UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

Instrumentation Tutorial

Lorenz Braun, Sotirios Nikas, Chen Song, Holger Fröning

January 19, 2021

ZITI/EMCL, Heidelberg University

GPU Instrumentation and Profiling Primer

Motivation

Why should we model performance and power on GPUs?

- 7 of the top10 supercomputers from the green top500 list use NVIDIA GPUs (see: `https://www.top500.org/lists/green500/2020/11/`)
- Climate change requires HPC to optimize performance and power efficiency
- Hardware and software is constantly evolving as is the need for better understanding performance impact of the latest developments
- Profiling use cases:
  - Performance optimizations
  - Building prediction models for performance and power
  - Optimizing scheduling

Why do we need better prediction models and profiling methods?

Things to be Improved

- Very few public models on GPU performance exists
- Usually very few GPUs are supported
- Significant overhead for data collection
- Models often depend on metrics which can change with newer hardware generations

Currently Available Tools for Profiling

Hardware performance-counter based:

nvprof

- CUDA API trace
- Light to heavy performance impact
- Slowdown due to kernel replays

GPU simulators:

GPGPU-Sim, Multi2Sim, Barra

- Very detailed analyses possible
- Very slow
- Usually behind currently available hardware

Instrumentation based:

GPU Ocelot/Lynx,SASSI, NVBit (Research Prototype), *CUDA Flux*

- Custom profiling
- No hardware metrics such as cache hit-rate
- Fast, low overhead
- Longevity often limited

Our Approach

Our Goal: Portable Performance and Power Prediction

- Providing modeling methods which can be easily transfered to new devices
- Characterize applications and workloads
- Light-weight performance analysis
- Challenging restrictions, but increased *portability* and predictability

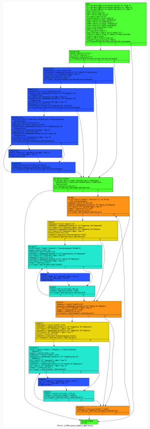

Figure 1: control flow graph colored by execution frequency

CUDA Flux - Code Block Based Profiling

- LLVM based profiler
- Profiling on PTX assembly level
- Instruments each basic block in the control flow graph of the kernel
- Basic block executions are counted which allows to derive the executed instructions

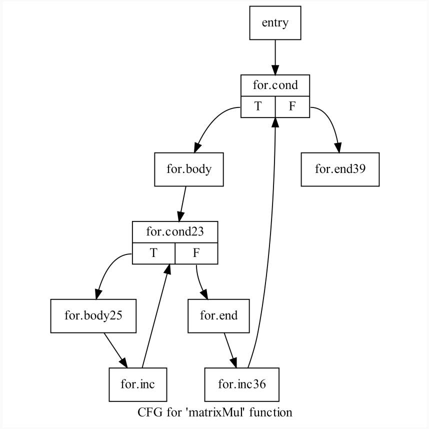

Figure 2: simple control flow graph of a CUDA kernel

Advantages/Disadvantages

+ Portability
+ Low overhead
– Less precision due to unknown dynamic hardware effects like e.g. caching
– Need to use clang++ instead of nvcc

Instrumentation for Performance Measurements

Instrumentation Target

Kernel execution time as perceived from application level is to be measured.

Picking a statistic to summarize the time measurements:

- Mean -> average execution time
- Median -> close to typical execution
- Distribution -> detailed info on all measured outcomes

Time distribution gives the most information, but is harder to make use of!

Factors for measurement accuracy:

- Timer resolution
- Correlation to begin and end of kernel

Instrumentation Implementation

Several options exist to measure execution time spent in CUDA kernels:

- CUDA Events

```
cudaEventRecord(start);
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);
```

- Profiler

```
nvprof ./saxpy
```

- CPU Timers

```
t1 = myCPUTimer();
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
cudaDeviceSynchronize();
t2 = myCPUTimer();
```

Constraints

- Only single kernel can be executed and measured at the same time
- Host code should be executed single threaded
- Multiple streams should be avoided

Instrumentation for Power Measurements

Instrumentation Target

Measuring power on **system level** versus **device level**

System level:

- Noisy due to many power consumers
- Memory transfers to device will be included 100%

Device level:

- Allows for better correlation to kernel executions due to single power consumer
- PSU efficiency does not matter

Factors for measurement accuracy:

- Sample rate
- Sample resolution
- Correlation to begin and end of kernel

Instrumentation Implementation

- On-chip power meter
    - Very easy to use (via CUDA CUPTI or profiler)
    - Less accurate
- External device only
    - High precision
    - Power draw from PCIe interface difficult to measure
- External complete system
    - Less accurate
    - Easy to setup

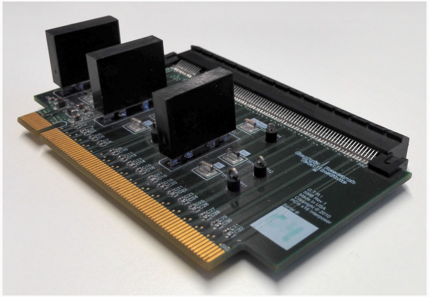

Figure 3: PCIe Riser.

NVML Usage

Measure power concurrent to kernel execution

```
void powerPolling()
{
  t_begin = t_old = now();
  while (kernel_running) {
    power = nvmlDeviceGetPowerUsage(...);
    t_new = now()
    weighted_power += power * (t_new - t_old);
  }
  power_result = weighted_power;
}
```

Constraints

- In general same as for performance measurements
- Kernels need to run longer for precise measurements
- Warm-up needed for consistent power measurements

Modeling GPU Kernels

Mapping Features to Predictions

Guessing execution time and power from features of a kernel.

Which metrics make good features?

- Instructions executed

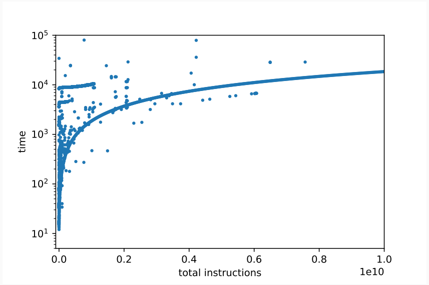

Figure 4: time over total instructions on K20 (zoomed section)

Mapping Features to Predictions

Guessing execution time and power from features of a kernel.

Which metrics make good features?

- Instructions executed
- FLOPS

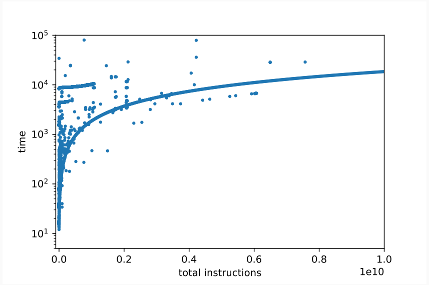

Figure 4: time over total instructions on K20 (zoomed section)

Mapping Features to Predictions

Guessing execution time and power from features of a kernel.

Which metrics make good features?

- Instructions executed
- FLOPS
- Memory footprint

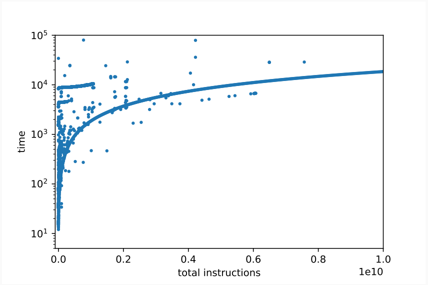

Figure 4: time over total instructions on K20 (zoomed section)

Mapping Features to Predictions

Guessing execution time and power from features of a kernel.

Which metrics make good features?

- Instructions executed
- FLOPS
- Memory footprint
- Size of thread grid

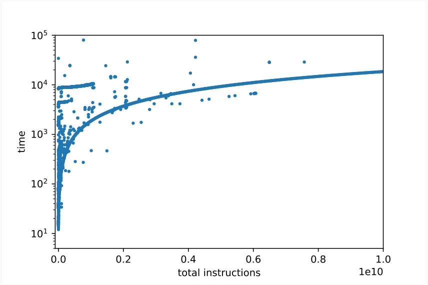

Figure 4: time over total instructions on K20 (zoomed section)

Mapping Features to Predictions

Guessing execution time and power from features of a kernel.

Which metrics make good features?

- Instructions executed
- FLOPS
- Memory footprint
- Size of thread grid
- FLOPS per load

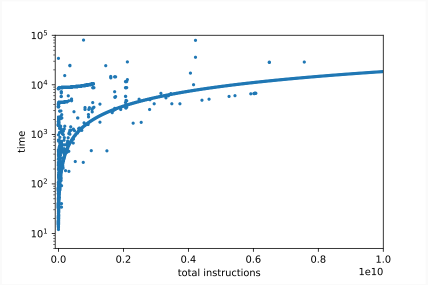

Figure 4: time over total instructions on K20 (zoomed section)

Mapping Features to Predictions

Guessing execution time and power from features of a kernel.

Which metrics make good features?

- Instructions executed
- FLOPS
- Memory footprint
- Size of thread grid
- FLOPS per load
- Synchronizations

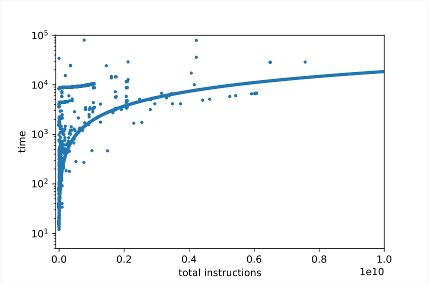

Figure 4: time over total instructions on K20 (zoomed section)

Portable Code Features

- Portable code features only depend on the kernel and the data handed to it
- Hardware metrics like cache-hit rates not allowed
- Creation of models for new GPUs requires only time and power measurements
- Instruction counter are essential; represent actual work of the processing units

Instructions have different levels of abstractions:

- programming language
- LLVM IR
- PTX
- SASS

PTX is portable **and** low level and therefore the candidate of choice.

Feature Engineering

Feature Engineering is used to provide more meaningful features.

- Correlation of raw metrics and outcome is often opaque
- Feature engineering improves situation by e.g.
    - Standardization (Gaussian distribution)
    - Normalization
    - Generating polynomial features
    - Aggregation of features

GPU Mangrove mainly works with aggregation of similar instructions and computing analytical features such as global memory volume read/written.

Model Construction and Training

Selection of learning method

- Resource usage vs accuracy
- Generalization vs over-fitting
- Quantity and quality of training data

Random Forest

- Light computational workload
- Likely to over-fit (but can be improved by training method)
- Works well with even few samples
- Interpolation outside range of training data is difficult

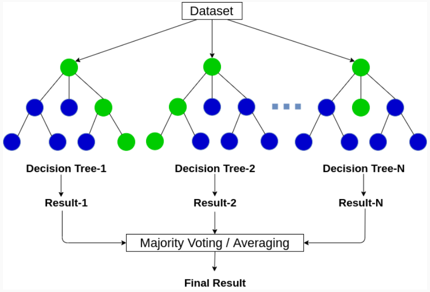

Figure 5: Diagram of a random decision forest, Shubham Gupta. Web `https://medium.com/@gupta020295/random-forest-easily-explained-4b8094feb9` 02.12.2020.

Training Methods

- Simple -> split data into training (~80%) and test set (~20%).
- Advanced -> Cross-Valitation
  - Evaluate multiple times on different test sets
  - Score is the average over all evaluations
  - Computational expensive but allows to use more data for training

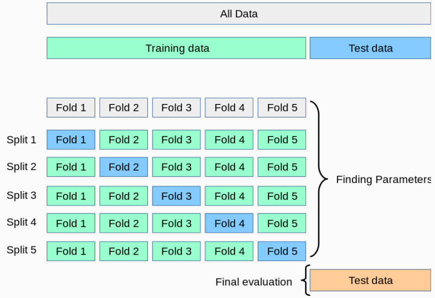

Figure 6: scikit-learn developers. Web `https://scikit-learn.org/stable/modules/cross_validation.html`. 08.12.2020

Requirements for GPU Mangrove

- Architecture/Hyper-parameter search
- Validation of model
- Waste as few data as possible

-> Nested Cross-Validation

Nested Cross-Validation

- Nested cross-validation offers validation, hyper-parameter serach and works well with few samples
- Model training and hyper-parameters are optimized with out biasing the model towards the dataset
- Even more training time required compared to standard cross-validation

Our hyper-parameter space: number of trees and split criteria

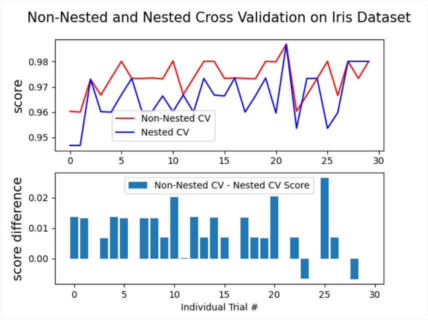

Figure 7: scikit-learn developers. Web `https://scikit-learn.org/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html`. 02.12.2020.

Prediction Results

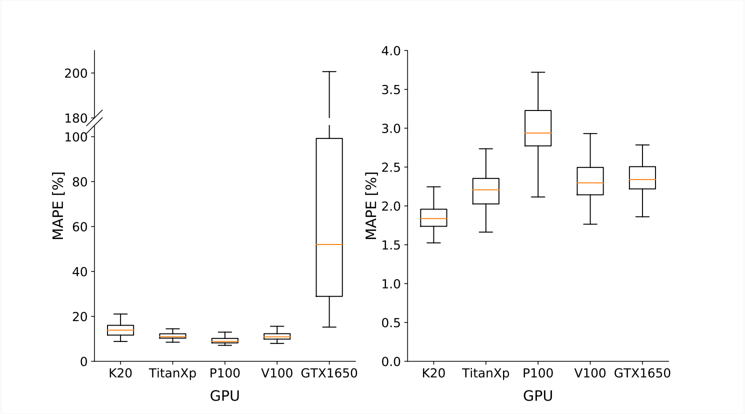

Figure 8: MAPE scores of time (left) and power (right) for all iterations of nested cross-validation with median, first and third quartile. Whiskers are limited to 1.5 times of the interquartile range (Q3-Q1)