A Tool for Performance Analysis of GPU-accelerated Applications

Keren Zhou and John Mellor-Crummey
Department of Computer Science, Rice University

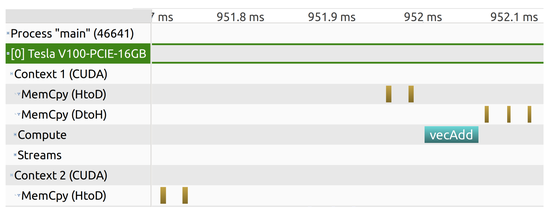
Abstract

We extended HPCToolkit to build a complete profile view for analyzing the runtime characteristics of GPU-accelerated applications. Our tool has the following key innovations:

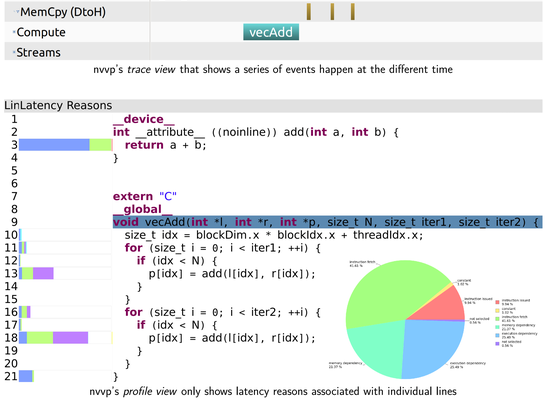
- A wait-free sample collection system in multi-threaded environment with low contention and memory overhead.
- A complete profile view with calling contexts and control flows for both CPU and GPU codes.
- A heuristic method to attribute costs associated with GPU instruction samples to the appropriate contexts.

Background

- A variety of programming models are developed for emerging GPU-accelerated HPC systems, such as `OpenMP target` and `RAJA`.
- Due to the lack of a complete *profile view*, existing performance tools are not helpful for programs written by these programming models with complicated runtime contexts.
- `nvvp` ignores calling contexts and control flows in GPU code.
- Other tools, including `Vampir`, `TAU`, `Allinea MAP`, and `Open|SpeedShop`, only provide a *trace view*.



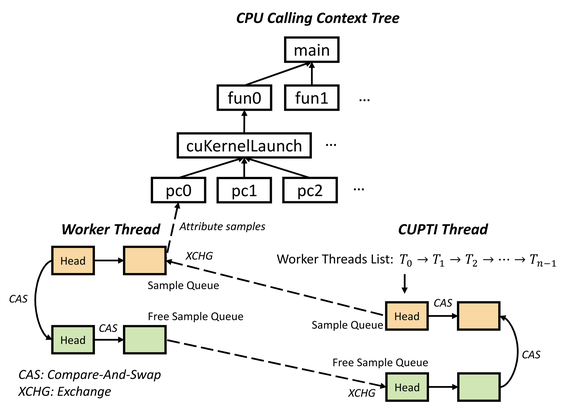
nvvp's *trace view* that shows a series of events happen at the different time



nvvp's *profile view* only shows latency reasons associated with individual lines

Sample Collection

- We designed a wait-free sample collection system to attribute costs associated with GPU instruction samples back to threads that launched the corresponding GPU kernels.
- **Mechanism:** We define two sets of threads in our system. GPU instruction samples are collected using NVIDIA's CUPTI API, which spawns a background *CUPTI thread* at program launch. *Worker threads* are responsible for handling CPU workloads, including kernel launch and GPU memory allocation.
  Our system manages inter-thread communication using two types of records. When a worker thread T launches a kernel, the worker thread assigns a correlation ID C to the kernel instance and sends a *notification record* to the CUPTI thread indicating that C belongs to T. When the CUPTI thread collects samples associated with C, it communicates a *sample attribution record* back to thread T.
- **Requirements:** The collection system must be **non-blocking**. Neither the CUPTI thread nor the worker threads should delay each other.
  The system also needs **explicit memory management**. For long-running applications that generate many notification and sample attribution records, records must be reclaimed after they are consumed.
- **Approach:** To satisfy the requirements, each worker thread and the CUPTI thread share two pairs of non-blocking queues. Because notification and sample attribution queues employ the same mechanism and only differ in the direction and data passed, we only explain the sample collection queues in the figure below.

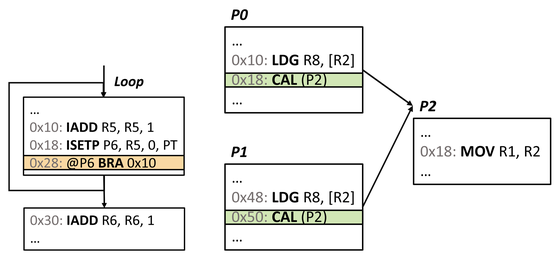


Samples are collected by the CUPTI thread and stolen by worker threads; "free" sample attribution records are stolen back by the CUPTI thread

- The CUPTI thread maintains a list of sample attribution queues, where each queue stores sample attribution records that need to be delivered to the corresponding worker thread.
- The CUPTI thread adds records to sample attribution queues using CAS. Each worker thread swaps the head of its sample attribution queue with NULL to steal all of its records.
- Wait-free progress is guaranteed because a CUPTI thread CAS on a queue fails at most once when tries to add records.
- After a worker thread attributes its samples to its calling context tree, the worker puts its sample attribution records into a free queue, where they can be stolen back by the CUPTI thread using a swap.

Recover Calling Contexts and Control Flows

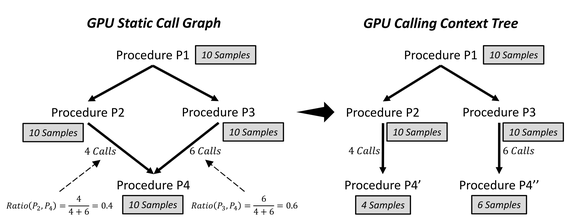
- We recover draft control flow graphs, using NVIDIA's `nvdiasm`. In their original form, these graphs are unsuitable for analysis. We modify them to address their shortcomings.
- We split blocks into basic blocks that end with either a function call or a branch instruction.
- We link dangling blocks by matching the offset of the first instructions in the dangling blocks and last instructions in control flow graphs.
- Since `nvdisasm` fails to parse all the functions in GPU binaries if some of the functions are unparsable due to invalid instructions or branch targets, we parse each function individually and skip ones that cause failures.
- We apply the `Dyninst` binary analysis toolkit to analyze loop nests in control flow graphs.
- We recovery static call graphs by parsing call instructions and linking callers and callees.



(a) Control Flow Graph (b) Static Call Graph

Heuristic Sample Attribution

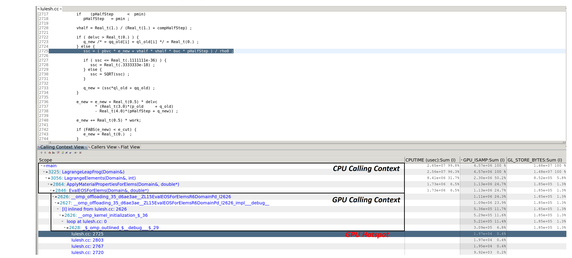
- We attribute costs to inline functions, loop nests, and individual source lines.
- We transform static call graphs into calling context trees by splitting call edges and cloning called procedures.
- We adopt a heuristic method to apportion samples to procedures called from multiple call sites.
- To handle recursive calls, we merge all procedures in the same strongly connected component group into a 'supernode' and apportion costs for the 'supernode'.



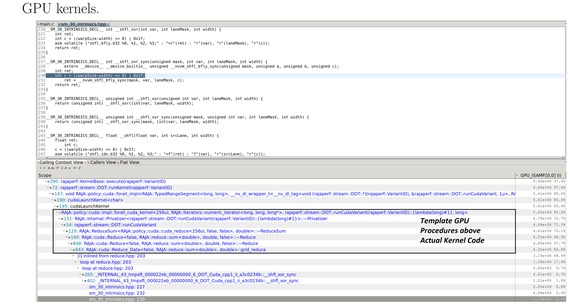
Costs associated with each call site to a procedure are approximated using the number of samples at each call site

Applications

- We have used our tools to analyze HPC applications on the Summit supercomputer whose computing nodes equipped with IBM POWER9 processors and NVIDIA Volta GPUs.
- We profiled *LULESH*—an ultra-shock hydrodynamics benchmark code that offloads computation to GPUs using OpenMP TARGET directives.



- We profiled the *RAJA performance suite* which uses C++ templates to implement loop-based GPU kernels.



Next Steps

Currently, nvvp takes multiple phases to collect kernel performance, data movement, compute utilization, and PC sampling information because of limitations in the CUPTI API, which allows neither PC sampling information to be collected with memory activities nor activity records to be collected with event records. We have devised an innovative method to measure parallel efficiency with only sampling information. In this way, we can collect a few activity records that are compatible with PC sampling and gather all the necessary performance information in a single phase.

We plan to enhance our graphical interface to facilitate hierarchical analysis of multiple metrics. Also, we want to augment the information we present with GPU parallelism metrics and gain experience analyzing MPI-based HPC applications.