

Performance Measurement Support for Asynchronous Multi-Tasking with APEX

Kevin Huck, Allen Malony, Sameer Shende

khuck@cs.uoregon.edu

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

<https://github.com/khuck/apex-tutorial>



Office of
Science



UNIVERSITY OF OREGON



Download slides:

These slides are available at:

<https://github.com/khuck/apex-tutorial>

Click on: “slides”

Click on: “2023-CSC-APEX.pdf”

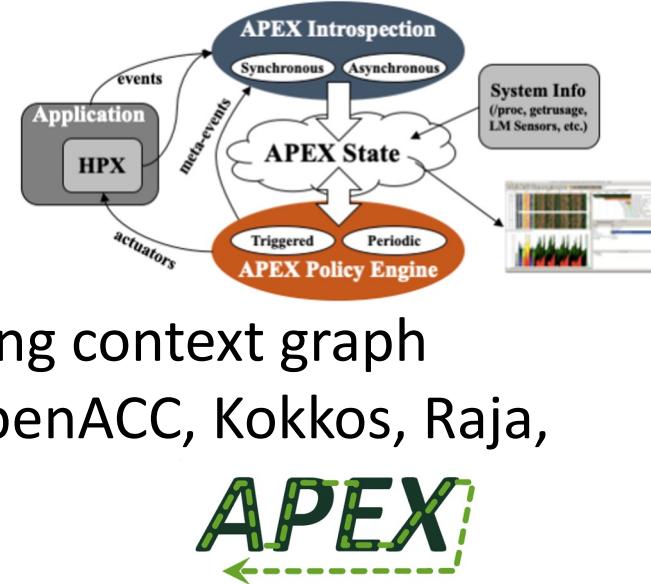
...download if you like, or just follow along on github.

Tutorial setup on Mahti:

```
# Clone the APEX tutorial repository:  
git clone https://github.com/khuck/apex-tutorial.git  
# change directory  
cd apex-tutorial  
# load modules  
source sourceme.mahti  
# build the tutorial examples  
.src/scripts/mahti.sh  
# submit the job  
sbatch ./job.sh
```

APEX Introduction

- Autonomic Performance Environment for eXascale
- **Performance Measurement**
- **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...
- <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 Publications

<https://doi.org/10.14529/jsfi150305> 2015

The screenshot shows the homepage of the SUPERCOMPUTING FRONTIERS AND INNOVATIONS journal. The header includes the journal's name and a sub-header 'An International Journal'. Navigation links for 'Focus and Scope', 'Editorial Board', 'Current Issue', and 'Archive' are visible. The main content area displays an article titled 'An Autonomic Performance Environment for Exascale' by Kevin A. Huck, et al. The article summary, authors, and a PDF download link are shown. The sidebar contains links for 'For Authors' (Author Guidelines, Publishing Ethics, Contact), 'Make a Submission', and a search bar.

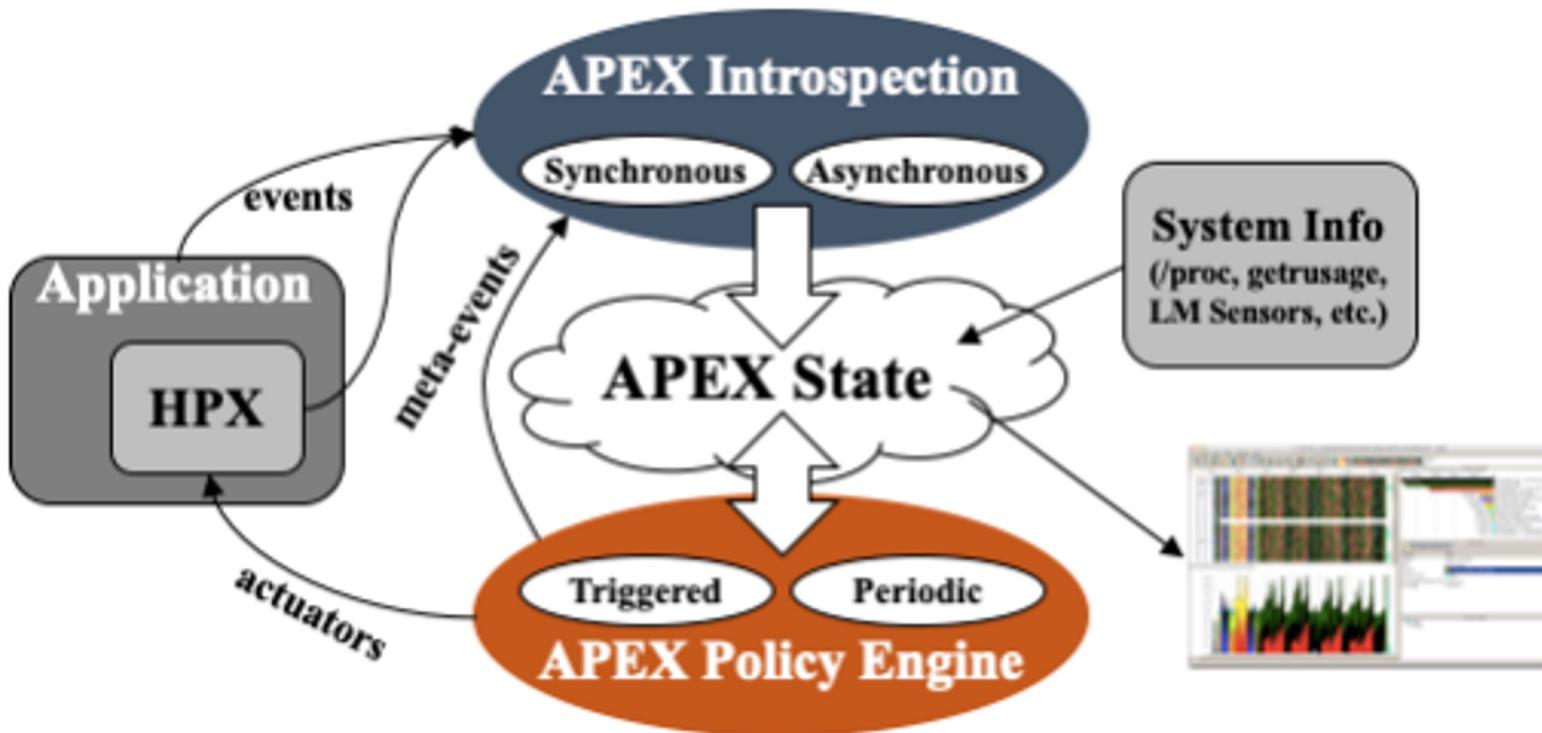
<https://doi.org/10.1109/ESPM256814.2022.00008> 2022

The screenshot shows a search result for a conference paper on the IEEE Xplore platform. The search bar indicates the search term 'Conferences > 2022 IEEE/ACM 7th International...'. The results list a paper titled 'Broad Performance Measurement Support for Asynchronous Multi-Tasking with APEX' by Kevin A. Huck, et al. The paper is published by IEEE and has citation and PDF download links. The abstract, document sections, and a detailed description of the APEX library are provided. The right sidebar features 'More Like This' recommendations and other conference papers.

How is APEX different?

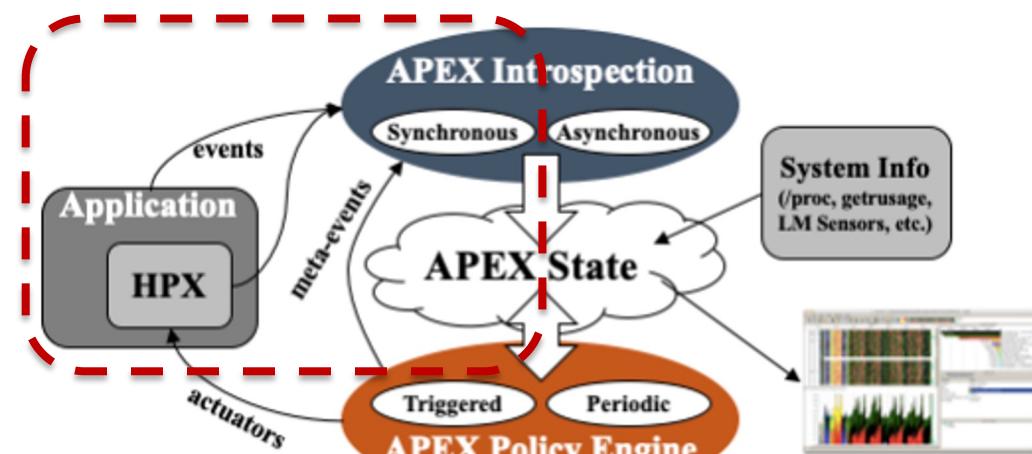
- No shortage of existing performance measurement tools
 - Primarily designed for post-mortem analysis
 - First-person measurement of perfectly nested, tied tasks/functions on an OS thread – not untied tasks, third-person measurement, runtime control
 - Not often designed for permanent integration into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

APEX system diagram



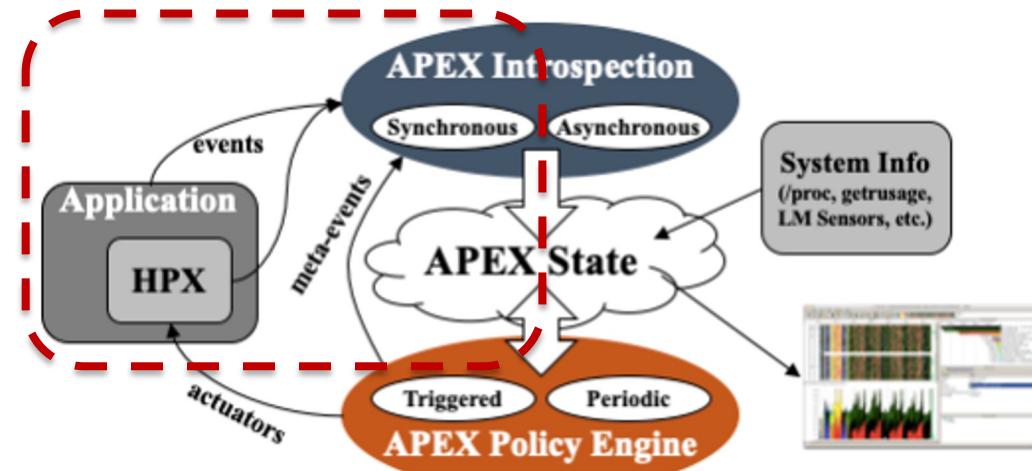
Performance Measurement

- Timers designed for asynchronous tasking runtimes
 - Task creation
 - Start task
 - Create dependent tasks
 - Yield task
 - Resume task
 - Stop task
- “First-person” measurement – what is this runtime thread doing?



Performance Measurement

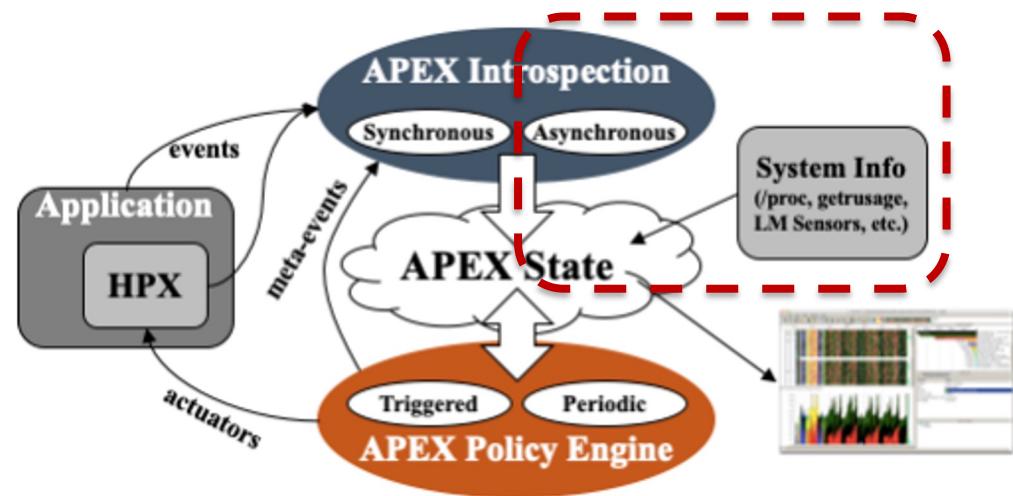
- Synchronous counters associated with timers
 - Bytes sent
 - Bytes received
 - GPU kernel data
 - Memory allocated
 - Memory freed



- “First-person” measurement – what is this runtime thread doing?

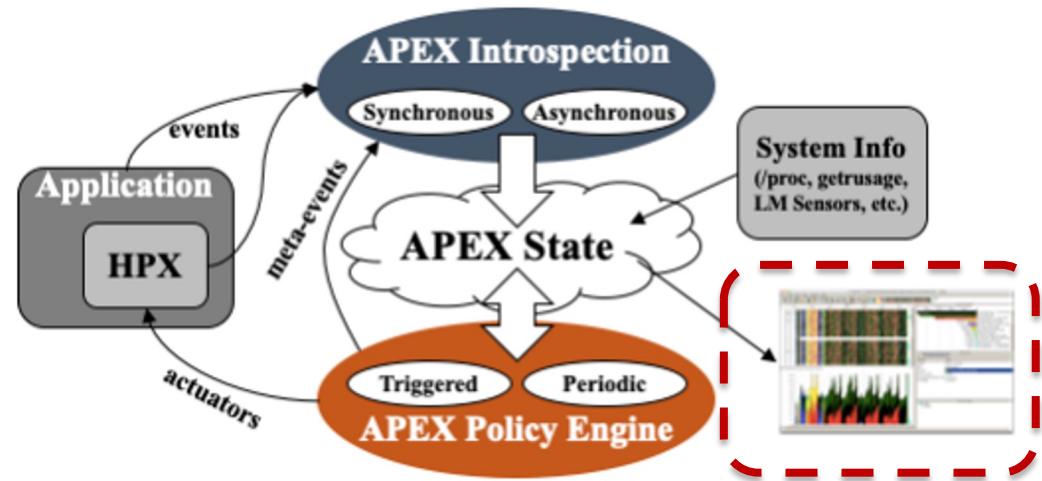
System Monitoring

- Asynchronous counters collected periodically to measure system health
 - Memory usage
 - Runtime data
 - CPU utilization
 - GPU utilization
 - Temperature
 - Energy/Power
 - PAPI metrics
- “Third-person” measurements – not directly related to what the application / runtime is doing



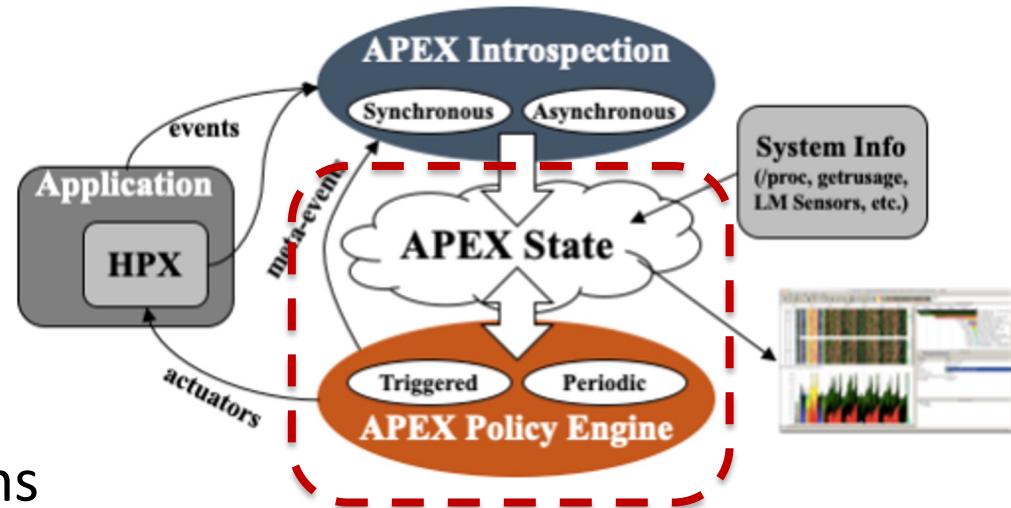
Performance Analysis

- Performance data is saved for post-mortem analysis
 - Profile data
 - Concurrency reports
 - Scatterplot data
 - Task dependency graph
 - Task dependency tree
 - OTF2 traces (Vampir)
 - Perfetto traces
- Any online analysis/policy *should* be done offline first



Policy Engine

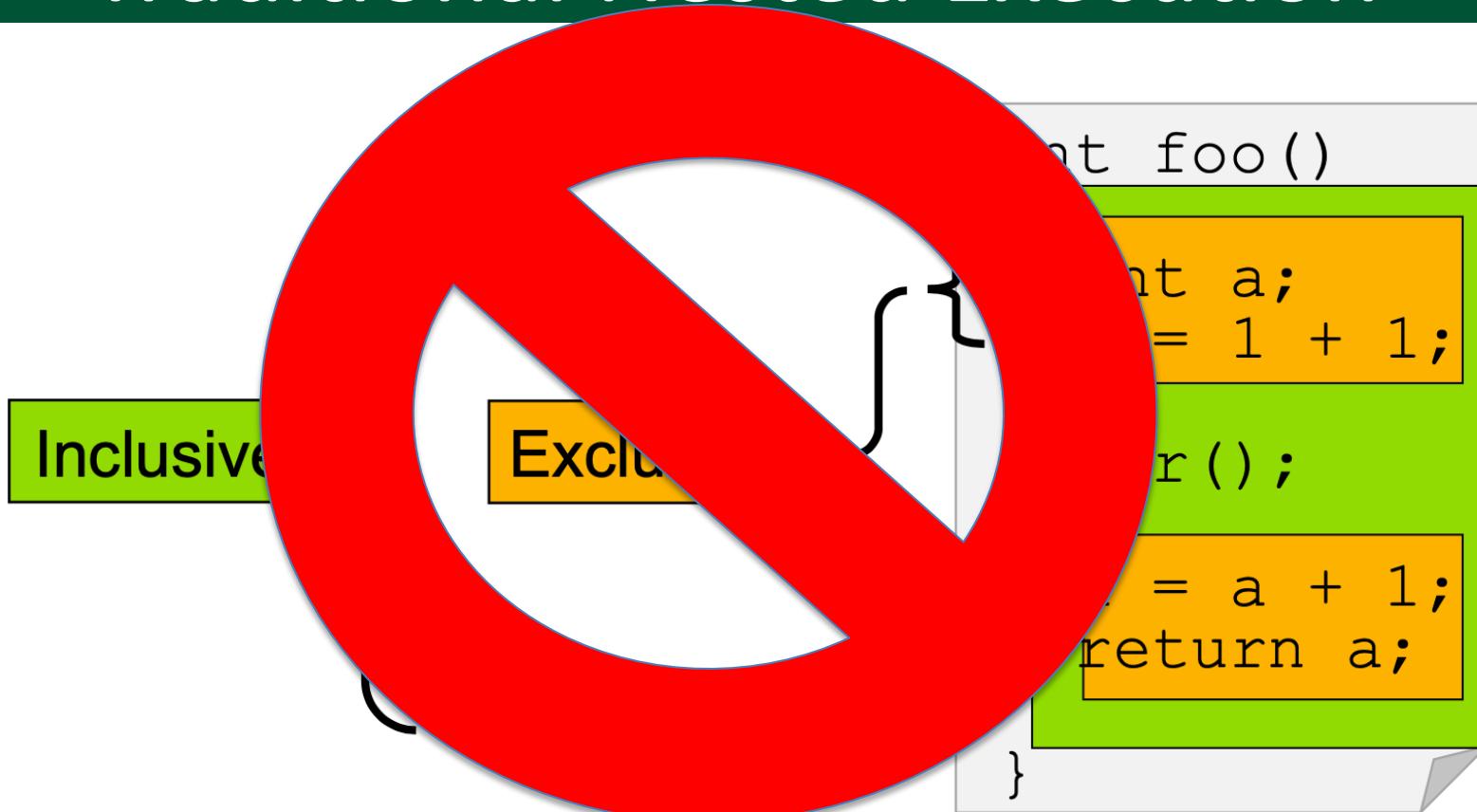
- Performance data is used to control parametric values
 - “Magic numbers”
 - Timeouts
 - Queue depths
 - Max worker threads
 - Tiling parameters
 - Colaescing parameters
 - When to switch algorithms
 - Power/energy management
- Computational steering based on performance state



Asynchronous Tasking Models

- Lots of examples – C++ std::async, OpenMP tasks, HPX, StarPU, Charm++, ...even CUDA/HIP/OneAPI
- ...With lots of complications:
 - How to track task dependencies?
 - What about “untied” tasks?
 - How to track dependencies across OS threads?
 - What about task migration?
 - What about yielding for dependencies?
 - What about futures? Continuations? ...

Traditional Nested Execution



Problem: Migrating Tasks, Asynchrony

- If the OS pre-empts a thread and migrates it to another core, that's fine for TAU, Score-P, OTF2, PAPI, etc.
- If the RUNTIME pre-empts a thread and migrates it to another OS thread – that's NOT OK.
- If a parent task creates a child task that outlives the parent task – that's NOT OK.
- If a parent task creates a child task that doesn't return control to the parent task – that's NOT OK.

APEX and HPX

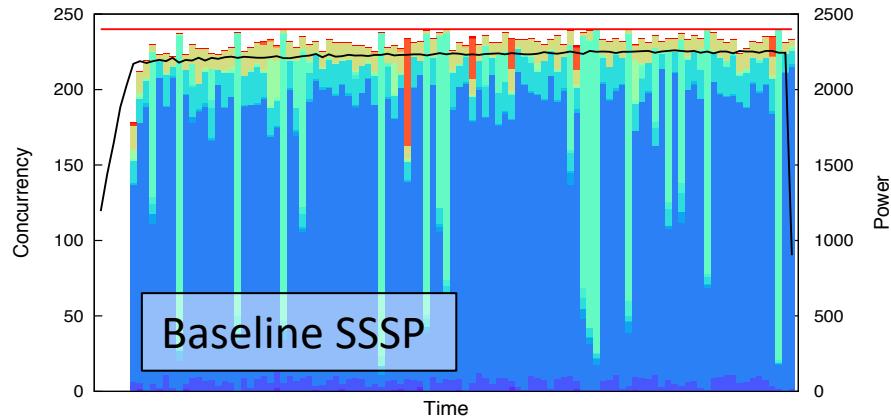
- HPX: Asynchronous Many-Task Runtime system in C++
- <https://hpx.stellar-group.org>
- Data and task dependencies can be expressed with HPX **futures** and **continuations**, chained together in an **execution graph**.
- The graph can be built asynchronously
- HPX tasks are created, scheduled, executed, 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



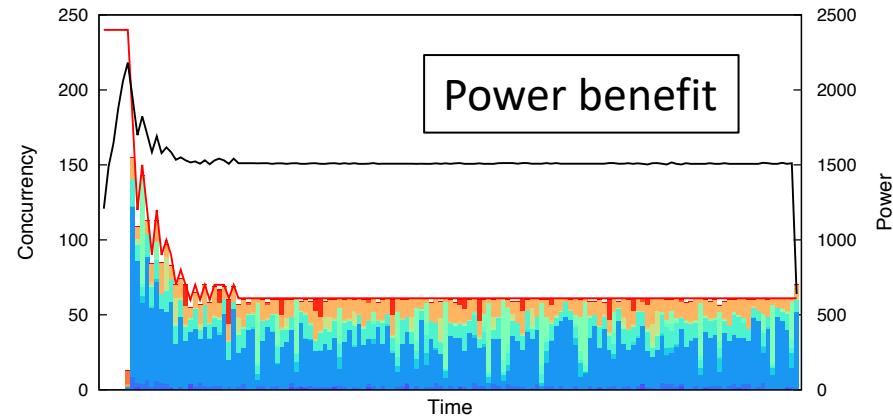
APEX and HPX

- APEX is integrated into the HPX thread scheduler, uniquely identifies each task with a GUID, and tracks all state transitions for a given task.
- HPX performance counters are also passed to APEX
- If HPX is configured with MPI support, the MPI functions are also captured (`MPI_Isend`, `MPI_Irecv`, `MPI_Wait*`)
- Policy Engine used for tuning heuristic control knobs in HPX thread scheduler, networking
 - Soft power caps, maximize throughput, reduce network latency, ...

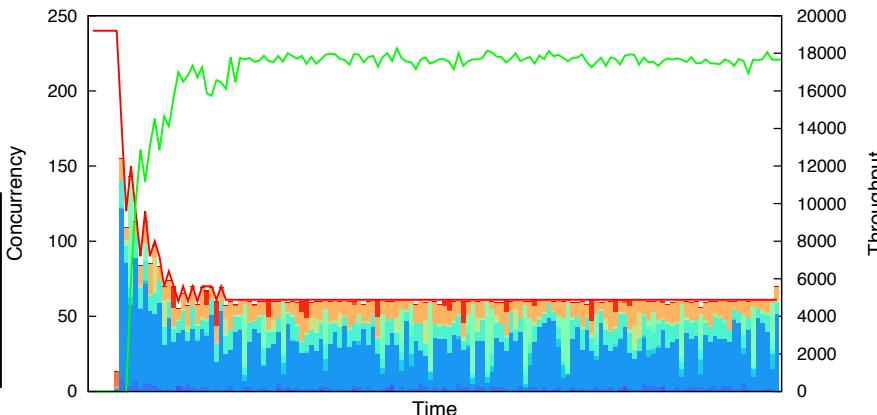
APEX and HPX



Baseline SSSP



Power benefit



Throttled to
maximize
throughput

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

Measurement Capabilities

- Timers – “per-process” aggregation for all timers
 - Start, stop, **yield, resume**
 - All times are *inclusive* (unless yielded/resumed)
- Counters
 - 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

See:

https://github.com/STELLAR-GROUP/hpx/blob/master/tests/unit/apex/apex_action_count.cpp

HPX Example – Fibonacci (5)

```
std::uint64_t fibonacci(std::uint64_t n);
HPX_PLAIN_ACTION(fibonacci, fibonacci_action)
std::uint64_t fibonacci(std::uint64_t n) {
    if (n < 2)  return n;
    // execute the Fibonacci function locally.
    hpx::id_type const locality_id = hpx::find_here();
    fibonacci_action fib;
    hpx::future<std::uint64_t> n1 = hpx::async(fib, locality_id, n - 1);
    hpx::future<std::uint64_t> n2 = hpx::async(fib, locality_id, n - 2);
    // wait for the Futures to return their values
    return n1.get() + n2.get();
}

int hpx_main(hpx::program_options::variables_map& vm) {
    // extract command line argument, i.e. fib(N)
    std::uint64_t n = vm["n-value"].as<std::uint64_t>();
    {
        // Keep track of the time required to execute.
        hpx::chrono::high_resolution_timer t;
        // Wait for fib() to return the value
        fibonacci_action fib;
        std::uint64_t r = fib(hpx::find_here(), n);
        char const* fmt =
            "fibonacci({1}) == {2}\nelapsed time: {3} [s]\n";
        hpx::util::format_to(std::cout, fmt, n, r, t.elapsed());
    }
    return hpx::finalize();    // Handles HPX shutdown
}

int main(int argc, char* argv[])
{
    // Set some APEX options
    apex::apex_options::use_screen_output(true);
    // Configure application-specific options
    hpx::program_options::options_description
desc_commandline(
        "Usage: " HPX_APPLICATION_STRING " [options]");
    desc_commandline.add_options()("n-value",
        hpx::program_options::value<std::uint64_t>()-
>default_value(10),
        "n value for the Fibonacci function");

    // Initialize and run HPX
    hpx::init_params init_args;
    init_args.desc_cmdline = desc_commandline;

    int status = hpx::init(argc, argv, init_args);

    return hpx::util::report_errors();
}

apex_exec --apex:tasktree ./bin/apex_action_count_test \
--hpx:threads=8 --n-value 5
```

HPX Example – screen output

```
Elapsed time: 0.00760714 seconds
Total processes detected: 1
HW Threads detected on rank 0: 96
Worker Threads observed on rank 0: 8
Available CPU time on rank 0: 0.0608571 seconds
Available CPU time on all ranks: 0.0608571 seconds
```

Counter	:	#samp		mean		max
status:Threads	:	1	9.00	9.00		
status:VmHWM kB	:	1	1.92e+04	1.92e+04		
status:VmRSS kB	:	1	1.92e+04	1.92e+04		
status:nonvoluntary_ctxt_switches	:	1	97.00	97.00		
status:voluntary_ctxt_switches	:	1	29.00	29.00		

CPU Timers	:	#calls		mean		total
APEX MAIN	:	1	0.01	0.01		
load_components_action	:	1	0.00	0.00		
run_helper	:	1	0.00	0.00		
fibonacci_action	:	14	0.00	0.00		
shutdown_all_action	:	1	0.00	0.00		
call_startup_functions_action	:	2	0.00	0.00		

Counters
sampled from
OS/Hardware

HPX events/tasks

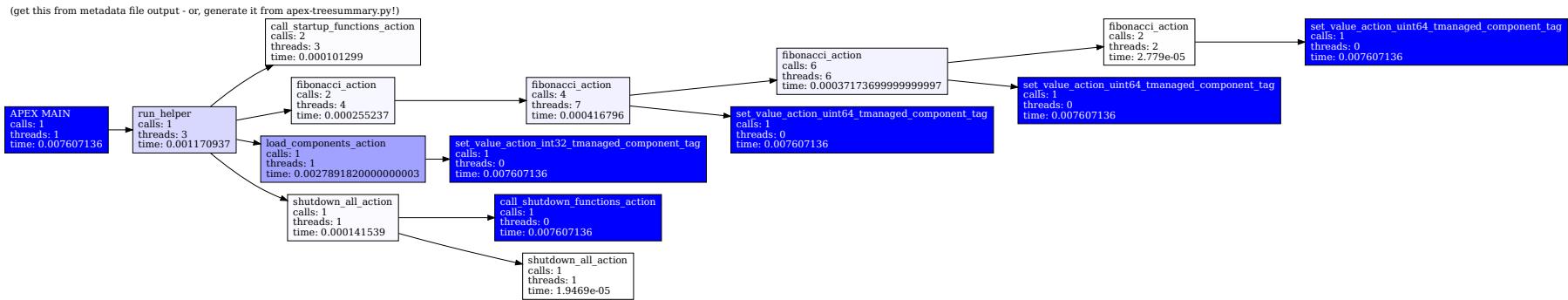
HPX Example – task tree (ASCII)

```
[khuck@gilgamesh build]$ /home/users/khuck/src/xpress-apex/install_gilgamesh_5.2.0/bin/apex-treesummary.py --dot --ascii
Reading tasktree...
Read 15 rows
Found 0 ranks, with max graph node index of 14 and depth of 6
building common tree...
Rank 0 ...
1-> 0.008 - 100.000% [1] {min=0.008, max=0.008, mean=0.008, threads=1} APEX MAIN
1 |-> 0.001 - 15.393% [1] {min=0.001, max=0.001, mean=0.001, threads=3} run_helper
1 | |-> 0.003 - 36.665% [1] {min=0.003, max=0.003, mean=0.003, threads=1} load_components_action
1 | | |-> 0.008 - 100.000% [1] {min=0.008, max=0.008, mean=0.008, threads=0} set_value_action_int32_tmanaged_component_tag
1 | | |-> 0.000 - 3.355% [2] {min=0.000, max=0.000, mean=0.000, threads=4} fibonacci_action
1 | | | |-> 0.000 - 5.479% [4] {min=0.000, max=0.000, mean=0.000, threads=7} fibonacci_action
1 | | | | |-> 0.008 - 100.000% [1] {min=0.008, max=0.008, mean=0.008, threads=0} set_value_action_uint64_tmanaged_component_tag
1 | | | | | |-> 0.000 - 4.887% [6] {min=0.000, max=0.000, mean=0.000, threads=6} fibonacci_action
1 | | | | | | |-> 0.008 - 100.000% [1] {min=0.008, max=0.008, mean=0.008, threads=0} set_value_action_uint64_tmanaged_component_tag
1 | | | | | | | |-> 0.000 - 0.365% [2] {min=0.000, max=0.000, mean=0.000, threads=2} fibonacci_action
1 | | | | | | | | |-> 0.008 - 100.000% [1] {min=0.008, max=0.008, mean=0.008, threads=0} set_value_action_uint64_tmanaged_component_tag
1 | | | | | | | | | |-> 0.000 - 1.861% [1] {min=0.000, max=0.000, mean=0.000, threads=1} shutdown_all_action
1 | | | | | | | | | | |-> 0.008 - 100.000% [1] {min=0.008, max=0.008, mean=0.008, threads=0} call_shutdown_functions_action
1 | | | | | | | | | | | |-> 0.000 - 0.256% [1] {min=0.000, max=0.000, mean=0.000, threads=1} shutdown_all_action
1 | | | | | | | | | | | | |-> 0.000 - 1.332% [2] {min=0.000, max=0.000, mean=0.000, threads=3} call_startup_functions_action
16 total graph nodes

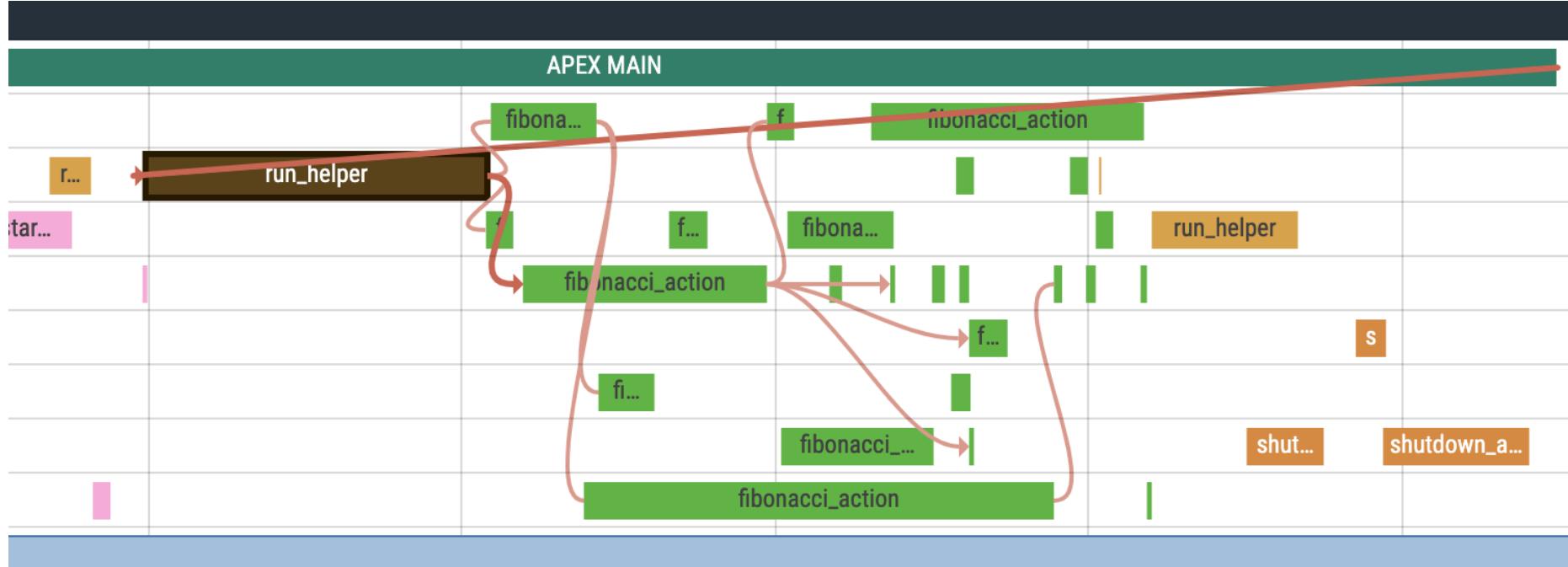
Task tree also written to tasktree.txt.
Computing new stats...
Building dot file
done.
```



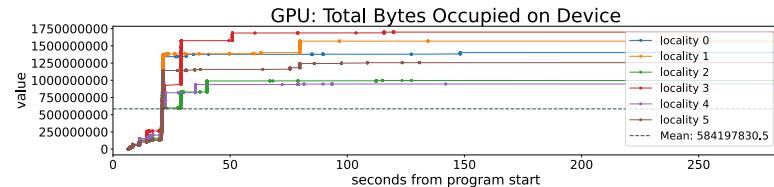
HPX Example – tasktree (DOT)



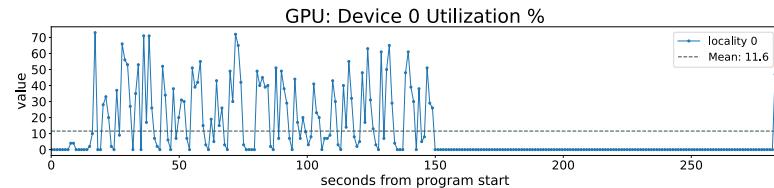
HPX Example – Perfetto trace



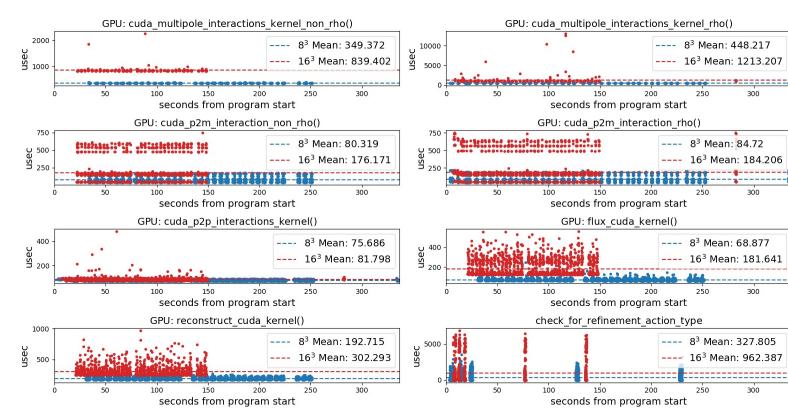
APEX example – Octo-Tiger (HPX)



Tracking GPU memory usage with CUPTI

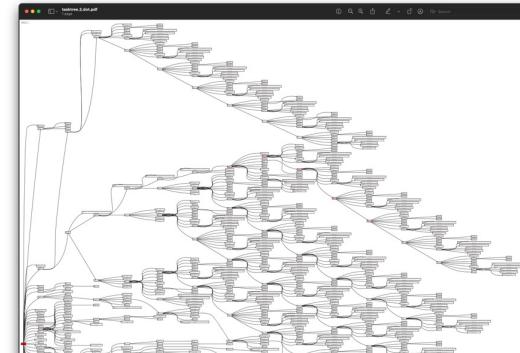


Monitoring GPU utilization with NVML library

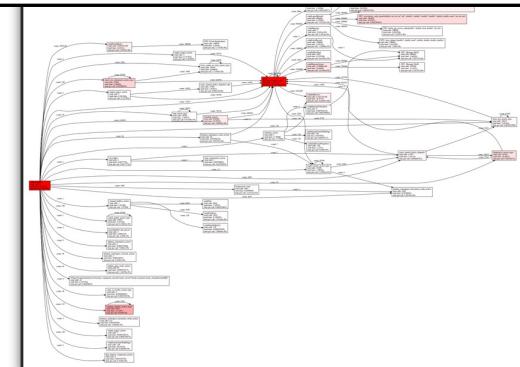
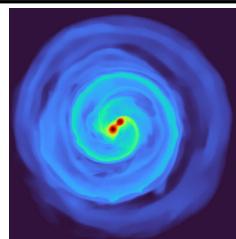


Comparing subgrid sizes and relative kernel performance with CUPTI device activity

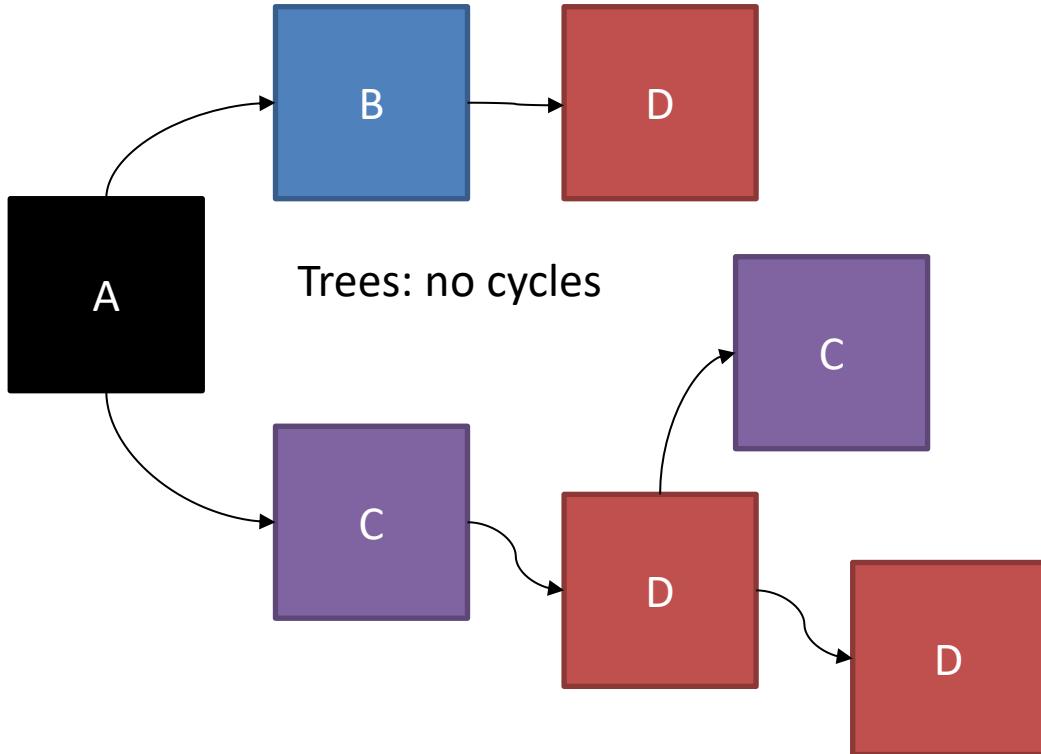
<https://github.com/STEIIAR-GROUP/octotiger>
<https://github.com/STEIIAR-GROUP/hpx>



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

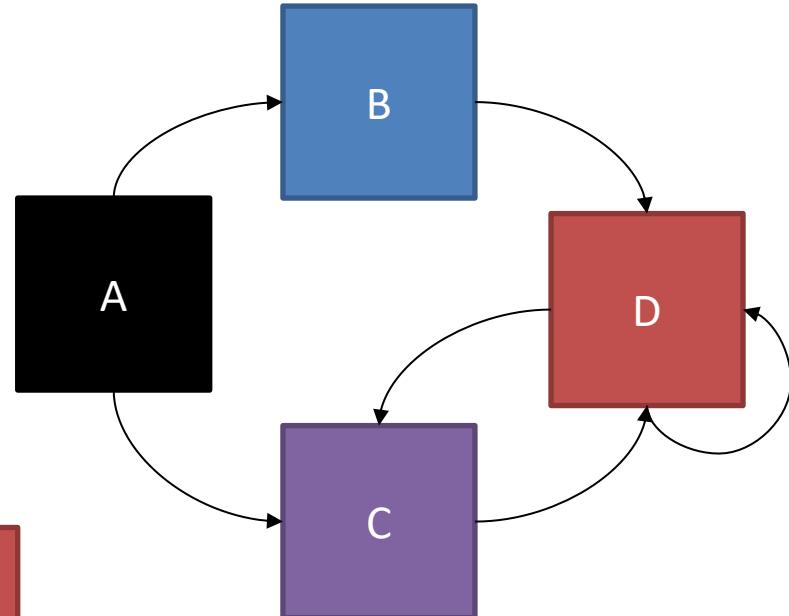


Task Dependencies: Trees & Graphs



Trees: no cycles

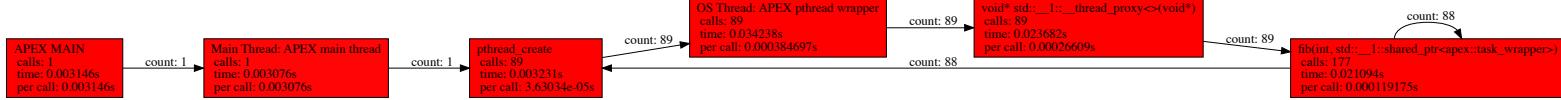
Graphs: cycles, each node appears only once



Task Dependencies: C++ 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



```
int fib (int in, std::shared_ptr<apex::task_wrapper> parent) {
    apex::scoped_thread st("fib thread");
    apex::scoped_timer ast((uint64_t)&fib,
        apex::apex_options::top_level_os_threads() ? nullptr : parent);
    if (in == 0) { return 0; }
    else if (in == 1) { return 1; }
    int a = in-1;
    int b = in-2;
    auto future_a = std::async(std::launch::async, fib, a,
        ast.get_task_wrapper());
    //ast.yield();
    int result_b = fib(b, ast.get_task_wrapper());
    int result_a = future_a.get();
    //ast.start();
    int result = result_a + result_b;
    return result;
}
```

```
int main(int argc, char *argv[]) {
    apex::init(argv[0], 0, 1);
    int i = 10;

    if (argc != 2) {
        std::cerr << "usage: " << argv[0] << " <integer value>" << std::endl;
        std::cerr << "Using default value of 10" << std::endl;
    } else {
        i = atol(argv[1]);
    }

    if (i < 1) {
        std::cerr << i << " must be >= 1" << std::endl;
        return -1;
    }

    std::shared_ptr<apex::task_wrapper> parent = nullptr;
    auto future = std::async(fib, i, parent);
    int result = future.get();
    std::cout << "fib of " << i << " is " << result
        << " (valid value: " << fib_results[i] << ")" << std::endl;
    apex::finalize();
    return 0;
}
```

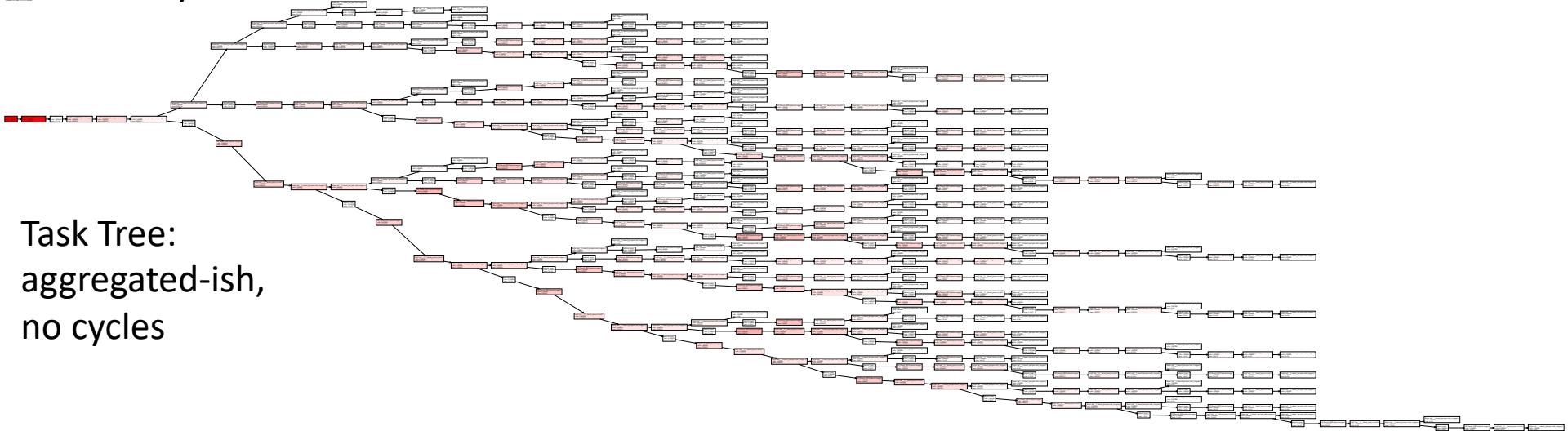
See: src/C++/std_thread/apex_fibonacci.cpp

Task Dependencies: fib(10)

Task Graph:
aggregated,
contains cycles



Task Tree:
aggregated-ish,
no cycles

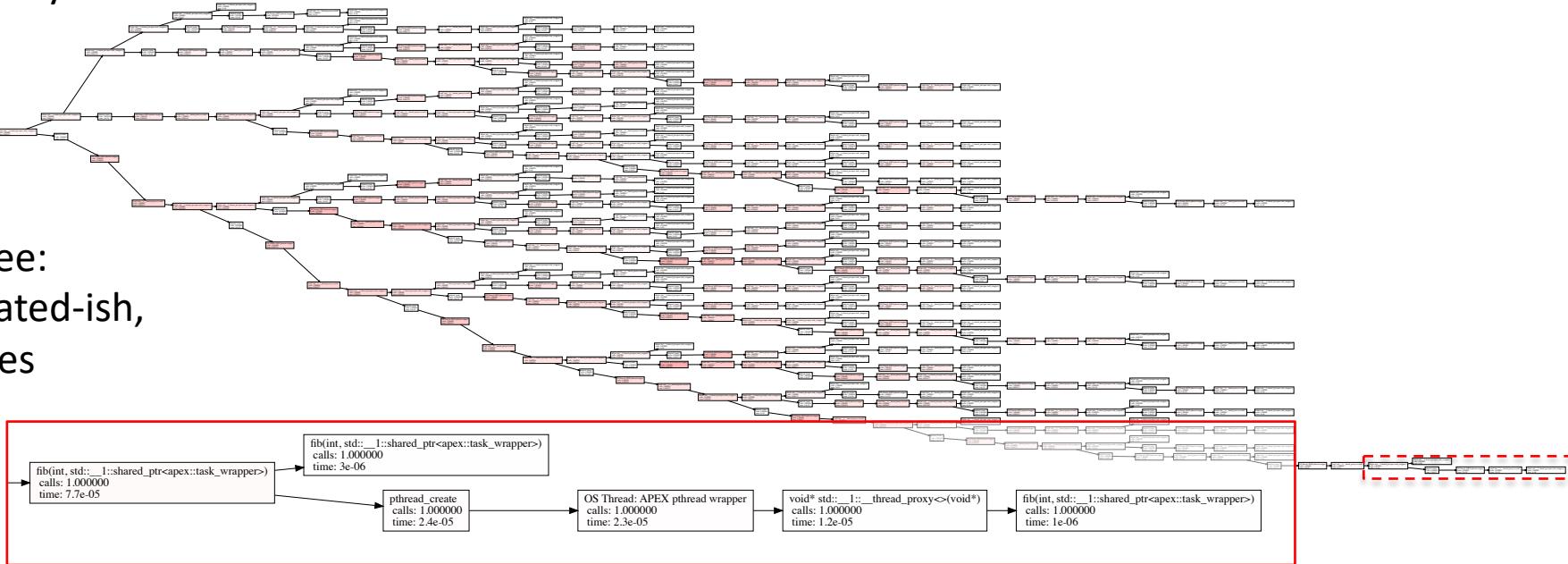


Task Dependencies: fib(10)

Task Graph:
aggregated,
contains cycles

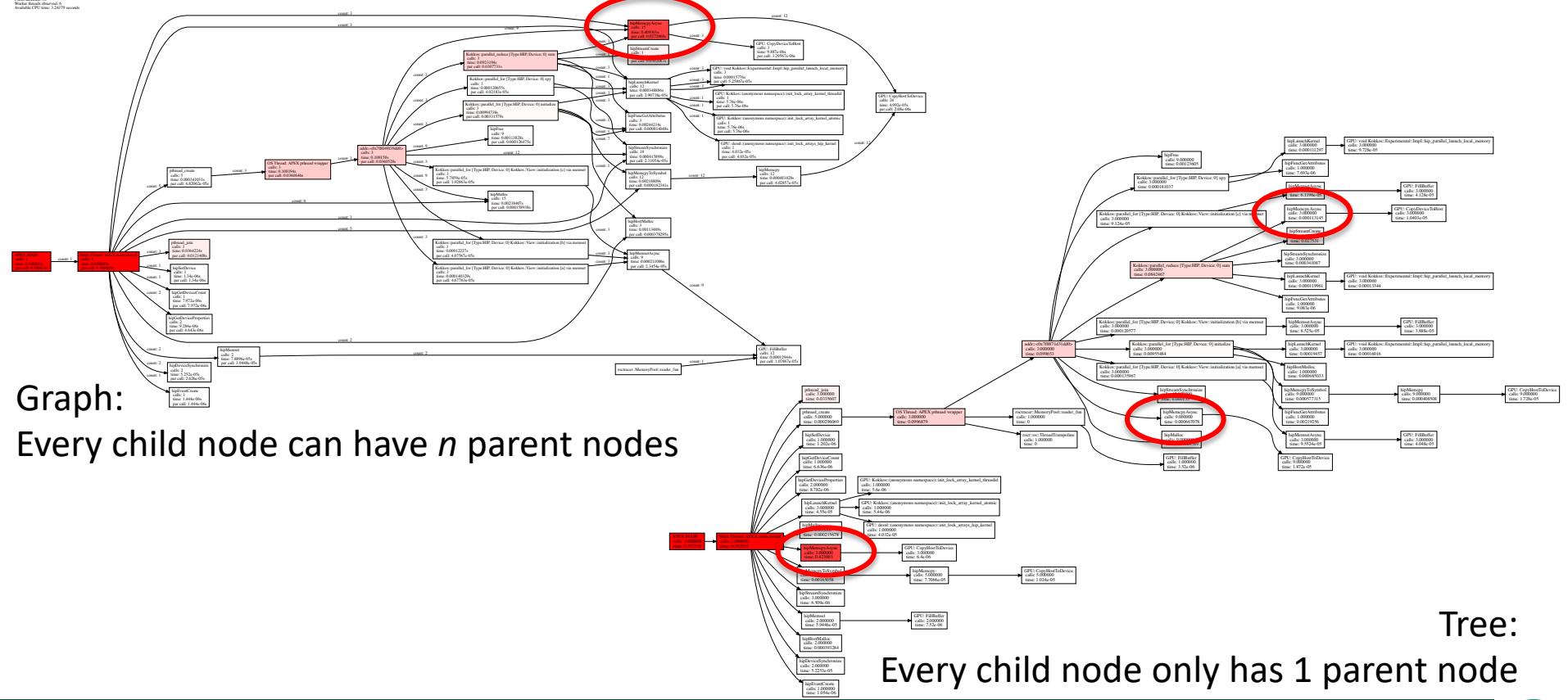


Task Tree:
aggregated-ish,
no cycles



Better Example

Elapsed Time: 0.00001 seconds
Core duration: 0 seconds
Warp duration: 0 seconds
Available CPU time: 3.24779 seconds



Hardware Counters

- APEX is integrated with PAPI (Performance Application Programming Interface) <https://icl.utk.edu/papi/>
- Portable access to native hardware counters
 - CPU
 - GPU
 - Off-core (permission dependent)
 - Node/OS health
 - Power/energy
 - Filesystems
- HW counters collected with timers, or periodically

PAPI timer example

```
APEX_SCREEN_OUTPUT_DETAIL=1 APEX_PAPI_METRICS="PAPI TOT INS PAPI BR MSP"  
srun - ./apex/install/bin/apex_exec --apex:cuda build/bin/multiGpuThread_cu
```

CPU Timers								
#thread	mean	total	tot/thr	%total	%wall	TOT_INS	BR_MSP	: #calls #yields
<hr/>								
<hr/>								
				APEX MAIN :	1		0	
1	0.13	0.13	0.13	100.0	100.0	7.52e+07	1.47e+05	
				int apex_reload_main(int, char**, char**)	:	1	0	
1	0.13	0.13	0.13	19.6	98.2	7.04e+07	1.36e+05	
				cudaMalloc :	2		0	
1	0.05	0.09	0.09	13.9	69.3	6.32e+07	1.04e+05	
				cudaMemcpy :	80		0	
4	0.00	0.01	0.00	0.9	1.2	5.01e+06	2.98e+04	
				cudaLaunchKernel :	40		0	
4	0.00	0.00	0.00	0.2	0.2	1.01e+06	1.69e+04	

PAPI periodic example

```
APEX_PAPI_COMPONENTS="perf perf_event"
APEX_PAPI_COMPONENT_METRICS="perf::INSTRUCTIONS:u=0" srun
./apex/install/bin/apex_exec --apex:cuda build/bin/multiGpuThread_cu --
apex:period 10000
```

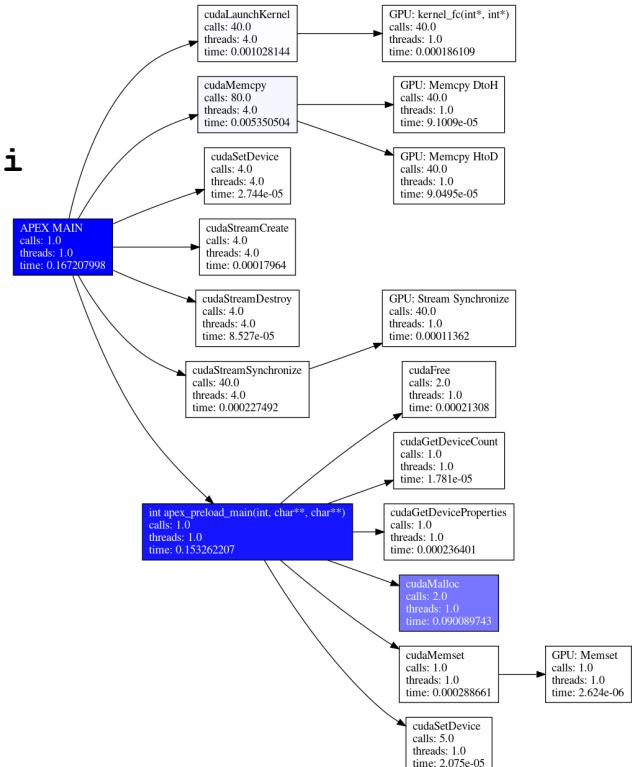
Counter	:	#samp		mean		max
<hr/>						
	CPU Guest % :	20		0.00		0.00
	CPU I/O Wait % :	20		0.00		0.00
	CPU IRQ % :	20		0.02		0.37
	CPU Idle % :	20		99.73		100.00
	CPU Nice % :	20		0.00		0.00
	CPU Steal % :	20		0.00		0.00
	CPU System % :	20		0.21		0.73
	CPU User % :	20		0.04		0.37
	CPU soft IRQ % :	20		0.01		0.16
...(they should show up here. Didn't work on mahti ☹)						

CUDA Measurement

- Support provided with NVIDIA **CUPTI** library
- Monitoring support provided with NVIDIA **NVML** library
- Hardware counters provided from **CUPTI**, indirectly through **PAPI**
- Host callback and device activity dependencies linked **using correlation IDs**
- Some kernel counters optionally captured (register usage, occupancy, block sizes, grid sizes, ...)

CUDA example

```
[kehuck1@mahti-login11 apex-tutorial]$ srun  
./apex/install/bin/apex_exec --apex:cuda  
build/bin/multiGpuThread_cu --apex:monitor_gpu --  
apex:tasktree  
  
[kehuck1@mahti-login11 apex-tutorial]$  
$HOME/src/apex/install/bin/apex-treesummary.py --ascii  
--dot  
  
Reading tasktree...  
Read 19 rows  
  
Found 0 ranks, with max graph node index of 18 and  
depth of 3  
  
building common tree...  
  
Rank 0 ...  
  
...  
  
Task tree also written to tasktree.txt.  
  
Computing new stats...  
  
Building dot file  
  
done.  
  
[kehuck1@mahti-login11 apex-tutorial]$ dot -Tpng -O  
tasktree.dot
```

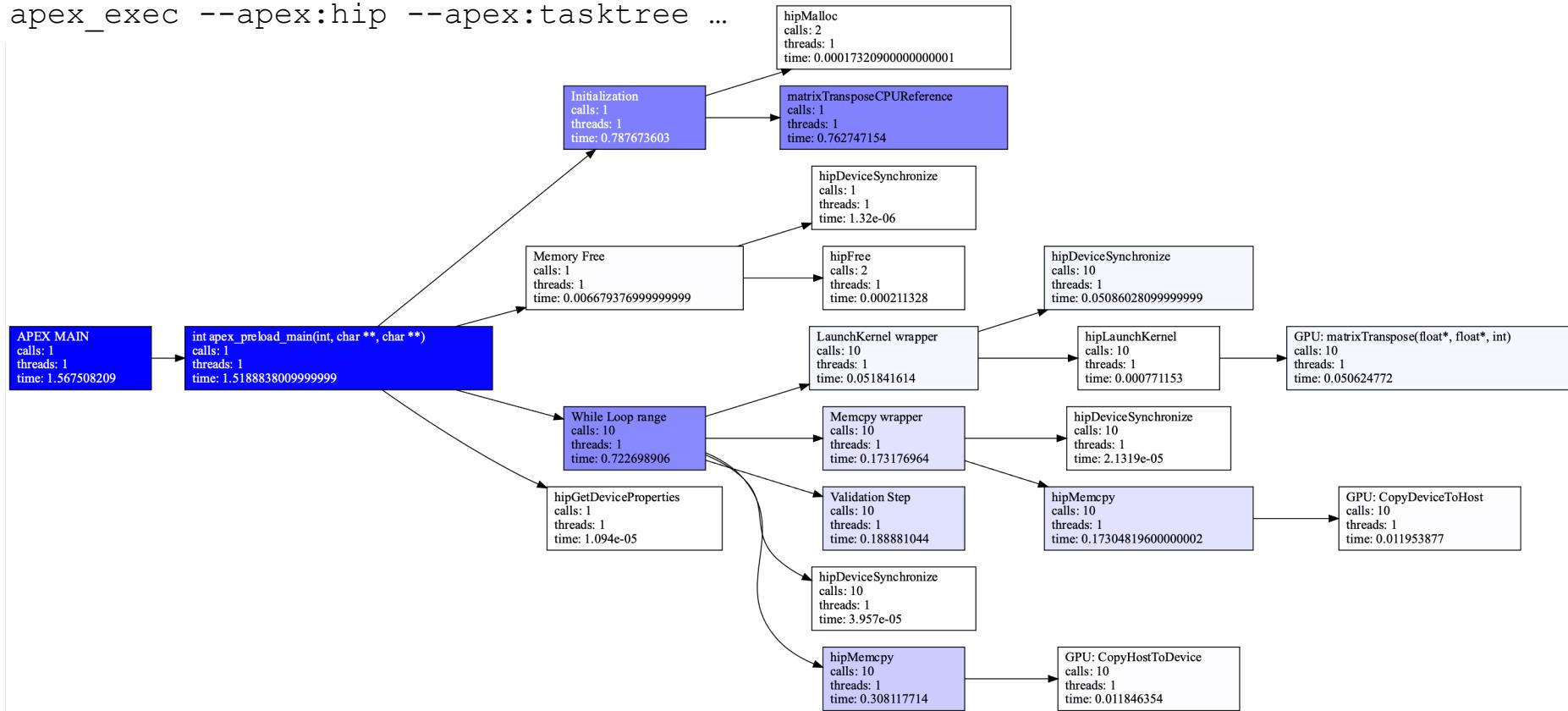


HIP/ROCm Measurement

- Support provided by AMD **Rotracer** and **Rocprofiler** libraries (deprecated, soon to be replaced)
- Monitoring support provided by AMD **rocm-smi** library
- Hardware counters provided through **PAPI**
- Host callback and device activity dependencies linked using **correlation IDs**
- Some kernel counters optionally captured (block sizes, grid sizes)

HIP example

```
apex_exec --apex:hip --apex:tasktree ...
```



GPU Monitoring

Adding --apex:monitor_gpu will enable NVML or ROCM-SMI support

Counter	:	#samp		mean		max
GPU: Device 0 Clock Memory (MHz)	:	2	1215.00	1215.00		
GPU: Device 0 Clock SM (MHz)	:	2	210.00	210.00		
GPU: Device 0 Memory Free (GB)	:	2	42.47	42.48		
GPU: Device 0 Memory Total (GB)	:	1	42.95	42.95		
GPU: Device 0 Memory Used (GB)	:	2	0.48	0.48		
GPU: Device 0 Memory Utilization %	:	2	0.00	0.00		
GPU: Device 0 NvLink Link Count	:	1	12.00	12.00		
GPU: Device 0 NvLink Speed (GB/s)	:	1	25.00	25.00		
GPU: Device 0 NvLink Throughput Data RX	:	2	1.06e+11	1.06e+11		
GPU: Device 0 NvLink Throughput Data TX	:	2	1.04e+11	1.04e+11		
GPU: Device 0 NvLink Throughput Raw RX	:	2	1.68e+11	1.68e+11		
GPU: Device 0 NvLink Throughput Raw TX	:	2	1.67e+11	1.67e+11		
GPU: Device 0 PCIe RX Throughput (MB/s)	:	2	76.50	137.00		
GPU: Device 0 PCIe TX Throughput (MB/s)	:	2	2.50	5.00		
GPU: Device 0 Power (W)	:	2	55.87	57.49		
GPU: Device 0 Temperature (C)	:	2	37.00	37.00		
GPU: Device 0 Utilization %	:	2	0.00	0.00		

...

OneAPI Measurement

- Support provided with Intel **Level0** library
- Monitoring support not yet provided
- Hardware counters not yet provided
- Host callback and device activity dependencies linked using **correlation IDs** – generated by APEX
- Very early stages of support

GPU Memory Tracking

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

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

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

cuda-memcheck

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

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.



(b) Distributed G_t^d method with sub-ring size of three.

Example Program: memory error

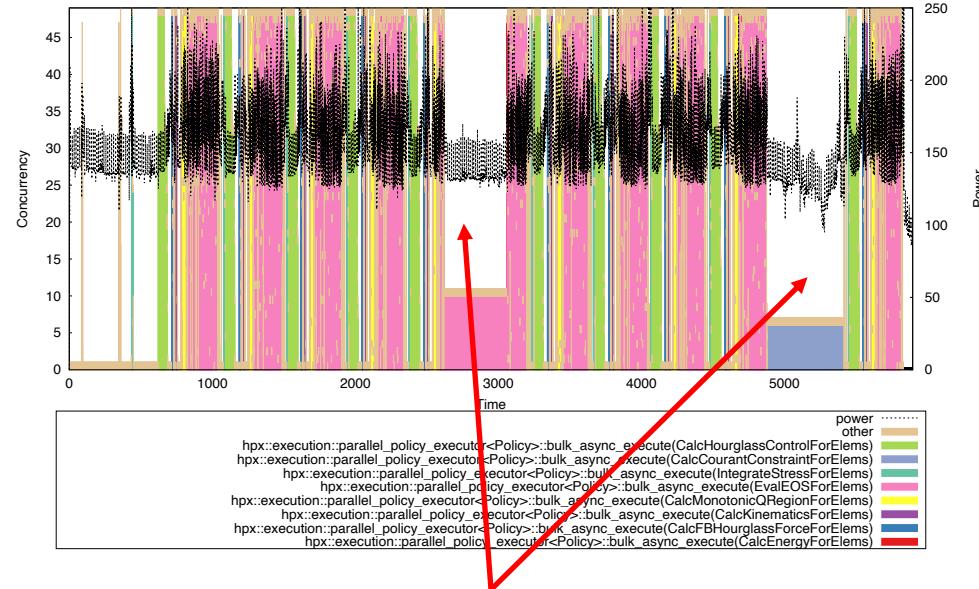
```
apex_exec --apex:gpu_memory ...
```

```
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}]
```

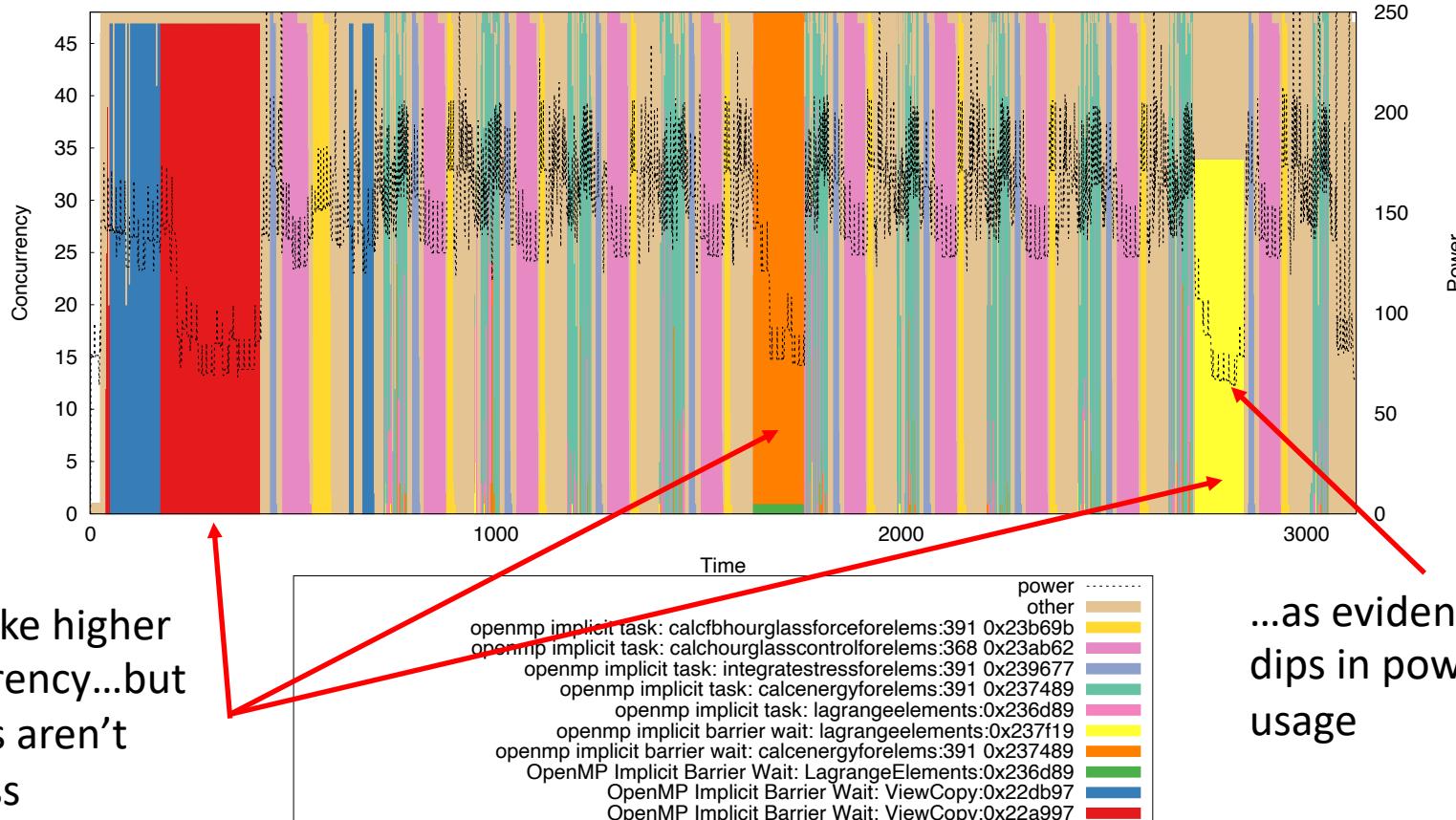
Concurrency Tracking



Helps identify regions of low concurrency

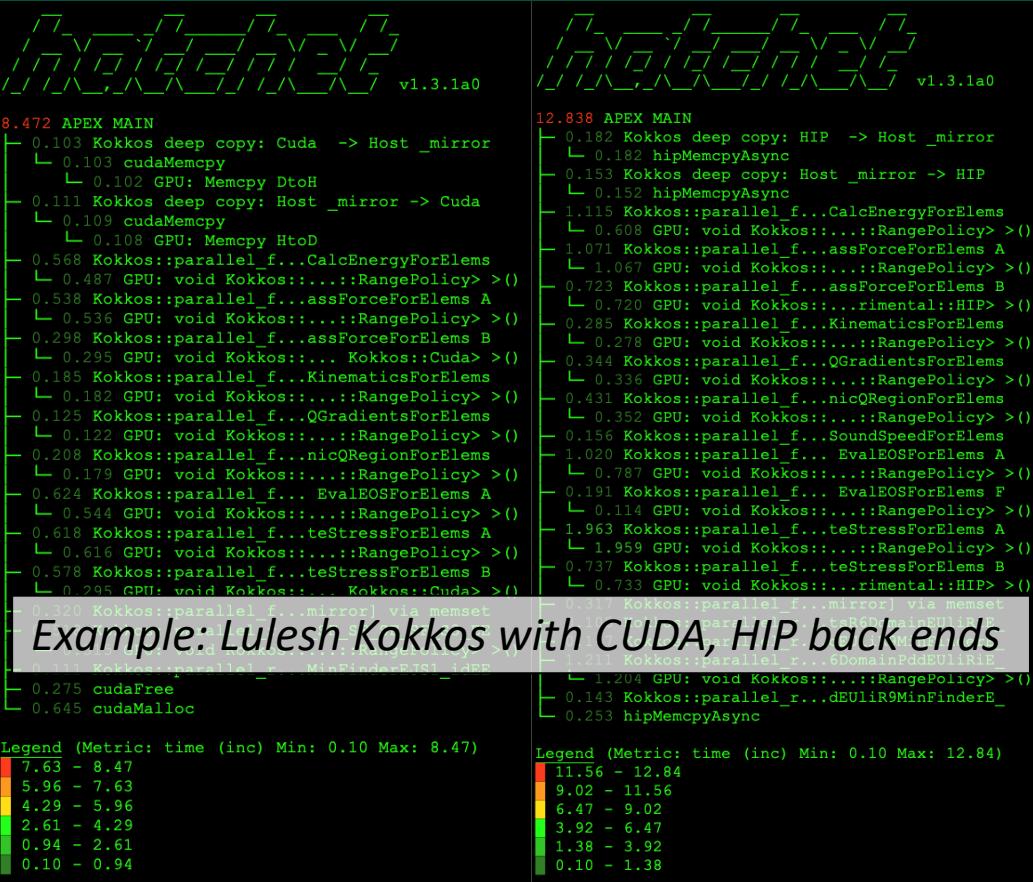
- Periodically sample all the currently executing tasks
- Aggregate across N timers
- Example shown:
 - Kokkos Lulesh with HPX back end
 - Sampled 200 times per second
 - 10 iterations, size 256

Concurrency: OpenMP back end



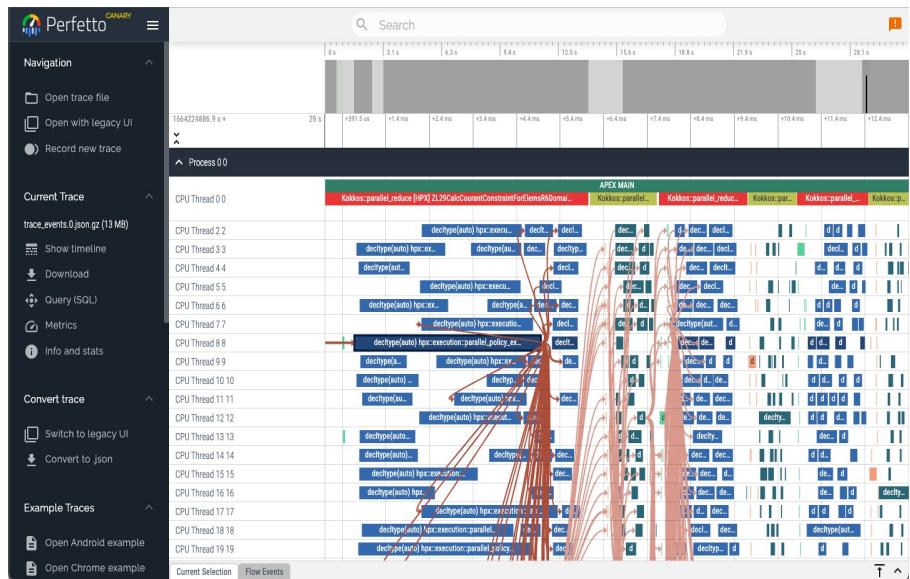
Profile Formats

- 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)~~
 - ... these are now replaced by single CSV file that is post-processed.



Trace Formats

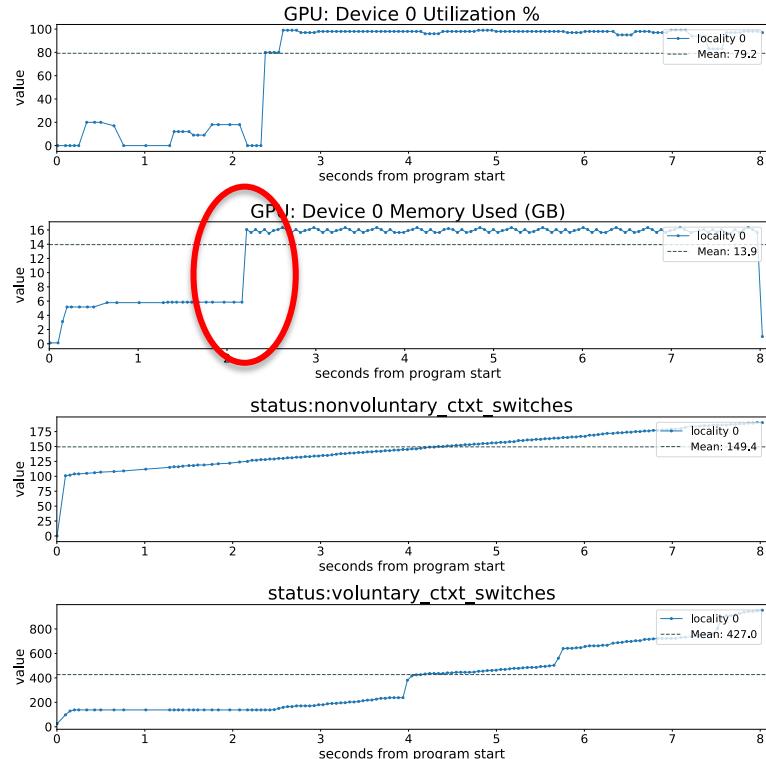
- OTF2 up to v 2.3
 - 3.0 has API changes, haven't been incorporated yet
 - Visualized with Vampir or Traveler / JetLag
- Google Trace Events Format
 - JSON support
 - Perfetto native support
 - Visualized with Perfetto
 - Some scaling issues (memory limit of web browser)



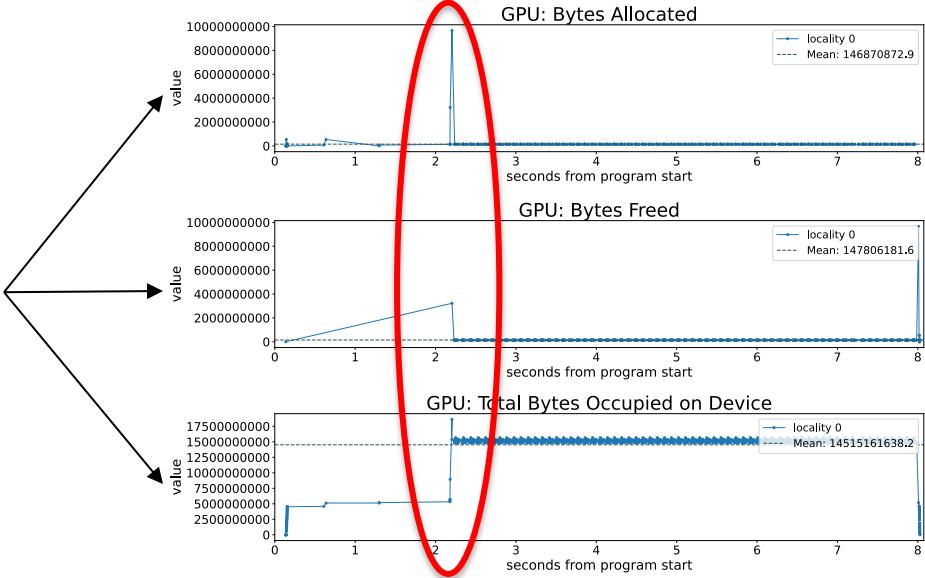
Example: Lulesh Kokkos with HPX back end

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, OneAPI, OpenMP target
- Abstractions: Kokkos (SNL), Raja (LLNL)
- MPI (subset)
- In development / experimental: StarPU (?), Iris (ORNL)

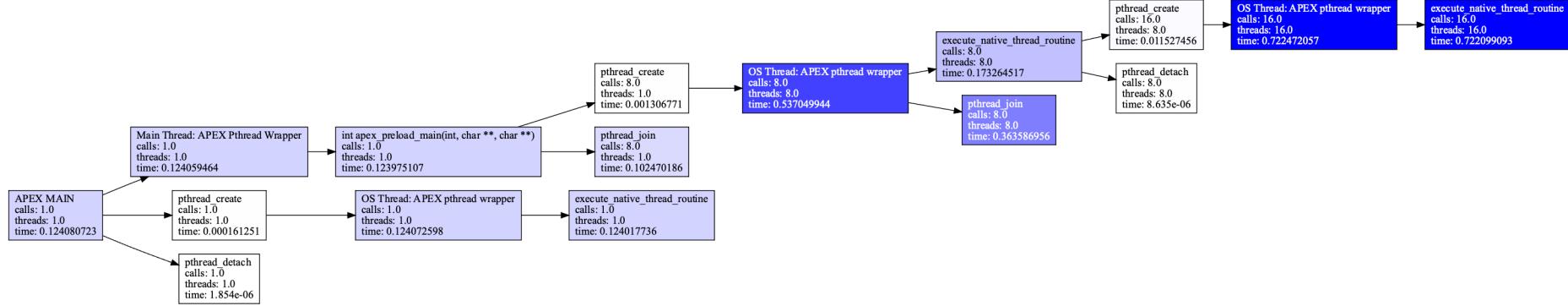
POSIX / C++ Threads

- 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`
 - Detached threads



Thread example

(get this from metadata file output - or, generate it from apex-treesummary.py!)



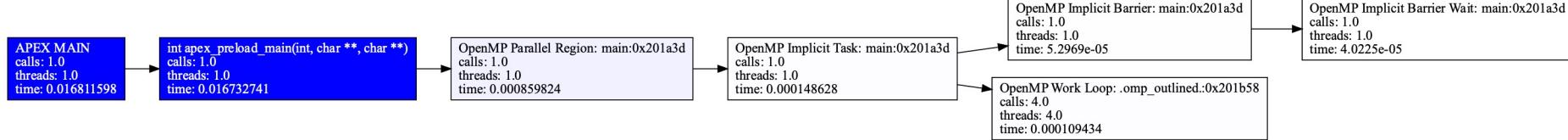
OpenMP and OpenACC

- 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



OpenMP example

(get this from metadata file output - or, generate it from apex-treesummary.py!)



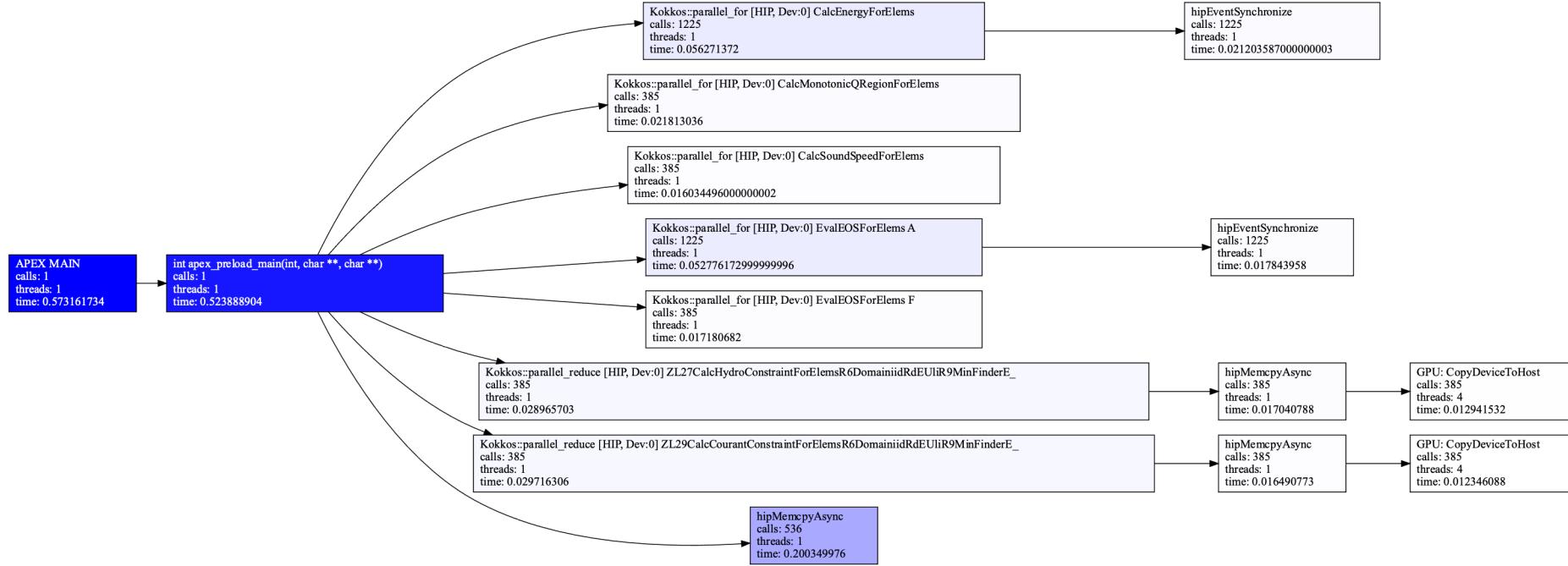
Kokkos and Raja

- 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 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



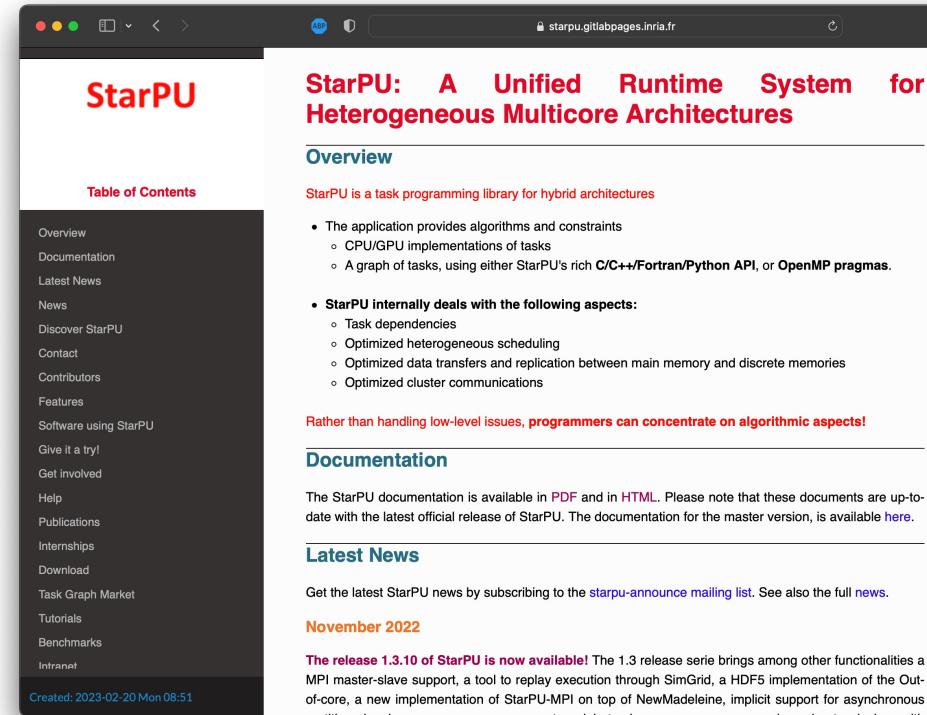
Kokkos Example

(get this from metadata file output - or, generate it from apex-treesummary.py!)



StarPU: A Unified Runtime System for Heterogeneous Multicore Architectures

<https://starpu.gitlabpages.inria.fr>



The screenshot shows the homepage of the StarPU website. The header features the word "StarPU" in large white letters. Below the header is a navigation bar with links for Overview, Documentation, Latest News, News, Discover StarPU, Contact, Contributors, Features, Software using StarPU, Give it a try!, Get involved, Help, Publications, Internships, Download, Task Graph Market, Tutorials, Benchmarks, and Intranet. The main content area has a red background and displays the following text:

StarPU: A Unified Runtime System for Heterogeneous Multicore Architectures

Overview

StarPU is a task programming library for hybrid architectures

- The application provides algorithms and constraints
 - CPU/GPU implementations of tasks
 - A graph of tasks, using either StarPU's rich C/C++/Fortran/Python API, or OpenMP pragmas.
- StarPU internally deals with the following aspects:
 - Task dependencies
 - Optimized heterogeneous scheduling
 - Optimized data transfers and replication between main memory and discrete memories
 - Optimized cluster communications

Rather than handling low-level issues, programmers can concentrate on algorithmic aspects!

Documentation

The StarPU documentation is available in [PDF](#) and in [HTML](#). Please note that these documents are up-to-date with the latest official release of StarPU. The documentation for the master version, is available [here](#).

Latest News

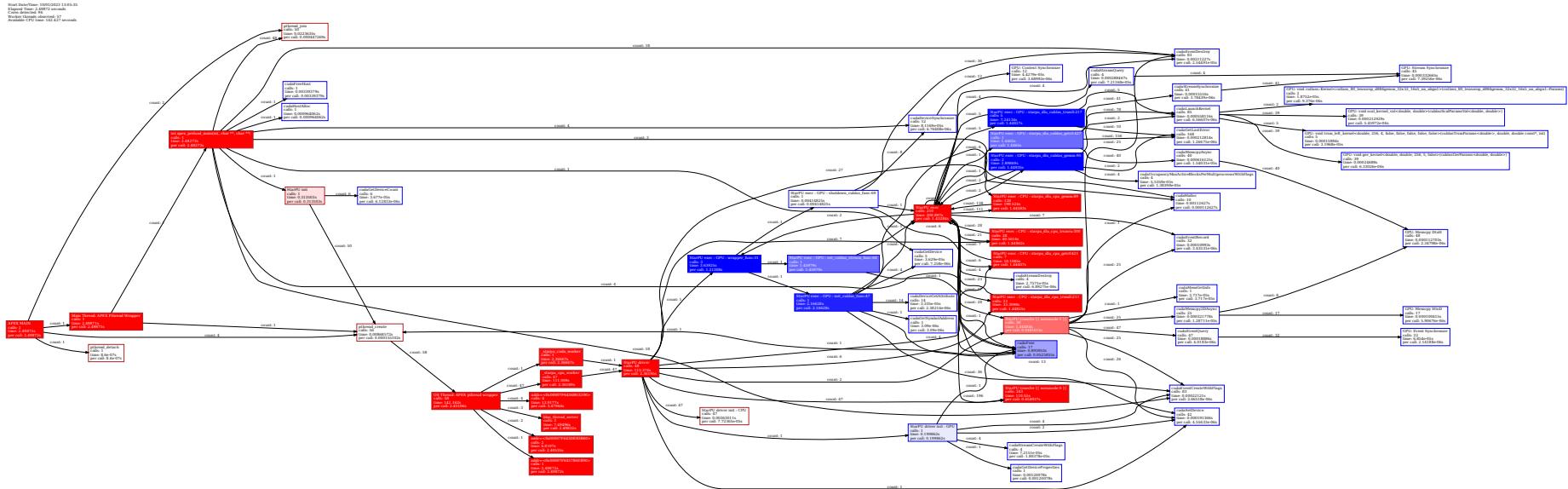
Get the latest StarPU news by subscribing to the [starpu-announce mailing list](#). See also the full [news](#).

November 2022

The release 1.3.10 of StarPU is now available! The 1.3 release serie brings among other functionalities a MPI master-slave support, a tool to replay execution through SimGrid, a HDF5 implementation of the Out-of-core, a new implementation of StarPU-MPI on top of NewMadeleine, implicit support for asynchronous

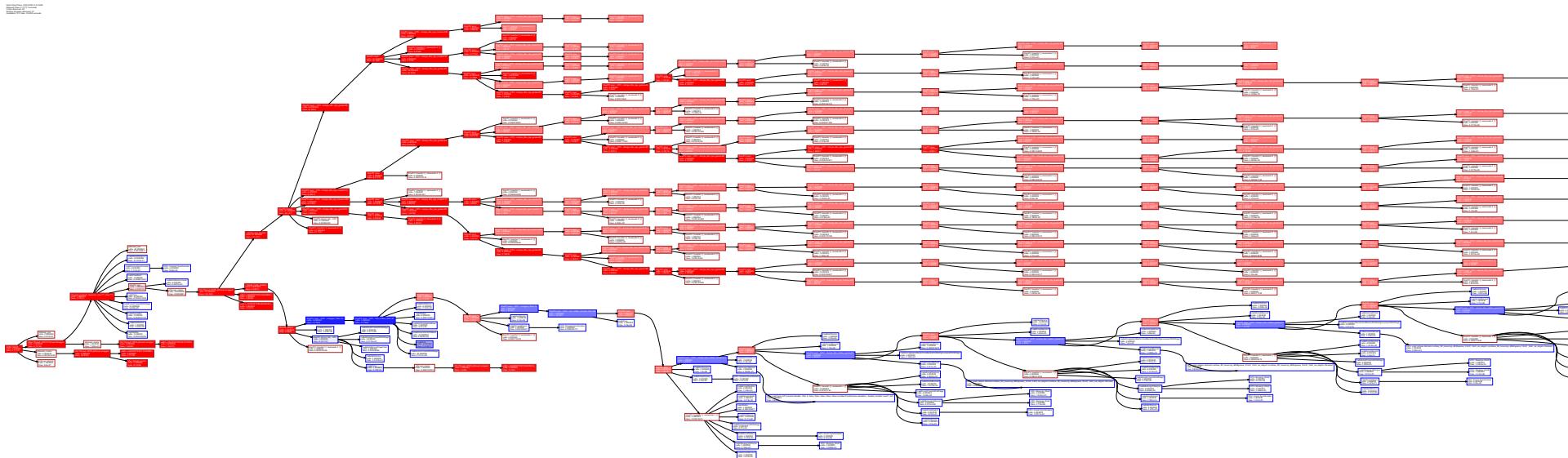
StarPU example: LU Factorization

Red: CPU, Blue: GPU



StarPU: LU Task Tree

Red: CPU, Blue: GPU



Lulesh w/ Kokkos Experiments

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

- HPX
- OpenMP
- CUDA
- HIP

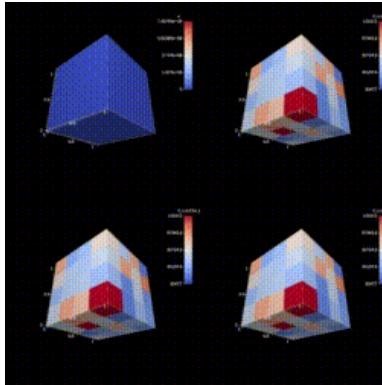


Figure: [LULESH 2.0.3](#) executed with 64 MPI ranks, measured by TAU. Time spent in main loop rendered every 100 timesteps by [Alpine-Ascent](#). Clockwise from upper left: Computed energy, accumulated time in main loop, time in main loop during last 100 timesteps, Δ time in main loop from previous 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.

The GitHub repository page for `kokkos/kokkos-miniapps` shows the following details:

- Code**: master branch, 1 branch, 0 tags
- Issues**: 1
- Pull requests**: 1
- Actions**: 13 commits
- Projects**: 1
- Security**: 0
- Insights**: 1
- About**: Mini-applications that exclusively use the Kokkos programming model
- Tags**: performance, programming-model, kokkos, mini-app, snl-mini-apps, snl-prog-models-runtimes
- Watchers**: 31
- Forks**: 6
- Stars**: 31
- Readme**: 9 stars
- 9 stars**
- 31 watching**
- 6 forks**
- Releases**: No releases published
- Packages**: No packages published

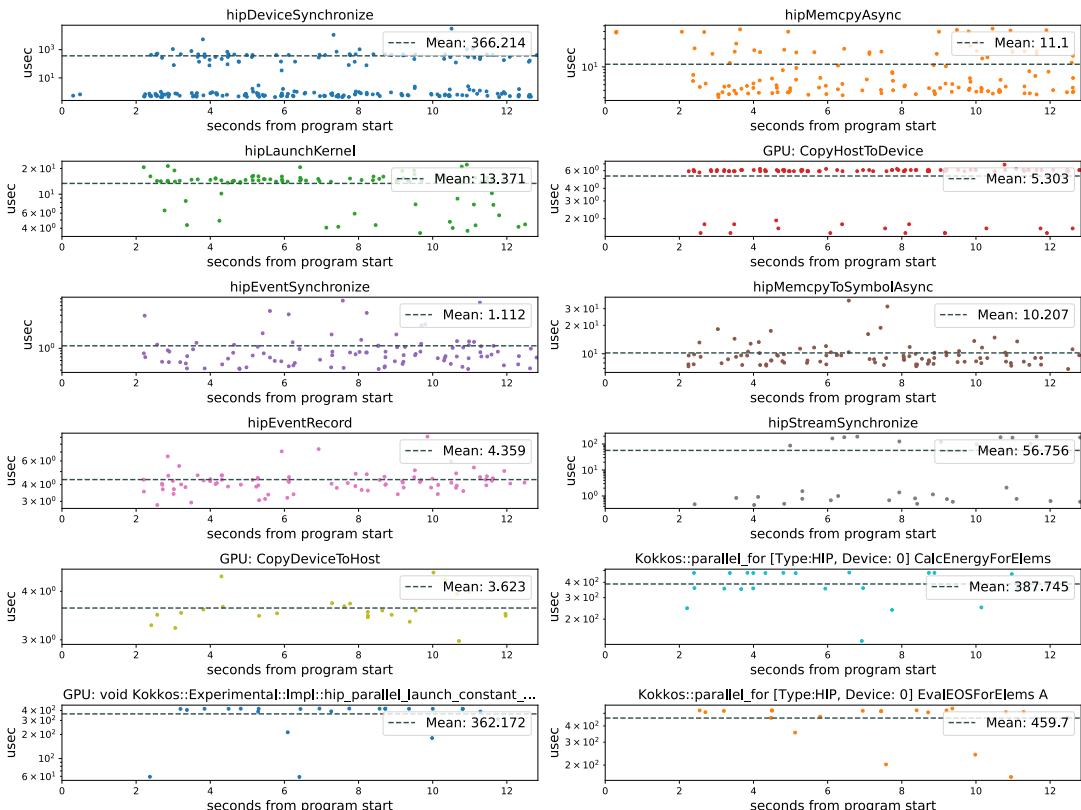
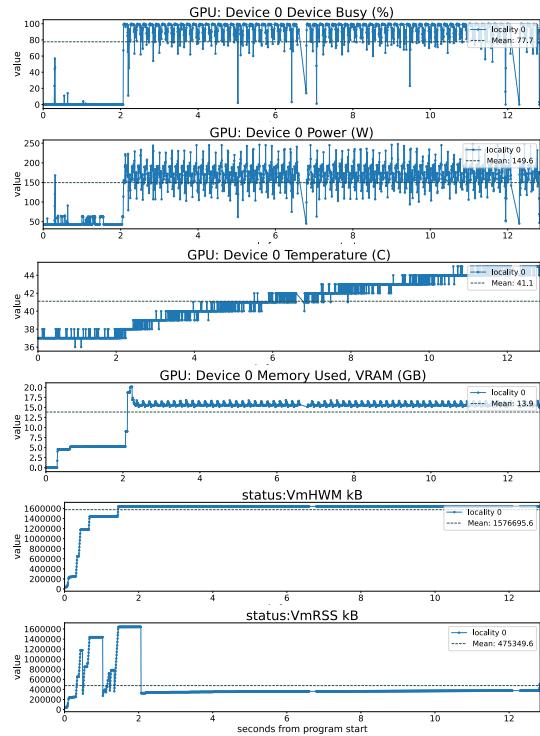
Lulesh: CUDA back end

```
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		
CCPU Timers	:	#calls		#yields		mean		total		% total
CF	:									
		APEX MAIN :	1	0		8.967		8.967		100.000
CPU		cudaDeviceSynchronize :	22901	0		0.000		4.004		44.647
I		cudaStreamSynchronize :	4398	0		0.000		0.767		8.550
		cudaMalloc :	946	0		0.001		0.727		8.106
GPU: Bytes										
		GPU: E[Kokkos::parallel_for [Cuda, Dev:0] IntegrateStressF...	64	0		0.010		0.632		7.048
GPU: Device 0 Clock		M[Kokkos::parallel_for [Cuda, Dev:0] EvalEOSForElems A :	2240	0		0.000		0.630		7.031
GPU: Device 0 Cloc		Kokkos::parallel_for [Cuda, Dev:01 CalcEnergyForEle...	2240	0		0.000		0.577		6.433
GPU: Device 0 Memory		Kokkos::parallel_for [Cuda, Dev:01 CalcEnergyForEle...								
GPU Timers	:	#calls		mean		total		% total		
Kokkos::parallel_for [Cuda, Dev:01 CalcEnergyForEle...	:									
		GPU: Context Synchronize :	22901	0.000		3.705		3.705		41.318
		GPU: Stream Synchronize :	4398	0.000		0.756		0.756		8.429
		GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	64	0.010		0.628		0.628		7.000
		GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	2240	0.000		0.543		0.543		6.051
		GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	64	0.008		0.542		0.542		6.043
		GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	64	0.008		0.524		0.524		5.846
		GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	2240	0.000		0.487		0.487		5.432
		GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	64	0.005		0.299		0.299		3.331
		GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	64	0.005		0.298		0.298		3.322
		GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	64	0.003		0.185		0.185		2.066
		GPU: void Kokkos::Impl::cuda_parallel_launch_consta...	704	0.000		0.178		0.178		1.990
		GPU: Memcpy HtoD :	10261	0.000		0.170		0.170		1.892

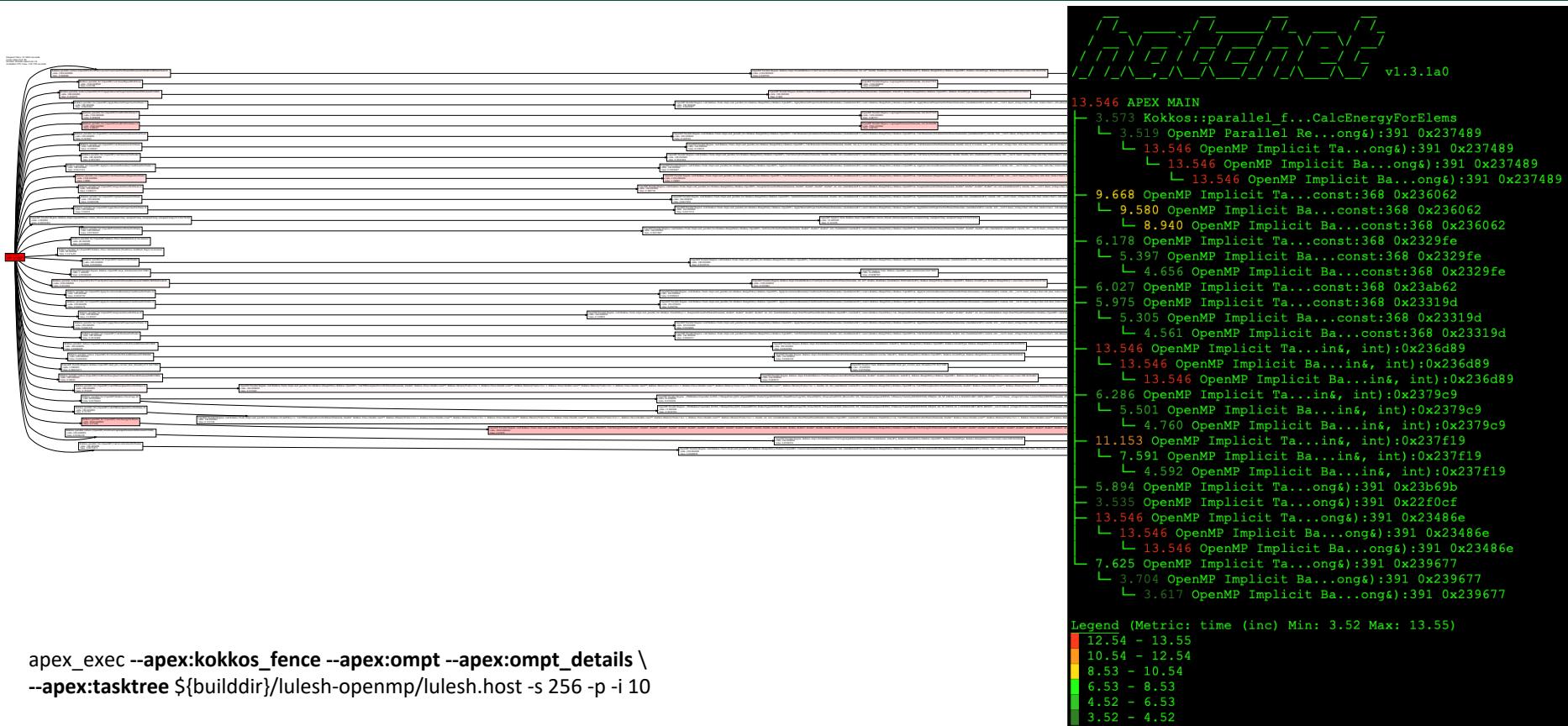
```
apex_exec --apex:kokkos_fence --apex:cuda \
--apex:monitor_gpu --apex:period 5000 \
${builddir}/lulesh-cuda/lulesh.cuda -s 256
```

Lulesh: HIP back end



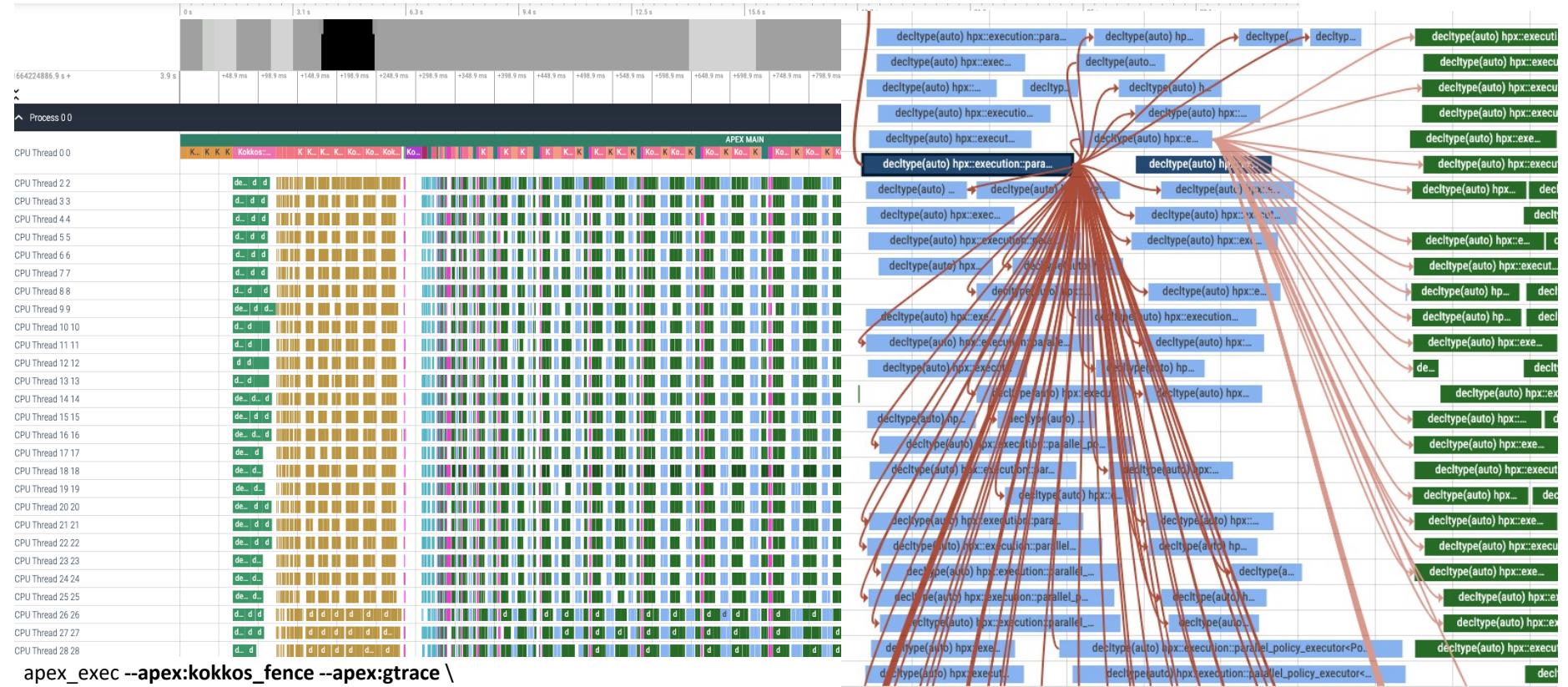
```
apex_exec --apex:kokkos_fence --apex:scatter \
--apex:hip --apex:monitor_gpu \
--apex:period 5000 ${builddir}/lulesh-hip/lulesh.hip -s 256
```

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



Policy Handling in APEX

- APEX has different methods for tuning: Active Harmony (Nelder Mead search), hill climbing, simulated annealing
- API:
 - `setup_power_cap_throttling()`, `setup_timer_throttling()`,
 - `setup_throughput_tuning()`, `setup_custom_tuning()`,
 - `shutdown_throttling()`, `get_thread_cap()`,
 - `has_session_converged()`, `get_best_values()`
- Policies are executed at an event, or periodically

Notes About Policies

- “Autotuning” any algorithm/model/feature/function has some pre-requisites:
 - Reconfigurable, parameterized problem with finite search space
 - Have to account for all causes of variability
 - Noise will influence the search
 - Have to have an observable, quantifiable affect
 - Single objective (no multi-objective tuning support yet)
- Not a magic solution

Typical Policy Example

```
int num_inputs = 3; // 2 for threads, block size; 3 for threads, block size, method
long * inputs[3] = {0L,0L,0L};
/* Set the minimum values */
long mins[3] = {2,128,DIVIDE_METHOD};
/* We'll set these later */
long maxs[3] = {0,0,0}; // we'll set these later
/* Set the step sizes */
long steps[3] = {2,128,1};
/* Active Harmony will update the values in these addresses */
inputs[0] = &active_threads;
inputs[1] = &block_size;
inputs[2] = &method;
/* Set the true max values */
maxs[0] = active_threads;
maxs[1] = num_cells/omp_get_max_threads();
maxs[2] = MULTIPLY_METHOD;
/* Register a custom event when we'll update the inputs */
my_custom_event = apex::register_custom_event("Perform Re-block");
/* Set up the tuning session */
apex::setup_throughput_tuning((apex_function_address)solve_iteration,
                               APEX_MINIMIZE_ACCUMULATED, my_custom_event, num_inputs,
                               inputs, mins, maxs, steps);
```

See: [src/C++/tuning/1d_stencil.cpp](#)

Custom Policy – use tuning request

```
enum class apex_param_type : int {NONE, LONG, DOUBLE, ENUM};  
enum class apex_ah_tuning_strategy : int {EXHAUSTIVE, RANDOM, NELDER_MEAD,  
PARALLEL_RANK_ORDER, SIMULATED_ANNEALING, APEX_EXHAUSTIVE, APEX_RANDOM};  
  
class apex_tuning_request {  
protected:  
    std::string name;  
    std::function<double()> metric;  
    std::map<std::string, std::shared_ptr<apex_param>> params;  
    apex_event_type trigger;  
    apex_tuning_session_handle tuning_session_handle;  
    bool running;  
    apex_ah_tuning_strategy strategy;  
    double radius;  
    int aggregation_times;  
    std::string aggregation_function;  
    ...
```

See: src/C++/tuning/tuning_request.cpp

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 – in progress

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.



U.S. DEPARTMENT OF
ENERGY

Office of
Science

