Hewlett Packard Enterprise

Advanced GPU Programming and Debugging

Harvey Richardson & Tim Dykes, HPE HPC&AI EMEA Research Lab

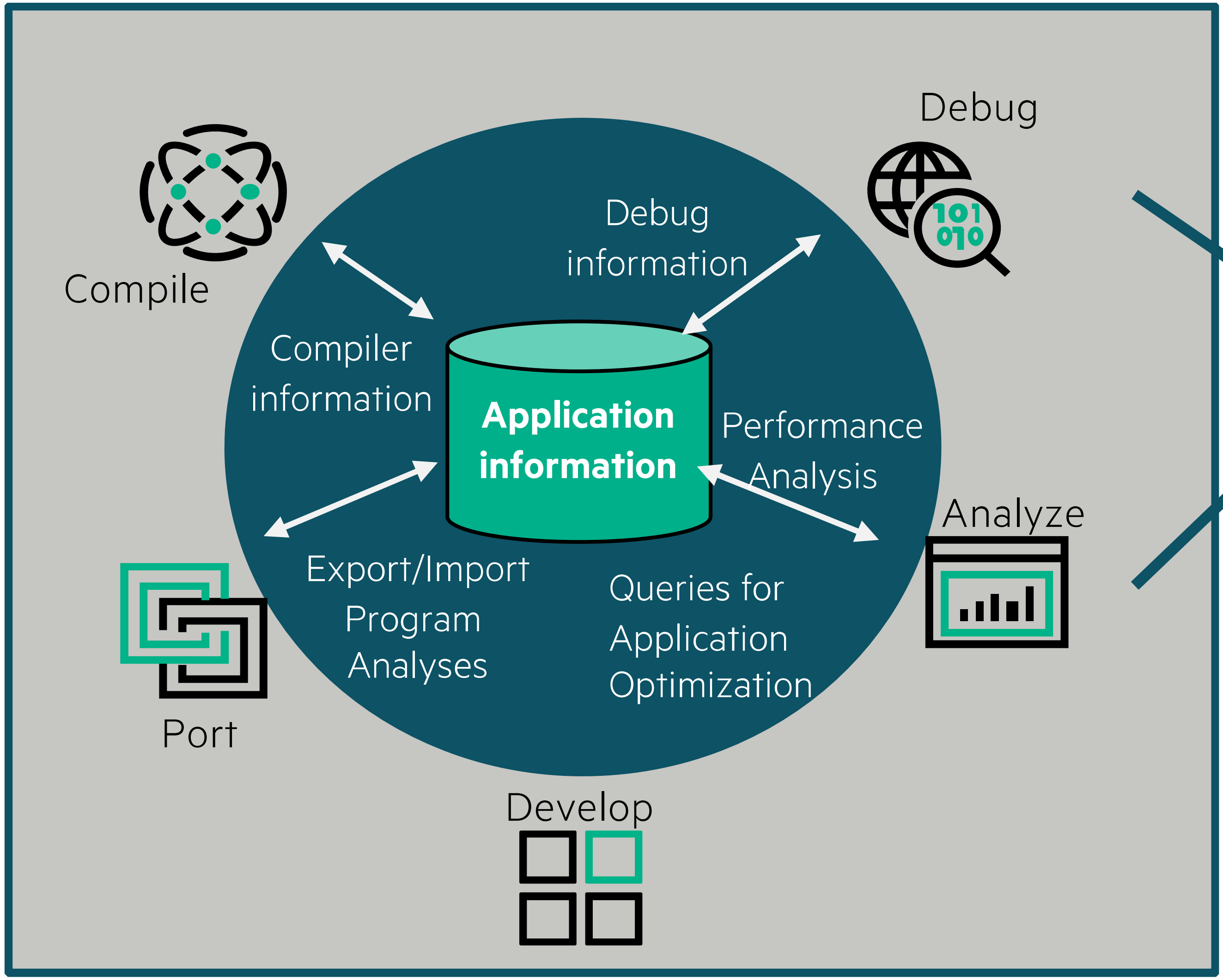
PDC Summer School, Aug. 21-22, 2024

Topics

- Application lifecycle
- Debugging intro and tools available on Cray EX platforms
- Review of signals and causes

Tools
- The ROCm debugger (rocgdb)
- Performance analysis and approaches
- The rocprof profiling tool
- Cray Performance Tools (if time allows)

The Application Porting Life Cycle



The topic of this talk

Debugging 101: the major types of bugs

- Crashing bugs
  - One or more processes in your application terminate
  - Generally (but not always) the easiest kind to solve

- Hangs
  - Deadlocks – everyone is stuck waiting for something that never happens
  - Livelocks – everyone is playing hot potato, calling different functions but not progressing

- Race conditions
  - One or more actors accessing the same data at the same time in a nondeterministic way
  - Shows up as changing results or sometimes crashes
  - Turns up in new contexts on GPUs

- Generally incorrect results
  - Could be many things - a race condition, misuse of external API
  - Possibly your code is just wrong

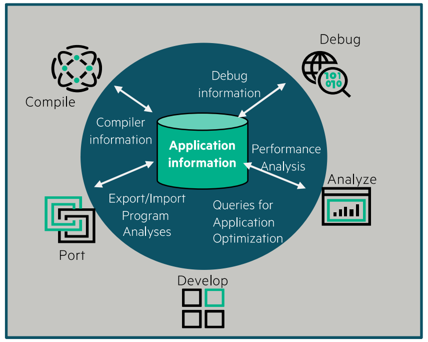
Debugging in production and scale

- Even with the most rigorous testing, bugs may occur during development or production runs.
  - It can be very difficult to recreate a crash without additional information
  - Even worse, for production codes need to be efficient so usually have debugging disabled

- Large parallel and/or heterogenenous applications can be hard to debug…

- The failing application may have been using tens of or hundreds of thousands of processes
  - If a crash occurs one, many, or all of the processes might issue a signal.
  - We don’t want the core files from every crashed process, they’re slow to write and too big!
  - We don’t want a backtrace from every process, they’re difficult to comprehend and analyze.

Tools overview typically available on Cray EX with AMD GPUs

	**Light weight** No modification of the build chain, pre-/postprocessing. Get a first picture of a performance or problems during execution.	**In-depth** Requires modifications of the build system, recompile/relink, , pre-/postprocessing. Provides detailed information at user routine level.
Debugging Get your code up and running correctly.	ATP STAT CRAY_ACC_DEBUG	rocgdb, gdb4hpc valgrind4hpc, Sanitizers4hpc Linaro (Arm) Forge Tools
Profiling Locate performance bottlenecks.	perftools-lite	perftools rocprof Apprentice2 Reveal

Review of common signals

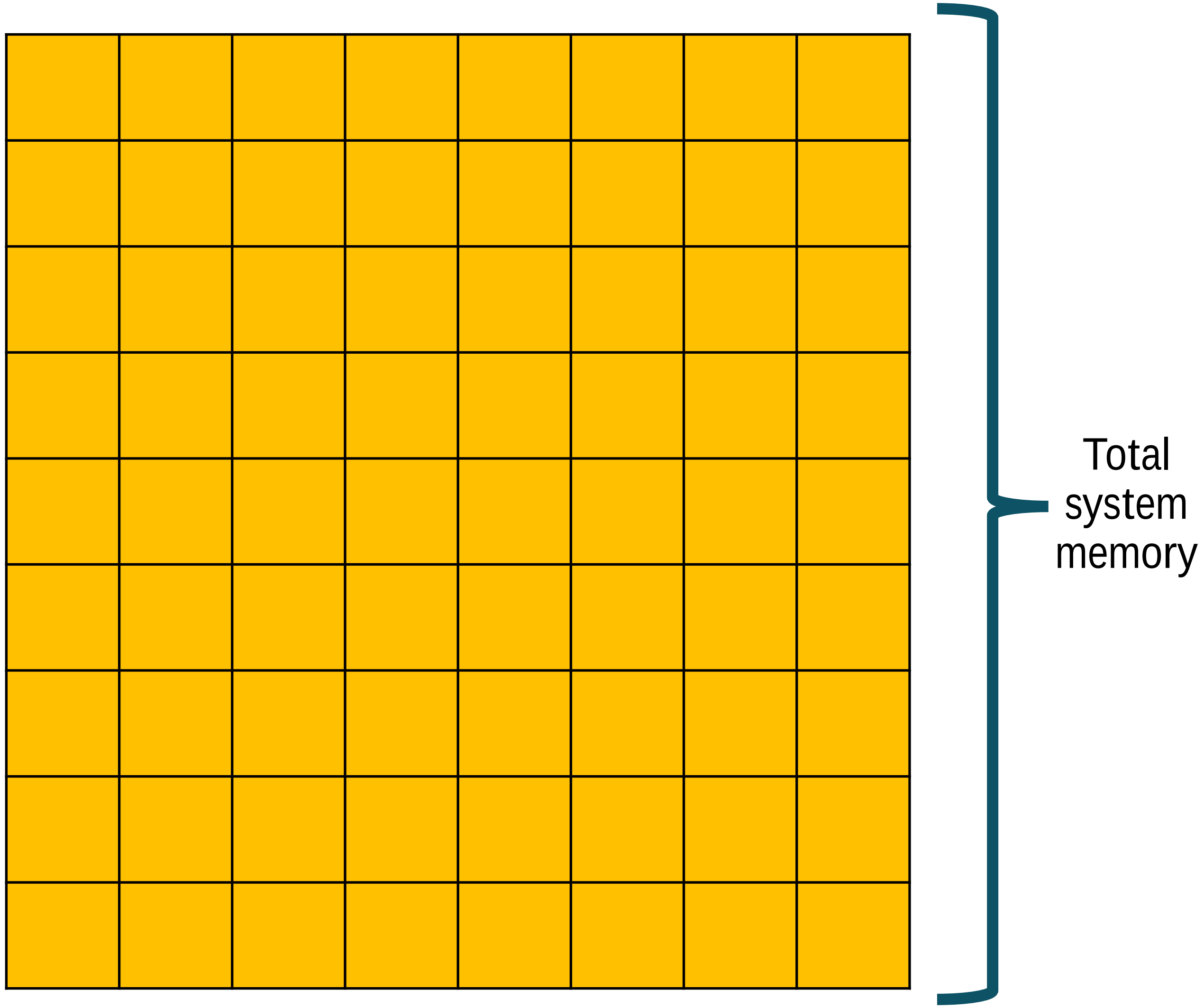
What the error message means

Common Signals from “crashes”

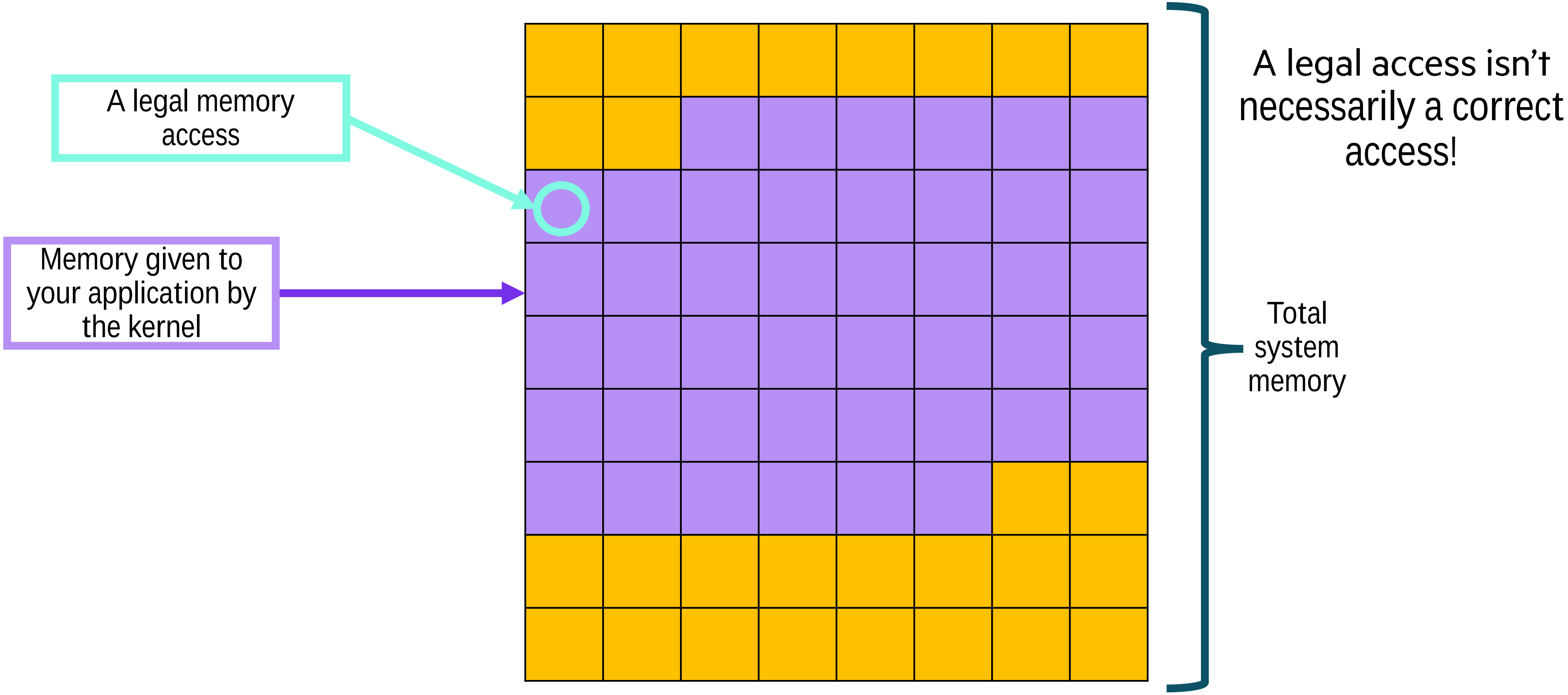
- “Words mean things”
- "man 7 signal" can act as a cheat sheet

Signal Abbreviation (No. on x86/ARM)	Signal Name	What it means
SIGSEGV (11)	Segmentation Fault, AKA SegFault	You attempted to access memory that technically exists on the machine but is outside the virtual address space the kernel gave you
SIGBUS (7)	Bus error	You attempted to access memory that cannot possibly be accessed (most likely culprit nowadays: requirement for aligned memory not met)
SIGABRT (6)	Abort	Your application, or a library it uses, realized something was wrong and crashed intentionally
SIGFPE (8)	Floating Point Exception	You did some dangerous math and asked to be notified about it

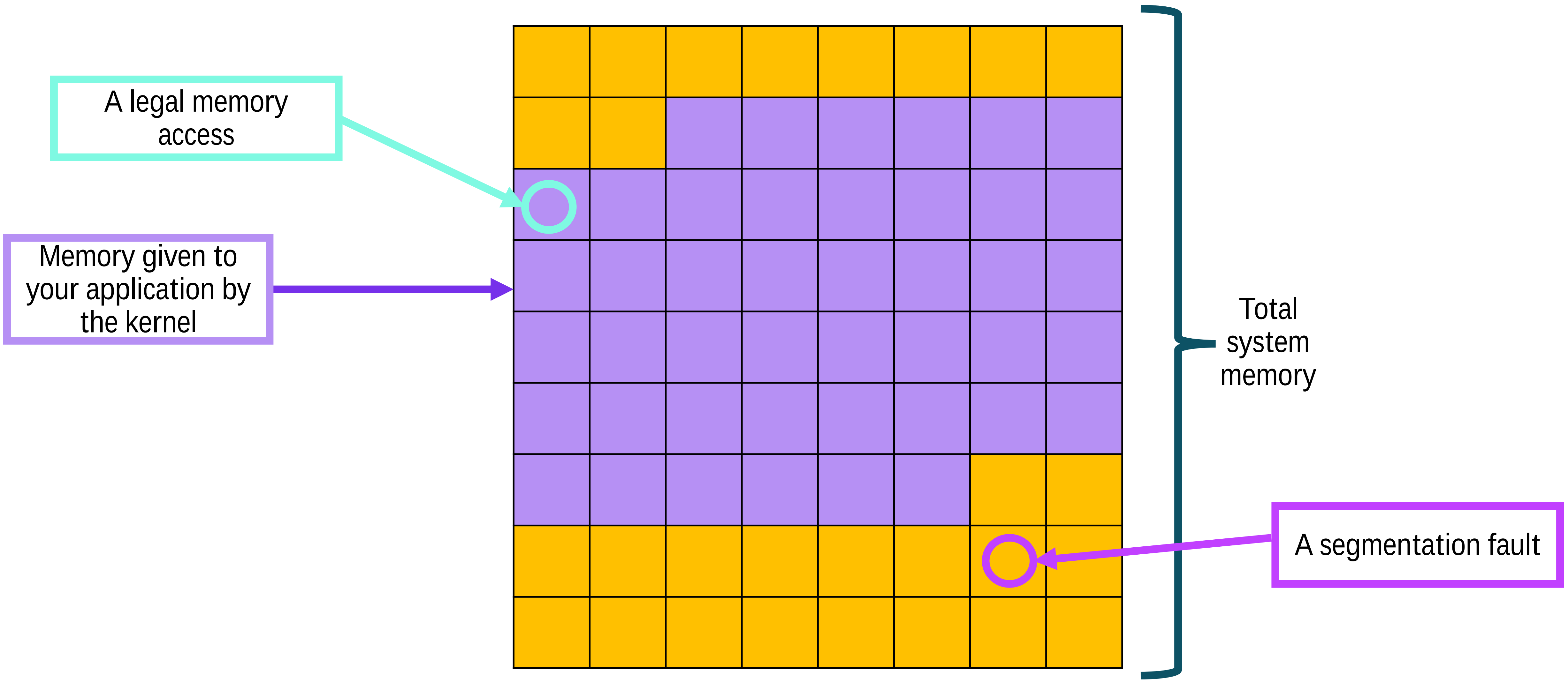
SegFault visualized



SegFault visualized



SegFault visualized



SegFault/SIGBUS techniques

- Bounds checking
  - CCE Fortran `-hbounds`
  - Intel Fortran `-check bounds` or `--CB`
  - GNU Fortran `-fcheck=bounds` or `--fbounds-check`
  - C++ containers
- Address sanitizing (Use CPE sanitizers4hpc for parallel aggregation)
  - CCE C/C++/Fortran –fsanitize=address
  - LLVM –fsanitize=address
  - GNU C/C++ -fsanitize=address
- Valgrind (valgrind4hpc for parallel aggregation)
- For a comparison of these techniques see: https://developers.redhat.com/blog/2021/05/05/memory-error-checking-in-c-and-c-comparing-sanitizers-and-valgrind

- Core files/gdb/ATP

Trapping math and Clang

- Some compiler optimizations may generate intermediate arithmetic errors
- GNU compilers
  - DO NOT allow these optimizations at –O levels
  - ENABLE these with `-funsafe-math-optimizations` or `-fno-trapping-math`, but don’t turn on trapping!
- Clang compilers
  - DO allow some of the optimizations at –O levels
  - DISABLE these with `-ffp-exception-behavior=maytrap`

So what about the GPU?

- Exceptions can occur on the GPU
- These will be reported on the CPU
- GPUs have a limited support for exceptions, which prevents exception handling on the GPU.
- The existing solution forwards GPU memory faults to the CPU
- The faulting instruction is stalled in the GPU pipeline.

- There is some software to check correctness e.g., NVIDIA's Compute Sanitizer, LLVM Asan on GPU for AMD (beta)

```
void check_error(void)
{
  hipError_t err = hipGetLastError();
  if (err != hipSuccess)
  {
    std::cerr << "Error: " <<
hipGetErrorString(err) << std::endl;
    exit(err);
  }
}
```

ROCm debugger (rocgdb)

AMD‘s ROCm source-level debugger

Description

- Based on the GNU debugger (GDB)
- Standard GDB commands work on both CPU and GPU
- In active development
  - https://github.com/ROCm/ROCgdb
- The focus for rocgdb is on source line debugging
- Supports core file inspection

- Use this when your focus is on behaviour of an individual GPU

Usage of rocgdb

- Load the rocm module to access rocgdb and the relevant target architecture module

```
> module load craype-accel-amd-gfx90a
> module load rocm
```

- Compile the application using –ggdb

```
> CC -O0 -g -ggdb -x hip -std=c++11 main.cpp –o main.x
```

- Request an interactive terminal on a GPU node and run rocgdb with application

```
> srun --gpus=1 --time=20 --partition=gpu \
       --account=edu24.summer --pty /bin/bash
> rocgdb main.x
```


rocgdb example: Babelstream HIP

- https://github.com/UoB-HPC/BabelStream/blob/main/src/hip/HIPStream.cpp
- We cause an error by removing all the device allocations

```
>> CC -O0 -g -ggdb -x hip -I. -Ihip -std=c++11 -DHIP main.cpp hip/HIPStream.cpp -o hip.x
>> ./hip.x
BabelStream
Version: 4.0
Implementation: HIP
Running kernels 100 times
Precision: double
Array size: 268.4 MB (=0.3 GB)
Total size: 805.3 MB (=0.8 GB)
Using HIP device
Driver: 50221153
Memory: DEFAULT
Memory access fault by GPU node-4 (Agent handle: 0x995b50) on address 0x85000. Reason: Unknown.
Aborted (core dumped)
```

```
#else
  //hipMalloc(&d_a, array_bytes);
  check_error();
  //hipMalloc(&d_b, array_bytes);
  check_error();
  //hipMalloc(&d_c, array_bytes);
  check_error();
#endif
}
//
//
template <class T>
HIPStream<T>::~HIPStream()
{
  hipHostFree(sums);
  check_error();
  //
  hipFree(d_a);
  check_error();
  hipFree(d_b);
  check_error();
  hipFree(d_c);
  check_error();
}
//
//
template <typename T>
__global__ void init_kernel(T * a, T * b, T * c, T initA, T initB,
T initC)
{
  const size_t i = blockDim.x * blockIdx.x + threadIdx.x;
  a[i] = initA;
  b[i] = initB;
  c[i] = initC;
}
//
template <class T>
void HIPStream<T>::init_arrays(T initA, T initB, T initC)
{
  init_kernel<T><<<dim3(array_size/TBSIZE), dim3(TBSIZE), 0,
0>>>(d_a, d_b, d_c, initA, initB, initC);
  check_error();
  hipDeviceSynchronize();
  check_error();
}
```

Use of rocgdb

```
>> rocgdb babel-hip.x
GNU gdb (rocm-rel-5.2-109) 11.2
Copyright (C) 2022 Free Software Foundation, Inc.
…
Reading symbols from babel-hip.x...
(gdb) run
Starting program: /cfs/klemming/home/t/tdykes/…/babelstream_hip/babel-hip.x
…
Thread 6 "babel-hip.x" received signal SIGSEGV, Segmentation fault.
[Switching to thread 6, lane 0 (AMDGPU Lane 1:2:1:1/0 (0,0,0)[0,0,0])]
0x00007fffdfe594d0 in init_kernel<double> (a=0x0, b=0x0, c=0x0, initA=0.10000000000000001,
initB=0.20000000000000001, initC=0)
    at hip/HIPStream.cpp:125
125 b[i] = initB;
(gdb)
```

- Seg fault is reported inside init_kernel with the line number shown

GPU threads

```
(gdb) info threads
Id   Target Id                                          Frame
  1    Thread 0x7fffdfe7d600 (LWP 141550) "babel-hip.x"  0x00007fffe07169cb in ?? () from /opt/rocm-5.7.0/lib/libhsa-
runtime64.so.1
  2    Thread 0x7fffdf3ff700 (LWP 141557) "babel-hip.x"  0x00007fffe8e564a7 in ioctl () from /lib64/libc.so.6
  5    Thread 0x7fffdfb3f700 (LWP 141560) "babel-hip.x"  0x00007fffe8e564a7 in ioctl () from /lib64/libc.so.6
* 6    AMDGPU Wave 1:2:1:1 (0,0,0)/0 "babel-hip.x"       0x00007fffdfe594d0 in init_kernel<double> (a=0x0, b=0x0, c=0x0,
    initA=0.10000000000000001, initB=0.20000000000000001, initC=0) at hip/HIPStream.cpp:125
  7    AMDGPU Wave 1:2:1:2 (0,0,0)/1 "babel-hip.x"       0x00007fffdfe594d0 in init_kernel<double> (a=0x0, b=0x0, c=0x0,
    initA=0.10000000000000001, initB=0.20000000000000001, initC=0) at hip/HIPStream.cpp:125
  8    AMDGPU Wave 1:2:1:3 (0,0,0)/2 "babel-hip.x"       0x00007fffdfe594d0 in init_kernel<double> (a=0x0, b=0x0, c=0x0,
    initA=0.10000000000000001, initB=0.20000000000000001, initC=0) at hip/HIPStream.cpp:125
  9    AMDGPU Wave 1:2:1:4 (0,0,0)/3 "babel-hip.x"       0x00007fffdfe594d0 in init_kernel<double> (a=0x0, b=0x0, c=0x0,
    initA=0.10000000000000001, initB=0.20000000000000001, initC=0) at hip/HIPStream.cpp:125
  10   AMDGPU Wave 1:2:1:5 (0,0,0)/4 "babel-hip.x"       0x00007fffdfe594d0 in init_kernel<double> (a=0x0, b=0x0, c=0x0,
    initA=0.10000000000000001, initB=0.20000000000000001, initC=0) at hip/HIPStream.cpp:125
  11   AMDGPU Wave 1:2:1:6 (0,0,0)/5 "babel-hip.x"       0x00007fffdfe594d0 in init_kernel<double> (a=0x0, b=0x0, c=0x0,
    initA=0.10000000000000001, initB=0.20000000000000001, initC=0) at hip/HIPStream.cpp:125
  12   AMDGPU Wave 1:2:1:7 (0,0,0)/6 "babel-hip.x"       0x00007fffdfe594d0 in init_kernel<double> (a=0x0, b=0x0, c=0x0,
    initA=0.10000000000000001, initB=0.20000000000000001, initC=0) at hip/HIPStream.cpp:125
…
```

- Shows us the stack trace and tells us that a GPU wave seg faulted

Explain wave details

```
Id   Target Id
23   AMDGPU Wave 1:2:1:19 (1,0,0)/10 "hip.x"
24   AMDGPU Wave 1:2:1:20 (1,0,0)/11 "hip.x"
25   AMDGPU Wave 1:2:1:25 (2,0,0)/0 "hip.x"
26   AMDGPU Wave 1:2:1:26 (2,0,0)/1 "hip.x"
```

- GPU ID
- HSA queue ID
- Dispatch number
- Wave ID
- Workgroup (x, y, z)
- Wave ID (within a group)

CPU threads

```
(gdb) thread 1
[Switching to thread 1 (Thread 0x7fffdfe7d600 (LWP 141550))]
#0  0x00007fffe07169cb in ?? () from /opt/rocm-5.7.0/lib/libhsa-runtime64.so.1
(gdb) where
#0  0x00007fffe07169cb in ?? () from /opt/rocm-5.7.0/lib/libhsa-runtime64.so.1
#1  0x00007fffe071684a in ?? () from /opt/rocm-5.7.0/lib/libhsa-runtime64.so.1
#2  0x00007fffe0709fa9 in ?? () from /opt/rocm-5.7.0/lib/libhsa-runtime64.so.1
#3  0x00007fffec5ba793 in ?? () from /opt/rocm-5.7.0/lib/libamdhip64.so.5
#4  0x00007fffec5b1318 in ?? () from /opt/rocm-5.7.0/lib/libamdhip64.so.5
…
#11 0x00007fffec58bb28 in ?? () from /opt/rocm-5.7.0/lib/libamdhip64.so.5
#12 0x00007fffec4eb92b in ?? () from /opt/rocm-5.7.0/lib/libamdhip64.so.5
#13 0x00007fffec3886fa in hipDeviceSynchronize () from /opt/rocm-5.7.0/lib/libamdhip64.so.5
#14 0x000000000023d2a3 in HIPStream<double>::init_arrays (this=0x939e50, initA=0.10000000000000001, initB=0.20000000000000001,
    initC=0) at hip/HIPStream.cpp:134
#15 0x000000000023206d in run<double> () at main.cpp:309
#16 0x000000000022e6a1 in main (argc=1, argv=0x7fffffff7488) at main.cpp:99
```

- Shows us that CPU thread is in HSA runtime and the route to how it got there

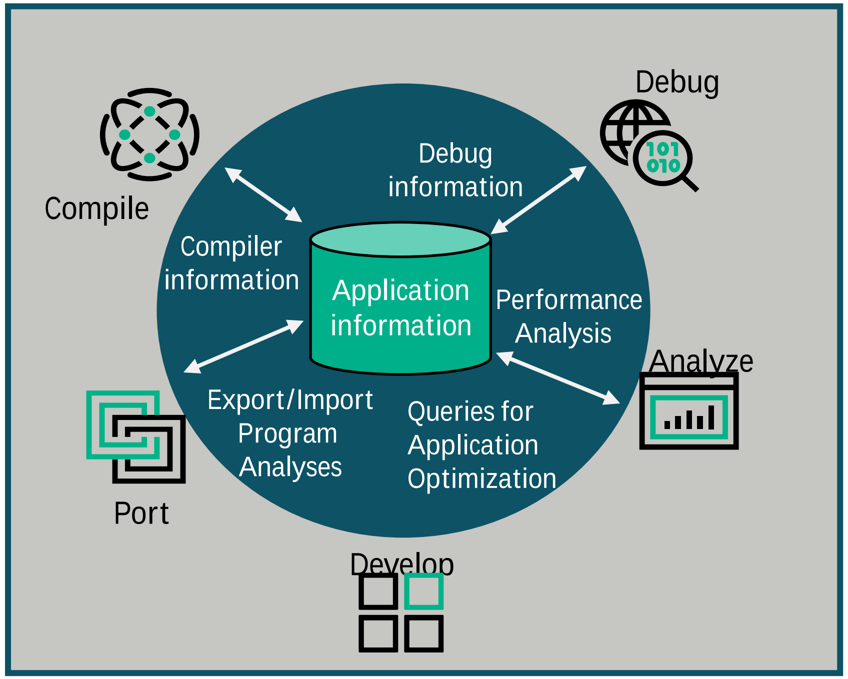
Inspecting GPU data

```
Thread 6 "babel-hip.x" hit Breakpoint 1.2, with lanes [0-63], init_kernel<double> (a=0x7ffe9fe00000, b=0x7ffe8fc00000,
    c=0x7ffe7fa00000, initA=0.10000000000000001, initB=0.20000000000000001, initC=0) at hip/HIPStream.cpp:126
126             c[i] = initC;
(gdb) list
121           __global__ void init_kernel(T * a, T * b, T * c, T initA, T initB, T initC)
122           {
123             const size_t i = blockDim.x * blockIdx.x + threadIdx.x;
124             a[i] = initA;
125             b[i] = initB;
126             c[i] = initC;
127           }
128
129           template <class T>
130           void HIPStream<T>::init_arrays(T initA, T initB, T initC)
(gdb) print initA
$1 = 0.10000000000000001
(gdb) print a[1]
$2 = 0.10000000000000001
(gdb) print initB
$3 = 0.20000000000000001
(gdb) print b[100]
$4 = 0.20000000000000001
```

- We fix HIPStream.cpp and inspect data within the GPU kernel
- Setting `break HIPStream.cpp:126` and running the applications

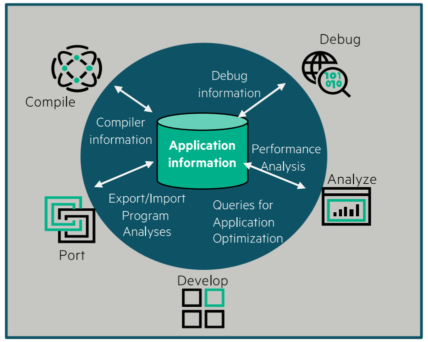
Performance Analysis: Motivation

- Very tempting to skip performance analysis when tests validate and time to solution is smaller after port or algorithm improvement.
- But performance does matter!
  - Might still be inefficient most of the of time.
  - Poor code performance can affect other users as well, for example because of bad I / O access patterns.
  - Want to efficiently use expensive resources and get as much information as possible for the allocated resources.
  - Simulating larger models may only be feasible after optimization.



- Applies to various scenarios
  - Code has been ported to a new system or otherwise significantly changed.
  - Application is running in production since a while.
  - Makes extensive use of third-party libraries (distributed ML, dense LA, …) or even fully proprietary or it is mostly home-grown code.

Tools Overview

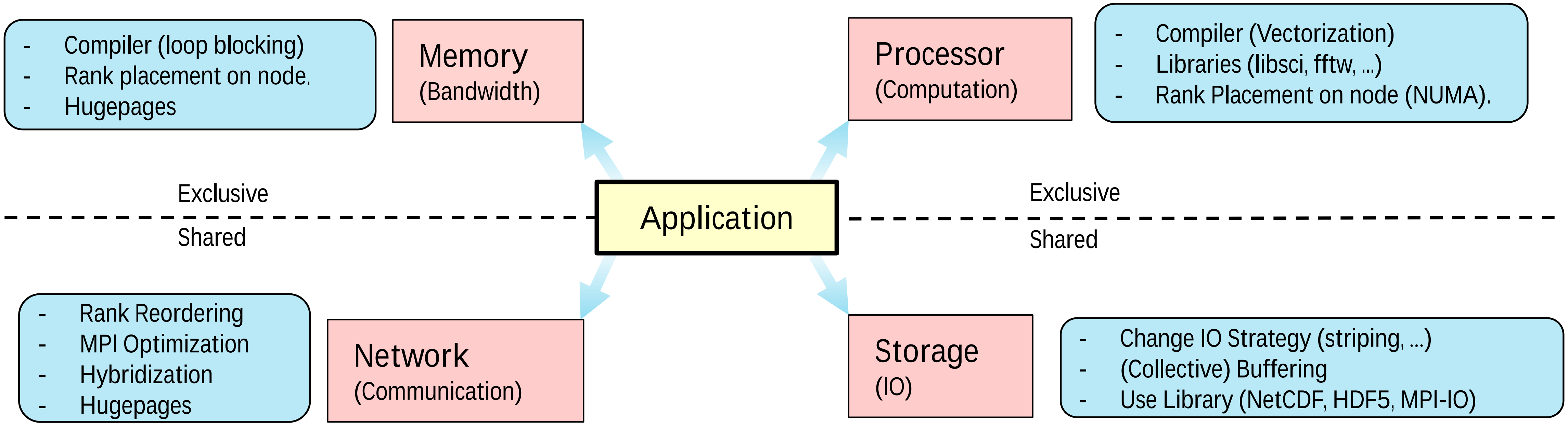
	**Light weight** No modification of the build chain, pre-/postprocessing. Get a first picture of a performance or problems during execution.	**In-depth** Requires modifications of the build system, recompile/relink, , pre-/postprocessing. Provides detailed information at user routine level.
Debugging Get your code up and running correctly.	ATP STAT CRAY_ACC_DEBUG	rocgdb, gdb4hpc valgrind4hpc, Sanitizers4hpc Linaro (Arm) Forge Tools
Profiling Locate performance bottlenecks.	perftools-lite	perftools rocprof Apprentice2 Reveal

General Remarks on Performance Analysis

- Performance is usually associated to FLOPs/sec.
  - But what if your code does many scattered memory references like in Graph Analytics and not many FLOPs ?
  - Chose the metric which suits your application (like time to solution or updates/sec instead of FLOPs/sec) and keep that metric throughout the optimization process.
- Use examples with different sizes for your experiments.
  - Small, medium, large, where node count differs by an order of magnitude (strong or weak scaling or both).
    - Try the same model with different grid sizes or number of particles
  - Do not use fully artificial examples but rather meaningful representatives of your target (large scale) simulation.

1. Start with low hanging fruits, i.e. avoid code modifications first.
   - Compiler flags, manual rank reordering, optimized libraries, huge pages, use hyperthreads, …
2. Use performance analysis tools
   - Identify critical code regions, guided rank reordering, Automatic parallelization (OpenMP), …
   - A good understanding of the workflow of your application (Communication, Computation, IO, …) helps to better interpret the profiles.

Bottlenecks and Remedies



- Good: One bottleneck which can be easily resolved without creating a new one.
- Bad: Several bottlenecks interacting with each other and changing over time.
- Need a profiler to identify bottleneck(s) and a model to estimate optimization potential.

Two fundamental ways of profiling

1. Sampling
   - By taking regular snapshots of the applications call stack we can create a statistical profile of where the application spends most time.
   - Snapshots can be taken at regular intervals in time or when some other external event occurs, like a hardware counter overflowing

Advantages
- Only need to instrument main routine
- Low Overhead – depends only on sampling frequency
- Smaller volumes of data produced

Disadvantages
- Only statistical averages available
- Limited information from performance counters

2. Event Tracing
   - Alternatively, we can record performance information every time a specific program event occurs, e.g. entering or exiting a function.
   - We can get accurate information about specific areas of the code every time the event occurs
   - Event tracing code can be added automatically or included manually through API calls.

Advantages
- More accurate and more detailed information
- Data collected from every traced function call not statistical averages

Disadvantages
- Increased overheads as number of function calls increases
- Huge volumes of data generated

AMD rocprof

ROCm Profiler

- The command line frontend for AMD's GPU profiling libraries
- Targets single process
- Allows for application traces and counter collection
- Included as part of ROCm software stack
- Can produce text output in various formats
- Output from `rocprof` can be visualised with Perfetto (https://ui.perfetto.dev/)
- Very low-level profiler i.e. raw GPU counters and traces
- For higher level profiling consider:
  - Omnitrace
  - Omniperf
  - Both require a separate local installation and will not be discussed further here

ROCm Profiler flags

- I/O and file names
  - `--timestamp <on|off>,` GPU kernel timestamps
  - `--basenames <on|off>,` truncate GPU kernel names
  - `-o <output csv file>,` direct counter information to a particular file name
  - `-d <data directory>,` send profiling data to a particular directory
  - `--stats,` generate kernel execution stats to file <output name>.stats.csv

- Profiling and tracing flags
  - `--hsa-trace,` trace GPU Kernels, host HSA events and HIP memory copies.
  - `--hip-trace,` trace HIP API calls
  - `--roctx-trace,` trace roctx markers
  - `--kfd-trace,` trace GPU driver calls

rocprof Profiler usage

```
$> module load rocm
```

```
outdir=rocprof_${SLURM_JOB_ID}
outfile="rocprof.csv"

if [[ "$SLURM_PROCID" == 0 ]]; then
      rocprof -d ${outdir} --basenames on --stats -o \
                       ${outdir}/${outfile} $*
else
      $*
fi
```

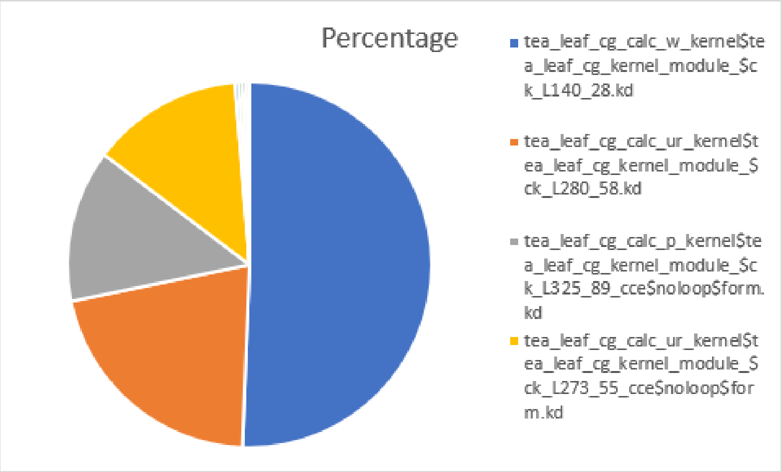
```
$> srun -n4 ./rocprof_trace.sh $EXE
```

- No MPI support, run on a single rank using a script like the above

rocprof stats output

- Two csv files are produced
  - rocprof.csv which contains details of all kernel calls
  - rocprof.stats.csv which contains statistics for each kernel (see below)

	A	B	C	D	E
1	Name	Calls	TotalDurationNs	AverageNs	Percentage
2	tea_leaf_cg_calc_w_kernel$tea_leaf_cg_kernel_module_$ck_L140_28.kd	74428	86463127329	1161701	50.62851192
3	tea_leaf_cg_calc_ur_kernel$tea_leaf_cg_kernel_module_$ck_L280_58.kd	74428	36135374977	485507	21.1590804
4	tea_leaf_cg_calc_p_kernel$tea_leaf_cg_kernel_module_$ck_L325_89_cce$noloop$form.kd	74428	23056011849	309776	13.50045513
5	tea_leaf_cg_calc_ur_kernel$tea_leaf_cg_kernel_module_$ck_L273_55_cce$noloop$form.kd	74428	22932613145	308118	13.42819898
6	tea_pack_message_top$pack_kernel_module_$ck_L438_30_cce$noloop$form.kd	74491	581211595	7802	0.340328636
7	tea_unpack_message_top$pack_kernel_module_$ck_L464_37_cce$noloop$form.kd	74491	562872733	7556	0.329590309
8	tea_pack_message_right$pack_kernel_module_$ck_L388_16_cce$noloop$form.kd	74491	504727525	6775	0.295543364
9	tea_unpack_message_right$pack_kernel_module_$ck_L414_23_cce$noloop$form.kd	74491	488141974	6553	0.285831689
10	field_summary_kernel$field_summary_kernel_module_$ck_L51_1.kd	3	17727178	5909059	0.010380155
11	tea_leaf_calc_residual_kernel$tea_leaf_common_kernel_module_$ck_L253_78_cce$noloop$form.kd	10	7449810	744981	0.004362239
12	tea_leaf_common_init_kernel$tea_leaf_common_kernel_module_$ck_L187_28_cce$noloop$form.kd	10	6893486	689348	0.004036483
13	tea_leaf_common_init_kernel$tea_leaf_common_kernel_module_$ck_L89_1_cce$noloop$form.kd	10	4036025	403602	0.002363296
14	tea_leaf_common_init_kernel$tea_leaf_common_kernel_module_$ck_L169_25_cce$noloop$form.kd	10	3089300	308930	0.001808941
15	tea_leaf_kernel_finalise$tea_leaf_common_kernel_module_$ck_L220_71_cce$noloop$form.kd	10	3087065	308706	0.001807632
16	tea_leaf_common_init_kernel$tea_leaf_common_kernel_module_$ck_L119_10_cce$noloop$form.kd	10	3020821	302082	0.001768843
17	tea_leaf_cg_init_kernel$tea_leaf_cg_kernel_module_$ck_L102_10.kd	10	2575538	257553	0.001508107
18	tea_leaf_cg_init_kernel$tea_leaf_cg_kernel_module_$ck_L93_7_cce$noloop$form.kd	10	2007854	200785	0.0011757
19	tea_leaf_cg_init_kernel$tea_leaf_cg_kernel_module_$ck_L68_1_cce$noloop$form.kd	10	1967372	196737	0.001151995
20	tea_leaf_common_init_kernel$tea_leaf_common_kernel_module_$ck_L110_7_cce$noloop$form.kd	10	1924493	192449	0.001126887
21	tea_leaf_calc_2norm_kernel$tea_leaf_common_kernel_module_$ck_L284_85.kd	10	1353292	135329	0.000792421
22	set_field_kernel$set_field_kernel_module_$ck_L40_1_cce$noloop$form.kd	1	199680	199680	0.000116923
23	tea_leaf_common_init_kernel$tea_leaf_common_kernel_module_$ck_L131_13_cce$noloop$form.kd	10	53120	5312	3.11E-05
24	tea_leaf_common_init_kernel$tea_leaf_common_kernel_module_$ck_L149_19_cce$noloop$form.kd	10	49280	4928	2.89E-05
25					



rocprof tracing usage

```
$> module load rocm
```

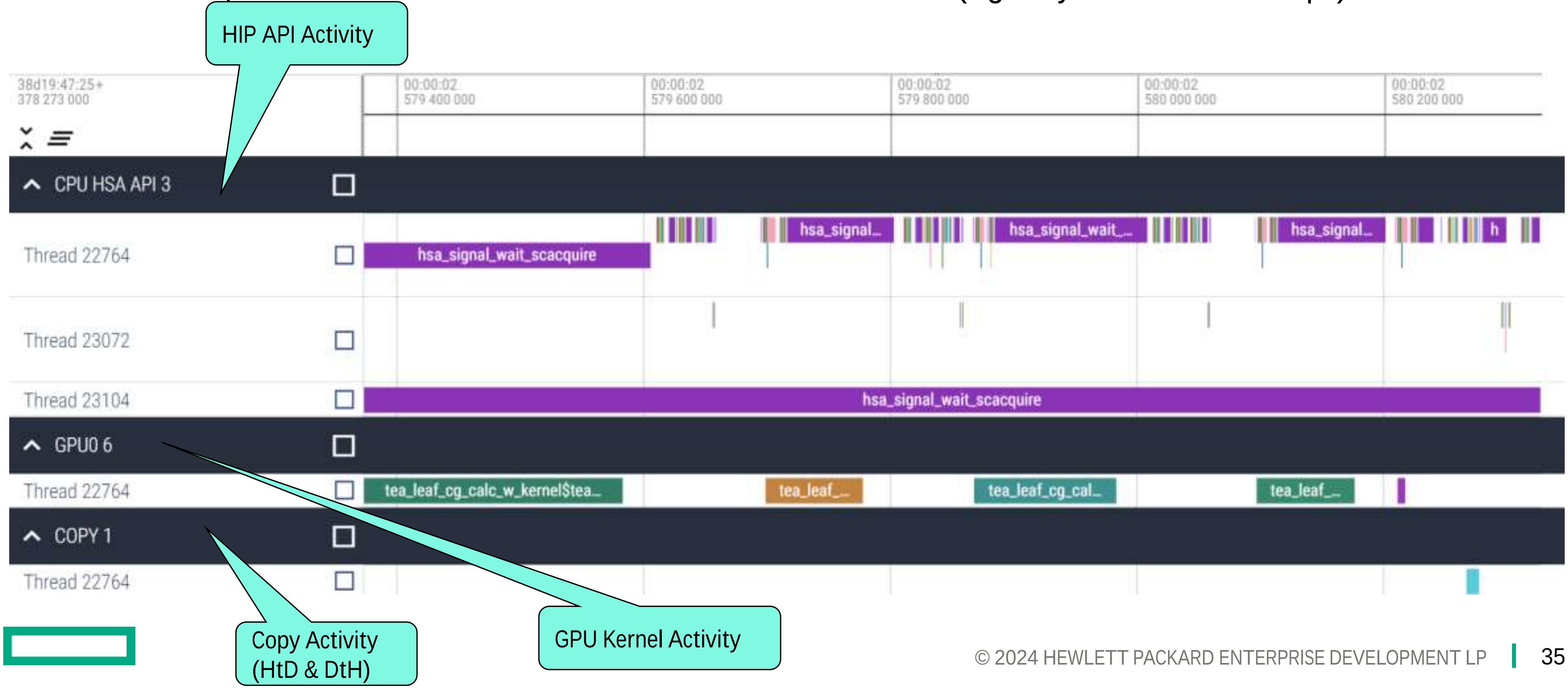
```
outdir=rocprof_${SLURM_JOB_ID}
outfile="rocprof.csv"

if [[ "$SLURM_PROCID" == 0 ]]; then
      rocprof -d ${outdir} --basenames on --hsa-trace -o \
                       ${outdir}/${outfile} $*
else
      $*
fi
```

```
$> srun -n4 ./rocprof_trace.sh $EXE
```

rocprof tracing output

- A .json file will be produced which can be visualised with Perfetto in Chrome browser
  - As this can produce GBs of data it is best to limit collection window (e.g. only run a few timesteps)



HPE Cray Perftools

Perftools Performance Measurement and Analysis Tool

Perftools Landscape

perftools-base

perftools-lite

perftools-lite-events

perftools-lite-loops

perftools-lite-gpu

For novices or convenience

perftools

pat_build
pat_report
API

For in-depth performance analysis. Includes everything from `lite`.

grid_order

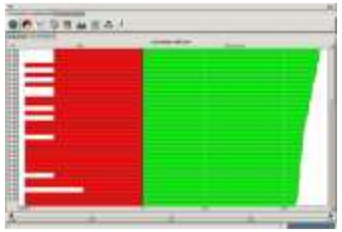
MPI rank order list generator

pat_help
pat_info

Advanced help system

Reveal

Apprentice2



Analysis, visualization, and parallelization.

Generate an Event Profile for GPU Experiments

```
$> module load perftools-lite-gpu
```

- If `perftools-lite` module not loaded, load subsequently `perftools-base` and `perftools-lite-gpu`.

```
$> rm app.exe; make
```

- Only relink of `app.exe` necessary if object files and user libraries have been generated with another `perftools-lite*` module.
- Otherwise do a `make clean; make`

```
$> srun -n 4 app.exe >& job.out
```

- The report is printed to stdout.
- Successful execution creates a `app.exe+*/` directory for further analysis.

Functions with significant time

- From the TeaLeaf mini-app (OpenMP offload)

```
CrayPat/X:  Version 22.12.0 Revision 8379d561d  11/09/22 20:11:43
Experiment:                  lite  lite-gpu
Number of PEs (MPI ranks):      4
Numbers of PEs per Node:        4
Numbers of Threads per PE:      1
Number of Cores per Socket:    32
Execution start time:  Mon Feb 19 10:48:07 2024
System name and speed:  nid200001  2.842 GHz (nominal)
AMD   Milan                CPU  Family: 25  Model:  1  Stepping:  1
Core Performance Boost:  12 PEs have CPB capability



Avg Process Time:       253.54 secs
High Memory:          12,794.4 MiBytes     3,198.6 MiBytes per PE
I/O Read Rate:        4.433147 MiBytes/sec
I/O Write Rate:   3,297.829910 MiBytes/sec

Notes for table 1:

  This table shows functions that have significant exclusive time,
    averaged across ranks.
  For further explanation, use:  pat_report -v -O profile ...
```

```
Table 1:  Profile by Function Group and Function

  Time% |       Time |     Imb. |  Imb. |        Calls | Group
        |            |     Time | Time% |              |  Function=[MAX10]
        |            |          |       |              |   PE=HIDE
        |            |          |       |              |    Thread=HIDE

 100.0% | 236.110242 |       -- |    -- | 14,148,837.8 | Total
|---------------------------------------------------------------------------
|  93.2% | 220.027578 |       -- |    -- |  6,703,790.0 | HIP
||--------------------------------------------------------------------------
||  74.2% | 175.296494 | 1.498353 |  1.1% |    744,794.0 | hipStreamSynchronize
||   4.7% |  11.067520 | 0.793795 |  8.9% |    297,917.0 | hipMemcpyDtoH
||   4.4% |  10.282723 | 0.474613 |  5.9% |    298,030.0 | hipMemcpyHtoD
||   4.2% |   9.936957 | 1.145356 | 13.8% |  2,383,514.0 | hipEventRecord
||   2.5% |   5.866568 | 1.567319 | 28.1% |  1,191,757.0 | hipEventSynchronize
||   2.0% |   4.663409 | 0.803697 | 19.6% |  1,191,757.0 | hipEventElapsedTime
||==========================================================================
|   2.6% |   6.167850 |       -- |    -- |    148,919.0 | MPI_SYNC
||--------------------------------------------------------------------------
||   2.2% |   5.204909 | 4.415734 | 84.8% |    148,886.0 | mpi_allreduce_(sync)
||==========================================================================
|   2.0% |   4.648060 |       -- |    -- |  3,798,224.2 | USER
|   1.3% |   3.060905 |       -- |    -- |  2,606,797.0 | OACC
|===========================================================================
```

Memory Transfer Between Host and Device

- From the Himeno benchmark (OpenMP offload)

```
Notes for table 3:

  This table shows functions that have significant exclusive host or
    accelerator time, averaged across ranks, and also data copied in
    and out, and event counts.
  For further explanation, use:  pat_report -v -O acc_fu ...

Table 3:  Time and Bytes Transferred for Accelerator Regions

  Time% |       Time |    Acc |     Acc |  Acc Copy |  Acc Copy |     Events | Function=[max10]
        |            |  Time% |    Time |        In |       Out |            |  PE=HIDE
        |            |        |         | (MiBytes) | (MiBytes) |            |   Thread=HIDE

 100.0% | 236.110242 | 100.0% | 198.65 |   111,381 |   105,759 | 10,427,758 | Total
|-----------------------------------------------------------------------------
|  74.2% | 175.296494 |     -- |     -- |        -- |        -- |    744,794 | hipStreamSynchronize
|   4.7% |  11.067520 |     -- |     -- |        -- |    52,880 |    297,917 | hipMemcpyDtoH
|   4.4% |  10.282723 |     -- |     -- |    55,691 |        -- |    298,030 | hipMemcpyHtoD
|   4.2% |   9.936957 |     -- |     -- |        -- |        -- |  2,383,514 | hipEventRecord
|   2.5% |   5.866568 |     -- |     -- |        -- |        -- |  1,191,757 | hipEventSynchronize
|   2.2% |   5.204909 |     -- |     -- |        -- |        -- |          0 | mpi_allreduce_(sync)
|   2.0% |   4.663409 |     -- |     -- |        -- |        -- |  1,191,757 | hipEventElapsedTime
|   0.0% |   0.085721 |  43.7% |  86.77 |        -- |        -- |     74,428 | tea_leaf_cg_calc_w_kernel$tea_leaf_cg_kernel_module_.ACC_ASYNC_KERNEL@li.140
|   0.0% |   0.084245 |  11.8% |  23.48 |        -- |        -- |     74,428 | tea_leaf_cg_calc_ur_kernel$tea_leaf_cg_kernel_module_.ACC_ASYNC_KERNEL@li.273
|   0.0% |   0.081310 |  11.9% |  23.54 |        -- |        -- |     74,428 | tea_leaf_cg_calc_p_kernel$tea_leaf_cg_kernel_module_.ACC_ASYNC_KERNEL@li.325
```

Observations and Remarks

- No intervention needed for build system and batch scripts.
  - Only make sure to use the compiler driver wrappers `CC`, `cc`, and `ftn`.
- What we did not see with these simple tests:
  - `perftools-lite` can produce rank reordering files for MPI to optimize the communication.
  - The resulting `app+exe*/` directory can be processed with `pat_report`, Apprentice2, and Reveal for further analysis. From the sample experiment one can retrieve hardware performance counter information.
- Tailored profiling, i.e., for specific routines, trace groups, or specific portions of the code is not possible.
  - Need the regular `perftools` module for this in-depth analysis.
  - Compile application and then use e.g., `pat_build -u -g mpi,hip`
- `CRAYPAT_LITE` environment variable can be used to distinct output files.
- Record Subset of PEs during execution: `export PAT_RT_EXPFILE_PES=0,4,5,10`

Further analysis (without re-running)

- Generate full report

```
$> pat_report app.exe+pat*/ > rpt
```

- Generate report with call tree (or by callers)

```
$> pat_report -O calltree+src
$> pat_report -O callers+src
```

```
 100.0% | 236.110242 | 14,148,837.8 | Total
|-----------------------------------------------------------------------------
|  78.8% | 186.076820 |  3,128,878.2 | ETC
||----------------------------------------------------------------------------
||  76.1% | 179.794808 |  1,936,414.0 | _cray_acc_wait
|||---------------------------------------------------------------------------
3||  37.1% |  87.570638 |    223,284.0 |
tea_leaf_cg_calc_w_kernel$tea_leaf_cg_kernel_module_.ACC_SYNC_WAIT@li.149:tea_leaf_cg_kernel.f90:line.149
4||         |            |              |
tea_leaf_cg_calc_w_kernel$tea_leaf_cg_kernel_module_.ACC_REGION@li.140:tea_leaf_cg_kernel.f90:line.149
5||         |            |              |
tea_leaf_cg_calc_w_kernel$tea_leaf_cg_kernel_module_:tea_leaf_cg_kernel.f90:line.140
6||         |            |              |
tea_leaf_cg_calc_w$tea_leaf_cg_module_.LOOP@li.63:tea_leaf_cg.f90:line.67
7||         |            |              |
tea_leaf_cg_calc_w$tea_leaf_cg_module_.REGION@li.64:tea_leaf_cg.f90:line.63
8||         |            |              |      tea_leaf_cg_calc_w$tea_leaf_cg_module_:tea_leaf_cg.f90:line.62
9||         |            |              |       tea_leaf$tea_leaf_module_:tea_solve.f90:line.348
10|         |            |              |        diffuse_.ACC_DATA_REGION@li.45:diffuse.f90:line.97
11|         |            |              |         diffuse_:diffuse.f90:line.45
12|         |            |              |          tea_leaf_:tea_leaf.f90:line.72
```

- Show each MPI rank or each OpenMP thread in report

```
$> pat_report -s pe=ALL
$> pat_report -s th=ALL
```

- Generate a preview of data before processing the full report

```
$> pat_report -Q1
```

  - Produces report from single (lexically first) '.ap2' file
  - Useful for jobs with large number of processes

Further analysis (without re-running)

- Generate report from specific subset of ranks

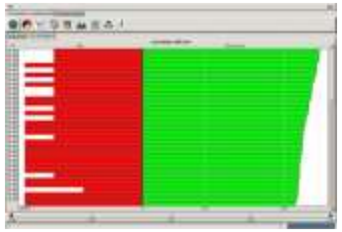
  `$> pat_report –s filter_input='pe==0'`

  - Report with only PE 0 data

  `$> pat_report –s filter_intput='pe<5'`

  - Report with data from first 5 ranks
  - Use `pat_help report filtering` for more details
- Don't see an expected function?
  - Use the `pat_report –P` option to disable pruning.
  - You should be able to see the caller/callee relationship with `pat_report -P -O callers`
  - Use '`pat_report –T`' to see functions that didn't take much time
  - Still don't see it? Check the compiler listing to see if the function was inlined.
- Also try the GUI for analyzing performance analysis results.

  `$> app2 app.exe+*/`



Documentation

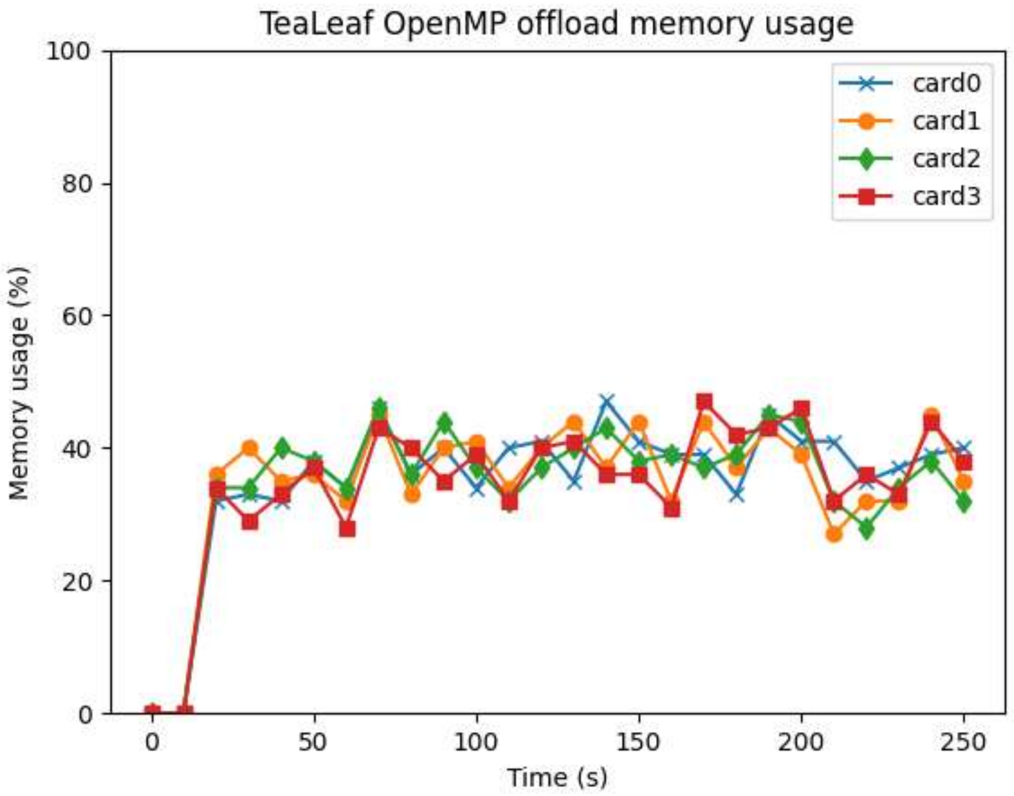
- Module help
  - module help perftools-base
  - module help perftools
  - module help perftools-lite
  - module help perftools-lite-*
- Man pages
  - man pat_build
  - man pat_report
- Advanced help system
  - pat_help
  - pat_info [[.ap2_file] [experiment_data_directory]…]
- Online at https://cpe.ext.hpe.com/docs/performance-tools/

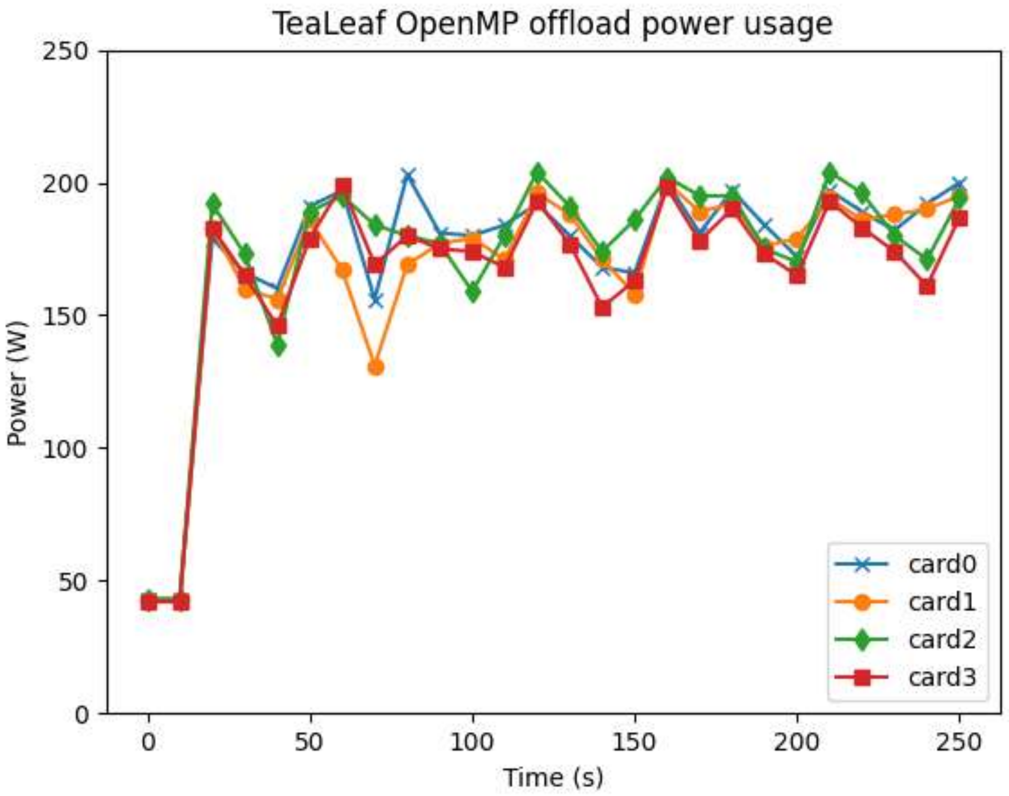
Energy usage of applications

- Historically performance has been measured for benchmarks by the total wall time in seconds
- Increasingly in the current climate energy consumption is important to bear in mind
- Therefore, it is important to consider both when running an application
  - Can one strike a balance between energy usage and time to solution?
  - e.g., EPCC reduced ARCHER2 CPU clock speed to default of 2GHz with minimal impact for most applications
- On HPE Cray EX liquid cooled systems multiple options are available
  - Perftools can report on memory usage of CPU applications
  - Counters can be found at `/sys/cray/pm_counters/`
- It is possible to monitor power/memory usage with AMD's **`rocm-smi`** (run in background with wrapper)

```
$>  rocm-smi --showpower –csv
$>  rocm-smi --showmemuse –csv
```

Energy usage of applications





[some of] The Tools we Didn't Cover

- AMD Visual Profiling and Analysis tools
  - Omniperf (rocprof + GUI & post-analysis for GPU specific analysis)
  - Omnitrace (wider system analysis, CPU + CPU + Network etc)
  - Radeon GPU Profiler (for computer graphics profiling)
  - Roctracer API (profiling library used for collecting data
- NVIDIA tools
  - NSight (debugging and profiling, IDE integration)
  - Visual Profiler (standalone visual profiling tool)
  - CUPTI (profiling library used for collecting data)
- Intel tools
  - Intel GDB (debugging with GDB but for intel devices)
  - Intel VTune (Visual profiling and debugging)
- Valgrind/Address sanitisers in detail
- Apprentice2/Reveal, Forge, TAU, VampirTrace, PAPI, various other tools/librarys that can support profiling efforts

If all else fails

- Print() from kernel
  - `__global__ void run_printf() { printf("Hello World\n"); }`

- AMD_LOG_LEVEL

```
enum LogLevel {
   LOG_NONE    = 0,
   LOG_ERROR   = 1,
   LOG_WARNING = 2,
   LOG_INFO    = 3,
   LOG_DEBUG   = 4
};
```

  - https://rocm.docs.amd.com/projects/HIP/en/latest/how-to/logging.html#logging-level