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) only 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

First and foremost, I would like to express my deepest gratitude to my supervisor, Tobias Grosser, for his unwavering support, guidance, and encouragement throughout the course of this research. His extensive knowledge, valuable insights, and patience have been instrumental in shaping my academic and personal growth.

I am also immensely grateful to Arjun Pitchanathan, a fellow Ph.D. student under the supervision of Tobias Grosser, for his invaluable assistance and collaboration. His expertise, constructive feedback, and willingness to share his knowledge have significantly contributed to the progress and quality of this work.

Furthermore, I would like to acknowledge the generous contributions from Marisa Kirisame, Emanon, gjz010, and lyzh, whose insightful comments, technical support, and camaraderie have enriched my research experience. Their shared wisdom and passion for the subject matter have made this journey both enjoyable and rewarding.

Finally, I am thankful to all my friends, colleagues, and family members who have provided me with the emotional support and encouragement needed to complete this paper. Their unwavering belief in my abilities has been a source of strength and motivation throughout this challenging process. TODO:

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 [4]. Its Presburger library provides polyhedral compilation techniques to make dependence analysis and loop optimization [3] and cache modeling [7]. 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 % [5] 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 [6]. 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 [6]. 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, especially `Zen4`, through a matrix element-wise fused-multiply-add toy example under the various configurations regarding vectorization methods, matrix data structures, element data types and data widths (Section 3).

It is discovered that optimal performance can be achieved by selecting clang builtin vector type as vectorization methods and use flat list as matrix data structure. However, it is quite difficult to decide whether `int16_t` or `float` is better, because the former benefits form bigger vector size and less instruction count, while the latter has minimal overhead on overflow checking.

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

Then two detached versions of `pivot` function from the Presburger library are built from the most optimal configurations derived form the toy example, one using `int16_t` and the other using `float`. Some further optimizations were made by inspecting `perf` reports and assembly, including:

1. Reduce memory operation
2. Unroll loops
3. TODO: what are the other optimizations?

TODO: Update here after Integrating into library

Chapter 2

Background

2.1 Presburger library

2.1.1 Overview

The FPL paper obtained 465,460 representative linear problems encountered during integer set coalescing by analyzing linear programs in cache analytical modeling, polyhedral loop optimization, and accelerator code generation. It is found that most of the constraint matrices are low in dimensionality and small in the value of each element. Specifically, more than 99% of the coefficients require less than 10 bits and 95% of them are less than 20 columns [5]. Thus, most of the rows fit inside a 512-bit vector register of 32 `int16_t` elements, and a row operation can be done in a single instruction.

However, in rare and corner cases, there can be larger coefficients up to 127 bits. Practically, the upper bound of coefficient size is unknown, making it required to have arbitrary precision arithmetic `LargeInteger` as a backup. Also, the maximum observed column count is 28 and there is not a certain maximum column count as well.

Therefore the FPL paper presents a 3-way transprecision implementation for the Presburger library, from row-wise vectorized `int16_t` to element-wise scalar `int64_t` and element-wise scalar `LargeInteger`, as illustrated in Figure 2.1. But unfortunately the MLIR upstream only presents a 2-layer transprecision, consisting of element-wise scalar operation using `int64_t` and `LargeInteger`. The `int16_t` version is not merged with the upstream for two reasons:

1. `int16_t` vectors require AVX-512 ISA extension, but hardware support are rare (Section 2.2).
2. Despite the `int16_t` version is fast TODO: find how much faster in FPL paper, overflow checking overhead is ??%TODO: find how much is overhead [6]. Using floating points could significantly reduce this overhead and potentially be faster (Section 2.3).

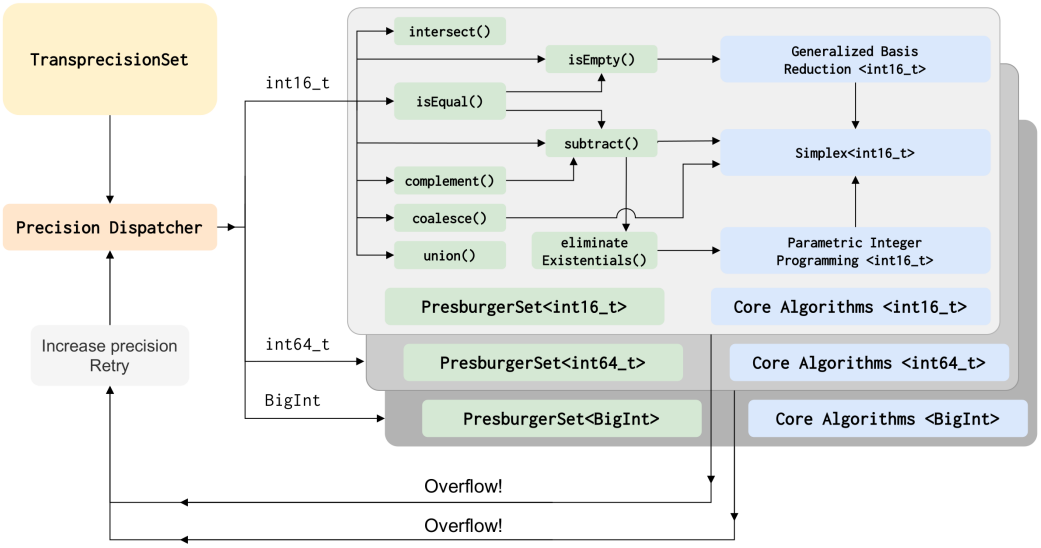

Figure 2.1: The The architecture of FPL.

2.1.2 Implementation of detecting overflow for `int16_t`

By comparing the result of a conventional addition and saturated addition, it indicates whether an addition has gone overflown or not. In case of overflow, with saturated add, the result always retains at the maximum possible value of `int16_t`: `0x7FFF`, while the result of a conventional add is always smaller, because the overflowed output from conventional addition can't go all the way around and become `INT16_MAX` again. In the two's complement binary form for integer, the overflow sum is "trapped" in the negative number space. For example:

```
INT16_MAX + 1 = INT16_MIN = -32768,
INT16_MAX + 2 = -32767,
...
INT16_MAX + INT16_MAX = -2,
```

For multiplication, as two 16-bit numbers produces 32-bit products but only lower 16 bits can be stored, overflow can be detected by checking whether any of the upper 16 bits are set.

Inspecting these approaches from a instruction-level perspective (Table 2.1), when overflow is ignored, both add and multiply takes 1 instruction, `vpaddw` and `vpmullw`. To obtain and process overflow-related information, an additional computation instruction `vpaddsw` or `vpmulhw` is required, followed with 2 or 3 comparison, shuffling and branch instructions: `vpsraw`, `vpcmpneqw` and `kord`. By enabling overflow checking, it brings 4x or 5x more instruction count and 60% more runtime [6]. (TODO: verify this number?)

	Addition	Multiplication
Disabled	`vpaddw %zmm4,%zmm2,%zmm3`	`vpmullw %zmm1,%zmm3,%zmm2`

Enabled	`vpaddw %zmm4,%zmm2,%zmm3`
	`vpaddsw %zmm2,%zmm4,%zmm2`
	`vpcmpneqw %zmm3,%zmm2,%k1`
	`kord %k1,%k0,%k0`
		`vpmullw %zmm1,%zmm3,%zmm2`
		`vpmulhw %zmm1,%zmm3,%zmm3`
		`vpsraw $0xf,%zmm2,%zmm5`
		`vpcmpneqw %zmm3,%zmm5,%k1`
		`kord %k0,%k1,%k0`

Table 2.1: (TODO: write caption)

2.2 Modern CPU micro-architecture

A recent trend in x86-64 architecture's development is to include AVX-512 instruction set architecture (ISA) extension. AVX-512 succeeds AVX2, the vector width is increased from AVX-2's 256 bits to 512 bits. AVX-512 also provides new instructions, for example, `int16_t` saturated addition.

Even though its specification was released by Intel in 2013, it had been unpopular, as it did not bring practical performance improvements. The primary reason was that it consumed a lot more power than usual, causing severe overheating. The micro-architecture Skylake from Intel, and its AVX-512 enabled counterpart Skylake-X is a classic example. Skylake provides 2 256 bits FMA AVX-2 execution units [1] and Intel provides 2 512-bit AVX-512 FMA units by fusing the existing AVX-2 units into a AVX-512 unit, then introduces an additional FMA AVX-512 unit [8]. The additional AVX-512 unit increases the heat flux density of the chip, causing server thermal throttling issues.

Intel attempted to mitigate this problem by introducing the "AVX-offset" mode. When a workload involving AVX-512 instructions is encountered, the CPU automatically enters AVX-offset mode and reduces its clock frequency [9]. However, in practice it is more common to have a mix of control flow, SSE and AVX-512 instructions, causing many scenario could run faster if the additional AVX-512 is disabled (and does not thermal throttle) [10].

AMD's implementation of AVX-512 in Zen4 has slightly less computing power, but much more efficient. Zen4 can be considered as modernized version of Zen3 or Zen2, where Zen2 and Zen3 supports AVX2 by providing 2 FADD units [2] and 2 FMA units of 256-bit width [11]. Zen4 "double-pumps" these existing circuits to create a single 512-bit FADD and a single 512-bit FMA, without introducing any new execution units for floating point arithmetic [10]. Zen2 and Zen3 are reputable for its high performance per watt [12], and it is expected to be better on zen4 with more advanced lithography. Benchmarks (TODO Some section) indicate that this design indeed does not cause throttling in AVX-512 workloads

[1] Fused-multiply-add (FMA) execution units are a type of floating point execution units, capable of doing addition, multiplication or both in a single instruction. See Section 2.3.2.

[2] Floating-point add units (FADD) can execute addition instructions only. They may be considered as simplified FMA

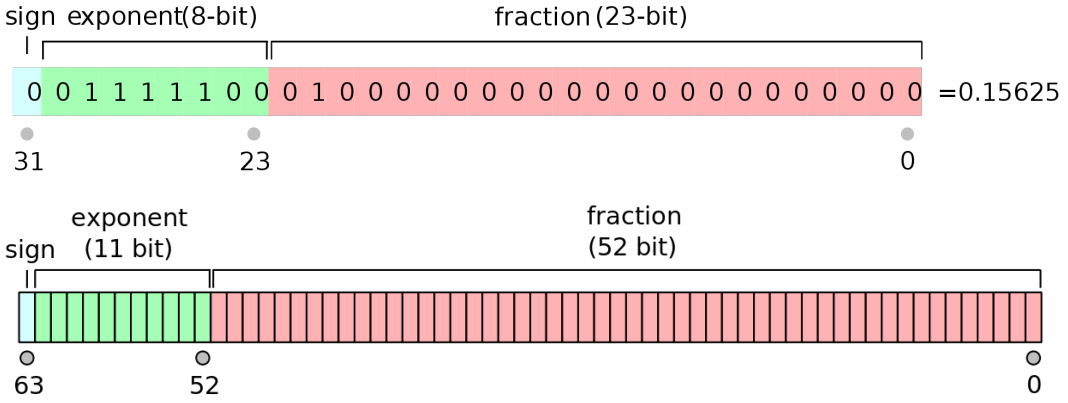

Figure 2.2: IEEE 754 single (32 bits) and double (64 bits) precision floating point. [13]

[10].

Additionally, rebuilding existing software to target AVX-512 may bring slight performance improvement. One benefit of AVX-512 is that it reduces front-end pressure. In the case of Zen4 micro-architecture, though the back-end is possible to commit 2 AVX-2 FADD and 2 AVX-2 FMA every cycle, the front-end has to dispatch 4 instructions per cycle, which is quite difficult. The equivalency in AVX-512 only takes 2 instructions, this is much more likely to be sustained by the frontend [10].

2.3 "int23_t" and "int52_t"

2.3.1 IEEE 754

IEEE 754 is the standard for representing and manipulating floating-point numbers in modern x86 computers. The standard defines several different formats for representing floating point numbers, the most common ones are 32-bit single precision (float) and 64-bit double precision (double). For each format, it specifies how many bits are used to represent the sign, exponent, and mantissa.

For float and double, the sign bit is a single bit that indicates whether the number is positive or negative. As figure 2.2 shows, there are 8 bits and 11 bits for exponent in float and double respectively, to represent represents the order of magnitude. The remaining 23 bits in float and 52 bits in double are mantissae, to store the fractional part of the number. The value of a floating point number can be computed through this formula: $(-1)^s * 2^{(e-B)} * (1 + f)$ where s is sign, e is exponent, f is mantissa and B is a constant bias value: 127 for float, 1023 for double.

2.3.2 Fused-multiply-add

After doing floating point arithmetic, it is required to normalize the result of floating-point arithmetic before it can be used further. However, by feeding the result of a floating-point multiplication (FMUL) directly into the floating-point addition (FADD) logic without the need for normalization and rounding in between, a fused multiply-add (FMA) operation is effectively created: $Y = (A * B) + C$, where A, B and C are the operands, Y is the result [14].

FMA saves cycles and reduces accumulation of rounding errors, while at the same time not adding significant complexity to the circuit. A FMA execution unit is capable to do FMUL, FADD, and FSUB as well:

Addition: $Y = (A * 1.0) + C$
Multiplication: $Y = (A * B) + 0.0$
Subtraction: $Y = (A * -1.0) + C$

This is a useful feature in many numerical computations that involve simultaneous multiplication and addition operations, such as dot product and matrix multiplication. Since the pivot function performs multiplication and addition between the pivot row, some constant value and each row in the matrix, the performance of FMA is critical to the overall efficiency of the algorithm.

2.3.3 Floating point exceptions and Fenv library

Arjun: I feel this is not really background, we are getting into the implementation now

2.4 Google Benchmark

Google benchmark is a library to measure the performance of a code snippet. It provides unit-test like interfaces to setup benchmarks around a code snippet [1]. The given example from https://github.com/google/benchmark is self-explanatory for its usage:

```
#include <benchmark/benchmark.h>

static void BM_SomeFunction(benchmark::State& state) {
    // Perform setup here
    for (auto _ : state) {
    // This code gets timed
    SomeFunction();
    }
}
// Register the function as a benchmark
BENCHMARK(BM_SomeFunction);
// Run the benchmark
BENCHMARK_MAIN();
```

The library first starts a timer, repeatedly executes its core loop: `for (auto _ : state) ...` multiple times then pauses the timer. This method ensures that the results are consistent and minimizes the overhead required for recording the timing information.

Executing the benchmarks will not only report both elapsed real time and CPU time, but also much other useful information to help reduce variance.

```
Running ./build/example
***WARNING*** CPU scaling is enabled, the benchmark real time
measurements may be noisy and will incur extra overhead.
Run on (32 X 5800.00 MHz CPU s)
CPU Caches:
    L1 Data 32 KiB (x16)
    L1 Instruction 32 KiB (x16)
    L2 Unified 1024 KiB (x16)
    L3 Unified 32768 KiB (x2)
Load Average: 8.10, 5.14, 1.14
-------------------------------------------------------
Benchmark                Time             CPU   Iterations
-------------------------------------------------------
BM_SomeFunction       18.5 ns          18.5 ns     37935734
```

The warning: "CPU scaling is enabled, the benchmark real-time measurements may be noisy and will incur extra overhead." is saying that CPU clock frequency is not consistent. It can be dynamically determined by the governor algorithm, according on the system's needs. For example, with the `performance` governor, the OS locks the CPU to the highest possible clock frequency, specified at `/sys/devices/system/cpu/cpu*/cpufreq/scaling_max_freq`, while the `ondemand` governor will push the CPU to the highest frequency on demand and then gradually reduce the frequency as the idle time increases [15].

However, it is also dependent on the manufacture and other hardware constraints. By default, both Intel (Turbo Boost) and AMD (Precision Boost Overdrive) have support for raising clock frequency, beyond the control of the governor [16]. On the other hand, CPUs have self-protecting thermal throttling mechanisms that reduces its clock frequency and voltage when it is too hot.

The benchmark mentioned in this report were performed on a AMD 7950x desktop computer. The computer system went through the following these setups for consistent results:

1. Set the governor to `performance`,
2. Disable AMD Precision Boost Overdrive (or Intel Turbo Boost),
3. Lock clock frequency at a 5 GHz, or any desired fixed value,
4. Make sure heat dissipation is working properly.

2.5 `llvm-mca`

`llvm-mca`, LLVM Machine Code Analyzer, is a tool to analyze performance of executing some instructions on a specific CPU micro-architecture, according to scheduling information provided by LLVM [17]. This tool may help with predicting and explaining performance characteristics TODO: Is it necessary to mention mca? Only use case is I found it not helpful

Chapter 3

Experiments with Toy Example

The pivot function does multiply and add for each row in the matrix, therefore the performance of FMA a simple vector toy can be an effective indicator. This chapter reports performance analysis on simple toy examples that do vector add or vector FMA with various setups, including:

1. Vectorization method
   (a) Clang's automatic vectorization from scalar source code
   (b) Clang builtin vector datatype with occasional AVX intrinsics
2. Matrix data structure
   (a) Nested list
   (b) Flat list
3. Element data width
   (a) 16 bits: `int16_t`
   (b) 32 bits: `int32_t`, `float`
   (c) 64 bits: `int64_t`, `double`
4. Element data type
   (a) Integer
   (b) Floating point

3.1 Vectorization method

3.1.1 Clang's automatic vectorization

Clang is capable of generating vectorized instructions from scalar source code, using the flags `-O3 -march=native` on a platform with vector ISA enabled. Starting with an example (Listing 3.1), the simple `vec_add` function adds every element from two arrays and saves it to the third.

TODO: these hex code are actually incorrect, I assume this does not really matter?

Source code

```
#define size 128
void vec_add(float* src1_ptr, float* src2_ptr, float* dst_ptr) {
    for (uint32_t i = 0; i < size; i += 1 ){
        dst_ptr[i] = src1_ptr[i] + src2_ptr[i];
    }
}
```

Assembly snippet of the hot loop, compiled with `-O3 -march=native` (vectorization on)

```
1458: c4 c1 7c 58 84 87 20   vaddps -0x1e0(%r15,%rax,4),%zmm0,%zmm0
145f: fe ff ff
1462: c4 c1 74 58 8c 87 40   vaddps -0x1a0(%r15,%rax,4),%zmm1,%zmm1
1469: fe ff ff
```

Assembly snippet of the hot loop, compiled with `-O3 -march=native -mno-avx -mno-sse` (vectorization off)

```
120d: d8 44 82 04    fadds  0x4(%rdx,%rax,4)
1211: d9 5c 81 04    fstps  0x4(%rcx,%rax,4)
1215: d9 44 86 08    flds   0x8(%rsi,%rax,4)
```

Listing 3.1

After compiling on a AVX-512 enabled computer and disassembling the binary, it is observed that clang automatically packs 16 `float` (512 bits) as a operand of the `VADDPS` instruction.

Alternatively, vectorization could be disabled by adding the `-mno-avx -mno-sse` flags on top of `-O3 -march=native`. These two sets of flags guarantee that the binary are going to be equally optimized, with the only difference been whether vector instructions are generated or not. In this case, scalar instructions `fadds`, `fstps` and `flds` are selected.

3.1.2 Clang's vector datatype and AVX intrinsics

Another approach is to write source code with vectorization in mind in the first place. Clang provides extension that allows programmers to declare a new type that represents a vector of elements of the same data type. The syntax is `typedef` *`ty vec_ty`* `__attribute__((ext_vector_type(`*`vec_width`*`)))`, where *`vec_ty`* is the name of vector type being defined, *`vec_width`* is its size and *`ty`* is the type of the elements in the vector. For example, `typedef int16_t int16x32 __attribute__((ext_vector_type(32)))` defines a 512-bit vector type of `int16x32`, consisting of 32 `int16_t` and fits inside an AVX-512 ZMM register.

After defining a vector datatype, a vector variable can be created by casting from a pointer of the target array. Then arithmetic operators can be applied between the vectors to performed element-wise operations. The previous `vec_add` example can be rewritten as the code snippet shown in Listing 3.2:

Source code

```
#define size 128
#define FloatZmmSize 16
typedef float floatZmm __attribute__((ext_vector_type(FloatZmmSize)));
void vec_add(float* src1_ptr, float* src2_ptr, float* dst_ptr) {
    for (uint32_t i = 0; i < size; i += FloatZmmSize ){
        floatZmm src1Vec = *(floatZmm *)(src1_ptr + i);
        floatZmm src2Vec = *(floatZmm*)(src2_ptr + i);
        floatZmm resultVec = src1Vec + src2Vec;
        *(floatZmm *)(dst_ptr + i) = resultVec;
    }
}
```

Listing 3.2

3.1.3 Evaluation

When comparing the performance of code written with and without the vector type and examining their assembly, it has been discovered that the the automatic vectorization feature in clang can be unpredictable and may lead to undesired behaviors. It operates as a black box and may take a lot of effort to understand its mechanisms. One of the issues is that clang may select a suboptimal vector width.

Consider the `vec_fma` function in Listing 3.3 and 3.4, a slightly more complicated version of the previous `vec_add` example, where an additional array is introduced and the element-wise operation is changed from addition to FMA. The disassembly reveals that clang decides to use FMA vector instructions of 128-bit width, but when vector size is constrained to 512-bit width by defining a vector type, more optimal binary can be generated. Benchmark indicates that the 512-bit vector width version is xx% faster. TODO: determine this number

Source code

```
void vec_fma(float* src1_ptr, float* src2_ptr,
             float* src3_ptr, float* dst_ptr) {
    for (uint32_t i = 0; i < size; i += 1 ){
        dst_ptr[i] = src1_ptr[i] * src2_ptr[i] + src3_ptr[i];
    }
}
```

Assembly snippet of the hot loop

```
80e4: c5 fa 10 04 b2     vmovss (%rdx,%rsi,4),%xmm0
80f1: c5 fa 10 0c 81     vmovss (%rcx,%rax,4),%xmm1
80fe: c4 c2 79 a9 0c b8  vfmadd213ss (%r8,%rdi,4),%xmm0,%xmm1
810c: c4 c1 7a 11 0c 81  vmovss %xmm1,(%r9,%rax,4)
```

Listing 3.3

Source code

```
#define size 128
#define FloatZmmSize 16
typedef float floatZmm __attribute__((ext_vector_type(FloatZmmSize)));
void vec_fma(float* src1_ptr, float* src2_ptr,
             float* src3_ptr, float* dst_ptr) {
    for (uint32_t i = 0; i < size; i += 1 ){
        floatZmm src1Vec = *(floatZmm *)(src1_ptr + i);
        floatZmm src2Vec = *(floatZmm *)(src2_ptr + i);
        floatZmm src3Vec = *(floatZmm *)(src3_ptr + i);
        *(floatZmm *)(dst_ptr + i) = src1Vec * src2Vec + src3Vec;
    }
}
```

Assembly snippet of the hot loop

```
82c0: 62 b1 7c 48 10 04 80   vmovups (%rax,%r8,4),%zmm0
82c7: 62 b1 7c 48 10 0c 81   vmovups (%rcx,%r8,4),%zmm1
82ce: 62 b2 7d 48 a8 0c 86   vfmadd213ps (%rsi,%r8,4),%zmm0,%zmm1
82d5: 62 b1 7c 48 11 0c 87   vmovups %zmm1,(%rdi,%r8,4)
```

Listing 3.4

In some cases clang could be even worse, it may fail to recognize vectorization patterns from element-wise loop operations, leading to more reduction in performance. In the `vec_fma` example, by changing the type signature from `float` to `int`, clang decides to dispatch scalar instructions for addition (`add`) and multiplication (`imul`)

completely (Listing 3.5). Their vectorized equivalency `vpaddd` and `vpmulld` are more performant options. (Listing 3.5).

Source code

```
#define size 128
void vec_fma(int* src1_ptr, int* src2_ptr,
             int* src3_ptr, int* dst_ptr) {
    for (uint32_t i = 0; i < size; i += 1 ){
        dst_ptr[i] = src1_ptr[i] * src2_ptr[i] + src3_ptr[i];
    }
}
```

Assembly snippet of the hot loop

```
88b0: 44 8b 69 1c  mov    0x1c(%rcx),%r13d
88b4: 44 8b 62 1c  mov    0x1c(%rdx),%r12d
88b8: 45 0f af ee  imul   %r14d,%r13d
88bc: 45 0f af e6  imul   %r14d,%r12d
88c0: 45 01 fd     add    %r15d,%r13d
88c3: 45 01 fc     add    %r15d,%r12d
```

Listing 3.5

Source code

```
#define size 128
#define IntZmmSize 16
typedef int intZmm __attribute__((ext_vector_type(IntZmmSize)));
void vec_fma(int* src1_ptr, int* src2_ptr,
             int* src3_ptr, int* dst_ptr) {
    for (uint32_t i = 0; i < size; i += 1 ){
        intZmm src1Vec = *(intZmm *)(src1_ptr + i);
        intZmm src2Vec = *(intZmm *)(src2_ptr + i);
        intZmm src3Vec = *(intZmm *)(src3_ptr + i);
        *(intZmm *)(dst_ptr + i) = src1Vec * src2Vec + src3Vec;
    }
}
```

Assembly snippet of the hot loop

```
8880: 62 b1 fe 48 6f 04 81  vmovdqu64 (%rcx,%r8,4),%zmm0
8887: 62 b2 7d 48 40 04 80  vpmulld (%rax,%r8,4),%zmm0,%zmm0
888e: 62 b1 7d 48 fe 04 86  vpaddd (%rsi,%r8,4),%zmm0,%zmm0
8895: 62 b1 fe 48 7f 04 87  vmovdqu64 %zmm0,(%rdi,%r8,4)
```

Listing 3.6

3.2 Matrix data structure

The most intuitive data structure of a matrix is a list of lists, where each list represents a row and a list of rows is a matrix. In C++ this can be represented using `std::vector<std::vector<T>>`, where `T` could be `float`, `double`, `int32_t`, etc. The std::vector class provides an intuitive interface for accessing and modifying elements, making it easy to write code with.

One potential drawback of nested `std::vector` is that it requires two indexing operations to access an element. An alternative implementation is to "flatten" a matrix into a single `std::vector`, by simply concatenating one row after another. To access a specific element, an index can be computed manually using the given row and column: `column_count * row + column`. This reduces half of the memory indexing operation at the cost of additional arithmetic. The differences between the two patterns are illustrated by a example provided in Table 3.7.

	Nested	Flat
Type	`std::vector<` `  std::vector<int32_t>>`	`std::vector<int32_t>`

Structure in Memory	`vector of 4 = {`
	`  vector of 4 = {0, 0, 0, 0},`
	`  vector of 4 = {0, 0, 0, 0},`
	`  vector of 4 = {0, 0, 0, 1},`
	`  vector of 4 = {0, 0, 0, 0}`
	`}`
		`vector of 16 = {`
		`  0, 0, 0, 0,`
		`  0, 0, 0, 0,`
		`  0, 0, 0, 1,`
		`  0, 0, 0, 0`
		`}`

Accessing row 2, column 3	Index row first:
	`std::vector<int32_t>[2]`
	then index column:
	`std::vector<int32_t>[2][3]`
		Compute `i = col_count * row + col = 4 * 2 + 3 = 11`,
		then index once:
		`std::vector<int32_t>[11]`

Table 3.7: This is an example of a 4 by 4 matrix, to highlight the differences between the two matrix data structures: nested list and flat list.

Empirically indexing costs more time than integer multiplication and addition, thereby improving performance. Both the toy example and the pivot function perform sequential load-compute-store operations on each row and each column, allowing the index of the next element to be computed by simply adding the step size or column size, further reducing memory overhead. In fact, the pivot function can be optimized to only compute index once (See Section 4.1.4) by placing the pivot row as first row. Benchmark (Figure 3.1) on the toy example confirms that when there are 16 rows, the nested vector matrix is about 8 ns faster than the flat matrix.

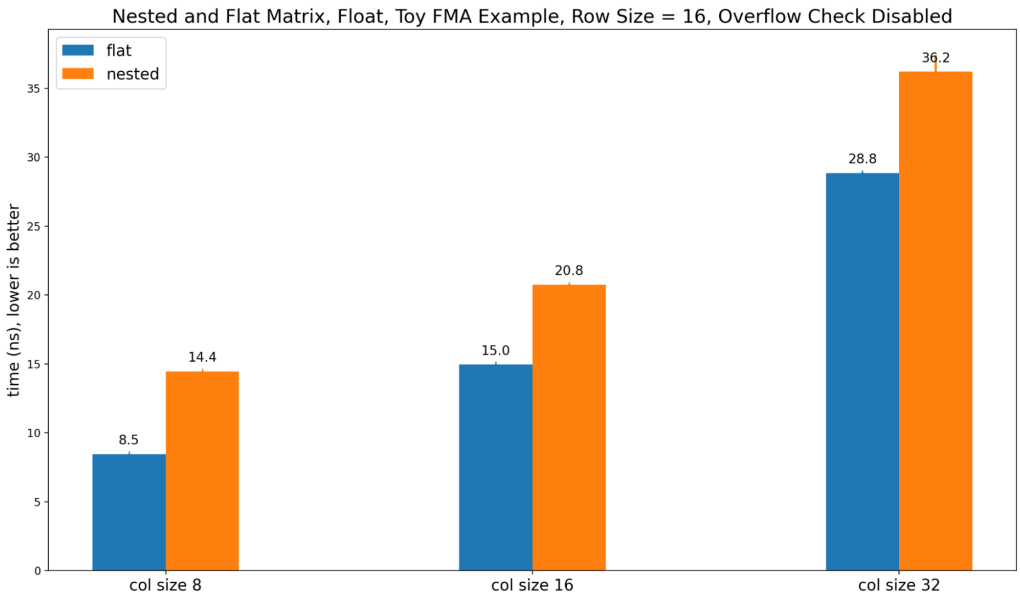

Figure 3.1: TODO: write caption

3.3 Matrix element data type

3.3.1 Width

Since the numbers stored in the matrix are almost always less than 10 bits, using shorter data types can be more advantageous than longer ones because they allow more numbers to be packed into a single vector register (Table 3.3). The number of instructions can be cut by half when data width is reduced to half, and less instruction count always leads to less execution time. Given that the Zen4 micro-architecture provides approximately same amount of execution units for both integers and floating points, it is reasonable to estimate that the execution time is inversely proportional to the bit width of data type. As confirmed in Plot 3.2, `int32_t` and `float` costs nearly same amount of time, while `int32_t` and `double` costs double the amount of time than `int16_t` and `float` respectively.

3.3.2 Overflow checking for integers

A benefit of int16 is that AVX-512 supports saturated add and multiply higher bits instructions (Table 3.3), but equivalent instruction does not exist for int32 and int64. When the result of a conventional add does not match the result of a saturated add, or when multiplying for the higher 16 bits gives non-zero results, it implies an overflow.

3.3.3 Overflow checking for floating points

Integers and floating points have distinct mechanisms for overflow checking

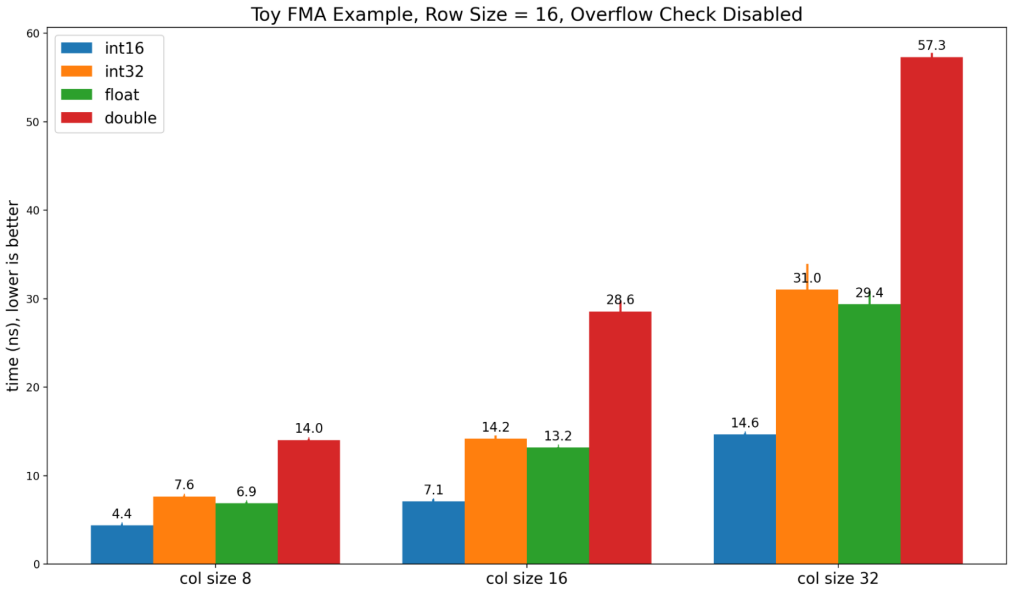

Figure 3.2: TODO: write caption

	f32	**f64**	**i16**	**i32**	**i64**
Execution units	1 512-bit FADD + 1 512-bit FMA		2 512-bit ALU		
Fused-multiply-add	Yes		No		only on lower 52 bits, no overflow exception
Saturated add	N/A		Yes: _mm512_adds_epi32	No	No
Multiply higher bits	N/A		Yes: _mm512_mulhi_epi16	No	No
SIMD Floating-Point Exceptions	Overflow, Underflow, Invalid, Precision, Denormal		No		
512-bit vector size	16	8	32	16	8
Overflow checking and time cost	single time overhead: fetestexcept_mxcsr 4.09 ns feclearexcept_mxcsr 9.42 ns		additional arithmetic about 4x more instructions (saturated add or multiply higher bits)	Require more instructions than int16 due to lack of saturated add and multiply higher bits operations.	

3.3: TODO: write caption

Chapter 4

Pivot

4.1 Optimization

4.1.1 Alignment

4.1.2 Vector size specialization

4.1.3 Reduce floating point status register manipulation

4.1.4 Reduce number of matrix index computation

Chapter 5

Conclusions

5.1 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] Google Benchmark. https://github.com/google/benchmark.

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

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

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

[5] 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.

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

[7] 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.

[8] TODO. Todo, TODO.

[9] TODO. Todo, TODO.

[10] TODO. Todo, TODO.

[11] TODO. Todo, TODO.

[12] TODO. Todo, TODO.

[13] TODO. Todo, TODO.

[14] TODO. Todo, TODO.

[15] TODO. Todo, TODO.

[16] TODO. Todo, TODO.

[17] TODO. Todo, TODO.