Broad Performance Measurement Support for Asynchronous Multi-Tasking with APEX

Kevin A. Huck
Oregon Advanced Computing Institute for Science and Society (OACISS)
University of Oregon
Eugene, Oregon USA
khuck@cs.uoregon.edu 0000-0001-7064-8417

***Abstract*—APEX (Autonomic Performance Environment for eXascale) is a performance measurement library for distributed, asynchronous multitasking runtime systems. It provides support for both lightweight measurement and high concurrency. To support performance measurement in systems that employ user-level threading, APEX uses a dependency chain in addition to the call stack to produce traces and task dependency graphs. APEX also provides a runtime adaptation system based on the observed system performance. In this paper, we describe the evolution of APEX from its design for HPX to support an array of programming models and abstraction layers and describe some of the features that have evolved to help understand the asynchrony and high concurrency of asynchronous tasking models.**

***Index Terms*—asynchronous multitasking, performance measurement, HPX, OpenMP, Kokkos, GPU programming**

I. Introduction

APEX (Autonomic Performance Environment for eXascale) [1] is a performance measurement library for distributed, asynchronous multitasking runtime systems. Implemented in C++, it provides support for both lightweight measurement and high concurrency. To support performance measurement in systems that employ user-level threading, APEX uses a dependency chain in addition to the call stack to produce traces and task dependency graphs. The first key component of APEX is the performance measurement support. APEX supports both synchronous (so-called *first person*) and asynchronous (*third person*) measurements. The synchronous measurement component in APEX uses an event API and event listeners. Whenever a task is created, started, yielded or stopped, APEX will respectively create, start/resume, yield, or stop timers for measurements. Dependencies between tasks are also tracked, using globally unique identifiers (GUID). The asynchronous measurement involves periodic or on-demand interrogation of operating system, hardware or runtime states (e.g. CPU utilization, memory usage, energy consumption). The third person measurement also includes background buffer processing to record GPU kernel execution and memory transfers to and from GPUs. Available runtime counters (e.g. idle rate, queue lengths) are also captured on-demand or on a periodic basis.

This work was supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR) under contract DE-SC0021299.

The other key component of APEX is the *Policy Engine*. The Policy Engine of APEX provides a lightweight API to engineer policies that can improve the performance of the application, execute a desired functionality on the runtime or select important runtime and application parameters. There are two ways to register a policy: either explicitly triggered or asynchronously periodic. A triggered policy can be initiated by a specific event within a runtime. There is a set of generic events provided by APEX, such as initialization/finalization, creation of a new thread, timer start/stop events, or message send/receive events. Additionally, it is also possible to provide a user defined event, also known as a custom trigger. The second class of policies, the periodic policy, operates without any event. Instead, this policy uses a defined timer which is specified during the policy's registration. All policies are stored in a policy queue and executed as instructed. The policy engine is integrated with Active Harmony [2], an online tuning library. Defined policies can use this library to search for a set of optimum parameters by minimizing a measurement value from APEX, such as wall time of a measured region/task or by looking at any other introspection data gathered by APEX.

APEX and HPX

APEX was originally designed to support the specific needs of HPX [3], [4]. HPX is an Asynchronous Many-Task Runtime system that is implemented in C++. The library implements all APIs related to concurrency and parallelism as mandated by recent ISO standards C++20 and C++23 in a conforming way. In this context, it implements all the parallel algorithms as described in the C++ standard. It has been described in detail in other publications, such as [5]–[7]. HPX has been used as an underlying runtime platform for several large scale scientific simulations, such as the Octo-Tiger astrophysics application (see [8] for more details). HPX manages local and distributed parallelism while observing all necessary data dependencies. Data and task dependencies can be expressed with HPX futures, and chained together in an execution graph. This graph can be built asynchronously, with the HPX worker threads processing the tasks when their dependencies are fulfilled. This task-based programming model is particularly useful for parallel implementations of adaptive, tree-based codes like Octo-Tiger, as the task graph is built quickly when traversing the tree to make concurrent work available to the system. Since

its first inception in 2015, APEX has been used in several HPX-related performance studies, most recently [9], [10] and with the Quantum Monte Carlo application DCA++ [11].

HPX poses a particularly difficult problem for operating system (OS) thread-centric measurement systems. HPX tasks are created, scheduled, executed, and usually yielded and resumed by the runtime system. 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. For this reason, APEX is integrated into the HPX thread scheduler, uniquely identifies each task with a GUID, and tracks all state transitions for a given task. A more complete description of this integration can be found in [1].

II. Related

Several performance tools use measurement for the purposes of offline performance analysis, including TAU [12], HPCToolkit [13], Score-P [14], Scalasca [15], Extrae [16], Caliper [17], Timemory [18], as well as many architecture-specific vendor offerings. All are powerful and capable tools in their own right, however, they were designed for offline performance analysis and tuning, typically focusing on first-person performance measurement of tied tasks/functions on a per-thread (OS thread) basis. Existing and emerging programming models present technical challenges that the designers of those measurement systems had not considered: untied task execution and migration, runtime thread control and execution, third-person observation, and runtime performance tuning. Also, as these tools are inescapably intrusive, they are not designed to be integrated permanently into an application for continuous performance introspection, but rather to be used in an iterative execute-analyze-tune cycle. In contrast, APEX is designed to perform asynchronous first- and third-person measurement for the purpose of supporting runtime introspection and performance adaptation, as well as offline analysis. Sampling support like that provided by HPCToolkit and TAU provide no insight into the task dependency chain, but rather focus on the explicit call stack that often includes many layers of runtime functions that provide no value to the user. Finally, vendor tools focus on a particular architecture, and don't provide portable performance measurement and analysis required for cross-platform studies. What also sets APEX apart from runtime-specific solutions is that it has been refactored from its HPX-centric design to a more general purpose asynchronous multi-tasking runtime profiling library.

III. Measurement Capabilities

For all of the programming models mentioned in section IV, APEX will capture timers associated with each of the first person measurements, and capture counters associated with third person measurements. The first person measurements come from preloaded library wrappers, runtime callbacks, and in some cases instrumentation. The third person measurements are captured when APEX periodically (configurable, as frequently as 200Hz) queries operating system, monitoring software, and hardware counters to observe utilization and health statistics. This includes not only CPU, memory, network, power/energy, and filesystem stats but also GPU utilization metrics and power consumption, where available. For asynchronous GPU activity, runtimes typically buffer profiling events and provide these buffers to performance tools to be consumed by a background processing thread. These events include kernel execution, memory transfer events, and in some cases synchronization events.

A. Task Dependencies

One of the key features of asynchronous multi-tasking runtimes is the ability to spawn millions of tasks within an algorithm, providing the runtime scheduler the opportunity to avoid latency, contention for shared objects, and starvation for work [19]. However, this presents a problem for performance measurement tools that are designed to handle critical paths with respect to a calling context tree. With runtimes like HPX or even GPU acceleration libraries like CUDA, HIP or SYCL, a conventional calling context tree may bear little resemblance to the logical task dependency graph associated with the algorithm. While APEX can and does keep track of calling context trees, it also has the ability to track task dependencies across threads and devices. One of the ways APEX represents the task dependencies is with task graphs and trees. Figure 1 shows a task graph representation of a trivial fibonacci program that executes the $fib(n-1)$ tasks with a C++ `std::async()` future, and the $fib(n-2)$ tasks are executed with synchronous function call. In addition, the `pthread_create()` wrapper is used to capture how the LLVM runtime spawns a new POSIX thread for each asynchronous future request. In this graph, each task type is only represented once, and nodes are revisited in the graph. Contrast that with Figure 2, which shows a task tree representation. In this case, each unique task relationship is represented based on whether the path to that task is unique. In this example, each parent task only creates one of each child task type. Figure 4 demonstrates how task relationships are represented in trace data, using flow events (see Section III-G).

B. Hardware Counters

APEX is also integrated with PAPI [20], a hardware counter abstraction and measurement library. PAPI provides a portable interface to capture hardware metrics on vastly different and heterogeneous systems. PAPI provides support to the counters through a component interface. Several components of use by APEX include Linux perf_event counters, "uncore" counters (where supported), LM sensors, power and energy counters, as well as GPU counter support for both kernel measurement as well as utilization. When used with performance portability wrappers like Kokkos (see Section IV-F), PAPI component measurement can easily be used to capture GPU hardware metrics even when the kernels and memory transfers aren't being explicitly measured via vendor support libraries.

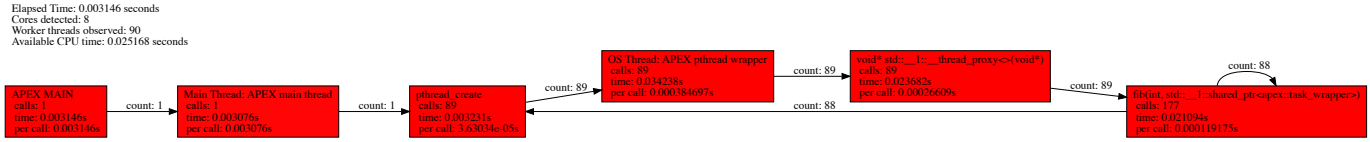

Fig. 1. Example of a task graph of fibonacci(10) in standard C++. For each sub-task, `fib(n-1)` is executed with `std::async()` and `fib(n-2)` is executed synchronously. Recursive calls to `fib()` revisit existing nodes in the graph. The more intensely the graph node is colored red, the more it contributes to the overall runtime.

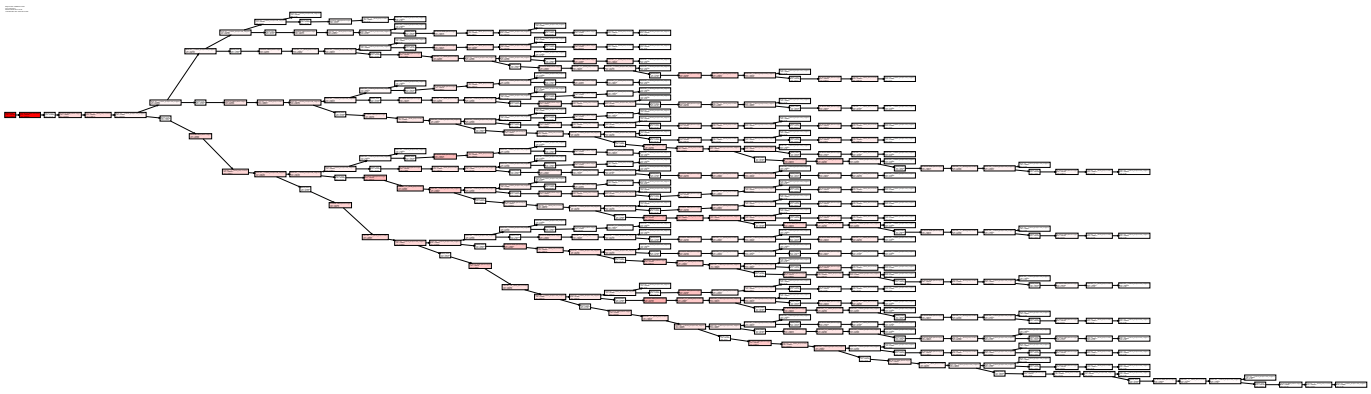

Fig. 2. Example of a task tree of fibonacci(10) in standard C++. For each sub-task, `fib(n-1)` is executed with `std::async()` and `fib(n-2)` is executed synchronously. Recursive calls to `fib()` create new sub-trees. The more intensely the tree node is colored red, the more it contributes to the overall runtime.

C. GPU Metrics

As described in the CUDA and HIP subsections IV-D and IV-E, support for observing both host-side callbacks and asynchronous device event activity is provided by vendor libraries. These libraries also provide hardware counter support for capturing metrics associated with kernel activity on GPUs. While APEX could observe these metrics directly, it makes more sense to leverage the component support in PAPI to access both CUPTI and Rocprofiler hardware metrics. Like other PAPI components, GPU hardware counter support is leveraged by APEX.

D. GPU Memory Tracking

Similar to the CUDA `cuda-memcheck` [21] program, APEX has the ability to detect memory leaks in GPU code, however it works for all supported GPU architectures, using a common software approach. For both CUDA and HIP programs, APEX keeps track of all GPU memory allocations and records the number of bytes allocated, the address returned by the allocator, and the backtrace from when the allocation occurred. The record is stored in a `std::unordered_map`, indexed by the pointer address. Later, when the allocation is freed, the record with that pointer address is removed from the map. At the end of execution, any records remaining in the map are assumed to have been leaked. Interestingly, the following simple program leaks 1024 bytes from line 9, and it is not reported by `cuda-memcheck`:

```
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 reports:

```
1 CUDA-MEMCHECK version 11.6.124 ID:(46)
2 ========= CUDA-MEMCHECK
3 ========= ERROR SUMMARY: 0 errors
```

However, APEX will report it as a leak:

```
1 1024 bytes leaked at 0x1465937ea400 from task
    cudaMalloc on tid 0 with backtrace:
2       gpu_device_malloc
3       addr=<0x1465fa0f409b> [{(unknown)} {0
    x1465fa0f409b}]
4       addr=<0x1465fa0f43b7> [{(unknown)} {0
    x1465fa0f43b7}]
5       addr=<0x1465fa0f6c1c> [{(unknown)} {0
    x1465fa0f6c1c}]
6       addr=<0x1465fe5e3143> [{(unknown)} {0
    x1465fe5e3143}]
7       addr=<0x1465fdfb247b> [{(unknown)} {0
    x1465fdfb247b}]
8       main [{/home/khuck/polaris/test/test.cpp}
    {10,0}]
9       __libc_start_main [{/lib64/libc-2.31.so} {0
    x1465fb4e434d}]
```

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

The test was performed on Polaris, an HPE Cray system at Argonne National Laboratory with AMD EPYC Milan processors and NVIDIA A100 GPUs and NVHPC 21.9 compilers[1]. This APEX feature was also helpful in finding and reporting a minor memory leak in Kokkos[2].

E. Concurrency Tracking

Occasionally, it is useful to obtain a time-series view of the profile data, usually to identify regions of low system utilization. This view is also useful when validating policies that modify OS behavior, like throttling threads to stay under a soft power cap [1]. APEX provides a concurrency view with the *concurrency listener*, which will periodically sample the timer stack on all known OS threads. The samples are written to a file at the end of execution, and post-processed with Gnuplot. An example of the resulting stacked bar chart of time-series samples is shown in Figure 3, sampled 200 times per second for the Lulesh example running with 48 Kokkos OpenMP threads (see Section V). The second y axis shows the power measurement captured at the same time interval. This example shows effective use of the available cores/threads with little idle periods, although the initialization phase shows all threads waiting at barriers while Kokkos Views are copied.

F. Profile Formats

APEX has native support for performance profiling, in which all tasks scheduled by the runtime are measured and a report is output to disk and/or the screen at the end of execution. The profile data contains the number of times each task was executed and the total time spent executing that type of task. The profile data also contains all of the sampled counters encountered during execution. Each process maintains its own profile data, and writes a different output file in various optional formats, including TAU profiles or CSV files. Profile data is optionally reported to `stdout` at program exit.

As described in section III-A, APEX can also capture the task dependency relationships. The dependencies can be captured as a graph or a tree, and the data can be stored as a TAU call path profile, task dependency trees in ASCII text, Graphviz Dot files, and Hatchet JSON files [22].

G. Trace Formats

In order to perform detailed performance analysis involving synchronization and/or task dependency analysis, full event traces including event identification and start/stop times have to be captured. To that end, APEX is integrated with the Open Trace Format 2 [23] (OTF2) library – an open, robust format for large scale parallel application event trace data. OTF2 is a robust reader/writer library and binary format specification that is typically used for high-performance computing (HPC) trace data. In order to capture full task dependency chains in HPX applications, all tasks are uniquely identified by its GUID and the GUID of its parent task. These GUIDs are captured as part of the OTF2 trace output. OTF2 data can be visualized by the Vampir [24] trace analysis tool or by Traveler [25]. To provide another free, open source alternative for trace visualization APEX can also write trace data in an informal JSON format called the Google Trace Events Format [26], which can be visualized in the web-based Perfetto [27] trace visualizer. The format includes support for flow events, an event type that crosses from one thread of execution to another, rather than a functional parent-child relationship. Figure 4 shows zoomed in regions of flow events. In this example, the HPX events generated from a single Deep Copy are connected by virtue of the task dependencies.

H. Scatterplots

Another time-series format that is useful for understanding variability between task instances is to visualize how long each task instance took with respect to when it was called. Figures 5 and 6 show examples of task and counter scatterplots, respectively. At runtime, APEX captures a configurable fraction of the tasks that are timed in the system (default: 1%), recording both how long the task took and a timestamp. Task scatterplots are helpful in understanding task distributions, whether there are outliers, and whether the times change as the simulation evolves. Counters are sampled whenever the value is updated, whether synchronously or periodically. Utilization, consumption and headroom counters are helpful in understanding how a simulation evolves and helps to interpret whether implemented runtime adaptation policies are having the desired effect.

IV. Supported Programming Models

A. POSIX and C++ Threads

Both C++ threads (`std::thread`, `std::async`) and POSIX C threads can be measured by APEX using a `pthread_create()` function wrapper. By using `LD_PRELOAD`, the wrapper replaces the underlying POSIX thread functions and captures all spawned threads, measuring the time spent in those top level functions. The `pthread_create()` function wrapper wraps the target function with a proxy function that measures its lifetime and links the task dependency chain to the current task that called `pthread_create()`. An example of this support is shown in Figures 1 and 2.

B. OpenMP

OpenMP is a pragma based extension to Fortran, C, and C++ to provide concurrent execution support on shared memory systems. Since version 5.0, The OpenMP specification has included the *OpenMP Tools* interface, OMPT [28]. OMPT provides callbacks from the OpenMP runtime to a software tool to track host-side parallel regions, loops, threads, tasks, synchronization, teams, and target offload events. The specification also includes support for providing a buffer of asynchronous target maps and kernels that execute on OpenMP capable

[1]https://www.alcf.anl.gov/polaris

[2]https://github.com/kokkos/kokkos/issues/5269

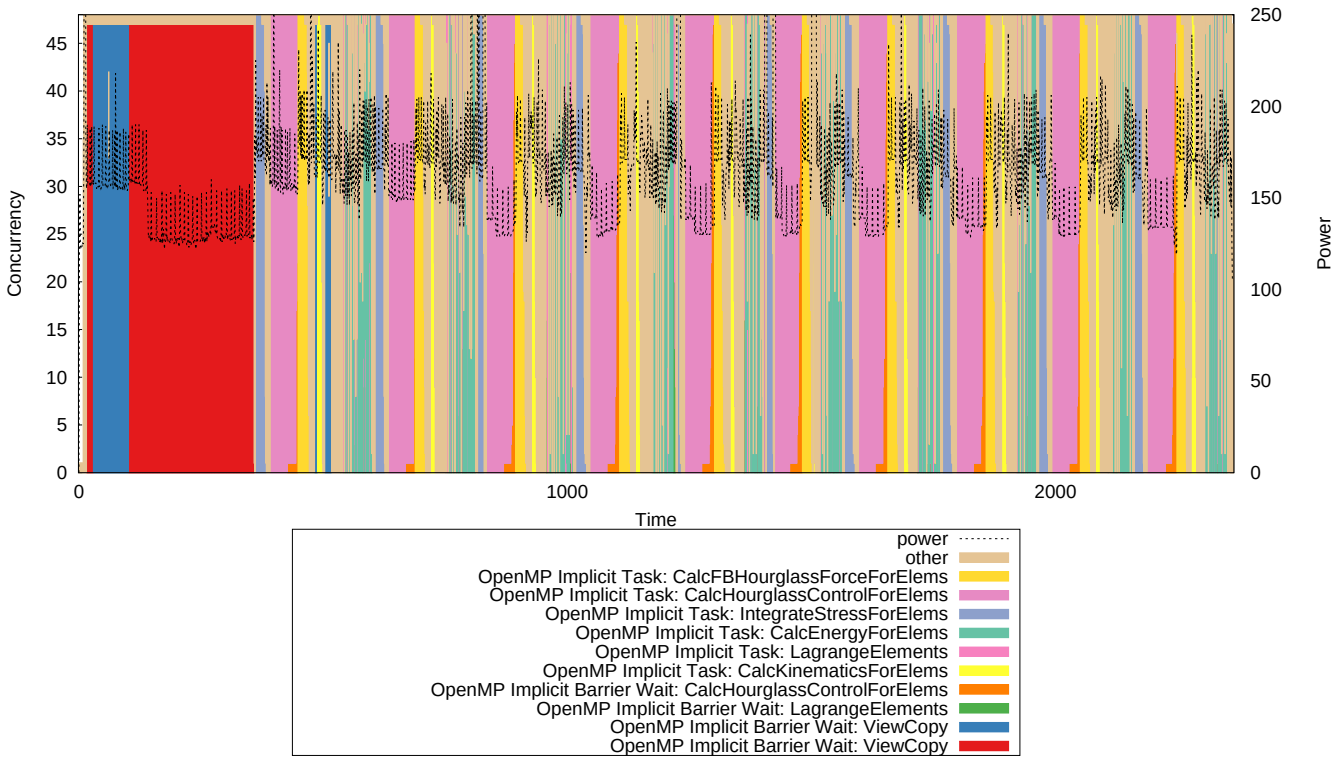

Fig. 3. Concurrency view example of Lulesh-Kokkos running with the OpenMP back end. The samples were taken 200 times per second, and the y axis indicates how many threads were executing each of the timer types when the sample was taken. The timer names have been shortened from the Kokkos lambda template instantiations for brevity. The application was executed with 48 OpenMP threads, and a problem size of 256, and only the first 10 iterations. The second y axis shows the power draw reported by RAPL during the interval, scaled to Watts per second. Due to irregularities in the power measurement, some noisy spikes occur.

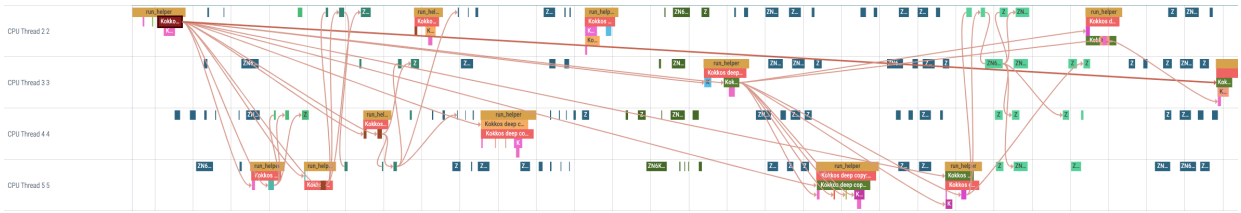

Fig. 4. Example of trace output data from `hpx-kokkos/tests/executors_test` executed with `apex_exec --apex:gtrace --apex:kokkos --apex:kokkos_fence ./tests/executors_test --hpx:threads 4`. The flow events from the deep copy future are shown.

devices like GPUs. To date, several compiler vendors have provided OMPT support in their runtimes including LLVM, AMD, Intel, and IBM. Of these, only AMD has prototyped the target offload OMPT support. APEX uses the provided OMPT support to measure OpenMP activity on both the host and any target offload devices. In addition to timing events, APEX will capture how many bytes were transferred to/from the device, allocated, and freed.

C. OpenACC

Like OpenMP, OpenACC is a pragma based extension to Fortran, C and C++ to provide computational offload support on heterogeneous systems with accelerated hardware like GPUs. The OpenACC specification [29] includes support for profiling callbacks so that tools can intercept the entry and exit to OpenACC routines. At initialization, the tool registers itself with the OpenACC runtime and provides function pointers to be called when particular events happen on the host side. In addition, some vendor back ends like CUDA also provide a buffer of asynchronous offload events that executed on the device, such as data maps to and from the device and kernel executions. APEX provides support for both the host-side events as well as the asynchronous device activity. Correlation IDs are used to track the task dependencies between the task that called an OpenACC region and the activity that was executed on the device. For each kernel executed, APEX will also capture the number of gangs, workers, and vector lanes associated with the kernel. Like with OpenMP target offload, APEX will capture how many bytes were transferred to/from the device, allocated, and freed.

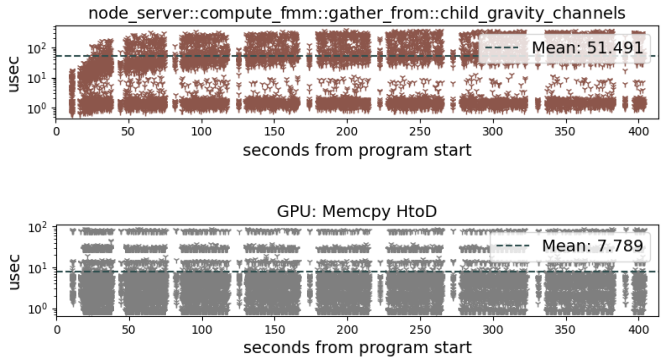

Fig. 5. Timer scatterplot examples from the Octo-Tiger simulation. In the `child_gravity_channels` graph, it is clear that the time spent in that task slowly increases over time. The `Memcpy` graph shows clear striations in the times spent to transfer different data sizes to the GPU.

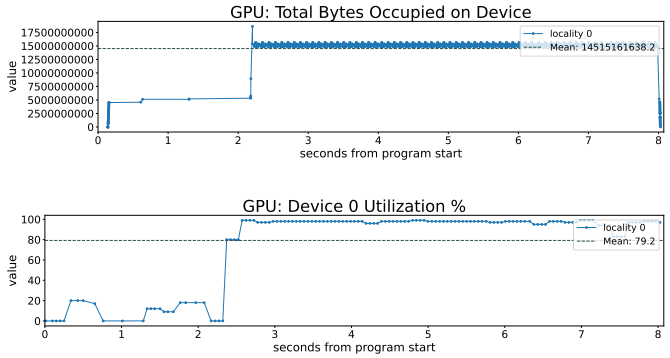

Fig. 6. Counter scatterplot examples from the Kokkos Lulesh example with the CUDA back end. The Total bytes counter is updated whenever a `CudaMalloc*` or `CudaFree` function is called. The Device Utilization graph is updated periodically and queried through the NVML library.

D. CUDA

Before the DCA++ work mentioned in section I, the *first person* measurement in APEX was only integrated with the technologies described to this point. The *third person* measurement in APEX was mostly limited to extracting data from HPX and the Linux `/proc` virtual filesystem. Because most of the DCA++ computation is offloaded to GPUs using the CUDA library, APEX was extended and integrated with the CUDA Profiling Tools Interface (CUPTI) [30] and the NVIDIA Management Library (NVML) [31]. Synchronous CUDA API callback timers and some counters (e.g. bytes transferred, bandwidth, vector lanes) from the CUDA runtime and/or device API are captured synchronously, whereas the NVML counters (e.g. utilization, bandwidth, power) are periodically captured asynchronously. Using APEX GUIDs mapped from CUDA correlation IDs, the GPU activity such as memory transfers and kernel executions are captured and linked to the host-side tasks that launched them. APEX also supports the NVTX [32] instrumentation API. The CUDA support has subsequently been used in other performance studies such as [9], [10], [33].

E. HIP

With the advent of HPC systems based on AMD GPGPUs, and the anticipated ORNL Frontier exascale system, HIP/ROCm support became a collaboration requirement. APEX was integrated with the Rocprofiler and Roctracer Tools Interface [34], [35] and the ROCm System Management Interface (ROCm-SMI) [36]. Synchronous HIP API callback timers and some counters (e.g. bytes transferred, bandwidth, vector lanes) from the HIP runtime and/or device API are captured synchronously, whereas the ROCm-SMI counters (e.g. utilization, bandwidth, power) are periodically captured asynchronously. Using APEX GUIDs mapped from HIP correlation IDs, the GPU activity such as memory transfers and kernel executions are captured and linked to the host-side tasks that launched them. APEX also supports the ROCTX [32] instrumentation API.

F. Kokkos and Raja

Both the Kokkos [37]–[39] and Raja [40] performance portability models include a mechanism for profiling tool integration and support, and APEX provides support for both of them. In the case of Kokkos, it provides a cross-platform suite of programming hooks that are consistent regardless of the selected back end. Kokkos currently provides back end support for CUDA, HIP, SYCL, HPX, OpenMP and C++ threads. Because APEX already supports most of those back ends, it was a straightforward exercise to provide Kokkos profiling support through APEX. In addition, Kokkos has an experimental runtime adaptation interface that can be used to runtime tune RangePolicy and TeamPolicy parameters associated with kernel launch, but only for some back ends. That functionality is still experimental, however it is a natural fit for the Policy Engine support in APEX. We have prototyped some experiments with that interface but have not yet converged on an effective approach.

V. Sample Experiment with Kokkos

To demonstrate the portable performance measurement available with APEX, we built and ran the Lulesh Kokkos miniapp[3]. We built versions with OpenMP, HPX, CUDA and HIP back ends, all on Gilgamesh, a development server at OACISS[4]. Gilgamesh has two 24-core Epyc Milan 7413 @ 2.6GHz CPUs, two AMD MI210 GPUs, and one NVIDIA A100 GPU. With this system, we can demonstrate performance measurement results for the HPX, OpenMP, HIP and CUDA back ends.

The tests were all performed with the `apex_exec` wrapper script which preloads the APEX library and sets any necessary environment variables to enable measurement support for Kokkos, HIP, CUDA, or OpenMP. For example, the following command would execute the HIP implementation with HIP support enabled, scatterplot, CSV, and tracing output, GPU monitoring and a monitoring period of 5ms:

[3]https://github.com/kokkos/kokkos-miniapps

[4]https://systems.nic.uoregon.edu/internal-wiki/index.php?title=Category:Servers

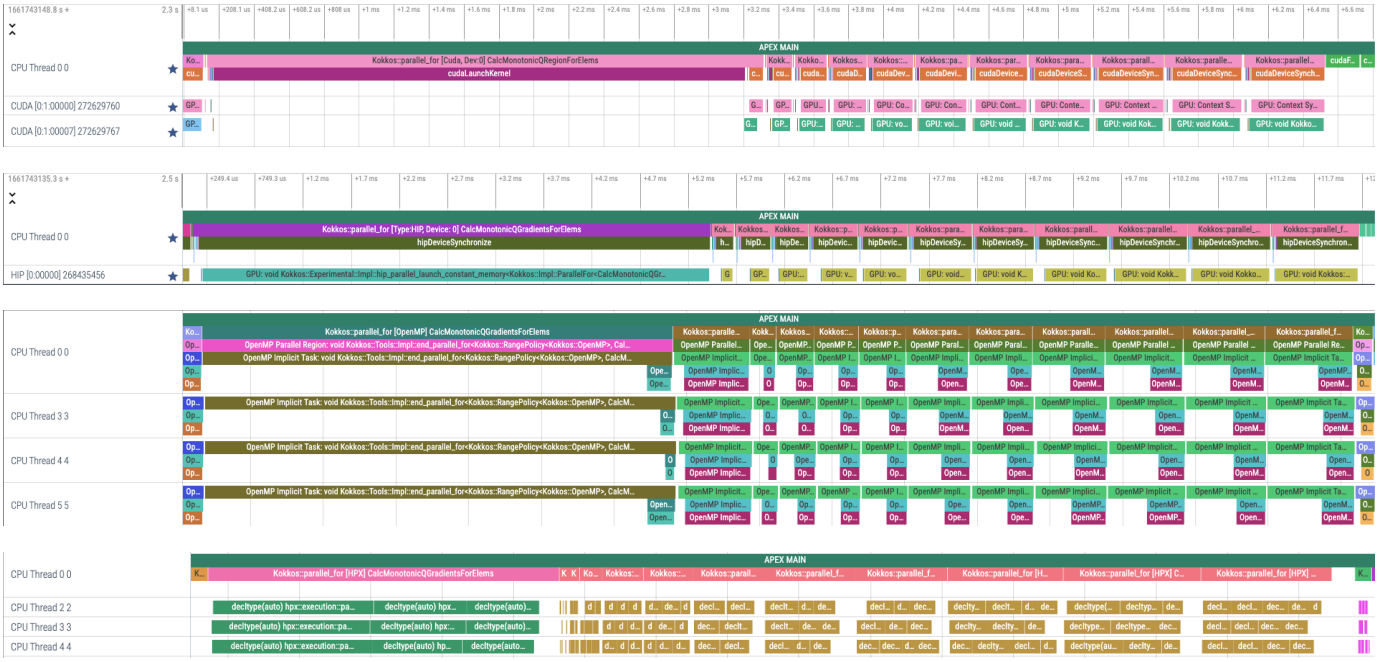

Fig. 7. CUDA, HIP, OpenMP, and HPX back end execution traces of Lulesh-Kokkos. The CUPTI (CUDA) implementation, top, provides asynchronous activity for synchronization events, which APEX segregates to a different stream than for kernel and memory transfer activity. The Roctracer (HIP) events, top middle, are similar, however it only provides synchronization events on the host side. The OMPT (OpenMP) and HPX traces, bottom, show only 4 of the 48 threads each for brevity. All four traces capture the Kokkos profiling callback events on the host side.

```
apex_exec --apex:kokkos_fence --apex:scatter --apex:
    csv --apex:tasktree --apex:hip --apex:gtrace --
    apex:monitor_gpu --apex:period 5000 lulesh.hip -
    s 256
```

Figure 7 shows Perfetto traces of one phase of the proxy app, a sequence of Kokkos `parallel_for` loops performing the `CalcMonotonicQRegionForElems` operation. In all cases, the Kokkos profiling callback events are shown, annotated with the device type and index on which it was executed. Although not shown, there are flow events that link the `cudaLaunchKernel` / `hipLaunchKernel` events with the kernels on the GPU. For the CUDA case, there are flow events that link the synchronization requests on the host with synchronization events on the device. In the OpenMP example there are flow events that link the parallel region with the implicit tasks that execute the work on each of the threads in the team. In the HPX example, there are flow events that link the Kokkos region with all of the tasks launched to complete the region. While obviously not a fair comparison, the CUDA and HIP back ends are predictably much faster than the OpenMP and HPX back ends that execute on the CPU. However, even though each back end implementation is significantly different, APEX provides the ability to compare the performance of each implementation with the same set of data collection and analysis tools.

VI. Conclusion and Future Work

In this paper, we've reviewed the latest capabilities and support in APEX that have been added since the initial APEX publication. This new support includes updated tracing capabilities, heterogeneous hardware support for CUDA and HIP, performance portability support for OpenACC, OpenMP, Kokkos and Raja, and support for capturing and visualizing task dependencies within large asynchronous applications. Future work for APEX includes adding broader power and energy measurement support with PowerAPI[5], which would provide a more portable approach instead of multiple code paths for MSR events, Powercap and RAPL. APEX also has prototyped support for Intel Level0/OneAPI and StarPU programming models, but has not yet been thoroughly tested and integrated. We also hope to replace/augment the current JSON trace output with the native Perfetto trace library, which would provide a more compact data representation and lower overhead while buffering and streaming traces out to disk. Finally, we plan to further develop and improve the runtime adaptation support for Kokkos kernels.

References

[1] K. A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, "An autonomic performance environment for exascale," *Supercomputing Frontiers and Innovations*, vol. 2, no. 3, p. 49–66, Nov. 2015. [Online]. Available: https://superfri.org/index.php/superfri/article/view/64

[2] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: Towards automated performance tuning," in *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 2002, pp. 44–44.

[3] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang, "HPX - The C++ Standard Library for Parallelism

[5]http://powerapi.org

and Concurrency," *Journal of Open Source Software*, vol. 5, no. 53, p. 2352, 2020.

[4] H. Kaiser, M. Simberg, B. Adelstein Lelbach, T. Heller, A. Berge, J. Biddiscombe, A. Reverdell, A. Bikineev, G. Mercer, A. Schaefer, K. Huck, A. Lemoine, T. Kwon, J. Habraken, M. Anderson, S. Brandt, M. Copik, S. Yadav, M. Stumpf, D. Bourgeois, A. Nair, D. Blank, G. Gonidelis, R. Stobaugh, N. Gupta, S. Jakobovits, V. Amatya, L. Viklund, P. Diehl, and Z. Khatami, "STEllAR-GROUP/hpx: HPX V1.8.1: The C++ Standards Library for Parallelism and Concurrency," Jul. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.598202

[5] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. ACM, 2014, pp. 6:1–6:11.

[6] H. Kaiser, T. Heller, D. Bourgeois, and D. Fey, "Higher-level parallelization for local and distributed asynchronous task-based programming," in *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM '15. New York, NY, USA: ACM, 2015, pp. 29–37.

[7] T. Heller, H. Kaiser, P. Diehl, D. Fey, and M. A. Schweitzer, "Closing the Performance Gap with Modern C++," in *High Performance Computing*, ser. Lecture Notes in Computer Science, M. Taufer, B. Mohr, and J. M. Kunkel, Eds., vol. 9945. Springer International Publishing, 2016, pp. 18–31.

[8] G. Daiß, P. Amini, J. Biddiscombe, P. Diehl, J. Frank, K. Huck, H. Kaiser, D. Marcello, D. Pfander, and D. Pfüger, "From piz daint to the stars: Simulation of stellar mergers using high-level abstractions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356221

[9] P. Diehl, G. Daiß, D. Marcello, K. Huck, S. Shiber, H. Kaiser, J. Frank, G. C. Clayton, and D. Pflüger, "Octo-tiger's new hydro module and performance using hpx+ cuda on ornl's summit," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 204–214.

[10] P. Diehl, D. Marcello, P. Amini, H. Kaiser, S. Shiber, G. C. Clayton, J. Frank, G. Daiß, D. Pflüger, D. Eder *et al.*, "Performance measurements within asynchronous task-based runtime systems: A double white dwarf merger as an application," *Computing in Science & Engineering*, vol. 23, no. 3, pp. 73–81, 2021.

[11] W. Wei, A. Chatterjee, K. Huck, O. Hernandez, and H. Kaiser, "Performance analysis of a quantum monte carlo application on multiple hardware architectures using the hpx runtime," in *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE, 2020, pp. 77–84.

[12] S. Shende and A. D. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, Summer 2006.

[13] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 685–701, April 2010.

[14] A. Knüpfer, C. Rössel, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel *et al.*, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[15] F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, W. Frings, K. Furlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi, "Usage of the scalasca toolset for scalable performance analysis of large-scale parallel applications," in *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2008, pp. 157–167.

[16] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, vol. 44. mar, 1995, pp. 17–31.

[17] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: performance introspection for hpc software stacks," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 550–560.

[18] J. R. Madsen, M. G. Awan, H. Brunie, J. Deslippe, R. Gayatri, L. Oliker, Y. Wang, C. Yang, and S. Williams, "Timemory: modular performance analysis for hpc," in *International Conference on High Performance Computing*. Springer, 2020, pp. 434–452.

[19] T. Sterling, D. Kogler, M. Anderson, and M. Brodowicz, "Slower: A performance model for exascale computing," *Supercomputing frontiers and innovations*, vol. 1, no. 2, pp. 42–57, 2014.

[20] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[21] "CUDA Toolkit Documentation," August 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-memcheck/index.html

[22] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: pruning the overgrowth in parallel profiles," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–21.

[23] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open trace format 2: The next generation of scalable trace formats and support libraries," in *Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 2012, pp. 481–490.

[24] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir performance analysis toolset," in *Tools for High Performance Computing*. Springer, 2008, pp. 139–155.

[25] S. A. Sakin, A. Bigelow, R. Tohid, C. Scully-Allison, C. Scheidegger, S. R. Brandt, C. Taylor, K. A. Huck, H. Kaiser, and K. E. Isaacs, "Traveler: Navigating task parallel traces for performance analysis," in *IEEE TVCG Proceedings of IEEE VIS*. IEEE, 2023, to appear.

[26] Google, "Trace Events Format," August 2022. [Online]. Available: https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU

[27] "Perfetto," August 2022. [Online]. Available: https://perfetto.dev

[28] OpenMP. (2022) Openmp specifications. [Online]. Available: https://www.openmp.org/specifications/

[29] OpenACC. (2022) Openacc specifications. [Online]. Available: https://www.openacc.org/specification

[30] "CUDA Profiling Tools Interface," August 2020. [Online]. Available: https://docs.nvidia.com/cupti/Cupti/index.html

[31] "NVIDIA Management Library," August 2020. [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml

[32] "NVIDIA Tools Extension Library," August 2020. [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/2020.1/nvtx/index.html

[33] W. Wei, E. D'Azevedo, K. Huck, A. Chatterjee, O. Hernandez, and H. Kaiser, "Memory reduction using a ring abstraction over gpu rdma for distributed quantum monte carlo solver," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2021, pp. 1–9.

[34] "ROC-Profiler Library," August 2022. [Online]. Available: https://github.com/ROCm-Developer-Tools/rocprofiler

[35] "ROC-Tracer Library," August 2022. [Online]. Available: https://github.com/ROCm-Developer-Tools/roctracer

[36] "ROCm System Management Interface," August 2022. [Online]. Available: https://github.com/RadeonOpenCompute/rocm_smi_lib

[37] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[38] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The kokkos ecosystem: Comprehensive performance portability for high performance computing," *Computing in Science Engineering*, vol. 23, no. 5, pp. 10–18, 2021.

[39] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[40] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 2019, pp. 71–81.

I. Artifact Description Appendix: [Broad Performance Measurement Support for Asynchronous Multi-Tasking with APEX]

A. Abstract

This description contains the information needed to launch some experiments of the ESPM22 paper "Broad Performance Measurement Support for Asynchronous Multi-Tasking with APEX". In this appendix, we explain how to compile and run the modified APEX examples used in section VI. Examples from throughout the paper are also explained.

B. Description

1) Check-list (artifact meta information):

- **Program:** C++ binaries, C and C++ libraries.
- **Compilation:** Varies by platform, hipcc 5.2.0 for AMD GPUs, NVHPC 22.5 for NVIDIA GPUs, LLVM 14 otherwise. Cmake 3.22.1 with `CMAKE_BUILD_TYPE=Release`.
- **Run-time environment:** Linux 4.18.0-372.9.1.el8.x86_64
- **Hardware:** Server with AMD EPYC 7413 24-Core Processor, NVIDIA A100 GPU, AMD MI210 GPU.
- **Experiment workflow:** Download source, configure dependencies, build dependencies, build APEX, build Lulesh with Kokkos implementation.
- **Publicly available?:** Yes.

2) How software can be obtained (if available):

- **APEX:** https://github.com/UO-OACISS/apex
- **HPX:** https://github.com/STEllAR-GROUP/hpx
- **Kokkos:** https://github.com/kokkos/kokkos
- **Lulesh with Kokkos:** https://github.com/kokkos/kokkos-miniapps
- **HWLOC:** https://www.open-mpi.org/software/hwloc/v2.8/
- **Complete build instructions and scripts are available at:** https://github.com/khuck/publication-artifacts/tree/main/2022-ESPM

3) Hardware dependencies: None.

4) Software dependencies: HPX examples require a modern C++ compiler supporting C++17 or newer. AMD GPU support requires hipcc 4.5 or newer. NVIDIA GPU support requires CUDA 10 or newer, and/or NVHPC compilers.

5) Datasets: None.

C. Installation

1) Set up the build environment by sourcing the `sourceme.sh` script in the build instructions.
2) Get the source by running the `getsrc.sh` script in the build instructions.
3) Build the APEX CUDA configuration by running the `apex-cuda.sh` script in the build instructions.
4) Build the APEX HIP configuration by running the `apex-hip.sh` script in the build instructions.
5) Build the APEX OpenMP configuration by running the `apex-openmp.sh` script in the build instructions.
6) Build HPX by running the `hpx.sh` script in the build instructions.
7) Build Lulesh-Kokkos with CUDA back end by running the `lulesh-cuda.sh` script in the build instructions.
8) Build Lulesh-Kokkos with HIP back end by running the `lulesh-hip.sh` script in the build instructions.
9) Build Lulesh-Kokkos with HPX back end by running the `lulesh-hpx.sh` script in the build instructions.
10) Build Lulesh-Kokkos with OpenMP back end by running the `lulesh-openmp.sh` script in the build instructions.

D. Experiment workflow

Run Lulesh-Kokkos with CUDA back end:

```
export OMP_PROC_BIND=spread
export OMP_PLACES=threads
export OMP_NUM_THREADS=48
export KOKKOS_NUM_THREADS=${OMP_NUM_THREADS}

${installdir}/apex-cuda/bin/apex_exec \
--apex:kokkos --apex:csv --apex:tasktree --apex:cuda
    --apex:gtrace --apex:monitor_gpu --apex:period
    5000 \
${builddir}/lulesh-cuda/lulesh.cuda -s 256
```

Run Lulesh-Kokkos with HIP back end:

```
export OMP_PROC_BIND=spread
export OMP_PLACES=threads
export OMP_NUM_THREADS=48
export KOKKOS_NUM_THREADS=${OMP_NUM_THREADS}

${installdir}/apex-hip/bin/apex_exec \
--apex:kokkos --apex:csv --apex:tasktree --apex:hip
    --apex:gtrace --apex:monitor_gpu --apex:period
    5000 \
${builddir}/lulesh-hip/lulesh.hip -s 256
```

Run Lulesh-Kokkos with HPX back end:

```
export OMP_NUM_THREADS=48
export KOKKOS_NUM_THREADS=${OMP_NUM_THREADS}
export LD_LIBRARY_PATH=${installdir}/hpx/lib64:/usr/
    local/packages/boost/1.75.0/lib

export APEX_SCREEN_OUTPUT=1
${builddir}/lulesh-hpx/lulesh.host
```

Run Lulesh-Kokkos with OpenMP back end:

```
export OMP_PROC_BIND=spread
export OMP_PLACES=threads
export OMP_NUM_THREADS=24
export KOKKOS_NUM_THREADS=${OMP_NUM_THREADS}
export APEX_MEASURE_CONCURRENCY=1
#export APEX_MEASURE_CONCURRENCY_PERIOD=500000

${installdir}/apex-openmp/bin/apex_exec \
--apex:kokkos --apex:csv --apex:tasktree --apex:ompt
    --apex:ompt_details \
${builddir}/lulesh-openmp/lulesh.host -s 64 -p -i
    200
```

E. Evaluation and expected result

Lulesh should run to completion in all cases, and APEX should provide results similar to those presented in the paper.

F. Experiment customization

In order to allow the command line of `make` to override the values in the Lulesh makefiles, the files `kokkos-miniapps/lulesh-2.0/kokkos-*/Makefile*` had to be changed from (for example):

```
KOKKOS_PATH = ${HOME}/Kokkos/kokkos
KOKKOS_DEVICES = "Cuda,OpenMP"
KOKKOS_ARCH = "Volta70"
KOKKOS_CUDA_OPTIONS = force_uvm,enable_lambda
```

to:

```
KOKKOS_PATH := ${HOME}/Kokkos/kokkos
KOKKOS_DEVICES := "Cuda,OpenMP"
KOKKOS_ARCH := "Volta70"
KOKKOS_CUDA_OPTIONS := force_uvm,enable_lambda
```

G. Notes

Complete build instructions and scripts are available at: https://github.com/khuck/publication-artifacts/tree/main/2022-ESPM