Multi-Platform SYCL Profiling with TAU

Nicholas Chaimov
Sameer Shende
Allen Malony

ParaTools, Inc.

IWOCL 2020
• Motivation: platform-agnostic performance counter profiling
• What is TAU?
• Early Implementation Work
  – NVIDIA: hipSYCL + CUPTI
  – AMD: hipSYCL + rocprofiler
  – Intel: OpenCL library wrapping
  – Intel: oneAPI Level Zero tool interface
Motivation

• Performance portability
  – We want code to be not just portable, but performance portable
  – Analyzing requires ability to make measurements across platforms.
  – Vendor-specific tools are not cross-platform.
  – TAU with SYCL
    • Provide a cross-platform performance tool for a cross-platform programming model
The TAU Performance System®

- Tuning and Analysis Utilities (25+ year project)
- Comprehensive performance profiling and tracing
  - Integrated, scalable, flexible, portable
  - Targets all parallel programming/execution paradigms

- Integrated performance toolkit
  Instrumentation, measurement, analysis, visualization
  Widely-ported performance profiling / tracing system
  Performance data management and data mining
  Open source (BSD-style license)

- Integrates with runtimes and application frameworks
TAU Supports All HPC Platforms

C/C++

Fortran

CUDA

OpenACC

Intel MIC

LLVM

PGI

Intel

GNU

MinGW

Upstream

Insert yours here

Python

GPI

Java

MPI

OpenMP

Cray

Sun

AIX

Windows

Fujitsu

ARM

Android

MPC

OpenSHMEM

ParaTools
TAU Supports All HPC Platforms

- C/C++
- Fortran
- pthreads
- Intel
- GNU
- MinGW
- CUDA
- UPC
- OpenACC
- Intel MIC
- OpenMP
- Python
- GPI
- Java
- MPI
- LLVM
- PGI
- Cray
- OpenACC
- Intel MIC
- OpenSHMEM
- BlueGene
- Fujitsu
- MPC
- ARM
- Android
- Windows
- Linux
- AIX
- Sun
- AIX
- SYCL
Measurement Approaches

## Profiling

Shows how much time was spent in each routine

<table>
<thead>
<tr>
<th>Routine</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEQ_IKSWEEP</td>
<td>9647.318</td>
</tr>
<tr>
<td>LEQ_BICGSO</td>
<td>4357.213</td>
</tr>
<tr>
<td>LEQ_MATVECT</td>
<td>2669.887</td>
</tr>
<tr>
<td>SOLVE_SPECIES_EQ</td>
<td>1777.752</td>
</tr>
<tr>
<td>SOLVE_LIN_EQ</td>
<td>1417.986</td>
</tr>
<tr>
<td>PHYSICAL_PROP</td>
<td>1028.448</td>
</tr>
<tr>
<td>RRATES</td>
<td>783.402</td>
</tr>
<tr>
<td>LEQ_MSOLVE</td>
<td>682.376</td>
</tr>
<tr>
<td>INIT_AB_M</td>
<td>530.858</td>
</tr>
<tr>
<td>CALC_MASS_FLUX_SPHR</td>
<td>463.788</td>
</tr>
<tr>
<td>INIT_MU_S</td>
<td>446.025</td>
</tr>
</tbody>
</table>

## Tracing

Shows when events take place on a timeline
Performance Data Measurement

Direct via Probes

- Exact measurement
- Fine-grain control
- Calls inserted into code or runtime

Indirect via Sampling

- No code modification
- Minimal effort
- Relies on debug symbols (-g option)

```c
// code
call TAU_START('name')
call TAU_STOP('name')
```
Questions TAU Can Answer

- **How much time** is spent in each application routine and outer loops? Within loops, what is the contribution of each statement?

- **How many instructions** are executed in these code regions? Floating point, Level 1 and 2 *data cache misses*, hits, branches taken?

- **What is the memory usage** of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?

- **What are the I/O characteristics** of the code? What is the peak read and write *bandwidth* of individual calls, total volume?

- **What is the extent of data transfer** between host and a GPU? In applications using various programming models, such as CUDA, HIP, OpenCL, Kokkos, SYCL, etc.

- **What is the contribution of each phase** of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?

- **How does the application scale**? What is the efficiency, runtime breakdown of performance across different core counts?
• **OpenCL**
  – OpenCL profiling interface
  – Track timings of kernels

• **OpenACC**
  – OpenACC instrumentation API
  – Track data transfers between host and device (per-variable)
  – Track time spent in kernels

• **CUDA**
  – Cuda Profiling Tools Interface (CUPTI)
  – Track data transfers between host and GPU
  – Track access to uniform shared memory between host and GPU

• **ROCm**
  – Rocprofiler and Roctracer instrumentation interfaces
  – Track data transfers and kernel execution between host and GPU

• **Python**
  – Python interpreter instrumentation API
  – Tracks Python routine transitions as well as Python to C transitions
• Proof-of-concept implementation using hipSYCL.

• CUPTI
  – Synchronous callbacks for host-side API calls.
  – Asynchronous callbacks for device-side events.
    • Hardware performance counter access.

• Phase-based profiling to correlate CUDA kernels back to SYCL code.
  – CUPTI external correlation ID

```c
parallel_for(count, kernel functor(id<> item) {
    int i = item.get_global(0);
    r[i] = a[i] + b[i] + c[i];
});
```
SYCL Profiling on AMD GPUs

• As with NVIDIA, our proof-of-concept implementation uses hipSYCL.

• rocProfiler library for callbacks from AMD ROCm.
  – No equivalent to CUPTI’s external correlation IDs.
  – Interception API allows user-provided data to be attached to interception callback.
    • But interception API requires serializing kernel dispatches.

```c
parallel_for(count, kernel_functor([=](id<> item) {
    int i = item.get_global(0);
    r[i] = a[i] + b[i] + c[i];
  }));
```
SYCL Profiling on Intel Embedded GPUs (1)

- Initial implementation of Intel SYCL based on OpenCL backend.
- TAU provides wrapper libraries around OpenCL API functions which replace the runtime-provided versions.

- Wrapper for `clCreateCommandQueue` and `clCreateCommandQueueWithProperties` force profiling on.
- Each wrapper
  - Starts a timer
  - If relevant, records a context event indicating the size and source line of a transfer
  - Calls the underlying system version of the function
  - Stops the timer
- Loaded into unmodified application with LD_PRELOAD or through linker script at link time

```cpp
parallel_for(count, kernel_functor([=](id<> item) {
    int i = item.get_global(0);
    r[i] = a[i] + b[i] + c[i];
});
```
• Event names from OpenCL profiling interface provide mangled name of originating functor from SYCL code.

• Intel Level Zero Tools Interface
  – No external correlation ID support
  – However, event name contains enough context information to avoid need
  – Tracer Markers allow user-provided data to be inserted into the event stream
Download TAU

http://tau.uoregon.edu
http://taucommander.com
https://e4s.io
Free download, open source, BSD license

Questions?
Contact support@paratools.com