POSTER: An LLVM-based Open-Source Compiler for NVIDIA GPUs

Da Yan
HKUST
dyanab@cse.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

Xiaowen Chu
HKUST(GZ)
xwchu@ust.hk

Abstract

We present GASS, an LLVM-based open-source compiler for NVIDIA GPU's SASS machine assembly. GASS is the first open-source compiler targeting SASS, and it provides a unified toolchain for currently fragmented low-level performance research on NVIDIA GPUs. GASS supports all recent architectures, including Volta, Turing, and Ampere.

Our evaluation shows that our specialized optimizations deliver significant speedup over LLVM's algorithms.

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

Keywords: GPU, SASS, compiler, LLVM

1 Introduction

The compiler toolchain for NVIDIA GPUs is not yet fully open-sourced. At the time of writing, the only way to automatically generate NVIDIA GPU machine assembly (SASS) [2] is to use NVIDIA's proprietary compiler, ptxas [1]. The lack of open-source support makes many optimizations inaccessible to the public.

In this paper, we present GASS, an open-source GPU compiler that translates LLVM-IR into optimized SASS code.

GASS employs the following optimization techniques in compilation: 1. **Instruction scheduling:** We design and implement optimized instruction scheduler for compute-bound kernels. The evaluation shows that for matrix multiplication, our scheduler improves the performance by 5% to 23%, compared to LLVM's scheduler. 2. **If-conversion:** We place our if-conversion pass, which translates short branches into predicated instructions, before instruction scheduling. This pass gives 23% extra throughput for matrix multiplication.

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508428

GASS opens up a new avenue for a wide spectrum of GPU research in the future, such as **• Branch divergence analysis. • Performance modeling. • Register allocation. • Compile-time instrumentation.**

GASS is based on LLVM 12.0, and is implemented in C++ and LLVM's TableGen language in about 14k lines of code.

We summarize our major contributions as follows: 1. We present GASS [1], the first open-source compiler targeting SASS, making implementations of new GPU optimizations much easier on NVIDIA GPUs. 2. We introduce our high-performance instruction scheduler for compute-bound kernels. 3. We evaluate optimizations including if-conversion and instruction scheduling.

1.1 Background: CUDA Compilation Trajectory

NVIDIA GPUs can only execute SASS code. There are several ways to compile high-level languages to SASS, as Fig 1 shows.

The first approach is to use the toolchain provided by NVIDIA. **nvcc** compiles CUDA C++ to PTX. The PTX code is then compiled to SASS by **ptxas**. Note that both nvcc and ptxas are proprietary compilers and are close-sourced.

Another approach is to leverage the NVPTX backend in LLVM. Compilers can first generate LLVM-IR, which is further compiled to PTX by the NVPTX backend. Finally, the PTX code is compiled to SASS by ptxas.

Ptxas hides important transformations, including instruction scheduling. In comparison, we present **GASS**, which gives users a full control over the entire compilation pipeline, thus enabling customized optimizations.

1.2 Modeling GPU Hardware

First we need to model the GPU hardware by modeling the register set, the instruction set, and the schedule model.

GASS models the GPU **register set** by providing 255 allocatable 32-bit registers and 7 allocatable 1-bit vector predicate register. GASS also supports 64-bit and 128-bit vector registers. We also model uniform registers, which store values that are shared between all threads within a warp.

The GASS **instruction representation** is a direct SASS representation. Each GASS instruction has its opcode and a list of operands and flags. A GASS operand can be a register, immediate, or constant memory address. Each GASS instruction has a predicate mask, which can mask the execution of the instruction.

[1] https://github.com/hkust-adsl/gass

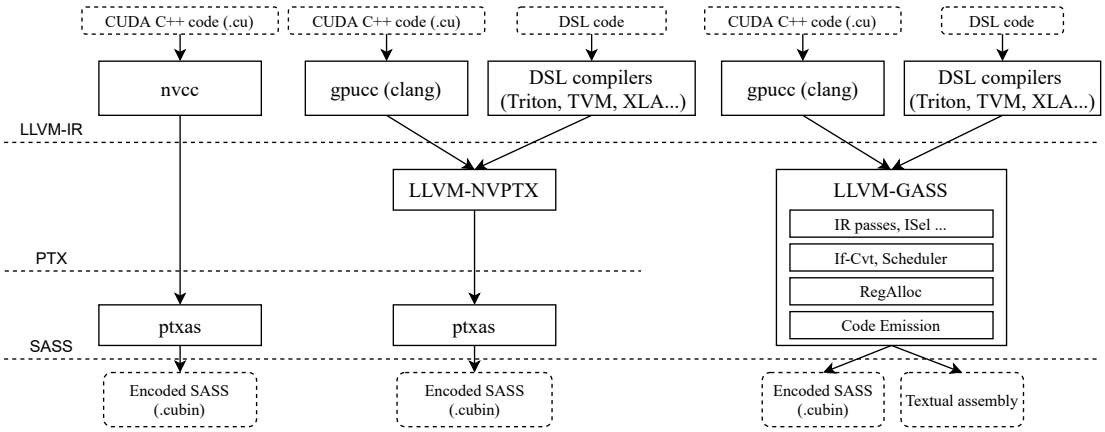

Figure 1. Different CUDA compilation trajectories.

The **scheduling model** describes the latency and throughput of each instruction. Such information is critical for instruction schedulers to generate high-quality outputs. We conduct micro-benchmarking to retrieve enough information to construct complete scheduling models.

1.3 Compilation Pipeline

The LLVM-based GASS backend takes LLVM-IR as input, and generates encoded SASS code (.cubin) or textual assembly.

The input LLVM-IR first go through the **IR passes**, including Dead Code Elimination (DCE). After IR passes, GASS does **instruction selection** (ISel), which translates LLVM-IR to the GASS instruction representation.

After common SSA optimizations (e.g., Loop Invariant Code Motion) the program needs to exit SSA to get ready for **Scheduling** (PreRA Sched) and **Register Allocation** (RegAlloc). We insert our customized **if-conversion** before the PreRA schedule pass. The if-conversion pass transforms short branches into predicated instructions. We customize the PreRA instruction scheduler for compute-bound kernels.

NVIDIA GPUs require the compiler to set proper control information to prevent data hazards. After RegAlloc, GASS **sets wait barriers** to prevent data hazards for instructions with variant latency, and **sets stall cycles** to prevent data hazards for instructions with a fixed latency.

Finally, GASS emits binary or textual output.

2 Experimental Results

We evaluate GASS in the context of tensor core matrix multiplication on V100 GPUs. Kernels are generated by Triton [3].

We compare GASS with LLVM 12.0's NVPTX backend along with ptxas 11.4.

2.1 Tensor Core Matrix Multiplication on V100

We evaluate Tensor Core matrix multiplication ($C = A \times B$) on V100 GPUs. Both A, B, and C are in `float16`, and the accumulators of computation are of `float32` type. We evaluate two settings, one is to let each thread block load a 128×32 tile of A and a 128×32 tile of B in each iteration. Global memory loads are double-buffered. This setting is denoted as `128x128_32x2`. The other setting is `64x64_32x2`.

As Fig 2 shows, GASS achieves a geometric-average speedup of 1.03×.

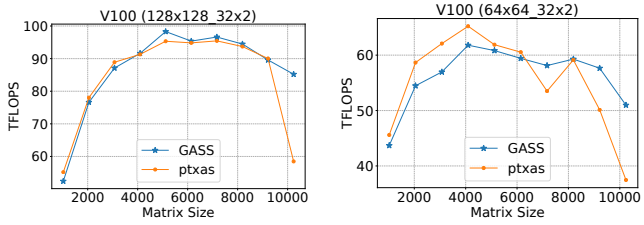

Figure 2. The throughput of Tensor Core matrix multiplication on V100. GASS achieves 1.03× speedup over ptxas 11.4 under different settings.

Ablation study. We evaluate the effectiveness of different optimizations. The if-conversion pass gives 23% extra throughput. The benefits of this pass are in three-fold: • Exposing more opportunities for the instruction scheduler to reorder instructions. • Exposing more opportunities for the barrier setter to set wait barriers. • Eliminating branch instructions. Our specialized instruction scheduler gives us 5% higher throughput over LLVM's scheduler as it gives a more balanced schedule, avoiding overwhelming load/store units.

References

[1] NVIDIA. 2019. *Parallel Thread Execution ISA.* Retrieved Jul 2, 2021 from https://docs.nvidia.com/cuda/parallel-thread-execution/

[2] NVIDIA. 2020. *CUDA Binary Utilities.* Retrieved Aug 1, 2021 from https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html

[3] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'19).* ACM, Phoenix, AZ, USA, 10–19.