CUDA Flux: A Lightweight Instruction Profiler for CUDA Applications

Lorenz Braun
Institute of Computer Engineering
Heidelberg University
Heidelberg, Germany
lorenz.braun@ziti.uni-heidelberg.de

Holger Fröning
Institute of Computer Engineering
Heidelberg University
Heidelberg, Germany
holger.froening@ziti.uni-heidelberg.de

***Abstract*—GPUs are powerful, massively parallel processors, which require a vast amount of thread parallelism to keep their thousands of execution units busy, and to tolerate latency when accessing its high-throughput memory system. Understanding the behavior of massively threaded GPU programs can be difficult, even though recent GPUs provide an abundance of hardware performance counters, which collect statistics about certain events. Profiling tools that assist the user in such analysis for their GPUs, like NVIDIA's nvprof and cupti, are state-of-the-art. However, instrumentation based on reading hardware performance counters can be slow, in particular when the number of metrics is large. Furthermore, the results can be inaccurate as instructions are grouped to match the available set of hardware counters.**

In this work we introduce CUDA Flux, an alternative to profiling based on hardware performance counters. As part of CUDA compilation, code is instrumented to collect statistics about the control flow. The resulting instruction count is then calculated based on these statistics in combination with an analysis of PTX assembly. In general, it is possible to trade profiling overhead for profiling accuracy, as the number of threads to instrument can be varied. Our experiments show that code instrumentation and associated data acquisition is usually faster than reading out a large amount of hardware performance counters, like being done by nvprof. Ultimately, we see code instrumentation as highly flexible, with many possibilities to trade accuracy for resource requirements, while the fundamental techniques can be preserved.

***Index Terms*—GPU, CUDA, LLVM, Profiling, PTX**

I. Introduction

Characterizing workloads is a recurring topic in various fields of computing, including GPGPUs. In order to develop new innovations, computer scientists have to understand fundamental properties of a variety of workloads. For instance, this understanding can be used for developing new hardware that is better adapted to the workloads or improving software like compilation tool chains to produce more efficient code. In general, this understanding should be close to the metal, meaning such a characterization should report metrics that can be easily mapped to the processor's ISA. While there are multiple ISA levels for GPUs, for NVIDIA GPUs the PTX ISA is best documented and most heavily used.

The currently dominating tool used to characterize CUDA kernels is NVIDIA's profiler `nvprof`. While this tool offers lots of metrics which provide insights on what happens in hardware, it lacks verbosity when it comes to analyzing the different types of instructions executed. This problem is aggravated by the recently frequent introduction of new instructions, for instance floating point operations with reduced precision or special function operations including Tensor Cores for machine learning. Similarly, information on the use of vectorized operations is often lost. Characterizing workloads on PTX level is desirable as PTX allows us to determine the exact types of the instructions a kernel executes. On the contrary, nvprof metrics are based on a lower-level representation called SASS. Still, even if there was an exact mapping, some information on the exact type would be lost as many instructions of similar type have only one metric. One example for such a loss of information is the usage of special floating-point units for computation of e.g. square-roots or trigonometric functions.

Characterizing GPU kernels on current CUDA implementations is easily done with nvprof. However, GPU applications are often computationally intensive and executed for a considerable amount of time. Even with only a few metrics, using nvprof to profile an application will result in substantial overhead and increase of execution time. In the worst case, probably due to limited hardware resources, kernels have to be replayed to gather all metrics requested by the user. This can result in different behavior for data-driven applications.

There are use cases that cannot be satisfied with the limited information of profiling based on hardware performance counters, thus the community developed a couple of alternatives. One class of proposed solutions includes GPU simulators like GPGPU-Sim [1], Multi2Sim [2] and Barra [3], which are certainly powerful tools to characterize GPU applications as nearly every aspect of an application can be investigated. However, substantial application slow-down and lack of support for recent GPU architectures often rule them out. Tools like GPU Ocelot [4] and Lynx [5] replace the CUDA runtime to instrument and recompile GPU kernels before execution, which are much more lightweight than simulation. According to Farooqui et al. [6], just-in-time compilation introduces 14.6 % overhead for a selected application which is a good result considering that overhead for long-running kernels will be lower and this step has only to be done once per kernel. Unfortunately, the maintenance of such tools is cumbersome and for each new GPU generation substantial effort is required,

978-1-7281-5977-5/19/$31.00 ©2019 IEEE
DOI 10.1109/PMBS49563.2019.00014

which often results in limitations regarding their use, respectively in these projects being no longer active.

Code instrumentation is flexible as it is easy to reduce the scope of instrumentation. For instance, only certain parts of the code, like specific functions which are of interest to the user, can be instrumented, reducing overall overhead. Rather than limiting instrumentation of code, the number of instrumented threads can be reduced as GPUs are thread-collective processors and the threads exhibit identical or at least very similar control flow in many cases. Ultimately, one can reduce the scope of instrumentation to a single thread, while assuming that other threads of the same kernel behave very similar. Such an approximation has a huge impact on instrumentation overhead, as only one thread is profiled instead of thousands of threads per kernel.

In this work we introduce a new tool called CUDA Flux that characterizes the executed instructions of CUDA kernels. It is based on the LLVM compiler framework and instruments GPU kernels to collect instruction statistics. The detailed contributions are as follows:

- An extension to the existing LLVM compiler framework that allows instrumentation of CUDA kernels at compile time with no need for later just-in-time compilation.
- Exact characterization of all the types of PTX instructions that are being executed by a kernel, resulting in a large amount of *verbosity*.
- Instrumentation with *very low execution overhead* by approximating control flow divergence, resulting in minimal application slow-downs.
- Easy *extendability* of instrumentation, also leveraging LLVM community efforts regarding CUDA and PTX updates. Similarly, the tool maintenance overhead is low since many updates are already covered by the LLVM community.
- Adaptable fundamentals which allow very *selective* instrumentation. For example, instrumentation of selected parts of code for less overhead or higher accuracy on control flow divergence can be gained by allowing more instrumentation overhead.

The remainder of this work gives background on the compiler framework, source code assembly and limitations of other currently available tools for workload characterization. Details of the tool design and the methods follow. The remaining part addresses the methods for evaluation and their results.

II. Background

This section provides a brief background on the used compiler framework and the GPU instruction set. While this work is focused on CUDA, many techniques have corresponding counterparts in alternative GPU programming frameworks, in particular OpenCL.

A. LLVM

The compiler framework LLVM [7] natively supports the compilation of CUDA code, since work of `gpucc` [8] has been fully integrated. The framework can be divided in three different parts: the front end (Clang for C/C++), optimizer passes, and the back end. Furthermore, the framework provides a well-defined intermediate representation (IR) of the program code. These different parts are exchangeable because the IR neither depends on a particular programming language nor a target architecture. This allows for very similar compilation pipelines for CUDA and C/C++ code.

The compilation pipeline can be easily extended by adding passes to the optimizer passes that analyze or modify the IR. In the case of CUDA compilation, there are two compilation pipelines: one for host code and one for device (GPU) code. The compilation of device code results in PTX code which is assembled into a `fat binary`. The host code compilation pipeline will include this binary and additional code is added to register and launch kernels.

B. PTX vs. SASS

CUDA is mainly used for NVIDIA GPUs. The code of a CUDA kernel is usually compiled to PTX. Before execution on a specific GPU, the PTX code is translated to SASS. Thus, kernel code can be characterized two different ISAs: PTX and SASS.

The PTX ISA is an intermediate representation for kernel assembly [9]. PTX defines a virtual machine for which the CUDA compiler generates code for. When launching PTX code, it will be translated just-in-time to the target hardware instruction set and then executed on the GPU. This has the advantage that this ISA can be used across different GPU generations with different instructions sets.

The target hardware instruction set is called SASS [10]. While PTX is well documented by NVIDIA, SASS is less accessible. There is no guarantee that upcoming GPU architectures will have a similar ISA. The lack of information and not guaranteed portability makes the SASS unattractive for development. However, when optimizing kernels for specific GPU models, analyzing SASS instructions may be useful.

C. Limitations of Currently Available Tools for Workload Characterization

Two important criteria for the selection of a tool for a specific task are the effort that is required to use the tool, and the quality of the results. Depending on the goal of workload characterization, the qualities may change.

Considering instruction profiling of CUDA kernels, an important quality is the scope on which instructions are counted. For example, operating on high-level C/C++ operators will ignore that programs are often significantly optimized by compilers. Viewing PTX instructions is a good compromise between precision and portability. Emitted PTX code is already optimized by the compiler and expresses the kernel with universal operations that can be implemented on any CUDA device (given certain restrictions regarding supported compute capabilities).

Regarding the effort required to get the same quality of results, lower is always better. In some cases it is desirable to trade lower quality or precision for lower effort or less

time required, which is in particular true for long-running, computationally intensive programs.

As nvprof [11] and related tools are based on hardware performance counters, it is sometimes hard to map the reported metrics to PTX instructions. To the best of our knowledge, it is not possible to count a specific type of instruction. Besides this disadvantage, overhead for profiling can be quite high. These results are in line with other recent work, which documented the overhead associated with measuring an increasing amount of nvprof events [12].

Besides nvprof, there are other profilers for CUDA applications. SASSI [10] is another instrumentation tool that operates on low-level assembly language. Mapping these instructions to PTX instructions should be possible but would require additional effort for each GPU architecture. In addition, SASSI is not open source and thus cannot be extended.

CUDAAdvisor [13] is based on LLVM and provides a profiling framework to guide code optimizations. While the authors note that CUDAAdvisor is in principal extendable, the current instrumentation operates on LLVM IR level and does not directly consider PTX code.

Ocelot is a dynamic CUDA compiler that can generate code for GPUs and CPUs at runtime. With Lynx, which is built on top of Ocelot, Farooqui et al. showed how this concept can be used for instrumentation [5, 6]. Instrumenting on PTX level is used to count how often each basic block is executed. Fundamentally, part of Lynx is based on counting the execution of basic blocks, which is conceptually similar to this work. Contrary, the instrumentation is done dynamically at runtime, which adds some overhead that could be avoided. Furthermore, Lynx is not maintained anymore and PTX ISA higher than version 3.0 may not work. Compared to Lynx and its conceptual foundation on Ocelot, this work rather builds on top of the publicly maintained LLVM infrastructure, hopefully resulting in a continuous development and no constraints regarding PTX versions

Besides tools for instrumentation, there are also various GPU simulators [1, 2, 3]. While simulators provide very detailed insights, the execution time increase is usually substantial. Simulating long-running kernels is often not feasible. Furthermore, most simulators only support rather oldish GPU architectures and the effort for updating simulator models is very high.

III. Tool Design

CUDA Flux analyzes and instruments the kernel code at compile time. It is embedded into the LLVM compilation toolchain for CUDA source files. In this way, the instrumentation is neatly integrated and will be executed automatically when the associated wrapper for the Clang front end is used when compiling.

While executing the resulting binary, information on each launched kernel is collected and stored for later analysis. In order to keep the execution overhead low, further analysis is done separately from the application execution.

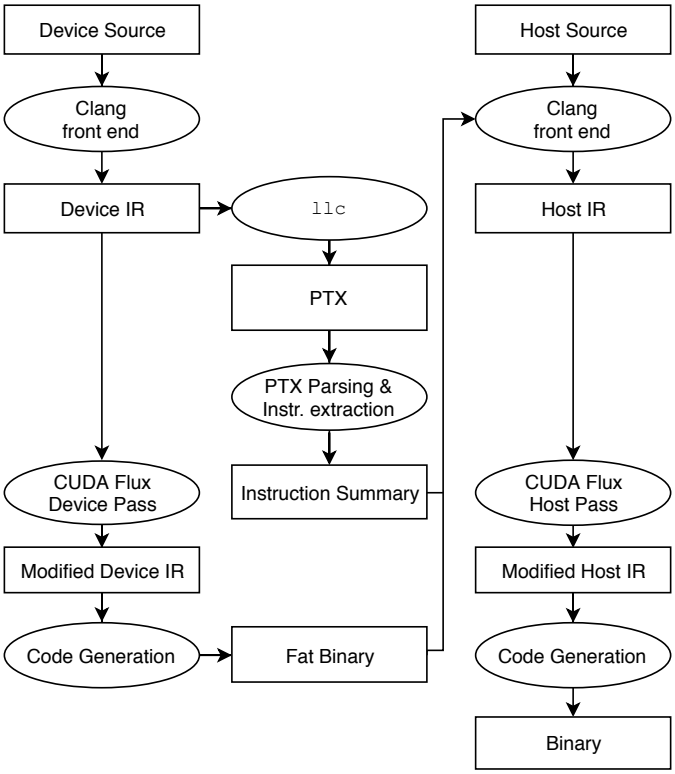

Fig. 1: Workflow of CUDA Flux Instrumentation

Three major components are added to the Clang compilation: a device instrumentation pass, a PTX parser and a host instrumentation pass. Figure 1 gives an overview on the work flow during CUDA compilation.

Since a single CUDA file can contain code for host and device, the same file is compiled in two pipelines, one for device code and another one for host code. Note that the device code pipeline has to be executed first because the generated fat binary has to be embedded into the host binary.

The device IR is modified by the device instrumentation pass. Before the modification, the IR is compiled to PTX which is processed by the PTX parser to an instruction summary.

The fat binary with the modified kernel is handed over to the Clang pipeline for the host code. During the host instrumentation, further modifications are added to complete the instrumentation.

The following subsections will describe in detail what is done in these three major components, and how the post-processing step produces the final results.

A. Device Instrumentation Pass

The device instrumentation implements the following functions:

- Provide infrastructure to the kernel to store basic block counters.
- Instrument basic blocks with code to increase their corresponding counter.

- Emit PTX code using `llc`.
- Invoke the PTX Parser.

B. PTX Parser

The PTX parser consumes the PTX assembly produced by `llc`. The parser does not support the full PTX ISA specification, as only the instructions and not their operators need to be processed. Besides the instruction types, the parser needs to be able to detect functions and basic blocks. The list of functions, basic blocks and instructions is extracted from the token list of the PTX assembly and processed into the instruction summary. The summary is stored for later use in post-processing.

C. Host Instrumentation Pass

The host instrumentation implements the following functions:

- Replacing kernel invocations with the corresponding instrumented kernels.
- Extending kernel launches by providing memory for the counters, measuring time and storing results.
- Embedding the instruction summary of the parser into the binary.

Embedding the instruction summary into the binary is only for convenience, as it allows for an integrated output, including the counter for the basic blocks along with the list of functions, basic blocks and instructions.

D. Post-Processing of Profiling Data

The execution of the binary itself only provides basic block execution frequencies. Based on this information, in combination with results of the PTX parser, the task of post-processing is to provide counters for all executed instructions. For every kernel launch the following information is stored:

- Basic Block execution counts
- Grid and thread block size
- Shared memory used by the kernel

For convenient processing of these executions counts, a list of all kernels, including their basic blocks and the instructions they are composed of, is stored. To obtain the final results, the occurrences of instructions in the kernel list will be multiplied with their corresponding block counter.

Depending on the usage of the counters, we suggest summarizing closely related instructions into groups. However, this is ultimately a decision that is left to the user.

E. Limitations

In order to achieve a very low profiling overhead and only little additional memory requirements, only one single thread is profiled. As mentioned previously, our use case is to profile regular workloads, where most of the threads will have a highly similar control flow. This is an approximation which should fit the GPU programming model well, as branch divergence and therefore control flow divergence are common performance bugs that should be avoided. In some cases, threads at the edge of the up to 3-dimensional CTA only assist other threads by e.g. loading data into shared memory and not taking part in the computation. Such a behavior is not tracked by the default configuration, however, this can easily be explored by modifying the instrumentation code. Because threads on the edges of the grid may behave different from the majority of threads, the thread performing profiling is located in the center of a CTA, which itself is in the center of its block grid.

The Clang front-end of LLVM does not support texture memory, therefore, applications making use of texture memory currently cannot be profiled. As soon as LLVM supports texture memory, the CUDA Flux will also natively support it.

Since the device pass and the host pass need to exchange information, the kernel has to be called in the compilation module in which it is defined. This limitation can be easily avoided by wrapping the function call. However, this workaround would require code modifications.

Concurrent kernel execution is currently not supported, mainly because there is no thread-safe interface to store the instrumentation results (yet).

Last, due to being still in development, the instrumentation does only work on kernels with inlined functions. However, this limitation has not caused any problem so far.

IV. Evaluation Methodology

LLVM with the CUDA Flux extension is used to compile four commonly used benchmark suites. The profiling results are compared with data gathered with nvprof. Execution time of the kernels and the applications is also measured. Three different sets of measurements are done to measure overhead and to compare with the overhead introduced profiling with nvprof:

- nvcc without profiling (baseline)
- nvcc with nvprof profiling: overhead of nvprof-based profiling
- LLVM with CUDA Flux instrumentation: overhead of profiling based on CUDA Flux

For the evaluation, four benchmark suites are used: Rodinia, Parboil, SHOC, and Polybench-GPU. Table I reports all applications which have been used and their number of unique kernels.

Due to missing support for texture memory within the LLVM tool chain, the following applications had to be excluded:

- Rodinia: hybridsort, mummergpu, leukocyte, kmeans
- Parboil: bfs, sad
- Shoc: DeviceMemory, MD, spmv

Additionally, some applications are not used because of compilation issues:

- Rodinia: cfd, huffman, pathfinder, hotspot
- Parboil: cutcp, mri-gridding
- Shoc: qtclustering

Finally, the GEMM benchmark from the shoc benchmark suite was not used because there is no kernel which can be

Benchmark	Application	Unique Kernels
parboil-2.5	**cutcp**	1
	histo	4
	lbm	1
	mri-gridding	8
	mri-q	2
	sgemm	1
	spmv	1
	stencil	1
	tpacf	1
polybench-gpu-1.0	**2DConvolution.exe**	1
	2mm.exe	2
	3DConvolution.exe	1
	3mm.exe	3
	atax.exe	2
	bicg.exe	2
	correlation.exe	4
	covariance.exe	3
	fdtd2d.exe	3
	gemm.exe	1
	gesummv.exe	1
	gramschmidt.exe	3
	mvt.exe	2
	syr2k.exe	1
	syrk.exe	1
rodinia-3.1	**3D**	1
	b+tree.out	2
	backprop	2
	dwt2d	4
	euler3d	4
	gaussian	2
	heartwall	1
	lavaMD	1
	lud_cuda	3
	myocyte.out	1
	needle	2
	particlefilter_float	4
	particlefilter_naive	1
	sc_gpu	1
	srad_v2	2
shoc	**DeviceMemory**	6
	FFT	6
	MaxFlops	36
	Reduction	2
	Scan	6
	Sort	5
	Stencil2D	2

TABLE I: Applications of all used benchmark suites and the corresponding number of unique kernels.

instrumented. The benchmark simply measures a cublas library call.

V. Accuracy Evaluation

In this section, we provide a case example to demonstrate the verbosity of the metrics that can be gathered using CUDA Flux. Furthermore, we investigate the accuracy of the approximation when using only one thread to profile a whole thread grid.

A. Case Example FFT

Replacing the instruction metrics provided by nvprof is not easily done due to missing documentation how PTX instructions are mapped to hardware counters. Despite that PTX and SASS instructions may not be translated one-to-one in general, the results of nvprof and CUDA Flux should be similar because the most frequently used PTX instructions are likely to have counterparts in SASS. To confirm this, we will create PTX instruction groups similar to nvprof metrics.

NVIDIA's profiler provides lots of metrics, but usually only a subset reflects the inherent characteristics of the executed kernel. As this work considers only profiling of instructions, we limit the considered metrics to those which represent executed instructions.

Before compute capability 5.0, only the following metrics represented executed instructions:

- flop_count_dp
- flop_count_dp_add
- flop_count_dp_fma
- flop_count_dp_mul
- flop_count_sp
- flop_count_sp_add
- flop_count_sp_fma
- flop_count_sp_mul
- flop_count_sp_special
- inst_bit_convert
- inst_compute_ld_st
- inst_control
- inst_executed
- inst_fp_32
- inst_fp_64
- inst_integer
- inst_inter_thread_communication
- inst_issued
- inst_misc

These metrics give only very limited insight on what an application does in detail. With modern NVIDIA GPUs, there exist more metrics, which can be used to count the executed memory operations. The categories of those metrics are still coarse and it is not possible to examine the execution count of specific operations. This applies to special function units for e.g. trigonometric functions, video encoding instructions, and instructions using the recently introduced Tensor Core or Ray-Tracing units.

The metrics are compared in detail using a kernel of the FFT application from the shoc benchmark suite as an example. Listing 1 shows the source code of this kernel. This particular kernel is suited well to show the differences between profiling with CUDA Flux and nvprof:

- All threads perform the same amount of work, which allows to scale the instruction counters with the number of the total threads executing the kernel.
- The FFT algorithm uses special function units for sine and cosine functions.
- The underlying data of the imaginary numbers can be accessed using vectorized loads and stores.

Type `T2` is an imaginary number that uses either double or float data type for each component. Loading imaginary numbers can be vectorized. The transpose function in line 18 and 25 operates on shared memory. In theory, there could be vectorized loads on shared memory as well. Sine and cosine functions are called from the twiddle functions in line 17 and 22.

```
template<class T2, class T> __global__
void FFT512_device( T2 *work )
{
    int tid = threadIdx.x;
    int hi = tid>>3;
    int lo = tid&7;

    work += (blockIdx.y * gridDim.x + blockIdx.x) *
        512 + tid;

    T2 a[8];
    __shared__ T smem[8*8*9];

    load<8, T2>( a, work, 64 );

    FFT8<T2,T>( a );

    twiddle<8,T2,T>( a, tid, 512 );
    transpose<8, T2, T>( a, &smem[hi*8+lo], 66, &
        smem[lo*66+hi], 8 );

    FFT8<T2,T>( a );

    twiddle<8,T2,T>( a, hi, 64);
    transpose<8, T2, T>( a, &smem[hi*8+lo], 8*9, &
        smem[hi*8*9+lo], 8, 0xE );

    FFT8<T2,T>( a );

    store<8, T2>( a, work, 64 );
}
```

Listing 1: FFT512 kernel code of the shoc benchmark suite

The results of the double-precision version of this kernel are show in Table II. These results of CUDA Flux have been multiplied by the total number of threads for a direct comparison with nvprof metrics. Part of the loads and stores are indeed vectorized, as the `.v2` identifier in the instruction name indicates. Note that access to shared memory is not vectorized. Also, sine and cosine instructions each have their own counter, respectively.

For a comparison with nvprof, the metrics from table II are grouped and, for warp-level nvprof metrics, scaled to reproduce the instruction metrics of nvprof. Table III compares the obtained results from nvprof and CUDA Flux. Metrics that start with `inst_executed` are counted per warp instead of per thread.

Based on this comparison, we can make the following observations:

- Vectorized loads and stores are counted as a single load or store by nvprof and not as multiple ones.
- `inst_compute_ld_st` counts access to global and shared memory.
- `flop_count_sp` does not include trigonometric instructions, and probably also not other special-function unit instructions.
- The metrics `flop_count_dp_add`, `inst_fp_32` and `inst_integer` show the largest deviation.

The last observation is difficult to explain. There are two factors that are probably responsible for the deviation.

1) Faulty mapping between PTX instructions and nvprof metrics

Metric Name	Value
add.rn.f32	0
add.rn.f64	144,703,488
add.s32	2,097,152
add.s64	8,388,608
and.b32	4,194,304
bar.sync	14,680,064
cos.approx.f32	29,360,128
cvt.f64.f32	58,720,256
cvt.rn.f32.f64	29,360,128
cvt.rn.f64.s32	4,194,304
cvta.to.global.u64	2,097,152
fma.rn.f32	0
fma.rn.f64	109,051,904
ld.global.f32	0
ld.global.v2.f32	0
ld.global.v2.f64	16,777,216
ld.global.v2.u64	0
ld.param.u64	2,097,152
ld.shared.f32	0
ld.shared.f64	67,108,864
mad.lo.s32	4,194,304
mov.b32	0
mov.b64	0
mov.u32	8,388,608
mov.u64	2,097,152
mul.lo.s32	2,097,152
mul.rn.f32	0
mul.rn.f64	113,246,208
mul.wide.u32	8,388,608
neg.f32	0
neg.f64	35,651,584
or.b32	2,097,152
ret	2,097,152
shl.b32	2,097,152
shr.u32	2,097,152
sin.approx.f32	29,360,128
st.global.v2.f32	0
st.global.v2.f64	16,777,216
st.global.v2.u32	0
st.global.v2.u64	0
st.shared.f32	0
st.shared.f64	67,108,864
sub.rn.f32	0
sub.rn.f64	144,703,488

TABLE II: Results of CUDA Flux instrumentation for the double-precision FFT512 kernel

2) Differences between SASS and PTX assembly

It is not exactly known how a PTX instruction translates into SASS. One PTX instruction may be translated into multiple SASS instructions. Furthermore, during translation of PTX to SASS, there is still the possibility to perform optimizations. For instance, some PTX instruction combinations could be replaced by a single SASS instruction.

B. Accuracy

If a single thread is representative for a whole kernel, the total instruction count of CUDA Flux should be in the same order of magnitude compared to nvprof results. Differences introduced by PTX to SASS translation should only lead to small deviations of the instruction count. In cases where a single thread is not representative, the approximation would fail and the instruction count could be off by a higher factor.

Figure 2 shows a histogram of the inst_executed metric measured with CUDA Flux, normalized to nvprof results for almost all applications. The applications spmv and histo crashed while profiling with nvprof, and the mri-gridding

Metric Name	nvprof	CUDA Flux	Ratio
flop_count_dp	645,922,816	547,356,672	1.18
flop_count_dp_add	301,989,888	144,703,488	2.09
flop_count_dp_fma	121,634,816	109,051,904	1.12
flop_count_dp_mul	100,663,296	113,246,208	0.89
flop_count_sp	0	58,720,256	0.00
flop_count_sp_add	0	0	-
flop_count_sp_fma	0	0	-
flop_count_sp_mul	0	0	-
flop_count_sp_special	58,720,256	58,720,256	1.00
inst_bit_convert	92,274,688	94,371,840	0.98
inst_compute_ld_st	167,772,160	167,772,160	1.00
inst_control	2,097,152	2,097,152	1.00
inst_executed	30,212,096	29,163,520	1.04
inst_executed_global_atomics	0	0	-
inst_executed_global_loads	524,288	524,288	1.00
inst_executed_global_reductions	0	0	-
inst_executed_global_stores	524,288	524,288	1.00
inst_executed_local_loads	0	0	-
inst_executed_local_stores	0	0	-
inst_executed_shared_atomics	0	0	-
inst_executed_shared_loads	2,097,152	2,097,152	1.00
inst_executed_shared_stores	2,097,152	2,097,152	1.00
inst_executed_surface_atomics	0	0	-
inst_executed_surface_loads	0	0	-
inst_executed_surface_reductions	0	0	-
inst_executed_surface_stores	0	0	-
inst_executed_tex_ops	0	0	-
inst_fp_32	88,080,384	58,720,256	1.50
inst_fp_64	524,288,000	547,356,672	0.96
inst_integer	35,651,584	20,971,520	1.70
inst_inter_thread_communication	0	0	-
inst_issued	30,216,709	0	inf
inst_misc	56,623,104	44,040,192	1.29

TABLE III: Instrumentation results of nvprof and CUDA Flux for the FFT512 kernel in comparison

benchmark did produce incorrect results. The cause of this still needs to be investigated.

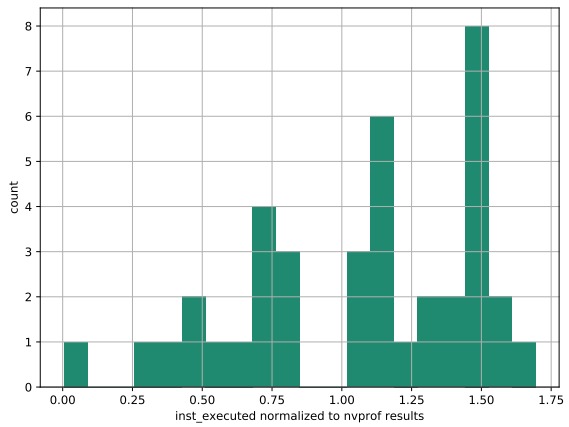

Fig. 2: Histogram of inst_executed metric measured with CUDA Flux Profiler normalized to nvprof results

The applications Sort and Stencil2D are also not included. For these applications, the current approximation is probably not feasible. For Sort, the CUDA Flux reported 9.02 times more instructions executed and for Stencil2D the ratio is 35.52. Looking into the kernels of the Sort applications there is some branch divergence that could explain the high difference. Regarding Stencil2D, there is also some branch divergence, because loading the halo of other tiles is only done by threads working on items at the edge of the tile. But it is unlikely that this divergence leads to such a high factor because the number of these threads should be rather low compared to total number of threads a CTA. Another outlier is myocyte.out with a ratio 0.0042. Because the application has many kernels, a thorough investigation would be needed to make a statement regarding the cause of this.

Referring to Figure 2, many applications have a ratio that is close to one, actually slightly larger than one to be concrete. Even though we chose a very simple approximation method for this work, these results suggest that actually such an approximation is appropriate, as the overall deviation from nvprof is quite low. Thus, it suggests that such an approximation is sufficient when considering regular workloads with very low or no branch divergence. Still, more studies are required to assess the trade-off in terms of accuracy and speed in more detail.

VI. Performance Evaluation

In this section, we will report on performance differences to an uninstrumented application as baseline; furthermore we also compare against execution time for nvprof-based profiling. All experiments are repeated five times and the average execution time is reported.

Figure 3 shows the execution time measurements. Note that the y-axis is of logarithmic scale. Benchmarks are ordered ascending by execution time without profiling. Except for the spmv application from the parboil benchmark suite, CUDA Flux outperforms nvprof consistently.

For some applications, the CUDA Flux binary actually executes faster than the baseline, but without separate measurement of kernel time is not possible to say whether the kernels are actually executed faster. Six of the seven applications that execute faster than the baseline are from the polybench benchmark suite. Those benchmarks compute the result also on the CPU to compare the accuracy. Since the polybench benchmark suite originates from the LLVM community, it might be possible that the binary produced by LLVM is better optimized compared to the binaries using nvcc. In general, according to Wu et al. [8], binaries produced by nvcc should be about as fast as clang.

Table IV reports the detailed results of the measurements. The nvprof and CUDA Flux columns are normalized to the baseline execution time. In average, the overhead for using CUDA FLUX is 13.2 which is a big improvement to nvprof, which is about 171.01 higher than the baseline.

VII. Conclusion

This paper introduces CUDA Flux, a profiler for lightweight instruction profiling. With the integration into the LLVM framework, we see this tool as viable alternative to current

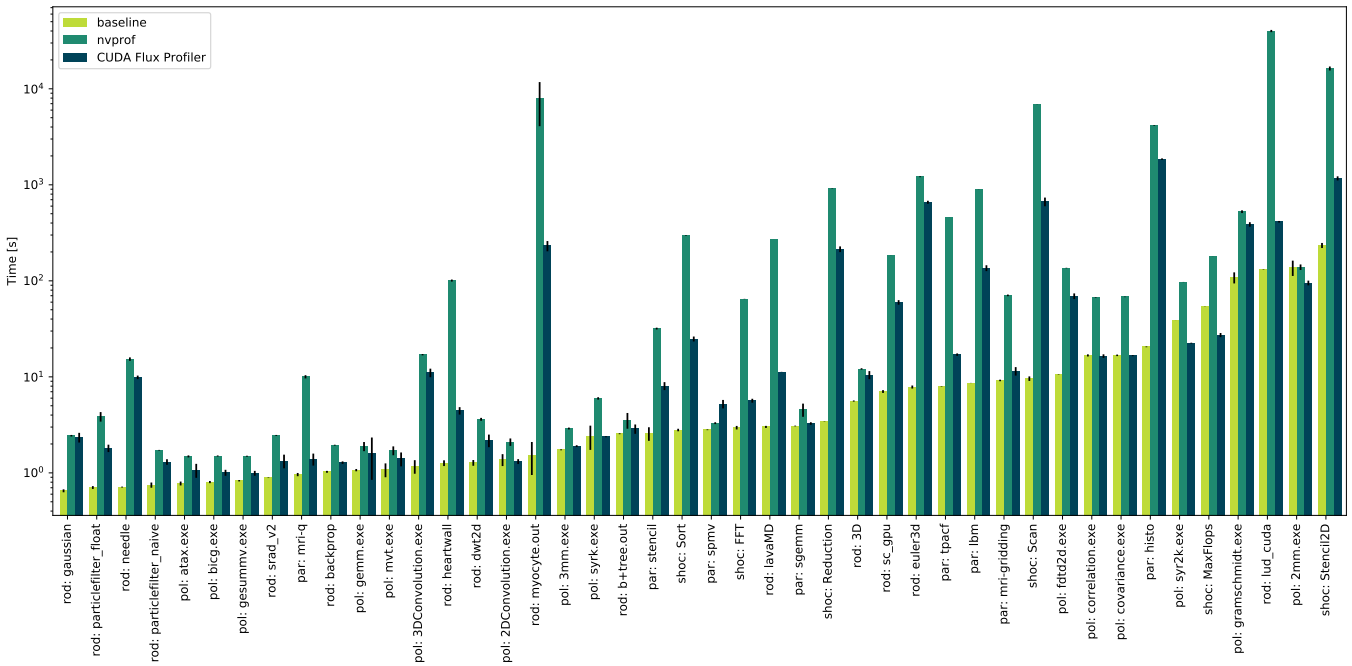

Fig. 3: Plot of benchmark execution times using no profiling, nvprof and CUDA Flux Profiler

tools for workload characterization. Describing CUDA applications with PTX instructions as an abstraction serves well for gaining deeper insights on application behavior.

We have shown a couple of advantages compared to hardware performance counters, including accuracy and speed: accuracy improves as code instrumentation allows a direct mapping of PTX instructions to counters, and furthermore avoids grouping effects that often result in ambiguous profiling results. Also, profiling of the application is much faster: in average, while nvprof increases execution time by a factor of 171.01, compared to a non-profiled execution, the total execution time of an application instrumented by CUDA Flux is only 13.2 times slower. This fast execution time is achieved by profiling only one representative thread of a kernel, and by avoiding costly kernel replays as often employed by nvprof. While this is an approximation of other threads' behavior, we have shown that this works very well for most CUDA application. Applications whose control flow do not depend on input data or threadID will have no noticeable accuracy degradation.

As the fundamental techniques of CUDA Flux allow profiling more threads, it is also possible to increase the instrumentation scope, which would greatly improve accuracy on workloads with divergent control flow. Profiling additional threads will increase the needed memory bandwidth. With similar control flow, this bandwidth can be expected to be increase linear with the number of threads. Depending on the complexity of the control flow, coalescing and caching will affect the overhead more or less, which makes it hard to estimate how much additional overhead is introduced by profiling more threads. We envision that using a sophisticated analysis at compile time, the selection of threads needed to produce representative results could be adapted to the kernel which is being instrumented.

Finally, it is possible to instrument only specific functions, which would lower the overhead even more and remove noise from other code parts.

References

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2009, pp. 163–174.

[2] X. Gong, R. Ubal, and D. Kaeli, "Multi2Sim Kepler: A detailed architectural GPU simulator," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2017, pp. 269–278.

[3] S. Collange, D. Defour, and D. Parello, "Barra, a modular functional gpu simulator for gpgpu," *University de Perpignan, Tech. Rep*, 2009.

[4] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques - PACT '10*. Vienna, Austria: ACM Press, 2010, p. 353.

[5] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili, "Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures," in *2012 IEEE International Symposium on Per-*

	Application	baseline [s]	nvprof [norm.]	CUDA Flux [norm.]	# k. launches
parboil-2.5	histo	20.54	202.66	90.19	40,000
	lbm	8.63	104.23	15.73	3,000
	mri-gridding	9.18	7.69	1.25	50
	mri-q	0.96	10.43	1.45	3
	sgemm	3.08	1.48	1.06	1
	spmv	2.81	1.17	1.86	50
	stencil	2.57	12.38	3.13	100
	tpacf	7.89	57.39	2.16	1
polybench-gpu-1.0	2DConvolution	1.37	1.53	0.96	1
	2mm	137.39	1.01	0.69	2
	3DConvolution	1.17	14.57	9.46	254
	3mm	1.74	1.67	1.09	3
	atax	0.78	1.91	1.37	2
	bicg	0.80	1.87	1.25	2
	correlation	16.73	3.99	0.98	4
	covariance	16.76	4.06	0.99	3
	fdtd2d	10.63	12.79	6.52	1,500
	gemm	1.07	1.77	1.49	1
	gesummv	0.83	1.80	1.20	1
	gramschmidt	108.27	4.85	3.58	6,144
	mvt	1.08	1.58	1.30	2
	syr2k	38.40	2.53	0.59	1
	syrk	2.42	2.47	0.98	1
rodinia-3.1	3D	5.60	2.16	1.87	100
	b+tree.out	2.56	1.39	1.12	2
	backprop	1.03	1.89	1.25	2
	dwt2d	1.27	2.84	1.71	10
	euler3d	7.84	154.75	83.98	14,003
	gaussian	0.65	3.75	3.60	30
	heartwall	1.26	79.63	3.51	20
	lavaMD	3.01	88.87	3.74	1
	lud_cuda	130.67	306.01	3.16	6,142
	myocyte.out	1.52	5193.69	151.82	6,071
	needle	0.71	21.53	13.90	255
	particlefilter_f	0.70	5.48	2.56	36
	particlefilter_n	0.75	2.30	1.74	9
	sc_gpu	7.04	26.29	8.48	1,611
	srad_v2	0.89	2.76	1.50	4
shoc	FFT	2.96	21.88	1.91	60
	MaxFlops	53.70	3.32	0.51	361
	Reduction	3.45	265.75	61.90	5,120
	Scan	9.59	709.20	69.57	15,360
	Sort	2.80	105.58	8.88	480
	Stencil2D	233.89	69.67	5.00	22,000
	Min	0.65	1.01	0.51	1
	Max	233.89	5,193.70	151.82	40,000
	Mean	19.70	171.01	13.20	2790.98

TABLE IV: Table of benchmark execution times using no profiling, nvprof and CUDA Flux Profile. nvprof and CUDA Flux results are normalized to the baseline. The last column refers to the number of kernel launches.

formance Analysis of Systems Software, Apr. 2012, pp. 58–67.

[6] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A Framework for Dynamically Instrumenting GPU Compute Applications Within GPU Ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 9:1–9:9.

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* San Jose, CA, USA: IEEE, 2004, pp. 75–86.

[8] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "Gpucc: An open-source GPGPU compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016, pp. 105–116.

[9] N. Corporation, "Parallel Thread Execution ISA Version 6.3," http://docs.nvidia.com/cuda/parallel-thread-execution/index.html, Oct. 2018.

[10] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *ACM SIGARCH Computer Architecture News*, vol. 43. ACM, 2015, pp. 185–197.

[11] N. Corporation, "Profiler User's Guide," http://docs.nvidia.com/cuda/profiler-users-guide/index.html, Oct. 2018.

[12] S.-K. Shekofteh, H. Noori, M. Naghibzadeh, H. S. Yazdi, and H. Fröning, "Metric Selection for GPU Kernel Classification," *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 4, pp. 1–27, Jan. 2019.

[13] D. Shen, S. L. Song, A. Li, and X. Liu, "CUDAAdvisor: LLVM-based runtime profiling for modern GPUs," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*. Vienna, Austria: ACM Press, 2018, pp. 214–227.