Automatic Retuning of Floating-Point Precision

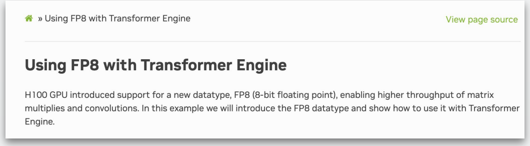
Ivan R. Ivanov (TokyoTech)   William S. Moses (UIUC)

ivanov.i.aa@m.titech.ac.jp, wsmoses@illinois.edu

Choosing the right floating point precision is important

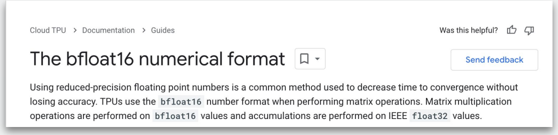
Too high a precision ⇨ For nothing - if we didn't need the precision

- More power
- Longer runtime


🏠 » Using FP8 with Transformer Engine   View page source

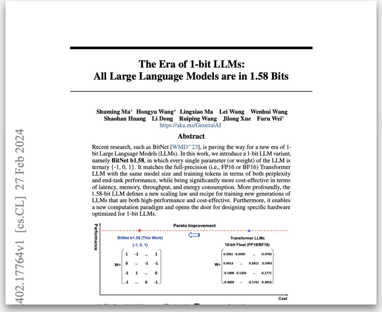
Using FP8 with Transformer Engine

H100 GPU introduced support for a new datatype, FP8 (8-bit floating point), enabling higher throughput of matrix multiplies and convolutions. In this example we will introduce the FP8 datatype and show how to use it with Transformer Engine.


Cloud TPU › Documentation › Guides   Was this helpful?

The bfloat16 numerical format   Send feedback

Using reduced-precision floating point numbers is a common method used to decrease time to convergence without losing accuracy. TPUs use the `bfloat16` number format when performing matrix operations. Matrix multiplication operations are performed on `bfloat16` values and accumulations are performed on IEEE `float32` values.


arXiv:2402.17764v1 [cs.CL] 27 Feb 2024

The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits

Shuming Ma* Hongyu Wang* Lingxiao Ma Lei Wang Wenhui Wang Shaohan Huang Li Dong Ruiping Wang Jilong Xue Furu Wei°
https://aka.ms/GeneralAI

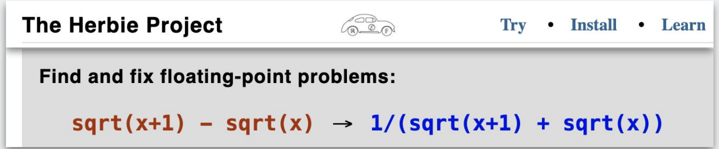
Abstract

Recent research, such as BitNet [WMD+23], is paving the way for a new era of 1-bit Large Language Models (LLMs). In this work, we introduce a 1-bit LLM variant, namely **BitNet b1.58**, in which every single parameter (or weight) of the LLM is ternary {-1, 0, 1}. It matches the full-precision (i.e., FP16 or BF16) Transformer LLM with the same model size and training tokens in terms of both perplexity and end-task performance, while being significantly more cost-effective in terms of latency, memory, throughput, and energy consumption. More profoundly, the 1.58-bit LLM defines a new scaling law and recipe for training new generations of LLMs that are both high-performance and cost-effective. Furthermore, it enables a new computation paradigm and opens the door for designing specific hardware optimized for 1-bit LLMs.

Too low?

- Useless results

  - some scientific applications require > 128 bit fp numbers to work


The Herbie Project   Try • Install • Learn

Find and fix floating-point problems:

`sqrt(x+1) - sqrt(x)  →  1/(sqrt(x+1) + sqrt(x))`

Future hardware...?

Existing Approaches to Re-tuning or exploration

Macros (#define float float8)

+ Easy to set up
- Programs will often fail to build (wrong size memory to copy, cannot cast to new type, conflicting names)
- Applies to entire program, not just the sub-computation we care about

Templating functions

- Requires rewriting code to be generic (and breaks library calls)
+ Easier compiler-level errors, and more brute force debugging than endless conflicts
+ Fast
- No analysis

Adapt

+ Analysis of errors
- Requires rewriting code
- Slow
- Does not reflect fp optimized state

Valgrind?

Existing Approaches to Re-tuning

Macros (#define float float8)

+ Easy to set up
- Programs will often fail to build (wrong size memory to copy, cannot cast to new type, conflicting names)
- Applies to entire program, not just the sub-computation we care about

Templating functions

- Requires rewriting code to be generic (and breaks library calls)
+ Easier compiler-level errors, and more brute force debugging than endless conflicts
+ Fast
- No analysis

Adapt

+ Analysis of errors
- Requires rewriting code
- Slow
- **Does not reflect fp optimized state**

Valgrind?

FP Optimizations

```
define fun():
  ...
  %c = fadd %a, %b : f32
  %e = fmul %d, %c : f32
  ...
```

```
define fun():
  ...
  %e = fma %a, %b, %d : f32
  ...
```

```
define fun():
  ...
  %c = call __fp_add(%a, %b) : %struct.fp
  %e = call __fp_mul(%d, %c) : %struct.fp
  ...
```

```
define fun():
  ...
  %c = call __fp_add(%a, %b) : %struct.fp
  %e = call __fp_mul(%d, %c) : %struct.fp
  ...
```

llvm: What is going on here???? I have no idea.

But what about optimization?

- Optimizations can change result! Sometimes good, sometimes bad
  - Program can now run faster at expense of error (e.g. fastmath)
  - Optimizers (e.g. Herbie) can optimize for more accurate!

⇩

- Source-level tooling can fail to predict actual results.
- Even better tools would be able to jointly optimize the precision and the computation!

`-fassociative-math`

> *Allow re-association of operands in series of floating-point operations.*

This allows the compiler to change the order of evaluation in a sequence of floating point operations. For example if you have an expression `(a + b) + c`, it can evaluate it instead as `a + (b + c)`. While these are mathematically equivalent with real numbers, they aren't equivalent in floating point arithmetic: the errors they incur can be different, in some cases quite significantly so:

```
julia> a = 1e9+1; b = -1e9; c = 0.1;

julia> (a+b)+c
1.1

julia> a+(b+c)
1.100000023841858
```

From Simon Byrne (https://simonbyrne.github.io/notes/fastmath/)

The Herbie Project Try • Install • Learn

Find and fix floating-point problems:

`sqrt(x+1) - sqrt(x)` → `1/(sqrt(x+1) + sqrt(x))`

Impact of optimizations?

On FPBench: **745 functions** with fp computation

```
clang -O3 -ffast-math ...
```

vs

```
clang -O0
```

Mismatched results on **124** of them

Impact of optimizations?

Mismatched results on **124**/745 functions

You may get unexpected results on real hardware

Our Approach

- We build off the Enzyme infrastructure for generating shadow computations from generic LLVM code (historically derivatives) to now generate new functions which replace floating representations.
- Apply optimization prior (and optionally after) floating tuning!
- No program rewriting required, just call the corresponding intrinsic function. (Or specify full-module effect on command line)

```
double
float    half    mpfr(5,20)    mpfr(5,20)    .....
```

Example Usage

```c
#define FROM 64
#define TO_E 5     // Exponent
#define TO_M 21    // Mantissa

double f(double x, double y) {
    return ...;
}

double a = 3, b = 4;

// Step 1: Call wrapper to convert inputs into specified precision
//         Here we convert from fp64 to fp32, using the old type
//         as storage for the truncated data.
a = __enzyme_truncate_mem_value(a, FROM, TO_E, TO_M);
b = __enzyme_truncate_mem_value(b, FROM, TO_E, TO_M);

// Step 2: Generate an fp32 version of the fp64 function
//         with our new intrinsic.
double res = __enzyme_truncate_mem_func(f, FROM, TO_E, TO_M)(a, b);

// Step 3: Convert the truncated result back into the full fp64.
res = __enzyme_expand_mem_value(res, FROM, TO_E, TO_M);
```

Example Usage

```
#define FROM 64
#define TO_E 5      // Exponent
#define TO_M 21     // Mantissa

double f(double x, double y) {
    return ...;
}

double a = 3, b = 4;

// Step 1: Call w
//         Here w
//         as sto
a = __enzyme_trun
b = __enzyme_trun

// Step 2: Genera
//         with o
double res = __en

// Step 3: Conver
res = __enzyme_expand_mem_value(res, FROM, TO_E, TO_M);
```

You can also specify on the command line:

```
clang ... --enzyme-truncate-all="64to5-21"
```

For full-module effect

Future Work & Conclusions

- Being aware of the floating point representation used is critical to both performance and accuracy!
- Optimizing floating point choices inherently requires understanding the impact of normal compiler optimization
- Our tool builds off Enzyme/LLVM to automatically retune floats alongside optimization without programmer rewriting
- Future work:
  - Combine with program analysis to determine which operations requiring tuning, and by what amount
  - Tune only individual parts of a function but leave other parts at full precision
  - Consider other sources of error than floating point
- All open source (https://github.com/EnzymeAD/Enzyme), please reach out if interested!