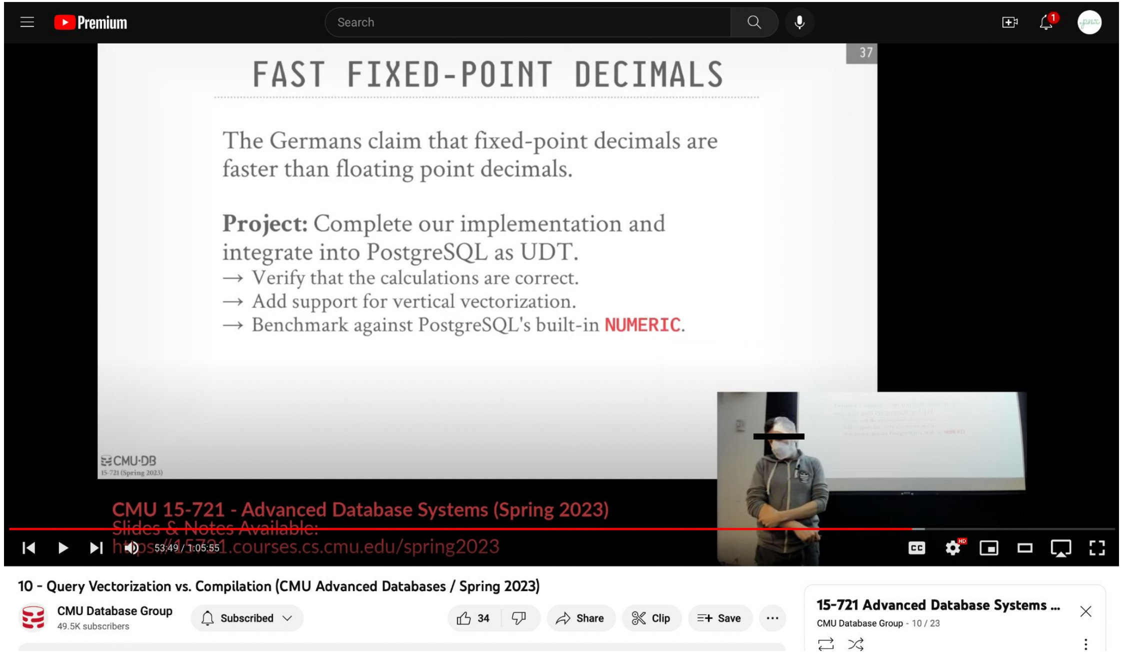
FIXEYPOINTY

Fast Fixed-Point Decimals

Final Presentation

Jekyeom Jeon
Taoxi Jiang
Yuttapichai (Guide) Kerdcharoen

Re-iterate the motivation…



Source: https://www.youtube.com/watch?v=rhxd_xaeMPU

What we did (in a slide)

LIBFIXEYPOINTY

CMU-DB's standalone
128-bit fixed-point decimal

- Documented **libfixeypointy**
- Hardened **libfixeypointy**
- Improved **libfixeypointy**'s multiplication and division performance
- Evaluated **libfixeypointy** against other standalone decimal implementations

FIXEYPOINTY

libfixeypointy as
PostgreSQL's UDT

- Integrated **libfixeypointy** as PostgreSQL user-defined types, **fxypty**
  - Including arithmetic and relative operators
  - Basic aggregators (sum, min, max, count)
- Evaluated **fxypty** against PostgreSQL NUMERIC (its built-in fixed-point decimal type) and DOUBLE, REAL (its floating-point type)

All the code are documented

LIBFIXEYPOINTY

Doxygen Comment

```
/**
 * Calculate product of two unsigned intergers of arbitrary
 * 64-bit chunks.
 * @par
 * @param half_words_b
 * @param
 * @param m The size of the half_words_a
 * @param
 */
void Ca
```

Complex Code Explanation

```
Decimal::NativeType Decimal::DivideByMagicNumbers256(const ui
                                                          Algorith
  // Hacker's Delight [2E Chapter 10 Integer Division by Cons
  // floor(a / x) = floor((a * m) / 2^p)
  // x = floor(2^p / m) where p and m are integers
  // p and m should be minimized
  // x = 1, then p = 0, m = 1
  // x = 2, then p = 1, m = 1
  // x = 3, then p = 4, m = 5 (since floor(16 / 5) = 3)

  uint128_t half_words_magic_result[8];
```

FIXEYPOINTY

Doxygen Comment

```
/// @brief Add two fxypty objects.
/// @param a The pointer to the first fxypty object.
/// @param b The pointer to the second fxypty object.
/// @return The pointer to the new fxypty object containing the sum of both the
/// fxypty objects.
extern "C" void *_fxypty_add(void *a, void *b) {
    FxyPty_Decimal *wrapped_a = (FxyPty_Decimal *)a;
    FxyPty_Decimal *wrapped_b = (FxyPty_Decimal *)b;
    assert(wrapped_a->scale == wrapped_b->scale);

    FxyPty_Decimal *result = (FxyPty_Decimal *)palloc(sizeof(FxyPty_Decimal));
    result->scale = wrapped_a->scale;

    try {
        libfixeypointy::Decimal tmp(_pack128(wrapped_a));
```

More libfixeypointy boundary cases are handled

LIBFIXEYPOINTY

Division by zero

```
Decimal::Divide(const Decimal &denominator, const ScaleType &scale
1. Multiply the dividend with 10^(denominator scale), with overflo
2. If overflow, divide by the denominator with multi-word 256-bit
3. If no overflow, divide by the denominator with magic numbers if
Moreover, the result is in the numerator's scale for technical rea
If the result were to be in the denominator's scale, the first ste
10^(2*denominator scale - numerator scale) which requires 256-bit

If value is 0
(denominator.ToNative() == 0) {
hrow std::runtime_error("Divided by 0");
```

Overflow

```
d Decimal::Add(const Decimal &other) {
/ If both values are positive, it is possible to get overflowed
f (value_ > 0 && other.value_ > 0) {
// Compute the maximum and safe value for other
// E.g., 63 - 62 = 1
// So, if other = 1, 62 + 1 = 63 (safe)
// But if other = 2 (> 1), 62 + 2 = 64 (overflowed)
int128_t other_bound = std::numeric_limits<__int128>::max() - valu
if (other.value_ > other_bound) {
  throw std::runtime_error("Result overflow > 128 bits");
}
```

The performance bottleneck is grade-school multiply

Observation

- Both multiplication and division (by magic number) use 128-bit grade-school multiplication
- An existing grade-school multiply implementation contains a number of loops and potential bubbles

Assumption

- Unwinding the loop and manually reordering instructions (to avoid bubbles) could improve multiplication and division performance

Optimizing by unwinding loops and reordering instructions

1. Let m and n = 2 and unwind loops

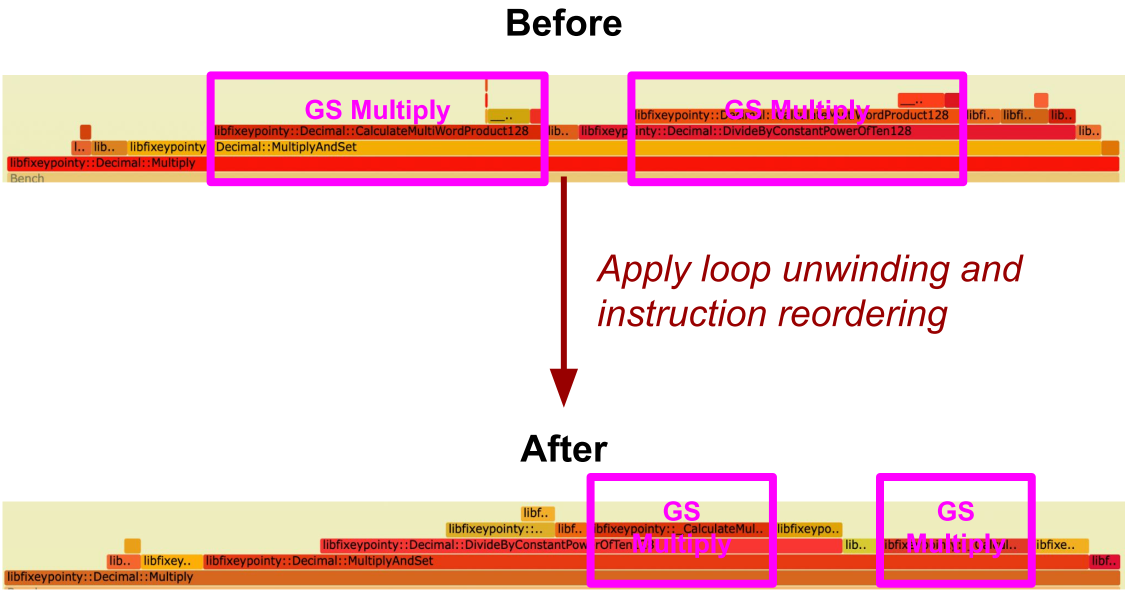
```
uint128_t k, t;
uint32_t i, j;
constexpr const uint128_t botto
// Initialize first m chunks wi
for (i = 0; i < m; i++) half_wo
// For each chunk in b
for (j = 0; j < n; j++) {
  k = 0;
  // Match with all chunks in a
  for (i = 0; i < m; i++) {
    // Product + Old Value (hal
    t = half_words_a[i] * half_
    // Take only bottom 64 bits
    half_words_result[i + j] =
    // Carry
    k = t >> 64;
  }
  half_words_result[j + m] = k;
}
```

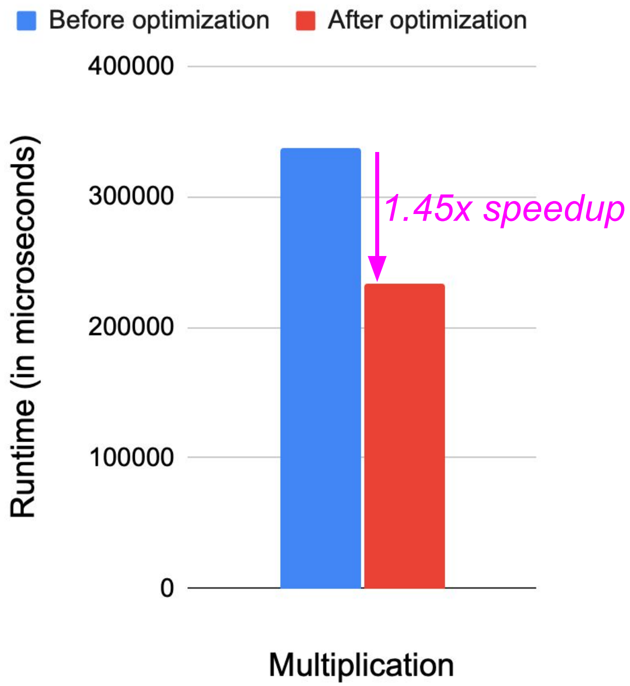
2. Reorder/remove instructions

```
uint128_t k, t;
constexpr const uint128_t bottom_mask = (uint
// Initialize first m chunks with 0
half_words_result[0] = 0;
half_words_result[1] = 0;
// For each chunk in b
// j = 0
k = 0;
// j = 0, i = 0
t = half_words_a[0] * half_words_b[0] + half_
half_words_result[0] = t & bottom_mask;
k = t >> 64;
// j = 0, i = 1
t = half_words_a[1] * half_words_b[0] + half_
half_words_result[1] = t & bottom_mask;
k = t >> 64;
// End
half_words_result[2] = k;
// j = 1
k = 0;
// j = 1, i = 0
t = half_words_a[0] * half_words_b[1] + half_
half_words_result[1] = t & bottom_mask;
k = t >> 64;
// j = 2, i = 1
t = half_words_a[1] * half_words_b[1] + half_
half_words_result[2] = t & bottom_mask;
k = t >> 64;
// End
half_words_result[3] = k;
```

```
void _CalculateMultiWordProduct128_2_2(const uint128_t *const half_words
                                        uint128_t *half_words_result) {
  constexpr const uint128_t bottom_mask = (uint128_t{1} << 64) - 1;
  uint128_t t2 = (half_words_a[1] * half_words_b[0]);
  uint128_t t1 = (half_words_a[0] * half_words_b[0]);
  t2 += (t1 >> 64);
  half_words_result[0] = t1 & bottom_mask;
  uint128_t t3 = (t2 >> 64);
  t3 += (half_words_a[1] * half_words_b[1]);
  t2 &= bottom_mask;
  t2 += (half_words_a[0] * half_words_b[1]);
  t3 += (t2 >> 64);
  half_words_result[3] = (t3 >> 64);
  half_words_result[1] = t2 & bottom_mask;
  half_words_result[2] = t3 & bottom_mask;
}
```

Grade-school multiply is no longer bottleneck





Division

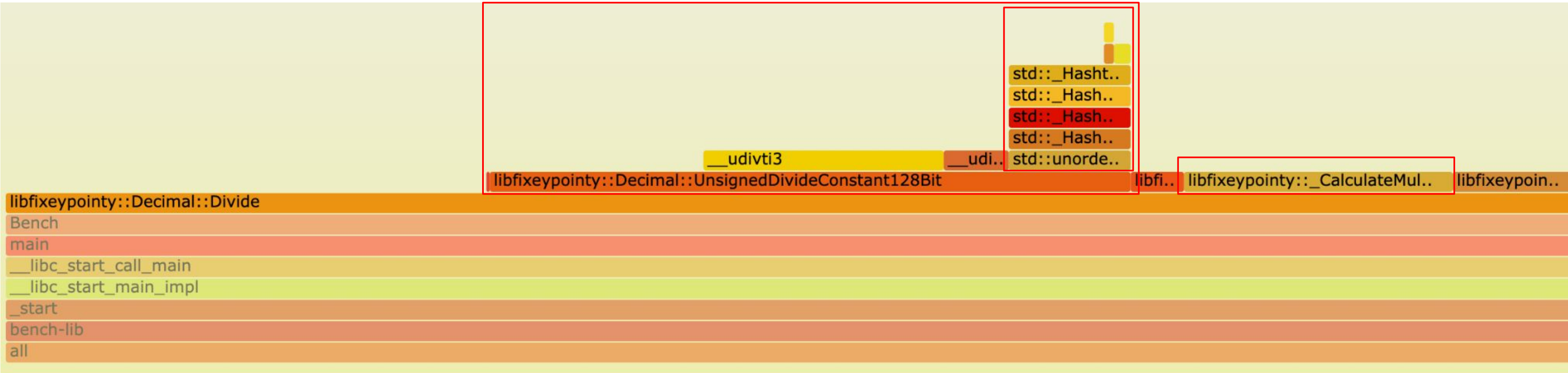
- Specify custom predefined magic number to speed up division (and multiplication)
- Not too many of them (depending on the operations you normally want to do)
- No magic number -> predicate small -> hot path taking the branch (good)
- A few magic number -> hot path not taking the branch -> predicate small (good)
- Lots of magic number -> predicate big -> access pattern uniform anyway, doesn’t make sense to add those magic number (bad)
- Generate and cache all seen magic numbers? -> will test in the future

Todo: **Macro** to convert static hashtable lookup to compiled predicates

… div by zero / power of 2 check

```
// 2. If not possible, regular division.
{
  if (MAGIC_CUSTOM_128BIT_CONSTANT_DIVISION.count(constant) == 0) {
    value_ = static_cast<uint128_t>(value_) / constant;
    return;
  }
}
```

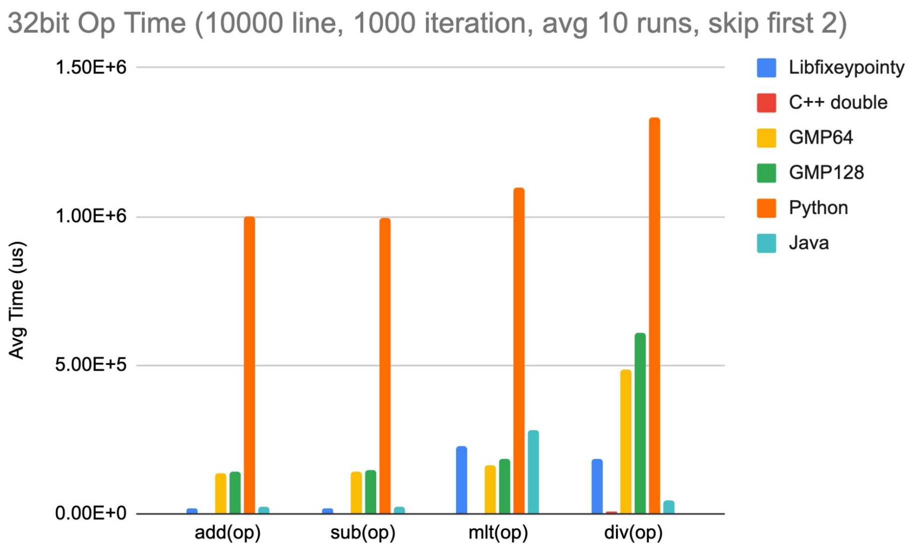
… Magic Number Division

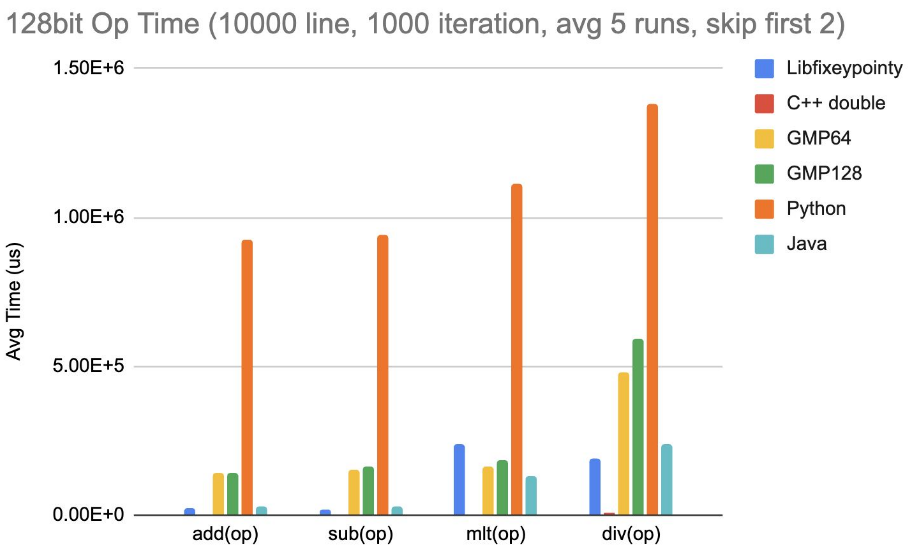


Verification

- Python decimal (hybrid of fixed point and float)
  - Random **op1 +-*/ op2**, repeated millions of times
  - Compare rounded off values
- Java BigDecimal
  - Long random chain **op1 +-*/ op2 +-*/ op3 …**
  - Compare error handling behavior (overflow), report and revert to previous value when error encountered
  - Compare exact values
  - Results exactly match

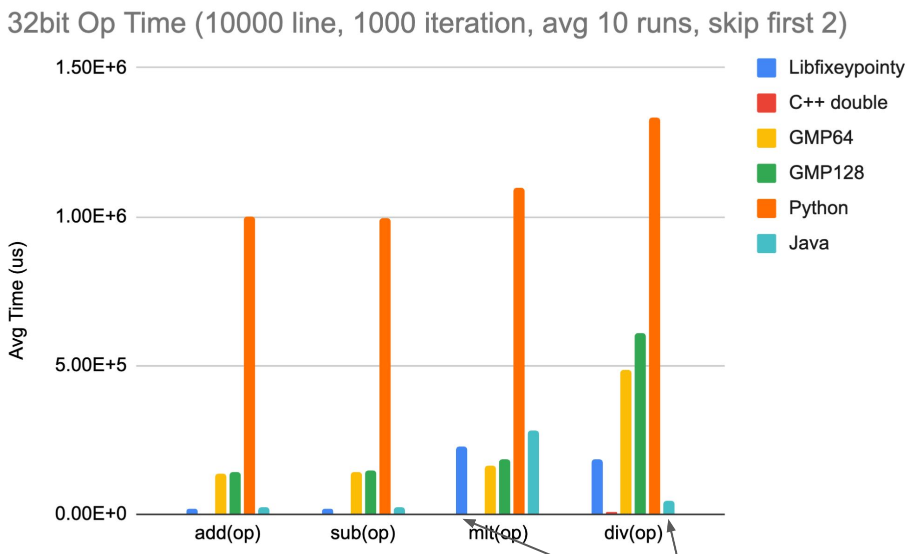
Evaluation (Standalone Lib performance)

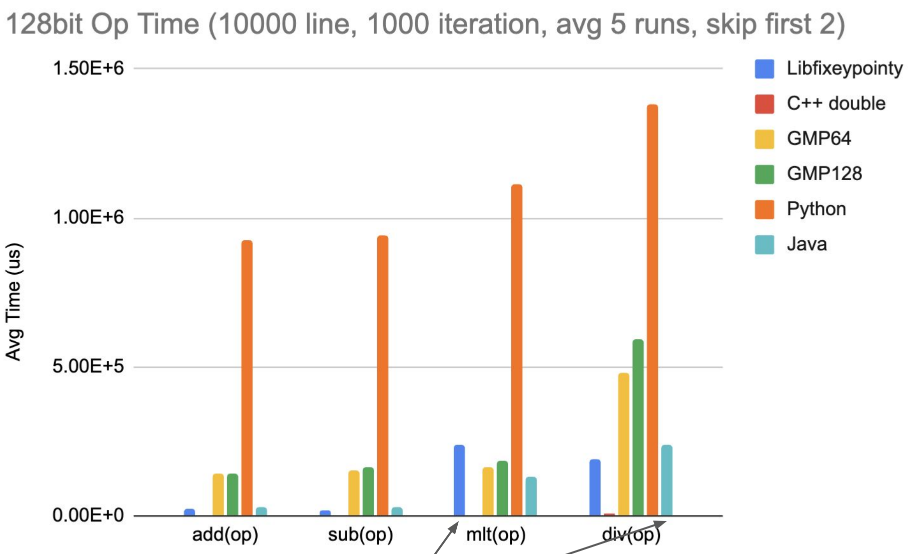




Operations on decimals stored in 32bit size, most digits **before** decimal point (scale is small)
Dataset fits in L3 cache. Larger dataset results will come in future.

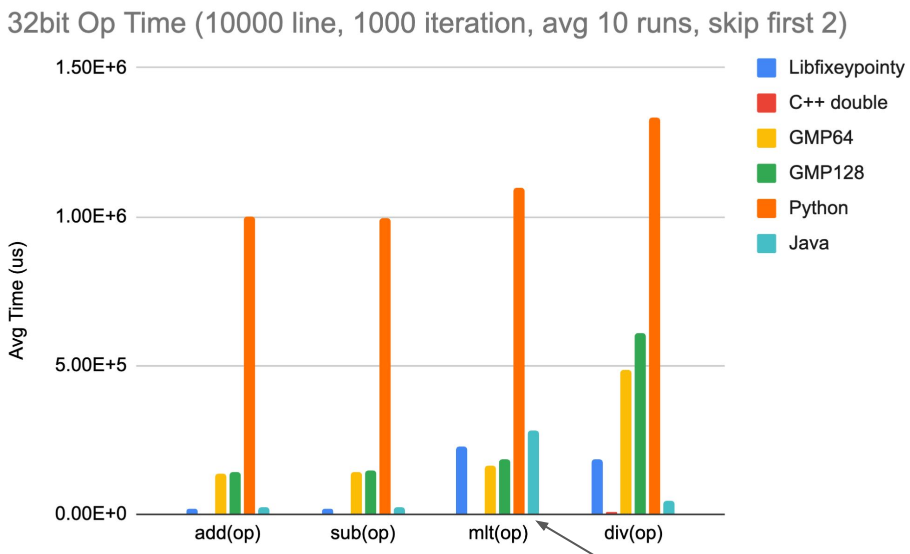
Evaluation (Standalone Lib performance)

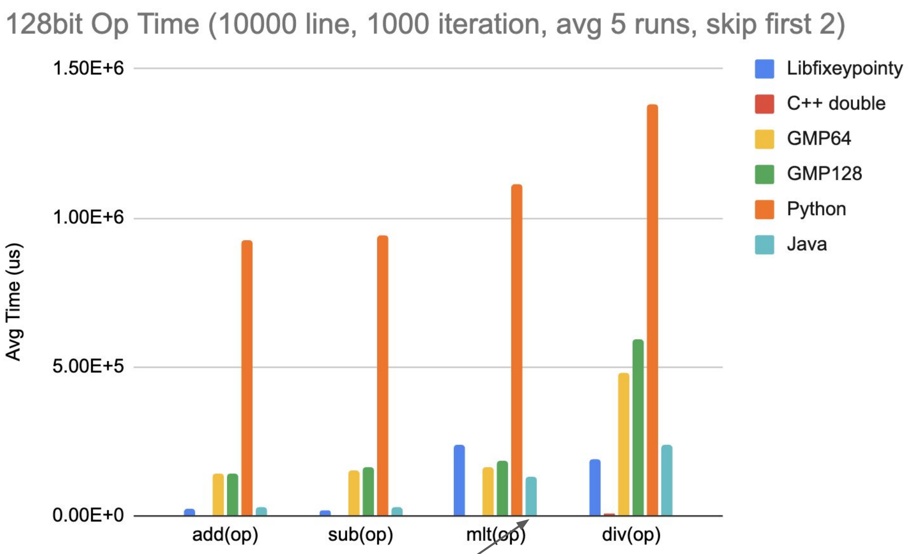




BigDecimal optim for small scale decimals
Libfixeypointy does not, runtime consistent

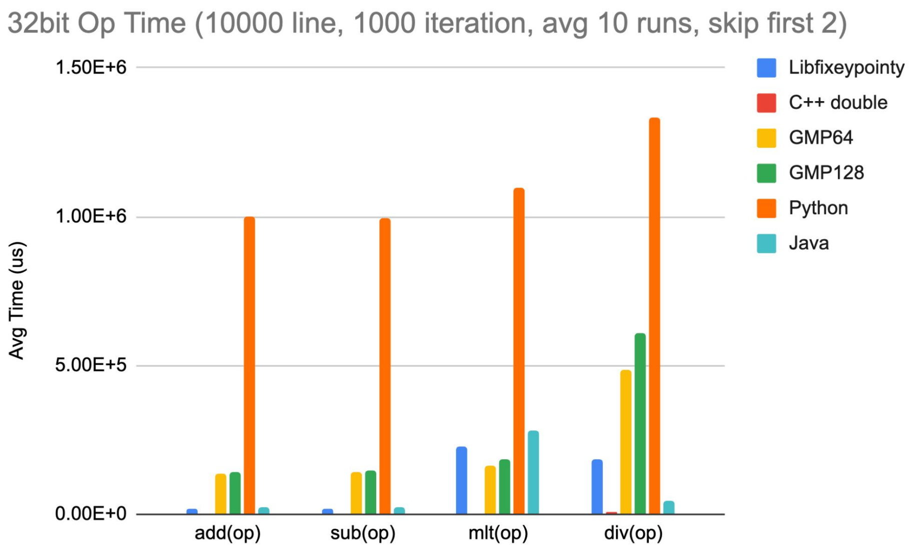
Evaluation (Standalone Lib performance)

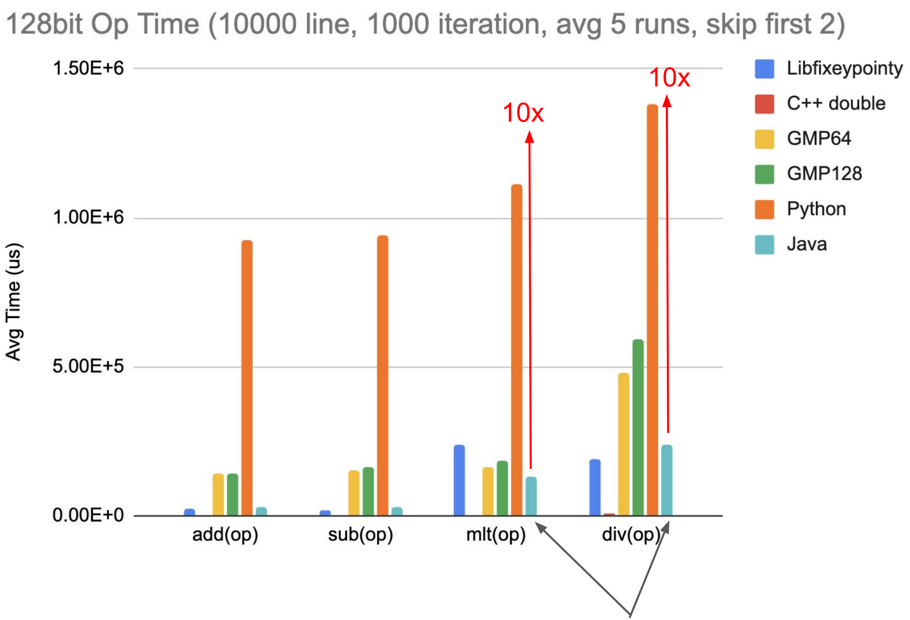




BigDecimal change both storage and mult algo for bigger decimals

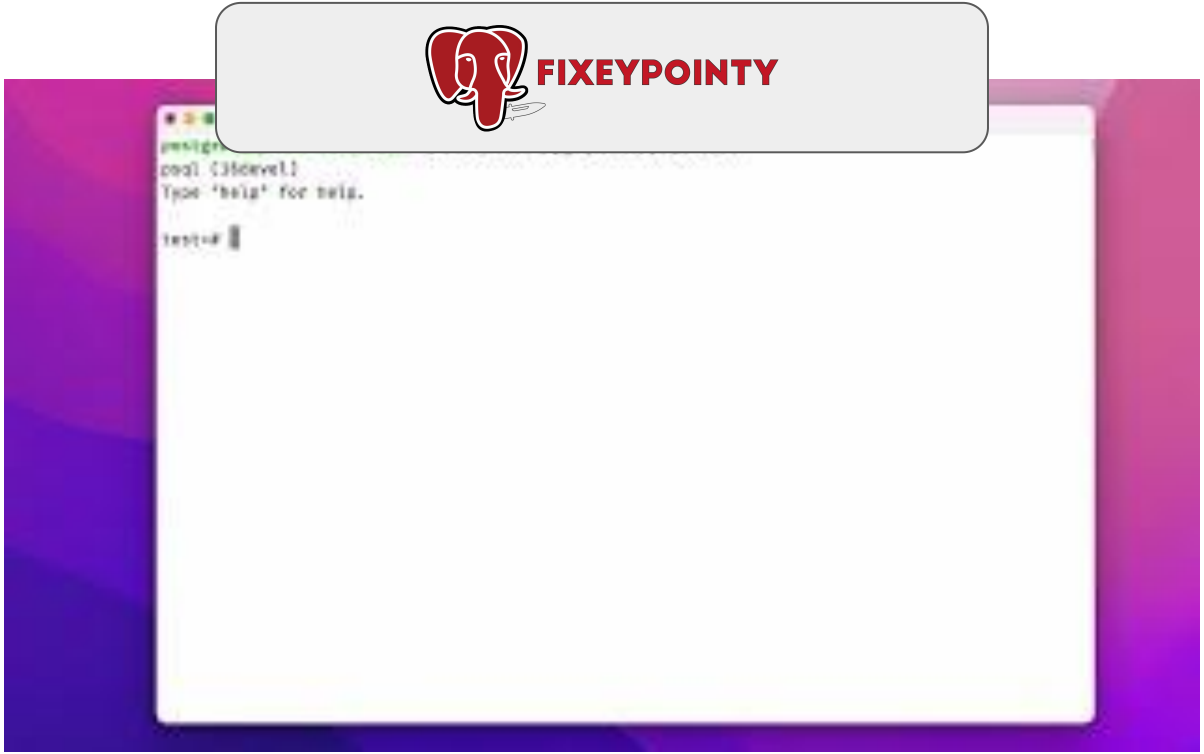
Evaluation (Standalone Lib performance)



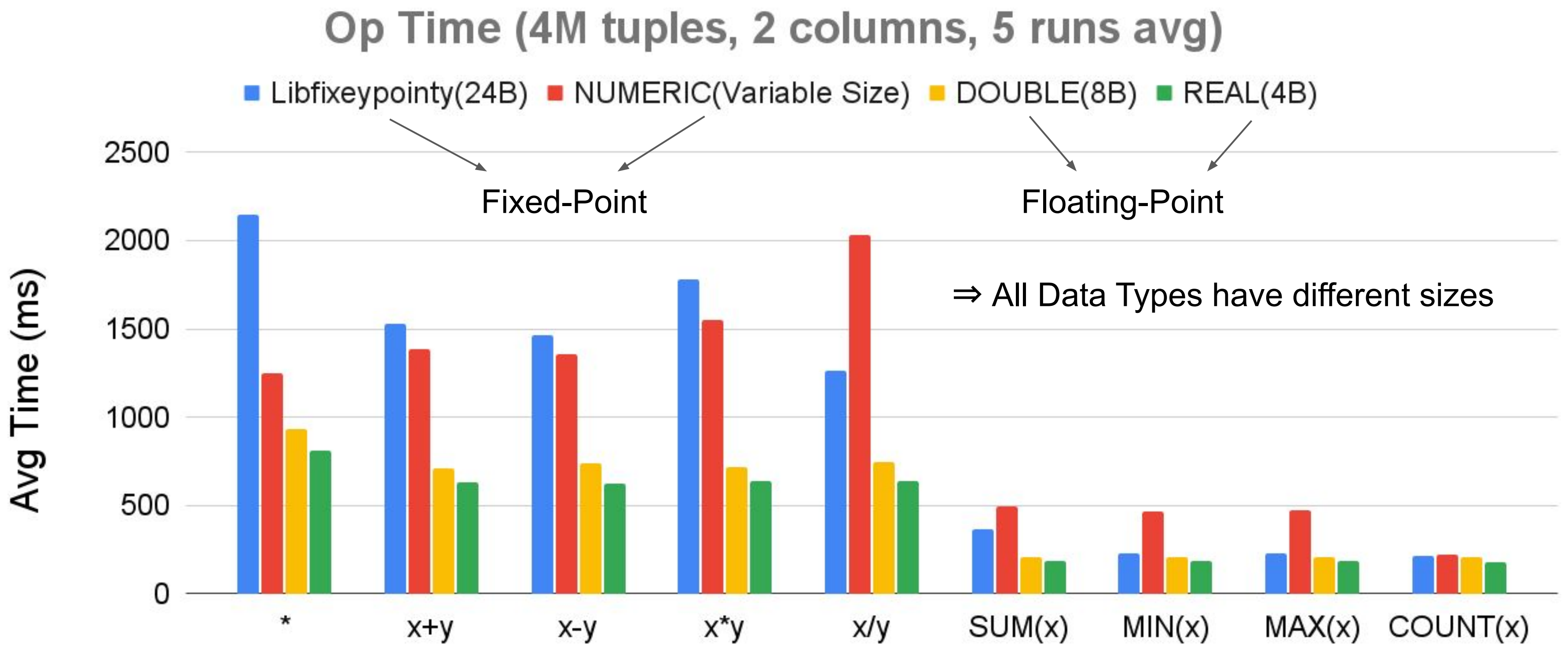


If we force BigDecimal to use our fixed max precision (28) and then round to scale, it becomes ~10x slower

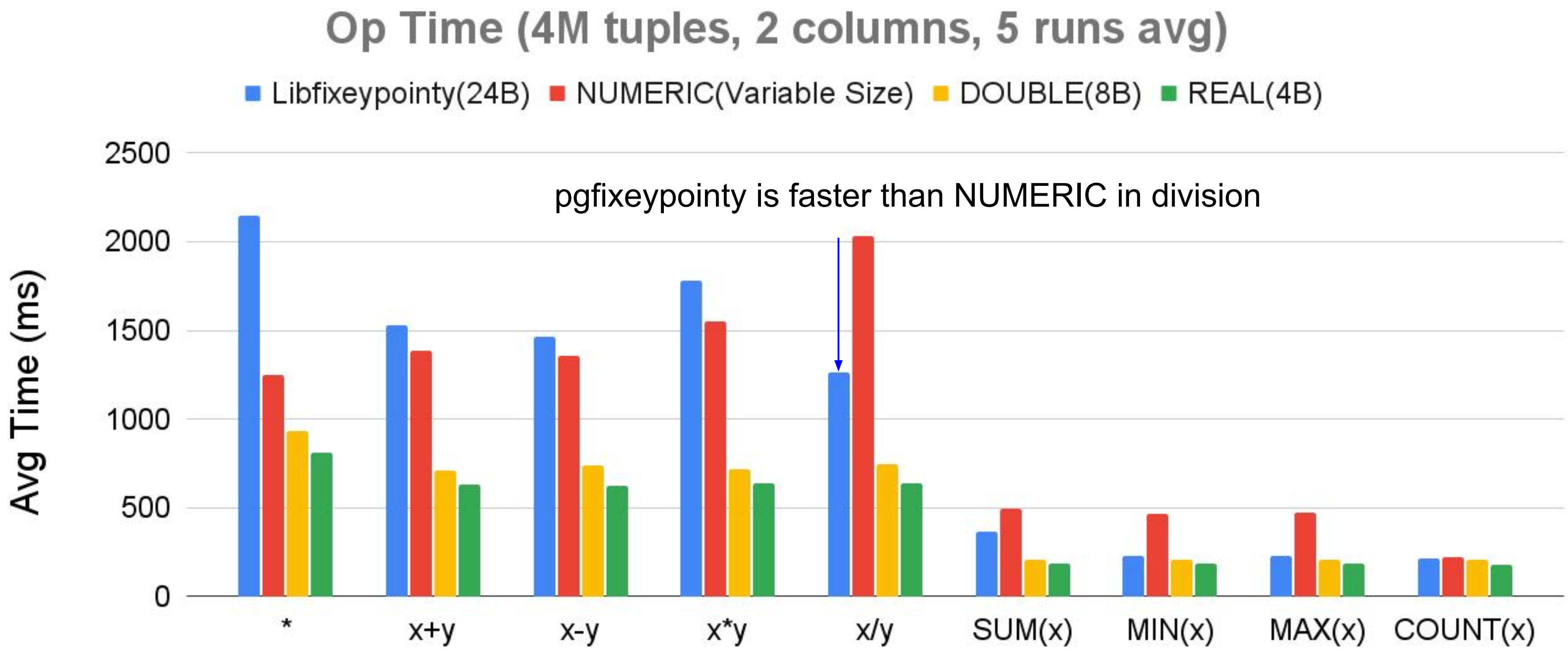
Integrating libfixeypointy as PostgreSQL's UDT



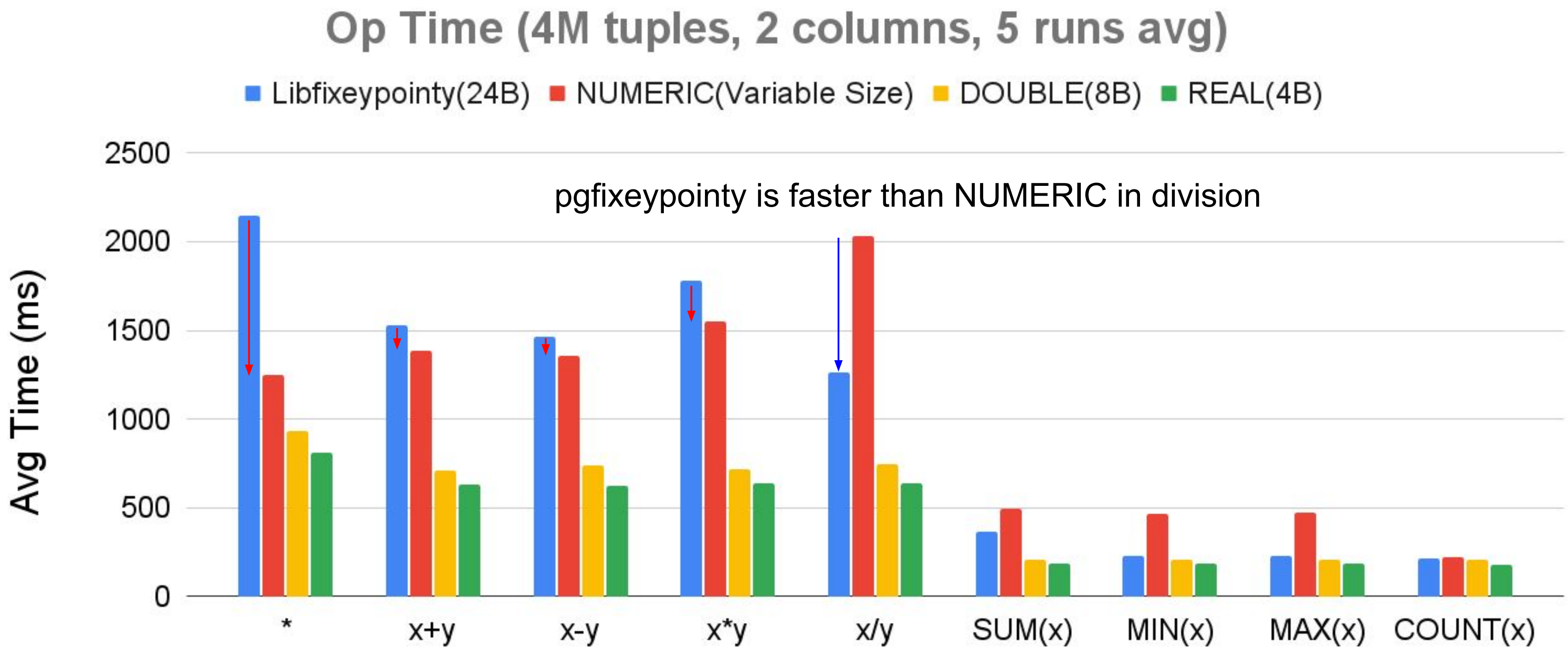
Evaluation (PostgreSQL operation performance)



Evaluation (PostgreSQL operation performance)

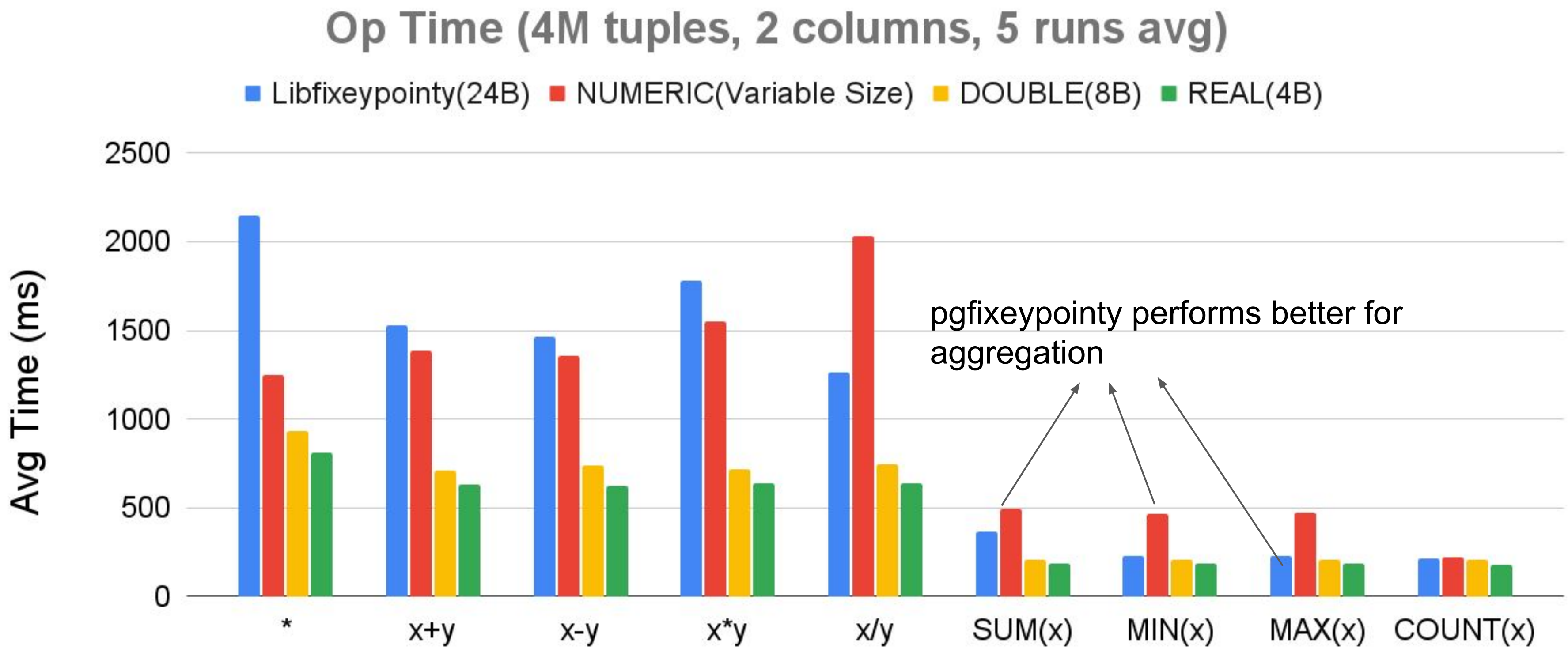


Evaluation (PostgreSQL operation performance)

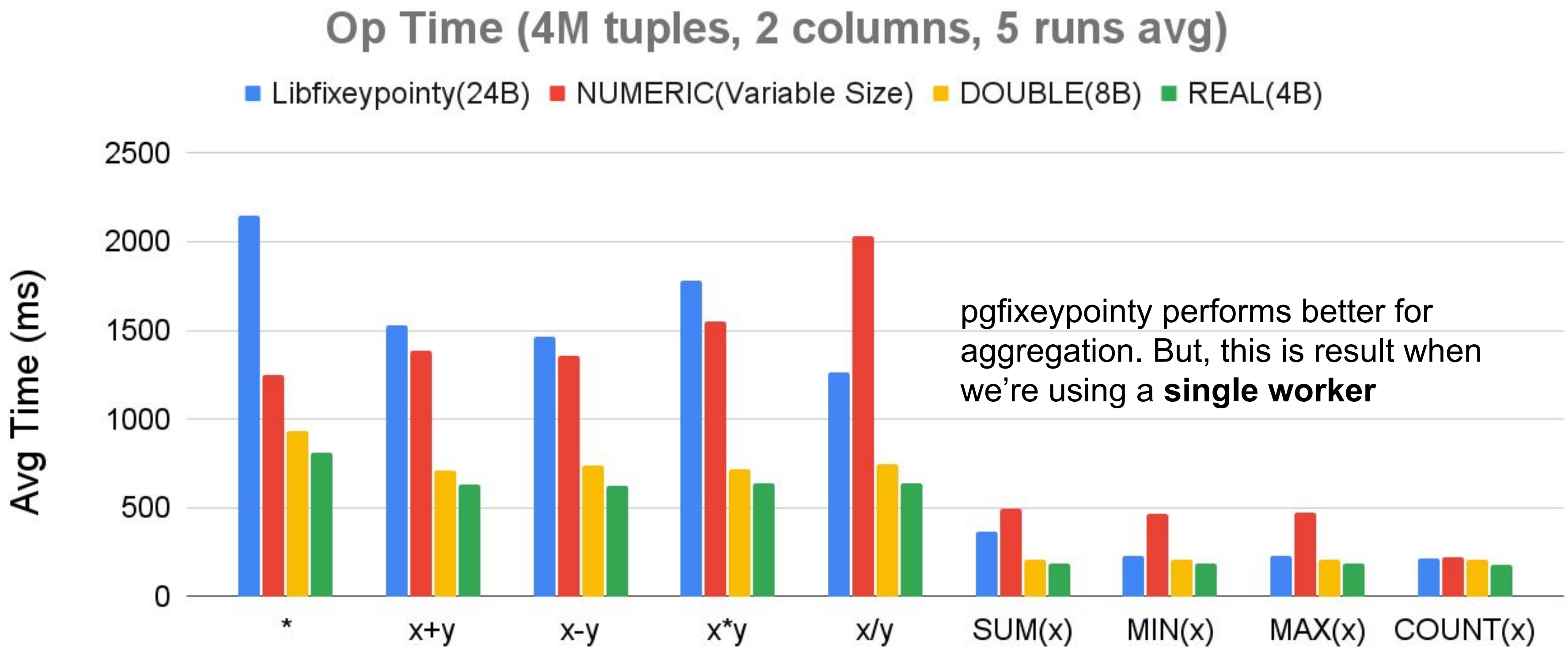


pgfixeypointy is slower than NUMERIC (ToString() is slow)

Evaluation (PostgreSQL operation performance)



Evaluation (PostgreSQL operation performance)



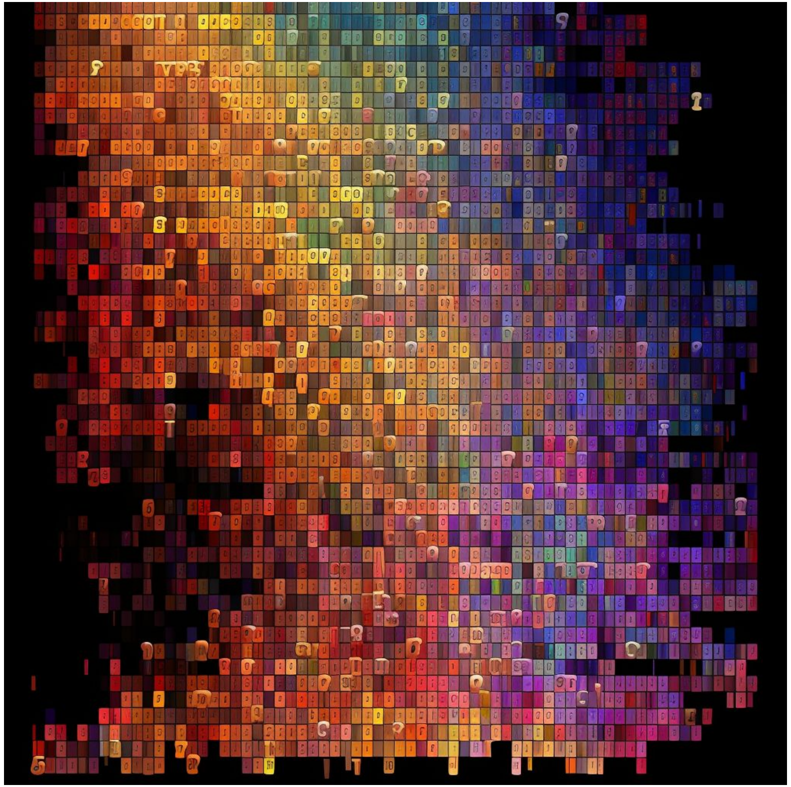
pgfixeypointy performs better for aggregation. But, this is result when we’re using a **single worker**

Future Work

- ➢ Support variable size: store a small decimal in a 64-bit or 32-bit
- ➢ Parallel aggregation to improve throughput
- ➢ Running perf to find an opportunity for optimizing multiplication performance
- ➢ Improve result writing (probably, string conversion) performance
- ➢ More Aggregator support: AVG, STD, VAR
- ➢ Type Casting: Operations between different types (e.g., double+libfixeypointy)
- ➢ More realistic workloads

Resources

- libfixeypointy - https://github.com/cmu-db/libfixeypointy/tree/develop
- pgfixeypointy - https://github.com/pnxguide/pgfixeypointy



Prompt: *a fast stream of 128-bit fixed-point decimals*