Optimization and GPU Offloading Workflow with Intel oneAPI

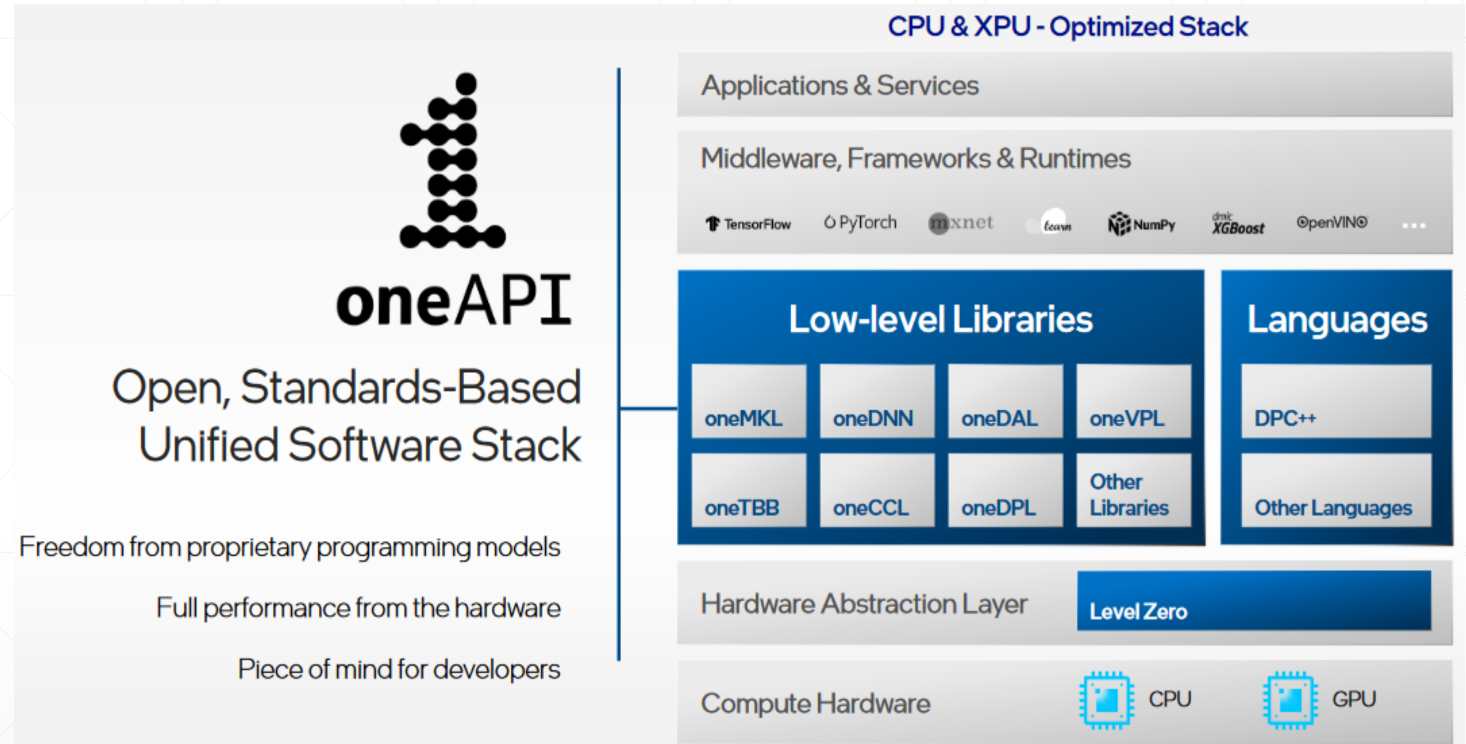
oneAPI – 가속 컴퓨팅을 개발하기 위한 스마트한 방식

2021. 10. 28.

MOASYS

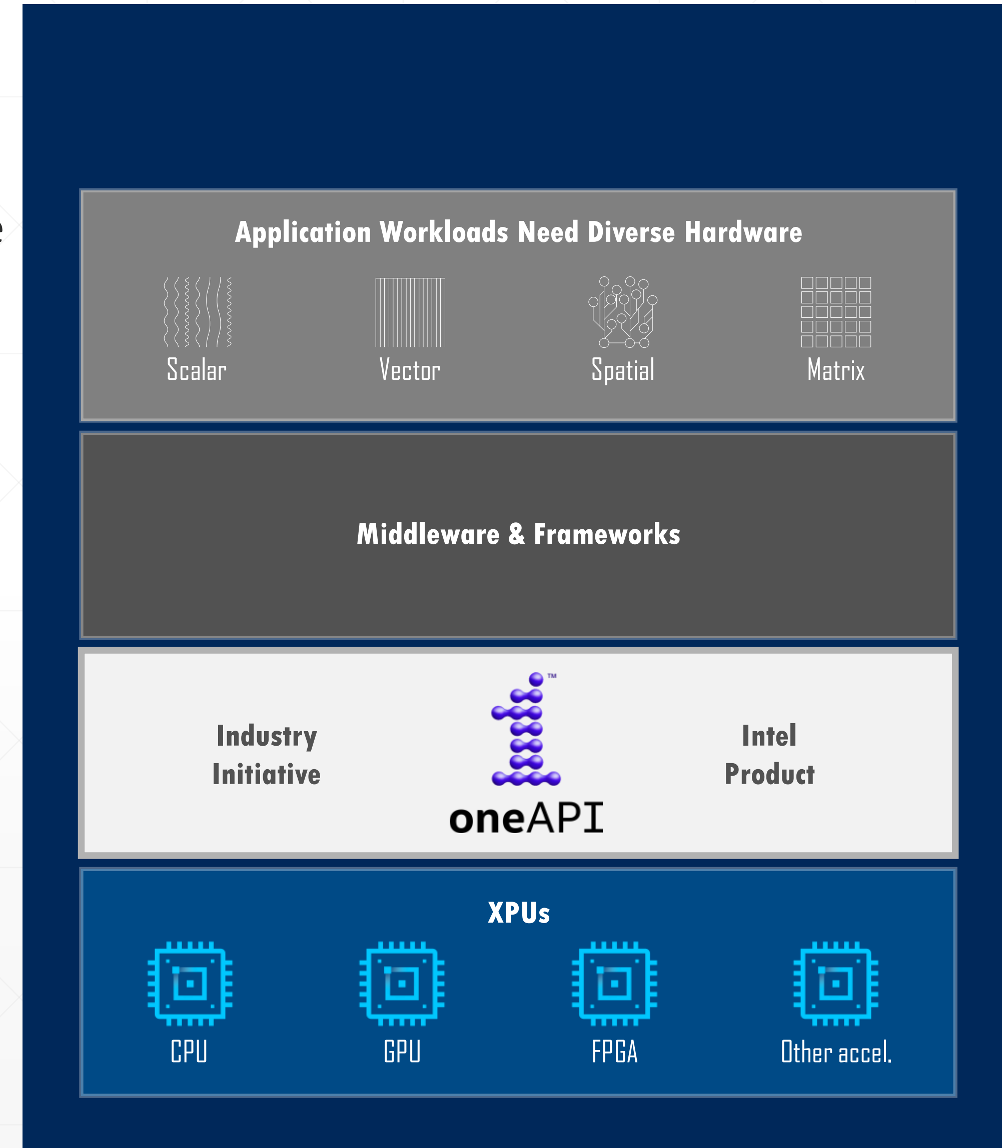
Content

- oneAPI Compilers and Analytics Tool
- Intel Optimization Workflow:
  I. Compiler Optimization Report
  II. Application Performance Snapshot
  III. Memory Access Analysis
  IV. CPU Roofline Analysis
  V. GPU Offload Modeling
  VI. GPU Roofline Analysis
  VII. Minimization of Analysis Overhead
- Conclusion



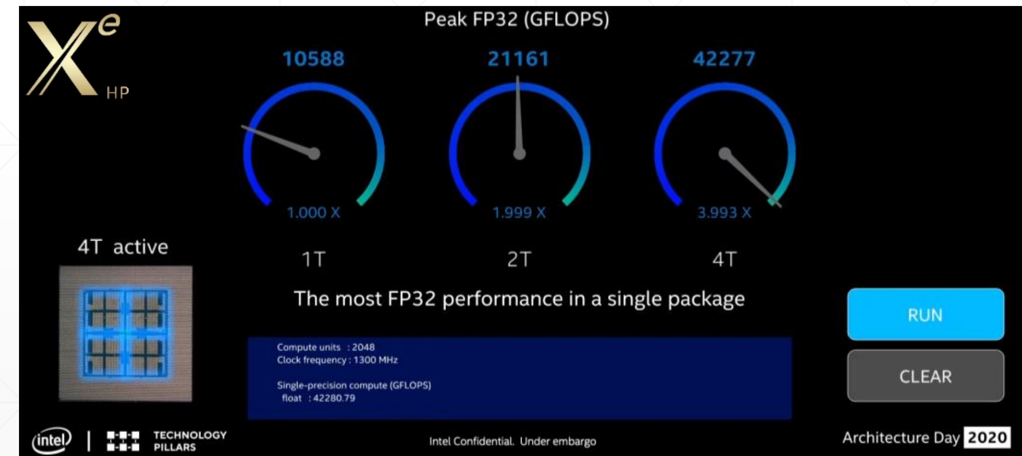
oneAPI: One Programming Model for Multiple Architectures and Vendors

- Freedom to Make Your Best Choice
  - Choose the best accelerated technology the software doesn't decide for you
- Realize all the Hardware Value
  - Performance across CPU, GPUs, FPGAs, and other accelerators
- Develop & Deploy Software with Peace of Mind
  - Open industry standards provide a safe, clear path to the future
  - Compatible with existing languages and programming models including C++, Python, SYCL, OpenMP, Fortran, and MPI



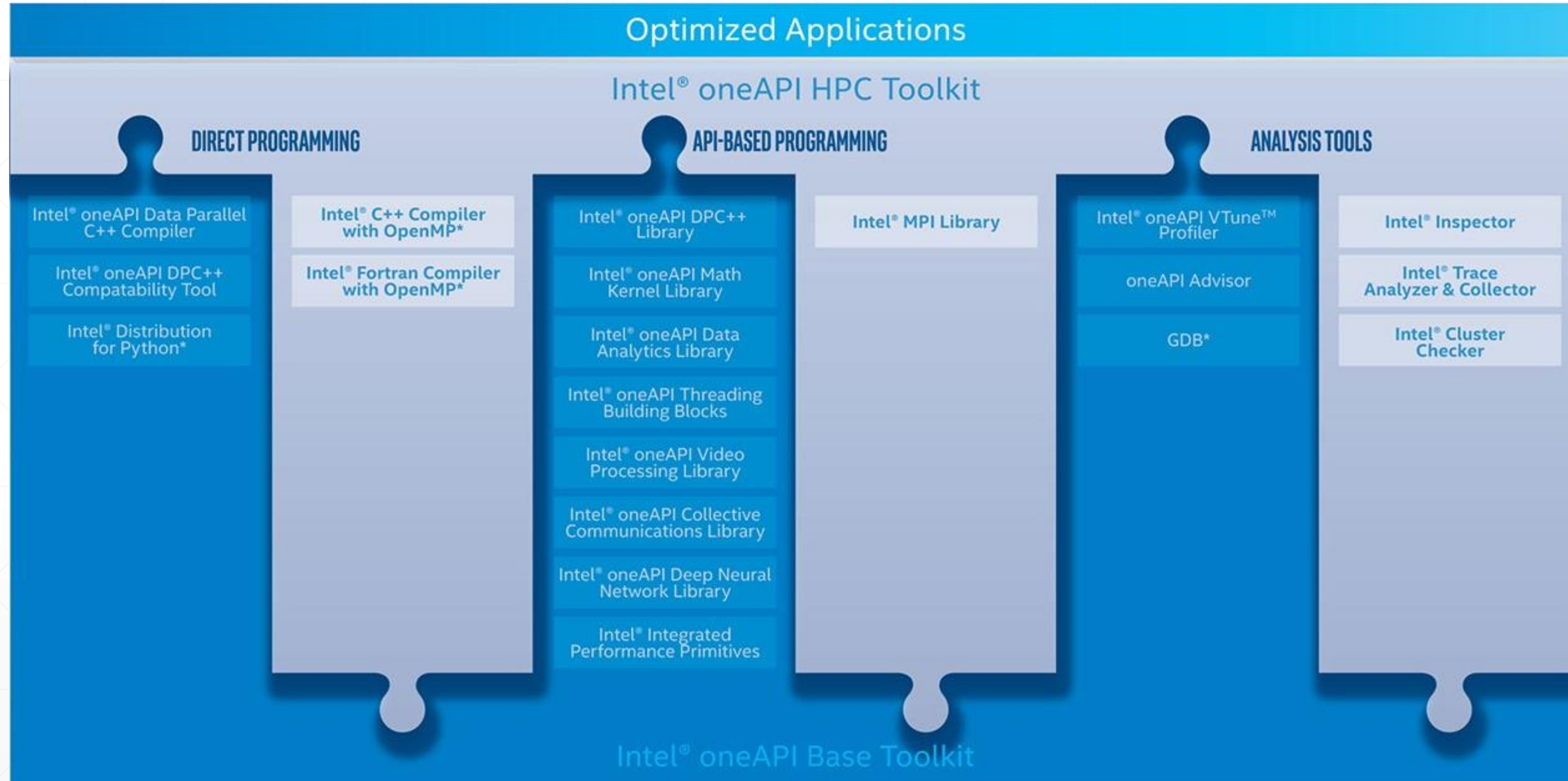
Intel Xe Architecture: Building the Foundation for Exascale Computing

µArchitecture		Packaging		Process
Xe HPC	PONTE VECCHIO	FOVEROS CO-EMIB	BASE TILE	Intel 10nm SuperFin
			COMPUTE TILE	Intel Next Gen & External
			RAMBO CACHE TILE	Intel 10nm Enhanced SuperFin
			Xe LINK I/O TILE	External
Xe HP	TBA	EMIB		Intel 10nm Enhanced SuperFin
Xe HPG	TBA	STANDARD		External
Xe LP	SG1 DG1 TIGER LAKE	STANDARD		Intel 10nm SuperFin



- Intel architecture day 2020:
  - https://newsroom.intel.com/wp-content/uploads/sites/11/2020/08/Intel-Architecture-Day-2020-Presentation-Slides.pdf
  - Xe-HP can scale up to 4 tiles with a peak FP32 performance of 42 Tflops

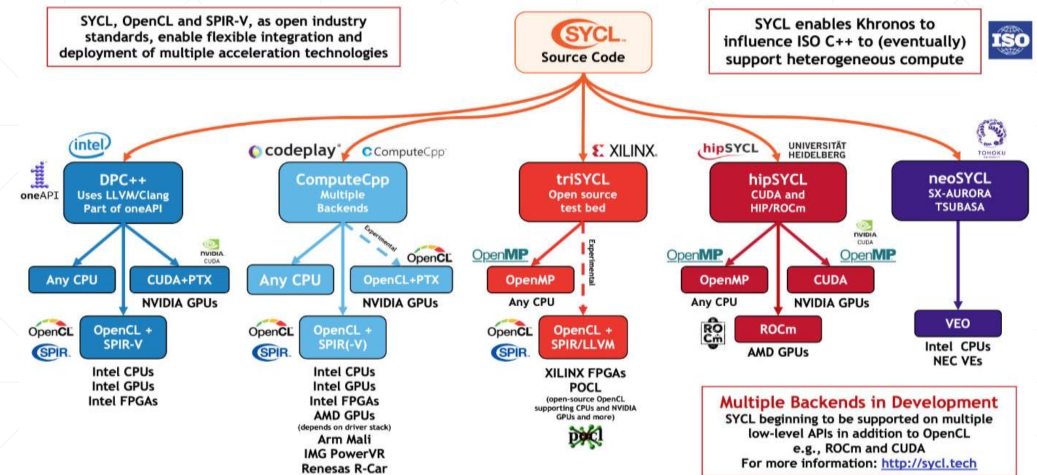
A New Era of Accelerated Computing



- Roofline Analysis: get insights about performance headroom against hardware limitations.
- Offload Advisor: get your code ready for efficient GPU offload before buying the hardware.

Heterogeneous Computing with Intel Compilers

Compiles	Targets	OpenMP	OpenMP Offload	Toolkits
icc/icpc	CPU	Yes	No	HPC
ifort	CPU	Yes	No	HPC
icx	CPU/GPU	Yes	Yes	Base
ifx	CPU/GPU	Yes	Yes	Base
dpcpp	CPU/GPU/FPGA	Yes	Yes	Base
intel-llvm	CPU/GPU	Yes	Yes	Open



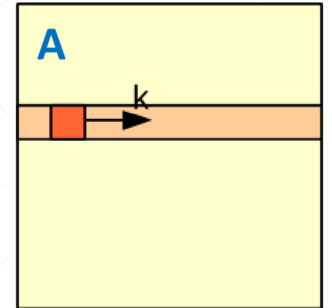
- icc/icpc/ifort: classic Intel HPC compilers
- icx/ifx: next generation compilers based on Clang/LLVM with Intel proprietary technologies
  - Support for OpenMP offloading to Intel GPUs
- dpcpp: Intel implementation of SYCL standard
  - https://www.khronos.org/sycl/
  - SYCL = High level abstraction C++ and OpenCL runtime to target heterogenous architectures.
- intel-llvm: open-source development version of dpcpp
  - https://github.com/intel/llvm
  - Experimental support for NVIDIA devices using CUDA PTX backend

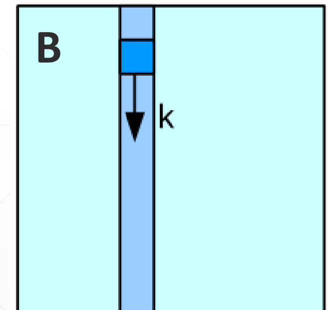
Optimization Workflow I: Compiler Optimization Report

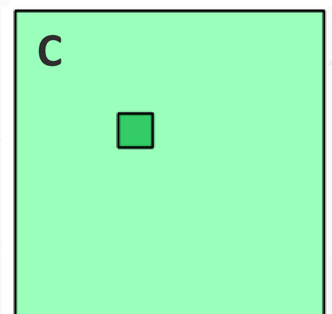
- Use compiler option -qopt-report=5
  - Detailed information regarding optimizations done by Intel compilers (-O2)

```
LOOP BEGIN at matmul_baseline.c(89,5)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at matmul_baseline.c(90,9)
      remark #15542: loop was not vectorized: inner loop was already vectorized

      LOOP BEGIN at matmul_baseline.c(92,13)
         remark #15388: vectorization support: reference A[i*p+k] has aligned access   [ matmul_baseline.c(93,29) ]
         remark #15328: vectorization support: non-unit strided load was emulated for the variable <B[k*n+j]>, stride
                                               is unknown to compiler [ matmul_baseline.c(93,40) ]
         remark #15305: vectorization support: vector length 4
         remark #15309: vectorization support: normalized vectorization overhead 0.250
         remark #15355: vectorization support: *(C+(i*n+j)*4) is float type reduction   [ matmul_baseline.c(93,17) ]
         remark #15300: LOOP WAS VECTORIZED
         remark #15442: entire loop may be executed in remainder
         remark #15448: unmasked aligned unit stride loads: 1
         remark #15452: unmasked strided loads: 1
         remark #15475: --- begin vector cost summary ---
         remark #15476: scalar cost: 11
         remark #15477: vector cost: 10.000
         remark #15478: estimated potential speedup: 1.090
         remark #15488: --- end vector cost summary ---
      LOOP END

    LOOP END

LOOP END
```

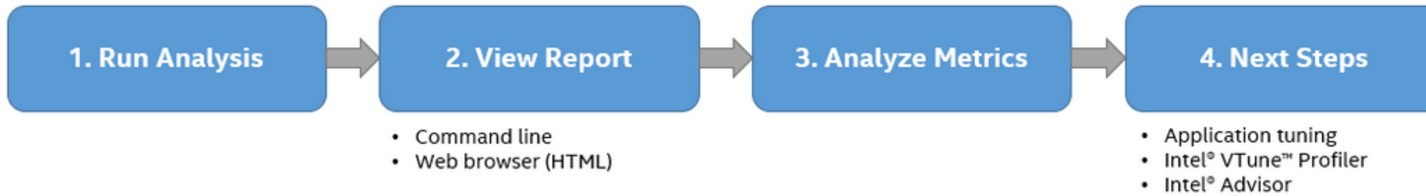
```
void mat_mul(float *A, float *B, float *C,
                   int m, int n, int p) {
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
        for (int k = 0; k < p; k++)
            C[i*n+j] += A[i*p+k] * B[k*n+j];
```







Optimization Workflow II : Application Performance Snapshot (APS)



- Command-line interface to generate HTML report: easy to use, low overhead, and high scalability
- For shared memory applications:

```
aps <my app> <app parameters>
```

- For MPI applications:

```
<mpi launcher> <mpi parameters> aps <my app> <app parameters>
```

- HTML report: *aps_result_<date>*

Optimization Workflow II : Application Performance Snapshot (APS)

Application Performance Snapshot

Application: *matmul_baseline.x*
Report creation date: *2021-10-24 19:39:38*
HW Platform: *Intel(R) Xeon(R) Processor code named Cascadelake*
Frequency: *2.99 GHz*
Logical Core Count per node: *32*
Collector type: *Driverless Perf per-process counting*

104.03 s Elapsed Time

0.09 IPC Rate

0.17 SP GFLOPS

0 DP GFLOPS

3.40 GHz Average CPU Frequency

Your application might underutilize the available logical CPU cores
because of insufficient parallel work, blocking on synchronization, or too much I/O. Perform function or source line-level profiling with tools like Intel® VTune™ Profiler to discover why the CPU is underutilized.

	Current run	Target	Tuning Potential
Physical Core Utilization	6.1%	>80%	
Memory Stalls	93%	<20%	
Vectorization			

Memory Stalls, % of Pipeline Slots

93%

0% 20% 100%

— better ▭ red flag zone ■ tuning potential

This metric indicates how memory subsystem issues affect the performance. It measures a fraction of slots where pipeline could be stalled due to demand load or store instructions.

The metric value can indicate that a significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. See the second level metrics to define if the application is cache- or DRAM-bound and the NUMA efficiency. Use Intel® VTune™ Profiler Memory Access analysis to review a detailed metric breakdown by memory hierarchy, memory bandwidth information, and correlation by memory objects.

To run **memory-access** analysis:

```
vtune -collect memory-access -data-limit=0 -r <results dir> -- matmul_baseline.x [arguments]
```

High demand of load/store

Physical Core Utilization
6.1%

Average Physical Core Utilization
0.98 out of 16 Physical Cores

Inefficient core utilization

Memory Stalls
93% of Pipeline Slots

Cache Stalls
0.2% of Cycles

DRAM Stalls
93.3% of Cycles

Average DRAM Bandwidth
N/A

NUMA
0% of Remote Accesses

Vectorization
99.8%

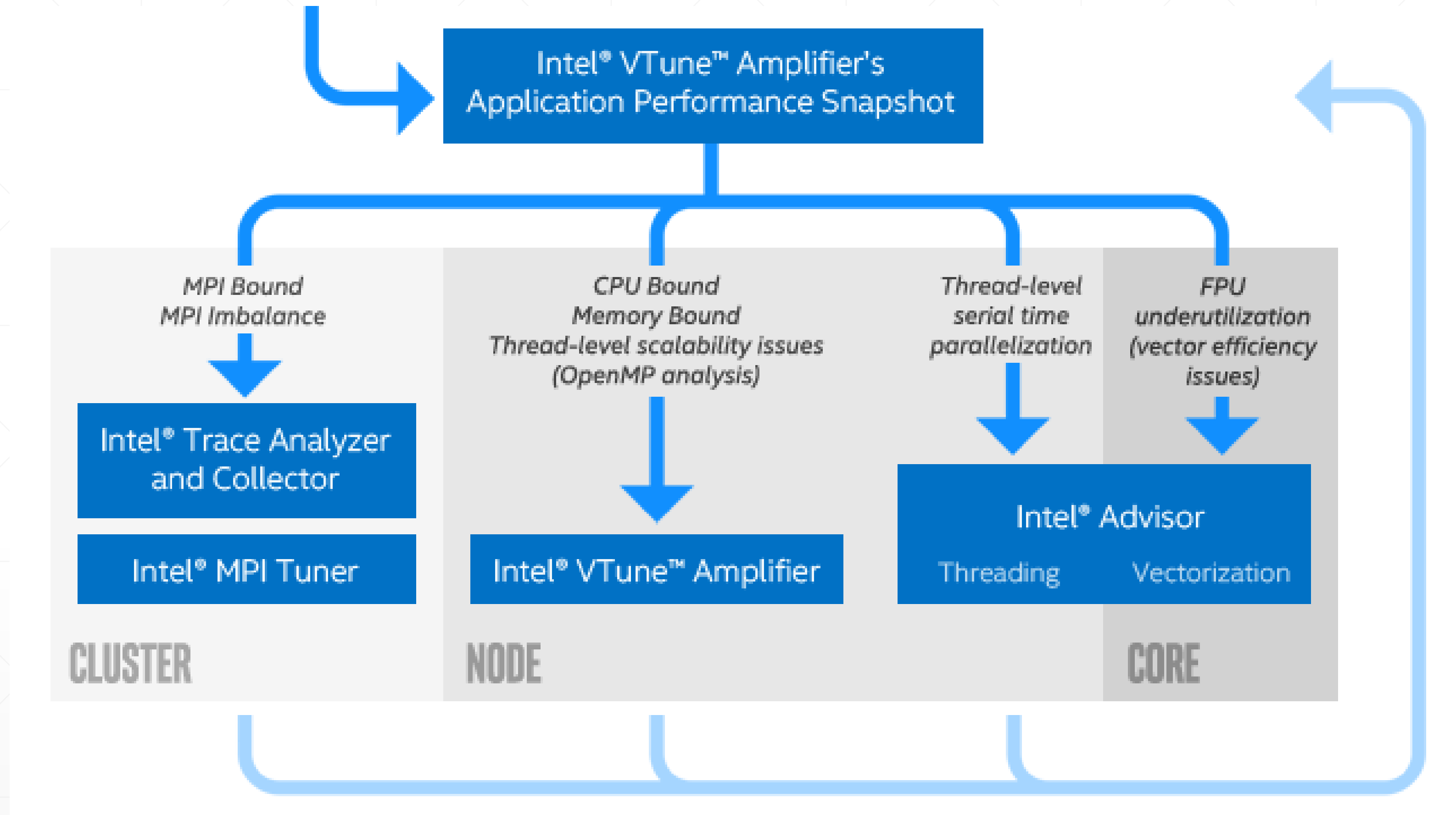
Instruction Mix

SP FLOPs
15.3% of uOps
Packed:
99.8% from SP FP
128-bit:
99.8%
256-bit:
0%
512-bit:

In-depth analysis of memory traffic

- <Memory Level> Stalls definition:
  - Percentage of cycles when the CPU is stalled (정지), waiting for data to come from <Memory Level>

In-depth Analysis with oneAPI Toolkits



- Trace Analyzer and Collector: understand MPI application for weak and strong scaling optimization
- VTune Profiler: CPU/GPU hotspot analysis, OpenMP threading efficiency, and memory access efficiency
- Advisor: vectorization efficiency, roofline analysis and GPU off-loading advisor

Optimization Workflow III: Memory Access Analysis

- The following command-line options are recommended for best experiences with Advisor:
  - *-g* full debug information
  - *-O2* moderate optimization
  - *-no-ipo* disable Intel's inter-procedural optimization during offload modeling

- Perform survey with Advisor

```
advisor -collect survey -project-dir ./result -- ./matmul.x
```

  - This shows loop hotspots and corresponding degree of vectorization

- Select loop on line #92 for memory access pattern (MAP) analysis:

```
advisor -collect map -select matmul.c:92 -project-dir ./result -- ./matmul.x
```

  - This shows whether an array has continuous memory access, i.e. unit stride
  - Unit stride allow compiler to effectively vectorize the loop

- Perform memory access analysis with VTune Profiler:

```
vtune -collect memory-access -knob analyze-mem-objects=true -result-dir ./mem -- ./matmul.x
```

  - This show the amount of load/store/LLC miss

Optimization Workflow III: Cache Optimization to Improve Vectorization

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern
[loop in mat_mul at matmul_baseline.c:93]	No Information Available	50% / 50% / 0%	Mixed Strides

```
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < p; k++)
            C[i*n+j] += A[i*p+k] * B[k*n+j]
```

Metric	Value	Unit
Memory Bound	**80.2%**	**of Pipeline Slots**
L1 Bound	0.1%	of Clockticks
L2 Bound	0.0%	of Clockticks
L3 Bound	61.4%	of Clockticks
DRAM Bound	**20.2%**	**of Clockticks**
Store Bound	0.0%	of Clockticks
NUMA: % of Remote Accesses	0.0%	
UPI Utilization Bound	0.0%	of Elapsed Time
Loads:	10,883,473,821	
Stores:	108,146,533	
LLC Miss Count	**176,974,128**	
Average Latency (cycles)	481	
Total Thread Count:	6	
Paused Time	0s	

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern
[loop in mat_mul at matmul_ikj.c:93]	No Information Available	100% / 0% / 0%	All Unit Strides

advisor *map*

```
for (int i = 0; i < m; i++)
    for (int k = 0; k < p; k++)
        for (int j = 0; j < n; j++)
            C[i*n+j] += A[i*p+k] * B[k*n+j]
```

Metric	Value	Unit
Memory Bound	**2.2%**	**of Pipeline Slots**
L1 Bound	2.5%	of Clockticks
L2 Bound	5.1%	of Clockticks
L3 Bound	0.0%	of Clockticks
DRAM Bound	**0.0%**	**of Clockticks**
Store Bound	0.0%	of Clockticks
NUMA: % of Remote Accesses	0.0%	
UPI Utilization Bound	0.0%	of Elapsed Time
Loads:	4,611,141,298	
Stores:	2,305,570,649	
LLC Miss Count	**0**	
Average Latency (cycles)	14	
Total Thread Count:	7	
Paused Time	0s	

vtune *memory-access*

Optimization Workflow III: Cache Optimization to Improve Vectorization

```
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < p; k++)
            C[i*n+j] += A[i*p+k] * B[k*n+j]
```

Top Time-Consuming Loops

Advisor

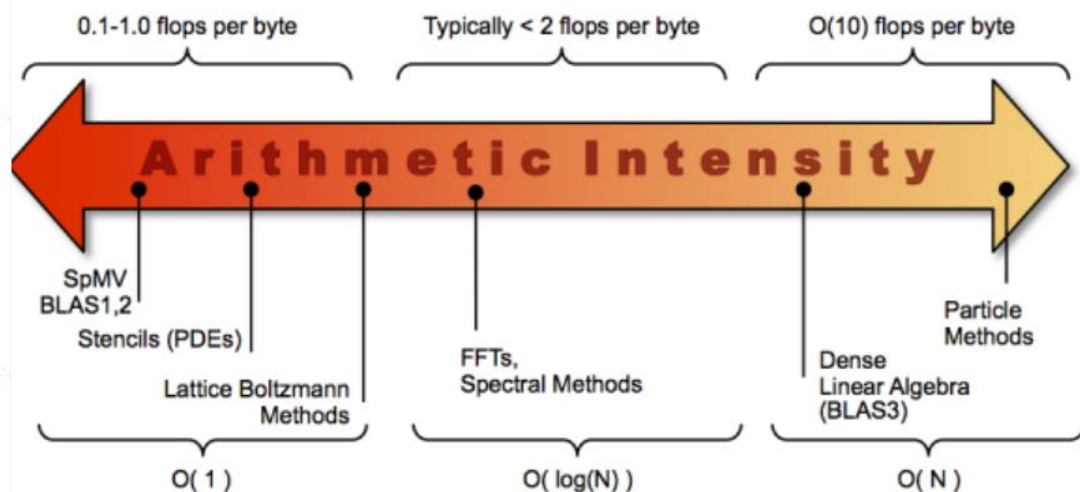
Loop	Self Time	Total Time	Trip Counts	Vector Efficiency
loop in mat_mul at matmul_baseline.c:92	105.650s	105.650s	512	28%
loop in random_matrix at matmul_baseline.c:78	0.010s	0.040s	2048	
loop in random_matrix at matmul_baseline.c:77	0.010s	0.050s	2048	
loop in __intel_avx_rep_memset	0.010s	0.010s		
loop in random_matrix at matmul_baseline.c:78	<0.001s	0.040s	2048	

```
for (int i = 0; i < m; i++)
    for (int k = 0; k < p; k++)
        for (int j = 0; j < n; j++)
            C[i*n+j] += A[i*p+k] * B[k*n+j]
```

Top Time-Consuming Loops

Loop	Self Time	Total Time	Trip Counts	Vector Efficiency
loop in mat_mul at matmul_ikj.c:92	3.180s	3.180s	512	>=100%
loop in random_matrix at matmul_ikj.c:78	0.020s	0.040s	2048	
loop in __intel_avx_rep_memset	0.010s	0.010s		
loop in main at matmul_ikj.c:90	<0.001s	3.200s	2048	
loop in random_matrix at matmul_ikj.c:78	<0.001s	0.040s	2048	

Optimization Workflow IV: Arithmetic Intensity




```
void mat_mul(float *A, float *B, float *C, int m, int n, int p) {
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
        for (int k = 0; k < p; k++)
            C[i*n+j] += A[i*p+k] * B[k*n+j];
```

L1 Arithmetic Intensity for naïve matrix multiplication

- 2 flops (add + multiply)
- 12 bytes transfer (read A, B & write C)
- AI = 2/12 = 0.167 (FLOP/byte)

- In a ***first order approximation***, the performance of an application is assumed to be bound by:
  - Machine theoretical Double Precision/Single Precision Peaks (FLOP/s)
  - Memory bandwidth such as DRAM, L1, L2, L3 caches (Byte/s)

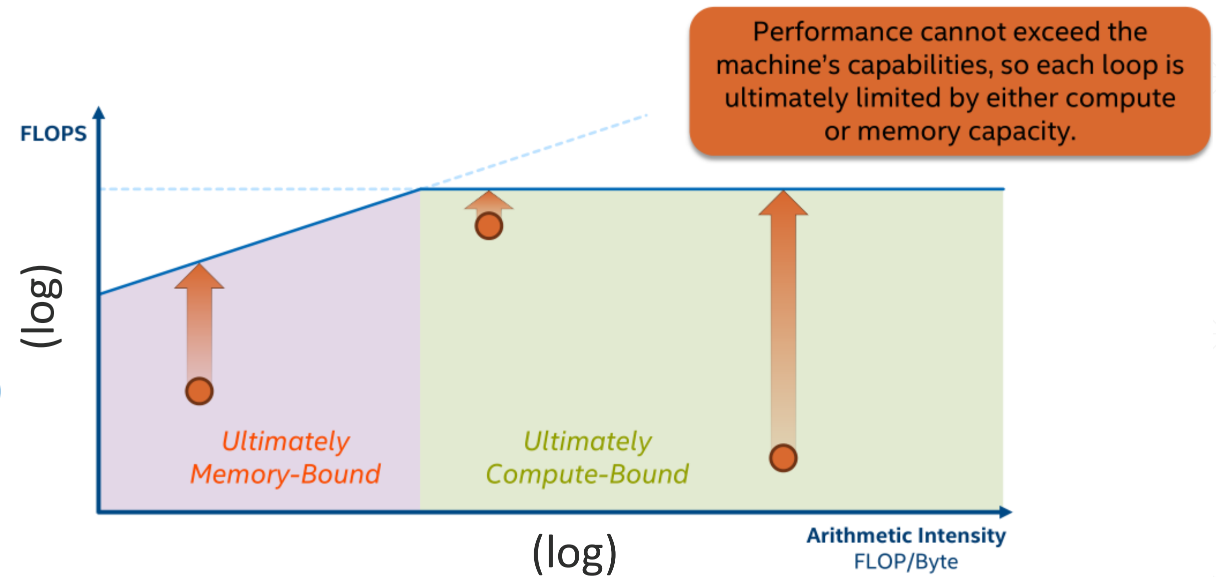
- Q: How can we combine machine's theorical FLOPs and memory bandwidth in a single model ?
- A: Arithmetic Intensity
  - Ratio of total floating-points operations to total data movement (FLOP/byte)
  - AI is an intrinsic properties of algorithm, reflecting how effectively data in cache is reused:
    - BLAS3 can archive higher AI via cache optimization techniques such as loop titling and low-level optimizations (oneMKL)

Optimization Workflow IV : Roofline Model

(FLOP/s)

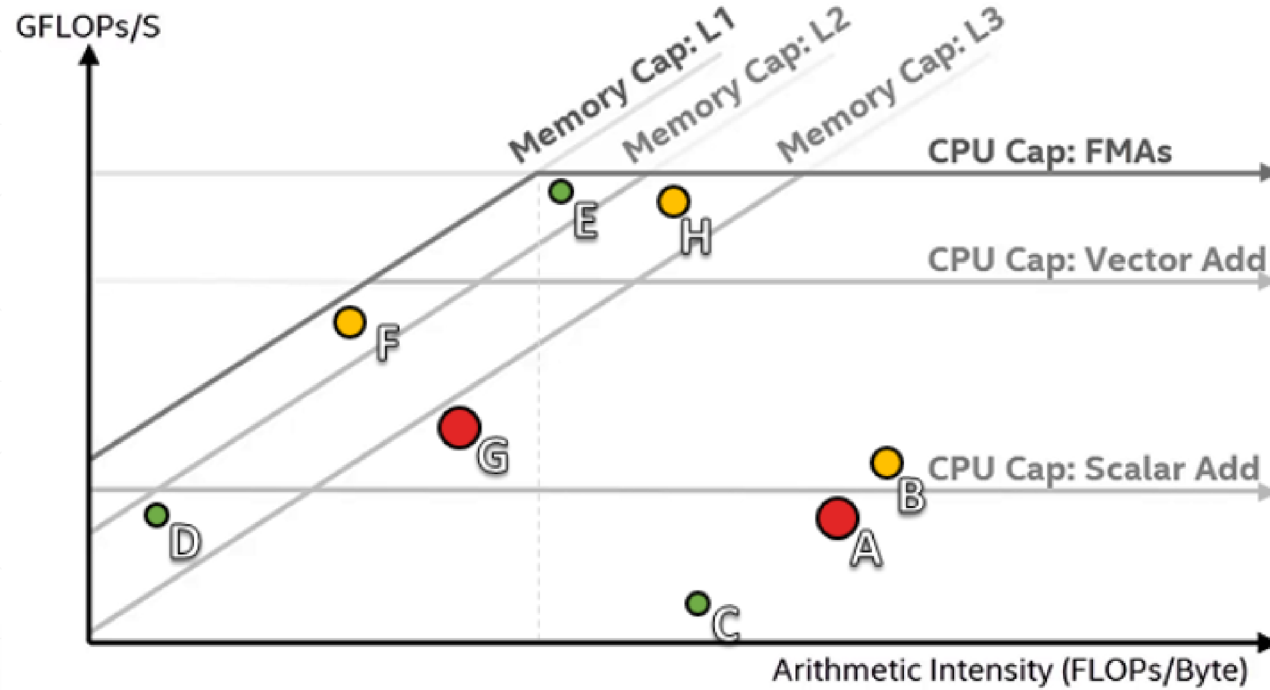
$$\text{Gflop/s} = \min \begin{cases} \text{Platform PEAK} \\ \text{Platform BW} * AI \end{cases}$$

(Byte/s) (FLOP/Byte)



- Product between AI (software-intrinsic) and Memory BW (hard-intrinsic) has unit of FLOP/s
  - Performance increases linearly as a function of AI (slope roof)
  - Performance is also bound by machine theoretical peaks (horizontal roof)
- Roofline graph is represented in log to log scale:
  - Increase memory bandwidth results in a vertical shift of the slope roof
  - Hierarchical structure of cache can be represented in a single roofline graph

Optimization Workflow IV: Hierarchical Roofline Model



- Each dot represents a loop:
  - Bigger dots are more time-consuming loops: red > yellow > green
  - Best candidate loops for optimizations: A and G
  - Vectorization and threading moves dots vertically (higher GFLOPS):
    - #pragma omp simd
    - #pragma vector aligned
  - Optimization of memory access moves dots horizontally (higher AI)

Optimization Workflow IV: Optimization Guides

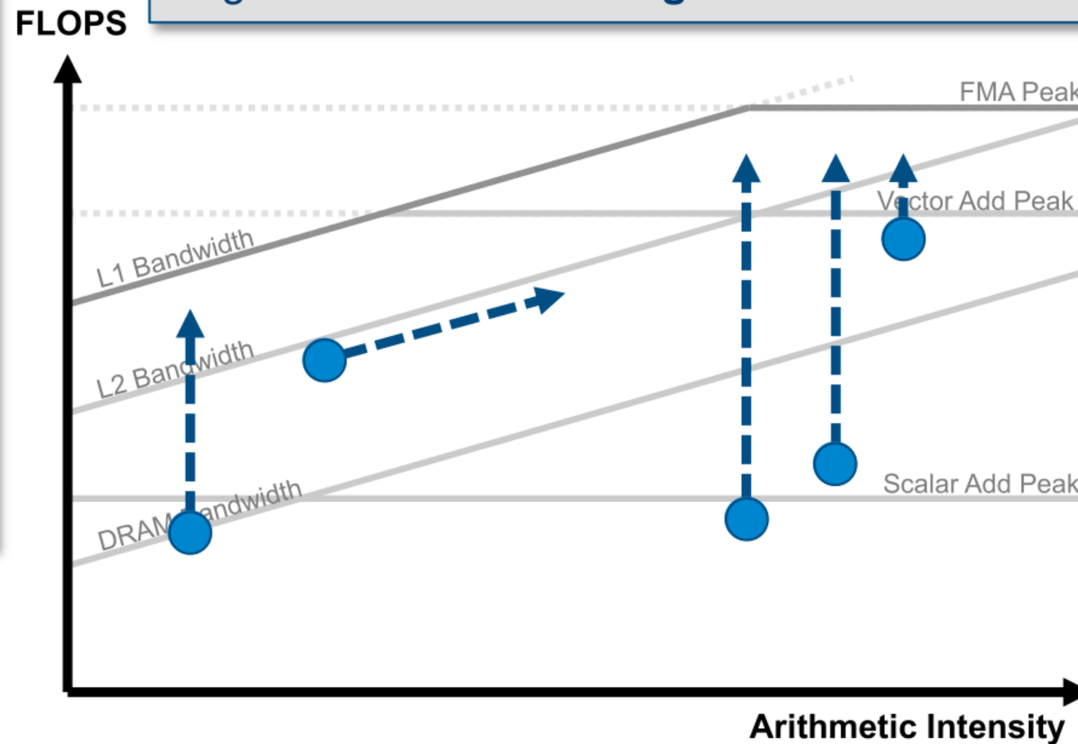
Next Steps

If under or near a memory roof…

- Try a MAP analysis. Make any appropriate **cache optimizations.**
- If cache optimization is impossible, try **reworking the algorithm to have a higher AI.**

If Under the Vector Add Peak

Check “Traits” in the Survey to see if FMAs are used. If not, try altering your code or compiler flags to **induce FMA usage.**



If just above the Scalar Add Peak

Check **vectorization efficiency** in the Survey. Follow the recommendations to improve it if it’s low.

If under the Scalar Add Peak…

Check the Survey Report to see if the loop vectorized. If not, try to **get it to vectorize** if possible. This may involve running Dependencies to see if it’s safe to force it.

Optimization Workflow IV: Intel® Advisor Roofline Analysis

- Generate performance survey and code analytics:

```
advisor -collect survey -project-dir ./result -- ./matmul.x
```

- Generate roofline graph:

```
advisor -collect tripcounts -flop -project-dir -enable-cache-simulation ./result -- ./matmul.x
```

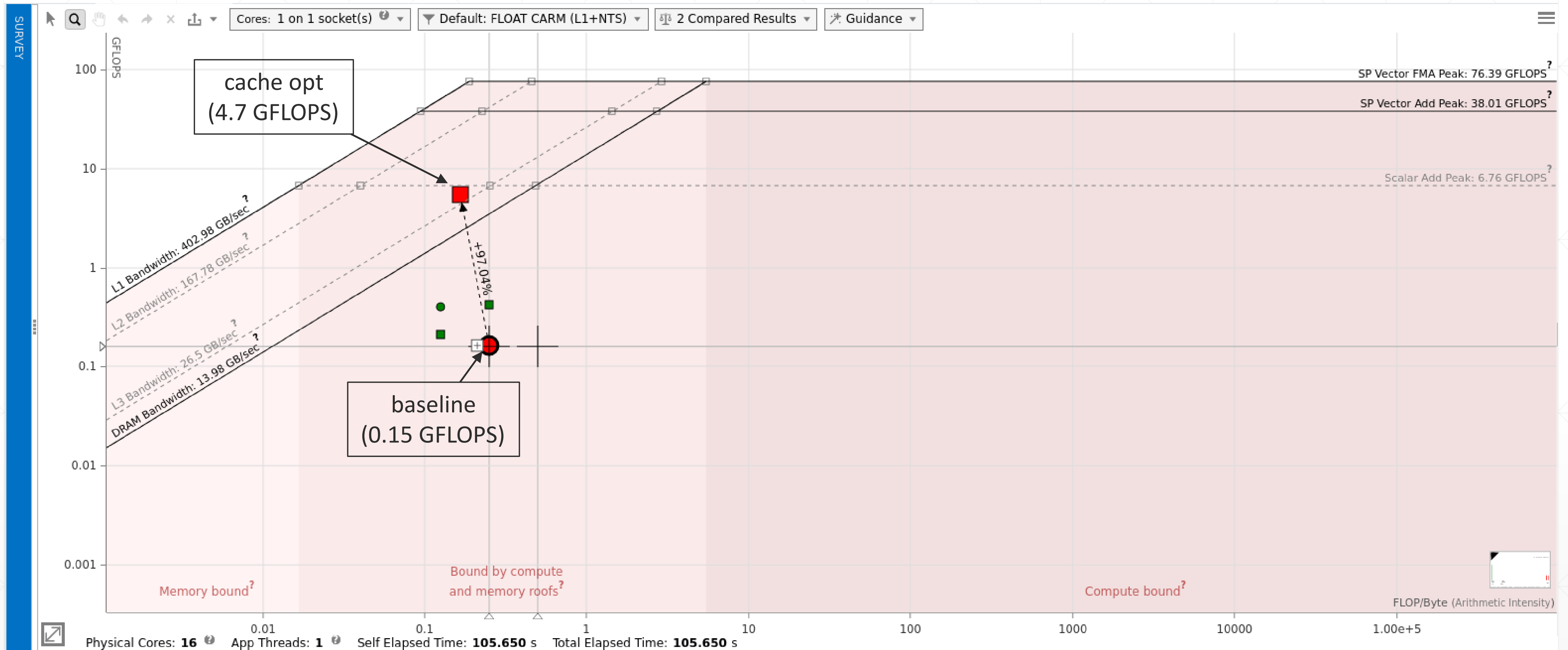
- Generate roofline report in HTML format:

```
advisor -report roofline -project-dir ./result -report-output ./roofline.html
```

- View result with Advisor GUI:

```
advisor-gui result/result.advixeproj
```

Optimization Workflow IV: Cache Optimization Roofline



- What is the machine theoretical FLOPS and memory bandwidth ?
- Is the application mainly memory bound or compute bound ?

Optimization Workflow IV: Data Aligned for Vectorization

```
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < p; k++)
            C[i*n+j] += A[i*p+k] * B[k*n+j]
```

```
for (i = 0; i < m; i++)
    for (k = 0; k < p; k++)
        for (j = 0; j < n; j++)
            C[i*n+j] += A[i*p+k] * B[k*n+j]
```

```
float* A = (float*) _mm_malloc(sizeof(float)*m*p,64);
...
for (i = 0; i < m; i++)
        for (k = 0; k < p; k++)
            #pragma vector aligned
            #pragma omp simd reduction(+:C[i*n+j])
            for (j = 0; j < n; j++)
                C[i*n+j] += A[i*p+k] * B[k*n+j];
_mm_free(A);
```

```
remark #15388: vectorization support: reference C[i*n+j] has aligned access [ matmul_aligned.c(96,17) ]
remark #15388: vectorization support: reference C[i*n+j] has aligned access [ matmul_aligned.c(96,17) ]
remark #15388: vectorization support: reference B[k*n+j] has aligned access [ matmul_aligned.c(96,40) ]
```

- Use Intel intrinsics to align vectors at 64-byte boundary for AVX512 vectorization

Optimization Workflow V: GPU Offload Modeling

- The following command-line options are recommended for best experiences with Advisor:
  - *-g* full debug information
  - *-O2* moderate optimization
  - *-no-ipo* disable Intel's inter-procedural optimization during offload modeling

- Modeling performance on Intel DG1 GPU:

```
advisor-python $(APM)/run_oa.py \
    result_gen9                 \
    --config gen9_gt4           \
    --collect basic             \
    --no-assume-dependencies
    -- ./matmul.x
```

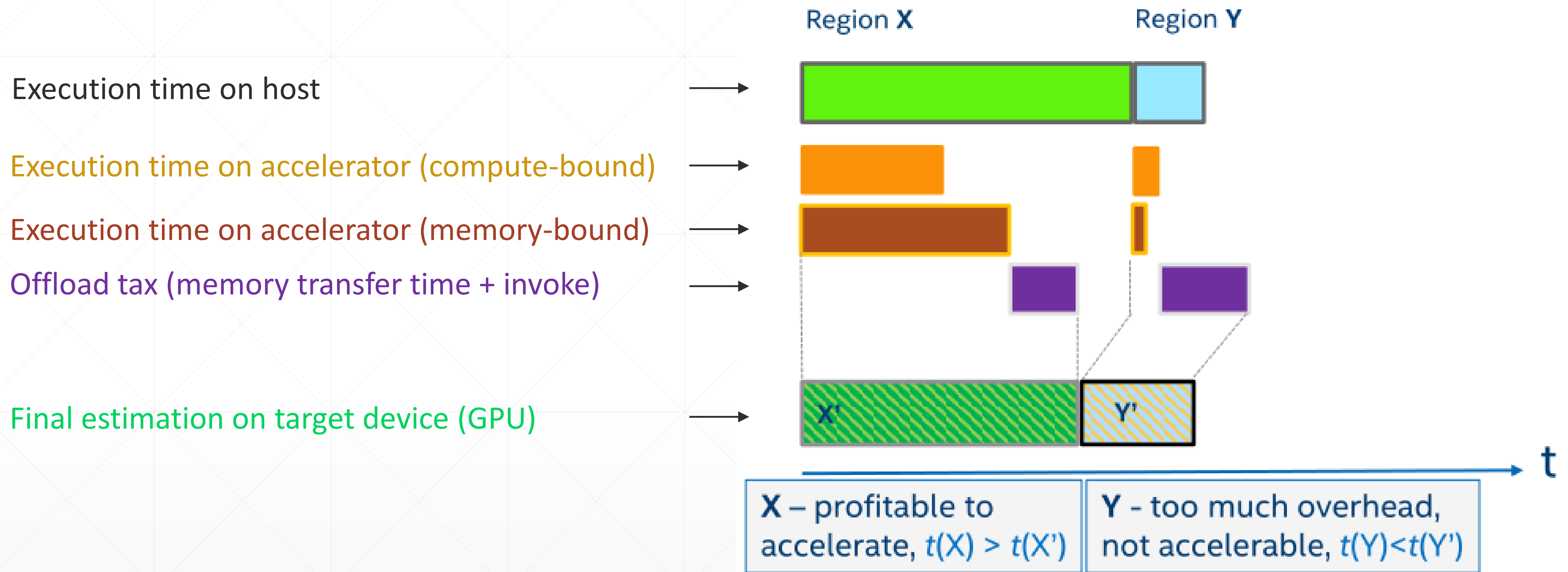
- Legacy HTML report:
  - result_gen9/rank.0/pp000/data.0/report.html

Arguments

<*string*> is one of the following device configurations:

Argument	Description
gen12_tgl	Intel® Iris® Xe graphics
gen12_dg1	Intel® Iris® Xe MAX graphics
gen11_icl	Intel® Iris® Plus graphics
gen9_gt2	Intel® HD Graphics 530
gen9_gt3	Intel® Iris® Graphics 550
gen9_gt4	Intel® Iris® Pro Graphics 580

Optimization Workflow V: Modeling Performance on GPU



$$t_{region} = \max(t_{compute}, t_{memory\ subsystem}) + t_{data\ transfer\ tax} + t_{invocation\ tax}$$

- Region X: memory bound and small offload tax
- Region Y: compute bound and high offload tax

Optimization Workflow V: Gen9 Offload Modelling

Top Metrics

2.1x	2.1x	100%	3
Speed Up for Accelerated Code	Amdahl's Law Speed Up	Fraction of Accelerated Code	Number of Offloads

Program Metrics

Original	23.25s
Accelerated	10.84s

Program Time on Host After Acceleration	0.02s
Time on Target	10.68s
Time in MPI calls	0s
Non Accelerated Time	0.14s

Target Platform	Gen9 GT4
Speed Up for Accelerated Code	2.1x
Amdahl's Law Speed Up	2.1x
Fraction of Accelerated Code	100%
Number of Offloads	3

Offload Bounded By

Compute	0%
L3 Cache BW	0%
LLC BW	0%
Memory BW	100%
Latencies	0%
Data Transfer	0%
Launch Tax	0%
Dependency	0%
Trip Count	0%
Atomics	0%
Unknown	0%
Non Offloaded	<1%

CPU+GPU

Loop/Function	Performance Issues	Measured Time	Basic Estimated Metrics: Speed-Up	Time	Offload Summary	Estimated Bounded By: Throughput	Taxes With Reuse
[loop in main at matmul.c:83]	Code region is recommended for offloading	23s	2.155x	10.67s	Offloaded	DRAM BW 10.67s; LLC BW 4.001s	Launch Tax < 0.1ms; All Taxes < 0.1ms
[loop in random_matrix at matmul.c:71]	Code region is recommended for offloading	120.0ms	2.177x	55.1ms	Offloaded	DRAM BW 5.3ms; LLC BW 2.0ms	Launch Tax < 0.1ms; All Taxes < 0.1ms
[loop in random_matrix at matmul.c:71]	Code region is recommended for offloading	110.0ms	1.162x	94.7ms	Offloaded	DRAM BW 5.0ms; LLC BW 2.0ms	Launch Tax < 0.1ms; All Taxes < 0.1ms

Data Transfer Estimations | **Details**

[loop in main at matmul.c:83]

ESTIMATED SPEED-UP: 2.155x

Estimated Time	Measured Time
10.67s	23s

BOUNDED BY: DRAM BW

Compute	119.6ms
DRAM BW	10.67s
L3 BW	1.359s
LLC BW	4.001s
Atomic Throughput	0ms
Data Transfer Tax	0ms
Kernel Launch Tax	< 0.1ms
Load Latency	< 0.1ms
Estimated Time	10.67s

Gen9 offers 2x potential speed up

Optimization Workflow V: Gen12 Offload Modelling

Top Metrics

4.6x	4.6x	100%	3
Speed Up for Accelerated Code	Amdahl's Law Speed Up	Fraction of Accelerated Code	Number of Offloads

Program Metrics

Original	22.61s
Accelerated	4.93s

Program Time on Host After Acceleration	0.03s
Time on Target	4.71s
Time in MPI calls	0s
Non Accelerated Time	0.15s

Target Platform	Xe LP Max
Speed Up for Accelerated Code	4.6x
Amdahl's Law Speed Up	4.6x
Fraction of Accelerated Code	100%
Number of Offloads	3

Offload Bounded By

Compute	0%
L3 Cache BW	0%
Memory BW	99%
Latencies	0%
Data Transfer	1%
Launch Tax	0%
Dependency	0%
Trip Count	0%
Atomics	0%
Unknown	0%
Non Offloaded	<1%

CPU+GPU

Loop/Function	Performance Issues	Measured Time	Basic Estimated Metrics: Speed-Up	Time	Offload Summary	Estimated Bounded By: Throughput	Taxes With Reuse
[loop in main at matmul.c:83]	Code region is recommended for offloading	22.32s	4.714x	4.734s	Offloaded	HBM BW 4.708s; L3 BW 1.003s	DT Tax 26.5ms; All Taxes 26.5ms
[loop in random_matrix at matmul.c:71]	Code region is recommended for offloading	140.0ms	1.835x	76.3ms	Offloaded	HBM BW 2.4ms; L3 BW 0.3ms	DT Tax 4.4ms; All Taxes 4.4ms
[loop in random_matrix at matmul.c:71]	Code region is recommended for offloading	120.0ms	1.388x	86.5ms	Offloaded	HBM BW 2.0ms; L3 BW 0.3ms	DT Tax 4.4ms; All Taxes 4.4ms

Data Transfer Estimations | **Details**

[loop in main at matmul.c:83]

ESTIMATED SPEED-UP: 4.714X

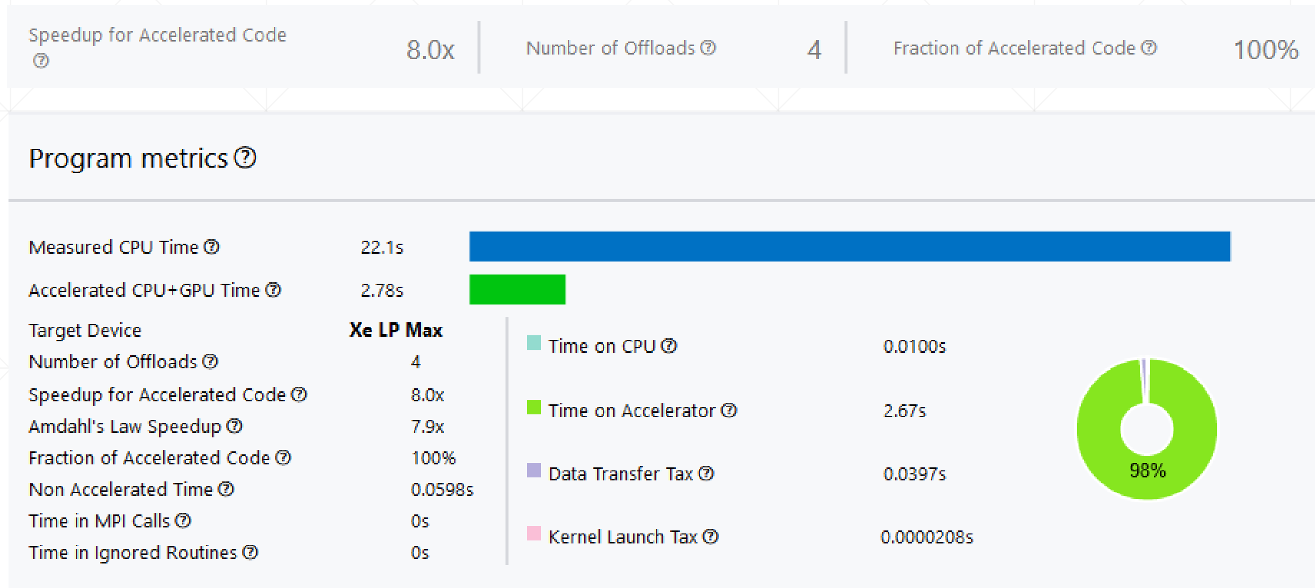
Estimated Time 4.734s | Measured Time 22.32s

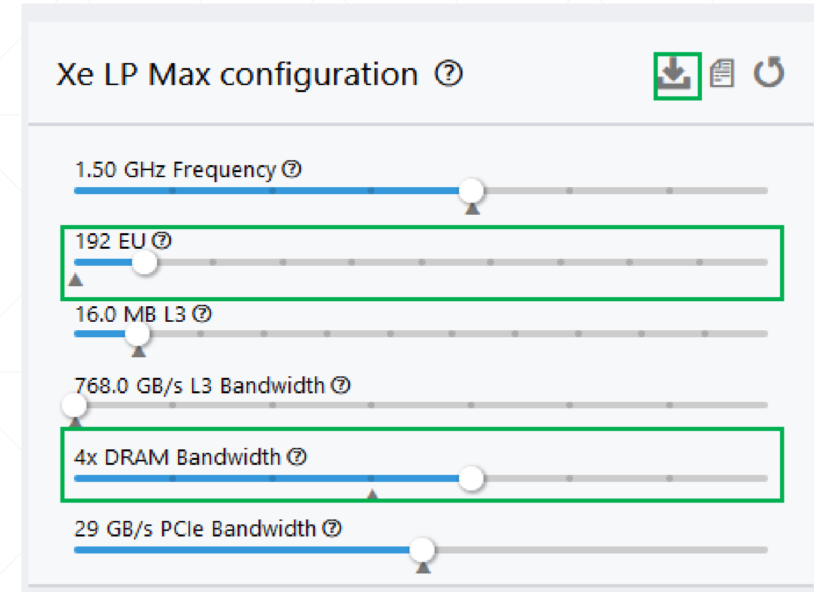
BOUNDED BY: HBM BW

Compute	68.3ms
L3 BW	1.003s
HBM BW	4.708s
Atomic Throughput	0ms
Data Transfer Tax	26.5ms
Kernel Launch Tax	< 0.1ms
Load Latency	< 0.1ms
Estimated Time	4.734s

Xe LP Max offers 4.6x speed up

Optimization Workflow V: Customized GPU with Configuration Slider



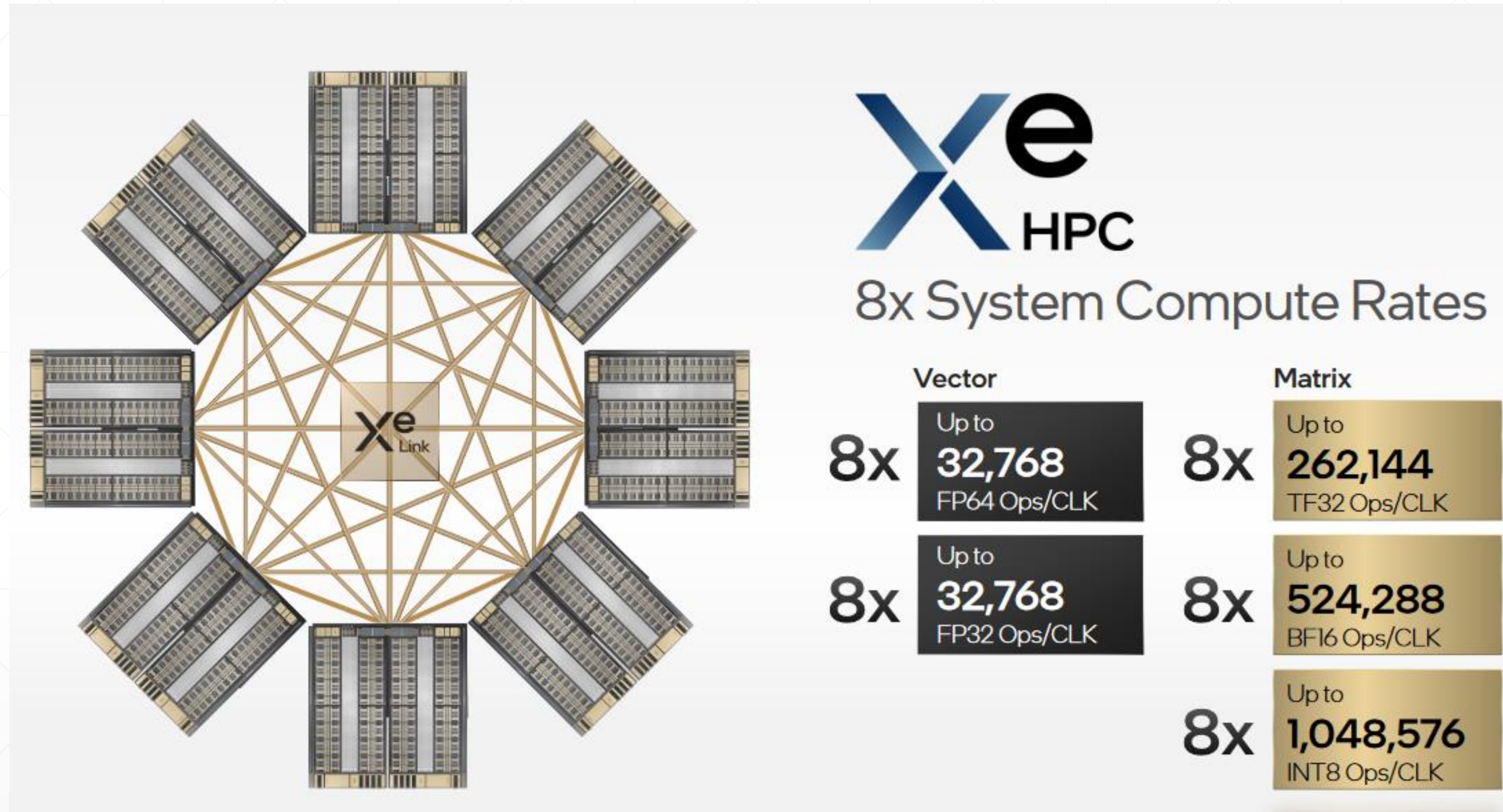


- Use configuration slider to model custom GPU:
  - Executing unit (EU): 96 → 192
  - HBW: 54 GB/s → 96 GB/s
  - Save new config file as *scalers.toml*
  - Redo offload modeling
- Results:
  - 8x performance gain vs 4.6x (default)
  - For example, Xe-HP can support up to 512 EUs

```
advisor-python $(APM)/run_oa.py \
     --result_new               \
     --config gen12_dg1         \
     --config scalers.toml      \
     --collect basic            \
     --no-assume-dependencies   \
     -- ./matmul.x
```

(*scalers.toml* overrides default EU and HBW)

Estimation of Performance Gain on Xe HPC with Offload Advisor



https://www.intel.com/content/www/us/en/newsroom/resources/press-kit-architecture-day-2021.html

- With Offload Advisor, you can estimate performance gain of your codes on new GPUs before buying.
- Configuration slider can be used to simulate higher-tier GPU such as Xe HP and Xe HPC

Optimization Workflow VI: DPCPP and OpenMP Offloading

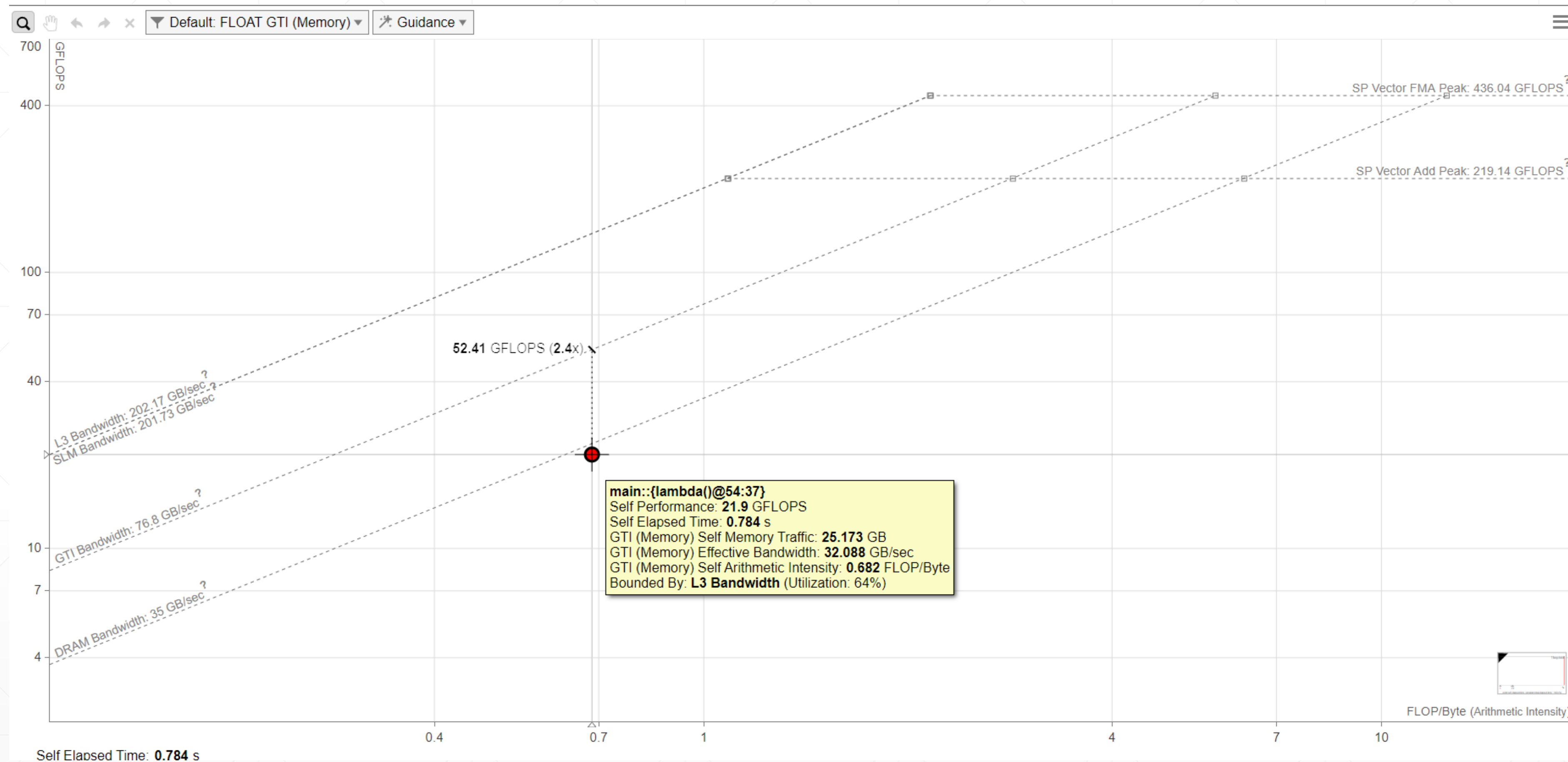
- DPCPP port

```
sycl::device device(sycl::default_selector{});
sycl::queue  queue(device);
...
float *A_USM = sycl::malloc_shared<float>(m * p, queue);
float *B_USM = sycl::malloc_shared<float>(p * n, queue);
float *C_USM = sycl::malloc_shared<float>(m * n, queue);
...
queue.parallel_for(range(m, n), [=](auto index) {
    auto i = index[0];
    auto j = index[1];
    for (int k=0; k<p; k++)
        C_USM[i*n+j] += A_USM[i*p+k] * B_USM[k*n+j];
});
```

- OpenMP offloading

```
#pragma omp target teams distribute parallel for
#pragma omp target data map(to: A[0:m*p], B[0:p*n]) map(tofrom: C[0:m*n])
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < p; k++)
            C[i*n+j] += A[i*p+k] * B[k*n+j];
```

Optimization Workflow VI: GPU Roofline Analysis of DPCPP Code



- Generate roofline for Gen9 graphics

```
advisor -collect survey -profile-gpu -project-dir ./gen9_result -- ./matmul_sycl.x
```

```
advisor -collect tripcounts -profile-gpu -stacks -flop -project-dir ./gen9_result -- ./matmul_sycl.x
```

Optimization Workflow VII : Minimization Analysis Overhead

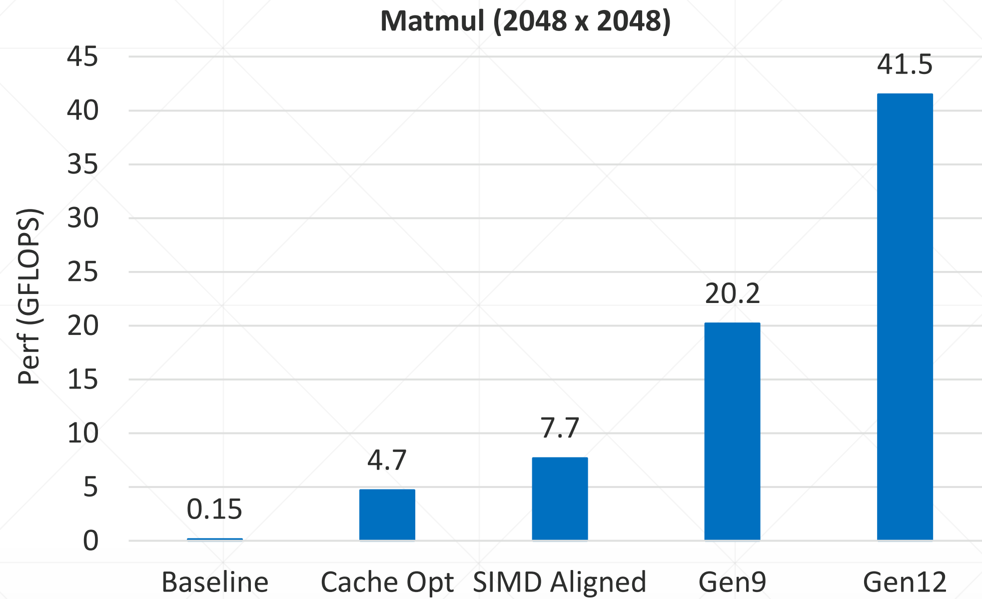
Runtime Overhead / Analysis	Survey	Characterization	Dependencies	MAP
Target application runtime with Intel® Advisor compared to runtime without Intel® Advisor	1.1x longer	2 - 55x longer	5 - 100x longer	5 - 20x longer

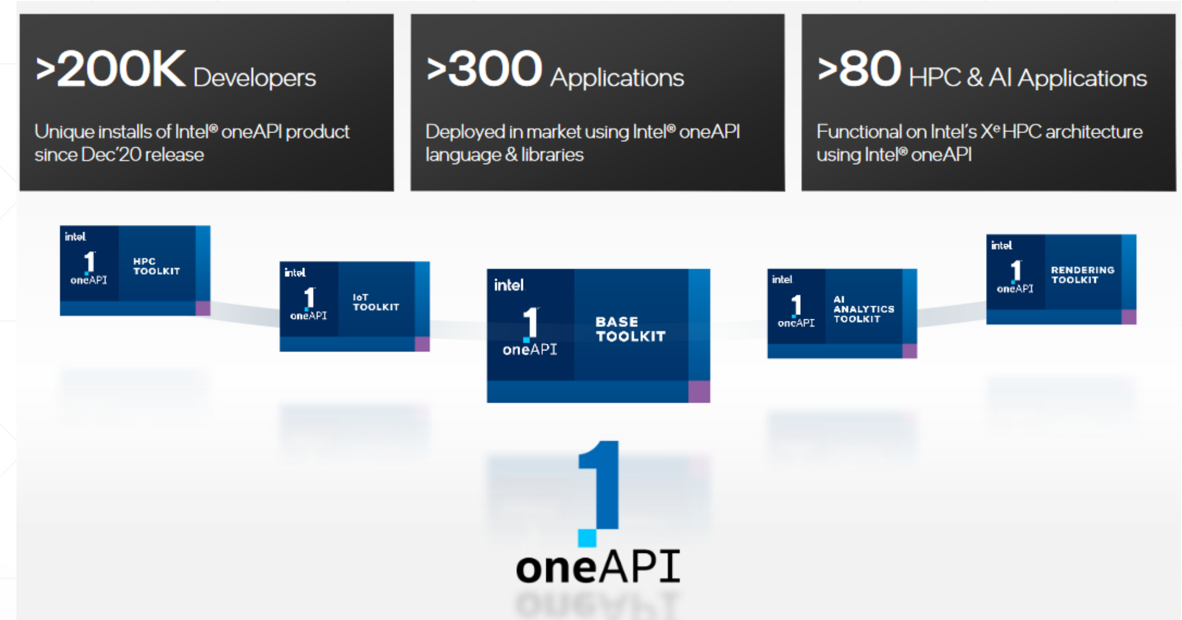
- Techniques to minimize overhead:
  - Collection controls:
    - Pause/resume long analysis
    - Stop collection after a specific time
    - Skip unimportant phase of code execution such as initialization
  - Loop markup:
    - Skip unimportant loops and focus only on important ones
  - Filtering:
    - Skip unimportant functions and focus only on important ones
  - Execution Speed/Duration/Scope Properties:
    - Disable stack collection, increase sampling interval, etc

```
#include "advisor-annotate.h"

void mat_mul(float *A, float *B, float *C,
             int m, int n, int p) {
    ANNOTATE_SITE_BEGIN();
    for (int i = 0; i < m; i++) {
        ANNOTATE_ITERATION_TASK();
        for (int j = 0; j < n; j++)
            for (int k = 0; k < p; k++)
                C[i*n+j] += A[i*p+k] * B[k*n+j];
    }
    ANNOTATE_SITE_END();
}
```

- https://software.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/minimize-analysis-overhead.html

Conclusion





- oneAPI allows developers archive best performance for heterogenous platforms:
  - Easy to use with well designed user interfaces
  - Memory access analysis to improve efficiency of vectorization
  - Automated roofline analysis to understand hardware limitations
  - Offload simulation to gauge potential performance gain on Intel GPUs before purchase