

Broad Performance Measurement Support for Asynchronous Multi-Tasking with APEX



Kevin Huck, Senior Research Associate, Oregon Advanced Computing Institute for Science and Society (OACISS), University of Oregon

Kevin Huck is a Senior Research Associate in the Oregon Advanced Computing Institute for Science and Society (OACISS) at the University of Oregon. Dr. Huck has several years of experience in the computer software industry prior to engaging in Computer Science research. He has worked in the areas of performance data mining, performance analysis, software tool design and software engineering. He is interested in the unique problems of intelligent, automated performance analysis of very large datasets as well as automated methods for diagnosing and treating performance problems both offline and with runtime controls. He has collaborated on over 70 publications and has served on several conference/workshop committees, including SC, ISC, ICCS, ICPP, VECPAR, IPDPS, and ParCo. His MS and PhD degrees in Computer and Information Science are from the University of Oregon, and his BS in Computer Science is from the University of Cincinnati.

Broad Performance Measurement Support for Asynchronous Multi-Tasking with APEX

Kevin Huck

 0000-0001-7064-8417

Email: khuck@cs.uoregon.edu

GitHub: <https://github.com/UO-OACISS/apex>

Documentation: <http://uo-oaciss.github.io/apex/>

Broad Performance Measurement Support for Asynchronous Multi-Tasking with APEX

Kevin Huck

khuck@cs.uoregon.edu

<https://github.com/UO-OACISS/apex>



UNIVERSITY OF OREGON

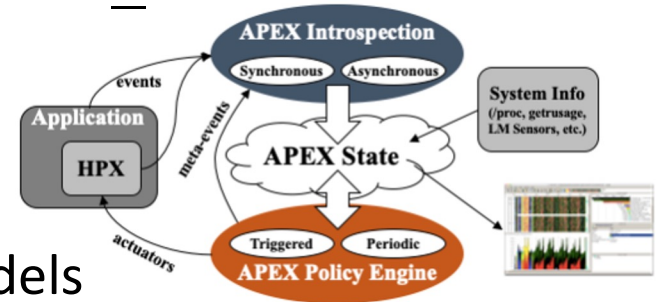


U.S. DEPARTMENT OF
ENERGY

Office of
Science

APEX Introduction

- Autonomic Performance Environment for eXascale
 1. Performance Measurement
 2. Runtime Adaptation
- Designed for AMT runtimes (HPX)
 - But works with “conventional” parallel models
- Focus on **task dependency** graph, not calling context graph
- Supports HPX, C/C++ threads, OpenMP, OpenACC, Kokkos, Raja, CUDA, HIP, (OneAPI, StarPU in dev)...
- <https://github.com/UO-OACISS/apex>
- Active Harmony* (Nelder Mead), Simulated Annealing, hill climbing for parametric search methods



APEX

*<https://www.dyninst.org/harmony>

APEX and HPX

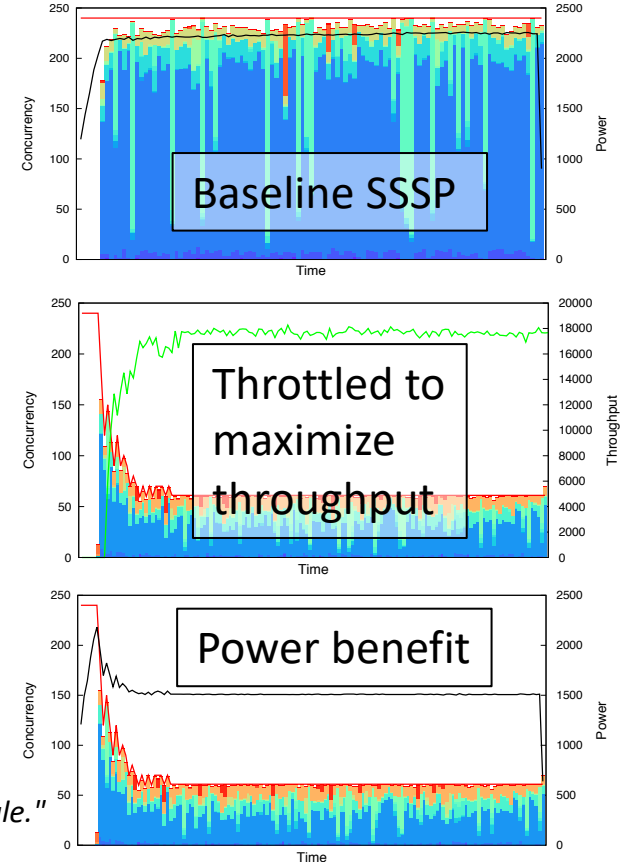
- **HPX**: Asynchronous Many-Task Runtime system in C++
- Data and task dependencies can be expressed with HPX **futures** and **continuations**, chained together in an **execution graph**.
- The graph can be built asynchronously
- HPX tasks are created, scheduled, executed, and usually yielded and resumed by the runtime system scheduler
- This is particularly challenging because many different OS threads may have participated in the execution of the HPX task during its lifetime, and the calling context tree is meaningless to the application developer because it consists of runtime system functions, not application tasks



APEX and HPX

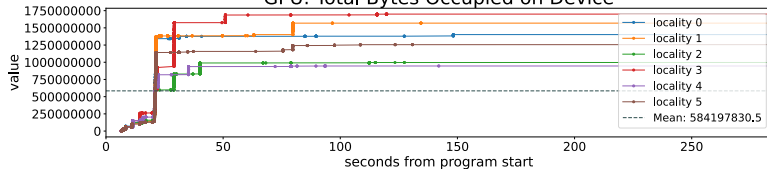
- APEX is **integrated into the HPX thread scheduler**, uniquely identifies each task with a **GUID**, and tracks all state transitions for a given task.
- Policy Engine used for **tuning heuristic control knobs** in HPX thread scheduler, networking
 - Soft power caps, maximize throughput, reduce network latency, ...

*Figures: Huck, et al. "An autonomic performance environment for exascale."
Supercomputing frontiers and innovations 2.3 (2015): 49-66.*



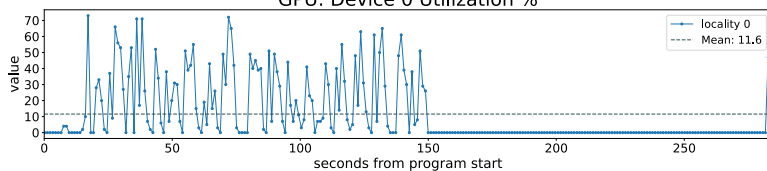
APEX example – Octo-Tiger (HPX)

GPU: Total Bytes Occupied on Device

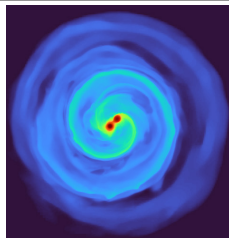


Tracking GPU memory usage with CUPTI

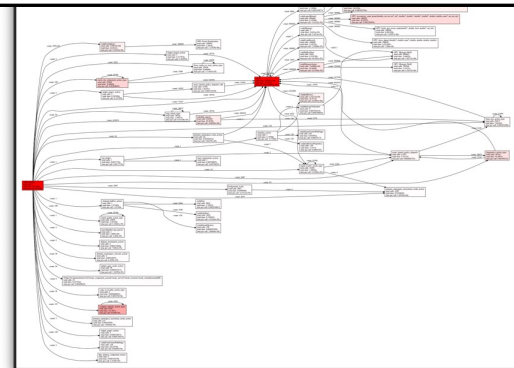
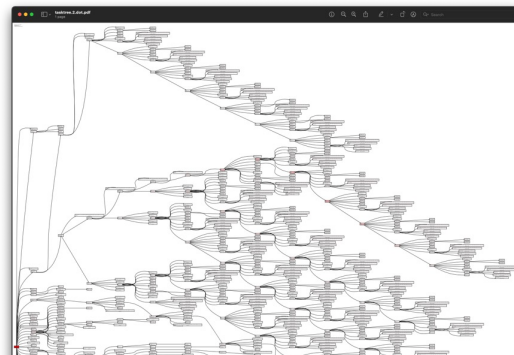
GPU: Device 0 Utilization %



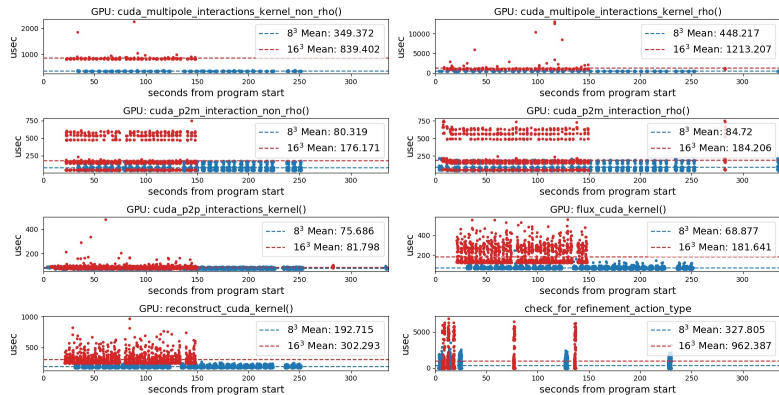
Monitoring GPU utilization with NVML library



Full task tree (above) and task graph (below) showing task dependencies



Comparing subgrid sizes and relative kernel performance with CUPTI device activity



<https://github.com/STELLAR-GROUP/octotiger>
<https://github.com/STELLAR-GROUP/hpx>

How is APEX different?

- **No shortage** of existing performance measurement tools
 - Primarily designed for **post-mortem** analysis
 - First-person measurement of **tied tasks/functions** on an OS thread – not untied tasks, runtime thread control, third-person measurement, runtime control of parameters
 - Not designed for **permanent integration** into applications
 - OS-thread context can be limiting
 - Vendor tools are great!...but limited to each vendors' architecture/process – can't easily do cross-platform analysis
- APEX helps address these needs

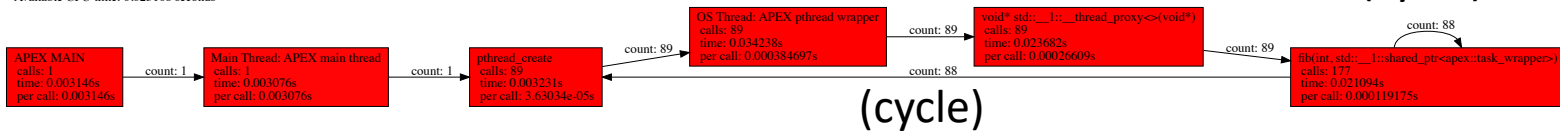
Measurement Capabilities

- **Timers** – “per-process” aggregation for all timers in runtime performance state
 - Start, stop, **yield**, **resume**
 - All times are inclusive (unless yielded/resumed)
- **Counters** – discrete sampling of some data point in time
 - Can be associated with timed regions, e.g. MPI_Send #bytes
 - Can be periodically captured and aggregated, e.g. power, utilization, hardware counters, OS counters
- **Task dependency chains**
 - Each task has a unique parent
 - APEX builds graphs/trees of dependencies at runtime

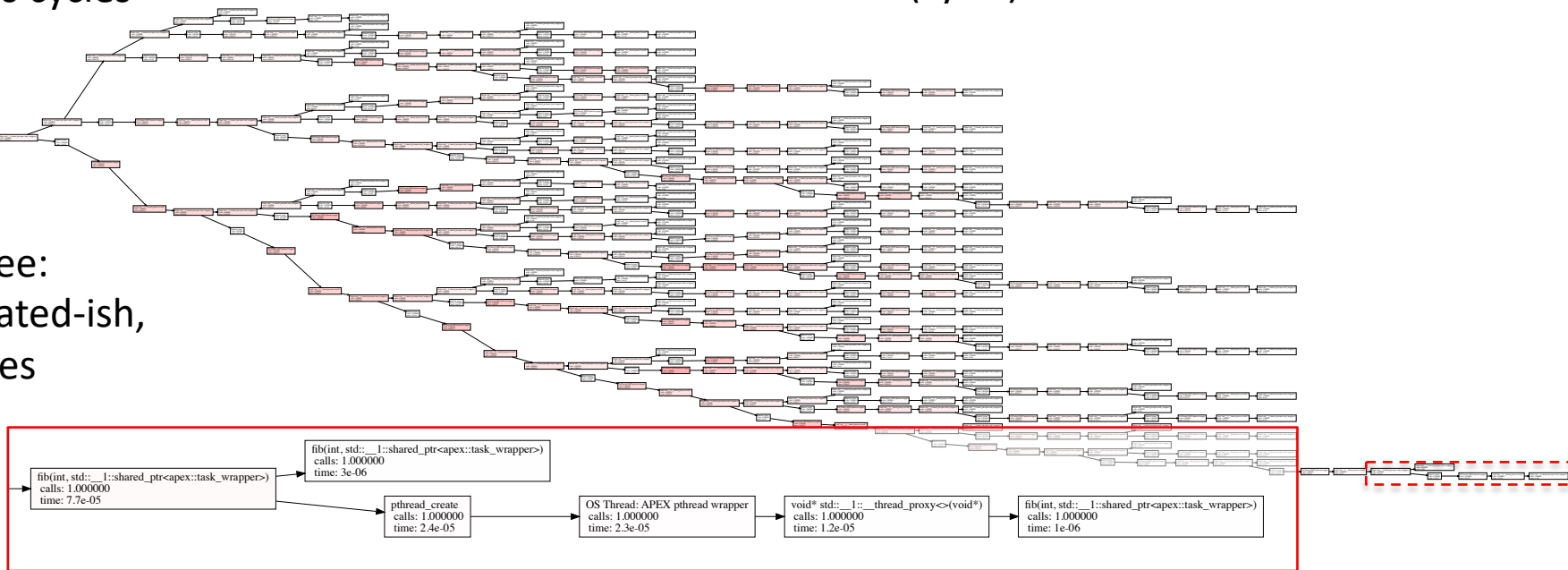
Example: C++ std::thread fib(10)

Task Graph:
aggregated,
contains cycles

Elapsed Time: 0.003146 seconds
Cores detected: 8
Worker threads observed: 90
Available CPU time: 0.025168 seconds



Task Tree:
aggregated-ish,
no cycles



UNIVERSITY OF OREGON

Tree:

Every child node only has 1 parent node



Hardware Counters

- APEX is integrated with **PAPI** (Performance Application Programming Interface) <https://icl.utk.edu/papi/>
- Portable access to native hardware counters
 - CPU (cache misses, FLOPs, instructions, stalls...)
 - GPU (cache misses, FLOPs, instructions, stalls...)
 - Off-core (permission dependent)
 - Node/OS health (LM Sensors, network)
 - Power/energy (RAPL, powercap...)
 - Filesystems
- HW counters collected with timers, or periodically
- APEX also monitors some/other counters natively

GPU Measurement

CUDA

- Support provided with **CUPTI** library
- Monitoring support provided with **NVML** library
- Hardware counters provided **CUPTI** and through **PAPI**
- Host callback and device activity dependencies linked using **correlation IDs**

HIP/ROCm

- Support provided by **Roctracer** and **Rocprofiler** libraries
- Monitoring support provided by **rocm-smi** library
- Hardware counters provided **CUPTI** and through **PAPI**
- Host callback and device activity dependencies linked using **correlation IDs**

Intel SYCL/DPC++/OneAPI support in development...

GPU Memory Tracking

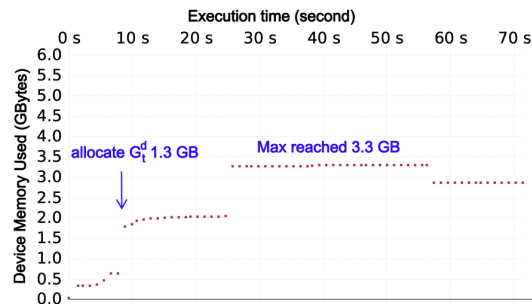
- For both CUDA and HIP, when memory is allocated or freed through the cuda/hip API, APEX captures:

- Allocation type (host/gpu)
- Bytes allocated
- thread ID that requested it
- Address of allocated memory
- Backtrace from when allocation happened

Motivating paper: Wei, Weile, et al. "Memory Reduction Using a Ring Abstraction Over GPU RDMA for Distributed Quantum Monte Carlo Solver." Proceedings of the Platform for Advanced Scientific Computing Conference, 2021.

- At application exit, any leaked allocations are reported to the user, similar to **cuda-memcheck**

- ...but finds leaks that it doesn't
- Counters saved by APEX (bytes allocated/freed/total – see figure)



(b) Distributed G^d_i method with sub-ring size of three.

Example Program: memory error

```
1 #include <Kokkos_Core.hpp>
2 #include <cmath>
3
4 int main(int argc, char* argv[]) {
5     Kokkos::initialize(argc, argv);
6     {
7         void * ptr;
8         // This memory will leak
9         cudaMalloc(&ptr, 1024);
10        int N = argc > 1 ? atoi(argv[1]) : 1000000;
11        int R = argc > 2 ? atoi(argv[2]) : 10;
12        double result;
13        Kokkos::parallel_reduce(N, KOKKOS_LAMBDA(int i, double& r) {
14            r+=i;
15        },result);
16        printf("%lf\n",result);
17    }
18    Kokkos::finalize();
19 }
```



CUDA-MEMCHECK version 11.6.124 ID:(46)

===== CUDA-MEMCHECK

===== ERROR SUMMARY: 0 errors

1024 bytes leaked at 0x1465937ea400 from
task cudaMalloc on tid 0 with backtrace:

gpu_device_malloc

addr=<0x1465fa0f409b> [{ (unknown) }
{0x1465fa0f409b}]

addr=<0x1465fa0f43b7> [{ (unknown) }
{0x1465fa0f43b7}]

addr=<0x1465fa0f6c1c> [{ (unknown) }
{0x1465fa0f6c1c}]

addr=<0x1465fe5e3143> [{ (unknown) }
{0x1465fe5e3143}]

addr=<0x1465fdfb247b> [{ (unknown) }
{0x1465fdfb247b}]

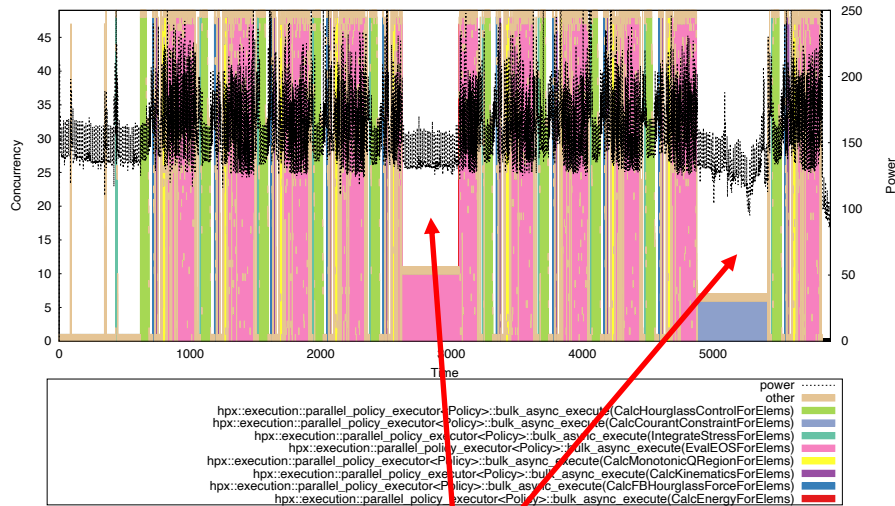
main [{/home/khuck/polaris/test/test.cpp}
{9,0}]

__libc_start_main [{/lib64/libc-2.31.so}
{0x1465fb4e434d}]

_start

[{/home/abuild/rpmbuild/BUILD/glibc-
2.31/csu/./sysdeps/x86_64/start.S}
{122,0}]

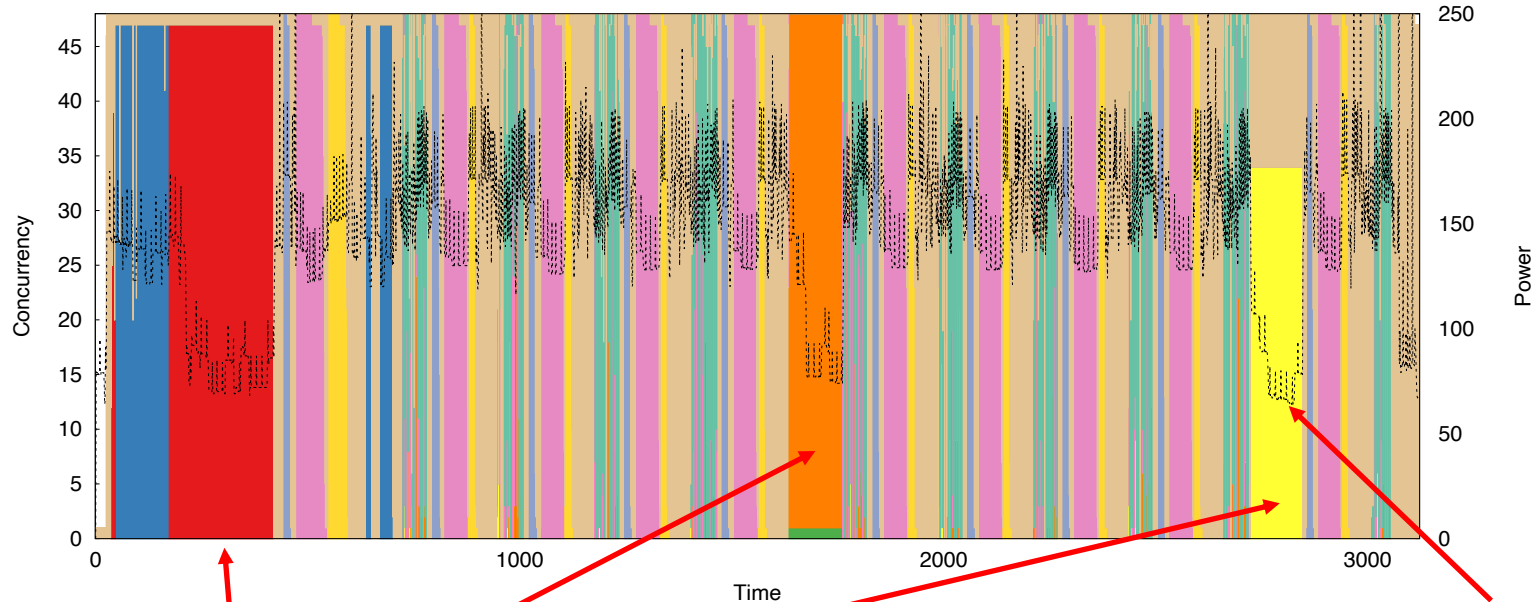
Concurrency Tracking



Helps identify regions of low concurrency

- Periodically sample all the currently executing tasks (timers, really)
- Aggregate across N timer types/names
- Example shown:
 - Kokkos Lulesh with HPX backend
 - Sampled 200 times per second
 - 10 iterations, size 256

Concurrency: OpenMP back end

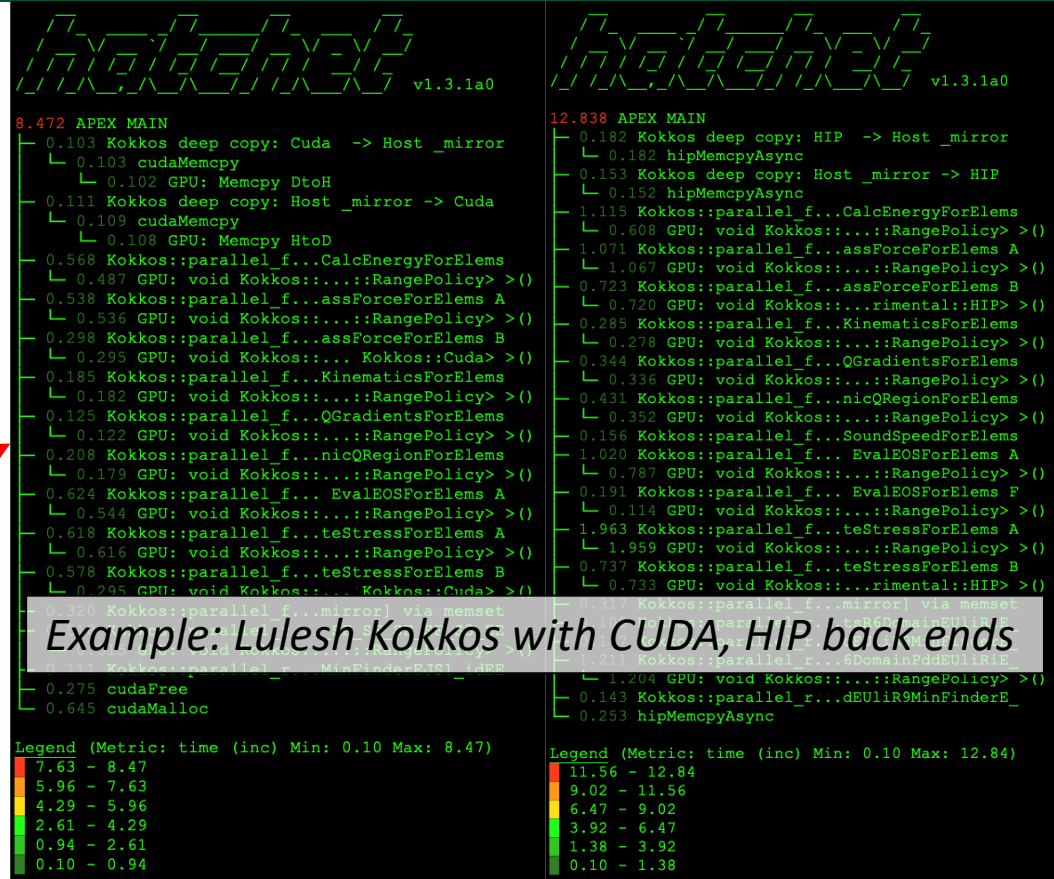


Looks like higher
concurrency...but
barriers aren't
progress

...as evidenced by
dips in power
usage

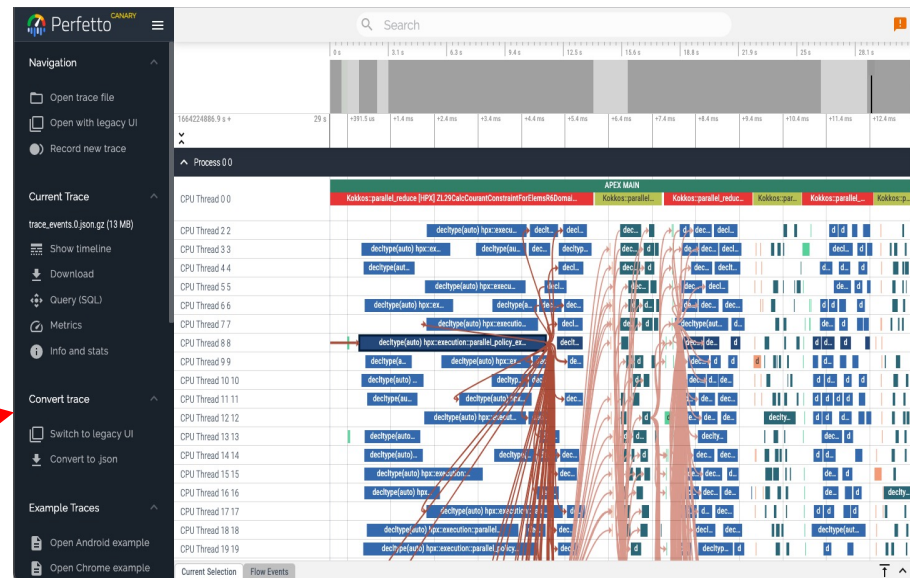
Profile Formats

- All timer & counter data
- Flat profile:
 - Text summary to screen – all ranks merged with MPI or HPX at end of execution
 - CSV (for Python ingestion)
 - TAU Profiles (ParaProf)
- Task graphs/trees:
 - Hatchet-like JSON (still working on importer library for Hatchet) <https://hatchet.readthedocs.io>
 - Graphviz dot files
 - Txt files (similar to Trilinos profiler output)



Trace Formats

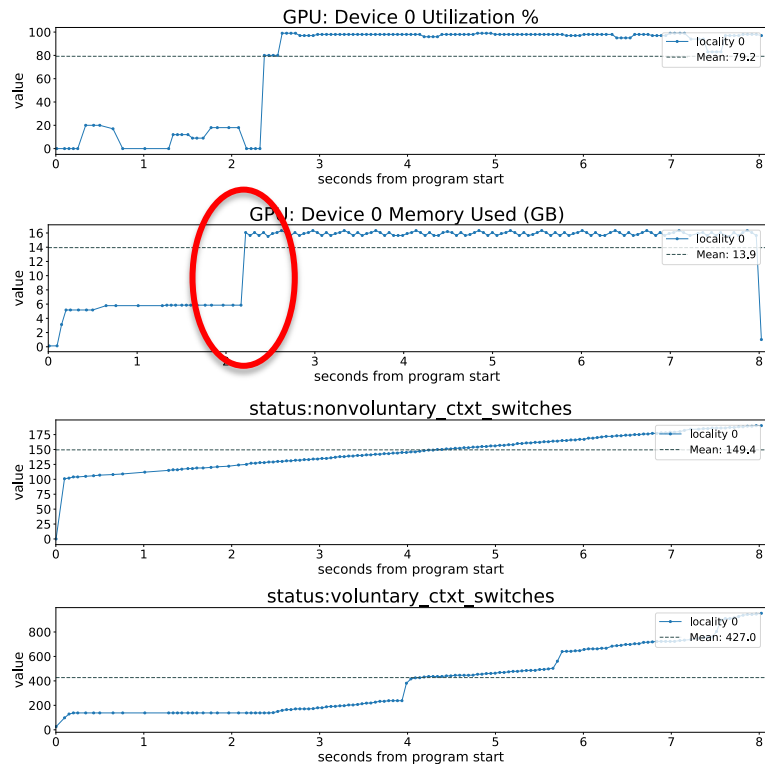
- **OTF2** up to v 2.3
 - 3.0 has API changes, APEX hasn't been updated yet
 - Visualized with Vampir or Traveler / JetLag
 - Problems with asynchrony, overlapping timers, high thread counts
- **Google Trace Events Format**
 - JSON support only (native support coming soon)
 - Visualized with Perfetto 
 - Some scaling issues (memory limit of web browser)
 - Handles asynchrony, overlapping timers just fine



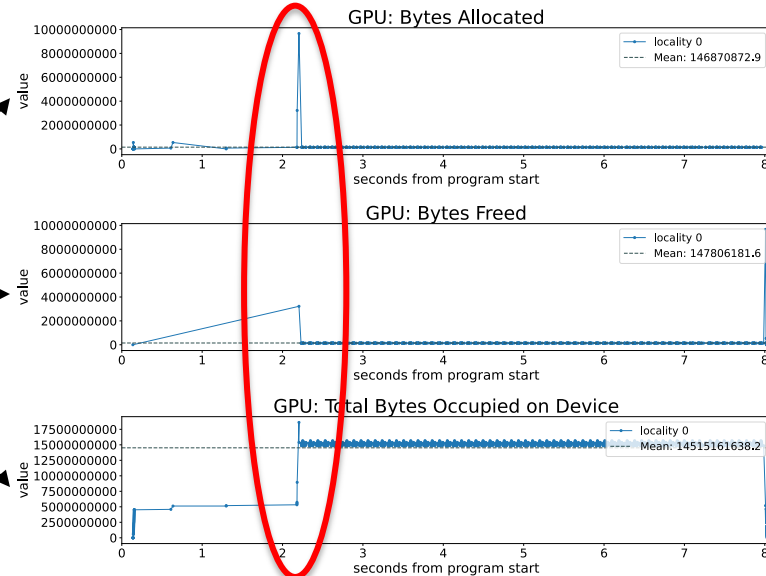
Example: Lulesh Kokkos with HPX back end

Scatterplots

Monitoring Data (periodic)



Progress Data (events)



Example: Lulesh Kokkos with CUDA back end

Supported Programming Models

- HPX
- POSIX / C++ threads
- OpenMP/OpenACC
- GPU offload: CUDA, HIP, OpenMP target
- Abstractions: Kokkos, Raja
- MPI (subset)
- In development: Intel Level0/OneAPI GPU support, StarPU

POSIX / C++ Threads

- APEX wraps the `pthread_create()` call:
 - Wraps target function with a proxy function
 - Times target function
 - Captures task dependency hierarchy between parent, child
- Provides support for C++ thread activity too
 - `std::thread`
 - `std::async`
 - Only on POSIX compliant systems



OpenMP and OpenACC

- OpenMP 5.0 included OMP-Tools (OMPT) API
 - Callbacks, query functions, sampling states
 - Buffer processing for **target offload** asynchronous activity
 - Tested with AMD Clang 5.0+, NVHPC 22.7+, Intel OneAPI 2022
- OpenACC profiling callbacks to intercept entry/exit of all OpenACC routines
 - CUDA/CUPTI provides support for device activity



Kokkos and Raja



- C++ abstraction models for **performance portability**
- 1 source code implementation to target different architectural / model back ends
 - Serial, Pthreads, OpenMP, OpenACC, CUDA, HIP, SYCL, etc.
- Both provide **host-side profiling callbacks** for tool support
- Kokkos includes a prototype “tuning” interface for tools to hook utilize at runtime
 - APEX has implemented tuning policies and tested with CUDA back end tuning Range, MDRange, Team policies

Examples: Lulesh with Kokkos

- <https://github.com/kokkos/kokkos-miniapps>
- Tested lulesh-2.0 mini-app
(<https://asc.llnl.gov/codes/proxy-apps/lulesh>)

- HPX
- OpenMP
- CUDA
- HIP

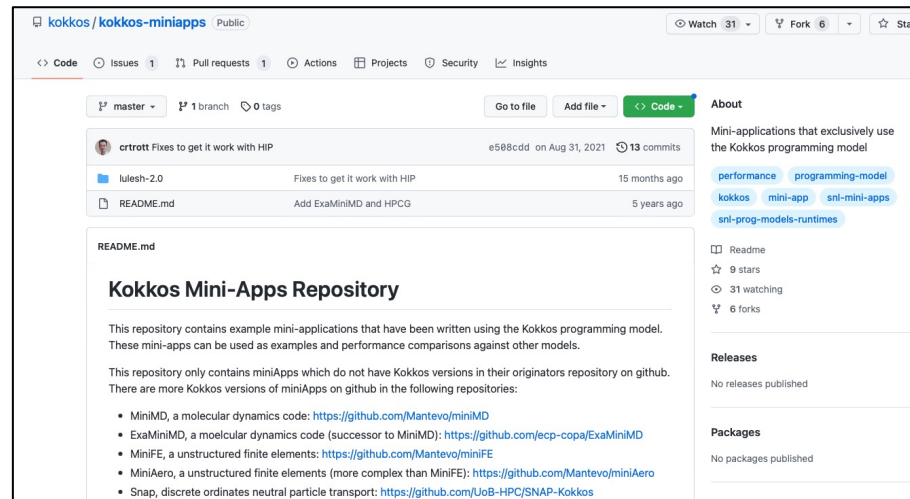
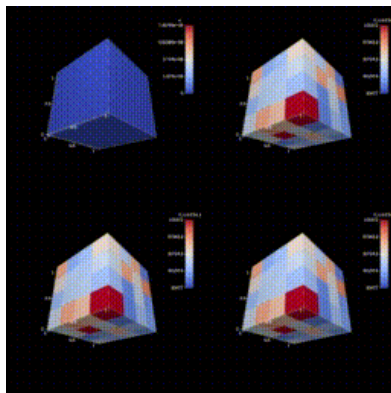


Figure: [LULESH 2.0.3](#) executed with 64 MPI ranks, measured by TAU. Time spent in main loop rendered every 100 timesteps by [Alpine-Ascent](#). Clockwise from upper left: Computed energy, accumulated time in main loop, time in main loop during last 100 timesteps, Δ time in main loop from previous 100 time steps. Source: Malony, et al. "When Parallel Performance Measurement and Analysis Meets In Situ Analytics and Visualization." Parallel Computing: Technology Trends. IOS Press, 2020. 521-530.

Lulesh: CUDA back end

```
Elapsed time: 8.96729 seconds
Total processes detected: 1
HW Threads detected on rank 0: 96
Worker Threads observed on rank 0: 1
Available CPU time on rank 0: 8.96729 seconds
Available CPU time on all ranks: 8.96729 seconds
```

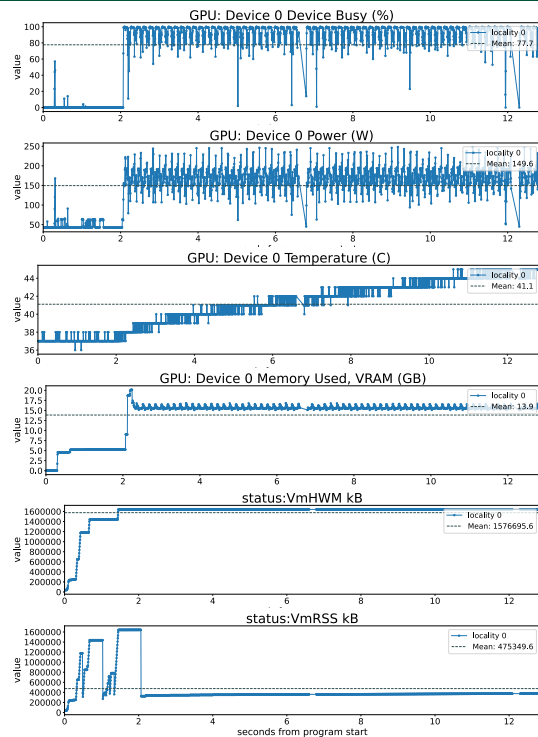
Counter		#samples	minimum	mean	maximum	stddev
1 Minute Load average	:	161	14.680	15.717	16.640	0.641
CPU Guest %	:	160	0.000	0.000	0.000	0.000
CPU I/O Wait %	:	160	0.000	0.001	0.192	0.015
CPU IRQ %	:	160	0.000	0.054	0.200	0.086
CPU Idle %	:	160	67.331	95.424	97.679	2.670
CPU Nice %	:	160	0.000	0.000	0.000	0.000

CPU Timers		#calls	#yields	mean	total	% total
APEX MAIN	:	1	0	8.967	8.967	100.000
CPU cudaDeviceSynchronize	:	22901	0	0.000	4.004	44.647
CPU cudaStreamSynchronize	:	4398	0	0.000	0.767	8.550
CPU cudaMalloc	:	946	0	0.001	0.727	8.106
GPU: Bytes Kokkos::parallel_for [Cuda, Dev:0] IntegrateStressF...	:	64	0	0.010	0.632	7.048
GPU: Device 0 Clock Kokkos::parallel_for [Cuda, Dev:0] EvalEOSForElems A	:	2240	0	0.000	0.630	7.031
GPU: Device 0 Clock Kokkos::parallel_for [Cuda, Dev:0] CalcEnergyForElems	:	2240	0	0.000	0.577	6.433
GPU: Device 0 Memory Kokkos::parallel_for [Cuda, Dev:0] Kokkos::parallel_for [Cuda, Dev:0] Kokkos::parallel_reduce [Cuda, Dev:0]	:					

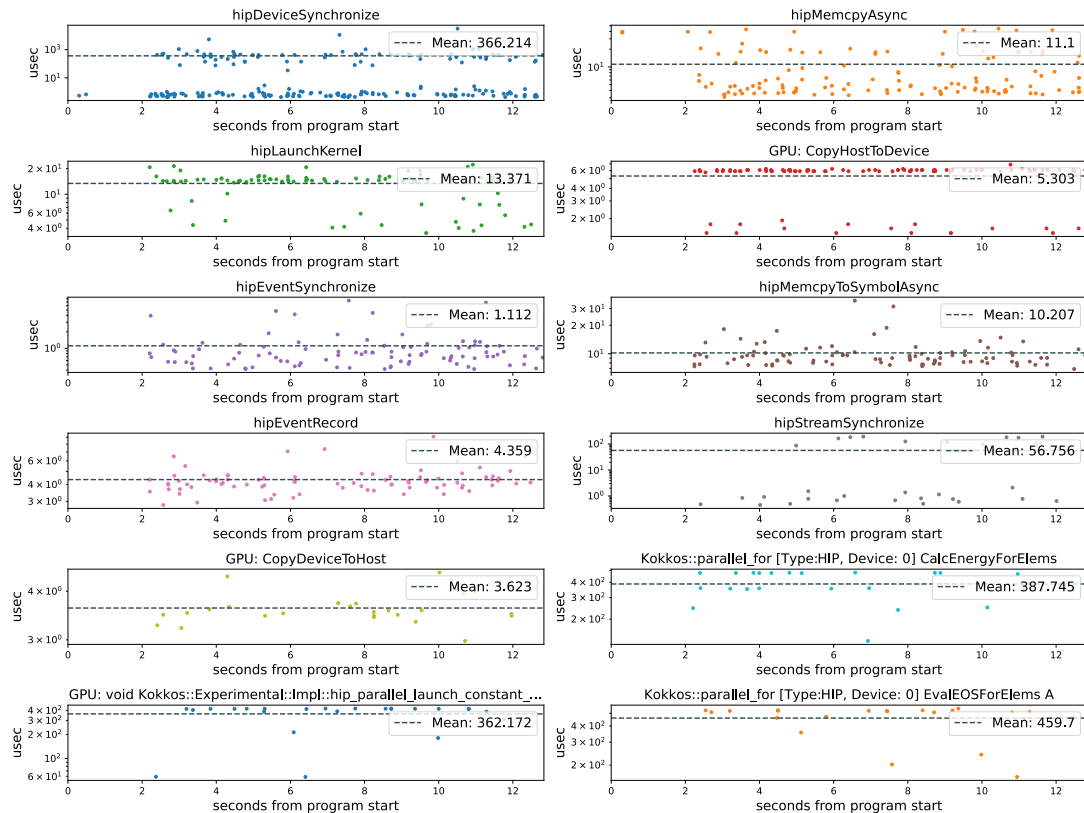
GPU Timers		#calls	mean	total	% total
GPU: Context Synchronize	:	22901	0.000	3.705	41.318
GPU: Stream Synchronize	:	4398	0.000	0.756	8.429
GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	:	64	0.010	0.628	7.000
GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	:	2240	0.000	0.543	6.051
GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	:	64	0.008	0.542	6.043
GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	:	64	0.008	0.524	5.846
GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	:	2240	0.000	0.487	5.432
GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	:	64	0.005	0.299	3.331
GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	:	64	0.005	0.298	3.322
GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	:	64	0.003	0.185	2.066
GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	:	704	0.000	0.178	1.990
GPU: Memcopy HtoD	:	10261	0.000	0.170	1.892

```
apex_exec --apex:kokkos_fence --apex:cuda \
--apex:monitor_gpu --apex:period 5000 \
${builddir}/lulesh-cuda/lulesh.cuda -s 256
```

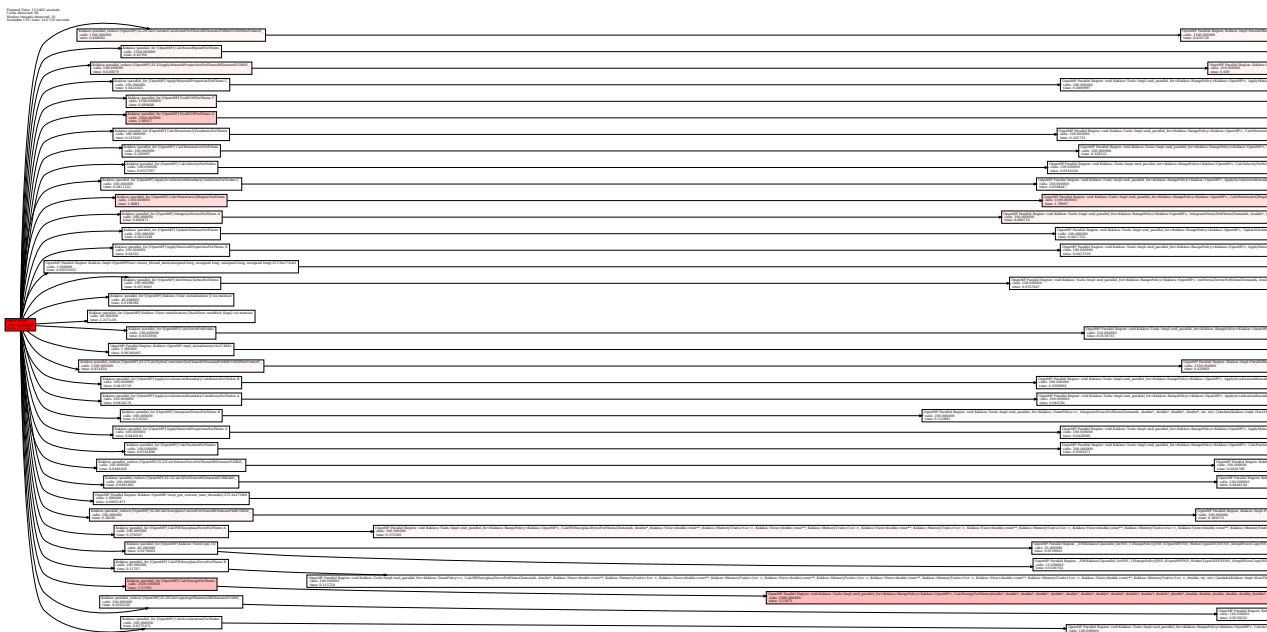
Lulesh: HIP back end



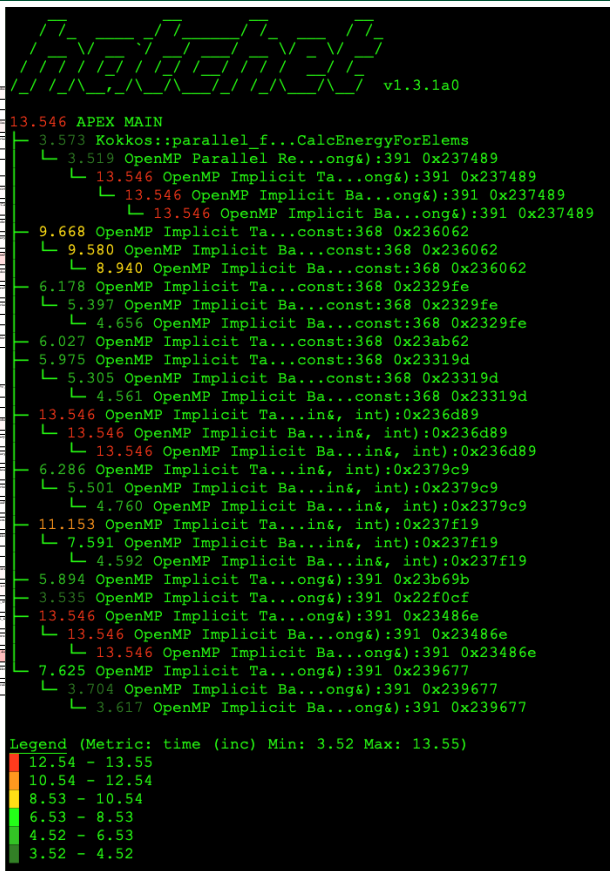
```
apex_exec --apex:kokkos_fence --apex:scatter \
--apex:hip --apex:monitor_gpu \
--apex:period 5000 ${builddir}/lulesh-hip/lulesh.hip -s 256
```



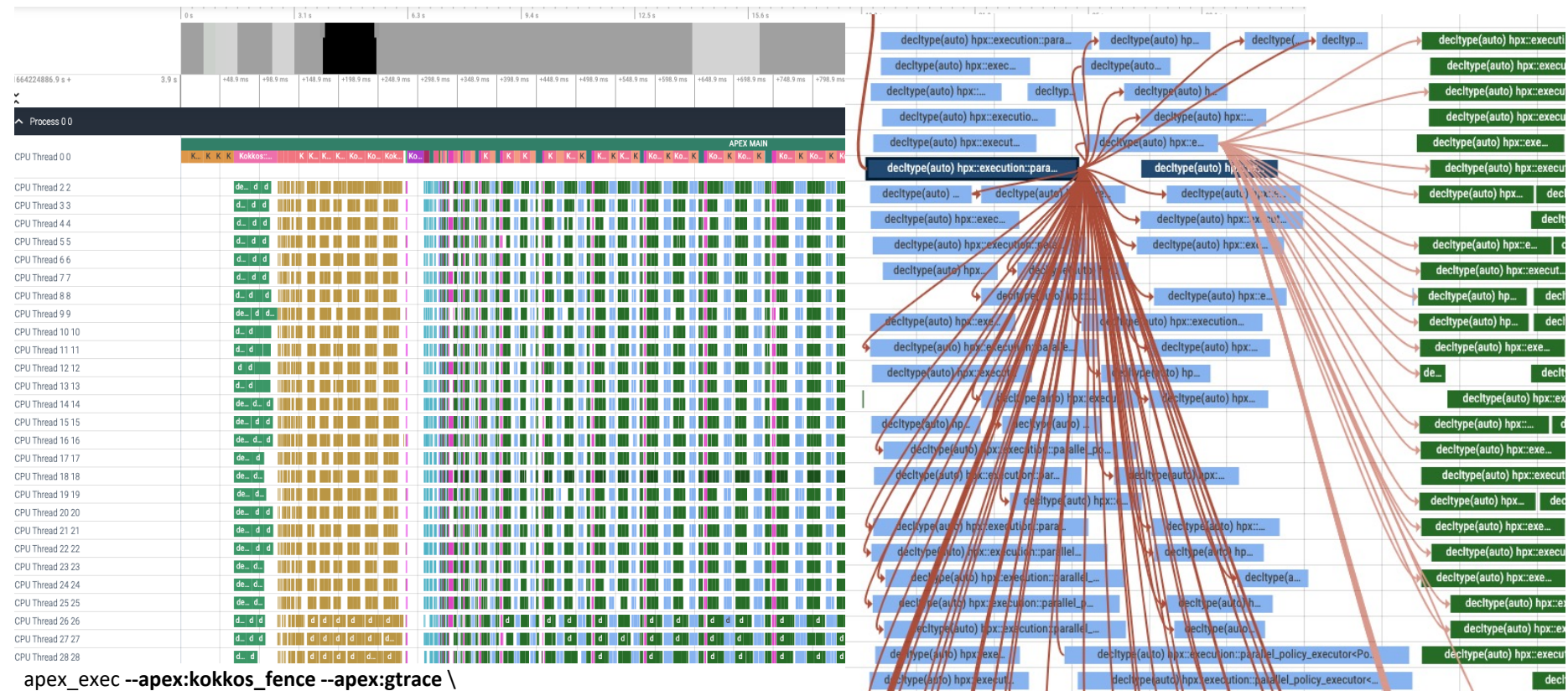
Lulesh: OpenMP back end



```
apex_exec --apex:kokkos_fence --apex:ompt --apex:ompt_details \  
--apex:tasktree ${builddir}/lulesh-openmp/lulesh.host -s 256 -p -i 10
```



Lulesh: HPX back end



Future Work

- **Intel Level0/OneAPI** support has been prototyped, but not yet merged
- **StarPU** support added by Camille Coti, needs additional testing and tighter integration
- Perfetto native trace output
- PowerAPI integration for broader power/energy support
- Kokkos runtime autotuning development

Acknowledgements

Parts of this research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

