Fast `pivot` Function for Presburger Library Through Vectorization and Integer Arithmetic in FPU

Zhou Qi

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

2023

Abstract

This report presents a fast implementation of the core function `pivot` for a math library in MLIR by performing vectorized integer arithmetics in FPU. The hot loop of the `pivot` function performs overflow-checked multiplication and addition on each element of an input matrix of low dimension and mostly small-value items. MLIR's upstream uses element-wise transprecision computing, where the data type of each element starts with `int64_t`, and will be switched to `LargeInteger` in case of overflow. Compilers cannot automatically vectorize this approach, and `int64_t` has a much larger bit width than what is typically needed for most items in the matrix. Additionally, extra arithmetics are required to perform overflow checking for integers, resulting in significant overhead. These issues can be addressed by taking advantage of SIMD, and reducing the bit width for every element. This report also introduces the `int23_t` data type, a 23-bit integer data type created from the 23-bit mantissa of a 32-bit floating point. `int23_t` overflow can be captured as floating point imprecision by a status register, making overflow awareness almost free. On a selected 30-row by 19-column representative input matrix, the runtime is reduced from 550 ns to 26 ns, achieving 20 times speedup.

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

I would like to express my deepest gratitude to my supervisor, Tobias Grosser, for his invaluable guidance and support throughout this project. His innovative ideas and enthusiasm have inspired me, and working under his mentorship was a great opportunity. I have learnt a lot about computer architectures throughout this exciting project, this would not have been possible without his involvement.

I am also grateful to Tobias's Ph.D. students, Arjun Pitchanathan and Sasha Lopoukhine. Arjun's assistance have been instrumental in the progress of this project. I want to extend special thanks to Sasha for generously granting me access to the powerful 7950x workstation.

Finally, I would like to express my heartfelt appreciation to all my friends who have been a constant source of encouragement and motivation. A special mention goes to Emanon42, lyzh, gjz010, and Marisa Kirisame, their camaraderie and support have made this journey enjoyable and memorable.

Table of Contents

1 Introduction **1**

2 Background **4**

2.1 Linear Programming and Simplex Algorithm 4
2.2 Presburger Library 5
2.3 Modern CPU Micro-architecture 6
2.4 Floating Points 8
2.4.1 IEEE 754 8
2.4.2 Fused-multiply-add 8
2.4.3 Representing “`int23_t`” and “`int52_t`” Using Floating Points 9
2.5 Google Benchmark 10
2.6 `llvm-mca` 11

3 Experiments with Toy Example **14**

3.1 Vectorization Method 14
3.1.1 Clang’s Automatic Vectorization 14
3.1.2 Clang’s Vector Data Type 15
3.1.3 Evaluation 16
3.2 Matrix Data Structure 20
3.3 Matrix Element Data Type 21
3.3.1 Width 21
3.3.2 Overflow Checking for Integers 22
3.3.3 Overflow Checking for Floating Points 24
3.3.4 Comparing `int16_t` and `float` 26

4 Implementation and Optimization of `pivot` **28**

4.1 Matrix-wise Transprecision 28
4.2 Double Buffering 28
4.3 Alignment 29
4.4 Reduce Number of Matrix Index Computation 29
4.5 Column Size Specialization 30
4.6 Evaluation 30

5 Conclusion and Future Work **32**

Bibliography **33**

Chapter 1

Introduction

MLIR, Multi-Level Intermediate Representation, is an infrastructure for building reusable and extensible compilers. It aims to reduce fragmentation in domain-specific languages and heterogeneous hardware [7]. Its Presburger library provides polyhedral compilation techniques for dependence analysis and loop optimization [6] and cache modeling [17]. Presburger arithmetics involves determining whether the conjunction of linear arithmetic constraints is satisfiable [4], and can be solved using the simplex method of linear programming, with its core function `pivot` is the main performance bottleneck [14].

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 the runtime is spent on test cases that we can compute using 16-bit integers and matrices with at most 32 columns [15]. These properties can be leveraged to utilize 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 matrix element in a transprecision manner. Each element in the matrix can either be `int64_t` or `LargeInteger` and 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 needed for most of the elements in the matrix.
2. The compiler is not capable of generating vectorized instructions from scalar source code.
3. Overflow is checked through additional arithmetic operations.

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

1. Utilize SIMD: preliminary benchmark (Section 3.1.3) indicates at least 10 times

performance improvement.

2. Use small bit width for every element: reducing the bit width by half doubles the amount of elements packed into a single vector register, and essentially reduces the instruction count by half (See Table 3.4).

3. Fast overflow checking: overflow has to be checked manually for integers, which introduces at least 65% overhead toward total runtime (Section 3.3.2.1). This is because the x86 architecture does not provide status registers to indicate integer overflow. However, there is one for floating points, making floating points overflow detection almost free.

Previously there was an attempt to vectorize `pivot` that utilizes `int16_t` and targets matrices with 32 columns or less [15]. 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 4 times more instruction count (Section 3.3.2.1). Additionally, this approach introduces a new disadvantage. The AVX-512 extension is required for CPUs to support vectorized `int16_t`, and this is very rare among CPUs manufactured in the last decade (Section 2.3).

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 3.3.3). 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. The cost of overflow checking for floating points is the time spent on resetting the status register once at the beginning of a sequence of calls to `pivot` [14], plus reading it in each `pivot` call. Even though reading the register takes 5 ns and resetting it costs 10.5 ns (Figure 3.5), the effective total overhead can be less than 1 ns. The average cost of resetting per `pivot` can be treated as negligible, while the superscalar and out-of-order execution pipeline hides the latency of reading the status register. Moreover, floating points offer better compatibility with old computers than `int16_t`. Vector `float` or `double` code can be executed on CPUs with AVX-2, the predecessor of AVX-512. Almost every x86 CPU from the last decade supports AVX-2.

This report will first analyze the capability of modern CPU micro-architecture, especially `Zen 4`, 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 vector type extension as the vectorization methods, and using a flat list as the matrix data structure. However, it is quite difficult to decide whether `int16_t` or `float` is better,

[1] IEEE 754 specification is introduced in Section 2.4.1

because the former benefits from bigger vector size and less instruction count, while the latter has a minimal overhead on overflow checking.

Then two detached versions of `pivot` function from the Presburger library are built from the most optimal configurations derived from 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. Implementing matrix-wise transprecision computing
2. Double buffering
3. Aligning the matrix to the size of vector registers
4. Reducing the number of matrix index computation
5. Specialization for different row sizes

After applying these optimizations, a benchmark is set up for a selected 30-row and 19-column matrix. The achievements are:

1. The vectorized `float` implementation of `pivot` significantly outperforms the upstream. Specifically, it takes only 26 ns, while the runtime of the upstream implementation is 550 ns.

2. The `float` implementation offers substantial compatibility advantage over `int16_t` for the vast amount of non-AVX-512 platforms. Even though the `int16_t` version is 6 ns faster than the `float` version, the significant improvement from the upstream together with AVX-2's compatibility renders the 6 ns gap trivial.

Chapter 2

Background

2.1 Linear Programming and Simplex Algorithm

Linear programming is a mathematical optimization technique used to model and find the best possible solution to a problem, using a set of constraints and the objective function to maximize or minimize. Its canonical form consists of a objective function: $Z = c_1x_1 + c_2x_2 + ... + c_nx_n$, subjecting to the constraints:
$x_1 ... x_n \geq 0$
$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1$
$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2$
...
$a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m$,
where $x_1 ... x_n$ are the variables, $c_1 ... c_n$ are the coefficients of the objective function, and non-negative $a_{11}, a_{12} ... a_{21} ... a_{m1} ... a_{mn}$ together with $b_1 ... b_m$ encodes the constraints of the problem in a matrix [14].

The simplex algorithm is an iterative approach to find $x_1 ... x_n$ that maximizes the objective function while satisfying constraints at the same time. The matrix goes through a sequence of transformations, until the solution appears or it is found that the solution is not feasible. The transformations are called "pivot", and it solves the linear equation at the pivot row for the variable at the pivot column:

Pivot item	α	$\rightarrow$	$\frac{1}{\alpha}$
Rest of the pivot row	β	$\rightarrow$	$-\frac{\beta}{\alpha}$
Rest of the pivot column	γ	$\rightarrow$	$\frac{\gamma}{\alpha}$
Other entries	δ	$\rightarrow$	$\delta - \frac{\beta\gamma}{\alpha}$

The pivot transformation involves division and thereby produces rational numbers. Rationals of base 10 cannot be expressed as binary floating points precisely due to inaccuracy caused by potential rounding. In addition, divisions are expensive operations compared to additions or multiplications. This issue can be addressed through the denormalization of rows, by multiplying each row with their common denominator [14]:

$$\begin{array}{lccc} \text{Pivot item} & \alpha & = & \frac{\alpha_n}{d_p} \\ \text{Rest of the pivot row} & \beta & = & \frac{\beta_n}{d_p} \\ \text{Rest of the pivot column} & \gamma & = & \frac{\gamma_n}{d_i} \\ \text{Other entries} & \delta & = & \frac{\delta_n}{d_i} \end{array}$$

where d_p is the denominator of the pivot row, d_i is the denominator of the i^{th} row. After demoralization α_n, β_n, γ_n, and δ_n, becomes α, β, γ, and δ.

Substituting into the processing formula, the pivot row becomes:

$$\begin{array}{lccc} \text{Pivot item} & \frac{\alpha_n}{d_p} & \rightarrow & \frac{d_p}{\alpha_n} \\ \text{Rest of the pivot row} & \frac{\beta_n}{d_p} & \rightarrow & -\frac{\beta_n}{\alpha_n} \end{array}$$

This transformation effectively becomes swapping α_n with d_p and negating every non-pivot-column item.

Likewise, the non-pivot rows are transformed as the following:

$$\begin{array}{lccc} \text{Rest of the pivot column} & \frac{\gamma_n}{d_i} & \rightarrow & \frac{\gamma_n d_p}{a_n d_i} \\ \text{Other entries} & \frac{\delta_n}{d_i} & \rightarrow & \frac{\delta_n a_n - \beta_n \gamma_n}{a_n d_i} \end{array}$$

and can be implemented in these procedures:

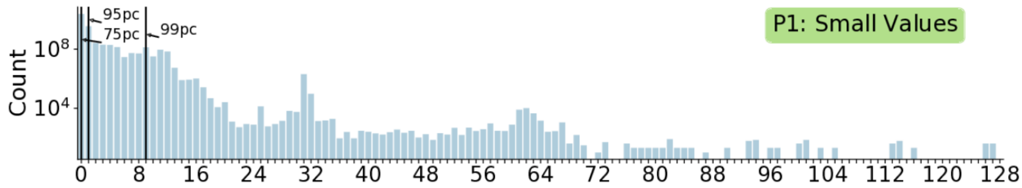
1. Update row denominator: $d_i' = d_i a_n$,
2. Multiply non-pivot-column items by a_n,
3. Subtract $\beta_n \gamma_n$ from every non-pivot row.

2.2 Presburger Library

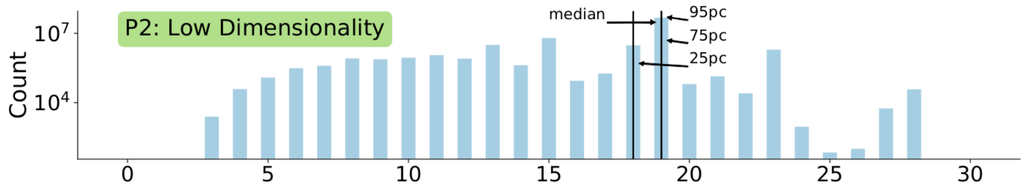
The Fast Presburger Library (FPL) paper collected 465,460 representative linear programming problems encountered during cache analytical modeling, polyhedral loop optimization, and accelerator code generation. It is found that most of the constraint matrices are low in dimensionality and small in the value of each element [14]. Specifically, more than 99% of the entries from matrices require less than 10 bits, and 95% of them are less than 20 columns (Figure 2.1). Thus, most 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, large coefficients can be 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 no specific maximum column count.

The FPL paper presents a 3-way transprecision implementation for the Presburger library's simplex solver using the algorithm described in Section 2.1. It starts from row-wise vectorized `int16_t`, and will switch to element-wise scalar `int64_t` or element-wise scalar `LargeInteger` in case of overflow, as illustrated in Figure 2.2. 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:



(a) Value Size - Bitwidth of coefficients in binary representation



(b) Dimensionality - Number of matrix columns

Figure 2.1: Linear programming problems for program analysis exhibits unique characteristics of small value size and low dimensionality [14].

1. `int16_t` vectors require AVX-512 ISA extension, but hardware support is rare (Section 2.3).
2. Despite the `int16_t` version being fast, overflow checking overhead is 65% [15]. Using floating points could significantly reduce this overhead and potentially be faster (Section 2.4).

2.3 Modern CPU Micro-architecture

A recent trend in the development of the x86-64 architecture is to include AVX-512 instruction set architecture (ISA) extension. AVX-512 succeeds AVX-2, 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 [29], as it did not bring practical performance improvements. The primary reason was that it consumed much more power than usual, causing severe overheating. A classic example is the micro-architecture Skylake from Intel and its AVX-512 enabled counterpart Skylake-X. Skylake has 2 256 bits FMA AVX-2 execution units[1]. For Skylake-X 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 [9]. 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 the AVX-offset mode and reduces its clock frequency [25]. This solution only works in

[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 2.4.2.

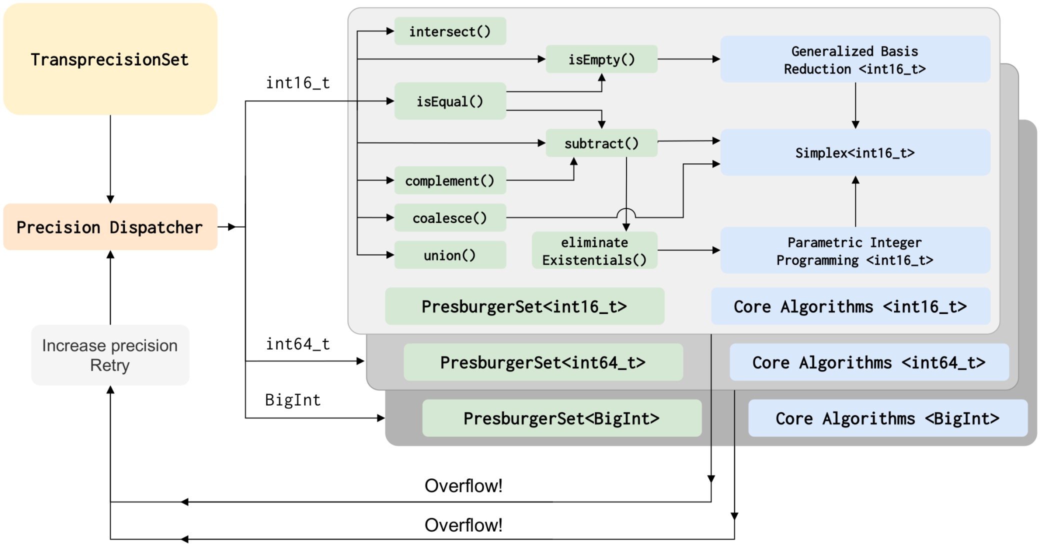


Figure 2.2: The architecture of FPL. The focus of this report is the “Simplex” method of the “Core Algorithms”, since it is the main performance bottleneck. [15]

theoretical benchmarks where AVX-512 instructions are present in large bulk, but in practice, it is more common to have a mix of control flow, scalar, SSE and AVX-512 instructions. The clock frequency of executing those non-AVX-512 instructions is decreased together with AVX-512 instructions, causing many workloads could run faster with disabled AVX-512 and higher clock frequency [23].

AMD recently decided to add support for AVX-512 in their latest micro-architecture Zen 4. It has slightly less computing power than Intel but is much more efficient. Zen 4 can be considered as the modernized version of Zen 3 or Zen 2, where Zen 2 and Zen 3 support AVX-2 by providing 2 FADD units[2] and 2 FMA units of 256-bit width [3]. Zen 4 “double-pumps” these existing circuits to create a single 512-bit FADD and a single 512-bit FMA, without introducing any new arithmetic units [23]. Zen 2 and Zen 3 are reputable for their high performance per watt [18], and Zen 4 would be better with its more advanced lithography [23].

Additionally, rebuilding existing software to target AVX-512 may improve performance. One benefit of AVX-512 is that it reduces front-end pressure. In the Zen 4 micro-architecture case, 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, and this is much more likely to be sustained by the frontend [23].

[2]Floating-point add units (FADD) can execute floating point addition instructions only. They are less capable compared to FMA.

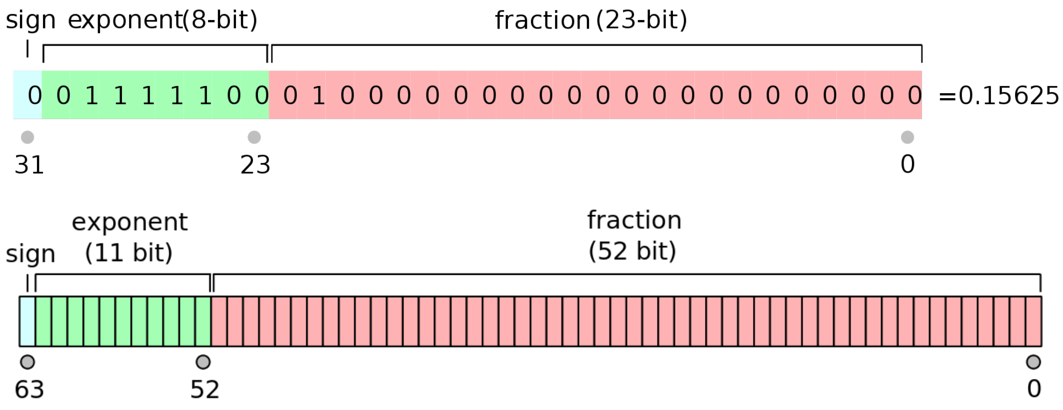


Figure 2.3: IEEE 754 specification for single (32 bits) and double (64 bits) precision floating point [26]. In some literature "mantissa" is referred as "fraction".

2.4 Floating Points

2.4.1 IEEE 754

IEEE 754 is the standard for representing and manipulating floating-point numbers in modern x86 computers. The standard defines several 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, the specification defines how many bits are used to represent the sign, the exponent, and the mantissa.

As Figure 2.3 shows, the sign bit is a single bit that indicates whether the floating point number is positive or negative. There are 8 bits and 11 bits for exponent in float and double, respectively, representing the order of magnitude. The remaining 23 bits in float and 52 bits in double are mantissae, the fractional part of the number is stored here. 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.

Figure 2.3 provides an example of `float` by presenting `0.15625` in binary form:

```
sign     = 0b0        -> 0
exponent = 0b01111100 -> 0b01111100 - 127 = 124 - 127 = -3
mantissa = 0b01       -> 0b1.01 = 1.25
```

Substituting the sign, the exponent, and the mantissa into the formula, we get:
`-1^0 * 2^(-3) * 1.25 = 0.15625.`

2.4.2 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 [28].

FMA saves cycles and reduces the accumulation of rounding errors, while at the same time not adding significant complexity to the circuit. An FMA execution unit is capable of doing 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 feature is useful in many numerical computations involving simultaneous multiplication and addition operations, such as dot products and matrix multiplications. Since the `pivot` function performs multiplications and additions 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.

2.4.3 Representing "`int23_t`" and "`int52_t`" Using Floating Points

There is a common stereotype that floating-point numbers are unreliable and likely to be imprecise, and are often illustrated in popular memes as shown in Figure 2.4 [30]. However, when storing integer values inside floating points, floating points can be pretty reliable. Historically floating point processing units in GPUs have been utilized for fast integer arithmetic, for example, modular exponentiation [13] and RSA algorithms [20], because often the architecture of GPU prioritizes the performance of floating points rather than integers.

Given that the mantissa part of a float consists of 23 bits, inexactness never occurs when representing integers less than 23 bits (ignoring the sign bit). Furthermore, in case of an integer overflow, floating point imprecision almost always occurs, and setting a corresponding status register. The same concept applies to double data types, which have a mantissa consisting of 52 bits.

This mechanism is reliable, as floating-point inexactness always implies integer -inexactness. For an integer value with a bit width bigger than the mantissa size, floating point rounding is triggered to fit the most significant bits of the integer in the mantissa, then adjust the order of magnitude in the exponent accordingly. The lower bits of the mantissa are truncated, therefore causing imprecision.

In some rare cases, integers longer than the size of mantissa can be represented in floating points precisely. An example of such numbers is a very large power of 2, like `2^30 = 0x40000000`. Its binary representation in `float` is:

```
Sign     = 0b0        -> 0
Exponent = 0b10011011 -> 0b10011011 - 127 = 155 - 127 = 28
Mantissa = 0b0        -> 0b1.0 = 1
```

Despite being greater than the size of the mantissa, they are normalized rather than being rounded and therefore does not break the mechanism of representing integers in floating points.

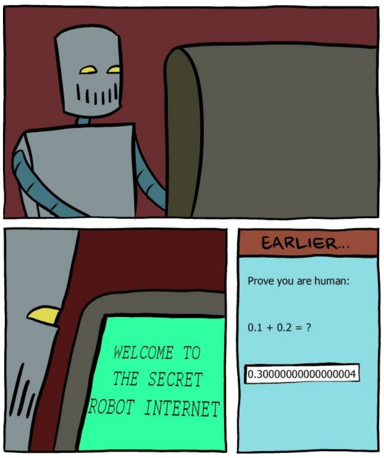


Figure 2.4: A floating point meme: `0.1 + 0.2 = 0.30000000000000004` [30].

2.5 Google Benchmark

Google benchmark is a library to measure the performance of a code snippet. It provides unit-test-like interfaces to set up benchmarks around a code snippet [10]. 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 the CPU clock frequency is not consistent. The clock frequency is dynamically determined by the governor algorithm according to the operating system. 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`. In contrast, the `ondemand` governor will push the CPU to the highest frequency on demand and then gradually reduce the frequency as the idle time increases [19].

However, the clock frequency 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 [11]. On the other hand, CPUs have self-protecting thermal throttling mechanisms that reduce their clock frequency and voltage when it is too hot.

The benchmark mentioned in this report were performed on an 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 4.5 GHz, or any desired and feasible value,
4. Make sure heat dissipation is working properly.

2.6 `llvm-mca`

`llvm-mca`, LLVM Machine Code Analyzer, is a tool to analyze the performance of executing some instructions on a specific CPU micro-architecture, according to scheduling information provided by LLVM [12].

By supplying `llvm-mca` with a piece of assembly code and the target micro-architecture codename, `llvm-mca` reports various metrics to indicate how fast the given instructions will execute on the specified micro-architecture. It first summarizes the instruction per clock (IPC) and throughput of the entire instruction block, then gives detailed information about each instruction, including the number of uOps, latency, throughput, potential load, store, and side effects. `llvm-mca` also reports resource pressure regarding arithmetic units and memory load or store units. When the optional `-timeline` flag is prompted, `llvm-mca` illustrates a timeline view of the analyzed code, showing how instructions progress through the pipeline stages of the target processor. The timeline helps understand the capability of complicated out-of-order superscalar architecture.

An example is provided below. The analysis from `llvm-mca` indicates that a `znver2` (Zen 2) CPU can repeatedly execute a combination of `vmovaps`[3] and `vfmadd213ps`[4] instructions every 1.3 cycles, or equivalently 2.74 instructions per cycle. The output from `llvm-mca` is slightly modified and truncated to fit inside the page.

```
$ llvm-mca-15 -timeline -mcpu=znver2 ./x.s
Iterations:        100
Instructions:      400
Total Cycles:      146
Total uOps:        400

Dispatch Width:    4
uOps Per Cycle:    2.74
IPC:               2.74
Block RThroughput: 1.3

Instruction Info:
[1]: #uOps
[2]: Latency
[3]: RThroughput
[4]: MayLoad
[5]: MayStore
[6]: HasSideEffects (U)

[1] [2] [3]  [4] [5] [6] Instructions:
 1   8  0.33  *           vmovaps  (%rdx,%rsi,4), %ymm0
 1   8  0.33  *           vmovaps  (%rcx,%rax,4), %ymm1
 1   12 0.50  *           vfmadd213ps  (%r8,%rdi,4), %ymm0, %ymm1
 1   1  0.33      *       vmovaps  %ymm1, (%r9,%rax,4)

Resources:
[0]   - Zn2AGU0
[1]   - Zn2AGU1
[2]   - Zn2AGU2
...
[8]   - Zn2FPU0
[9]   - Zn2FPU1
[10]  - Zn2FPU2
[11]  - Zn2FPU3
[12]  - Zn2Multiplier

Resource pressure per iteration:
```

[3] `vmovaps` can be either vector float load or store instruction, depending on how operands are structured [5].

[4] `vfmadd213ps` is the instruction for fused-multiply-add [5].

```
[0]  [1]  [2]  [3] [4] [5] [6] [7] [8]  [9] [10] [11] [12]
1.33 1.33 1.34  -   -   -   -   -  0.50  -   -   0.50  -

Resource pressure by instruction:
[0]  [1]  [2]  ... [11] Instructions:
0.01 0.38 0.61 ...  -   vmovaps  (%rdx,%rsi,4), %ymm0
0.23 0.68 0.09 ...  -   vmovaps  (%rcx,%rax,4), %ymm1
0.27 0.24 0.49 ... 0.50 vfmadd213ps  (%r8,%rdi,4), %ymm0, %ymm1
0.82 0.03 0.15 ...  -   vmovaps  %ymm1, (%r9,%rax,4)

Timeline view:
                0123456789
Index 0123456789

[0,0] DeeeeeeeeER    .    vmovaps  (%rdx,%rsi,4), %ymm0
[0,1] DeeeeeeeeER    .    vmovaps  (%rcx,%rax,4), %ymm1
[0,2] D=eeeeeeeeeeeeER    vfmadd213ps  (%r8,%rdi,4), %ymm0, %ymm1
[0,3] D=============eER   vmovaps  %ymm1, (%r9,%rax,4)
[1,0] .DeeeeeeeeE-----R   vmovaps  (%rdx,%rsi,4), %ymm0
[1,1] .DeeeeeeeeE-----R   vmovaps  (%rcx,%rax,4), %ymm1
[1,2] .D=eeeeeeeeeeeeER   vfmadd213ps  (%r8,%rdi,4), %ymm0, %ymm1
[1,3] .D=============eER  vmovaps  %ymm1, (%r9,%rax,4)
[2,0] . DeeeeeeeeE-----R  vmovaps  (%rdx,%rsi,4), %ymm0
[2,1] . DeeeeeeeeE-----R  vmovaps  (%rcx,%rax,4), %ymm1
...
```

However, when evaluating the identical assembly code on a more advanced micro-architecture `znver3` (Zen 3), `llvm-mca` reveals a reduction in IPC and throughput. This appears to be contradictory to both theoretical expectations and actual benchmarks. After submitting an issue [2], llvm maintainers explained that llvm's scheduling information are hand-crafted using `llvm-exegesis`, a micro-benchmark tool. The issue was subsequently resolved after rerunning `llvm-exegesis` and confirming that `znver3` indeed has higher throughput than expected.

This issue suggests that `llvm-mca` is not a reliable tool for analyzing machine code but rather an evaluator for Clang's behavior during instruction selection. Therefore, this report chooses Google Benchmark as the performance measuring tool.

Chapter 3

Experiments with Toy Example

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

1. Vectorization method
   (a) Clang's automatic vectorization from scalar source code
   (b) Writing source code using Clang's vector type extension
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 can generate 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.

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

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, 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, 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: The vectorized binary and scalar binary is derived by compiling with flags `-O3 -march=native` and `-O3 -march=native -mno-avx -mno-sse` respectively, on a Zen 4 computer with `clang-15`.

`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 will be equally optimized, with the only difference being whether vector instructions are generated or not. In this case, scalar instructions `fadds`, `fstps`, and `flds` are selected.

3.1.2 Clang's Vector Data Type

Another approach is to write source code with vectorization in mind in the first place. Clang provides an extension that allows programmers to declare a new type representing 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 data type, 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 perform 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
typedef float floatZmm __attribute__((ext_vector_type(16)));
void vec_add(float* src1_ptr, float* src2_ptr, float* dst_ptr) {
    for (uint32_t i = 0; i < size; i += VecSize){
        floatZmm src1Vec = *(floatZmm *)(src1_ptr + i);
        floatZmm src2Vec = *(floatZmm *)(src2_ptr + i);
        *(floatZmm *)(dst_ptr + i) = src1Vec + src2Vec;
    }
}
```

Listing 3.2: Compared to the C++ source code in Listing 3.1, coding with vector types is slightly more complicated.

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 automatic vectorization feature in Clang can be unpredictable and may lead to undesired behaviors. It operates as a black box and may take much 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, a slightly more complicated version of the previous `vec_add` example, where there are 3 input matrices and the element-wise operation is changed from addition to FMA. The disassembly reveals that Clang decides to use the FMA vector instructions of 128-bit width. However, when vector size is constrained to a bigger width by defining a vector type, a more optimal binary can be generated. Benchmark (Figure 3.1) shows that the vector type version is 6 times and 11 times faster than the automatic vectorization and vectorization disabled versions, respectively.

Scalar source code to be automatically vectorized by Clang

```
void vec_fma(matrix<float> & mat_src1, matrix<float> & mat_src2,
             matrix<float> & mat_src3, matrix<float> & mat_dst) {
    for (int i = 0; i < row; i += 1) {
        for (int j = 0; j < col; j += 1) {
            float src1 = mat_src1.get(i,j);
            float src2 = mat_src2.get(i,j);
            float src3 = mat_src3.get(i,j);
            mat_dst.set(i,j,  (src1 * src2) + src3);
        }
    }
}
```

Assembly snippet of the hot loop

```
vmovss (%rdx,%rsi,4),%xmm0
vmovss (%rcx,%rax,4),%xmm1
vfmadd213ss (%r8,%rdi,4),%xmm0,%xmm1
vmovss %xmm1,(%r9,%rax,4)
```

Source code written with Clang's vector type

```
typedef float floatZmm __attribute__((ext_vector_type(16)));
void vec_fma(matrix<float> & mat_src1, matrix<float> & mat_src2,
             matrix<float> & mat_src3, matrix<float> & mat_dst) {
    float * src1_ptr = (float *) mat_src1.getItemPointer(0,0);
    float * src2_ptr = (float *) mat_src2.getItemPointer(0,0);
    float * src3_ptr = (float *) mat_src3.getItemPointer(0,0);
    float * dst_ptr  = (float *) mat_dst.getItemPointer(0,0);
    for (uint32_t i = 0; i < row * col; i += 16){
        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

```
vmovups (%rax,%r8,4),%zmm0
vmovups (%rcx,%r8,4),%zmm1
vfmadd213ps (%rsi,%r8,4),%zmm0,%zmm1
vmovups %zmm1,(%rdi,%r8,4)
```

Listing 3.3: The toy example performs vectorized fused-multiply-add operation on every item from 3 input matrices and saves the result to the output matrix. The internal data structure of `matrix` is `std::vector`, and its member function `getItemPointer(r,c)` returns the pointer to the element at row `r` and column `c`. The key distinction between the two vectorization approaches is the register types. `XMM` are 128-bit registers, while `ZMM` are 512-bit long.

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`) entirely (Listing 3.4). Their vectorized equivalency `vpaddd` and `vpmulld` is 15 times more performant (Figure 3.1).

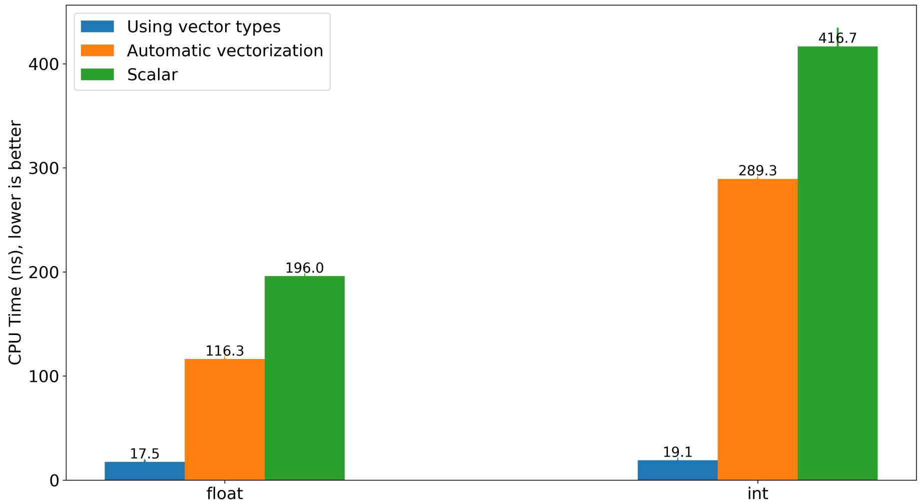


Figure 3.1: A benchmark for the element-wise multiply-add toy example on 16 by 16 matrices with different vectorization techniques. The toy example written with vector types is 5 to 10 times faster than their automatically vectorized or scalar counterpart.

Scalar source code to be automatically vectorized by Clang

```
void vec_fma(matrix<int> & mat_src1, matrix<int> & mat_src2,
             matrix<int> & mat_src3, matrix<int> & mat_dst) {
    for (int i = 0; i < row; i += 1) {
        for (int j = 0; j < col; j += 1) {
            int src1 = mat_src1.get(i,j);
            int src2 = mat_src2.get(i,j);
            int src3 = mat_src3.get(i,j);
            mat_dst.set(i,j,  (src1 * src2) + src3);
        }
    }
}
```

Assembly snippet of the hot loop

```
mov    0x1c(%rcx),%r13d
mov    0x1c(%rdx),%r12d
imul   %r14d,%r13d
imul   %r14d,%r12d
add    %r15d,%r13d
add    %r15d,%r12d
```

Vectorized source code using Clang's vector type extension

```
typedef int intZmm __attribute__((ext_vector_type(16)));
void vec_fma(matrix<int> & mat_src1, matrix<int> & mat_src2,
             matrix<int> & mat_src3, matrix<int> & mat_dst) {
    int * src1_ptr = (int *) mat_src1.getItemPointer(0,0);
    int * src2_ptr = (int *) mat_src2.getItemPointer(0,0);
    int * src3_ptr = (int *) mat_src3.getItemPointer(0,0);
    int * dst_ptr  = (int *) mat_dst.getItemPointer(0,0);
    for (uint32_t i = 0; i < row * col; i += 16){
        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

```
vmovdqu64 (%rcx,%r8,4),%zmm0
vpmulld (%rax,%r8,4),%zmm0,%zmm0
vpaddd (%rsi,%r8,4),%zmm0,%zmm0
vmovdqu64 %zmm0,(%rdi,%r8,4)
```

Listing 3.4: Comparing to the source code from Listing 3.3, the only change is replacing `float` with `int`. Clang fails to vectorize the scalar version, but vectorization is still successful with vector types.

3.2 Matrix Data Structure

The most intuitive data structure of a matrix is a nested 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 build a matrix data structure on top.

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`. Compared to the list-of-lists approach, this reduces half of the memory indexing operation at the cost of extra arithmetics. The differences between the two patterns are illustrated by an example provided in Table 3.5.

	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 3.5: This is an example of a 4 by 4 matrix structured using nested and flat lists, to highlight their differences.

Empirically indexing costs more time than integer multiplication and addition, thus 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 4.4) by placing the pivot row as the first row. Benchmark (Figure 3.2) on the toy example confirms that when there are 16 rows, the nested vector matrix is about 8 ns faster than the flat matrix.

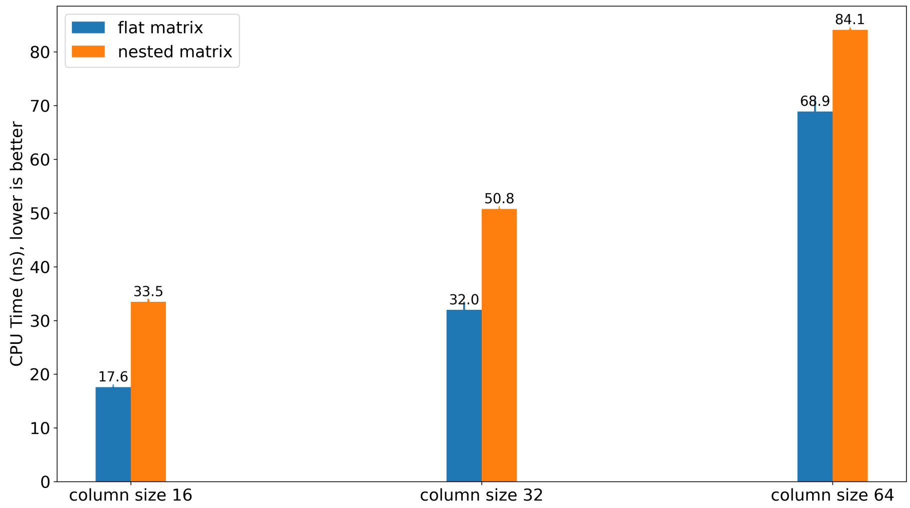


Figure 3.2: On a 16-row toy example doing `float` FMA, selecting the flat list as the data structure of the matrix is consistently faster than the nested list implementation.

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 (Figure 3.4). The number of instructions can be cut by half when the data width is reduced to half, and less instruction count always leads to less execution time. Given that the Zen 4 micro-architecture provides approximately the 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 the data type. As confirmed in Figure 3.3, when overflow is ignored, `int32_t` and `float` cost nearly the same amount of time, while `int32_t` and `double` cost double the amount of time than `int16_t` and `float` respectively.

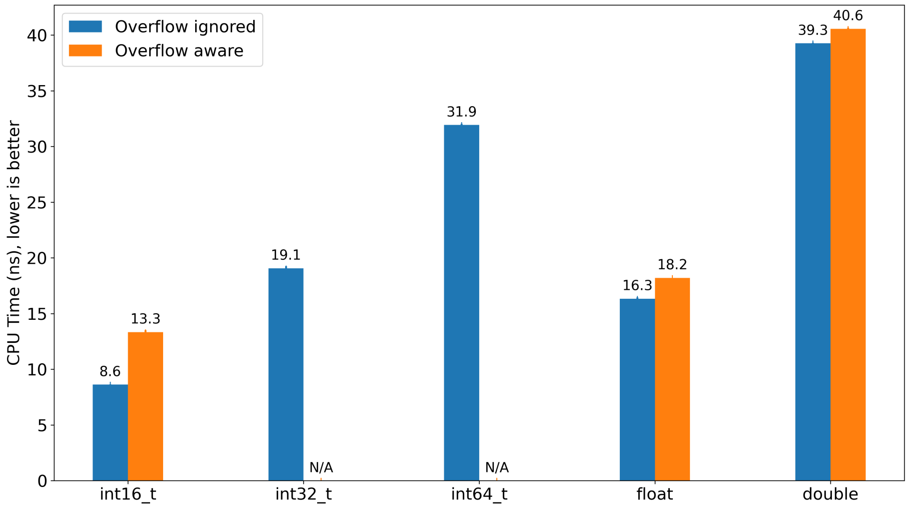


Figure 3.3: A benchmark for toy example on a 16 by 16 matrix of `int16_t`, `int32_t`, `int64_t`, `float`, and `double`, with overflow checking turned on or off. The runtime information for overflow checked `int32_t` and `int64_t` is not available, because it is difficult to implement vectorized overflow checker. It is discovered that (1) runtime is reduced by 50% as the bit width of the data type is cut by half, (2) overflow checking for integers is much more expensive than floating points.

	float	**double**	**int16_t**	**int32_t**	**int64_t**
512-bit units	1 512-bit FADD + 1 512-bit FMA		2 512-bit ALU		
256-bit units	2 256-bit FADD + 2 256-bit FMA		4 256-bit ALU		
Fused-multiply-add	Yes		No		only on lower 52 bits, no overflow exception
Saturated add	N/A		Yes	No	No
Multiply higher bits	N/A		Yes	No	No
SIMD Floating-Point Exceptions	Overflow, Underflow, Invalid, Precision, Denormal		No		
512-bit vector size	16	8	32	16	8
256-bit vector size	8	4	16	8	4
Overflow checking and time cost	single time overhead: read status register 4.5 ns clear status register 9.4 ns		additional arithmetic, about 4x more instructions	Must fallback to scalar code due to lack of saturated add and multiply higher bits.	

Figure 3.4: A summary of features and resources provided by the Zen 4 micro-architecture for different data types [23] [3].

3.3.2 Overflow Checking for Integers

The x86-64 micro-architecture provides the `seto` instruction to set some byte to `1` if overflow occurred as a result of integer arithmetic. However, `seto` only works for scalar operations. There is no instruction or status register to indicate whether a previous vector add or multiply instruction produced overflown results. Therefore, overflow has to be checked manually by some additional vector instructions. This would slow down the computation to some extent. Alternatively, arithmetics have to be carried out on each element individually in a scalar manner, resulting in even worse performance.

One advantage of `int16_t` is that it can be used with AVX-512's saturated add and multiply higher bits vector instructions (Figure 3.4), making it possible and convenient to write vectorized and overflow-aware code. However, Zen 4's implementation of AVX-512 extension does not provide equivalent instruction for `int32_t` or `int64_t` and therefore must be processed as scalar values.

3.3.2.1 Implementation of Vectorized `int16_t` Overflow Checking

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

Inspecting these approaches from an instruction-level perspective (Table 3.6), when overflow is ignored, both add and multiply takes 1 instruction, `vpaddw`[1] and `vpmullw`[2]. To obtain and process overflow-related information, an additional computation instruction `vpaddsw`[3] or `vpmulhw`[4] is required, followed with 2 or 3 comparison, shuffling and branch instructions: `vpsraw`[5], `vpcmpneqw`[6] and `kord`[7]. By enabling overflow checking, it brings 4 to 5 times more instruction count and 65% more runtime [15].

[1] Vector add for `int16_t`
[2] Vector multiply lower half bits for `int16_t`
[3] Vector saturated add for `int16_t`
[4] Vector multiple higher half bits for `int16_t`
[5] Shift packed data right arithmetic
[6] Compare packed data for equal
[7] Bitwise logical OR masks

	Addition	Multiplication
Overflow ignored	`vpaddw %zmm4,%zmm2,%zmm3`	`vpmullw %zmm1,%zmm3,%zmm2`

Overflow aware	`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 3.6: This table highlights the difference in instruction count when overflow checking is enabled or disabled for vectorized `int16_t`. Pink instructions are essential components as they compute the anticipated arithmetic results, while overflow information are provided by yellow and cyan instructions.

3.3.2.2 Implementation of Scalar `int32_t` and `int64_t` Overflow Checking

Clang's language extension provides functions to perform overflow-checked integer arithmetics:
`bool __builtin_add_overflow (type1 x, type2 y, type3 *sum);`
`bool __builtin_mul_overflow (type1 x, type2 y, type3 *prod);`
These functions take three arguments: `x` and `y` are the two input operands, and `sum` or `prod` is a pointer to the variable that will hold the result of the addition or multiplication. The return value of these functions is a boolean that indicates whether an overflow occurred during the operation. However, they do not accept vectors as input, and a loop around these functions cannot be compiled into vector instructions either.

3.3.3 Overflow Checking for Floating Points

To detect floating point overflow or imprecision, one approach is to enable floating point imprecision as a trap, then upon overflow, the interrupt `SIGFPE` is raised, and the PC (program counter) will be redirected to its handler. The method can be programmed by using useful functions from the `fenv` library as follow [8]:

```
void signal_handler(int signal) {
    // handle fpe
}

void function() {
    std::signal(SIGFPE, signal_handler);
    std::feclearexcept (FE_ALL_EXCEPT);
    feenableexcept (FE_INEXACT | FE_INVALID);
    // do something
    fedisableexcept (FE_INEXACT | FE_INVALID);
}
```

In the `function()` block, first, the `std::signal()` function is called to register the signal handler function for the `SIGFPE` interrupt. Next, the `feclearexcept` function is called to clear any previously set exception flags in the status register. Then, `FE_INEXACT` and `FE_INVALID` are passed to the `feenableexcept()` function to enable inexactness and invalid exceptions. Once the floating-point exceptions are enabled, computations can be performed. If some operation produces an inexact or invalid floating point number, `SIGFPE` is raised, and a call to `signal_handler` is triggered. After the computation is completed, the `fedisableexcept()` function is called to disable the previously enabled exceptions.

But it is difficult to recover back from `SIGFPE`. By design, the usage of `SIGFPE` is to do some cleanup in the handler function, then exit the program gracefully. If the handler does not exit the program after returning from the handler, the PC always points back to the instruction that caused `SIGFPE` and triggers `SIGFPE` again! However, the goal is to discard the current progress and continue the program using the LargeInteger algorithm. Although there could be potential workarounds, such as modifying the call stack and changing the return address, implementing these solutions can be challenging and introduce significant complexity to the codebase.

Alternatively, we may read status registers and check if the imprecision bit is set:

```
bool function(matrix & tableau) {
    std::feclearexcept (FE_ALL_EXCEPT);
    if (fetestexcept(FE_INEXACT | FE_INVALID)) {
        // false for overflow, will be handle by its caller
        return false;
    } return true; // true for safe
}
```

Instead of registering floating point imprecision as interrupts, it clears the floating point status register with `feclearexcept`. It reads the status register afterwards using the `fetestexcept` function to check whether imprecision has ever occurred in previous computations. It then returns a boolean value to notify its caller whether or not the test for floating-point exceptions is positive, `true` for safe and `false` for overflown. In case of returning `false`, the caller will retry computation using `LargeInteger` accordingly.

3.3.3.1 Evaluation

In x86_64 there are two status registers for floating points, the legacy x87 status register for traditional scalar floating point operation and the modern `mxcsr` register for SSE or AVX instructions. The source code of the library function `fetestexcept` indeed manipulates both registers [8]:

```
int fetestexcept(int excepts) {
    unsigned short status;
    unsigned int mxcsr;

    excepts &= FE_ALL_EXCEPT;

    /* Store the current x87 status register */
    __asm__ volatile ("fnstsw %0" : "=am" (status));
```

```
    /* Store the MXCSR register state */
    __asm__ volatile ("stmxcsr %0" : "=m" (mxcsr));

    return ((status | mxcsr) & excepts);
}
```

In the vectorized floating point scenario, the instruction for the x87 status register is unnecessary, and only the `mxcsr` register should be concerned. The hot loop is completely vectorized, and Clang dispatches 128-bit SSE instructions for occasional scalar operations. The reason is that the SSE execution units can handle both single and double precision floating point arithmetic natively. In contrast, the legacy x87 floating-point instructions operate on an 80-bit internal format, requiring additional conversions and delay. Using SSE instructions reduce register pressure as well. The SSE units have access to 16 `XMM` registers, but for x87 units, only 8 floating point registers are available [16].

As illustrated in Figure 3.5, the benchmark evaluates the performance of `fenv` functions alongside their respective revised versions. The modifications involve restricting operations solely to manipulating either the x87 status register or the `mxcsr` register. It indicates that it is significantly faster if we remove x87 status register related operations.

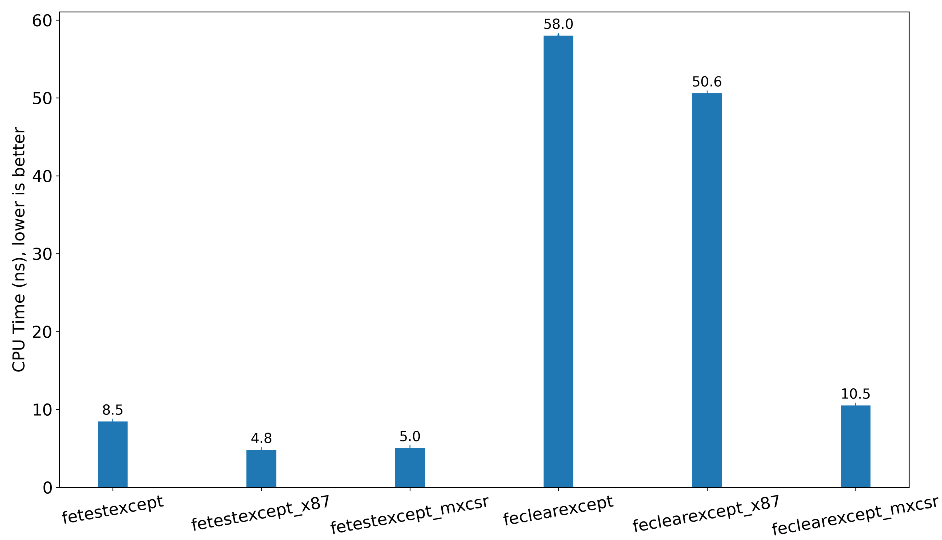


Figure 3.5: A benchmark for the floating point status register reading and resetting functions from `cfenv` library and their modified versions that only operate on either the x87 or the `mxcsr` status register. Excluding x87 related operations makes both `fetestexcept` `feclearexcept` faster.

3.3.4 Comparing `int16_t` and `float`

Section 3.3.1, 3.3.2, and 3.3.3 have concluded that `int16_t` is superior to any other integer data type and `float` is better than `double`.

Benchmark (Figure 3.3) on the toy example reveals that `int16_t`'s spends a considerably higher percentage of runtime on overflow checking compared with the floating point data types. This is consistent with the reasoning from previous chapters, where overhead for `float` is a one-time expense, but for `int16_t` it is always a portion of the total runtime.

Note that even though `int16_t` is faster than `float` in this benchmark, it is not sound to conclude that the `pivot` function implemented in `int16_t` will be faster than `float`. The toy example is different from the `pivot` function in many aspects, for example, the number of memory load operations. In the actual `pivot` function, `float` may potentially outperform `int16_t`.

Chapter 4

Implementation and Optimization of `pivot`

4.1 Matrix-wise Transprecision

Transprecision computing can be implemented at different levels of scale, such as element-wise, row-wise, and matrix-wise. The vectorized versions of `pivot` are implemented using the matrix-wise transprecision style.

The element-wise method is unsuitable in this scenario, as it defeats the purpose of vectorization. The row-wise method is not chosen either, due to that overflow is not likely to occur, and the matrix is small. Reading the `mxcsr` can be considered as an expensive operation compared to the time spent on pivoting through the entire matrix. Avoid wasting arithmetic instructions after overflowing at the cost of more `mxcsr` reads is not a cost-effective trade-off.

Even though the `pivot` function using `int16_t` was implemented using the row-transprecision approach in previous works [15], it is modified to match with the matrix-wise transprecision style to control differences between the `float` counterpart. After the overflow checking instructions, instead of a branch instruction pointing to the overflow handler, the boolean operator OR is applied between that overflow checking result register and an overflow flag.

4.2 Double Buffering

Double buffering is a powerful technique to resolve the issue of data pollution caused by overflown data. When matrix-wise computing is carried out, a potential overflow can contaminate the input matrix and write meaningless results into it. It is difficult to recover from overflown results, making it impossible to dispatch the same input matrix to an algorithm of higher precision and defeating the purpose of transprecision computing.

Double buffering addresses this problem by allocating two pieces of memory, one for the input matrix and the other for the output matrix. The input matrix is read-only, while the output matrix allows read and write. This separation of data storage ensures that the input matrix remains unpolluted by overflown data and that any potential overflow is encapsulated within the separate output matrix.

While making a copy of the input matrix is a simple and easy solution for protecting the original data from pollution, double buffering is a superior technique because it does not introduce additional memory operations. With double buffering, every operand needs a load operation from the input matrix, and every result takes a store operation to be written into the output matrix. This is the minimal amount of memory operation required for arithmetics. Zen 4 can load one `ZMM` vector per cycle and store one `ZMM` vector per two cycles. In comparison, it can do two multiplication or addition of `ZMM` vectors every cycle. The discrepancy in throughput between computing and IO suggests that memory copy is very expensive and inefficient.

4.3 Alignment

In the Listing 3.3 and Listing 3.4, the assembly for load and store are `vmovups` and `vmovdqu64`. `vmovups` for "Move Unaligned Packed Single-Precision Floating-Point Values" [5], and `instruction` for "Move Unaligned Packed Integer Values" [5].

An unaligned load may bring potential negative impacts on performance. When accessing memory, the CPU retrieves data from the main memory or cache in units of cache lines. On Zen 4, the size of a cache line is 512 bits, precisely the size of a `ZMM` register [1]. Alignment guarantees that a single cache line can cover a vector register. Otherwise, a `ZMM` register might be spitted on two cache lines. Both cache lines of 1028 bits have to be requested from the memory, then performs a shift to extract the desired 512 bits [21].

To address this issue, an aligned allocator is given to the constructor of `std::vector` when initializing a matrix [15]. Loading from an aligned address can be noticed by the compiler, then aligned load instructions `vmovdqa` or `vmovaps` will be issued accordingly.

4.4 Reduce Number of Matrix Index Computation

Section 3.2 discussed the benefits of using a single `std::vector` to represent a matrix. Given arbitrary row and column, the item of the matrix can be found in the array by computing the index: `column count * row + column`. The matrix data structure can be further optimized if the number of index computations can be reduced. By placing the pivot row as the first row in the matrix, it is only required to compute the address once for the first element of the matrix, and then every subsequent item can be indexed by incrementing the address.

4.5 Column Size Specialization

Figure 2.1 shows that 95% of the matrices have less than 20 columns while 75% of the matrices have more than 18 columns [14]. Therefore, a matrix of 19 columns is most likely to be encountered.

A `YMM` register can pack 8 `float`, and a `ZMM` register can pack 16 `float` (Figure 3.4). When padding each row of the 19-column matrix to 32 columns to fit in 2 `ZMM` registers, 13 `float` are wasted. Alternatively, the rows can be padded to 24 columns using 3 `YMM` registers, reducing the padding waste to 5 `float`.

Since `YMM` registers have double the amount of execution units than `ZMM` registers, while there will be only 50% more instruction count, the 3 `YMM` setup is expected to be faster than the more generic configuration of 32 columns and 2 `ZMM` registers.

4.6 Evaluation

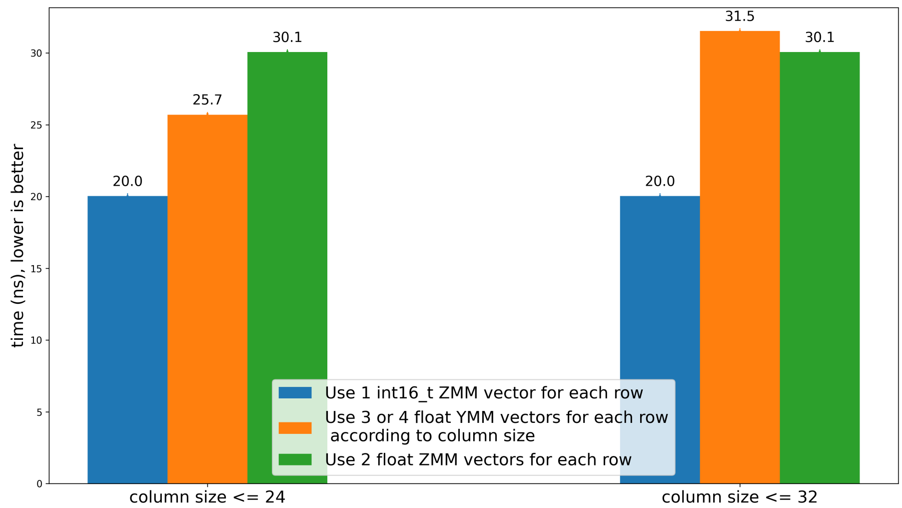


Figure 4.1: A benchmark of the `pivot` function. All implementations using `ZMM` require padding column size to 32. However, the combination of `float` and `YMM` allows some flexibility in padding size. In cases where the column size is less than 24, instead of padding to 32, the column size is padded to 24, and three `YMM` registers are allocated for each row. This approach has been found to be 15% faster than its `ZMM` counterpart.

By integrating the aforementioned optimizations and running a benchmark (Figure 4.1) for the `pivot` function, it is discovered that:

1. The `int16_t` approach from FPL [15] costs 20 ns, this is 6 ns faster than the `float` approach proposed by this report. A possible reason is that Zen 4 is more capable of doing arithmetics than memory IO [23]. The algorithm using `float` could be IO-bounded.

2. The benchmark confirms the expectation in Section 4.5.

3. For Zen 4, migrating AVX-2 code to AVX-512 indeed brings some performance increments. The `pivot` function using `float` and 2 `ZMM` for a row is 1.4 ns faster than the one using 4 `YMM` for a row. The benchmark confirms the theoretical benefit of AMD's implementation on AVX-512 mentioned in Section 2.3.

Chapter 5

Conclusion and Future Work

In conclusion, this report presents a fast implementation of the `pivot` function using `float` to address performance bottlenecks when MLIR analysis programs using linear programming of the simplex method. The procedures are:

1. Write source code using Clang's vector type extension to guarantee vectorization.
2. Reduce the bit width of each element in the input matrix. High-precision data types are unnecessary for `pivot`.
3. Perform integer arithmetics with FPU to leverage floating point's automatic overflow detection.

I achieved 20 times speedup over the upstream implementation, but unfortunately, it is about 20% slower than the FPL's approach. Nevertheless, my method is superior in terms of compatibility. The FPL's approach requires AVX-512, but mine works on old and vastly more common AVX-2 CPUs.

My techniques of optimizing the `pivot` function could make further progress with the 16-bit floating point data structure `half`. It consists of a 1-bit sign, a 5-bit exponent, and a 10-bit mantissa (Figure 5.1). For some AI applications where high precision is not needed, `half` provides performance benefits over `float` and `double` [24]. It has been supported natively on GPUs since 2016 [27], and if this data type is integrated into future AVX extensions, it would potentially further improve the performance of `pivot`. The `int10_t` data type can be defined using the 10-bit mantissa, which covers 99% of the elements in the constraint matrices [14]. It is expected to improve the runtime of the `pivot` function by 2 times if `int23_t` is replaced with `int10_t`.

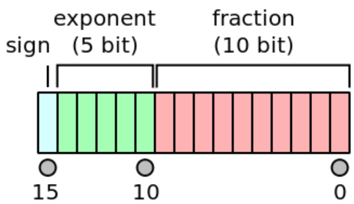


Figure 5.1: IEEE 754 half precision floating point (16 bits) [22].

Bibliography

[1] AMD. Software optimization guide for amd family 17h processors. https://www.amd.com/en/support/tech-docs/software-optimization-guide-for-amd-family-17h-processors, 2021.

[2] AOIDUO, LebedevRI, RKSimon, adibiagio, and llvmbot. Unexpected rthroughput for vfmadd* instructions in znver3. https://github.com/llvm/llvm-project/issues/59325, 2022.

[3] At32Hz. Zen 2 - microarchitectures - amd. https://en.wikichip.org/wiki/amd/microarchitectures/zen_2, 2022.

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

[5] Félix Cloutier. x86 and amd64 instruction reference. https://www.felixcloutier.com/x86/, 2022.

[6] LLVM Contributors. 'affine' dialect. https://mlir.llvm.org/docs/Dialects/Affine/, 2023.

[7] LLVM Contributors. Mlir llvm. https://mlir.llvm.org/, 2023.

[8] cppreference.com. Standard library header cfenv (c++11). https://en.cppreference.com/w/cpp/header/cfenv, 2022.

[9] Ian Cutress. The intel skylake-x review: Core i9 7900x, i7 7820x and i7 7800x tested. https://www.anandtech.com/show/11550/the-intel-skylakex-review-core-i9-7900x-i7-7820x-and-i7-7800x-tested, 2017.

[10] Google Benchmark Developers. google/benchmark. https://github.com/google/benchmark, 2023.

[11] Google Benchmark Developers. Reducing variance. https://github.com/google/benchmark/blob/main/docs/reducing_variance.md, 2023.

[12] LLVM Developers. llvm-mca - llvm machine code analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html, 2023.

[13] Niall Emmart, Fangyu Zheng, and Charles Weems. Faster modular exponentiation using double precision floating point arithmetic on the gpu. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 130–137, 2018.

[14] 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.

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

[16] Agner Fog. Optimizing software in c++. https://www.agner.org/optimize/optimizing_cpp.pdf, 2022.

[17] 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, June 2019.

[18] kingfish. How amd's zen 2 architecture boosts performance-per-watt. https://community.amd.com/t5/general-discussions/how-amd-s-zen-2-architecture-boosts-performance-per-watt/td-p/143054, 2019.

[19] Arch Linux. Cpu frequency scaling. https://wiki.archlinux.org/title/CPU_frequency_scaling, 2023.

[20] Zhe Liu, Jiankuo Dong, Fangyu Zheng, Wuqiong Pan, Jingqiang Lin, Jiwu Jing, and Yuan Zhao. Utilizing the double-precision floating-point computing power of gpus for rsa acceleration. In *Security and Communication Networks*, volume 2017, 2017.

[21] Mauricio Alvarez Mesa, Esther Salamí, Alex Ramírez, and Mateo Valero. Performance impact of unaligned memory operations in simd extensions for video codec applications. DBLP, April 2007.

[22] Islam Mohamed. Mixed precision training. https://islammohamedmosaad.github.io/journal/Mixed-Precision-Training.html, 2020.

[23] Mysticial. Zen4's avx512 teardown. https://www.mersenneforum.org/showthread.php?p=614191, 2022.

[24] NVIDIA. Train with mixed precision user's guide. https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html, 2023.

[25] robocat. Linus torvalds on avx512. https://news.ycombinator.com/item?id=23809335, 2020.

[26] Mariana Silva, Wanjun Jiang, Matthew West, Erin Carrier, Adam Stewart, and Luke Olson. Floating point representation. https://courses.physics.illinois.edu/cs357/sp2020/notes/ref-4-fp.html, 2020.

[27] Ryan Smith. The nvidia geforce gtx 1080 & gtx 1070 founders editions review: Kicking off the finfet generation. https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/5, 2016.

[28] Nigel Topham. Advanced arithmetic functions. Computer Architecture and Design (INFR10076) Lecture 21, 2021.

[29] Linus Torvalds. Alder lake and avx-512. https://www.realworldtech.com/forum/?threadid=193189&curpostid=193190, 2020.

[30] Xeon06. Floating point arithmetic. https://www.reddit.com/r/ProgrammerHumor/comments/1fq4jy/floating_point_arithmetic/, 2013.