
# POSTER: Simplifying Low-Level GPU Programming with GAS

Da Yan
HKUST
dyanab@cse.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

Xiaowen Chu
Hong Kong Baptist University
chxw@comp.hkbu.edu.hk

## Abstract

Many low-level optimizations for NVIDIA GPU can only be implemented in native hardware assembly (SASS). However, programming in SASS is unproductive and not portable.

To simplify low-level GPU programming, we present GAS (Gpu ASsembly), a PTX-like language that provides a stable instruction set across hardware architectures while giving programmers a low-level control of code execution. We demonstrate that GAS can be used with ease for low-level benchmarking and performance tuning in the context of Tensor Core HGEMM.

***CCS Concepts:*** • **Computer systems organization** → *Multicore architectures*; • **Software and its engineering** → *Source code generation.*

***Keywords:*** GPU, SASS, compiler

## 1 Introduction

CUDA programs are portable across different GPU architectures, thanks to the PTX abstraction. PTX provides a stable instruction set in spite of the changing hardware ISA. PTX instructions are further compiled to machine assembly code (SASS) on a target device.

The portability of PTX programs comes at a performance cost. Prior work [3, 5] shows that the compiler-generated SASS code is often suboptimal. Low-level optimization techniques have hence been proposed [1, 5]. Important operators in vendor libraries [2] are written in SASS.

However, SASS programming is unproductive, as it requires programmers to handle low-level details manually. Also, a SASS program is bound to a specific GPU architecture. For example, the Neon [1] deep learning framework has many hand-crafted SASS kernels (e.g., convolution). Neon demonstrates high performance on Maxwell and Pascal GPUs, but fails to work on the later Volta and Turing devices.

This paper presents GAS [1] (Gpu ASembly), a PTX-like language and compiler that simplify low-level GPU programming. GAS allows programmers to write *portable code* with ease while also giving them sufficient control of *low-level execution*. GAS has three major design goals. (1) **Productivity**: GAS should be as easy to program as PTX; (2) **Capability**: GAS should enable SASS-level optimizations; (3) **Portability**: GAS programs should be portable across architectures. GAS employs three designs to achieve these goals: (1) hiding *register allocation* and *data hazard resolving* from users, since they are usually easy for algorithms but tedious for programmers; (2) leaving *instruction selection* and *instruction scheduling* to users, since they are critical for performance but are hard for compilers to find the optimal solution; (3) providing a stable virtual instruction set, which is an abstraction of the core SASS instructions.

GAS now supports NVIDIA Volta and Turing GPUs, and can be easily extended to the newly released Ampere GPUs. In this paper, we evaluate the effectiveness and simplicity of GAS through two case studies: micro-benchmarking and HGEMM.

### 1.1 The GAS Compiler Architecture

The GAS compiler adopts a modular design. Fig. 1 shows an architecture overview. The compiler takes as input a GAS source file (`.gas`) and generates a CUDA binary file (`.cubin`). The GAS compiler is primarily composed of the parser, the decision making algorithms for register allocations and control information setting, and the output `cubin` file writer.

## 2 Case Studies

In this section, we present two case studies to demonstrate how GAS makes low-level GPU programming simplified. We compare the performance of GAS with the most recent `ptxas` 11.0 and cuBLAS 11.0 on NVIDIA Volta V100 and Turing RTX2070.

### 2.1 Case Study 1: GPU Micro-Benchmarking

We first show that GAS can be used for portable low-level micro-benchmarking. We use GAS to detect the instruction's CPI (cycles per instruction) and occupied pipe (Table 1), and characterize the instruction cache behavior (Fig. 2). This kind

[1] GAS is open-sourced at https://github.com/daadaada/gas


*PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea*

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441591

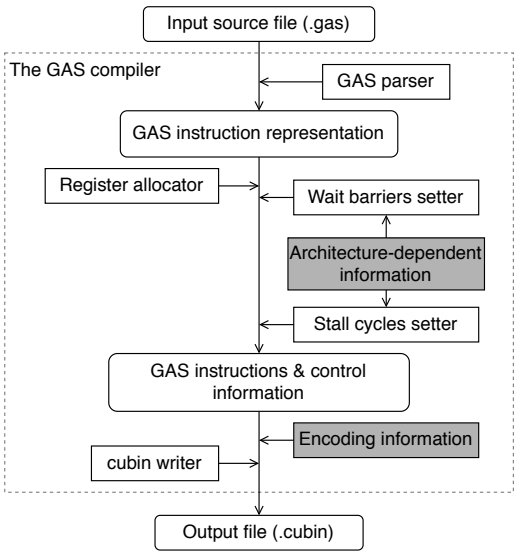


**Figure 1.** Overview of the GAS compiler architecture. The parser, the internal data structures, and the algorithms (e.g., register allocator) are architecture-independent. The architecture-specific information (gray boxes) is handled separately.

of benchmarking cannot be performed at the PTX level, as the `ptxas` reorders instructions and eliminates dead code.

**Table 1.** CPI on different GPUs.

| Instruction | Volta | Turing | pipe |
|---|---|---|---|
| `ffma` | 2 | 2 | FP32 |
| `imad.wide` | 4 | 4 | FP32 |
| `hmma.1688.f16` | N/A | 8 | TC/FP32/FP16 |
| `lds.32` | 2.10 | 2.13 | LDST |

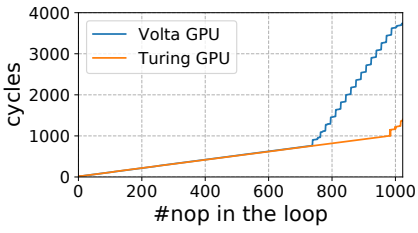


**Figure 2.** The L1 icache behavior on Volta and Turing GPUs. The L1 icache on Volta can hold around 740 instructions, while L1 icache on Turing can hold around 980 instructions.

GAS also helps to detect that the buffers of load/store units can hold at most 5 `lds.64`s on Turing GPUs.

## 2.2 Case Study 2: Tensor Core HGEMM

We use Tensor Core HGEMM as an example to demonstrate GAS's ability to optimize performance by tuning the instruction order. Our implementation follows the practice in [4]. Fig. 3 presents the code snippet of Tensor Core HGEMM in GAS.

```
// smem pre-fetching
hmma.1688.f16 c[0:1], a0[0:1], b0[0], c[0:1];
lds.32 a1[0], [loadAs+16];
hmma.1688.f16 c[2:3], a0[2:3], b0[0], c[2:3];
lds.32 a1[1], [loadAs+592];
```

**Figure 3.** Code snippet of Tensor Core HGEMM in GAS.

Fig. 4 shows the evaluation results on a Turing RTX2070. Our GAS implementation achieves on average 1.23× speedup over the equivalent PTX implementation, and achieves on average 1.45× speedup over cuBLAS 11.0.

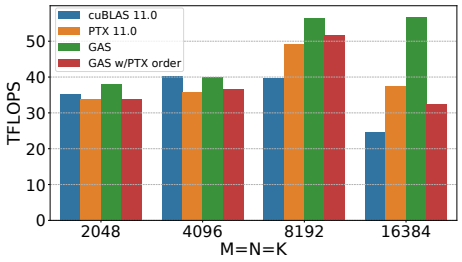


**Figure 4.** Tensor Core HGEMM's throughput on RTX2070.

The performance advantage of GAS comes from its optimized instruction order. To see this, we implemented a GAS HGEMM with the same instruction order as that of the `ptxas` generated code (*GAS w/PTX order*). As shown in Fig. 4, this implementation only achieves a comparable performance as `ptxas` (0.98× speedup on average).

## References

[1] NervanaSystems. 2016. *Neon.* Retrieved Jan 12, 2020 from https://github.com/NervanaSystems/neon

[2] NVIDIA. 2020. *cuBLAS.* Retrieved Aug 12, 2020 from https://docs.nvidia.com/cuda/cublas/index.html

[3] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18).* ACM, Vienna, Austria, 168–182.

[4] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS'2020).* IEEE, New Orleans, LA, USA, 634–643.

[5] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17).* ACM, Austin, TX, USA, 31–43.

# A Artifact Appendix

## A.1 Abstract

The artifact contains the code for the GAS compiler and examples including micro-benchamarking, SGEMM, and Tensor Core HGEMM. GAS targets at NVIDIA Volta and Turing GPUs. Building and running the examples will result in plots similar to the ones in Table 1, Fig. 2, and Fig. 4.

## A.2 Artifact check-list (meta-information)

- **Algorithm:** The artifact includes linear scan register allocation algorithm, and the algorithms for control logic setting.
- **Compilation:** The experiments in the paper used g++ version 9.3.0, which support c++17. The experiments requires nvcc version 11.0, which supports `mma` Tensor Core instructions.
- **Run-time environment:** A Linux operating system should be used. The experiments in the paper used Ubuntu 18.04.
- **Hardware:** An NVIDIA Volta or Turing GPU should be used. The experiments in the paper used a V100 GPU and an RTX2070 GPU.
- **Output:** Throughput for each execution is written to text files.
- **Experiment workflow:** Clone the repository and use the provided scripts to run the experiments.
- **How much disk space required (approximately)?:** 100 MB
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 30 minutes
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.4314598

## A.3 Description

**A.3.1 How to access** The artifact is available on Github at https://github.com/daadaada/gas

**A.3.2 Hardware dependencies** An NVIDIA Volta or Turing GPU.

**A.3.3 Software dependencies** A Linux operating system with CUDA toolkit 11.0 should be used to run the experiments. The artifact requires g++ version 9.3.0.

## A.4 Installation

The source code can be built using the cmake command. Detailed guidelines can be found in the `README.md` file in the repository.

## A.5 Evaluation and expected results

The results of instruction CPI experiments correspond to the numbers reported in Table 1. The results of the L1 instruction cache benchmarking corresponds to the number reported in Fig. 2. The results of the HGEMM experiments correspond to the numbers reported in Fig.4.

## A.6 Experiment customization

The CPI of other instructions can be detected by changing the instructions in the experiment.