CUDA Flux: A Lightweight Instruction Profiler for CUDA Applications

Lorenz Braun, Holger Fröning
{lorenz.braun, holger.froening}@ziti.uni-heidelberg.de
Heidelberg University, Germany

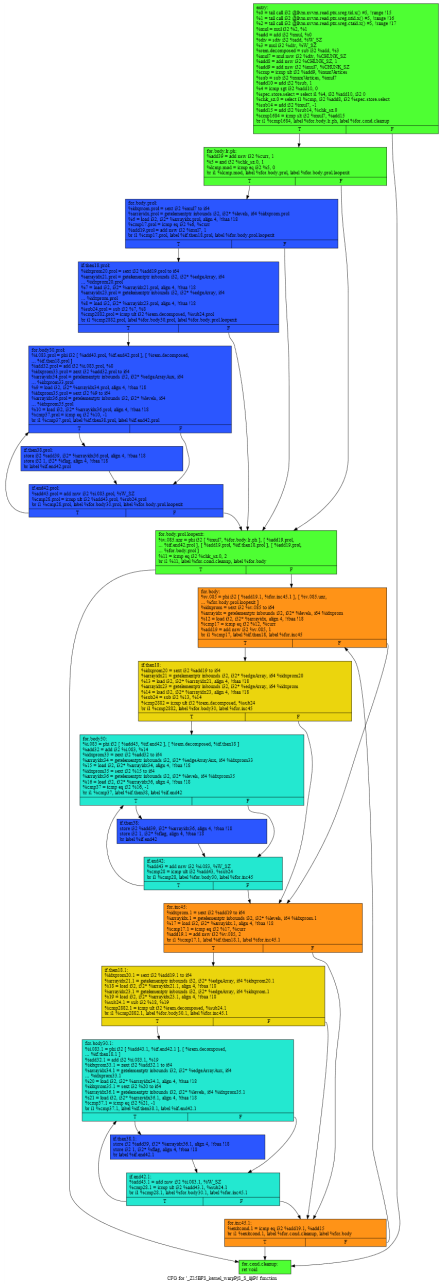
PMBS 2019, Denver, CO - November 18, 2019

Profiling GPU Applications

Our research goal: Performance prediction

- Many approaches for performance prediction already exist
- Prediction Models mostly use hardware related metrics
- Using only application related metrics is challenging, but also offers some advantages
  - Portability
  - Predictability

Nice side effect:
CUDA Flux is usable for other tasks such as performance debugging



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)

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

Currently available tools for profiling do not fit our needs well

-> development of CUDA Flux

CUDA Flux

- **LLVM Based** - a compiler framework supported by a large community
- **Low Overhead** - 6 to 10 times (average) faster than using nvprof
- **PTX Based** - a stable and well defined ISA
- **Fine Grained Instruction Counter** - for in-depth performance debugging
- **Portable Profiling Results** - enables performance modelling
- **Selectable Degree of Instrumentation** - accuracy/time trade-off
- **Open-Source** - available on github: **https://github.com/UniHD-CEG/cuda-flux**
- **Lightweight Code Base** - better maintainability and extendability

The LLVM Compiler Framework and CUDA

- Since integration of gpucc[1], CUDA code is natively supported
- Framework can be split up in front-end, 'middle-end' (optimizer) and back-end
- Middle-end can be easily extended by registering custom transformation passes
- CUDA compilation is implemented using mixed mode compilation flow

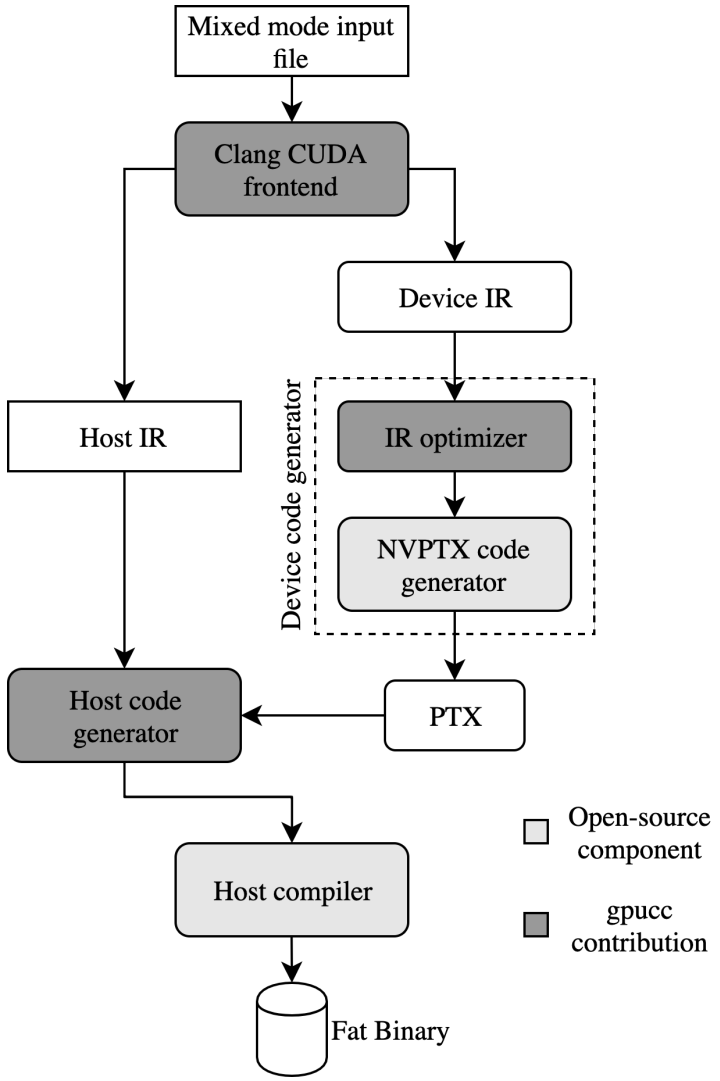


image modified from [1]

[1] Wu, Jingyue, et al. "gpucc: an open-source GPGPU compiler"
Proceedings of the 2016 International Symposium on Code Generation and Optimization. ACM, 2016

Tool Design

Contributions of this work:

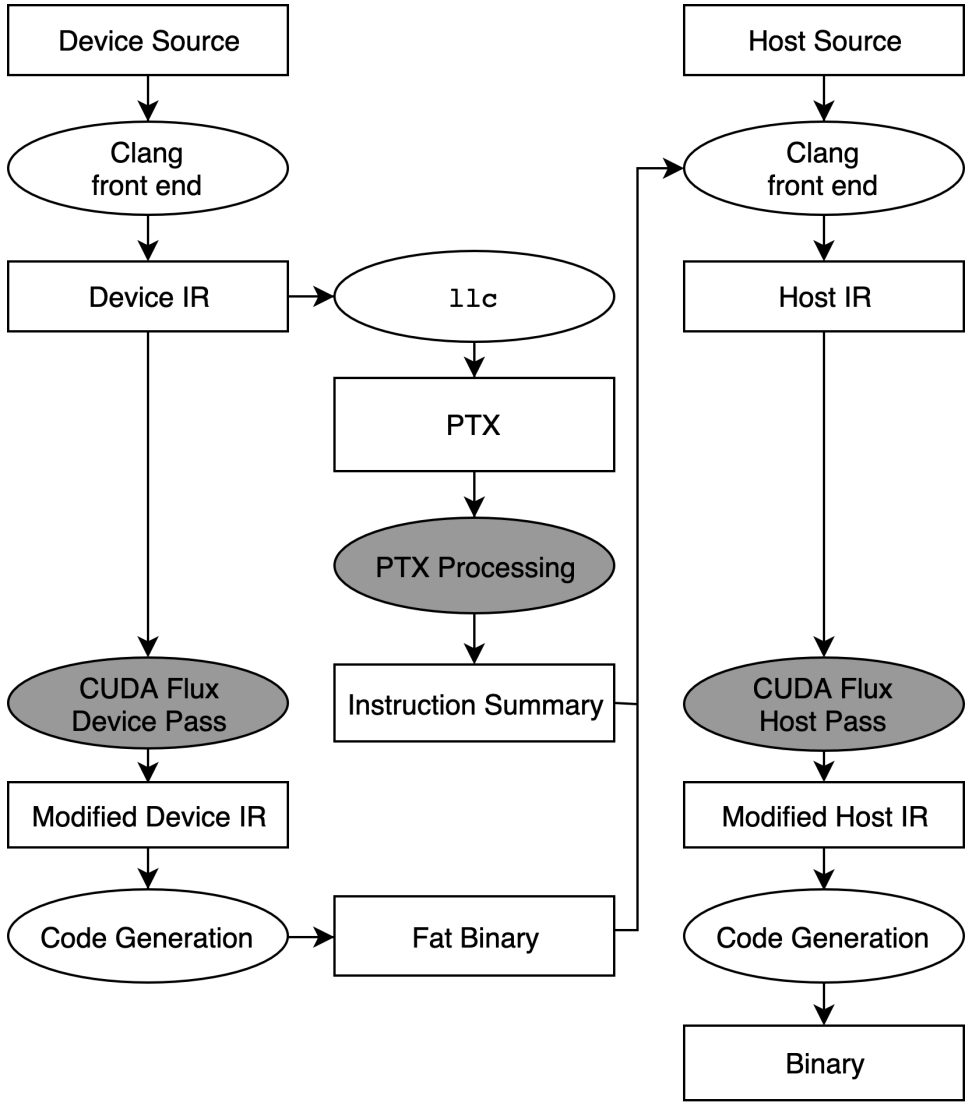
- CUDA Flux Device Pass
- PTX Processing
- CUDA Flux Host Pass

Host and device passes run before machine code generation.

Static runtimes to manage instrumentation counters are linked to host and device code before code generation.

PTX Processing iterates over all kernels and produces a PTX block summary which contains instructions counts of all sections in kernel.

Instrumentation on either warp-level, CTA-level or full thread-grid.

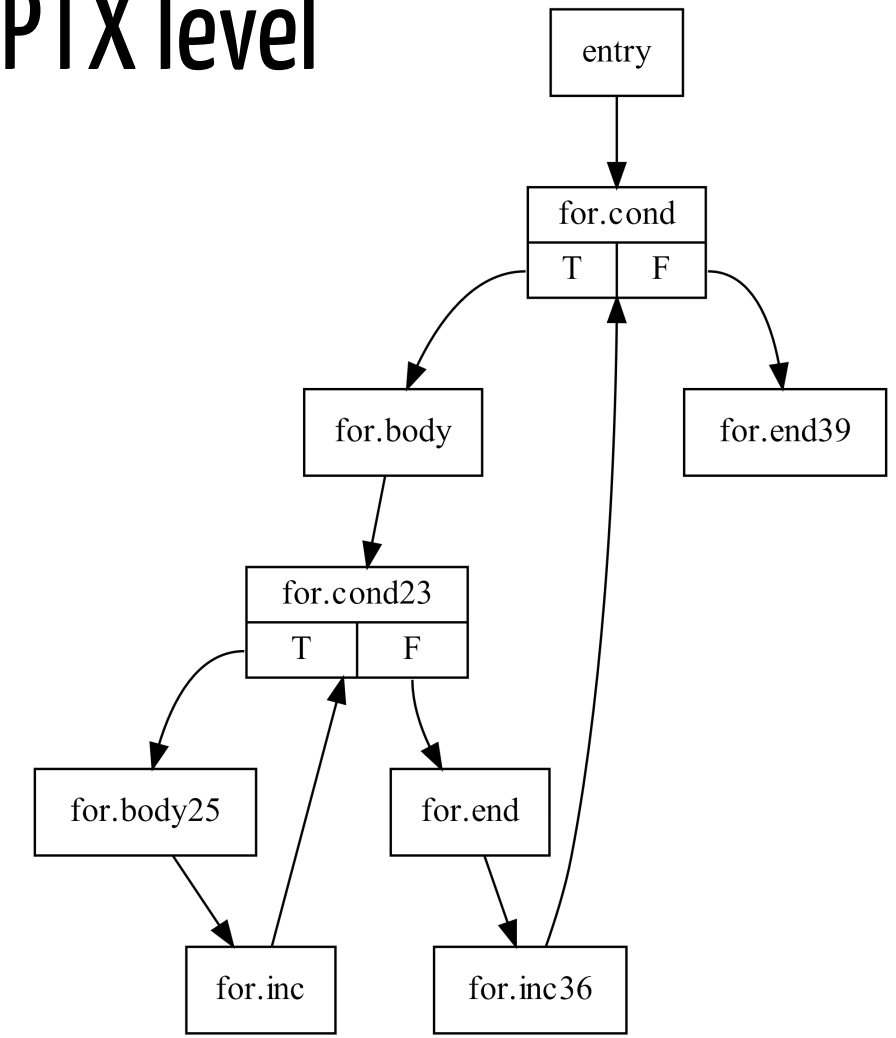


Computing Instruction Counter on PTX level

- Each `Basic Block` (BB) is instrumented.
- On entering a BB the corresponding counter for the block is increased.
- After kernel execution: PTX instruction counter are calculated using BB counter and the PTX instruction summary.

Advantages:

- Fine grained instruction counter regardless of GPU used and supported profiling metrics
- Profiling time does not depend on number of metrics monitored
- PTX is an accessible intermediate assembly for CUDA GPUs
  - Less changes, easier to keep updated
  - Same PTX code for GPU in with same compute capability



CFG for 'matrixMul' function

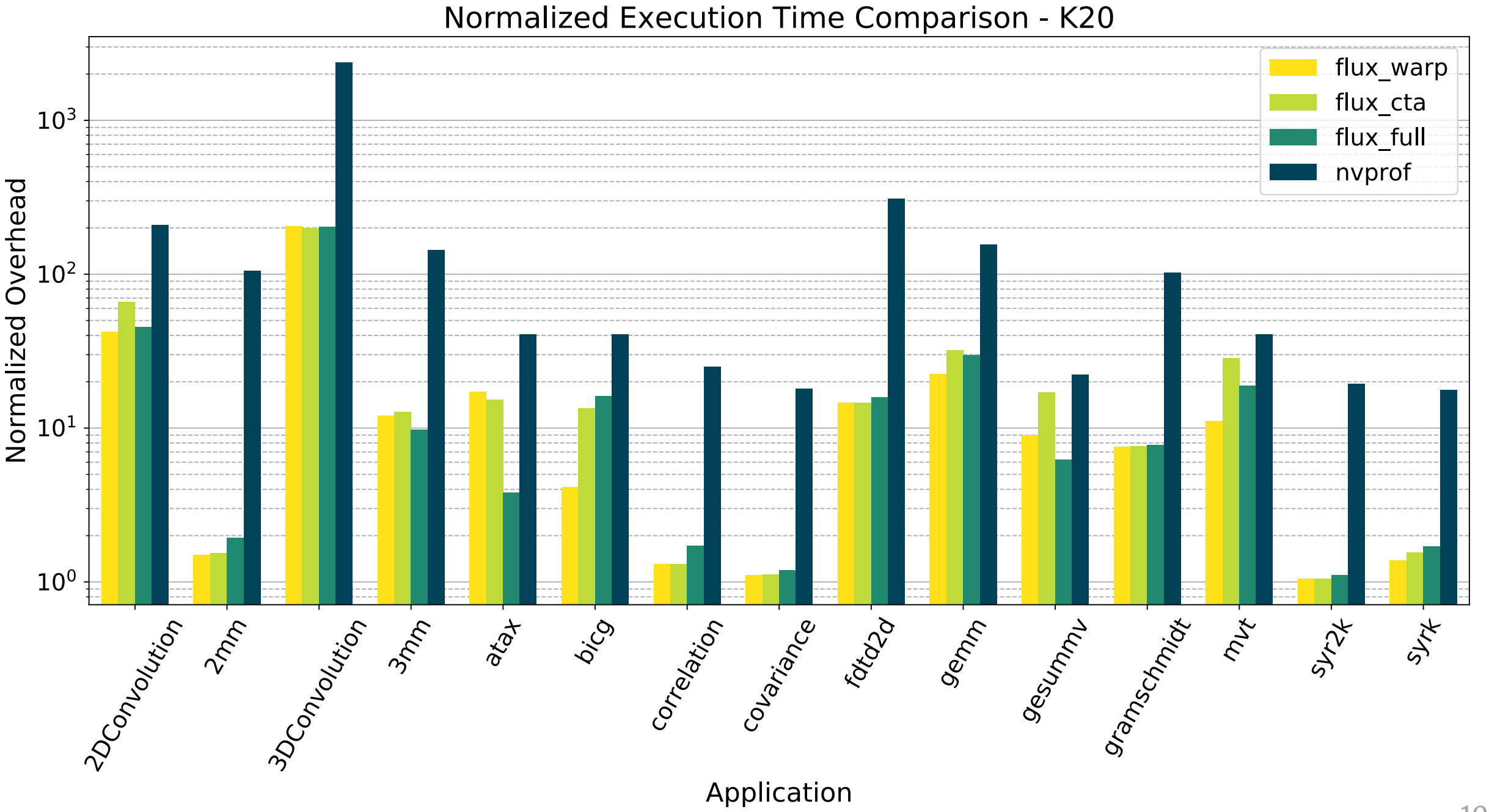
Limitations

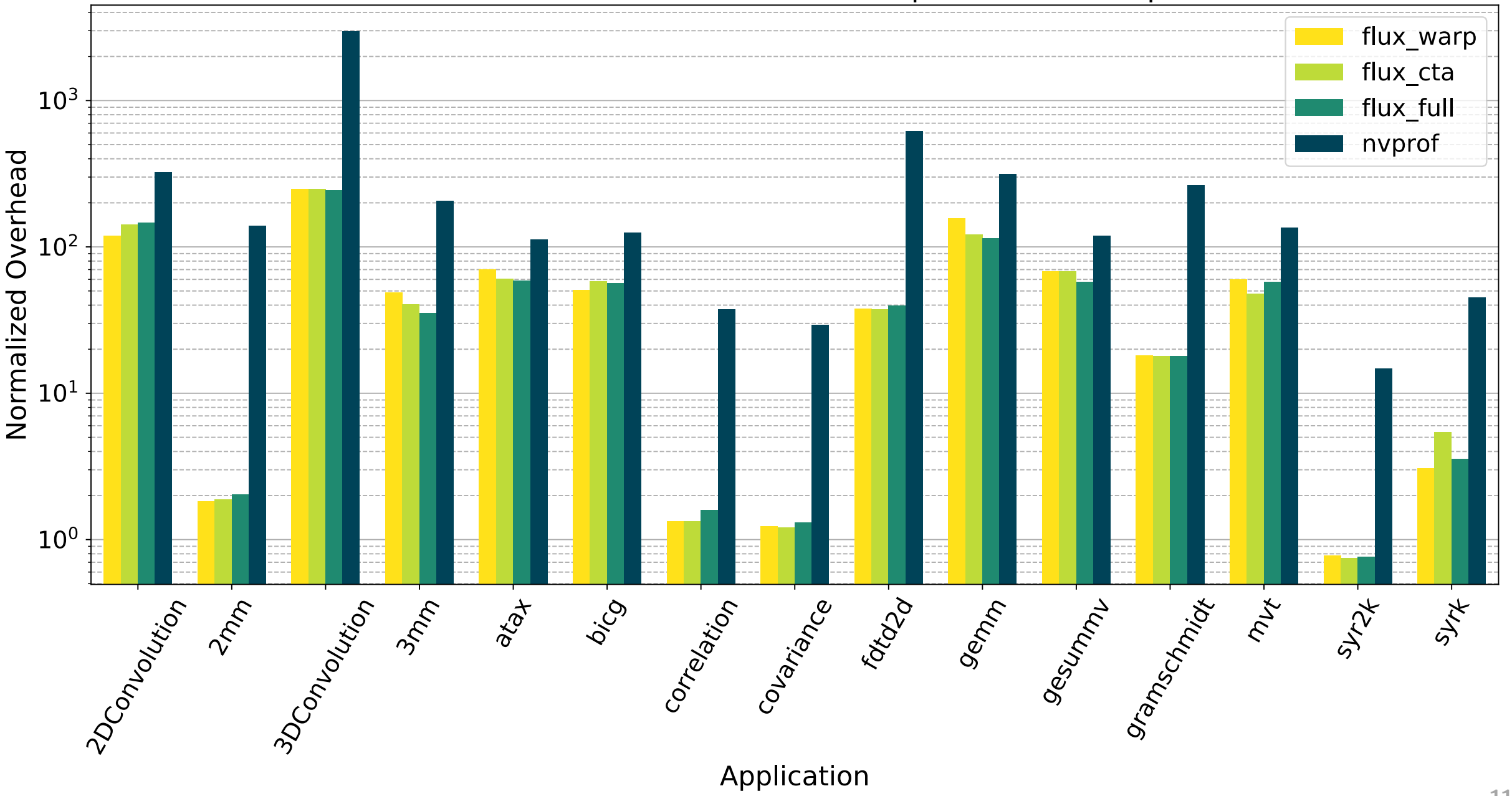
- Profiling on PTX level, not SASS
- Kernel definition and kernel launch need to be in the same compilation module
- Modification of build system needed (in majority of cases):
  - Change `nvcc` to `clang++`
  - Non compatible compiler flags
  - Easy on good/simple build systems, error-prone on complicated build systems
- Instrumentation takes place at IR level
- Applications with texture memory are not supported (clang limitation)

Performance Evaluation

CUDA Flux vs. nvprof:

- Polybench-GPU Benchmark
- Measurements on NVIDIA Tesla K20 and Titan Xp
- Four different profiling configurations:
  - **flux_warp:** all threads of one single warp
  - **flux_cta:** all threads of one single CTA (aka. threadblock)
  - **flux_full:** all threads of the complete threadgrid
  - **nvprof:** measurement with 8 different metrics instruction counter metrics
- Baseline measurement without any instrumentation or profiling is used to normalize the results
- Time measurements:
  - Only kernel time is measured
  - Median of five executions


Normalized Execution Time Comparison - K20
flux_warp
flux_cta
flux_full
nvprof
Normalized Overhead
10^3
10^2
10^1
10^0
2DConvolution
2mm
3DConvolution
3mm
atax
bicg
correlation
covariance
fdtd2d
gemm
gesummv
gramschmidt
mvt
syr2k
syrk
Application


Normalized Execution Time Comparison - TitanXp
flux_warp
flux_cta
flux_full
nvprof
Normalized Overhead
10^3
10^2
10^1
10^0
2DConvolution
2mm
3DConvolution
3mm
atax
bicg
correlation
covariance
fdtd2d
gemm
gesummv
gramschmidt
mvt
syr2k
syrk
Application

Overhead Summary

Tesla K20:

	flux_warp	flux_cta	flux_full	nvprof
min	1.05	1.05	1.10	17.63
mean	23.42	27.56	24.22	241.88
max	204.98	199.71	202.44	2378.64

Titan Xp:

	flux_warp	flux_cta	flux_full	nvprof
min	0.78	0.75	0.76	14.76
mean	59.21	57.04	55.92	363.59
max	248.68	250.24	244.25	2966.64

Outlook

- Optimizations
  - Basic block instrumentation
  - Counter implementation
  - Warp/CTA performance
- In-source integration into Clang/LLVM
- Compile-time analysis for semi-dynamic control flow graphs
- Performance modelling

Conclusion

PTX Level Instrumentation:

- Portable - PTX traces do not depend on specific GPU
- Fine Grained - Each instruction (sub)-type can be counted individually
- Lightweight Implementation - about 2200 lines of code

Performance:

- Overhead of ~25x (K20) up to ~60x (Titan Xp)
- Profiling mode does not affect overhead significantly
- Speed-up compared to profiling with nvprof of about ~10x (K20) to ~6x (Titan Xp)

Github:
https://github.com/UniHD-CEG/cuda-flux