SANDIA REPORT
SAND2020-xxxx
Printed September 2020

Sandia National Laboratories

LDMS-GPU: Lightweight Distributed Metric Service (LDMS) for NVIDIA GPGPUs

Ammar Elwazir
Klipsch School of Electrical and Computer Engineering
New Mexico State University
Las Cruces, NM 88003
ammarwa@nmsu.edu

Abdel-Hameed A. Badawy
Klipsch School of Electrical and Computer Engineering
New Mexico State University
Las Cruces, NM 88003
badawy@nmsu.edu

Omar Aaziz
Sandia National Laboratories
Albuquerque, NM
oazziz@sandia.gov

Jeanine Cook
Sandia National Laboratories
Albuquerque, NM
jeacook@sandia.gov

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

GPUs are now a fundamental accelerator for many high-performance computing applications. They are viewed by many as a technology facilitator for the surge in fields like machine learning and Convolutional Neural Networks. To deliver the best performance on a GPU, we need to create monitoring tools to ensure that we optimize the code to get the most performance and efficiency out of a GPU. Since NVIDIA GPUs are currently the most commonly implemented in HPC applications and systems, NVIDIA tools are the solution for performance monitoring. The Light-Weight Distributed Metric System (LDMS) [1] at Sandia is an infrastructure widely adopted for large-scale systems and application monitoring. Sandia has developed CPU application monitoring capability within LDMS. Therefore, we chose to develop a GPU monitoring capability within the same framework. In this report, we discuss the current limitations in the NVIDIA monitoring tools, how we overcame such limitations, and present an overview of the tool we built to monitor GPU performance in LDMS and its capabilities. Also, we discuss our current validation results. Most of the performance counter results are the same in both vendor tools and our tool when using LDMS to collect these results. Furthermore, our tool provides these statistics during the entire runtime of the tool as a time series and not just aggregate statistics at the end of the application run. This allows the user to see the progress of the behavior of the applications during their lifetime.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1. INTRODUCTION

GPUs deliver advanced computational power compared to CPUs. This is the reason GPUs have become an essential part of high-performance computers around the globe [2]. Although NVIDIA has led this field in the past, we are now seeing other GPU technologies emerge throughout the HPC system market. Nevertheless, because this new technology is emerging, NVIDIA is still seen as the leader in GPU accelerators. NVIDIA GPUs have thousands of computational CUDA cores that are backed by memory in the range of tens of gigabytes [3]. GPUs can outperform most other processing units in well-written GPU multi-threaded applications, especially if the data movement overhead is managed and minimized.

For a GPU application developer, the application must be optimized to use the CUDA cores and the memory without bottlenecks such as memory or branch divergence [4]. In order to achieve such a goal, GPU performance tools were introduced by vendors. At first, NVIDIA built its GPUs with hardware counters that only track events such as power and temperature. Later, events such as the number of instructions executed, the number of clock cycles, memory used, and other events were added to monitor performance during application execution on a GPU. Whenever a newer NVIDIA GPU is released, the number of counters increases since more events are monitored due to the hardware's increased complexity.

NVIDIA tools have limitations. Some of the tools can only report the results at the end of an application or kernel execution. Others need to instrument the CUDA source code itself to allow the activation and usage of the performance counters during the application run.

Our tool (LDMS-GPU) is implemented as a plugin in LDMS. LDMS provides scalable, lightweight monitoring of high-performance systems and applications. It can collect data at a user-defined frequency, is lightweight in its usage of system resources, it adds insignificant time overhead to the application and can easily be run in the background collecting data without instrumentation of the application.

In large multi-node systems, LDMS can run on a separate node monitoring and collecting data. LDMS has many capabilities. For example, it can store the data collected in different formats, and there is a back-end analysis layer and database through which data analysis and storage/retrieval can be done. It also has support to use Grafana for display of either real-time or post-processed data. LDMS comprises several plugin samplers (e.g., meminfo, procstat, Lustre, network (IB and Aries)), including a PAPI-based CPU sampler that uses the PAPI API [5] to read the hardware counters on a node. A configuration file identifies which metrics and events to collect on which node(s). It also defines the frequency of reading the counters.

LDMS collects data in the background with minimal overhead to the monitored applications. Furthermore, since LDMS is used to collect CPU performance data in HPC systems, we aim to have an integrated monitoring service that monitors the CPU host application as well as the application portion offloaded to the GPU. Therefore, we added to LDMS an NVIDIA GPU sampler plugin that can read the GPU performance counters that are specified by the user and collected during application execution. We can monitor GPI performance counters without instrumenting the CUDA source code of the application.

2. IMPLEMENTATION METHODOLOGY

In this section, we discuss the technical implementation details of the LDMS-GPU sampler. We outline the challenges we have faced and the limitations of the existing tools. We also discuss how we overcame these challenges, and explain in some detail our implementation methodology.

2.1. Challenges & Limitations of Existing Tools

This section discusses the various implementation hurdles we faced during our LDMS GPU sampler development. We describe our experience with the known existing tools that were considered initially as potential candidates for adoption for our purposes.

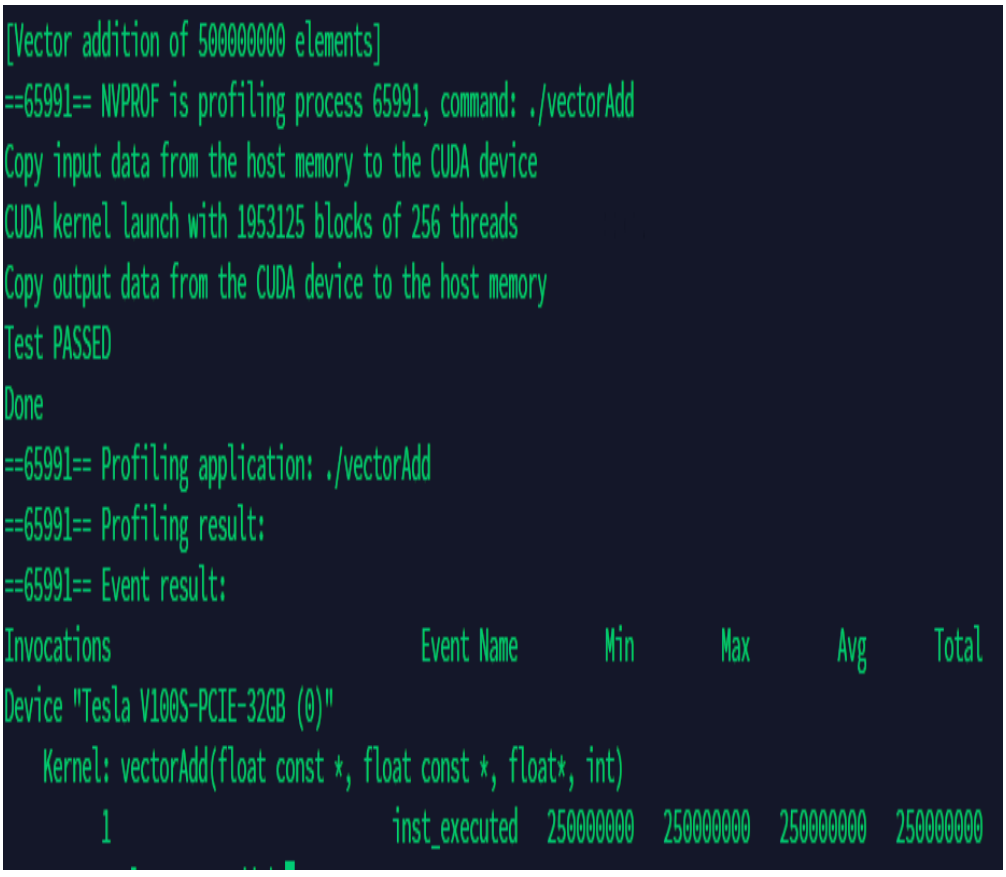

Figure 2-1 Example of NVPROF Output.

PAPI. Initially, we started exploring the PAPI-CUDA component at the start of tool development. The driver here is that the LDMS CPU sampler was already implemented using the PAPI [5] cpu component. Additionally, the PAPI-CUDA component can read events of interest, such as the number of instructions and clock cycles and other similar events. However, after trying to use it, we discovered that it would require instrumenting the source code of the GPU application to facilitate collecting the needed events and metrics due to an implementation detail that will

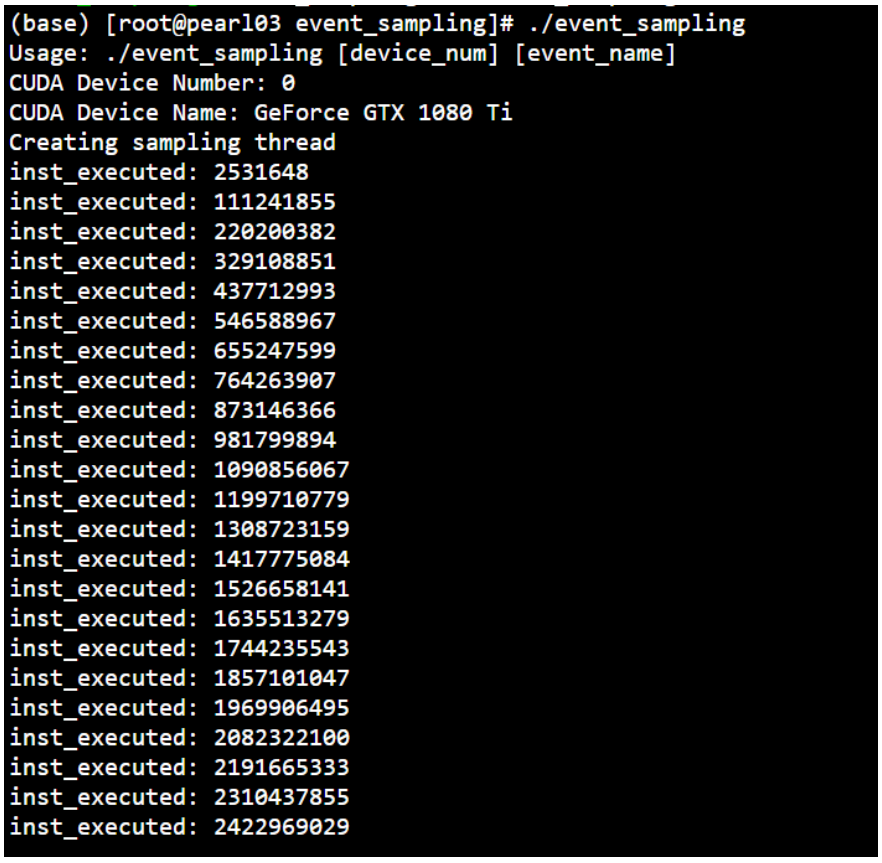

Figure 2-2 CUPTI Output Example

potentially be fixed in future releases of the PAPI component. Our goal is to run LDMS in the background to collect the data without any code instrumentation.

NVPROF. Then, we looked into existing NVIDIA performance analysis tools. First, we started to investigate NVPROF [6]. Unfortunately, NVPROF provides an aggregate reading of the performance counters at the end of the running application/kernel execution. Even with that limitation, it is hard to integrate NVPROF with LDMS since NVPROF often reruns the application multiple times on the GPU to provide readings for some of the counters. Thus, we decided not to consider NVPROF any further.

Figure 2-1 shows an example run for NVPROF. We are running the VECTORADD kernel. NVPROF runs the kernel, collects the statistics, and then dumps the statistics at the end of the run in aggregate. In this particular example, NVPROF prints out the total number of instructions executed. Seeing an aggregate total at the end of the a run is not what we are looking for. We want to see the progression of the performance as a function of the execution time. This is a feature NVPROF cannot provide. Since NVPROF does collect the statistics we are interested in, we wanted to figure out what NVPROF uses in the backend to read the performance counters.

NVML. NVPROF uses both NVML [7] and CUPTI [8] to read performance counters. Unfortunately, NVML can only read power and temperature related events and counters. The feature that we liked about NVML is that it does not instrument the application's code to read the

counters. We had to eliminate NVML from consideration, primarily because it did not collect events of interest. We will probably investigate NVML further in the future since power and temperature statistics are useful but not a priority at the moment.

CUPTI. The other possibility is CUPTI. CUPTI can read specific performance counters for any application, such as instructions executed. CUPTI unlike NVML requires modifying the source code of the kernel to be monitored. One of our goals is to be able to monitor applications where we do not necessarily have access to their source code. Thus, we opt for solutions where we would not need to modify/instrument any code.

Figure 2-2 shows that we can collect the "inst_executed" while the application is running. To the best of our knowledge, CUPTI is the only backend tool that we can use to collect the statistics we want and we can collect these statistics during rum-time of the application. The issue with CUPTI is that it would require us to modify the source code of the kernel/application we are interested in monitoring. In the next section, we will detail how we have managed to work with CUPTI and get around the issue of having to instrument the application's code.

2.2. Working with CUPTI

NVPROF was not a viable tool to work with for our current purposes since it only provided results at the end of an application/kernel run. Therefore, we abandoned NVPROF and focused on using CUPTI. However, CUPTI itself had its issues. We needed to instrument the source code of the application to allow CUPTI to collect the needed events/metrics. For very large scientific codes, with millions of lines of code and complex build systems, instrumenting the code is not an option.

Before we get deeper in explaining how we solved the CUPTI instrumentation issue, we need to introduce several concepts and terminology for the GPU. First, we need to explain what a GPU context is as it is a key aspect in our solution. A GPU context is analogous to a CPU process that encapsulates the current state of the executing GPU application. Thus, if we have multiple processes running on a GPU then every process creates a context before starting any activity on the GPU.

We created a library with the necessary CUPTI initialization. This is detailed in Algorithm 1 and will be explained in detail in Section 2.4). This CUPTI library gets the set of events and metrics to collect from a meta-data file where the user specifies the events and counters the tool should collect.

Since our goal is not to instrument the application's source code, we intercept every context and attach a CUPTI counter reader to it. Then, we initiate one CPU thread for every context to collect the counters and share their readings with LDMS. Additionally, we intercept the kernel launch and determine the association between each kernel and its context. Thus, we ensure that we have full details about the statistics collecting and the association of the events/statistics with a GPU kernel and context.

More elaboration on the implementation details will be in Section 2.4.

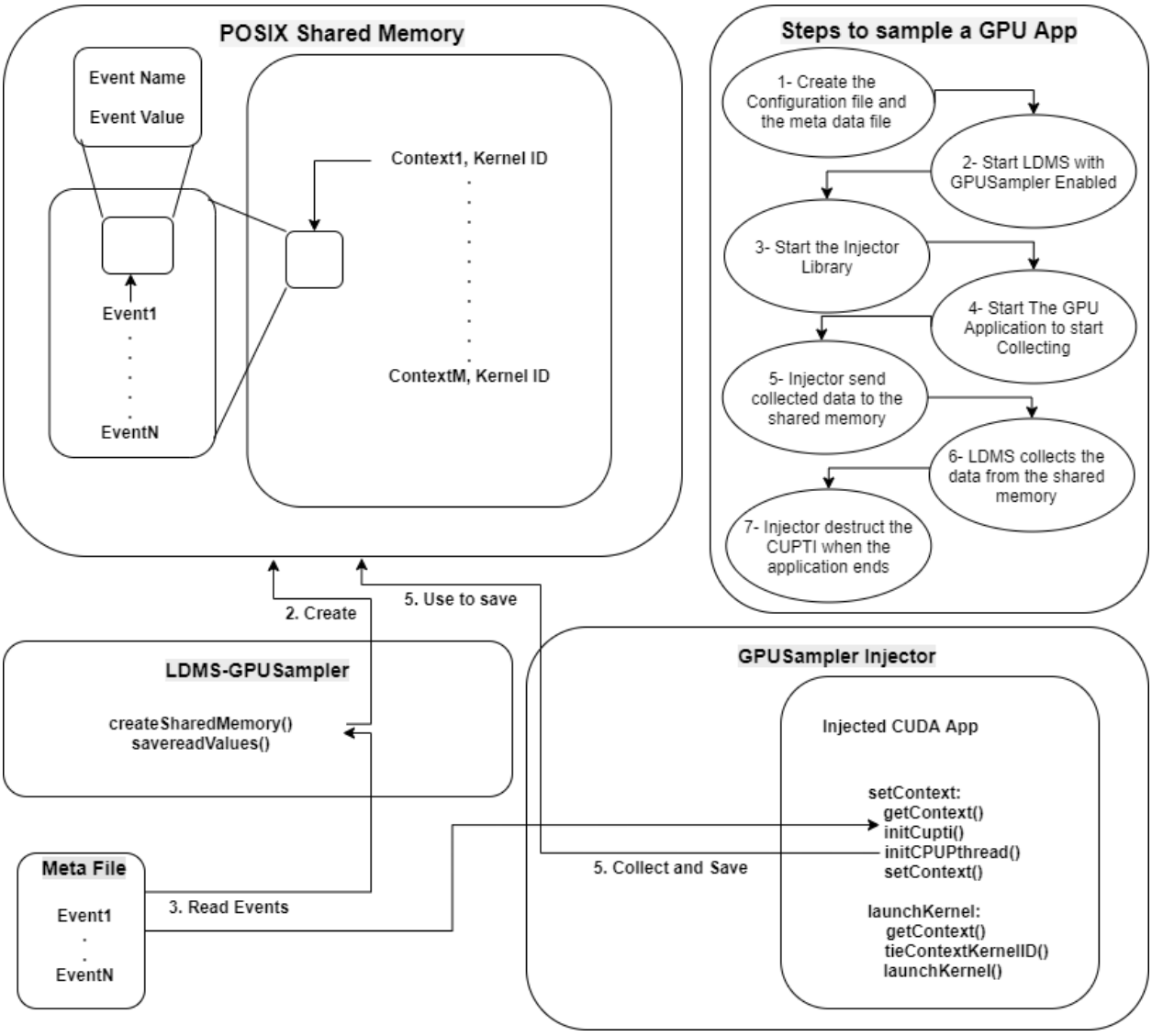

Figure 2-3 Sampler Implementation

2.3. The LDMS side of the Sampler

LDMS creates a POSIX shared memory space that communicates with our CUPTI sampler and waits for the sampler to populate the shared memory with readings. At the end of the application run, the shared memory space is un-linked after LDMS has saved the readings, and CUPTI destructs itself. Figure 2-3 captures the implementation details.

LDMS keeps the data saved as specified in an LDMS configuration file. We use plugin and sampler interchangeably. To run any LDMS plugin, the developer needs to have a configuration file to state which node(s) are being used as well as which plugins will run and what data it will save.

2.4. Implementation Details

In this section, we will explain in greater detail the development of our LDMS GPUSampler.

Algorithm 1 shows the code that we need to connect to the original application's code to initialize and run CUPTI so that it can collect the needed metrics/statistics. Note that we implement this code as a library and in conjunction with LD_PRELOAD (as explained below), we avoid any changes to the application's source code or build process. Below we explain the steps in Algorithm 1:

- Line 1 gets the current created context.
- Line 2 sets the event name.
- Lines 4 – 6 set the metric collection mode for the current context. There are two collection modes for CUPTI, kernel mode and continuous mode. In Kernel mode, CUPTI collects events for a specific kernel only and it serializes, in a CUDA Stream [9], all the parallel activities that happen in every context. While in continuous mode, CUPTI collects for everything running on the GPU, i.e., it collects for all the kernels/contexts running/existing on the GPU and it keeps the application running normally, in parallel. We set CUPTI to work in continuous mode, as we need to run the GPU application normally while collecting the data.
- Lines 8 and 9 create a group of events and tie them to the context.
- Lines 11 and 12 get the IDs for the events to be collected.
- Lines 14 and 15 add these IDs to the created event group.
- Lines 17 – 20 specifies that the event group will be collected for all the instances in the context.
- Lines 22 and 23 enable the event group to start collecting.

To get around having to instrument the application code, we use "LD_PRELOAD" [10] to initialize and instantiate our plugin and tie it to every application running in the system. We will call this the "Injector". Furthermore, we used the CUDA Driver API [11] to inject the CUPTI initialization into every context. The CUDA Driver API provides API calls that allow us to inject a function that can set the context on the device/GPU. The CUDA Driver API allows us to intercept the context set so we can run the CUPTI initialization code for every context as the setContext() function is called whenever the GPU changes the context it is working on. Therefore, we have effectively initialized our CUPTI plugin library and attached it to each context in the application. To differentiate which kernel runs on which context, we use the CUDA Runtime API [12] to intercept the launch of every kernel, which calls a function to identify the context associated with the kernel being launched. To pass the readings of the counters from our preloaded library to LDMS, we use a POSIX shared memory [13] space instantiated by LDMS to share the data using a common structure we defined in both the tool and LDMS as outlined in Figure 2-3.

Algorithms 2 and 3 show how we structure our POSIX shared memory and how we use it. The id represents the context id, and the name is the event name to collect. Each reading of every event is

an unsigned long long. Furthermore, we identify the size of our shared memory, and we instantiate the shared memory making it ready for writing the statistics to be collected by the tool.

Algorithms 4 and 5 show the interception of the context using the CUDA Driver API and the interception of the kernel launch (launchKernel()) using the CUDA Runtime API, respectively. Algorithm 4 uses the CUDA Driver API to intercept the context to attach CUPTI to it the same way as in Algorithm 1. We can initialize CUPTI for every context and determine if the context is created already or not to prevent re-initializing CUPTI multiple times for the same context. At the end of Algorithm 4 (line 28), we initiate a separate CPU thread to read the metrics we need to collect for every context. Algorithm 5 intercepts the CUDA launch kernel to have an association between the context and the kernel running on it. We save the context and its associated kernel in the shared memory to determine which context is running a particular kernel.

Multi-kernels running on a single GPU are challenging since we need to identify which GPU context and which kernel we are collecting for, i.e., we need to attribute performance events to a specific kernel running on a specific GPU context. To handle multi-kernels, we use the CUDA Driver API [11] and the CUDA Runtime API [12], to read the context and kernel IDs to attach the counters to them, respectively. Then, we make sure to include context and kernel IDs in the shared memory structure shared with LDMS. Thus, we can easily differentiate between collected statistics for the various contexts and kernels. This way, we have resolved any concerns in monitoring multi-kernel GPU Applications.

We created a meta-data file for the events and metrics, where the developer/user can specify which events and metrics should be collected, and then this meta-data file is read by both our library and the LDMS plugin.

2.5. Implementation of a Multi-Node Multi-GPU CUPTI Injector

In this section, we introduce how our implementation supports multi-GPU configurations.

If a CUDA application will run on multiple nodes that have multiple GPUs, then MPI is often used in conjunction with CUDA. Most MPI CUDA applications run in multiple contexts. Thus, we want to collect the performance counters and differentiate among them by the contexts. Also, for multiple GPUs running a single application cooperatively in parallel, it is challenging to get different counters on different GPUs and differentiate among them. Sometimes developers tend to have multiple kernels in the same application, and they would need to collect counters for every kernel separately. Therefore, since every context is tied to a GPU, we can differentiate the GPUs easily.

This is done through the information we already collected. Recall that, we have intercepted the CUDA launch kernel call through the CUDA Runtime API. Thus, we know which GPUs are running which kernels. Also, we intercepted the setContext() call using the CUDA Driver API. Thus, we know which context is running on which GPU. This way we have tied the context, the kernel and the GPU information together for proper tagging of the statistics.

Algorithm 1 CUPTI Code: CUPTI Initialization

```
cuGetCurrentCtx(&context);
eventName = "inst_executed";

cuptiErr = cuptiSetEventCollectionMode(context,
    CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS);
CHECK_CUPTI_ERROR(cuptiErr, "cuptiSetEventCollectionMode");

cuptiErr = cuptiEventGroupCreate(context, &eventGroup, 0);
CHECK_CUPTI_ERROR(cuptiErr, "cuptiEventGroupCreate");

cuptiErr = cuptiEventGetIdFromName(device, eventName, &eventId);
CHECK_CUPTI_ERROR(cuptiErr, "cuptiEventGetIdFromName");

cuptiErr = cuptiEventGroupAddEvent(eventGroup, eventId);
CHECK_CUPTI_ERROR(cuptiErr, "cuptiEventGroupAddEvent");

cuptiErr = cuptiEventGroupSetAttribute(eventGroup,
    CUPTI_EVENT_GROUP_ATTR_PROFILE_ALL_DOMAIN_INSTANCES,
    sizeof(profile_all), &profile_all);
CHECK_CUPTI_ERROR(cuptiErr, "cuptiEventGroupSetAttribute");

cuptiErr = cuptiEventGroupEnable(eventGroup);
CHECK_CUPTI_ERROR(cuptiErr, "cuptiEventGroupEnable");
```

Algorithm 2 POSIX Shared Memory Structure

```
struct memoryPosix {
    char id[10][100];
    char name[100][100];
    unsigned long long details[100][10];
};
```

2.6. Running the Sampler

To use the tool, the events and metrics need to be included in the meta-data file. Then, the LDMS GPUSampler needs to get configured with the path to the meta-data file. Afterwards, we need to start the LDMS sampler, the LD_PRELOAD tool library, and finally run the CUDA application. The results will be collected into an LDMS store as defined by the user in the configuration file.

Figure 2-3 shows the process of running the sampler. The process begins with starting LDMS through the GPUSampler plugin. Then, we start the injector by LD_PRELOAD. CUPTI will initialize for every CUDA application that is running on the system. Next, we start the CUDA

Algorithm 3 Setting and Initializing the POSIX Shared Memory

```
int pid = getpid();

const int SIZE = 16384;
int shm_fd;
memoryPosix* shmd;
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
ftruncate(shm_fd, SIZE);
shmd = (struct memoryPosix*)mmap(0, SIZE, PROT_WRITE, MAP_SHARED,
                                 shm_fd, 0);
```

application, and then the injector will read the events. The events will be saved to the shared memory structure. The injector will wait until the application ends, and then it will kill the CUPTI initializer.

Algorithm 4 Intercepting Context Set CUDA API Function

```
#define CU_HOOK_GENERATE_INTERCEPT_SET_CONTEXT(hooksymbol,          \
funcname, params, ...)                                              \
CUresult CUDAAPI funcname params                                    \
{                                                                   \
    static void* real_func = (void*)real_dlsym(RTLD_NEXT,           \
    CUDA_SYMBOL_STRING(funcname));                                  \
    CUresult result = CUDA_SUCCESS;                                 \
    cuCtxGetDevice(&device);                                        \
    if(!devCon[ctx]){                                               \
        cuCtxCreate(&ctx, CURRENT_FLAGS, device);                   \
        devCon.insert({ctx, device});                               \
        conI.con = ctx;                                             \
        cudaDeviceReset();                                          \
        CUPTI_CALL(cuptiSetEventCollectionMode(ctx,                 \
        CUPTI_EVENT_COLLECTION_MODE_CONTINUOUS));                   \
        CUPTI_CALL(cuptiEventGroupCreate(ctx,                       \
        &eventGroup[ctx], 0));                                      \
        for(int i = 0; i < Events_META_FILE; i++){                  \
            CUPTI_CALL(cuptiEventGetIdFromName(device,              \
            eventNames[i], &(eventIds[i])));                        \
            CUPTI_CALL(cuptiEventGroupAddEvent(eventGroup[ctx],     \
            eventIds[i]));                                          \
        }                                                           \
        CUPTI_CALL(cuptiEventGroupSetAttribute(eventGroup[ctx],     \
        CUPTI_EVENT_GROUP_ATTR_PROFILE_ALL_DOMAIN_INSTANCES,        \
        sizeof(profile_all), &profile_all));                        \
        CUPTI_CALL(cuptiEventGroupEnable(eventGroup[ctx]));         \
        status = pthread_create(&pThread, NULL,                     \
        sampling_func, ctx);                                        \
        if (status != 0) {                                          \
            perror("pthread_create");                               \
            exit(-1);                                               \
        }                                                           \
    }                                                               \
    result =                                                        \
    ((CUresult CUDAAPI (*)params)real_func)(__VA_ARGS__);           \
    return (result);                                                \
}                                                                   \

CU_HOOK_GENERATE_INTERCEPT_SET_CONTEXT(CU_HOOK_CTX_SET_CURRENT,
cuCtxSetCurrent, (CUcontext ctx), ctx)
```

Algorithm 5 Intercepting Launch Kernel CUDA Driver Function

```
#define CU_HOOK_GENERATE_INTERCEPT_LAUNCH_KERNEL(hooksymbol,      \
funcname, params, ...)                                            \
CUresult CUDAAPI funcname params                                  \
{                                                                 \
    static void* real_func =                                      \
    (void*)real_dlsym(RTLD_NEXT, CUDA_SYMBOL_STRING(funcname));   \
    CUresult result =                                             \
    ((CUresult CUDAAPI (*)params)real_func)(__VA_ARGS__);         \
    CUcontext ctx;                                                \
    cuCtxGetCurrent(&ctx);                                        \
    if(!kerCon[ctx]){                                             \
        kerCon.insert({ctx, f});                                  \
    }                                                             \
    return (result);                                              \
}                                                                 \

CU_HOOK_GENERATE_INTERCEPT_LAUNCH_KERNEL(CU_HOOK_LAUNCH_KERNEL,
cuLaunchKernel, ( CUfunction f, unsigned int gridDimX,
unsigned int gridDimY, unsigned int gridDimZ,
unsigned int blockDimX, unsigned int blockDimY,
unsigned int blockDimZ, unsigned int sharedMemBytes,
CUstream hStream, void** kernelParams, void** extra), f,
gridDimX, gridDimY, gridDimZ, blockDimX, blockDimY, blockDimZ,
sharedMemBytes, hStream, kernelParams, extra)
```

3. EXPERIMENTAL RESULTS

In this section, we discuss our experimental results that span two targets. First, we want to verify that our collection methods through CUPTI are picking up the correct value for the statistics compared to the NVIDIA tools such as NVPROF or NSight. Second, we want to make sure that the overhead of our injector and library adds negligible overhead to the runtime for GPU applications being monitored. Obviously, this is a work in progress and we need to verify our work using a large suite of applications, benchmarks, and use cases. But for the purposes of this report, we have only used two kernels, VectorAdd [14] and BFS [15] to verify the correctness of the results from our injector compared to the vendor tools. We have also used LULESH to show the runtime overhead of running the application with compared to running the application without our monitoring [16].

Table 3-1 shows the results from the VectorAdd kernel. Table 3-2 shows the results of BFS. Tables 3-1 and 3-2 report the ratio between the GPU sampler results and the NVPROF results. When the ratio is close to one, the GPU sampler results are almost the same as the NVPROF results. We collect the events ten times and take the average. We compare the averaged result to the NVPROF results for the same events without the sampler. As shown, the accuracy of our sampler compared to NVPROF is high.

Table 3-1 VectorAdd CUDA APP: Events collected by GPUSampler-LDMS and NVPROF

Event Name	*Ratio*	*GPUSampler-LDMS*	*NVPROF*
inst_executed	1	281250000	281250000
active_cycles	1.00005	366639640	366621830
inst_issued	1.00041	229122701	229028849
thread_inst_executed	1	9000000000	9000000000
warps_launched	1	15625000	15625000

Table 3-2 Rodinia BFS Benchmark: Events collected by GPUSampler-LDMS and NVPROF

Event Name	*Ratio*	*GPUSampler-LDMS*	*NVPROF*
inst_executed	1	375187	375187
active_cycles	1.01894	350547	357185
inst_issued	1	439578	439578
thread_inst$_e$xecuted	1	12002790	12002790
warps_launched	1	31264	31264

Figure 3-1 shows the LULESH-CUDA proxy application [16] results with and without our sampler on eight nodes. We use NVPROF to measure the execution time of application functions during execution. We calculate the average runtimes on the eight nodes; the resulting ratio is close to one. This indicates that the sampler time overhead is negligible.

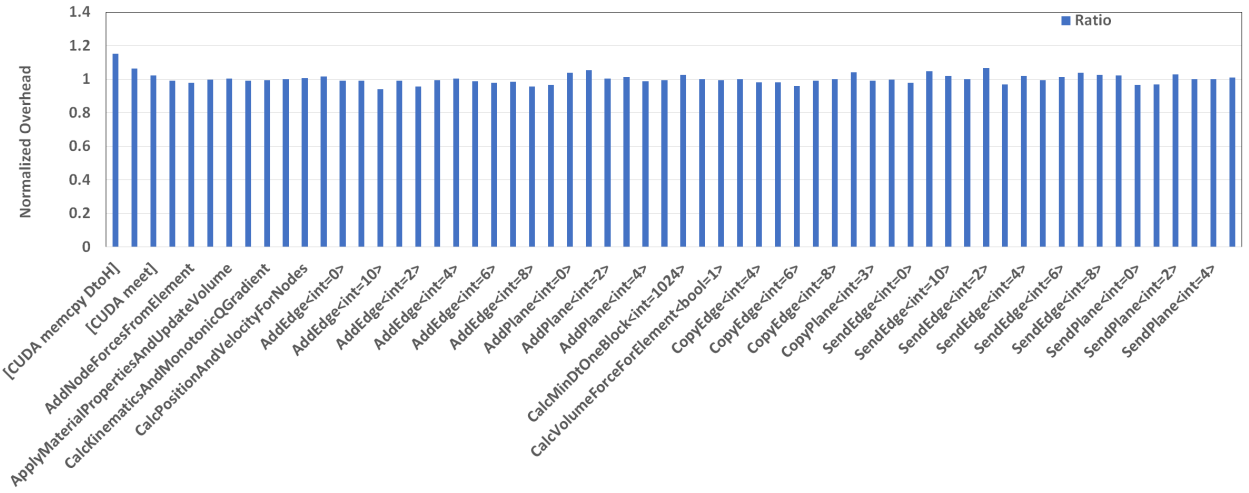

Figure 3-1 The ratio between running LULESH, a proxy application, with and without the injector.

4. CONCLUSION & FUTURE WORK

LDMS-GPU is an LDMS plugin for NVIDIA GPUs. The tool can monitor any CUDA application with minimal overhead and high accuracy. We have overcome a lot of problems to provide continuous monitoring of the events the performance counters can provide. We have adopted various methodologies to overcome the vendor tools' limitations, giving more freedom to the developers to get all the metrics and events they need while the application is running. Although this sampler uses the CUPTI library, the main limitation of CUPTI has been removed because of the injector.

We need to continue testing of LDMS-GPU with many more benchmarks and under various system configurations. However, with our limited testing thus far we see that the LDMS GPU sampler is lightweight, similar to LDMS. It provides a non-noticeable time overhead and thus shows negligible impact on the running applications. We can run the sampler all the time in the background, monitoring all the GPU CUDA applications.

For future work, we are looking forward to doing more testing for the tool. We want to test longer running benchmarks and more complicated scenarios like Multi-GPUs on the same system and distributed GPUs in a cluster. Also, we want to include the NVML library events and metrics to monitor power and thermals. We also want to show the kernel name instead of the kernel identifier, which will take some research and extension.

In our initial investigations, we looked into using PAPI [5] to monitor the GPU kernels, but unfortunately, an issue in the PAPI CUDA component prevented us from using it for the GPU sampler. We are working with the PAPI team to rectify this. We believe that a GPU sampler based on PAPI will be more robust in the future. Currently, it also depends on CUPTI, but PAPI is actively under development and will likely evolve to support non-NVIDIA GPUs in the future.

REFERENCES

[1] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[2] TOP500. (2020). [Online]. Available: https://www.top500.org/

[3] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[4] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," *IEEE Transactions on Multimedia*, vol. 15, no. 2, pp. 279–290, 2012.

[5] Performance Application Programming Interface (PAPI). (2019) Version 5.7. [Online]. Available: https://icl.utk.edu/papi

[6] NVIDIA Corporation. (2020) Visual profiler (nvprof). [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide

[7] ——. (2020) Nvidia management library (nvml). [Online]. Available: https://docs.nvidia.com/deploy/nvml-api

[8] NVIDIA Corporation. (2020) Cuda profiling tools interface (cupti). [Online]. Available: https://docs.nvidia.com/cupti/Cupti/index.html

[9] Nvidia CUPTI Documentation. (2020) Nvidia docs. [Online]. Available: https://docs.nvidia.com/cupti/Cupti/r_main.html#r_overhead_profiling

[10] Linux Programmer's Manual. (2020). [Online]. Available: http://man7.org/linux/man-pages/man8/ld.so.8.html

[11] NVIDIA Corporation. (2020) Cuda driver api. [Online]. Available: https://docs.nvidia.com/cuda/cuda-driver-api/index.html

[12] NVIDIA Corporation. (2020) Cuda runtime api. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[13] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. Wiley Publishing, 2012.

[14] CUDA Samples. (2020). [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/index.html

[15] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 303–314. [Online]. Available: https://doi.org/10.1145/1810479.1810534

[16] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

DISTRIBUTION

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	Technical Library	9536	MS-0899

Email—Internal [redacted]

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.