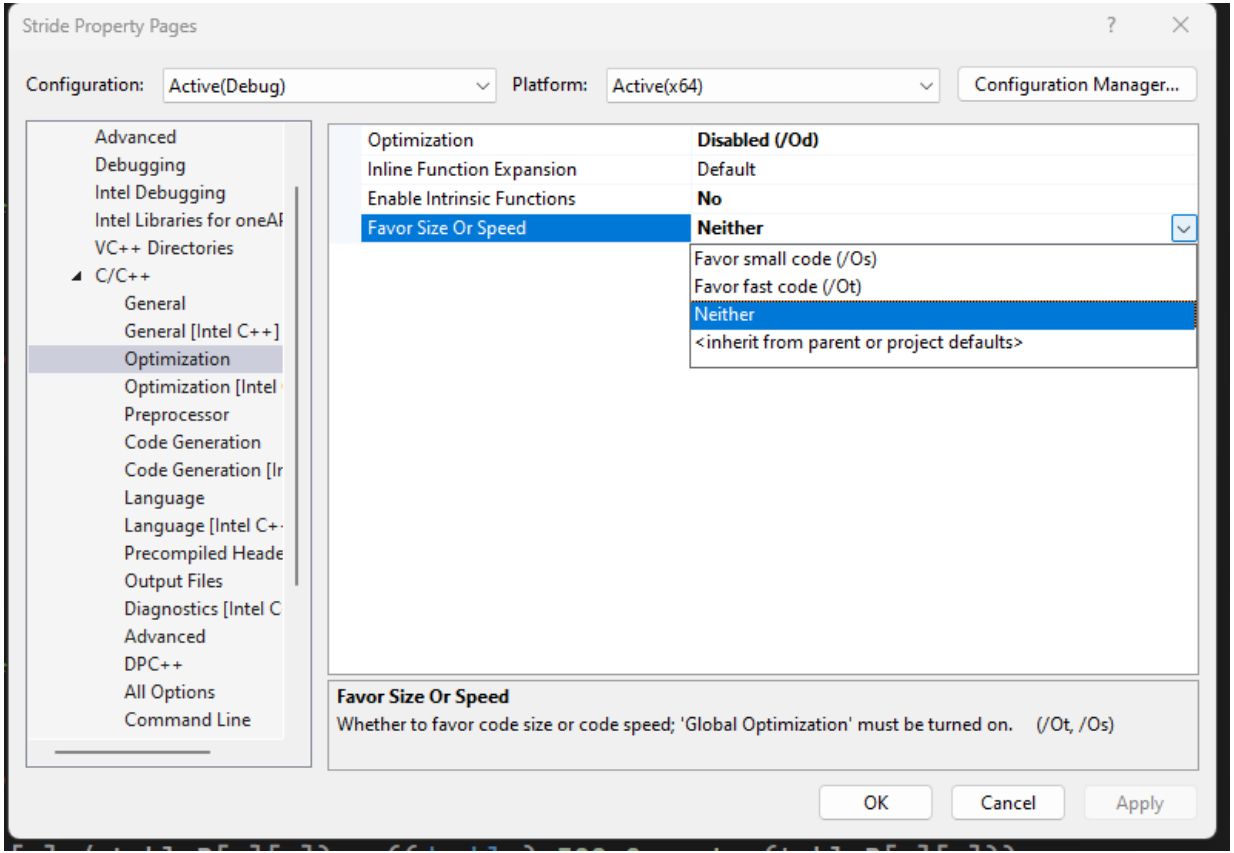
- Enable/Disable vectorization
  - Project Properties: /QxHOST /Ot

Intel Advisor – Roofline Model (Stride_and_MAP)



1. GFLOPS quite low
2. Low arithmetic intensity
3. 5 loops exhibit different Arithmetic intensity and GFLOPS

Enabling/Disabling Vectorization



To enable Vectorization:

- Optimization -> /O2
- Favor Size or Speed -> /Ot

Refer to readme.txt for other options

Performance Analysis Problems

Analysis

- Analyse
  - CPI rate
  - Cache miss
  - Arithmetic Intensity
  - Branch prediction
  - Hotspots

- Scenario:
  - Stride
  - Vectorization
  - OpenMP enabling
  - Loop Unrolling effect

Case #1: Auto-Vectorization effects

```
void multiply3(int msize, int tidx, int numt, TYPE a[][NUM], TYPE
b[][NUM], TYPE c[][NUM], TYPE t[][NUM])
{
    int i, j, k;

// Add compile option for vectorization report Windows: /Qvec-report3
Linux -vec-report3
    for(i=tidx; i<msize; i=i+numt) {
        for(k=0; k<msize; k++) {
#pragma ivdep
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

#1. Analysis & Comments

Attributes	No vectorization (r013ue)	With auto-vectorization (AVX2) r0014ue	Comments
Execution time	12.33 s	2.894 s	**Perf. Boost is due to SIMD**
CPI Rate	0.532	0.899	**SIMD instructions can take more cycles than scalar ones**
Instructions Retired	318,820,000,000	41,188,200,000	**SIMD instructions process more data per instruction**
CPU Front End usage	8.8 %	13.9 %	No significant change
Bad Speculation	1.8 %	0.0 %	No significant change
Pipeline slots utilization	87.3 %	52.5 %	**Indicates waiting for data**
Memory bound	0.4 %	17.3 %	**SIMD needs more data bandwidth**
Core bound	1.8 %	17.1 %	**SIMD lanes increases usage**

Case #2: Row vs. Column access

```
// Before -> Column access
for(i=tidx; i<msize; i=i+numt) {
    for(j=0; j<msize; j++) {
        for(k=0; k<msize; k++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}

// After -> Row access
for(i=tidx; i<msize; i=i+numt) {
    for(k=0; k<msize; k++) {
        for(j=0; j<msize; j++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

#2. Analysis & Comments

Attributes	Column Access (r017ue)	Row access (r0018ue)	Comments
Execution time	54.485 s	11. 676 s	**Accessing a column in a hot loop causes lot of cache (cold) misses**
CPI Rate	2.487	0.545	**Clocks not wasted waiting for data from memory**
CPU Front End usage	4.2 %	8.7 %	No significant change
Bad Speculation	0.4 %	0.4 %	No significant change
Pipeline slots utilization	19.9 %	90.0 %	**Better utilization, also indicates vectorization could help here**
Memory bound	64.4 %	0.2 %	**Significant reduction in memory access (for row access) due to caching**
Core bound	11.1 %	0.8 %	No significant change

Case #3. Loop Unrolling effects

```
for (i = tidx; i < msize; i = i + numt) {
        for (k = 0; k < msize; k++) {
            for (j = 0; j < msize; j+=8) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
                c[i][j+1] = c[i][j+1] + a[i][k] * b[k][j+1];
                c[i][j+2] = c[i][j+2] + a[i][k] * b[k][j+2];
                c[i][j+3] = c[i][j+3] + a[i][k] * b[k][j+3];

                c[i][j+4] = c[i][j+4] + a[i][k] * b[k][j+4];
                c[i][j+5] = c[i][j+5] + a[i][k] * b[k][j+5];
                c[i][j+6] = c[i][j+6] + a[i][k] * b[k][j+6];
                c[i][j+7] = c[i][j+7] + a[i][k] * b[k][j+7];
            }
        }
    }
```

#3. Analysis & Comments

Attributes	Row access (r0018ue)	Row access + loop unrolling (r0019ue)	Comments
Execution time	11. 676 s	9.941 s	**Improvement in performance due to loop unrolling**
CPI Rate	0.545	0.498	Marginal change
CPU Front End usage	8.7 %	2.7 %	No significant change
Bad Speculation	0.4 %	3.7 %	No significant change
Pipeline slots utilization	90.0 %	94.2 %	Better utilization, also indicates vectorization could help here
Memory bound	0.2 %	0.0 %	No significant change
Core bound	0.8 %	0.0 %	No significant change
Branch instructions	17,318,789,664	896,020,160	**~19x reduction**

Case #4. Branch misprediction (induced)

```
int direction[NUM];

    // Randomize branch direction
    for (i = 0; i < msize; i++) {
        direction[i] = rand() % 2;
    }

    for (i = tidx; i < msize; i = i + numt) {
        for (k = 0; k < msize; k++) {
            for (j = 0; j < msize; j++) {
                if (direction[j] == 1)
                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
                else
                    c[i][j] = c[i][j] + a[i][k] * b[k][j] + i;
            }
        }
    }
```

#4. Analysis & Comments

Attributes	Row access (r0018ue)	Row access + branch misprediction (r020ue)	Comments
Execution time	11. 676 s	27.90 s	**Less performance due to speculation failure**
CPI Rate	0.545	1.009	**CPI drops due to pipeline stalls**
CPU Front End usage	8.7 %	12.0 %	No significant change
Bad Speculation	0.4 %	37.8 %	**Big impact due to mis-predicted branches**
Pipeline slots utilization	90.0 %	49.1 %	**Pipeline underutilized due to stalls**
Memory bound	0.2 %	0.3 %	No significant change
Core bound	0.8 %	0.8 %	No significant change
Branch instructions	17,318,789,664	29,069,454,048	**~2x increase**

Case #5. Roofline impact of Vectorization on Loop

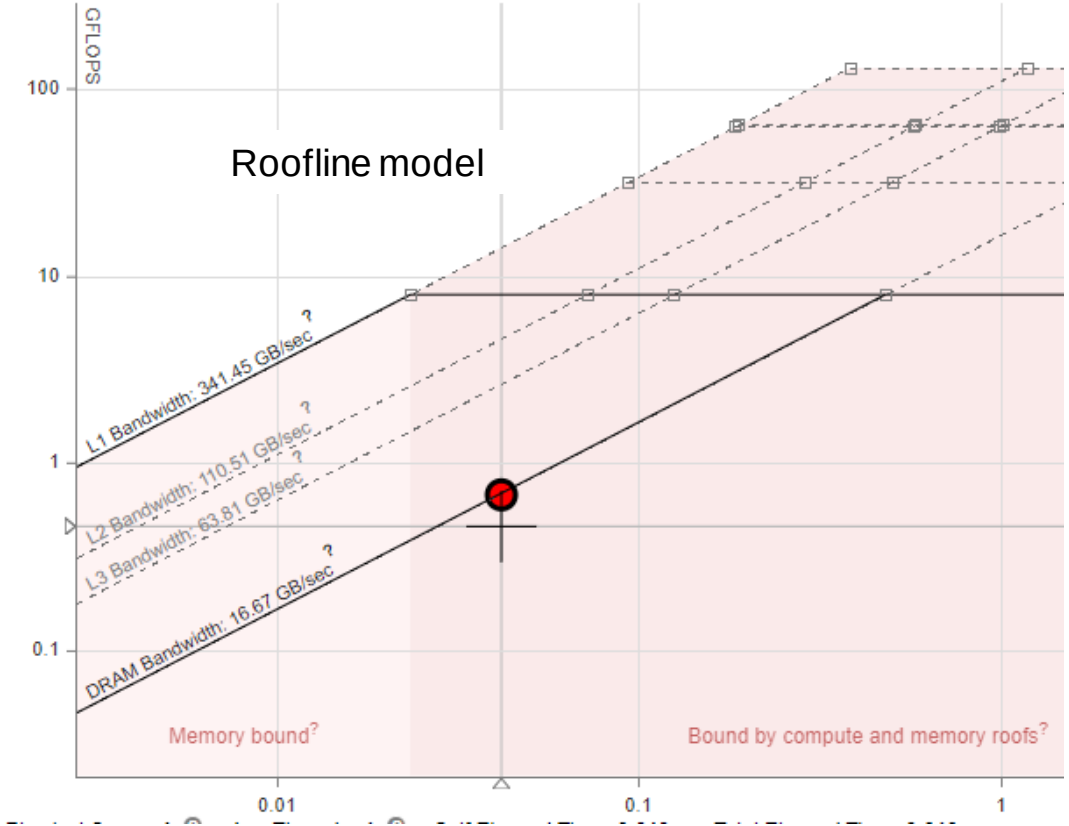
```
#define REPEAT 3000
#define SIZE 70000

for (int r = 0; r < REPEAT; r++)
{
    for (int i = 1; i < SIZE; i++)
    {
        doublesArray[i] = i * 42.0 + doublesArray[i];
    }
}
```

What are the loop dependences?

#5. Before Vectorization

```
doublesArray[i] = i * 42.0 + doublesArray[i];
```



Calculated:
Approximately 1 arithmetic ops for 16 bytes

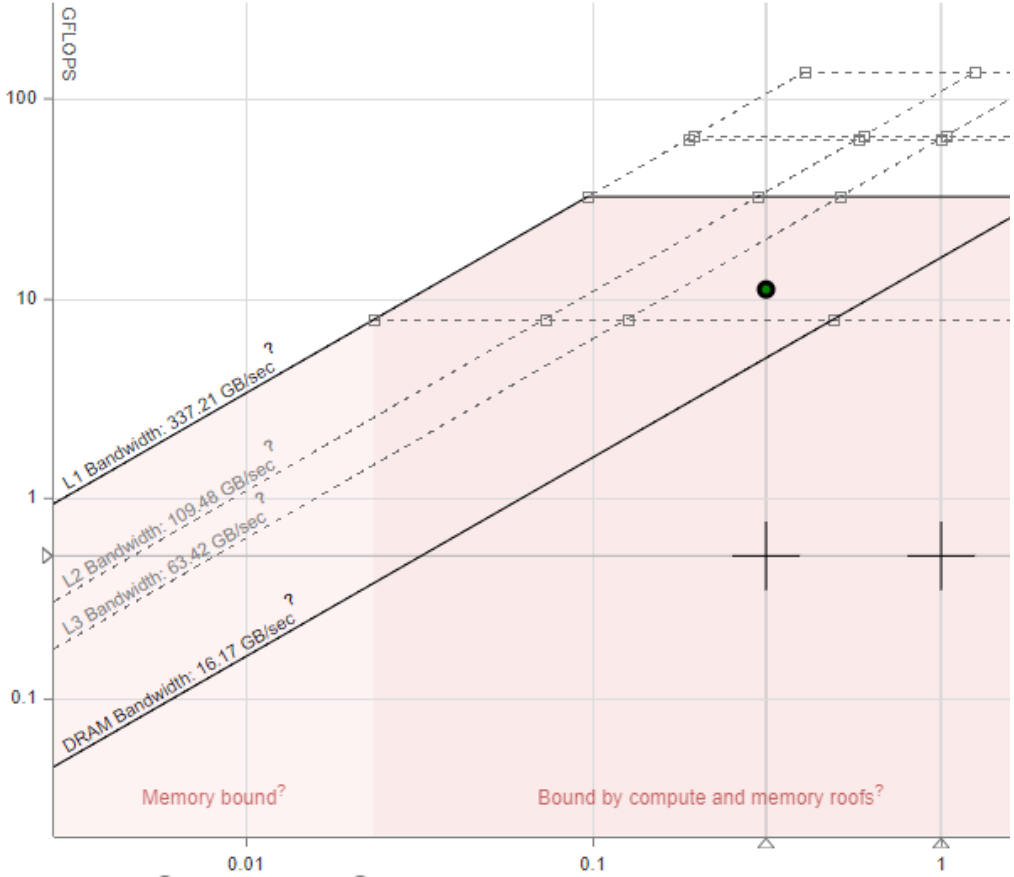
So, arithmetic Intensity 1/16 = 0.0625

Observed:

Arithmetic intensity = 0.042 GFLOPS/byte
Performance = 0.685 GFLOPS

What would be the impact of vectorization?

#5. After Vectorization



Before:

- Arithmetic intensity = 0.042 GFLOPS/byte
- Performance = 0.685 GFLOPS

After:

- Arithmetic intensity = 0.313 GFLOP/byte
- Performance = 11.2 GFLOPs

Big improvement in performance, since there is no loop carried dependence!

Case #6. Roofline impact of Vectorization on Loop

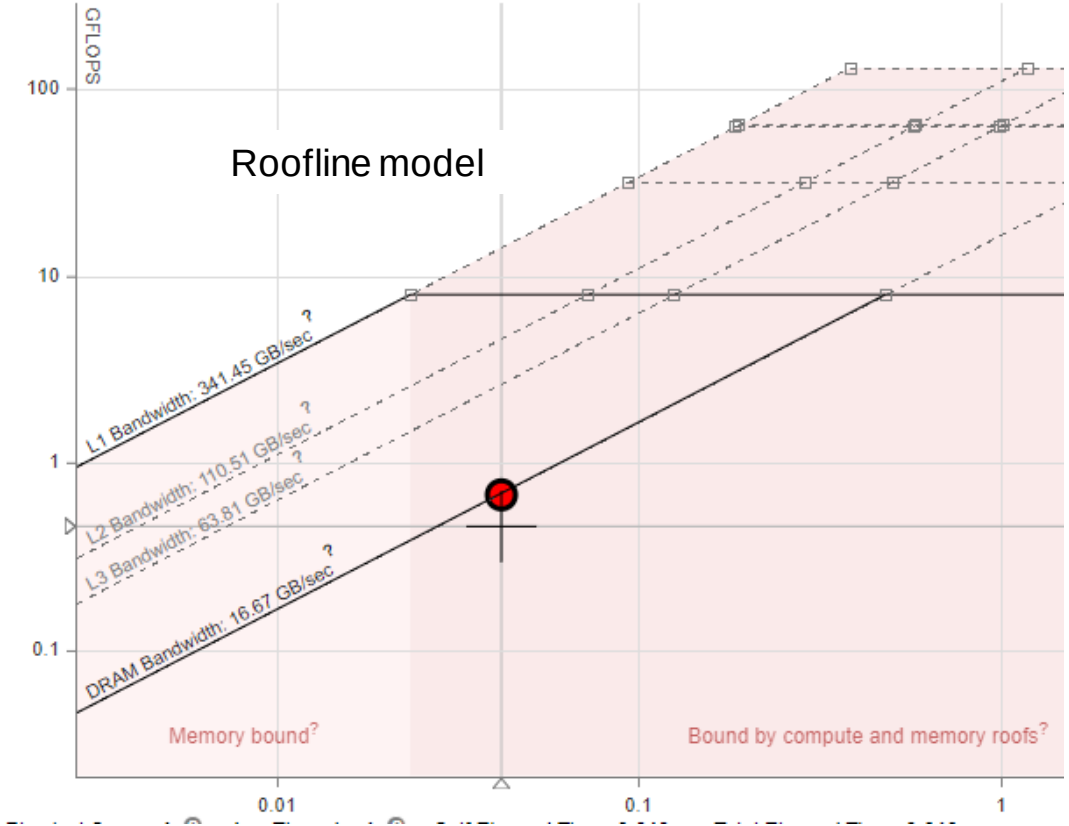
```
#define REPEAT 3000
#define SIZE 70000

for (int r = 0; r < REPEAT; r++)
{
    for (int i = 1; i < SIZE; i++)
    {
        doublesArray[i] = i * 42.0 + doublesArray[i-1];
    }
}
```

What are the loop dependences?

#6. Before Vectorization

```
doublesArray[i] = i * 42.0 + doublesArray[i-1];
```



Calculated:
Approximately 1 arithmetic ops for 16 bytes

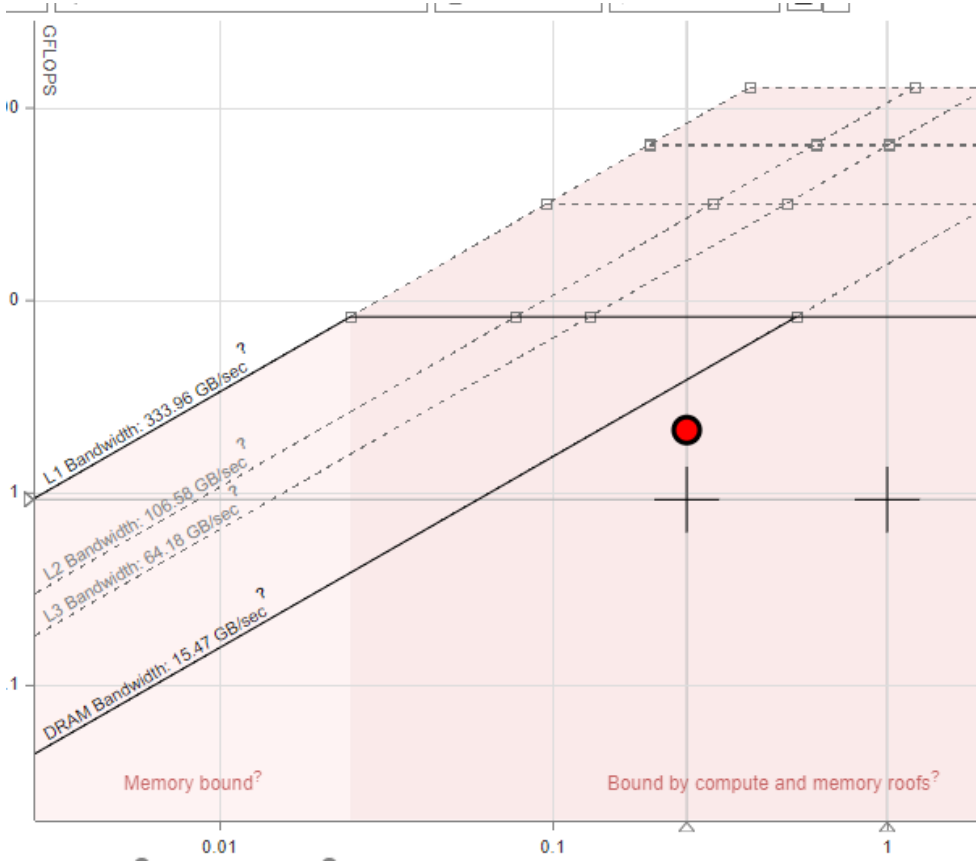
So, arithmetic Intensity 1/16 = 0.0625

Observed:

Arithmetic intensity = 0.042 GFLOPS/byte
Performance = 0.689 GFLOPS

What would be the impact of vectorization?

#6. After Vectorization



Before:

- Arithmetic intensity = 0.042 GFLOPS/byte
- Performance = 0.689 GFLOPS

After:

- Arithmetic intensity = 0.25 GFLOP/byte
- Performance = 2.138 GFLOPs

Loop carried dependence has impacted the performance gain from Vectorization