CSCI 415 Project Report

Holland Schutte

June 16, 2019

0.1 Overview

The assignment for CS415 was to provide the basis for a performance analysis tool which extends well beyond what can be found on GitHub or some other public source repository website. Using standard practices found in libraries such as pdump or OpenMP, the goal is to allow the user a high degree of flexibility while eliminating the need for them to alter or rewrite their code as much as possible. Through this interface the user is provided access to standard event counters found in Nvidia GPUs, as well as a suite of metrics which can be derived from these event counters.

Examples of these kinds of metrics range from basic values such as overall branching efficiency, the total amount of control flow instructions used, read and write throughput values for device ram; to more granular measurements like the throughput of interaction between error correction code RAM, L2 cache, and DRAM. Another example is the average amount of replays due to global memory cache misses (accounted for every executed instruction).

A complete list of metrics can be found at
https://docs.nvidia.com/cupti/Cupti/r_main.html#metrics-reference

0.2 Basic usage

0.2.1 Function calls

Listing 1: Benchmark example

```
__global__ void my_device_compute_kernel(...) {
  const int thread_index = /* compute thread index expression */;

  nvcd_device_begin(thread_index);

  //
  // Your device kernel execution code here
  //

  nvcd_device_end(thread_index);
}

__host__ void my_host_kernel_client(...) {
  dim3 dimGrid(gridWidth, gridHeight, gridLength);

  dim3 dimBlock(blockWidth, blockHeight, blockLength);

  int num_threads = dimGrid.x * dimGrid.y * dimGrid.z * dimBlock.x * dimBlock.y * di

  nvcd_host_begin(num_threads);

  my_device_compute_kernel<<<dimGrid, dimBlock>>>(...);

  nvcd_host_end();
}

int main(...) {
  nvcd_init();

  ...

  my_host_kernel_client(...)

  ...

  nvcd_terminate();
```

```
    return exit_code;
}
```

nvcd_device_begin() and nvcd_device_end() are both used to report per-thread information such as runtime, streaming multiprocessor (SM) the thread was executed on, etc.

nvcd_host_begin() and nvcd_host_end() are used to allocate device memory as well as cpu-side CUPTI metadata that's used for tracking various event counters and metrics.

Note that nvcd_init() and nvcd_terminate() only need to be called once during the execution of the application. nvcd_host_begin() and nvcd_host_end() can be called in multiple places. As it stands, the host begin function will currently reallocate all needed memory for computing the desired metrics/event counters for the amount of threads passed. Naturally, the host end function will free said memory.

Eventually, improvements on these capture methods will be provided, but currently the desired metrics and counts are reported to stdout once nvcd_host_end() is called.

0.2.2 Environment variables

Listing 2: Specifying metrics and event counters through environment variables

```
export BENCH_EVENTS="l1_global_load_miss:l2_subp0_write_sector_misses:l2_subp0_read_sector_miss
export BENCH_METRICS="branch_efficiency:sm_efficiency:ipc:l1_cache_global_hit_rate:dram_read_th
```

Desired metrics and events are specified through bash environment variables. These settings will apply to all calls to the nvcd functions. The user may also change the prefix from 'BENCH' to anything else they wish.

0.3 Building

0.3.1 Pre-requisites

- g++ 4.8 or above, for C++11 support.
- gcc 4.8 or above, for c99 + gnu extensions (i.e., -std=gnu99) support.
- A variant of CUDA 9.
- A Linux installation which is reasonably up to date (within the last few years).

0.3.2 Environment variables

1. Define necessary environment variables

   These include paths such as where CUPTI's main directory is located, the root of the NVCD directory itself, and target architecutres to compile cuda code for. What follows is an example script designed to setup the environment needed for building the library and its auxilery files.

Listing 3: XSEDE compute cluster build environment setup

```
#!/bin/bash

cuda_version=9.2

export NVCD_HOME=$HOME/cs415/Project
export NVCD_BIN_PATH=$NVCD_HOME/bin
export NVCD_INCLUDE_PATH=$NVCD_HOME/include

export LD_LIBRARY_PATH=$NVCD_BIN_PATH:$LD_LIBRARY_PATH

module purge
```

```
module load gnutools
module load cuda/$cuda_version
module load python/2.7.10
module load pgi
module load mkl/11.1.2.144
module load mvapich2_gdr/2.1
module load openmpi_ib/1.8.4
module load gnu/4.9.2

arch=

case $cuda_version in
8.0) arch=21 ;;

9.2) arch=37 ;;
esac

export CUDA_ARCH_SM=sm_${arch}
export CUDA_ARCH_COMPUTE=compute_${arch}

export CUDA_HOME=/usr/local/cuda-$cuda_version

export CUPTI_INCLUDE=$CUDA_HOME/extras/CUPTI/include
export CUPTI_LIB=$CUDA_HOME/extras/CUPTI/lib64

export PATH=$CUDA_HOME/bin:$PATH
export LD_LIBRARY_PATH=$CUPTI_LIB:$LD_LIBRARY_PATH

export JOB_OUTPUT_DIR=$NVCD_HOME/xsede-scripts/job-output


mkdir -p $JOB_OUTPUT_DIR
```

Out of these, the ones which are absolutely required are:

- CUDA_HOME
- CUDA_ARCH_SM
- CUDA_ARCH_COMPUTE

LD_LIBRARY_PATH may need to include libnvcd.so in its path as well.

0.3.3 Building

Simply a standard 'make libnvcd.so' can be invoked. Currently, compiler optimizations are mostly disabled to make verification and correctness easier to manage. Both debug and release builds will be provided, with O2 being the default level for the release build (unless something else is desired).

0.4 Architecture

Some quick definitions/clarifications:

- 'major' event counters/metrics: the event counters or metrics specified by the user via environment variables.
- 'minor' event counters: event counters which exist as a dependency to something else - each metric, for example, relies on its own specific set of event counters; many of these event counters may have not been specifically requested by the user.

- ‘gcc‘ is used in a few places throughout this section. For this section, it’s worth clarifying that gcc may also refer indirectly to g++ - it all depends on the context.

  The differences between g++ and gcc are significant enough to warrant such notes, and it also explains how NVCC will behave under different kinds of configurations (this is outside the scope of this document, but further elaboration can be provided upon request).

There are a few major characteristics to how the library is organized. In particular,

- The library is broken into two components: a header-only portion and a shared object which the user’s binary is directly linked against.

  This is mainly because NVCC doesn’t actually have a linker for device-specific source code. So, any CUDA ‘__device__‘ functions, data structures, or variables cannot be referred to at the object code level - portability can only be achieved through the use of source file sharing. Preprocessor directives - similar to how standard header include guards are used, but slightly different - are used to ensure that only single instances of global variables (for both host and device variables) are created in the event that the same header file is referred to across different translation units.

- NVCC is the compiler used for all source code, both cuda specific and non-cuda specific.
  - C++ support is necessary, since cuda source files are C++ - not C.
  - NVCC acts as a proxy for gcc; gcc-specific arguments are passed using a specific flag ‘–compiler-options‘, which forwards the arguments as a complete string. This is used in many areas, and can be used multiple times throughout the same command.
  - NVCC has its own set of nvcc-specific flags, some of which refer to features similar to what is available by gcc.
  - For .cu files, nvcc -std=c++11 must be passed directly to nvcc, but not through the string provided via ‘–compiler-options‘.
- CUPTI event data is organized in the following manner:
  - An instance of type cupti_event_data_t, which refers to major event counter data and a list of different major metrics.
  - Each major metric refers to its own unique instance of cupti_event_data_t, which in turn holds data references to event counters that the corresponding metric requires.
  - Each event counter belongs to a specific domain category.
  - Each event counter must be assigned to a specific group at runtime.
    - Groups are necessary for the event counters to actually be counted throughout the execution of a kernel.
    - A group of event counters cannot contain counters from different domains.
    - There is no guarantee that a group can contain all event counters from a domain. Often enough, multiple groups are necessary if an entire domain is to be covered.
    - Groups must be explicitly enabled and disabled by the CUPTI API before their counters may be recorded.
    - Not all groups can be enabled at once.
    - The implication: there’s a chance that multiple invocations of the kernel must be performed for all event counters or metrics to be recorded.
- A simple hooking mechanism is used to wrap around the user’s kernel invocation. This allows for the library to internally track which event counters have been computed, and then continuously invoke the same kernel if needed until the desired information has been acquired.