High-performance vector code for FPL

The Fast Presburger Library (FPL) spends a significant time on performing row operations on small matrices. These matrices are typically 8-16 columns wide, and the coefficients in these matrices are typically small (smaller than 16-bits). One of the hottest loops is the Simplex::pivot(), which scans over the matrix and performs on each row a mix of additions and multiplications. We already demonstrated in Fast linear programming through transprecision computing on small and sparse data that speculating on these small values can give nice speedups. Yet, these optimizations have not yet been upstreamed to FPL, and before doing so it might be worth carefully considering the various explored and unexplored approaches.

Overview

There are a number of approaches to perform overflow-checked vector operations on integers.

- Use vectorized integer operations
  (+) There is reasonable support for this in the LLVM backends
  (-) Overflow checks require additional operations

  ZEN 4 has fast 64-bit integer multiplication, which will make 32-bit overflow-checked integer arithmetic fast.

- Model integers with floating point operations + inexact exception

  (+) Zero overhead in the no-overflow case
  (+) Potentially a wider range for power-of-two numbers
  (+) Fast 52 bit integer multiply with check of the high bits
  (+) Works across many architectures

  (-) Catching exceptions is very hacky

Use vectorized integer operations

We can use LLVM's builtin for vectorized integer arithmetic with overflow checking.

Model integers with floating point operations + inexact exception

Double precision floating point numbers are encoded as follows:

- 1 bit: sign
- 11 bits: exponent
- 52 bits: sigificand precision

This means, they can store up to 52 bit integers without rounding.

Any such number has an exponent of '01111111111' (zero offset).

In fact, certain larger numbers can also be modeled precisely.

Distinguishing from pointer

User-space pointers on x86 Linux will always start from the lower part of the address space and will consequently have leading zeros [citation needed]. Consequently, they do not compare equal to any floating point number that is integer-exact. As a result, comparisons with integer constants (e.g., 0, -1, 1) should be fast.

Check if a number is a pointer

- Check if 3rd bit is zero, if we limit ourselves to numbers with zero offset
- Encode pointers as NANs and trap if we encounter one.

Check if a number has overflown

Floating point arithmetic allows us to check at zero cost if an operation has become inexact. That is very helpful.

- 'https://randomascii.wordpress.com/2012/04/21/exceptional-floating-point/'

```
#include <stdio.h>
#include <cfenv>

#pragma STDC FENV_ACCESS ON

int main() {
        std::feclearexcept (FE_ALL_EXCEPT);
        feenableexcept (FE_INEXACT | FE_INVALID);
        double nan = std::numeric_limits<double>::signaling_NaN();

        double a = (2l <<10) + 13;
        __sync_synchronize();
        double b = a * a;
        __sync_synchronize();
        if (std::fetestexcept(FE_INEXACT)) printf ("inexact 1st\n");
        double c = (2l <<10) + 13;
        __sync_synchronize();
        double d = c * nan;
        printf("%f", d);
        __sync_synchronize();
        if (std::fetestexcept(FE_INEXACT)) printf ("inexact 2nd\n");
}
```

FMA

Vector floating point instruction sets, e.g., AVX512, offer fast vector fused multiply-add instructions.

Division

Floating point division has been historically faster: https://stackoverflow.com/questions/70132913/fast-hardware-integer-division