WORKSHOP

# ALCF Hands-on HPC Workshop

September 23-25 & October 7-9, 2025
Argonne National Laboratory

U.S. DEPARTMENT of ENERGY

Argonne NATIONAL LABORATORY

# Intel Tools on Aurora

## APS, VTune, Advisor, xpu-smi, unitrace

JaeHyuk Kwack

# Agenda

- Intel Analyzer on Aurora
  - Intel VTune/APS
  - Intel Advisor
- Light-weight Intel tools on Aurora
  - Intel unitrace
  - Intel xpu-smi

# Intel Analyzer

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

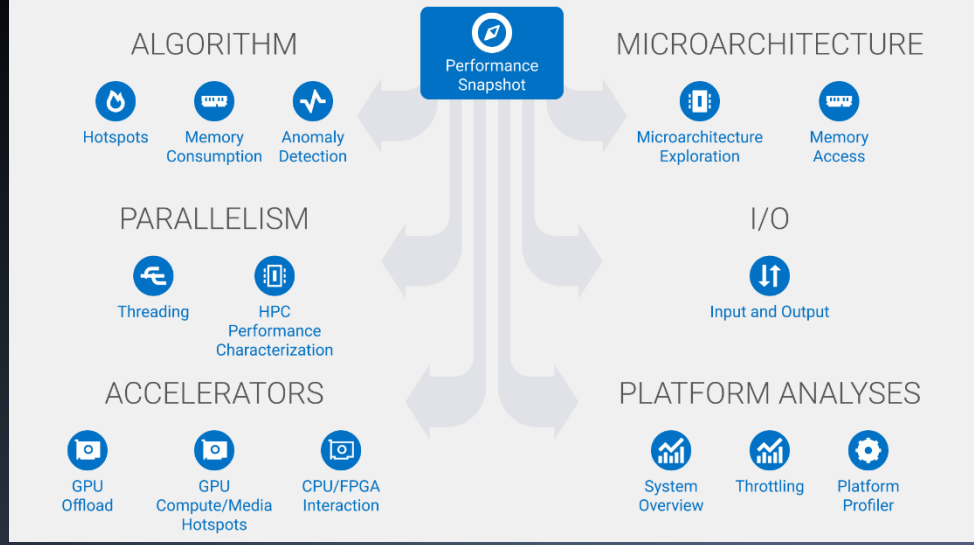
- A suite of profiling for CPU, GPU, NPU, memory, cache, storage, offload, power…
- Application or system-wide analysis
- SYCL, C, C++, Fortran, Python*, Java*, or a mix
- Linux, Windows, and more
- Containers and VMs

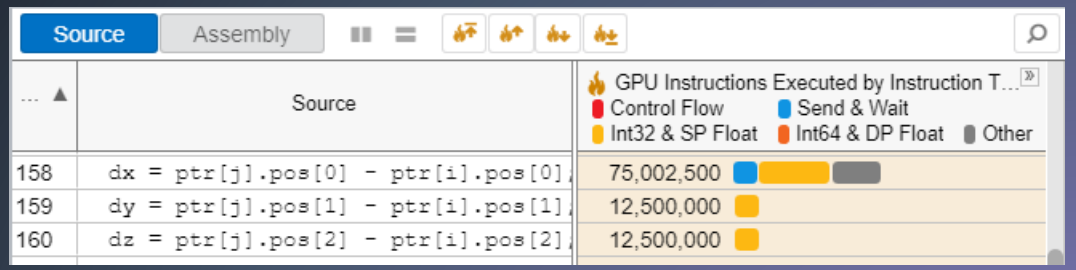
### Analyze Data Faster

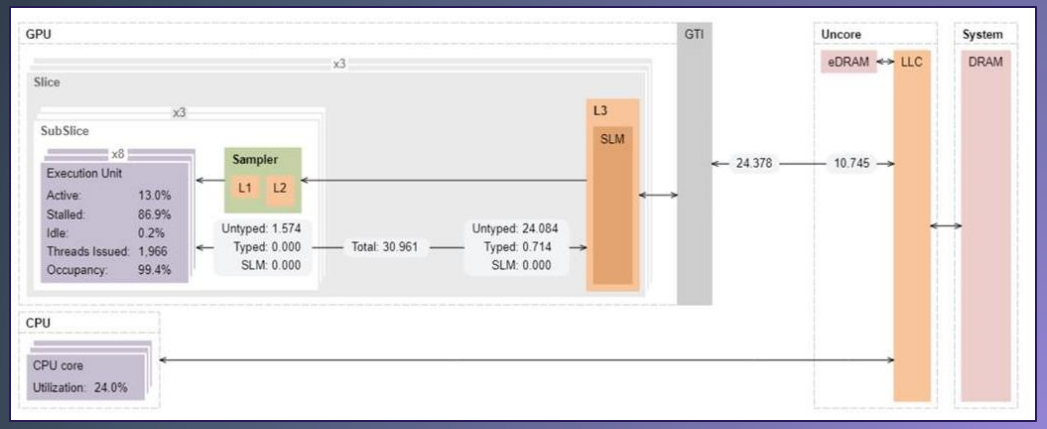
- Collect data HW/SW sampling and tracing w/o re-compilation
- See results on your source, in architecture diagrams, as a histogram, on a timeline…
- Filter and organize data to find answers

### Work Your Way

- User interface or command line
- Profile locally and remotely



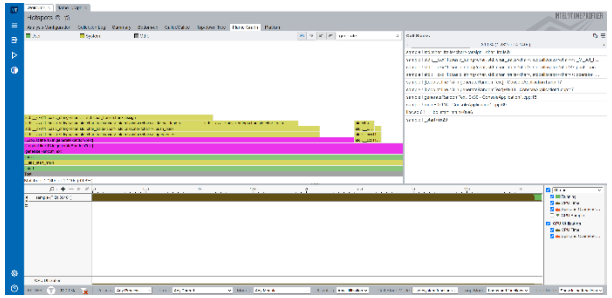






Argonne NATIONAL LABORATORY

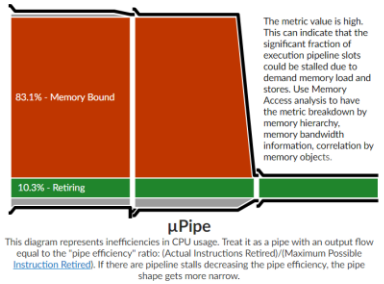
# Rich Set of Profiling Capabilities

## Intel® VTune™ Profiler



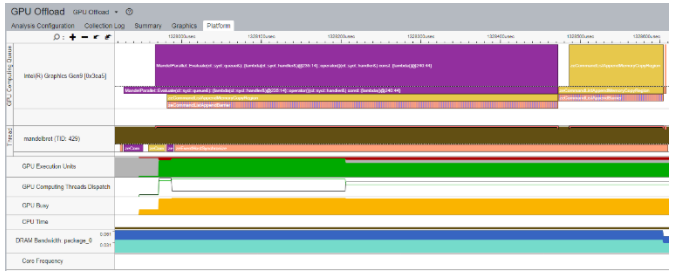
### Algorithm Optimization

- ✓ Hotspots
- ✓ Anomaly Detection
- ✓ Memory Consumption



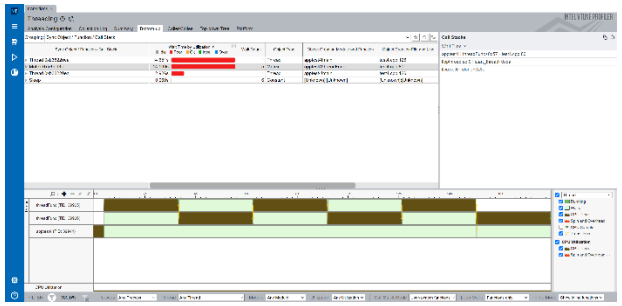

### Microarch.&Memory Bottlenecks

- ✓ Microarchitecture Exploration
- ✓ Memory Access



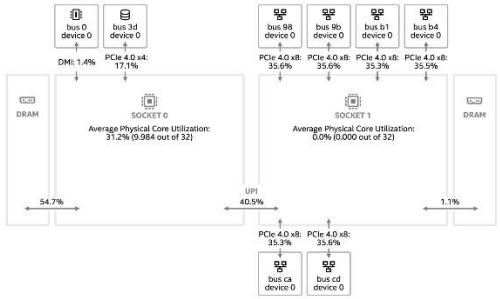

### Accelerators / xPU

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



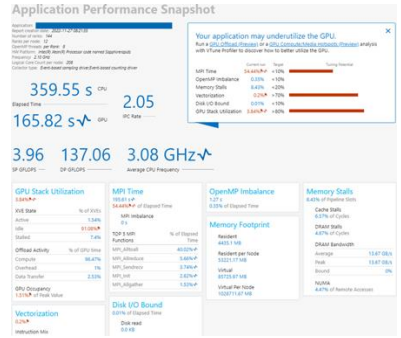
### Parallelism

- ✓ 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:**

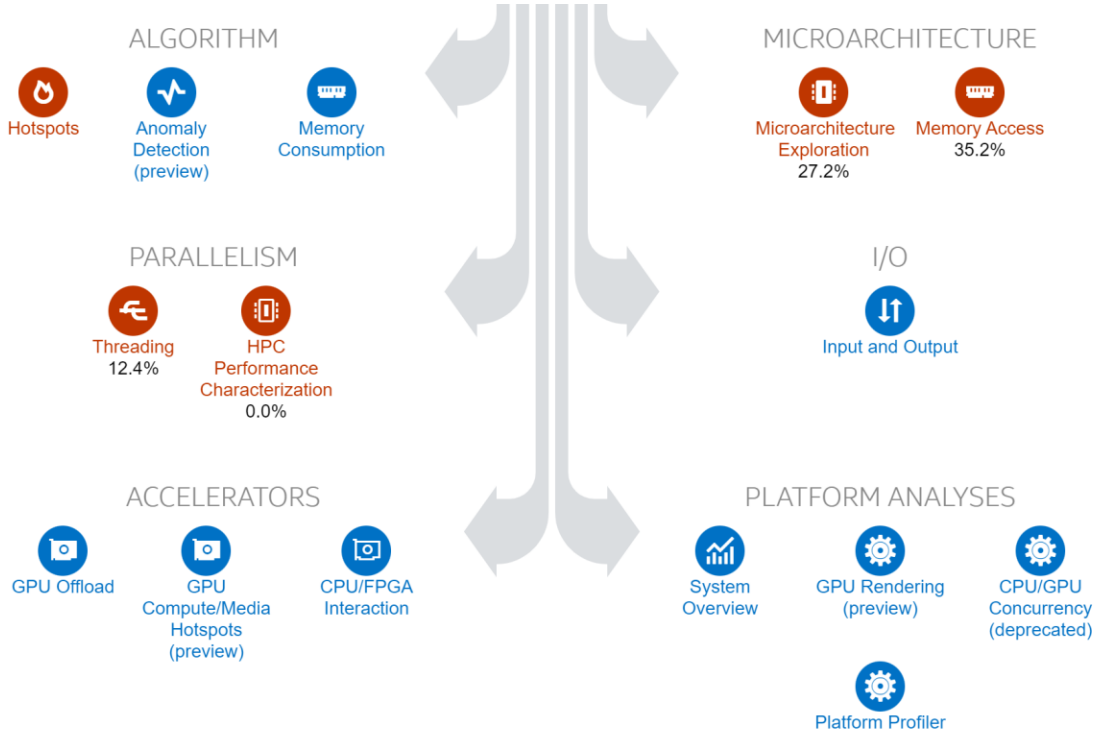
See the available knobs for a certain analysis type e.g., sampling-interval, enable-stack-collection

```
$ 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:

**Elapsed Time**: 7.906s

**Logical Core Utilization**: 12.4% (0.990 out of 8)

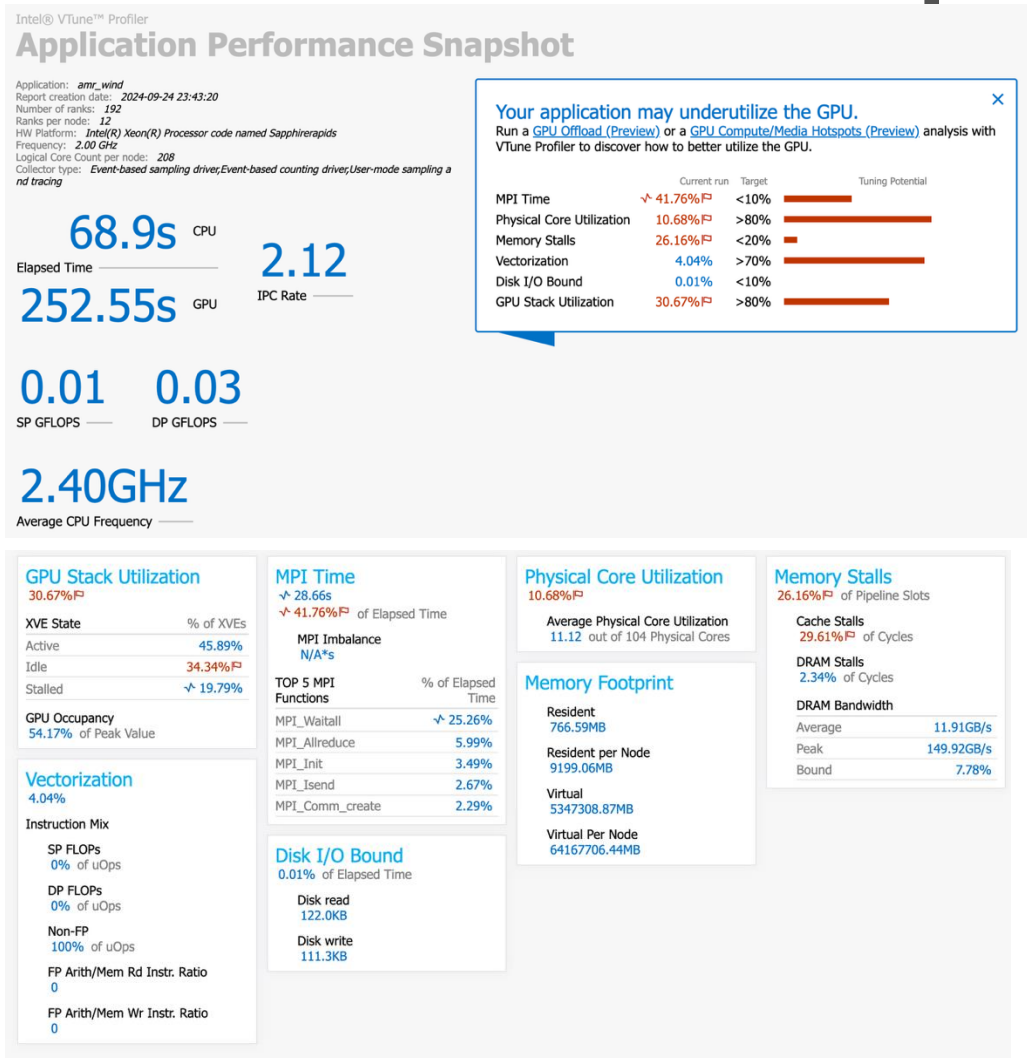
**Microarchitecture Usage**: 27.2% of Pipeline Slots

| | | |
|---|---|---|
| Retiring: | 27.2% | of Pipeline Slots |
| Front-End Bound: | 5.5% | of Pipeline Slots |
| Bad Speculation: | 3.7% | of Pipeline Slots |
| **Back-End Bound**: | 63.6% | of Pipeline Slots |
| **Memory Bound**: | 35.2% | of Pipeline Slots |
| Core Bound: | 28.4% | of Pipeline Slots |

**Memory Bound**: 35.2% of Pipeline Slots

**Vectorization**: 0.0% of Packed FP Operations

# 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}
```

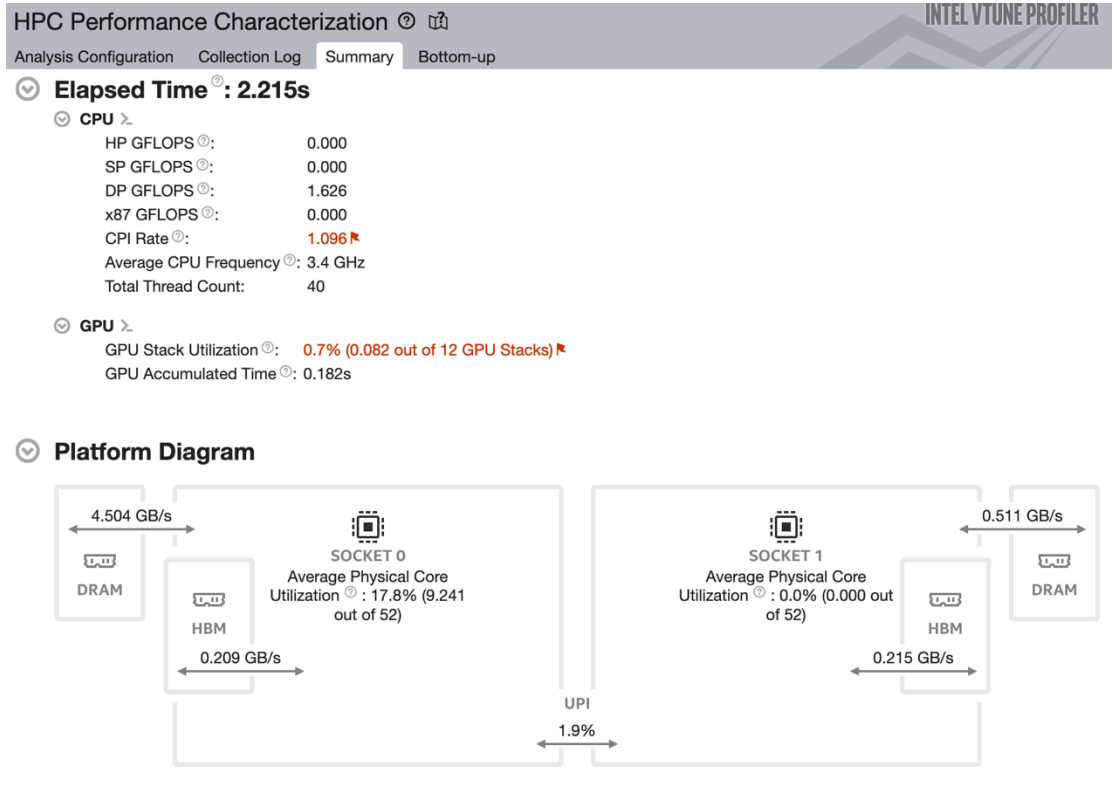
  - **HTML Report Generation:**

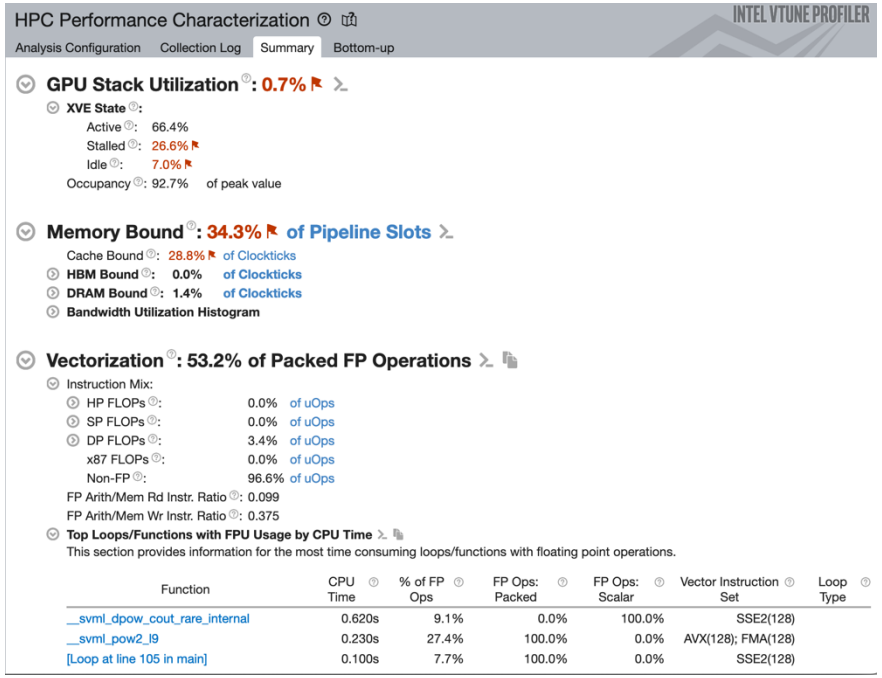
```
aps-report {aps_result_dir}
```

- **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:

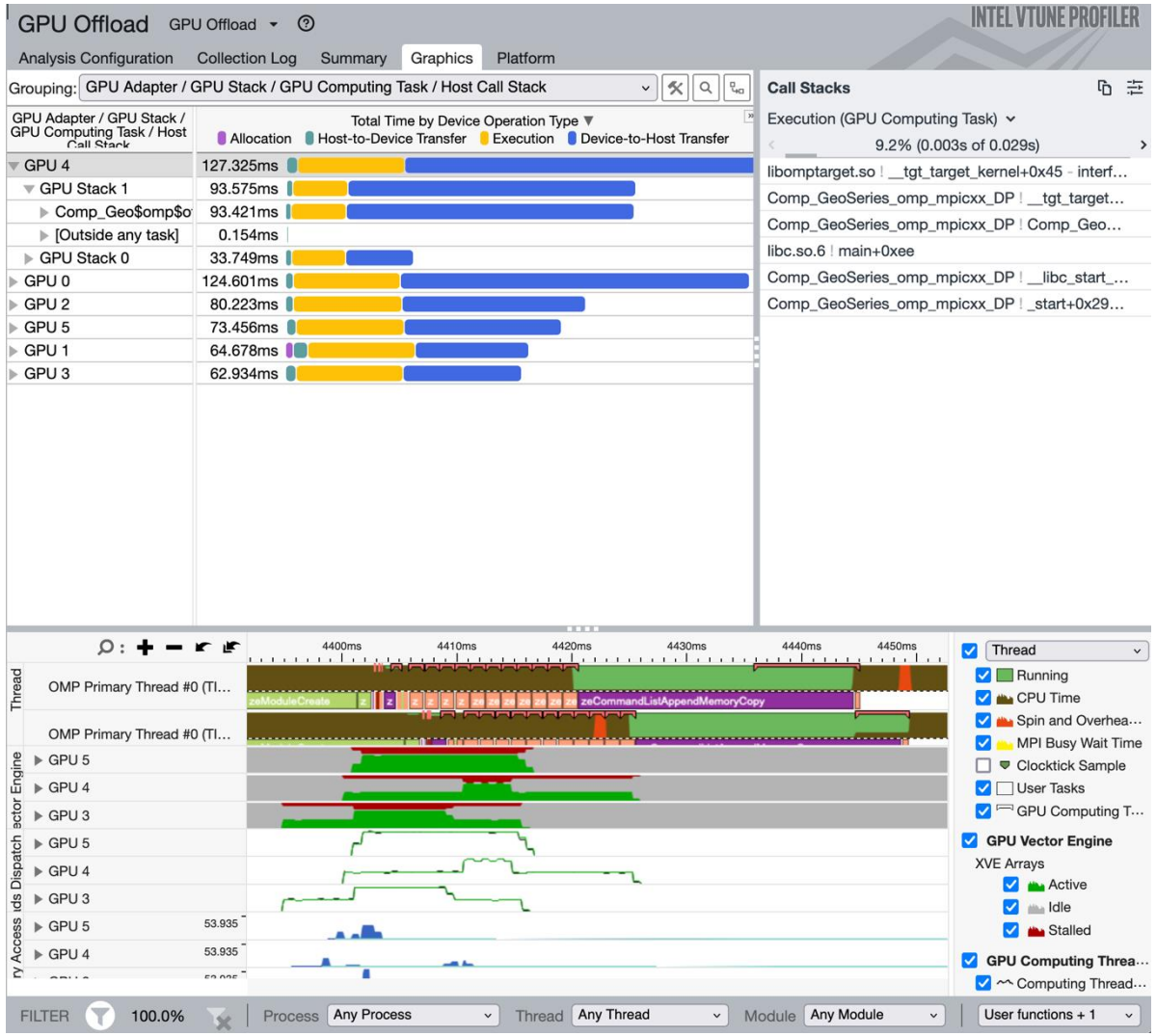
```
$ mpirun -n {Number_of_MPI} -ppn 12 gpu_tile_compact.sh vtune -collect hpc-performance -r {result_dir} ./{your_application} {Command_line_arguments_for_your_application}
```

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



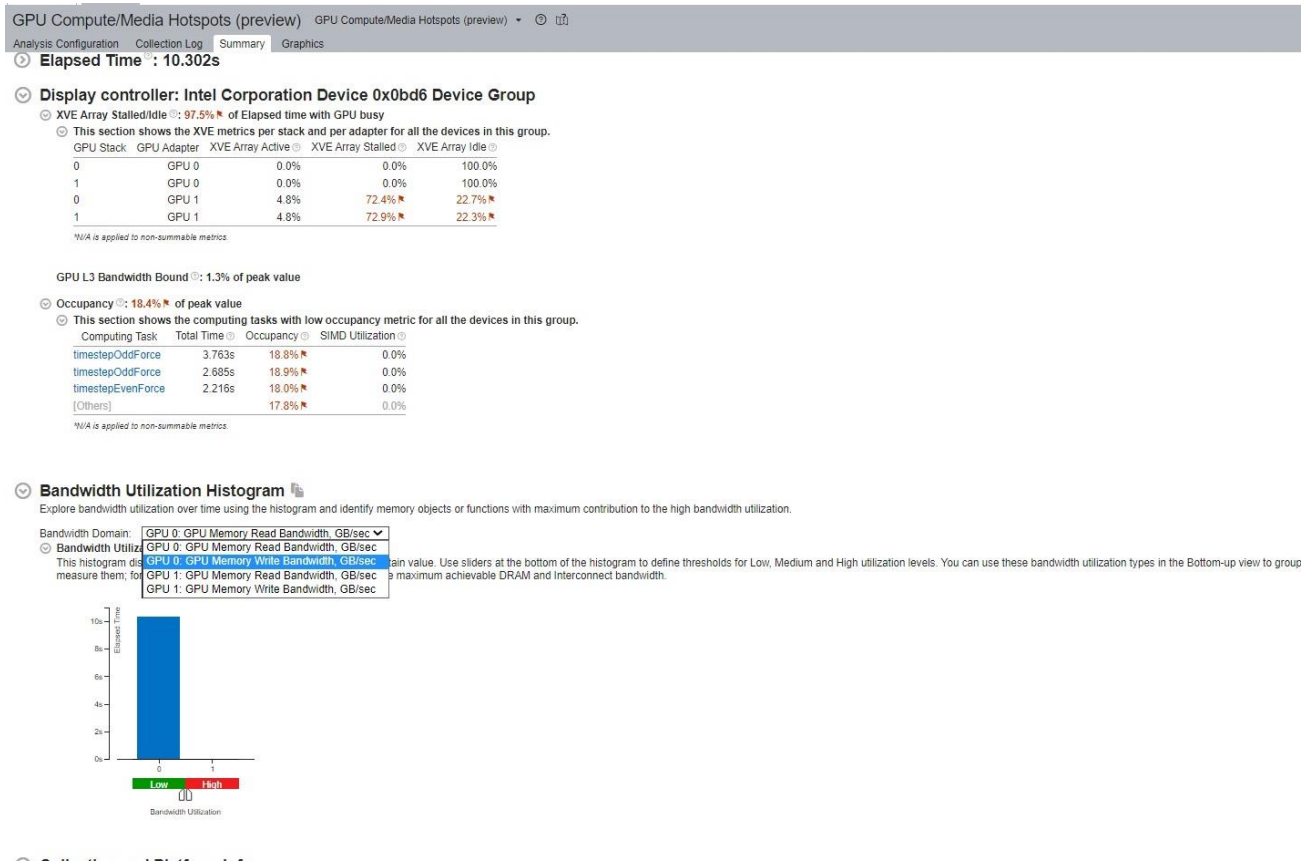

## Sample Command Line:

```
$ 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.



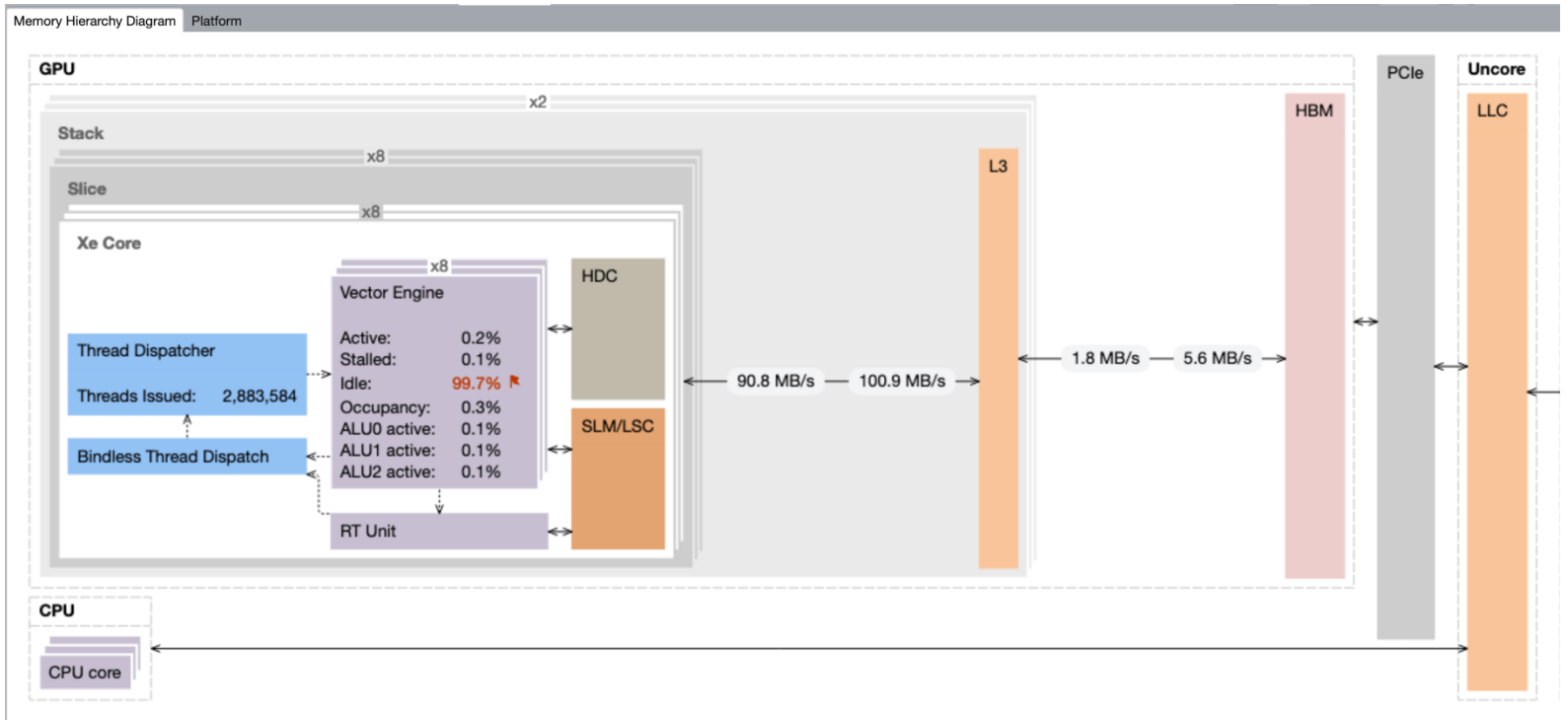

**Sample Command Line:**

```
$ 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

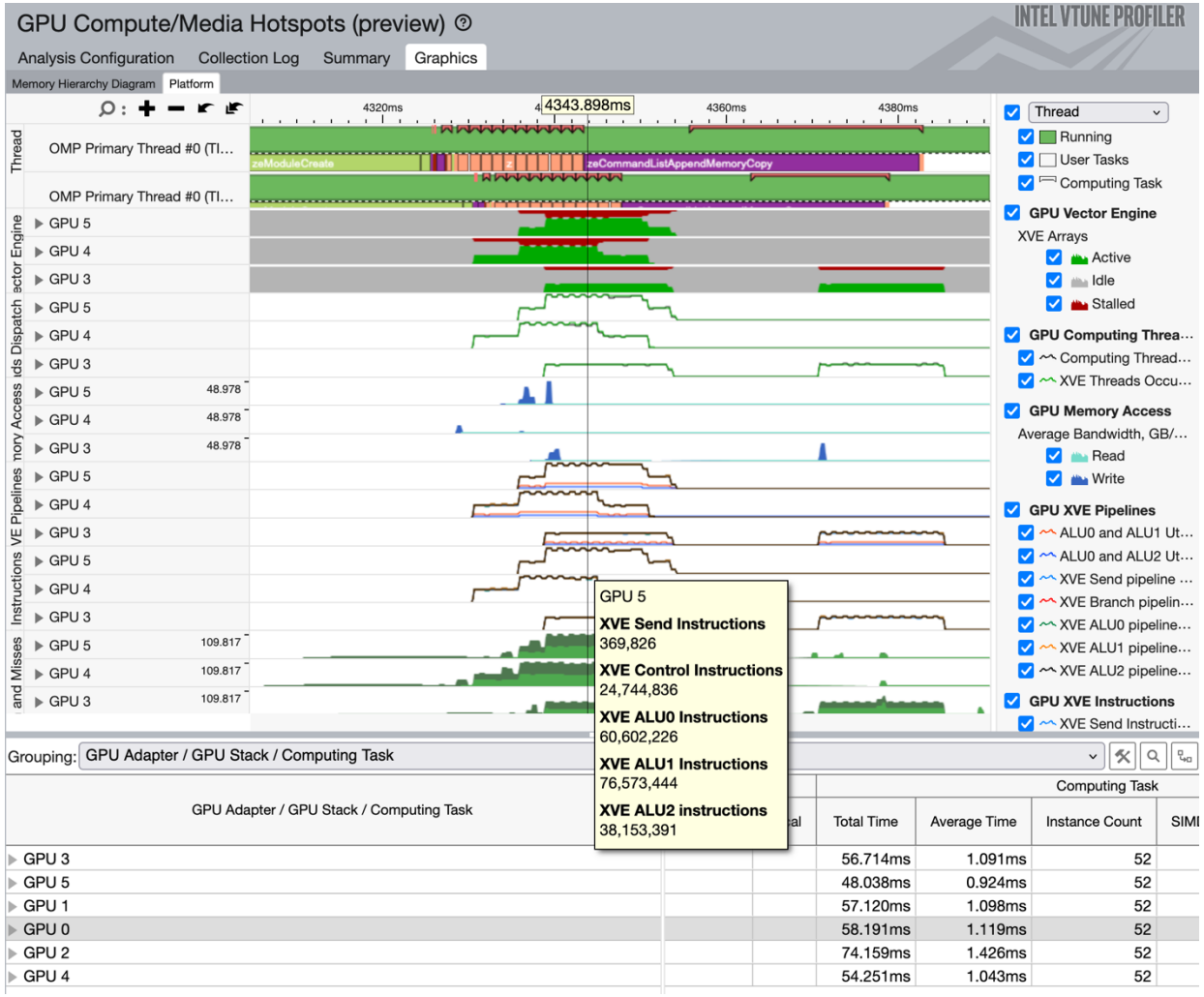
- Understand the GPU Architecture e.g. XVEs, Cores, Stacks
- Analyze data transfer/bandwidth metrics.
  - Total data movement
  - Bandwidth (Read/Write)
  - Percentage compared to Theoretical Peak
- Identify the memory/cache units that cause execution bottlenecks.
- Make decisions on data access patterns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

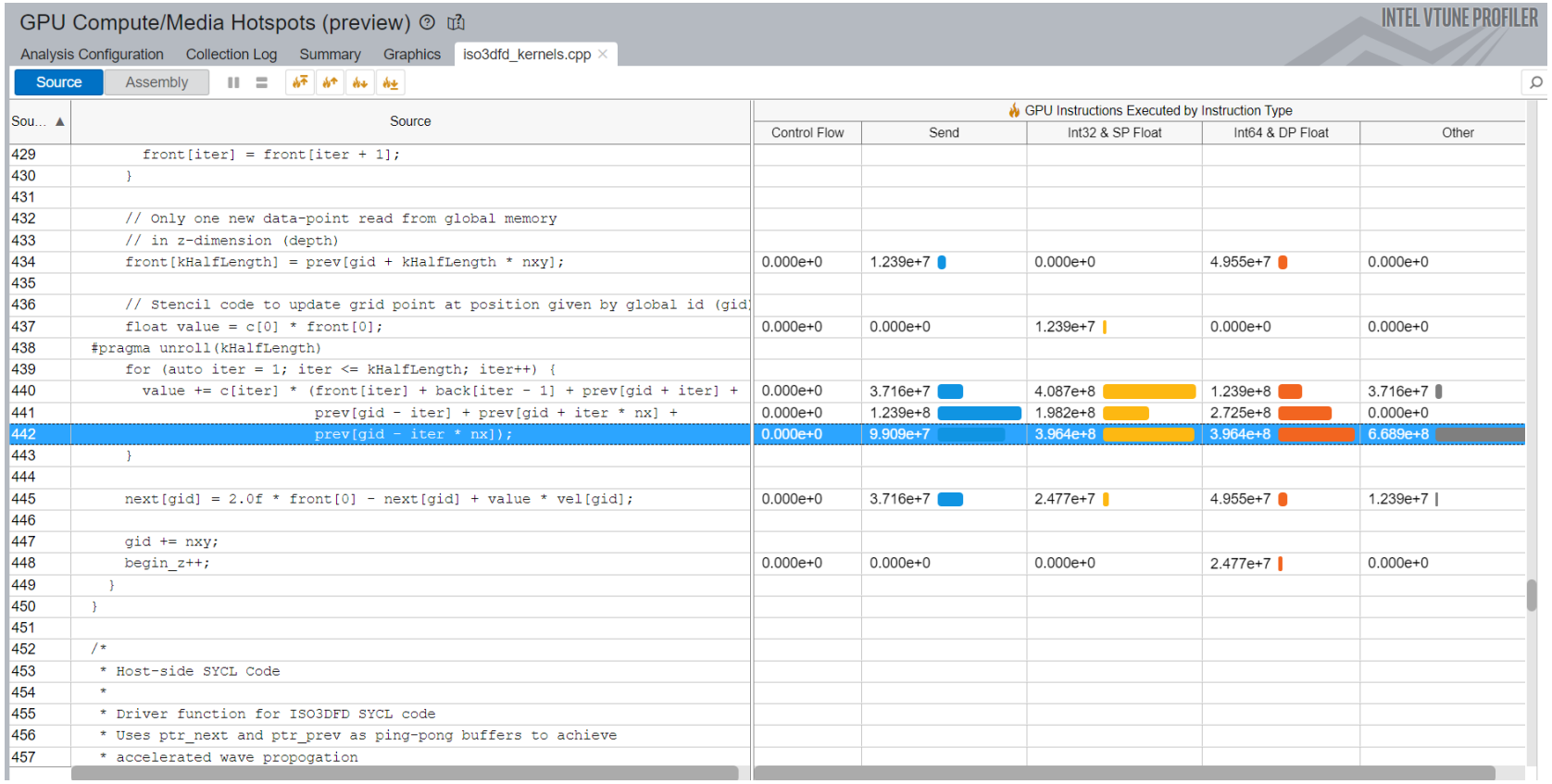


Source level in-kernel profiling requires building apps with "***-fdebug-info-for-profiling -gline-tables-only***")

# 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:




```
$ 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 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:

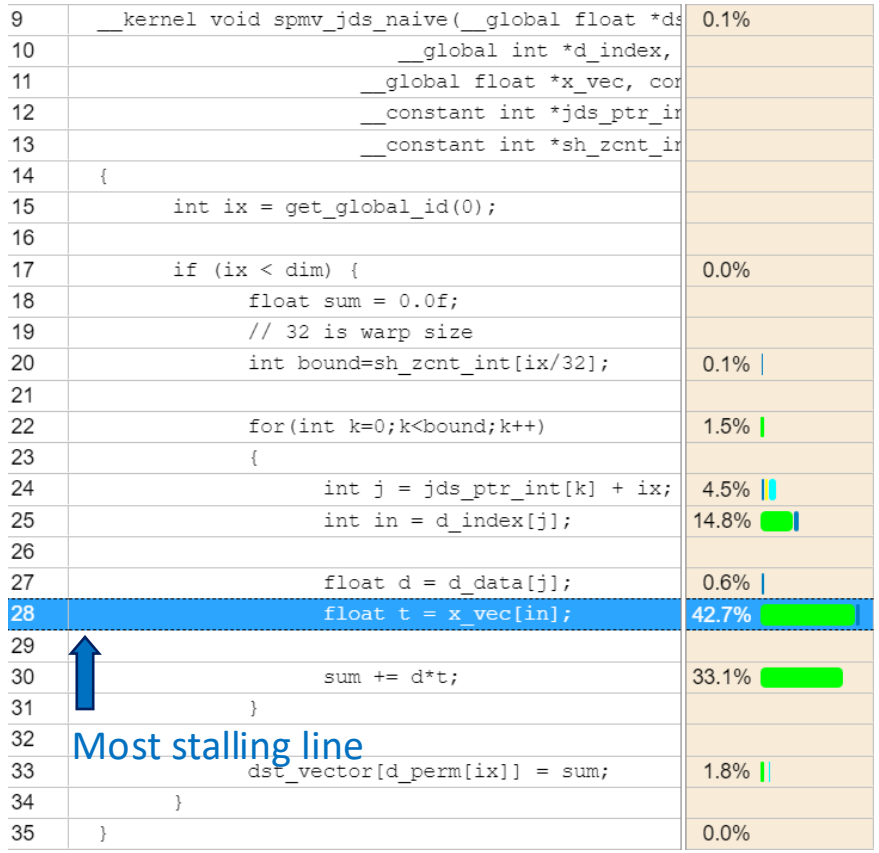
| Sou Lin | Source | Average Latency, Cycles: Total | Average Latency, Cycl... | | Estimated GPU Cycles: Total | Estimated GPU Cycles: Self |
|---|---|---|---|---|---|---|
| | | | Memory Re... | Synchro... | | |
| 19 | #else | | | | | |
| 20 | typedef double REAL; | | | | | |
| 21 | #endif | | | | | |
| 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}
```

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

# Hardware Assisted Stall Sampling

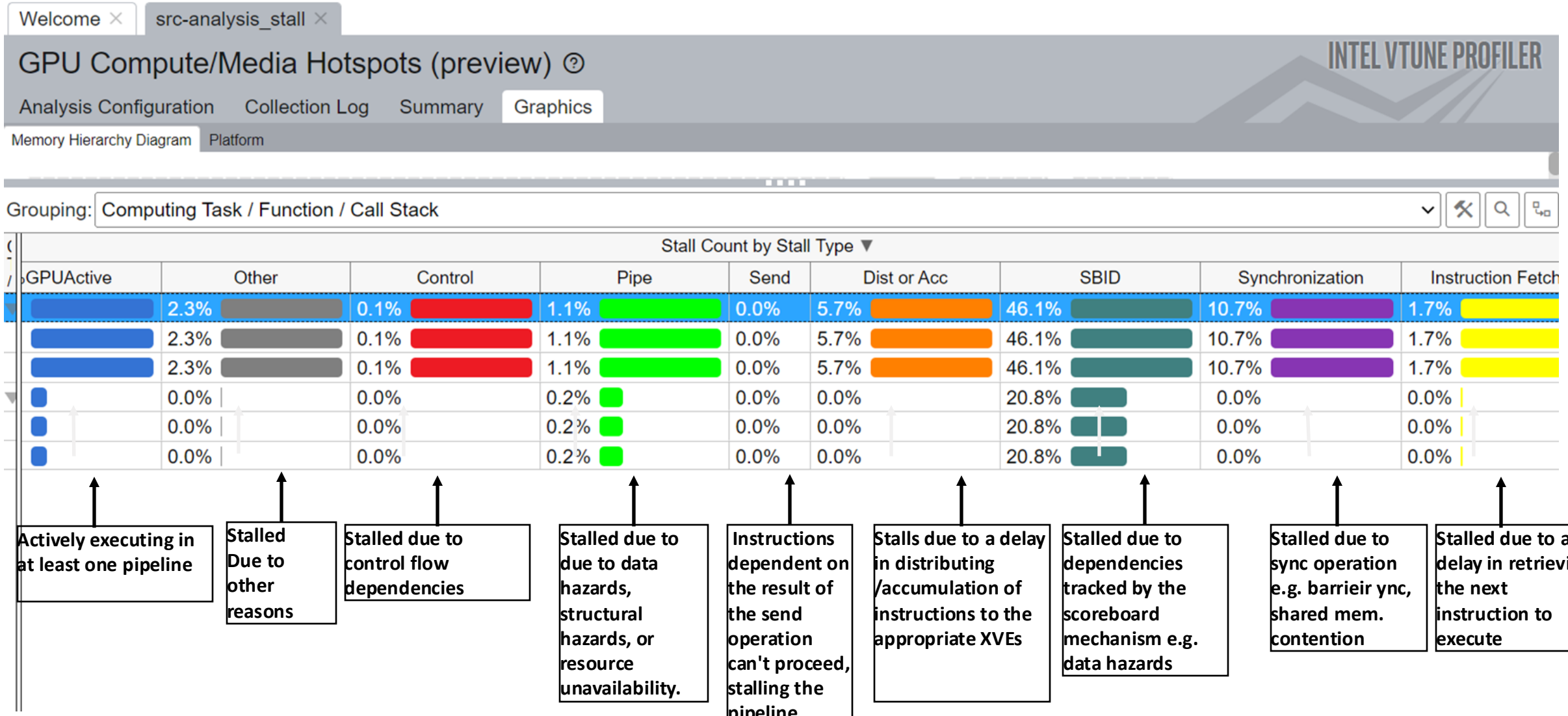
- Provides detailed breakdown of stalls and reasons
- HW-assisted Stall Sampling technology designed for Intel® Data Center GPU Max Series (code-named Ponte Vecchio or PVC)
- Capabilities similar to instruction execution efficiency characterization of NVIDIA® **Nsight™ Compute**
- Sample Command Line:




```
$ 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



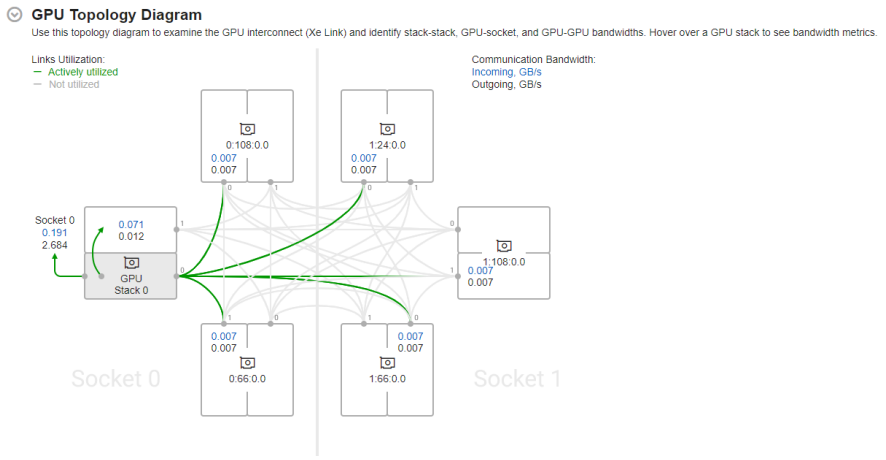
# Get Visibility into Xe Link Cross-card Traffic

## Intel® VTune™ Profiler

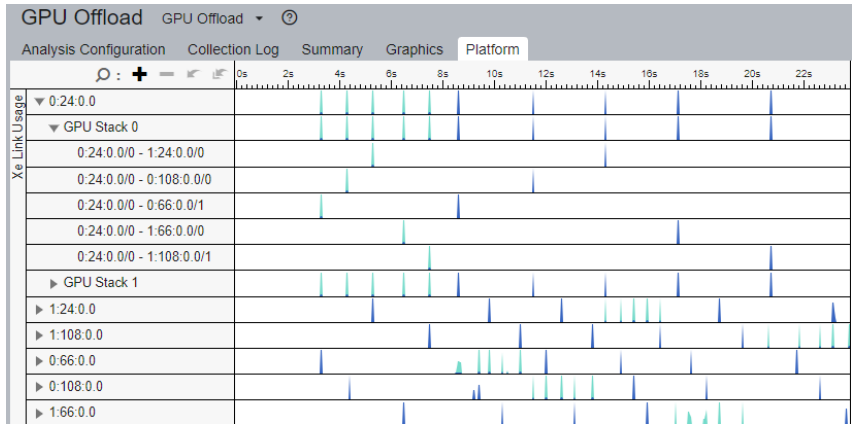
### Identify bottlenecks related to Xe Link

- 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.*

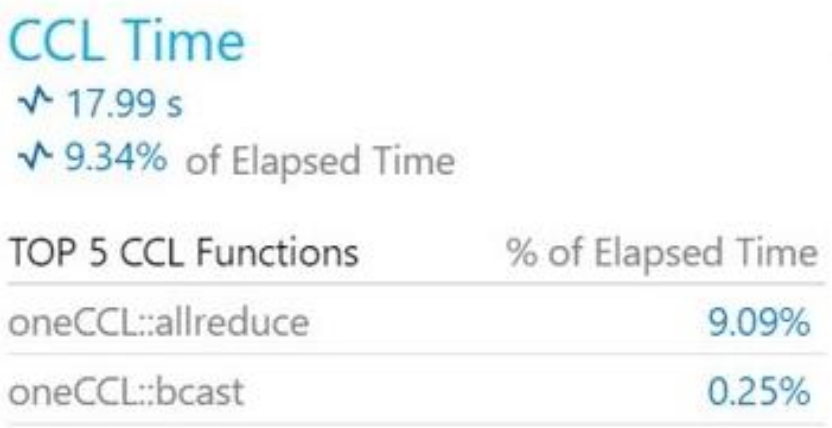



*Timeline view can show bandwidth usage of Xe Link over time.*

# 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





```
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>@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>@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

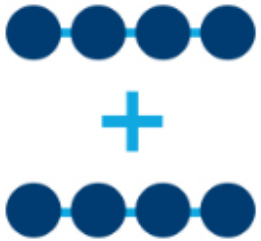
# Rich Set of Capabilities for High Performance Code Design
## Intel® Advisor

### 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.

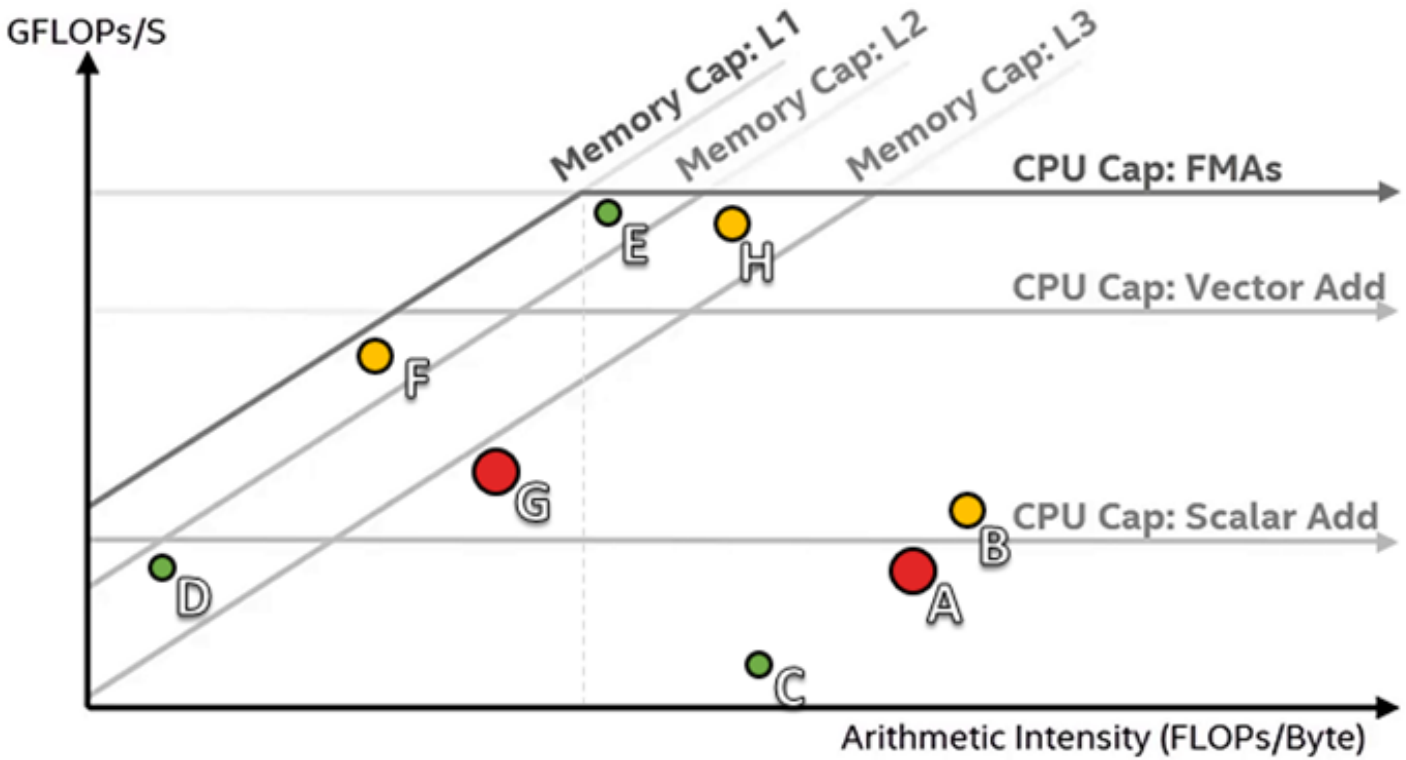
### Build Heterogeneous Algorithms

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
```

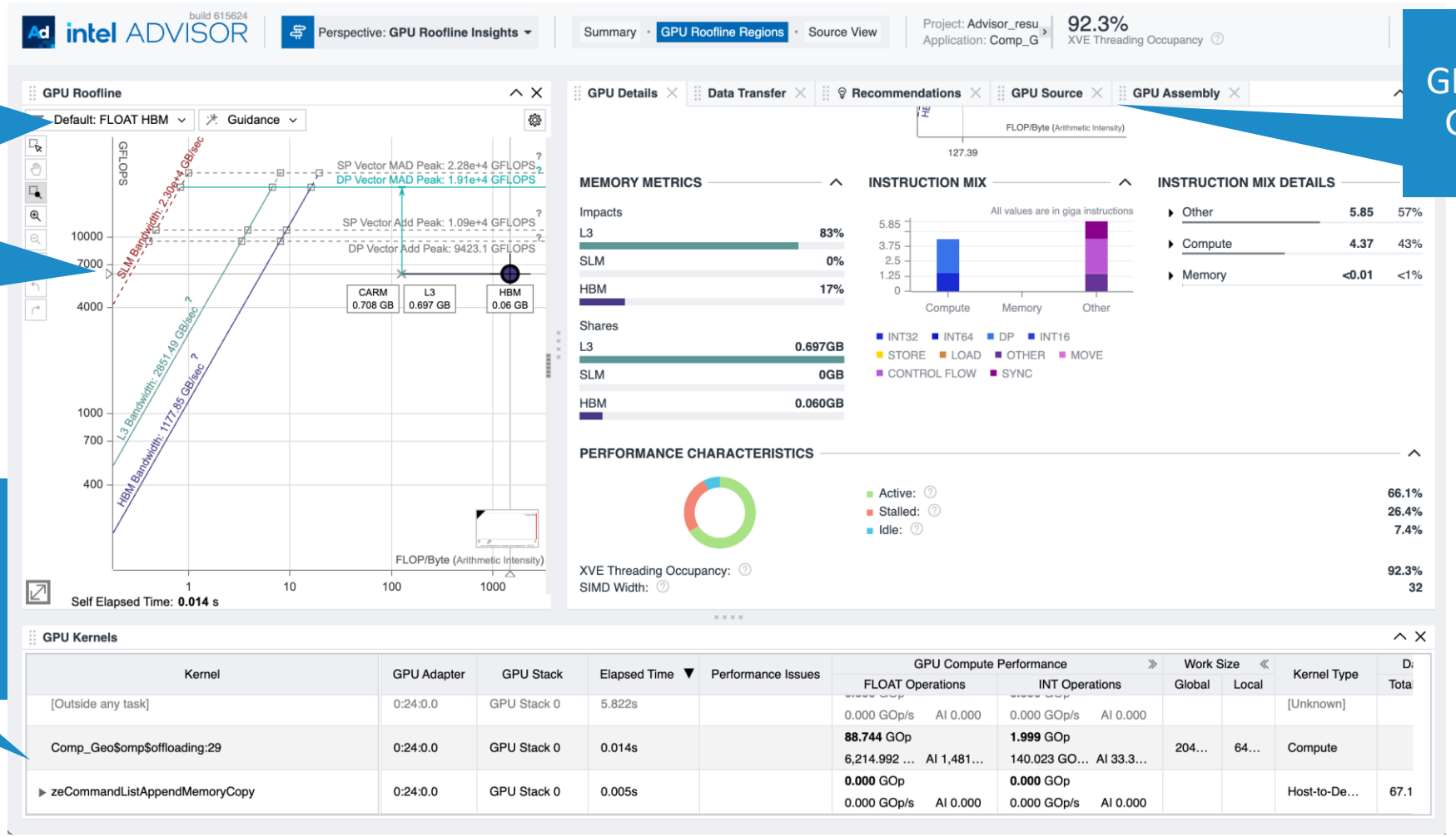
# GPU Roofline Insights

Switch between data types

Customizable GPU Roofline chart

GPU performance of compute tasks

View Details, GPU Source, and GPU Assembly info



# Demo

Create your folder on Aurora

```
$ cd /flare/alcf_training/
$ mkdir $USER
$ cd $USER
```

Copy the example to your space:

```
$ cp -rf /flare/alcf_training/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 on a compute node:

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

# Demo on compute nodes

## 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
```

# Demo on a login node

## aps example :

```
$ cp /flare/alcf_training/Tools_Intel/apsresult.tar.gz .
$ tar -zxvf apsresult.tar.gz
$ aps-report --metrics=? apsresult
$ aps-report --metrics="GPU Stack Utilization Per Device, OpenMP Offload Time, GPU Accumulated Time Per Device, MPI Time" apsresult
```

- Open aps_report_20250207_010006.html (/flare/alcf_training/Tools_Intel) on your web browser

## vtune example :

```
$ cp /flare/alcf_training/Tools_Intel/vtune_result_gh.tar.gz .
$ tar -zxvf vtune_result_gh.tar.gz
```

**Establish vtune web server mode following instruction on page 34**

```
$ vtune-backend --data-directory=/flare/alcf_training/$USER
```

## advisor example :

- Open roofline_all.html (in /flare/alcf_training/Tools_Intel) on your web browser

# Light-weight Intel tools on Aurora

- Intel unitrace
- Intel xpu-smi

# Unified Tracing and Profiling tools (unitrace)

- A performance tools for Intel onaAPI application that traces and profiles host/device activities, interactions and hardware utilizations for Intel GPU applications.
  - https://github.com/intel/pti-gpu/tree/master/tools/unitrace
- Features
  - Level Zero (L0) or Level Zero + OpenCL tracking/profiling
  - Host activities
  - Device and kernel activities
  - Trace and profile layers (e.g., MPI, SYCL, CCL, oneDNN) above L0/OpenCL
  - Categorizing GPU kernels
  - Profile hardware performance metrics

# Unified Tracing and Profiling tools (unitrace)

- Run with unitrace

```
$ unitrace [options] <application> [args]
```

The options are as follows:

--device-timing [-d] Report kernels execution time

--ccl-summary-report [-r] Report CCL execution time summary

--device-timeline [-t] Report device timeline

--chrome-mpi-logging Trace MPI

--chrome-sycl-logging Trace SYCL runtime and plugin

--chrome-ccl-logging Trace oneCCL

--chrome-kernel-logging Trace device and host kernel activities

--separate-tiles Trace each tile separately in case of implicit scaling

--output [-o] <filename> Output profiling result to file

--output-dir-path <path> Output directory path for result files

--metric-query [-q] Query hardware metrics for each kernel instance

--metric-sampling [-k] Sample hardware performance metrics for each kernel instance in time-based mode

--group [-g] <metric-group> Hardware metric group (ComputeBasic by default)

--sampling-interval [-i] <interval> Hardware performance metric sampling interval in us (default is 50 us) in time-based mode

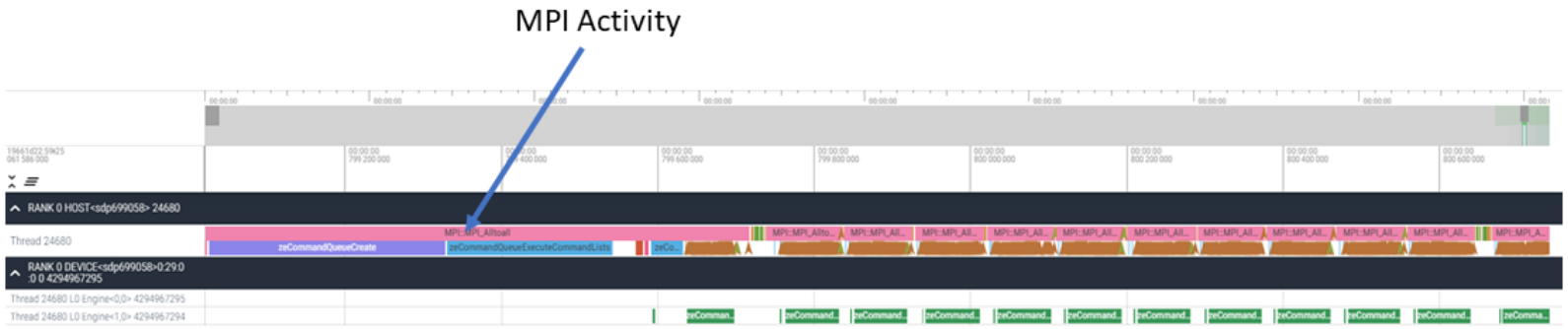
--device-list Print available devices

--metric-list Print available metric groups and metrics

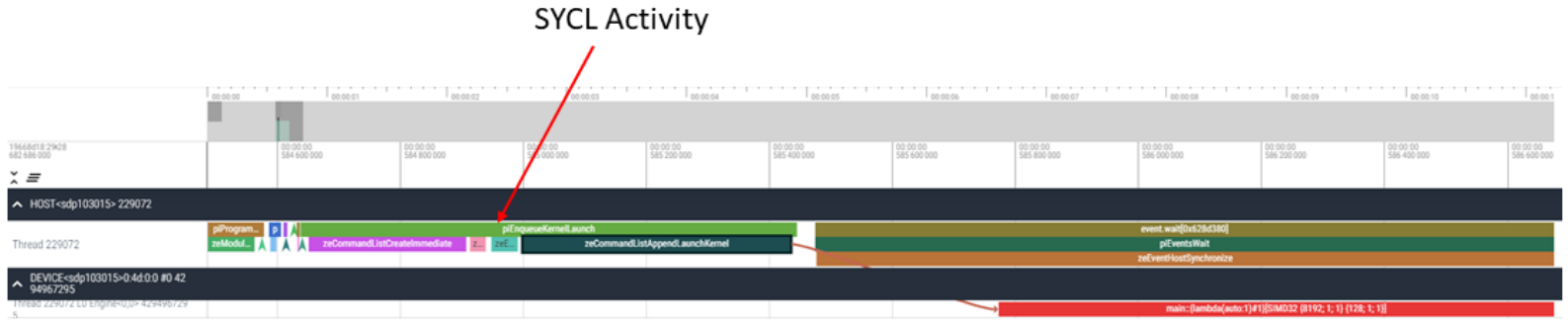
--stall-sampling Sample hardware execution unit stalls. Valid for Intel(R) Data Center GPU Max Series and later GPUs

--ranks-to-sample <ranks> MPI ranks to sample. The argument <ranks> is a list of comma separated MPI ranks

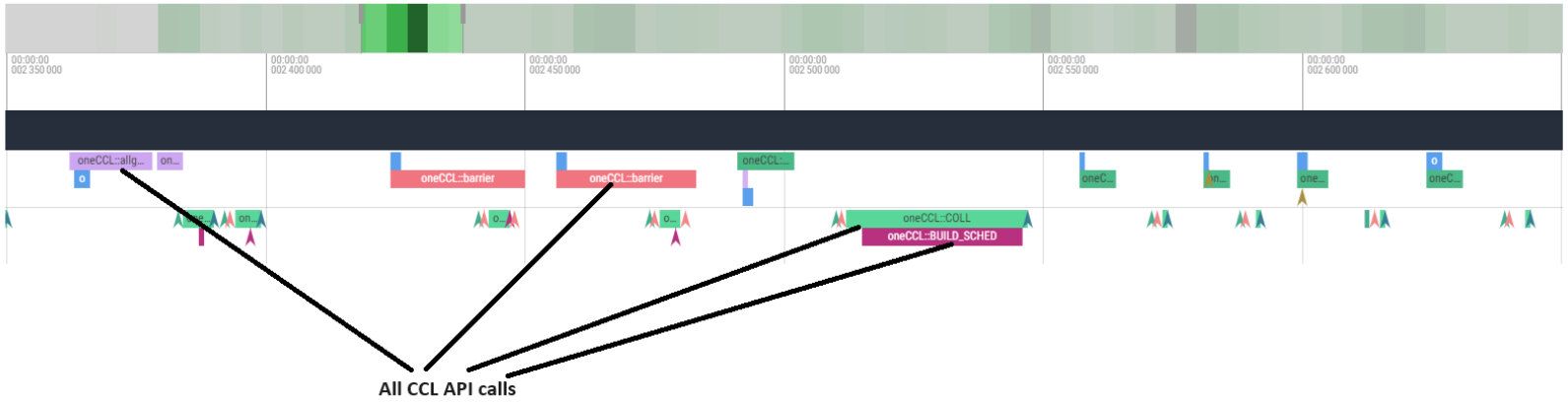
--chrome-mpi-logging




--chrome-sycl-logging




--chrome-ccl-logging



# unitrace on Aurora

```
$ module load pti-gpu
$ unitrace –version
2.1.2 (31dd08753125943b26475cec6a489b7c52c064dd)
$ unitrace --device-list
$ unitrace --metric-list | grep Group
```

# Intel XPU System Management Interface, xpu-smi

- Are you expecting something like nvidia-smi?
- xpu-smi provides similar features on Intel GPUs
  - —https://github.com/intel/xpumanager
- A tool for monitoring and managing Intel data center GPUs
- It is designed to simplify administration, maximize reliability and uptime, and improve utilization.
- Intel(R) XPU System Management Interface (XPU-SMI) is the daemon-less version of XPU Manager and it only provides the local interface.

- xpu-smi on Aurora

```
$ module load xpu-smi
$ xpu-smi --version
CLI:
    Version: 1.2.39.20240906
    Build ID: 11f3c29a
$ xpu-smi discovery
$ xpu-smi dump -d 0 1 2 3 4 5 -m 0 1 2 4 5
```

# Demo on compute nodes

unitrace demo:

```
$ module load pti-gpu

$ which unitrace

$ unitrace --device-list

$ unitrace --metric-list | grep Group

$ mpirun -n 11 gpu_tile_compact.sh ./Comp_GeoSeries_omp_mpicc_DP 2048 1000 : -n 1
gpu_tile_compact.sh unitrace ./Comp_GeoSeries_omp_mpicc_DP 2048 1000

$ mpirun -n 12 gpu_tile_compact.sh unitrace --chrome-mpi-logging
./Comp_GeoSeries_omp_mpicc_DP 2048 1000

$ mpirun -n 12 gpu_tile_compact.sh unitrace --chrome-kernel-logging --chrome-mpi-
logging ./Comp_GeoSeries_omp_mpicc_DP 2048 1000
```

# Demo on compute nodes

xpu-smi demo:

```
$ module load xpu-smi
$ xpu-smi –version
$ xpu-smi -h
$ xpu-smi discovery
$ xpu-smi –h dump
$ xpu-smi dump -d 0 1 2 3 4 5 -m 0 1 2 4 5
```