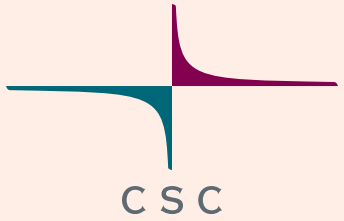



# Profiling on LUMI with AMD tools

A HIP + MPI walkthrough

rocprofv3 / rocprof-sys / rocprof-compute

# What we will work through

The example and its synchronization

How the profiling scripts launch on LUMI

What each tool records, and how to read it

# Which tool fits the question?

rocprofv3
Which kernels and HIP calls ran?

rocprof-sys
What was the host doing between GPU operations?

rocprof-compute
How does a selected kernel use the GPU?

# Traces and counters

A trace records events along a time axis.
It helps explain ordering, waiting, and concurrency.

A counter counts a hardware event.
It helps explain what happened during a kernel.

# The example is a 2D Jacobi stencil

```
u_new[c] = 0.25 * (u[c - 1] + u[c + 1]
                 + u[c - width] + u[c + width]);
```

A square grid, split into columns across MPI ranks

Each rank owns a strip and two ghost columns.
The boundary columns need data from neighbouring ranks.

# The run we are discussing

```
srun -n 8 ./halo_exchange -n 200
```

Default grid: 2048 × 2048
200 timed steps, preceded by 3 warmup steps
8 MPI ranks for the tracing scripts
Host staging is the default

# The baseline iteration

```
pack_halo<<<...>>>(...);
hipMemcpyAsync(..., hipMemcpyDeviceToHost, stream);
hipStreamSynchronize(stream);

// Post two sends and two receives
MPI_Waitall(4, req, MPI_STATUSES_IGNORE);

hipMemcpyAsync(..., hipMemcpyHostToDevice, stream);
unpack_halo<<<...>>>(...);
launch_stencil(..., stream);
```

The stream synchronization completes packing and the outgoing copy.
The stencil starts only after the exchange and unpacking.

# What the code tells us already

In the base variant, stencil work does not overlap MPI exchange.

The size of that cost is still a measurement question.

We cannot decide that communication dominates just by reading the loop.

# Eight GPU devices on a LUMI-G node

```
hipGetDeviceCount(&ndev);
const int dev = local_rank % ndev;
hipSetDevice(dev);
```

4 MI250X modules, with 2 graphics compute dies each

The usual layout here is one MPI rank per GCD.
The compiler target is gfx90a.

# The Slurm allocation

```
#SBATCH --partition=standard-g
#SBATCH --nodes=1
#SBATCH --gpus-per-node=8
#SBATCH --ntasks-per-node=8
#SBATCH --time=00:15:00
#SBATCH --account=<your-project>

source scripts/_env.sh
```

These directives reserve resources.
srun starts the tasks inside the allocation.

# The shared environment

```
set -euo pipefail

module load LUMI/25.03 partition/G
module load rocm rocm/6.3.4-extras

BIN="${BIN:-$PWD/halo_exchange}"
PROF="${PROF:-${SCRATCH:-$PWD}/prof/${SLURM_JOB_ID:-local}}"
export BIN PROF
```

The job scripts use this setup, rather than the older env/lumi.sh.
The selected module stack should also match the build.

# rocprofv3 inside each rank

```
exec rocprofv3 \
  --kernel-trace --hip-trace --stats \
  -d "$PROF" -o "rank_${SLURM_PROCID}" \
  -- "$@"
```

The flags before -- configure the profiler.
Everything after -- is the application command.

# Why the wrapper matters

```
cat > "$PROF/wrap_rocprofv3.sh" <<'WRAP'
#!/bin/bash
exec rocprofv3 \
  --kernel-trace --hip-trace --stats \
  -d "$PROF" -o "rank_${SLURM_PROCID}" \
  -- "$@"
WRAP
chmod +x "$PROF/wrap_rocprofv3.sh"
```

The job shell writes the wrapper once.
Each Slurm task expands its own rank ID when it runs it.

# What srun actually launches

```
srun -n 8 "$PROF/wrap_rocprofv3.sh" "$BIN" -n 200

# Task 0 uses: -o rank_0
# Task 1 uses: -o rank_1
# ...
# Task 7 uses: -o rank_7
```

Eight tasks each start a profiler and one application rank.
The profiler receives the application arguments through "$@".

# Reading the trace files

Kernel events: which dispatch, on which device, and when

HIP events: what the host called and how long it took

Statistics: counts and durations grouped by operation

# A launch call is not the kernel execution

```
stencil_kernel<<<grid, block, 0, stream>>>(...);
// CPU call interval and GPU execution interval differ
```

The CPU submits work and can return before it finishes.

Use the GPU event duration for kernel execution time.
Use API and GPU tracks together to inspect delays.

# Kernel time and wall time

Kernel intervals can overlap.
Different devices can run simultaneously.
The host may work while the GPU runs.

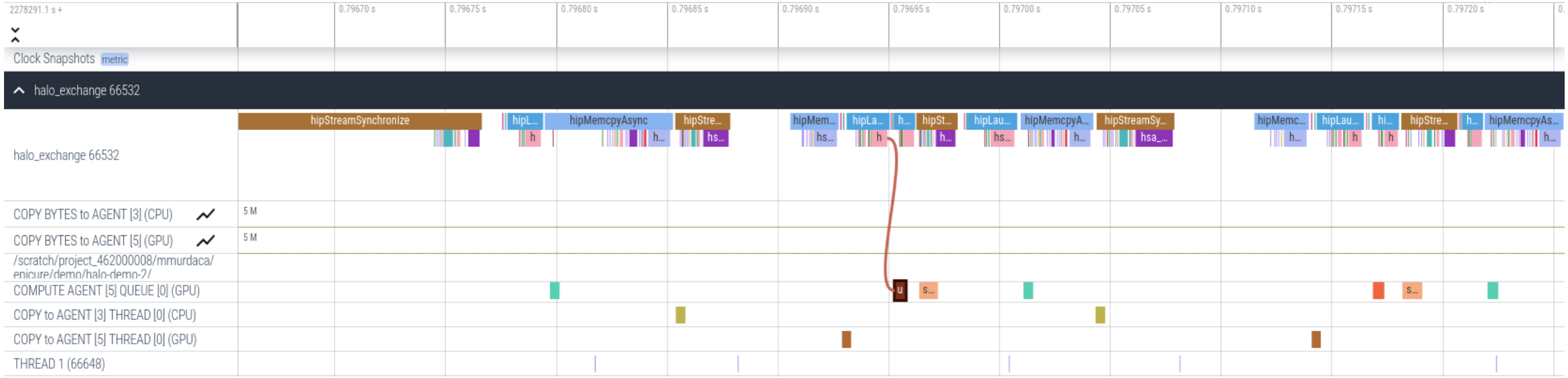
Adding every kernel duration does not give elapsed GPU busy time.

# Collecting a Perfetto trace

```
exec rocprofv3 \
  --sys-trace --output-format pftrace \
  -d "$PROF" -o "rank_${SLURM_PROCID}" \
  --truncate-kernels \
  -- "$@"
```

The rank wrapper and launch stay the same.
Only the profiler options change.

# HIP calls and GPU activity




Host API calls and device activity occupy different intervals.

# ROCTx gives application phases a name

```
roctxRangePush("halo exchange");
// Post the sends and receives
MPI_Waitall(4, req, MPI_STATUSES_IGNORE);
roctxRangePop();
```

Named ranges make the trace easier to relate to the algorithm.
This range measures host scope, not GPU completion.

# Counter collection uses a shorter run

```
# counters.txt
pmc: SQ_WAVES SQ_INSTS_VALU
pmc: TCC_HIT_sum TCC_MISS_sum

srun -n 1 rocprofv3 -i counters.txt \
  -d "$PROF" -o counters -- "$BIN" -n 20
```

The script requests one GPU and one rank.
Each pmc line defines a separate collection group.

# Counter values need context

```
hit ratio = hits / (hits + misses)
```

Compare the same kernel and problem size.
Check whether a value is a count, a rate, or a derived metric.

A hit ratio alone does not tell us how much time a kernel loses.

# rocprof-sys: the host side of the gap

CPU sampling asks where the host spends time.
GPU tracing records device activity alongside it.
MPI tracing can help identify communication intervals.

# The sampling configuration

```
CFG="$PWD/rocprof-sys.cfg"
[ -f "$CFG" ] || rocprof-sys-avail -G "$CFG"
export ROCPROFSYS_CONFIG_FILE="$CFG"
export ROCPROFSYS_OUTPUT_PATH="$PROF"
export ROCPROFSYS_PROFILE=ON
export ROCPROFSYS_TRACE=ON

srun -n 8 rocprof-sys-sample -- "$BIN" -n 200
```

An existing configuration is reused.
Environment settings in the script select the output and modes.

# Sampling and instrumentation

Sampling periodically observes the call stack.
Short functions may be missed.

Instrumentation adds hooks to selected functions.
It gives call detail, with potentially more overhead.

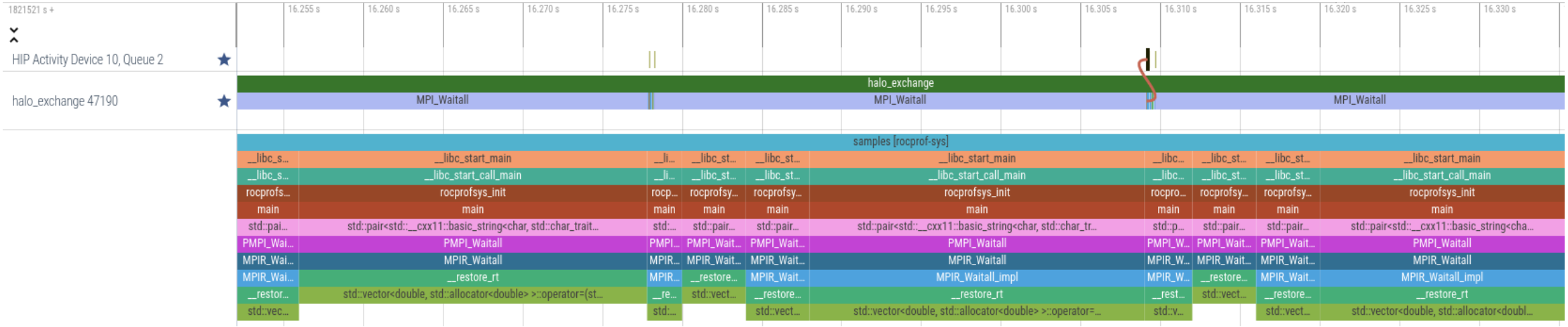
# Binary instrumentation

```
rocprof-sys-instrument \
  -o "$PWD/halo.inst" -- "$BIN"

srun -n 8 rocprof-sys-run \
  -- "$PWD/halo.inst" -n 200
```

The first command creates a rewritten executable.
The second runs that executable under the profiler.

# MPI waits and a GPU queue




Repeated MPI_Waitall intervals, with brief activity on the displayed GPU queue.

# The overlap variant

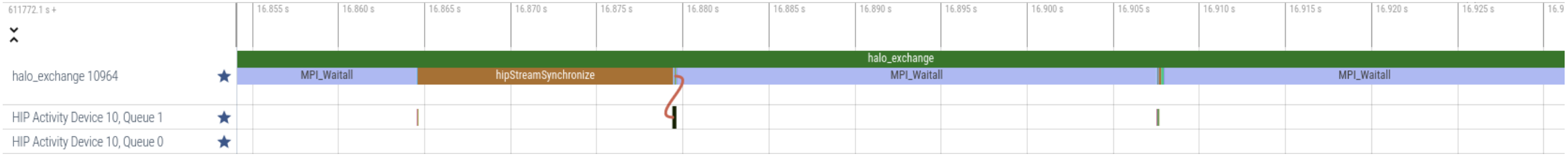
```
// After posting the MPI requests
if (overlap) {
  launch_stencil(u, u_new, 2, nx_local - 1, comp_stream);
}
MPI_Waitall(4, req, MPI_STATUSES_IGNORE);

// Receive, unpack, then update the boundary columns
```

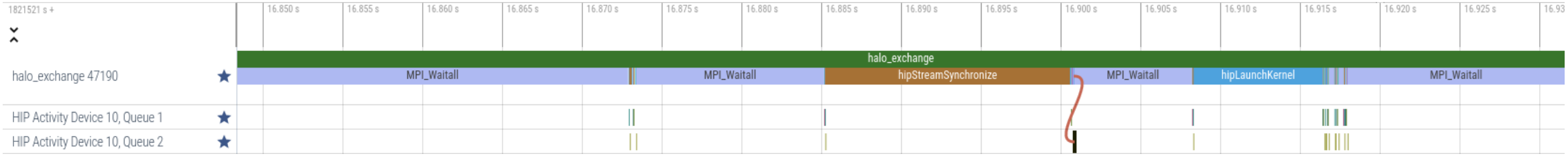
The interior does not need the incoming ghost columns.
It can run before the exchange completes.

# Standard and overlap in Perfetto

## Standard run




## Overlap run




5 ms between labelled ticks. GPU activity alongside host waits.

# rocprof-compute: a selected kernel

Once we choose a kernel, we can ask about:

Memory traffic and cache behaviour
Instruction mix
Occupancy and resource limits
Compute and bandwidth utilisation

# Collecting the two variants

```
cd "$PROF"

srun -n 1 rocprof-compute profile \
  -n base -k stencil_kernel -- "$BIN" -n 20

srun -n 1 rocprof-compute profile \
  -n tiled -k stencil_kernel -- "$BIN" -n 20 \
  --variant tiled
```

-n base names the profiler workload.
The final -n 20 sets application time steps.

# What “tiled” means in this code

```
double up  = u[(j0 - 1) * width + i];
double cur = u[j0 * width + i];
for (int r = 0; r < ROWS_PER_THREAD; ++r) {
  // Load the next row and compute one output
  up  = cur;
  cur = down;
}
```

Each thread computes up to four consecutive rows.
It carries vertical neighbours in registers.
There is no explicit shared-memory tile here.

# Analysis reads the saved workload

```
rocprof-compute analyze \
  -p workloads/base/MI200 --list-stats

rocprof-compute analyze \
  -p workloads/base/MI200 \
  -p workloads/tiled/MI200
```

Collection needs the application and GPU.
Analysis reads the stored results and can be repeated.

# Reading a roofline

Arithmetic intensity = operations / bytes transferred

Attainable rate ≤ min(compute ceiling, bandwidth × intensity)

The memory level and operation type must match the comparison.

# A bandwidth hypothesis

Evidence to look for:
High achieved bandwidth for the relevant memory level
Traffic consistent with the algorithm
A faster kernel after reducing that traffic

Cache hit rate or occupancy alone is insufficient.

# Profiling changes the experiment

Tracing adds collection overhead.
Counters can serialize dispatches and require repeated runs.

Check that a profile still represents the behaviour you care about.
Use ordinary runs to establish the speedup.

# Did the change help the application?

```
srun -n 8 ./halo_exchange -n 200 --variant base
srun -n 8 ./halo_exchange -n 200 --variant tiled
srun -n 8 ./halo_exchange -n 200 --variant overlap
```

Compare ordinary runs with the same input and rank count.
Repeat them and keep the run-to-run variation.
Check the result as well as the time.

# For your own code

Choose a short representative run.

Use a trace to locate the time that matters.
Collect counters when you have a specific kernel question.

Keep the command, environment, output, and ordinary runtime together.

# Questions

Which interval would you investigate first
in your own application?