Efficient MLIR Compiler Design: Vectorization for Presburger Library

Zhou Qi

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

This report presents a faster implementation for the core `pivot` function of MLIR's presburger library. Its hot loop is element-wise overflow-checked multiplication and addition on an input matrix of low dimension and mostly small value elements.

The current approach of upstream is element-wise multiplication and addition on transprecision integer matrices, from `int64_t` to `LargeInteger`. This can be improved by efficiently utilizing hardware resources, taking advantage of SIMD, and reducing the bit width for every element: the compiler is not capable of automatically generating vectorized instructions for element-wise transprecision computing, and `int64_t` has a much larger bit width than what is typically used for most of the elements in the matrix. Additionally, extra arithmetics are required to perform overflow checking for `int64_t`, resulting in significant performance overhead. This report "innovates" the `int23_t` [1] datatype, a 23-bit integer datatype that utilizes the 23-bit mantissa of a 32-bit floating point, to address these issues. The faster "pivot" performs matrix-wise transprecision computing, targeting 99% TODO: confirm this number of the case where elements fit inside `int23_t`. Overflow awareness overhead is almost free, as floating point imprecision implies `int23_t` overflow (See Section **??**) and can be captured by a status register. It takes as low as 1 ns to check the status register in the pipeline, and only takes 9 ns to reset the status register. Additionally the status register is only cleared once before a sequence of `pivot` calls, making the average cost of clearing the status register per `pivot` negligible.

On a 30-row by 16-column example matrix, it performs 30 times faster than the upstream scalar implementation. The time cost of a single `pivot` call is reduced from 550 ns to 18.6 ns. TODO: replace this with actual MLIR benchmark result

TODO:Question: mention int16 here?

TODO:Reminder: we don't use int16 for (1) compatibility (2) slightly faster than float

[1] This is not really a innovation. It it a common technique on GPUs because often they are more capable on floating points than integers. See Section 1 for more history and detail.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Zhou Qi*)

Acknowledgements

Any acknowledgements go here.

TODO: Marisa Kirisame

Table of Contents

Chapter 1

Introduction

MLIR, Multi-Level Intermediate Representation, is a infrastructure for building reusable and extensible compilers. Its aim is to reduce fragmentation in domain specific languages and heterogeneous hardwares [5]. Its Presburger library provides polyhedral compilation techniques to make dependence analysis and loop optimization [4] and cache modeling [8]. Presburger arithmetics involves determining whether conjunction of linear arithmetic constraints is satisfiable [2], and can be solved using the simplex method of linear programming, with its core function "pivot" consumes TODO: find the number % [6] of the runtime.

The `pivot` function involves two multiplication and one addition operation on every element in a matrix. Notably, the input matrices for this library tend to exhibit characteristics of small values and low dimensionality. For example, 90% of test cases work with 16-bit integers that never overflow, and 74% of isl's runtime is spent on test cases that we can compute using 16-bit integers and matrices with at most 32 columns [7]. These properties can be leveraged to take advantage from modern micro-architectural hardware resources, thereby accelerating the process.

Currently, the source code in MLIR upstream adopts a nested for-loop to iterate through every element of the matrix in a transprecision manner. Each number in the matrix can either be `int64_t` or `LargeInteger`. The algorithm starts by using `int64_t`, in case of overflow, it switches to the `LargeInteger` version. This approach is computationally expensive and inefficient, for the following reasons:

1. `int64_t` has a much larger bit width than what is typically used for most of the elements in the matrix,
2. the compiler is not capable of automatically generating vectorized instructions to further optimize the process,
3. overflow is checked manually through additional arithmetic operations.

To propose a faster alternative of the `pivot` function, we could consider constructing a new `pivot` algorithm that satisfies the following conditions:

1. Utilize SIMD: preliminary benchmarks (see Section **??**) indicate 8x TODO:

verify this number performance improvement on a simple vector element-wise add example.

2. Use small bit width for every element: reducing bit width by half doubles the amount of numbers packed into a single vector register, and essentially reduces the instruction count by half. (Some figure TODO)

3. Fast overflow checking: for integers, overflow has to be checked manually and this introduces 60% (TODO: verify this number?) overhead, as benchmarks in the Section **??** shown. This is because the x86 architecture does not provide status registers to indicate integer overflown. However, there is one for floating points, making floating points overflow detection almost free.

opencompl's fork of LLVM includes a modified version of `pivot` that utilizes `int16_t` and is designed for matrices with 32 columns or less [7]. This approach offers the advantage of being able to pack a row of 32 elements into a single `AVX-512` register and addresses issues 1 and 2. However, overflow is still checked manually, causing 4x or 5x more instruction count (Section **??**). Moreover, this approach introduces a new disadvantage, the support for vectorized `int16_t` is very rare among CPU manufactured in the last decade (Section **??**).

An alternative approach is to do 23-bit or 52-bit integer operations using float (32-bit floating point) or double (64-bit floating point) respectively. Though floating points are notorious for precision issues, they are reliable when representing integers that fit inside their mantissa, 23 bits for float and 52 bits for double [1]. When the result of some integer computation exceeds the bit size of the mantissa, floating point imprecision almost always occurs and a status register will be set automatically (Section **??**). Comparing to `int16_t`, even though vector size is sacrificed as there does not exist support for 16-bit floating point `half`, using floating points could still potentially be faster, because overflow checking overhead can be significantly reduced. With floating points, the cost of overflow checking is the time spent on resetting the status register once at the beginning of a sequence calls to 'pivot', plus reading it, per `pivot` call. Benchmarks TODO: some figure indicate that the total overhead is as low as 1 ns, as it only adds 1 ns to read the status register in the instruction execution pipeline, and the average cost of resetting per pivot is negligible.

This report will first analyze the capability of modern CPU micro-architecture under various configurations such as vectorization method (clang automatic from scalar code, AVX intrinsic, clang builtin vector type), matrix data structure (nested list or flat list), and element data type (int16, int32, int64, float, double), through a matrix element-wise fused-multiply-add toy example and a copy of the pivot function detached from the Presburger library. TODO: Fix this paragraph

Then we conclude that AVX intrinsic with clang builtin vector type and flat list matrix is superior to their counterpart. With overflow ignored, int16 is the fastest, but floating points are faster when overflow checking is turned on. The most optimal configuration takes 20 ns to finish pivot on a representative and

[1] IEEE 754 specification is introduced in Section **??**

general case input matrix of 30 rows by 16 columns, achieving 30x faster than the upstream version.

TODO: Update here after Integrating into library

Chapter 2

Background

2.1 Modern CPU micro-architecture

2.2 code block example

```
\begin{preliminary}
    ...
\end{preliminary}
```

2.3 Modern CPU micro-architecture

Computer architecture textbooks often deceipts

2.4 Citations

Citations (such as [1] or [3]) can be generated using `BibTeX`. For more advanced usage, we recommend using the `natbib` package or the newer `biblatex` system.

These examples use a numerical citation style. You may use any consistent reference style that you prefer, including "(Author, Year)" citations.

Chapter 3

Your next chapter

A dissertation usually contains several chapters.

Chapter 4

Conclusions

4.1 Final Reminder

The body of your dissertation, before the references and any appendices, *must* finish by page 40. The introduction, after preliminary material, should have started on page 1.

You may not change the dissertation format (e.g., reduce the font size, change the margins, or reduce the line spacing from the default single spacing). Be careful if you copy-paste packages into your document preamble from elsewhere. Some LaTeX packages, such as `fullpage` or `savetrees`, change the margins of your document. Do not include them!

Over-length or incorrectly-formatted dissertations will not be accepted and you would have to modify your dissertation and resubmit. You cannot assume we will check your submission before the final deadline and if it requires resubmission after the deadline to conform to the page and style requirements you will be subject to the usual late penalties based on your final submission time.

Bibliography

[1] Hiroki Arimura. Learning acyclic first-order horn sentences from entailment. In *Proc. of the 8th Intl. Conf. on Algorithmic Learning Theory, ALT '97*, pages 432–445, 1997.

[2] Mikolaj Bojanczyk and Joël Ouaknine. A simple and practical linear-time algorithm for presburger arithmetic. 2004.

[3] Chen-Chung Chang and H. Jerome Keisler. *Model Theory*. North-Holland, third edition, 1990.

[4] LLVM Contributors. Mlir llvm. https://mlir.llvm.org/docs/Dialects/Affine/, 2023.

[5] LLVM Contributors. Mlir llvm. https://mlir.llvm.org/, Accessed on 2023-04-03.

[6] Grosser et al. Fast linear programming through transprecision computing on small and sparse data. In *Proceedings of the ACM on Programming, Languages, Volume 4*, 2020.

[7] Pitchanathan et al. Fpl: fast presburger arithmetic through transprecision. In *Proceedings of the ACM on Programming Languages, Volume 5*, 2021.

[8] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. A fast analytical model of fully associative caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, jun 2019.

Appendix A

First appendix

A.1 First section

Any appendices, including any required ethics information, should be included after the references.

Markers do not have to consider appendices. Make sure that your contributions are made clear in the main body of the dissertation (within the page limit).

Appendix B

Participants' information sheet

If you had human participants, include key information that they were given in an appendix, and point to it from the ethics declaration.

Appendix C

Participants' consent form

If you had human participants, include information about how consent was gathered in an appendix, and point to it from the ethics declaration. This information is often a copy of a consent form.