Profiling on GPU

Filippo Spiga, HPCS, University of Cambridge

<fs395@cam.ac.uk>

Demonstrating CUDA tools

CUDA SDK provides you useful tools to support GPU programming:

- nvprof: CUDA command-line profiler
- nvvp: CUDA visual profiler
- cuda-gdb: CUDA debugger GDB-style
- cuda-memcheck: functional correctness checking suite

Live demo and then you are free to play with provided examples and previous GPU code

Why Performance Measurement Tools?

- You can only improve what you measure
- Need to identify:
  - Hotspots: Which function takes most of the run time?
  - Bottlenecks: What limits the performance of the hotspots?
- Manual timing is tedious and error prone
  - Possible for small application like vectorAdd or matrix multiplication
  - Impractical for larger/more complex application
  - Access to hardware counters (PAPI, CUPTI)

What Limits Communication with the GPU?

- PCIe bus connects GPU to CPU/network
  - Gen 2 (Fermi): 8 GB/s in each direction
  - Gen 3 (Kepler): 16 GB/s in each direction
- Tesla GPUs have dual DMA engines
  - Two memcpys (in different streams, different directions)
  - Overlap with kernel and CPU execution

What Limits Kernel Performance?

- memory-bound : most of kernel time is spent in executing memory instructions
  - exploit shared memory, memory access coalescing, …
- compute-bound : most of operations are ALU-FPU instructions.
  - reduce branch divergence, interleave computation between ALU-FPU and SFU*, and provide enough independent instructions to exploiting ILP.
- Latency-bound : poor math and memory overlapping
  - You address memory first and then compute

* Units that execute transcendental instructions such as sin, cosine, reciprocal, and square root.

What can be captured…

- Events API
  - Sample hardware counters – hardware dependent
- Metrics API
  - Sample metrics (composition of hardware counters) – hardware independent
- Callbacks API
  - callbacks for CUDA runtime and driver API
  - callbacks for resource and synchronization events
- Activity API
  - recording of asynchronous accelerator activities (e.g. kernels and memory copies)
  - recording of CUDA API, context and device events and more

Command Line Profiler (nvprof)

Usage:

```
nvprof [options] [CUDA application] [applicationarguments]
```

Options:

- `--devices <device ids>`
- `--events <event names> (--query-events)`
- `--print-gpu-trace`
- `--print-api-trace`

Source correlation requires that source/line information be embedded in executable

- Available in debug executables: `nvcc -G`
- New flag for optimized executables: `nvcc -lineinf`

Command Line Profiler (nvprof)

The command line profiler is controlled by environment variables, and can be enabled by setting COMPUTE_PROFILE=1

COMPUTE_PROFILE_LOG: Specifies the filename for the output. Include %d in the filename if you use multiple contexts and %p if multiple processes run on the same host (default values is “cuda_profile_%d.log”).

COMPUTE_PROFILE_CSV: set to 1 to enable CSV output

COMPUTE_PROFILE_CONFIG: Specifies the path to a configuration file that specifies what information/counters to collect. For a list of configuration options please refer to the documentation included with the toolkit.

```
export COMPUTE_PROFILE_CSV=1
export COMPUTE_PROFILE_LOG=“profile_%p.csv”
export COMPUTE_PROFILE_CONFIG=profile.cfg
```

Sample configuration file profile.cfg

```
gpustarttimestamp
gridsize3d
threadblocksize
dynsmemperblock
stasmemperblock
regperthread
memtransfersize
memtransferdir
streamid
countermodeaggregate
active_warps
active_cycles
```

Interesting profile metrics

- instructions_issued, instructions_executed
  Both incremented by 1 per warp
  "issued" includes replays, "executed" does not
- gld_request, gst_request
  Incremented by 1 per warp for each load/store instruction
  Instruction may be counted if it is "predicated out"
- l1_global_load_miss, l1_global_load_hit,
  global_store_transaction Incremented by 1 per L1 line (line is 128B)
- uncached_global_load_transaction
  Incremented by 1 per group of 1, 2, 3, or 4 transactions
  Better to look at L2_read_request counter (incremented by 1 per 32B transaction; per GPU, not per SM)

Annotations: NVIDIA Tools Extension

Developer API for CPU code installed with CUDA Toolkit (libnvToolsExt.so) that allow to:

- Naming
  - Host OS threads: `nvtxNameOsThread()`
  - CUDA device, context, stream: `nvtxNameCudaStream()`
- Time Ranges and Markers
  - Range: `nvtxRangeStart(), nvtxRangeEnd()`
  - Instantaneous marker: `nvtxMark()`

Focus profiling on region of interest → Reduce volume of profile data → Improve usability of Visual Profiler & Improve accuracy of analysis

Profile “just a bit” …

1. Include `cuda_profiler_api.h`
2. Add functions to start and stop profile data collection.
   - `cudaProfilerStart()` is used to start profiling
   - `cudaProfilerStop()` is used to stop profiling
3. Instruct the profiling tool to disable profiling at the start of the application.
   - `nvprof --profile-from-start-off`
4. Flush profile data to reduce profiling overhead and perturbing application behavior
   - call `cudaDeviceReset()` before exiting. Doing so forces all buffered profile information to be flushed.
5. Select the metrics required to be displayed and analyse the application behavior
   - e.g. `nvprof --metrics flops_dp`

Useful nvprof metrics

```
nvprof --devices 0 --query-metrics
nvprof --metrics metric-1,metric-2,…
```

- **achieved_occupancy**: Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
- **sm_efficiency**: The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU
- **flop_sp_efficiency** / **flop_dp_efficiency** : Ratio of achieved to peak SP/DP floating-point operations
- **gst_throughput**/ **gld_throughput** : Global memory store/load throughput
- **gst_efficiency** / **gld_efficiency**: Ratio of requested global memory store/load throughput to required global memory load throughput. Values greater than 100% indicate that, on average, the store/load requests of multiple threads in a warp fetched from the same memory address
- **gst_transactions** / **gld_transactions** : Number of global memory load transactions

nvprof – Profile Data Import

Produce profile into a file

```
$ nvprof --analysis-metrics -o profile.out <app>
```

Import into Visual Profiler…

File menu -> Import nvprof Profile...

Import into nvprof to generate textual outputs

```
$ nvprof -i profile.out
$ nvprof -i profile.out --print-gpu-trace
$ nvprof -i profile.out --print-api-trace
```

(Little trick) Profiling parallel applications

Just make sure you do not overlap profile outputs…

```
mpirun –np <np> <mpi-args> \
 nvprof --output-profile $out.$rank <nvprof_args> \
   <app> <app-args>
```

out = name of the output file

rank = unique global ID (e.g. MPI rank)

- MV2_COMM_WORLD_RANK for MVAPICH2
- MPI_RANKID for Platform MPI
- OMPI_COMM_WORLD_RANK for OpenMPI
- PMP_RANK for Intel MPI

Note for the profiler

- Most counters are reported per Streaming Multiprocessor (SM), not entire GPU!
  - Few exceptions, e.g. L2 and DRAM counters
- A single run can only collect a few counters, Multiple runs are needed when profiling more counters
  - Done automatically by the Visual Profiler
  - Have to be done manually using command-line profiler
- Use CUPTI API to have your application collect signals on its own
  - Counter values may not be exactly the same for repeated runs
- Threadblocks and warps are scheduled at run-time
  - So, “two counters being equal” usually means “two counters within a small delta”

See the profiler documentation for more information!

(Little trick) Timing a kernel?

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);
kernel<<<grid, threads>>>(..);
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```