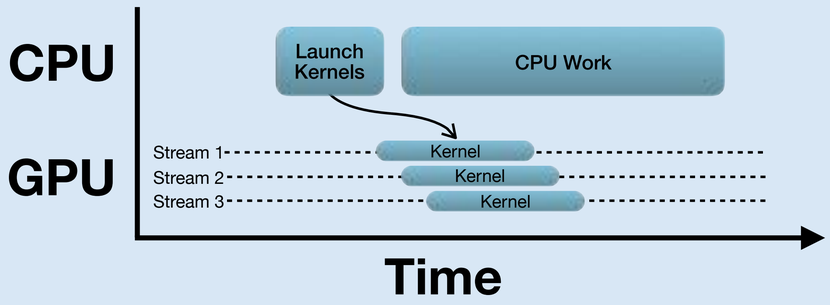
Use of CUDA Profiling Tools Interface (CUPTI) for Profiling Asynchronous GPU Activity

Michael E. Rowan (mrowan@lbl.gov), Jack R. Deslippe, Kevin N. Gott, Axel Huebl, Remi Lehe, Andrew T. Myers, Maxence Thévenet, Jean-Luc Vay, and Weiqun Zhang — Lawrence Berkeley National Laboratory

Introduction

On-the-fly performance monitoring of applications is needed for adaptive load balancing. For applications which employ GPU acceleration, active performance monitoring is complicated by the possibility of concurrent kernel execution on the device, and by asynchronous kernel launches from the host [1, 2].

How to measure kernel time?



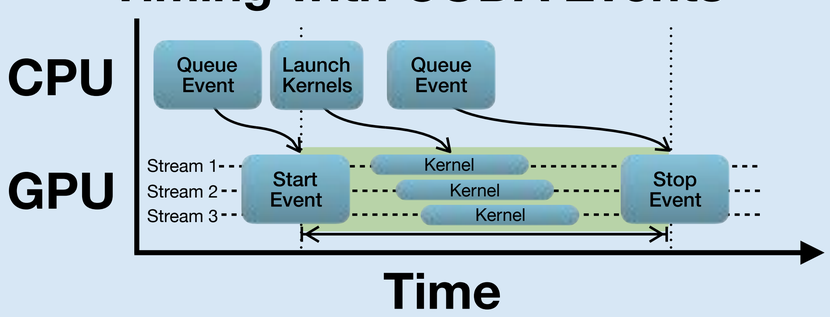
Here, we present a case study in the use of the **CUPTI** (CUDA Profiling Tools Interface) library for active performance monitoring of a simplified GPU kernel, and a more realistic case from the advanced electromagnetic particle-in-cell code **WarpX** [3, 4]. In the future, this implementation of performance monitoring with CUPTI may be used in load balancing for WarpX.

Method

To allow for **runtime-accessible kernel timing**, we use CUDA Profiling Tools Interface (**CUPTI**), a library which allows for active performance monitoring of CUDA applications [3]. The CUPTI callback scheme for timing kernel execution offers more fine-grained information than another technique (**CUDA Events**) commonly used for GPU timing (illustrated in the diagram below). With CUDA Events, the scheme for kernel timing is as follows:

- Create events (e.g. ‘start’ and ‘stop’)
- Record stream contents into event
- Compute elapsed time between events

Timing with CUDA Events

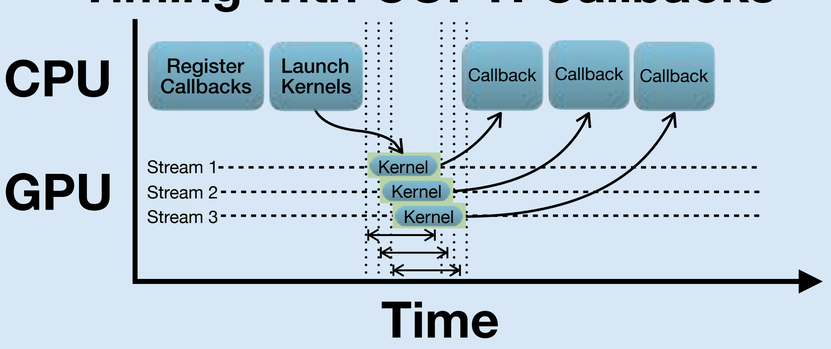


While it is simple to implement a CUDA Event-based timer, timing by queuing events to separate streams does not provide easy-to-interpret kernel timings due to asynchronous execution of kernels across streams. An alternative method is the CUPTI timing scheme (illustrated below). Timing with CUPTI Callback functions consists of a few key steps:

- Register callback functions to request and deliver records
- Trigger callback functions via GPU activity
- Compute kernel execution times from activity records

A key advantage of CUPTI callback-based timing over CUDA Events-based timing is access to individual kernel durations.

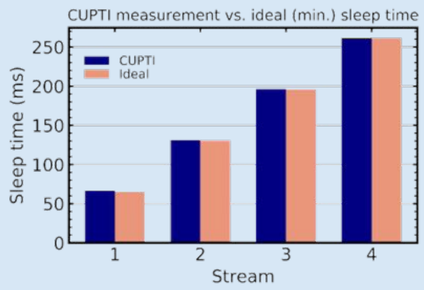
Timing with CUPTI Callbacks

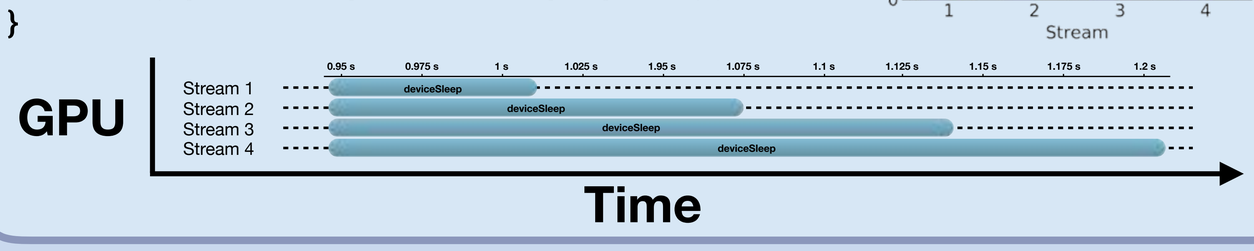


Implementation and tests

We implement the CUPTI-based timer as a module in the Adaptive Mesh Refinement (AMR) library **AMReX** [5]. We test the CUPTI-based timer using a simple function `deviceSleep`, which instructs the device to be inactive (on a per-stream basis) for a chosen number of cycles. For the test shown below, we use NVIDIA Volta V100, with a peak clock frequency of 1.53 GHz, and launch sleep kernels on four separate streams, for multiples of 1, 2, 3, and 4×10^8 cycles. The bar plot at the bottom illustrates the (concurrent) execution of these kernels across streams; the bar plot on the right shows a comparison between the instructed (light red), and measured (dark blue) sleep times obtained via CUPTI; close agreement between the two indicates that the CUPTI measurements are accurate.

```
void deviceSleep (clock_value_t sleep_cycles)
{
  clock_value_t start = clock64();
  clock_value_t cycles_elapsed;
  do { cycles_elapsed = clock64() - start; }
  while (cycles_elapsed < sleep_cycles);
}
```



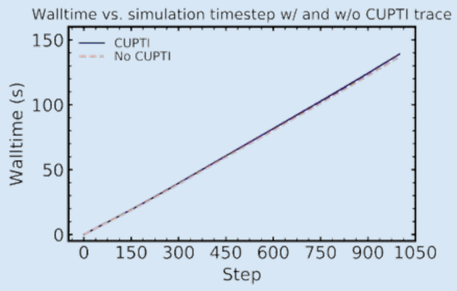


The code snippet below shows a sample of instrumented code from the particle-in-cell code WarpX; to collect activity records via CUPTI, a ‘trace’ is initialized before CUDA is initialized. Trace initialization consists of two steps:

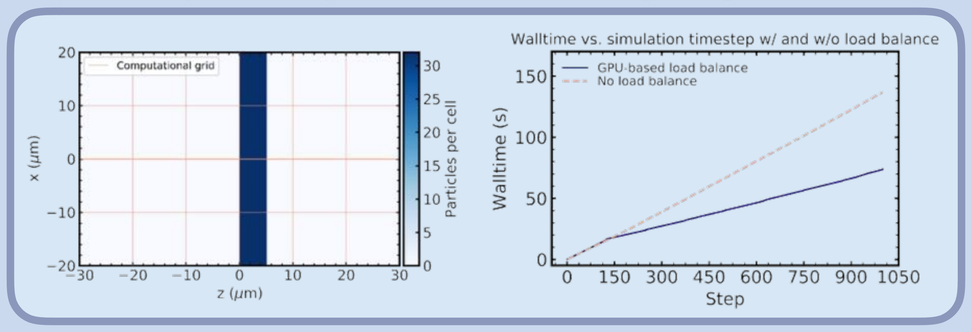
- Enable collection of kernel activity records via CUPTI
- Register callback functions to handle storage and delivery of kernel activity records

Callback functions are activated by GPU activity, for example with ‘`cuptiActivityFlushAll`’, which instructs CUPTI to deliver a buffer containing activity records to the host.

```
initCuptiTrace(); // Initialize CUPTI trace;
    :             // enable collection of
    :             // kernel activity records
BL_PROFILE_VAR_START_CUPTI(var); // Clr recs
PushPX(...); // GPU work; particle push step
BL_PROFILE_VAR_STOP_CUPTI(var); // Rtn recs
    cuptiActivityFlushAll(0);   // Wait for return of CUPTI records
        bfrCompleted(...);      // via callback function
```



The plot above on the right shows a comparison between the walltime for a WarpX simulation with CUPTI initialized (dark blue) and without CUPTI (light red). For the simulation presented here, agreement between the dark blue and light red lines indicates that using CUPTI, one incurs only a **relatively small overhead** on total walltime. For a WarpX test case that is load-imbalanced (see below), we test load balancing using CUPTI-obtained timings. The setup is shown to the left. Before load balancing, work is unevenly distributed over GPUs. After load balancing, boxes of the domain are redistributed to GPUs so that work is more evenly distributed, decreasing the walltime.



Summary and future work

- Tested CUPTI-based GPU timing for simple test cases, as well as a more realistic use case with WarpX
- CUPTI allows for accurate timing of kernel execution.
- CUPTI timing strategy will enable GPU timing and **load balancing** which incorporates GPU work in realistic WarpX problems, for example ion acceleration via laser-plasma interaction.

AMReX

SCAN ME

References

[1] A. D. Malony *et al.*, “Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs”, *ICPP* (2011).
[2] S. Rennich, “CUDA C/C++ Streams and Concurrency”, https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf (accessed 01/08/2020).
[3] NVIDIA, “API Reference Guide for CUPTI”, https://docs.nvidia.com/cuda/cupti/index.html#abstract (accessed 01/08/2020).
[4] J.-L. Vay *et al.*, “Warp-X: A new exascale computing platform for beam–plasma simulations”, *Nucl. Inst. Meth. A* (2018).
[5] Zhang *et al.,* “AMReX: A Framework for Block-Structured Adaptive Mesh Refinement”, *Journal of Open Source Software* (2019).

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.

WarpX

SCAN ME