ALCF INCITE GPU Hackathon May 20-22, 2025

Intel Analyzers

VTune, Advisor, APS

Rupak Roy (Intel), JaeHyuk Kwack (ANL)

Agenda

1 Overview of Intel® VTune™ Profiler

- Overview
- Profiling Capabilities

2 Running Intel® VTune™ Profiler on Aurora

- Configuring VTune
- Different CPU/GPU Analysis Types
- Controlling Collection Overhead
- Visualizing Results

3 Overview of Intel® Advisor

- Overview
- Configuring Intel® Advisor on Aurora
- GPU Roofline

Intel® VTune™ Profiler

Optimize Performance

Intel® VTune™ Profiler

Get the Right Data to Find Bottlenecks

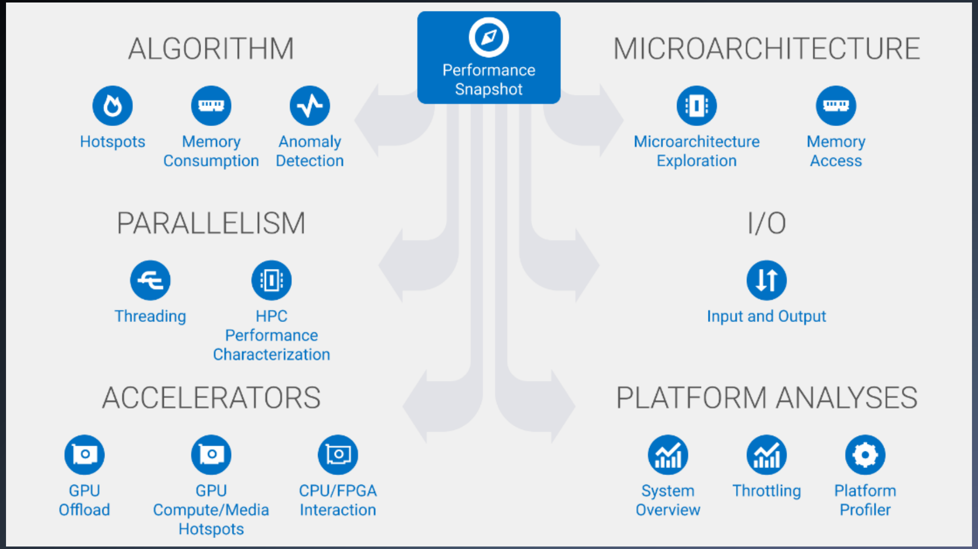
- A suite of profiling for CPU, GPU, NPU, memory, cache, storage, offload, power…
- Application or system-wide analysis
- SYCL, C, C++, Fortran, Python*, Go*, Java*, or a mix
- Linux, Windows, and more
- Containers and VMs

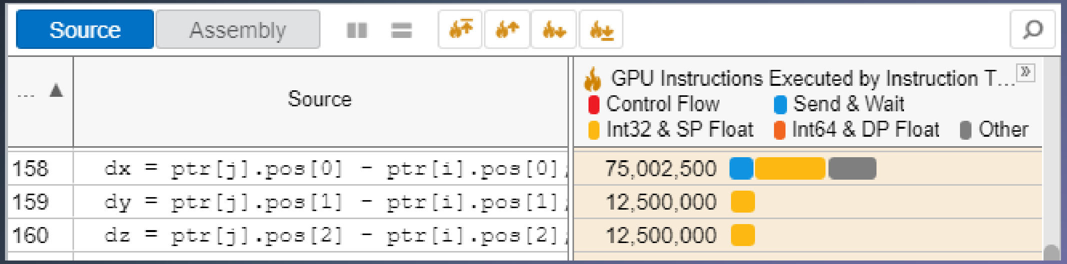
Analyze Data Faster

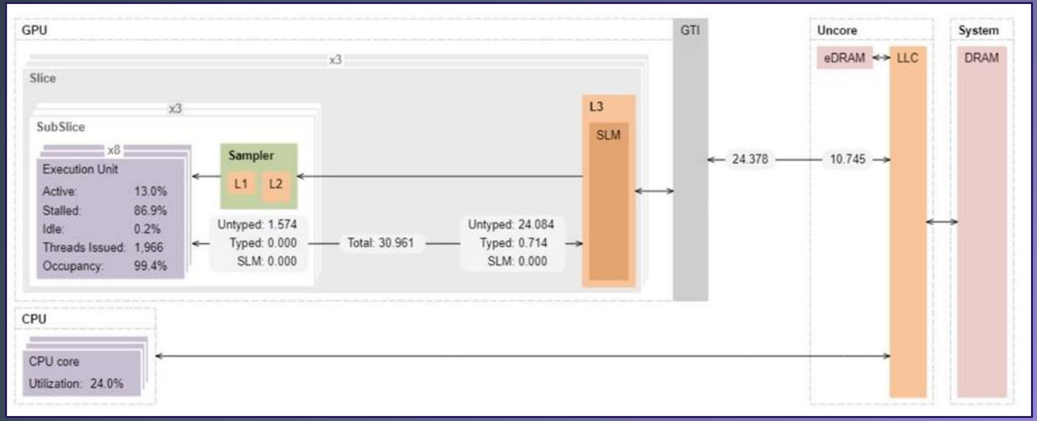
- Collect data HW/SW sampling and tracing w/o re-compilation
- See results on your source, in architecture diagrams, as a histogram, on a timeline…
- Filter and organize data to find answers

Work Your Way

- User interface or command line
- Profile locally and remotely



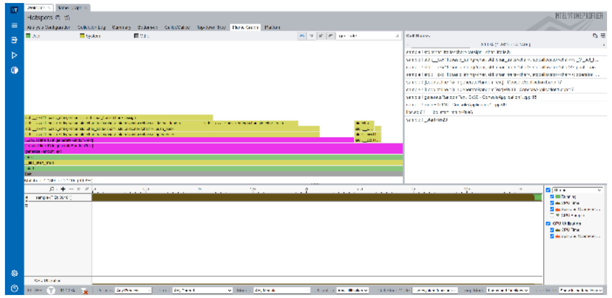


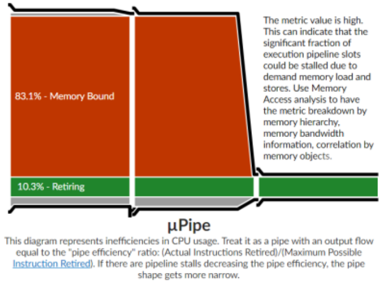


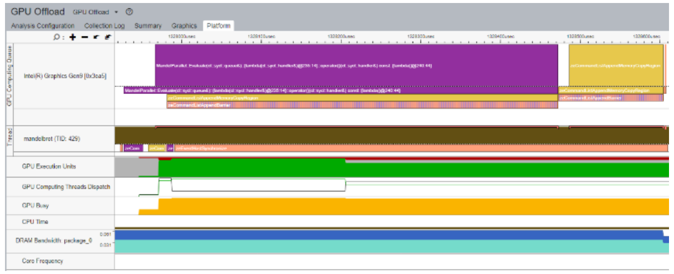
Argonne NATIONAL LABORATORY

Rich Set of Profiling Capabilities

Intel® VTune™ Profiler







Algorithm Optimization

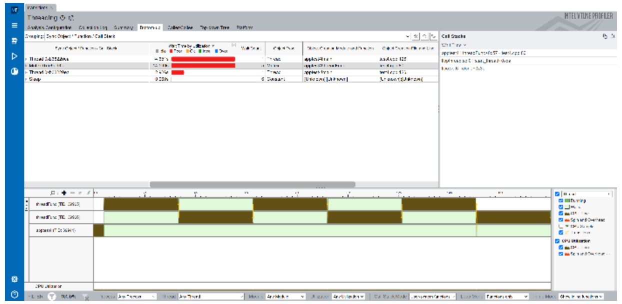
- ✓ Hotspots
- ✓ Anomaly Detection
- ✓ Memory Consumption

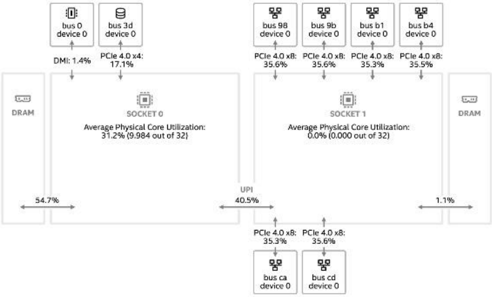
Microarch.&Memory Bottlenecks

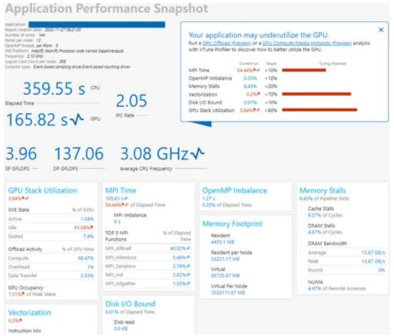
- ✓ Microarchitecture Exploration
- ✓ Memory Access

Accelerators / xPU

- ✓ GPU Offload
- ✓ GPU Compute / Media Hotspots
- ✓ CPU/FPGA Interaction







Parallelism

- ✓ Threading
- ✓ HPC Performance Characterization

Platform

- ✓ Input and Output
- ✓ System Overview

Multi-Node

- ✓ Application Performance Snapshot

Configuring VTune on Aurora

- **Loading the latest VTune on Aurora:**

```
$ module load oneapi/release/2025.0.5
$ vtune --version
Intel(R) VTune(TM) Profiler 2025.0.1 (build 629235) Command Line Tool
Copyright (C) 2009 Intel Corporation. All rights reserved.
```

- **Getting Started with VTune:**

Analysis types:

See the available analysis types e.g., gpu-hotspots, gpu-offload

```
$ vtune --help collect
```

Available knobs:

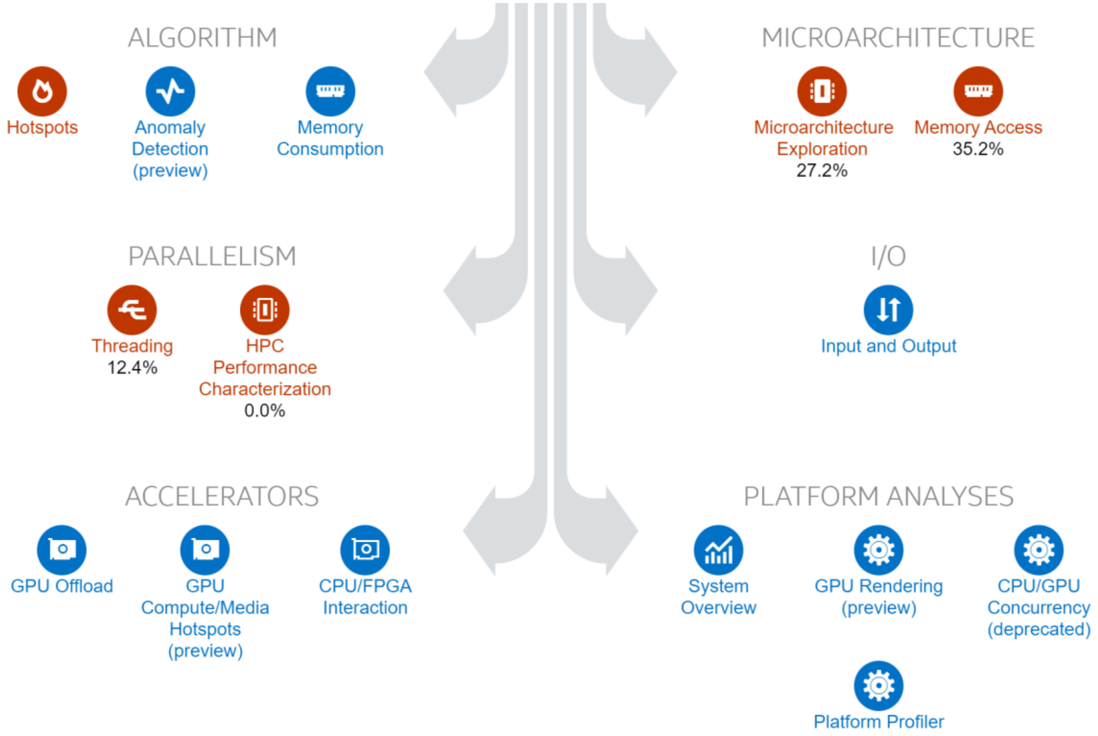
See the available knobs for a certain analysis type e.g., sampling-interval, enable-stack-collection

```
$ vtune --help collect gpu-hotspots
```

Quickly Identify Untapped Performance

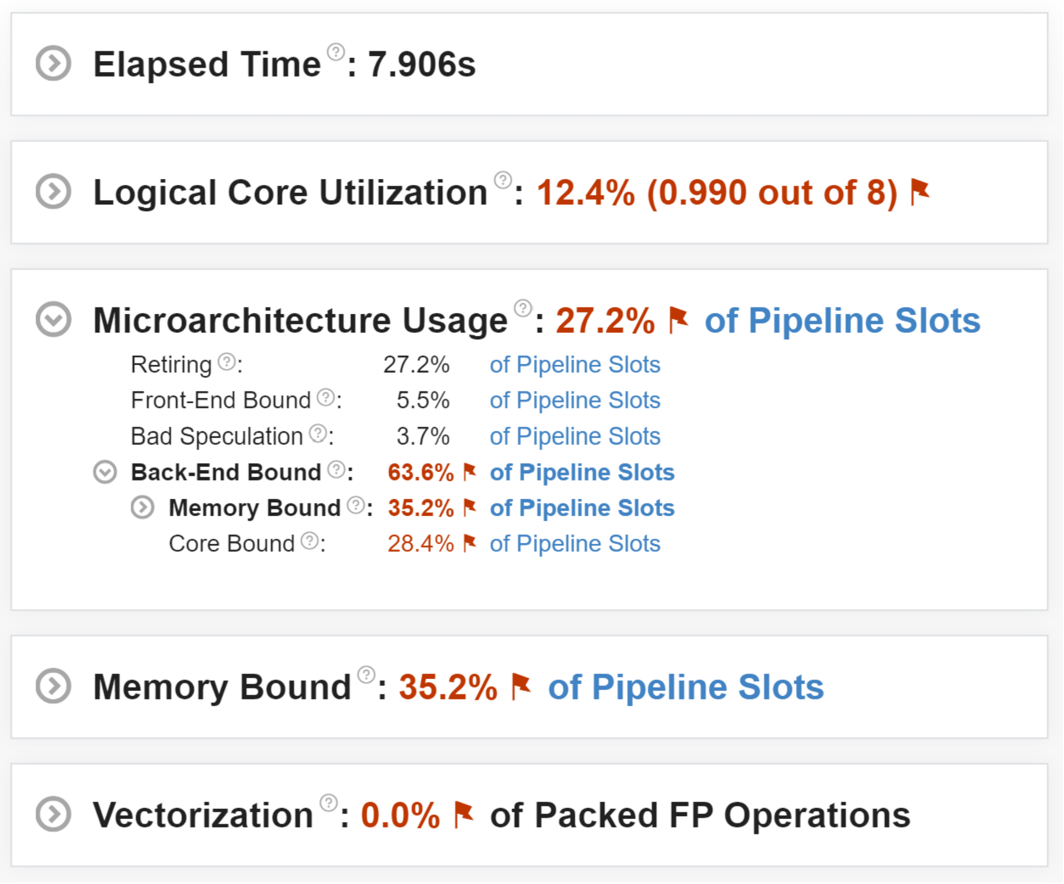
Intel® VTune™ Profiler - Performance Snapshot Analysis

Choose your next analysis:

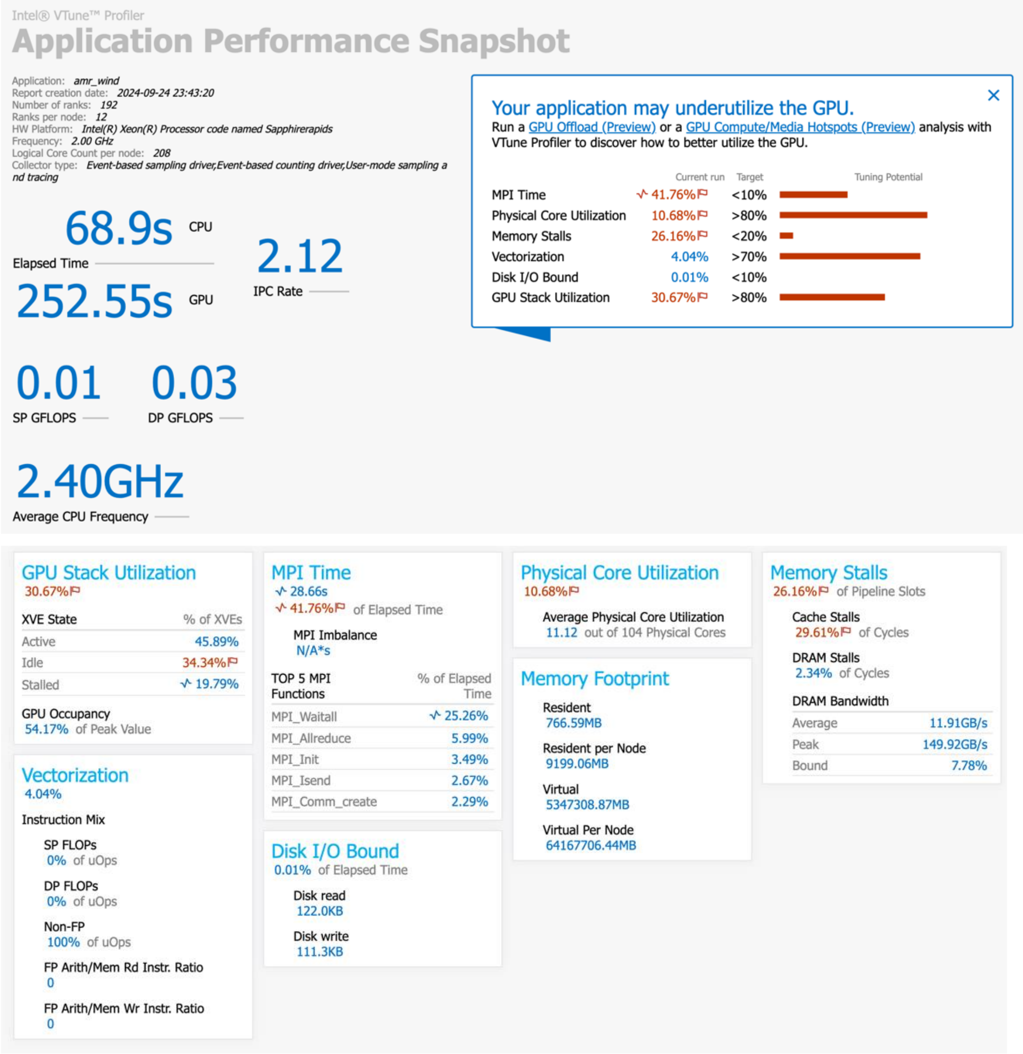



```
vtune -collect performance-snapshot --./app
```

Characterize high-level aspects:



Intel® VTune™ Profiler Application Performance Snapshot (APS)



- **Sample Command Line:**
  - **APS Collection**

```
mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh aps -r {aps_result_dir}
./{your_application} {Command_line_arguments_for_your_application}
```

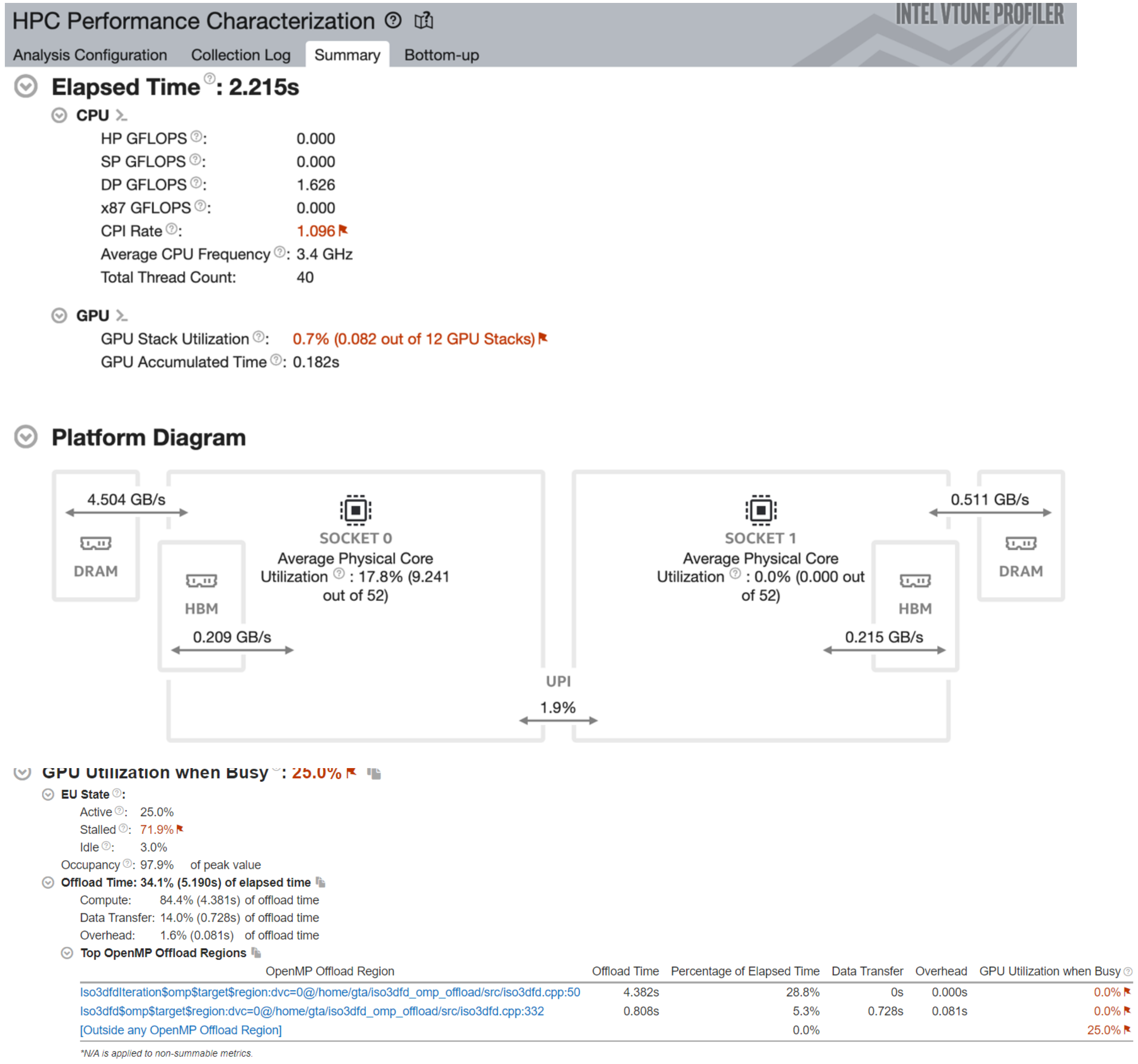
  - **HTML Report Generation:**

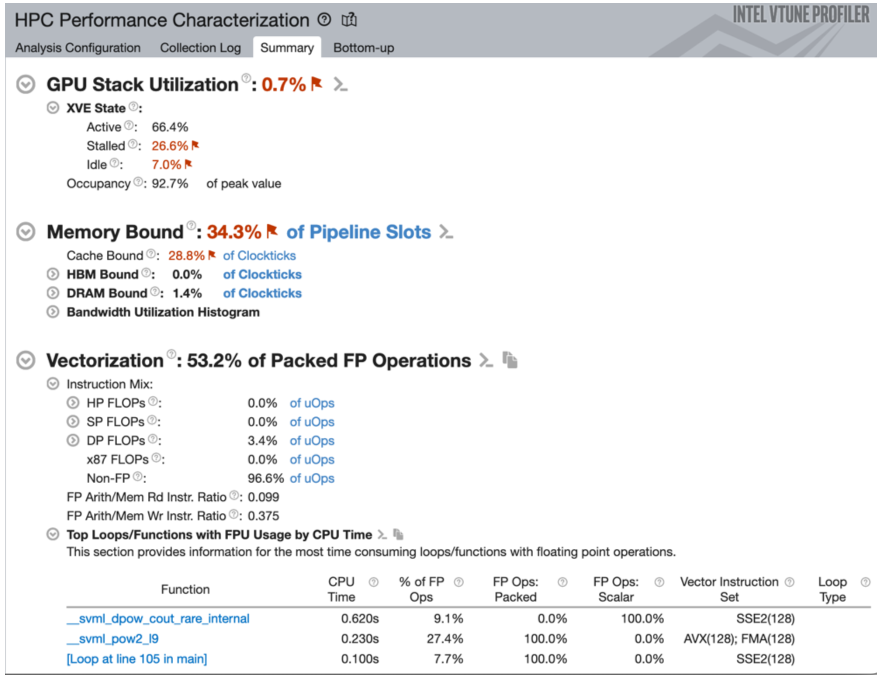
```
aps-report {aps_result_dir}
```

- **Observation**
  - High MPI Time
  - Low GPU Stack Utilization
  - Low Physical Core Utilization
  - High Memory Stalls
- **Next Steps**
  - Running HPC Performance Analysis for a deep dive into MPI Activity and Vectorization
  - Running GPU Analysis for understanding GPU underutilization

HPC Performance Characterization

Effectively analyze your compute-intensive application





Sample Command Line:

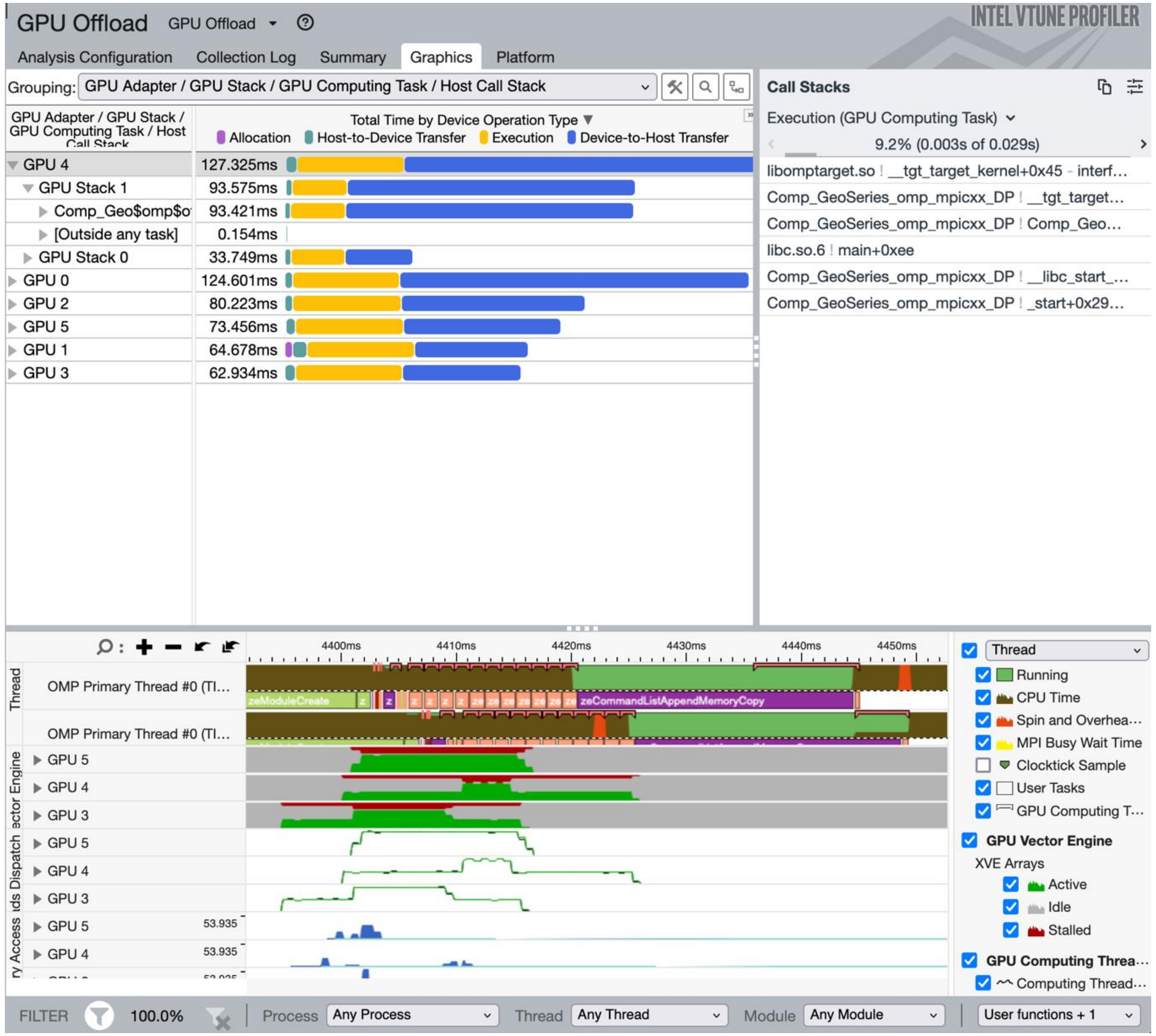
```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect hpc-performance -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

A starting point for performance optimization

- Effective Physical Core Utilization/ Vector Engine Utilization
- Memory Bound
- Vectorization
- OpenMP Offload Regions/ SYCL Compute Tasks

GPU Offload Analysis

- Identify whether the application is CPU (Host) or GPU (Device) bound
- See the detailed breakdown of different host and device operations for each GPU/Compute tasks
  - Allocation
  - Host-to-device data transfer
  - Device-to-host data transfer
  - Execution
  - Synchronization
- Correlation between CPU thread/core/process activity and GPU activity
- See Host/Device Compute and Memory activities
  - GPU Memory Access
  - System Memory Access
  - Host to GPU Memory Access
  - Stack to stack access
  - PCIe Bandwidth



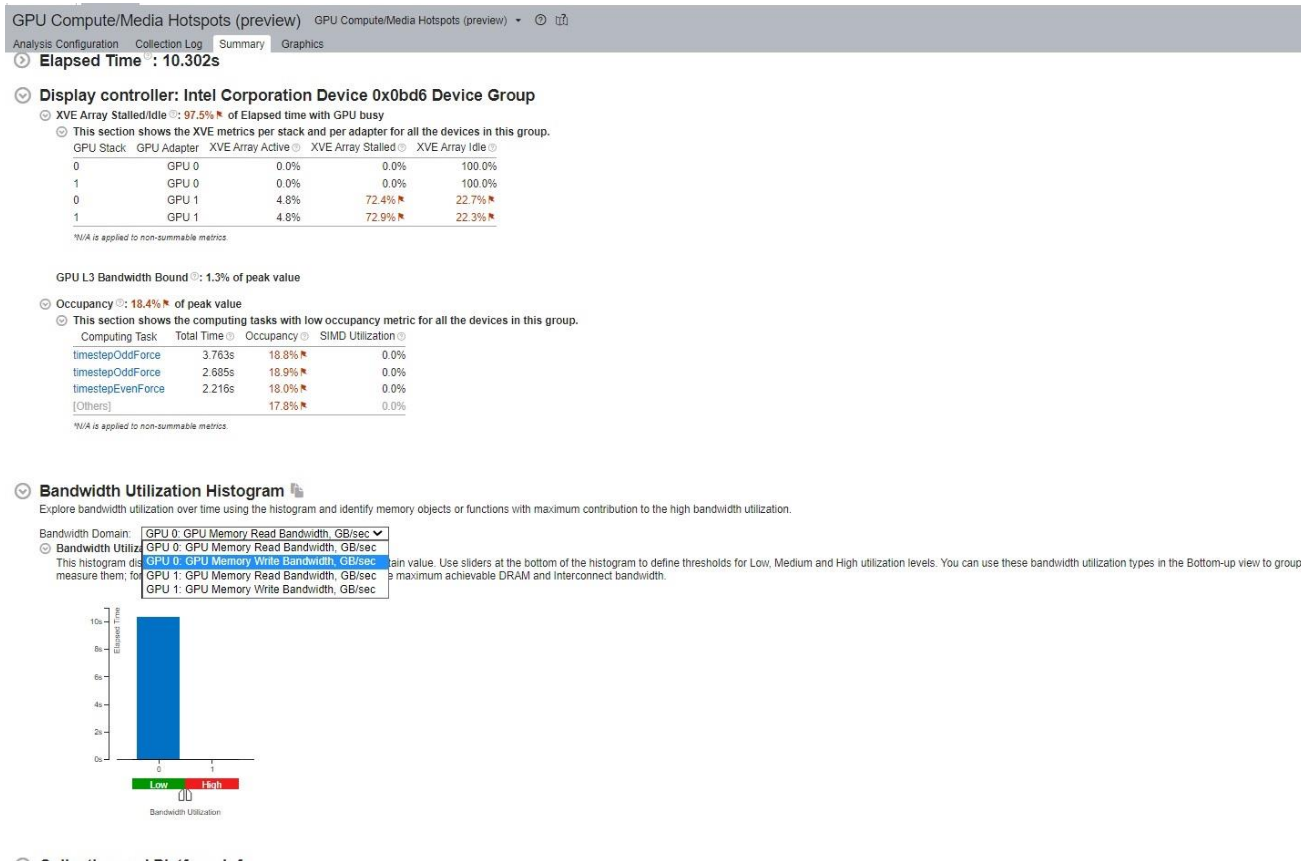
Sample Command Line:

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-offload -r {result_dir}
./{your_application} {Command_line_arguments_for_your_application}
```

GPU Compute Hotspots Analysis

Deep Dive into GPU Usage

- Explore GPU kernels with high GPU utilization,
- Identify possible reasons for stalls or low occupancy and options.
- Explore the performance of your application per selected GPU metrics over time.
- Analyze the hottest SYCL* standards or OpenCL™ kernels for inefficient kernel code algorithms or incorrect work item configuration.



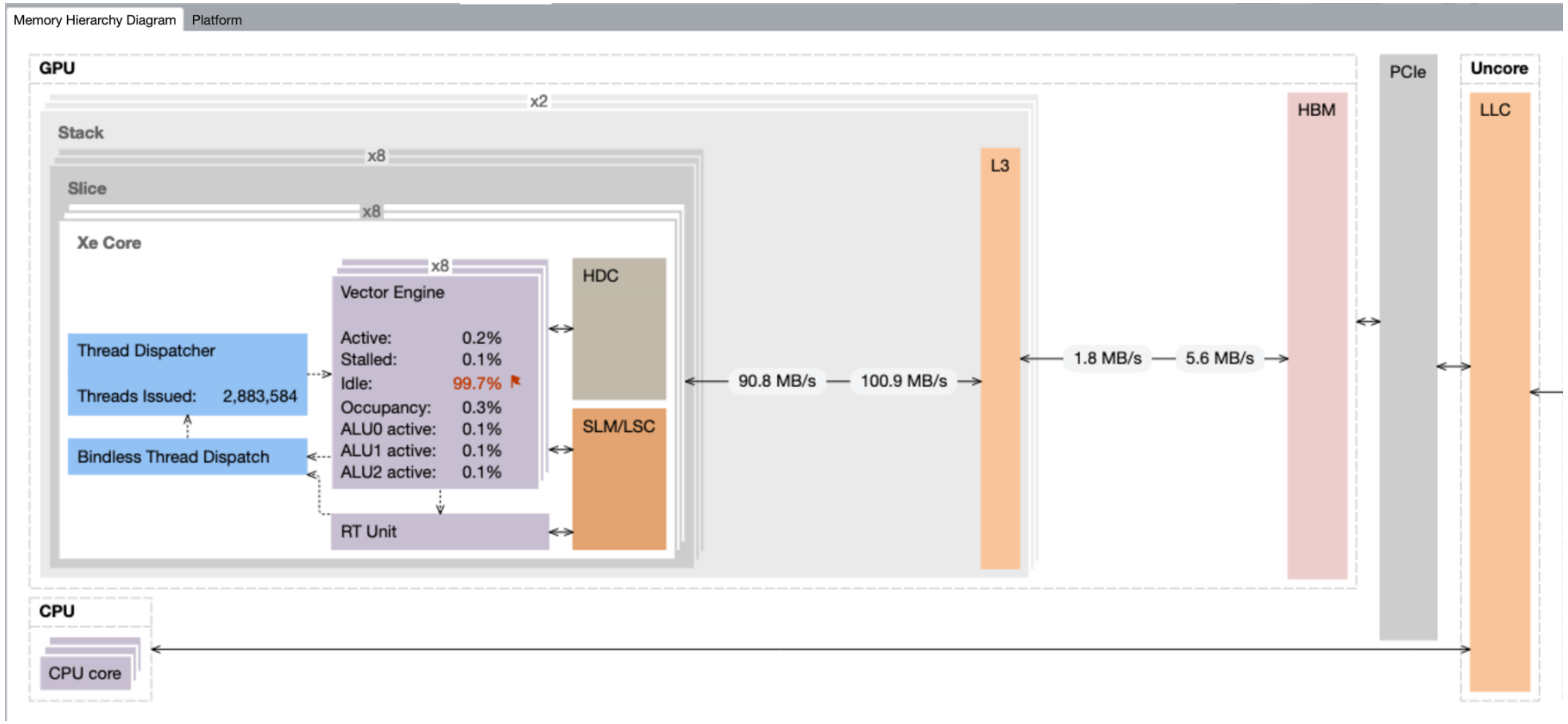
Sample Command Line:

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

GPU Compute Hotspots Analysis

Memory Hierarchy Diagram

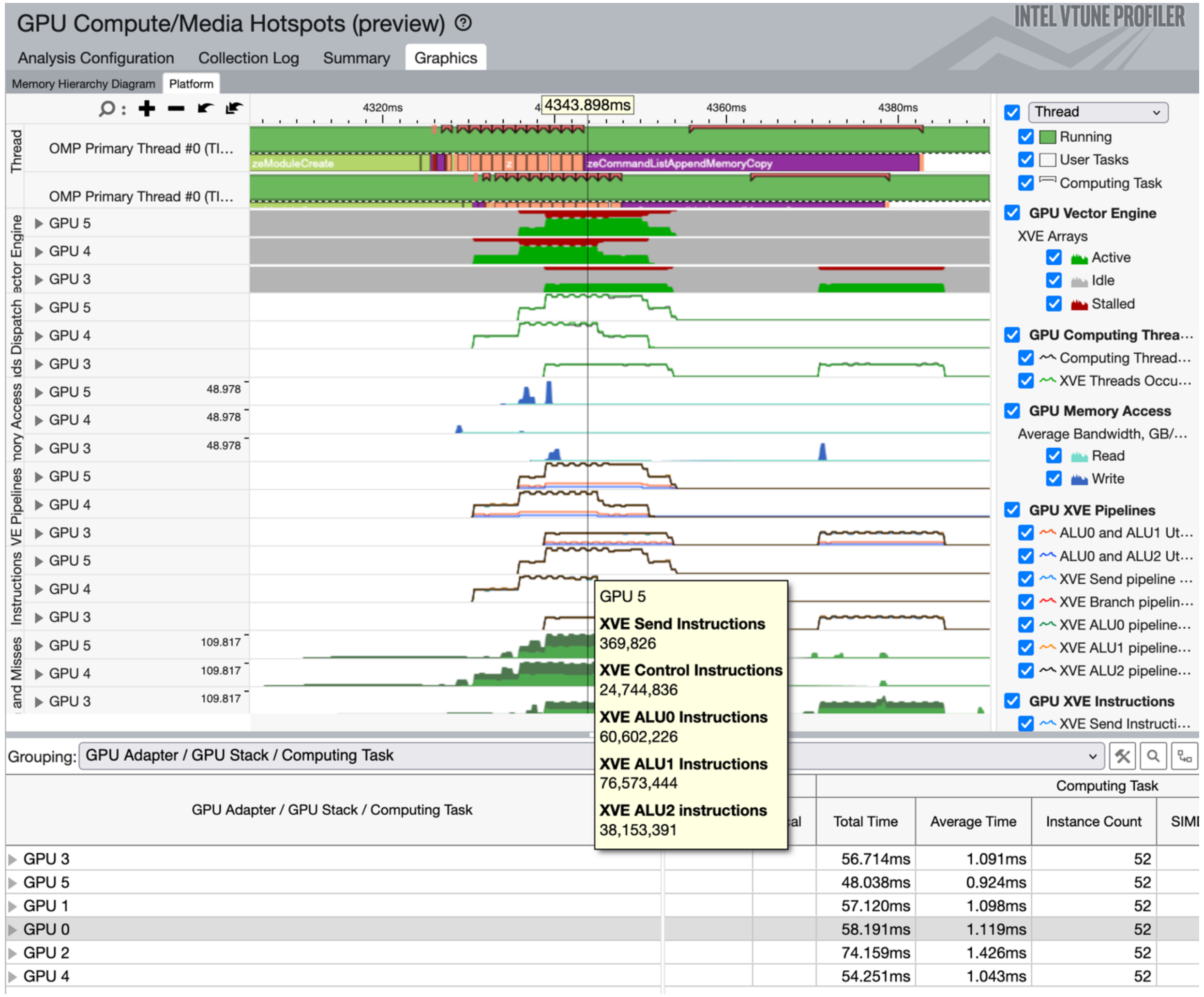
- Understand the GPU Architecture e.g. XVEs, Cores, Stacks
- Analyze data transfer/bandwidth metrics.
  - Total data movement
  - Bandwidth (Read/Write)
  - Percentage compared to Theoretical Peak
- Identify the memory/cache units that cause execution bottlenecks.
- Make decisions on data access patterns in your algorithm based on GPU microarchitectural constraints.



GPU Compute Hotspots Analysis

Platform View

- Observe correlation between Host and Device Metrics
- See the compute tasks/User tasks for eacf thread/process
- Observe the GPU Vector Engine Usage (Active, Idle, Stalled) for each GPU on the system
  - — GPU Vector Engine
  - — Computing Threads Dispatch
  - — XVE Pipeline/ Instructions
  - — GPU Busy/Frequency
- Different Memory subsystem related data:
  - — GPU Memory Access
  - — GPU L3 Cache Bandwidth and Misses

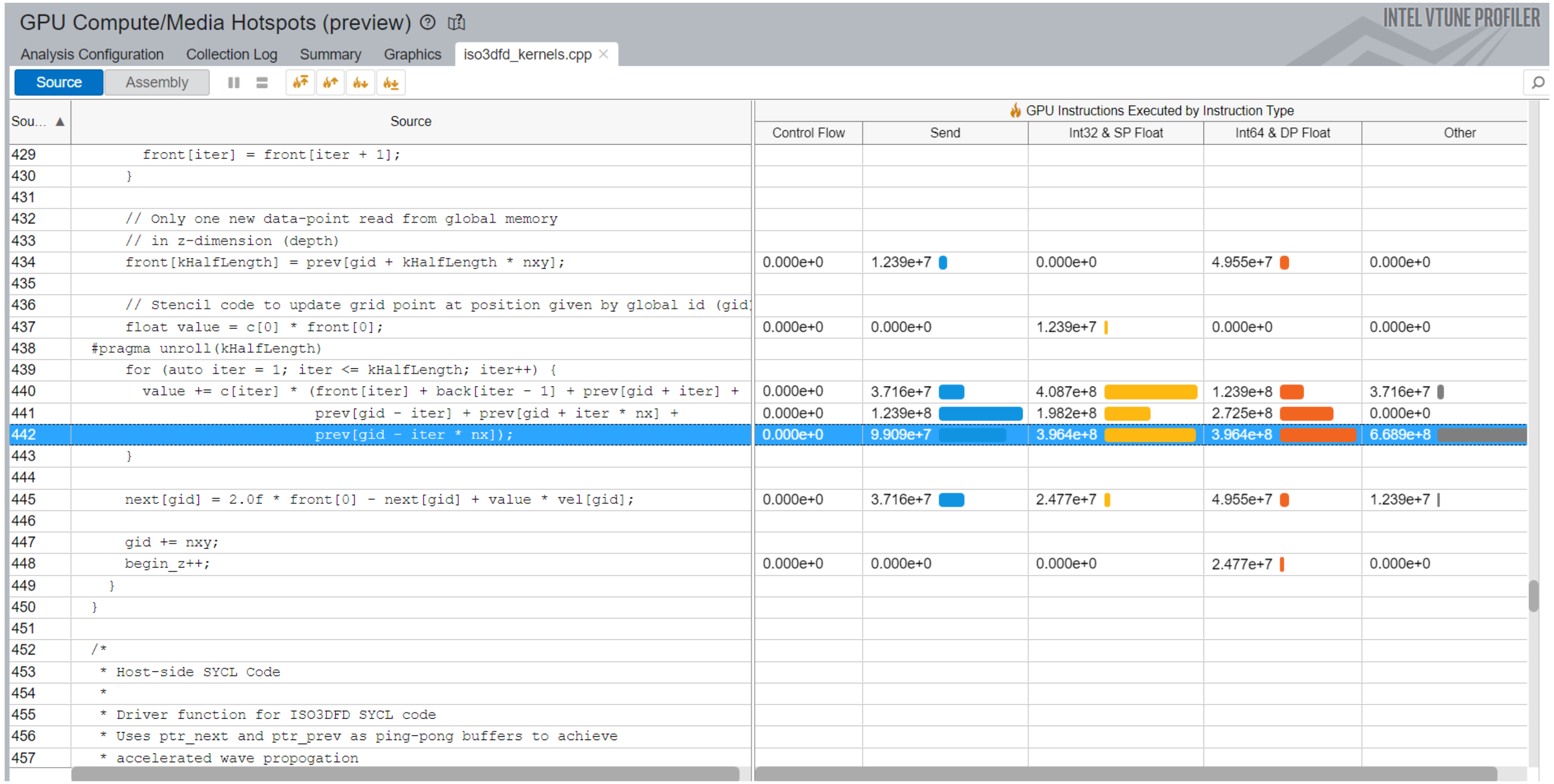


Argonne NATIONAL LABORATORY

Source level in-kernel profiling

Dynamic Instruction Count

- Counts the execution frequency of specific classes of instructions and group them:
  - — Control Flow
  - — Send
  - — Int32& SP Float
  - — Int64 & DP Float
  - — Other
- Insight into the efficiency of SIMD utilization by each kernel.
- Sample Command Line:



Sou...	Source	GPU Instructions Executed by Instruction Type: Control Flow	Send	Int32 & SP Float	Int64 & DP Float	Other
429	`front[iter] = front[iter + 1];`					
430	`}`					
431						
432	`// Only one new data-point read from global memory`					
433	`// in z-dimension (depth)`					
434	`front[kHalfLength] = prev[gid + kHalfLength * nxy];`	0.000e+0	1.239e+7	0.000e+0	4.955e+7	0.000e+0
435						
436	`// Stencil code to update grid point at position given by global id (gid)`					
437	`float value = c[0] * front[0];`	0.000e+0	0.000e+0	1.239e+7	0.000e+0	0.000e+0
438	`#pragma unroll(kHalfLength)`					
439	`for (auto iter = 1; iter <= kHalfLength; iter++) {`					
440	`value += c[iter] * (front[iter] + back[iter - 1] + prev[gid + iter] +`	0.000e+0	3.716e+7	4.087e+8	1.239e+8	3.716e+7
441	`prev[gid - iter] + prev[gid + iter * nx] +`	0.000e+0	1.239e+8	1.982e+8	2.725e+8	0.000e+0
442	`prev[gid - iter * nx]);`	0.000e+0	9.909e+7	3.964e+8	3.964e+8	6.689e+8
443	`}`					
444						
445	`next[gid] = 2.0f * front[0] - next[gid] + value * vel[gid];`	0.000e+0	3.716e+7	2.477e+7	4.955e+7	1.239e+7
446						
447	`gid += nxy;`					
448	`begin_z++;`	0.000e+0	0.000e+0	0.000e+0	2.477e+7	0.000e+0
449	`}`					
450	`}`					
451						
452	`/*`					
453	`* Host-side SYCL Code`					
454	`*`					
455	`* Driver function for ISO3DFD SYCL code`					
456	`* Uses ptr_next and ptr_prev as ping-pong buffers to achieve`					
457	`* accelerated wave propogation`					

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -knob characterization-mode=instruction-count -r {result_dir}
./{your_application} {Command_line_arguments_for_your_application}
```


Argonne NATIONAL LABORATORY

Source level in-kernel profiling requires building apps with “***-fdebug-info-for-profiling -gline-tables-only***”)

Basic Block Latency Analysis

- Helps identify issues caused by algorithm inefficiencies
- Measures the execution time of all basic blocks
- Calculates the execution time for each instruction in the basic block
- Useful for finding out the expensive operations/ instructions
- Sample Command Line:

Source Line ▲	Source	Estimated GPU Cycles: Total	Estimated GPU Cycles: Self
24			
25	`void Comp_Geo(REAL *GeoR, REAL *GeoResult, int n, int nGeo){`		
26	`int iGeo, i, j, id;`		
27	`REAL tmpR, tmpResult;`		
28			
29	`#pragma omp target teams distribute parallel for collapse(2)`	0.1%	0.1%
30	`for(j=0;j<n;j++){`	0.0%	0.0%
31	`for(i=0;i<n;i++){`	0.0%	0.0%
32	`id = i+j*n;`	0.0%	0.0%
33	`tmpR = GeoR[id];`	0.0%	0.0%
34	`tmpResult = 1.0E0;`		
35	`for (iGeo=1;iGeo<=nGeo;iGeo++){`	3.9%	3.9%
36	`tmpResult = 1.0E0 + tmpR*tmpResult;`	4.3%	4.3%
37	`}`		
38	`GeoResult[id] = tmpResult;`	0.0%	0.0%
39	`}`		
40	`}`		

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

Source level in-kernel profiling requires building apps with “***-fdebug-info-for-profiling -gline-tables-only***”)

Memory Latency Analysis

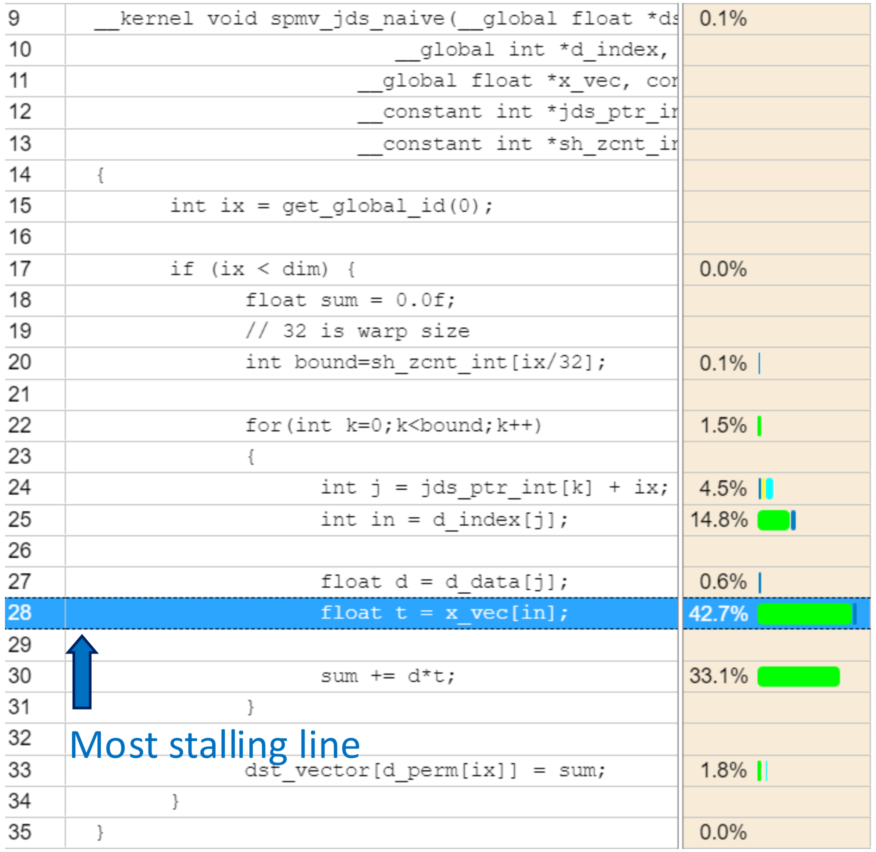
- Used for memory/Throghput bound applications
- Helps identify latency issues caused by memory accesses
- Profiles memory read/synchronization instructions to estimate their impact on the kernel execution time
- Explore which memory read/synchronization instructions from the same basic block take more time
- Sample Command Line:

Sou Lin	Source	Average Latency, Cycles: Total	Average Latency, Cycl…		Estimated GPU Cycles: Total	Estimated GPU Cycles: Self
			Memory Re…	Synchro…		
19	`#else`					
20	`typedef double REAL;`					
21	`#endif`					
22	`int PR=sizeof(REAL);`					
23						
24						
25	`void Comp_Geo(REAL *GeoR, REAL *GeoResult, int n, int nGeo){`					
26	`int iGeo, i, j, id;`					
27	`REAL tmpR, tmpResult;`					
28						
29	`#pragma omp target teams distribute parallel for collapse(2)`					
30	`for(j=0;j<n;j++){`					
31	`for(i=0;i<n;i++){`					
32	`id = i+j*n;`					
33	`tmpR = GeoR[id];`	118.9%	770	0	9.9%	9.9%
34	`tmpResult = 1.0E0;`					
35	`for (iGeo=1;iGeo<=nGeo;iGeo++){`					
36	`tmpResult = 1.0E0 + tmpR*tmpResult;`					
37	`}`					
38	`GeoResult[id] = tmpResult;`					
39	`}`					

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -knob source-analysis=mem-latency -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

Hardware Assisted Stall Sampling

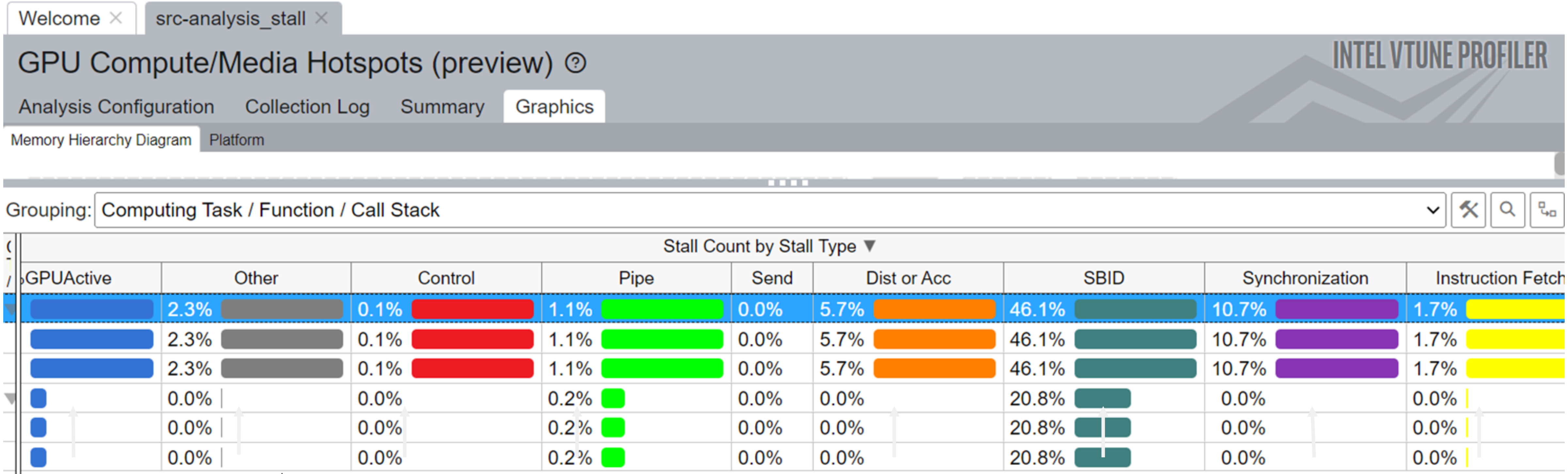
- Provides detailed breakdown of stalls and reasons
- HW-assisted Stall Sampling technology designed for Intel® Data Center GPU Max Series (code-named Ponte Vecchio or PVC)
- Capabilities similar to instruction execution efficiency characterization of NVIDIA® **Nsight™ Compute**
- Sample Command Line:




```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -knob source-analysis=stall-sampling  -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

Source level in-kernel profiling requires building apps with "*-fdebug-info-for-profiling -gline-tables-only*")

Hardware Assisted Stall Sampling



GPUActive	Other	Control	Pipe	Send	Dist or Acc	SBID	Synchronization	Instruction Fetch
	2.3%	0.1%	1.1%	0.0%	5.7%	46.1%	10.7%	1.7%
	2.3%	0.1%	1.1%	0.0%	5.7%	46.1%	10.7%	1.7%
	2.3%	0.1%	1.1%	0.0%	5.7%	46.1%	10.7%	1.7%
	0.0%	0.0%	0.2%	0.0%	0.0%	20.8%	0.0%	0.0%
	0.0%	0.0%	0.2%	0.0%	0.0%	20.8%	0.0%	0.0%
	0.0%	0.0%	0.2%	0.0%	0.0%	20.8%	0.0%	0.0%

- GPUActive: Actively executing in at least one pipeline
- Other: Stalled Due to other reasons
- Control: Stalled due to control flow dependencies
- Pipe: Stalled due to due to data hazards, structural hazards, or resource unavailability.
- Send: Instructions dependent on the result of the send operation can't proceed, stalling the pipeline
- Dist or Acc: Stalls due to a delay in distributing /accumulation of instructions to the appropriate XVEs
- SBID: Stalled due to dependencies tracked by the scoreboard mechanism e.g. data hazards
- Synchronization: Stalled due to sync operation e.g. barrieir ync, shared mem. contention
- Instruction Fetch: Stalled due to a delay in retrieving the next instruction to execute

Argonne NATIONAL LABORATORY

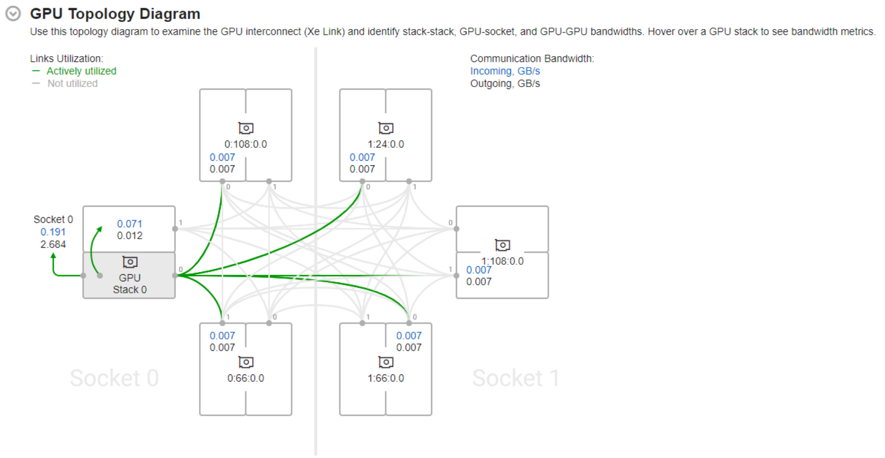
Get Visibility into Xe Link Cross-card Traffic

Intel® VTune™ Profiler

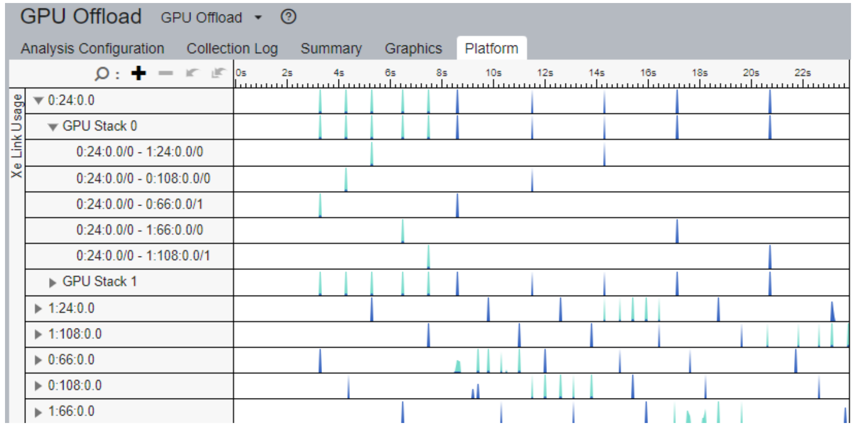
Identify bottlenecks related to Xe Link

- Understand cross-card memory transfers and Xe Link utilization
- Visualize GPU Topology of the system and estimate bandwidth of each link, stack or card.
- See usage of Xe Link and correlate with code execution.
- Sample Command Line:

$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-offload –knob `analyze-xelink-usage=true` -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}



Cross-card, stack-to-stack, and card-to-socket bandwidth are presented on GPU Topology Diagram.



Timeline view can show bandwidth usage of Xe Link over time.

Profile your oneCCL workloads using VTune Profiler

Supported using Application Performance Snapshot(APS) and HPC Performance Analysis

- APS:
  - Time spent on CCL Tasks
  - Percentage of total time
- HPC Performance Characterization
  - oneCCL Time
  - Top oneCCL Tasks
  - oneCCL communication tasks in the Summary window and on the Timeline

CCL Time : 4.320

Top oneCCL Tasks

This section lists the most active oneCCL communication tasks in your application.

Task Type	CCL Time	CCL Call Count
oneCCL::SYNC	1.048	12
oneCCL::oneCCL::Imbalance	1.000	2
oneCCL::barrier	1.000	2
oneCCL::BARRIER	0.999	2
oneCCL::allreduce	0.093	2
[Others]	0.181	28

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

CCL Time

17.99 s

9.34% of Elapsed Time

TOP 5 CCL Functions	% of Elapsed Time
oneCCL::allreduce	9.09%
oneCCL::bcast	0.25%

```
Elapsed Time:                                36.61 s
  MPI Time:                                   1.28 s          3.50% of Elapsed Time
    MPI Imbalance:                            0.02 s          0.07% of Elapsed Time
    Top 5 MPI functions (avg time):
        MPI_Init_thread:                      1.21 s          3.30% of Elapsed Time
        MPI_Comm_create_group:                0.02 s          0.06% of Elapsed Time
        MPI_Comm_split_type:                  0.02 s          0.04% of Elapsed Time
        MPI_Test:                             0.01 s          0.03% of Elapsed Time
        MPI_Wait:                             0.01 s          0.02% of Elapsed Time
  CCL Time:                                  14.54 s         39.70% of Elapsed Time
| Your application is CCL bound. This may be caused by high busy wait time
| inside the library (imbalance), non-optimal communication schema or CCL
| library settings.
    Top 5 CCL functions (avg time):
        oneCCL::allreduce:                   14.50 s         39.61% of Elapsed Time
        oneCCL::bcast:                        0.03 s          0.07% of Elapsed Time
        oneCCL::allgatherv:                   0.01 s          0.02% of Elapsed Time
        oneCCL::barrier:                      0.00 s          0.00% of Elapsed Time
```

Minimizing Collection Overhead Using VTune Knobs

- Disabling Stack Collection
  - Use the -knob enable-stack-collection=false option.

  Sample Command Line:

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-offload –knob enable-stack-collection=false -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

- Modifying sampling interval
  - Use the -knob gpu-sampling-interval=<value> option.
  - Sample Command Line:

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots –knob gpu-sampling-interval=10 -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

- Specify computing-tasks-of-interest
  - Specify comma-separated list of GPU computing task names.
  - Sample Command Line:

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots –knob computing-tasks-of-interest=*kernel* -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

- More details can be found here.

Selective Rank Profiling

- Using VTune you can profile specific ranks of interest of an MPI application
- Pros:
  - —Reduction in collection overhead
  - —Reduction in finalization time
  - —Reduction in storage overhead
- Sample Command Line:

```
mpiexec -np 1 --cpu-bind=${CPU_BIND}  gpu_tile_compact.sh vtune –c gpu-hotspots -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application} : -np 23 --cpu-bind=${CPU_BIND} gpu_tile_compact.sh {Command_line_arguments_for_your_application}
```

Selective Rank Profiling Contd.

- Using if-else block for selective launching (HACC):

```
time mpiexec -n 12 -ppn 12 --cpu-bind list:1-8:9-16:17-24:25-32:33-40:41-48:49-51:53-
60:61-68:69-76:77-84:85-92:93-100 --envall
./gpu_tile_compact.sh bash -c ‘

export SELECTED_RANK=$RANK_TO_BE_PROFILED
echo "Running MPI rank $PMIX_RANK..."
if [[ $PMIX_RANK -eq $SELECTED_RANK ]]; then
        echo "Profiling MPI rank $PMIX_RANK with VTune..."
        vtune -c gpu-offload -r ghs_rank_whole -- ./hacc_p3m_2 -n indat.params 1>
hacc.stdout_2.txt 2> hacc.stderr_2.txt
else
        echo "Profiling other MPI ranks with VTune..."
        ./hacc_p3m_2 -n indat.params 1> hacc.stdout_other.txt 2> hacc.stderr_other.txt
fi
'
```

Performance Benefits of Selective Profiling in HACC Application (Without Finalization)

- Execution Time

No. of Nodes	Application Wall Time	Profiled 1 rank	Profiled all ranks
1	52s	1m 4s	1m 24s
4	36s	47s	5m 15s

- Storage Overhead:

No. of Nodes	Profiled 1 rank	Profiled all ranks
1	248 MB	292 MB
4	360 MB	401 MB

Performance Benefits (With Finalization)

- Execution Time

Application Name	Profiled 1 rank	Profiled all (12) ranks
HACC	3m 12s	7m 36s

- Storage Overhead:

Application Name	Profiled 1 rank	Profiled all (12) ranks
HACC	1.1 GB	2.6 GB

Specifying Target GPUs

- Applications running on Multiple GPUs can benefit from the vtune knob **target-gpu**
- See the BDFs (Bus:Device:Function) of all the GPUs:

```
$ vtune --help collect gpu-hotspots
```

- Sample usage of the target-gpu knob:

```
vtune –collect gpu-hotspots –knob target-gpu 0:24:0.0 ./app
```

N.B.: By default the target-gpu selects all the GPUs on that node

Difference between selective rank vs target-gpu

Combination of them is recommended to minimize unnecessary data

Performance Improvement after Using target-gpu in HACC Application

- Execution Time:

Application Name	Profiled 1 GPU	Profiled all(6) GPUs
HACC	5m 1s	7m 36s

- Storage Overhead:

Application Name	Profiled 1 GPU	Profiled all (6) GPUs
HACC	2.1 GB	2.6 GB

Disabling Programming API Collection

- Set collect-programming-api=false
- Disable the call stack collection on GPU side
- Supported analysis: gpu-hotspots, gpu-offload, runsa

ITT APIs

- Key Features:
  - Controls application performance overhead based on the amount of traces that you collect.
  - Enables trace collection without recompiling your application.
  - Supports applications in C/C++ and Fortran environments on Windows*, Linux* systems.
  - Supports instrumentation for tracing application code.
- Build Configuration:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/aurora/24.347.0/oneapi/vtune/latest/lib64
$  export VTUNE_DIR=/opt/aurora/24.347.0/oneapi/vtune/latest
$  icx test.c –I$(VTUNE_DIR)/include –L$(VTUNE_DIR)/lib64 -littnotify -lpthread -ldl  -o test
```

Sample Code:

```
#include <ittnotify.h>
int main() {
   auto* domain = __itt_domain_create("Example.Domain");
   auto* task = __itt_string_handle_create("QuickTask");
   __itt_task_begin(domain, __itt_null, __itt_null, task);
//dummy work
    __itt_task_end(domain);
}
```

Annotating Python code using instrumentation

Pyitt APIs

- Python binding to Intel Instrumentation and Tracing Technology (ITT) API
- Features:
  - Controls application performance overhead based on the traces you collect.
  - Convenient way to mark up the Python code
  - Comes with easy-to-use wrappers
  - Very useful for Large AI and HPC workload
- Installation
  - PyPi package: pip install pyitt
  - Build from source: https://github.com/intel/ittapi
- C++ APIs
  - Bundled with VTune: C/C++ ITT APIs

```
import pyitt
@pyitt.task
def workload():
    pass
workload()
```

Annotating PyTorch Code using Native APIs

PyTorch* Framework with ITT APIs

1. is_available()
2. mark(msg)
3. range_push(msg)
4. range_pop()

```
itt.resume()
with torch.autograd.profiler.emit_itt():
        torch.profiler.itt.range_push('training')
        model.train()
        for batch_index, (data, y_ans) in enumerate(trainLoader):
            data = data.to(memory_format=torch.channels_last)
            optim.zero_grad()
            y = model(data)
            loss = crite(y, y_ans)
            loss.backward()
            optim.step()
        torch.profiler.itt.range_pop()
itt.pause()
```

1. Resume collection of profiling data.
2. To enable the explicit invocation, we use the torch.autograd.profiler.emit_itt() API right before the interesting code that we want to profile.
3. Push a range onto a stack of nested range span and mark it with a message (‘training’).
4. Pop a range from the stack of nested range spans using range_pop() API.
5. Pause the profiling data collection using itt.pause() API.

VTune Web Server

Visualizing VTune Results on Aurora

Step 1: Add the following lines to .ssh/config on your local system

```
host *.alcf.anl.gov
        ControlMaster auto
ControlPath ~/.ssh/ssh_mux_%h_%p_%r
```

Step 2: Open a new terminal and log into an Aurora login node (no X11 forwarding required)

```
$ ssh <username>@login.aurora.alcf.anl.gov
```

Step 3: Start VTune server on an Aurora login node

```
$ module load oneapi/release/2025.0. 5
$ vtune-backend --data-directory=<location of precollected VTune results>
```

Step 4: Open a new terminal with SSH port forwarding enabled

```
$ ssh -L 127.0.0.1:<port printed by vtune-backend>:127.0.0.1:<port printed by vtune-backend> <username>@login.aurora.alcf.anl.gov
```

Step 5: Open the URL printed by VTune server in firefox web browser on your local computer.

Intel® VTune™ Profiler CLI

characterization with gpu-offload and default knobs

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-offload  -r {result_dir} ./{your_application}
 {Command_line_arguments_for_your_application}
```

characterization with gpu hotspots and default knobs

```
mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots  -r {result_dir} ./{your_application}
 {Command_line_arguments_for_your_application}
```

characterization with gpu hotspots and instruction count

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -knob characterization-mode=instruction-count -r {result_dir}
 ./{your_application} {Command_line_arguments_for_your_application}
```

source analysis with gpu hotspots [with basic block latency - default]

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -r {result_dir} ./{your_application}
 {Command_line_arguments_for_your_application}
```

source analysis with gpu hotspots and memory latency

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -knob source-analysis=mem-latency
 -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

source analysis with gpu hotspots and stall sampling

```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -knob source-analysis=stall-sampling
 -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

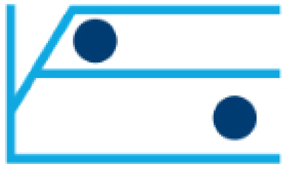
Intel® Advisor

Rich Set of Capabilities for High Performance Code Design

Intel® Advisor

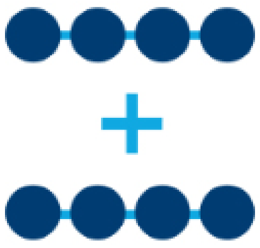
Offload Advisor

Design offload strategy and model performance on GPU.



Roofline Analysis

Optimize your application for memory and compute.

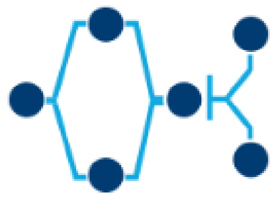


Vectorization Optimization

Enable more vector parallelism and improve its efficiency.

Thread Prototyping

Model, tune, and test multiple threading designs.



Build Heterogeneous Algorithms

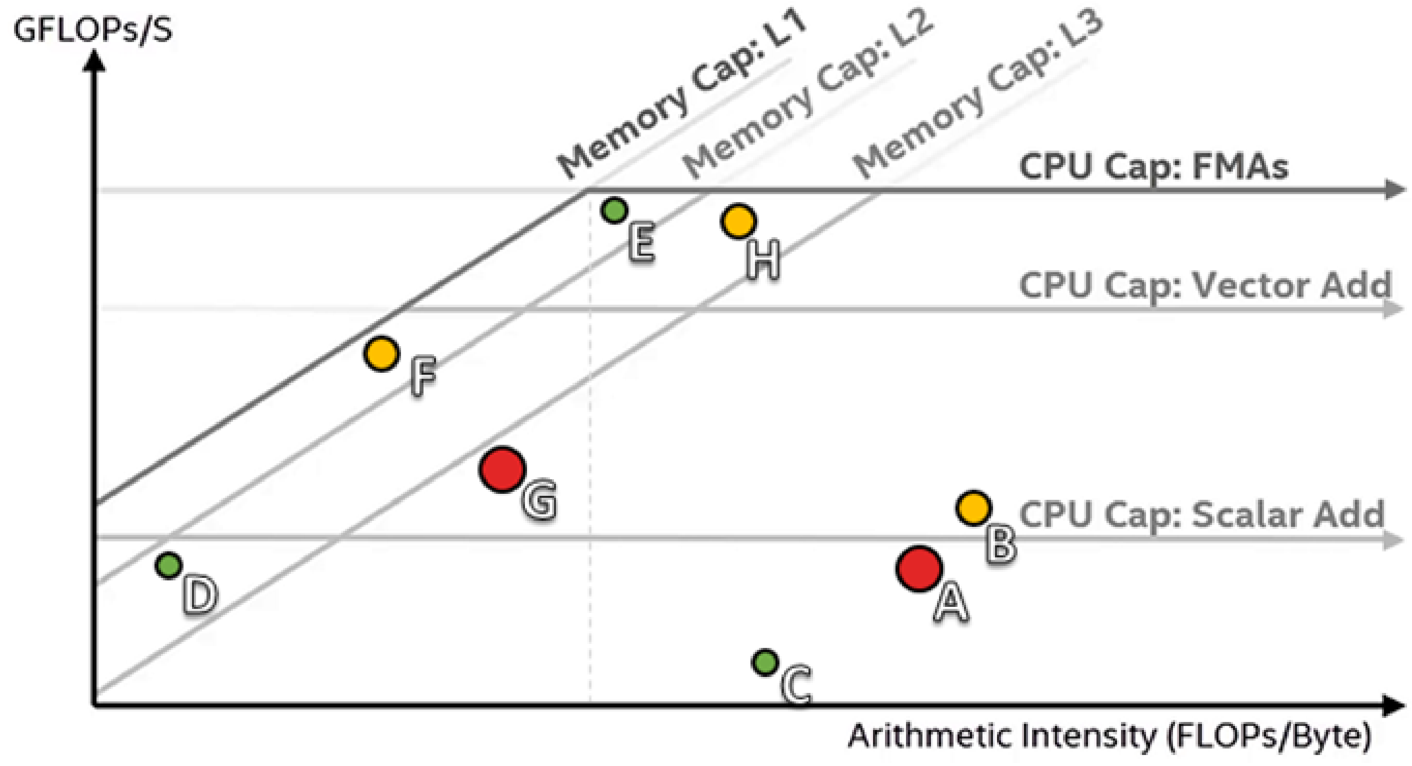
Create and analyze data flow and dependency computation graphs.

Identifying Good Optimization Candidates

Focus optimization effort where it makes the most difference

- Large, red loops have the most impact
- Loops far from the upper roofs have more room to improve

Additional roofs can be plotted for specific computation types or cache levels



Configuring Intel Advisor on Aurora

- Step1: Setting the environments

```
$ module load oneapi
$ export PRJ=<your_project_dir>
```

- Step 2-a: Collecting the GPU Roofline data on a single GPU (Survey analysis and Trip Count with FLOP analysis with a single command line)

```
$ advisor --collect=roofline --profile-gpu --project-dir=$PRJ -- <your_executable> <your_arguments>
```

- Step 2-b: Collecting the GPU Roofline data on one of MPI ranks (Survey analysis and Trip Count with FLOP analysis separately)

```
$ mpirun -n 1 gpu_tile_compact.sh advisor --collect=survey --profile-gpu --project-dir=$PRJ -- <your_executable> <your_arguments> : -n 11 gpu_tile_compact.sh <your_executable> <your_arguments>

$ mpirun -n 1 gpu_tile_compact.sh advisor --collect=tripcounts --profile-gpu --flop --no-trip-counts --project-dir=$PRJ -- <your_executable> <your_arguments> : -n 11 gpu_tile_compact.sh <your_executable> <your_arguments>
```

Configuring Intel Advisor on Aurora Contd.

- Step 3: Generate a GPU Roofline report, and then review the HTML report

```
$ advisor --report=all --project-dir=$PRJ --report-output=${PRJ}/roofline_all.html
```

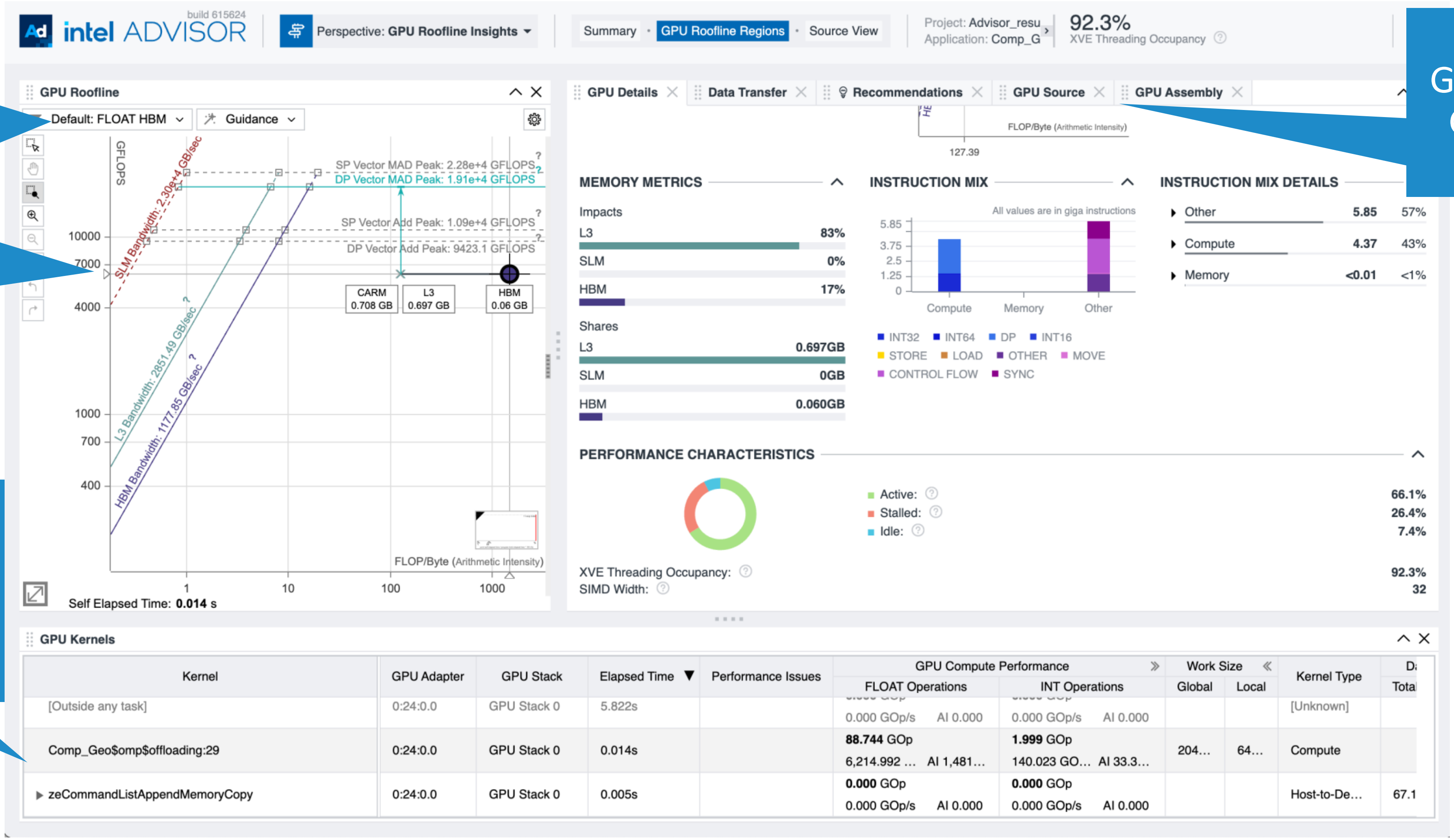
GPU Roofline Insights

Switch between data types

Customizable GPU Roofline chart

GPU performance of compute tasks

View Details, GPU Source, and GPU Assembly info



Argonne NATIONAL LABORATORY

Demo

Copy the example to your space:

```
$ cp -r /lus/flare/projects/gpu_hack/examples/tools_intel ~/

$ cd ~/tools_intel
```

Build the code:

```
$ make

mpicc -fiopenmp -fopenmp-targets=spir64  -O2 -fdebug-info-for-profiling -gline-tables-only Comp_GeoSeries_omp.c -o Comp_GeoSeries_omp_mpicc_DP

rm -rf *.o *.mod *.dSYM
```

Run the code:

```
$ mpirun -n 12 gpu_tile_compact.sh ./Comp_GeoSeries_omp_mpicc_DP 2048 1000
```

Demo

aps example :

```
$ mpirun -n 12 gpu_tile_compact.sh aps -r apsresult  ./Comp_GeoSeries_omp_mpicc_DP 2048 1000
```

```
$ aps-report --metrics=? apsresult
```

```
$ aps-report --metrics="GPU Stack Utilization Per Device, OpenMP Offload Time, GPU Accumulated Time Per Device, MPI Time" apsresult
```

vtune example :

```
$ mpirun -n 12 gpu_tile_compact.sh vtune -collect gpu-hotspots -r vtune_result_gh ./Comp_GeoSeries_omp_mpicc_DP 2048 1000
```

advisor example :

```
$ mpiexec -n 1 gpu_tile_compact.sh advisor --collect=survey --profile-gpu --project-dir=Advisor_results -- ./Comp_GeoSeries_omp_mpicc_DP 2048 1000 : -n 11 gpu_tile_compact.sh ./Comp_GeoSeries_omp_mpicc_DP 2048 1000
```

```
$ mpiexec -n 1 gpu_tile_compact.sh advisor --collect=tripcounts --profile-gpu --flop --no-trip-counts --project-dir=Advisor_results -- ./Comp_GeoSeries_omp_mpicc_DP 2048 1000 : -n 11 gpu_tile_compact.sh ./Comp_GeoSeries_omp_mpicc_DP 2048 1000
```

```
$ advisor --report=all --project-dir=Advisor_results --report-output=Advisor_results/roofline_all.html
```