Implementing High-Precision Decimal Arithmetic with CUDA int128

“Truth is much too complicated to allow anything but approximations.” — **John von Neumann**

The history of computing has demonstrated that there is no limit to what can be achieved with the relatively simple arithmetic implemented in computer hardware. But the “truth” that computers represent using finite-size numbers is fundamentally approximate. As David Goldberg wrote, “Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation.” Floating point is the most widely used representation of real numbers, implemented in many processors, including GPUs. It is popular due to its ability to represent a large dynamic range of values and to trade off range and precision.

Unfortunately, floating point’s flexibility and range can cause trouble in applications where accuracy within a fixed range is more important: think dollars and cents. Binary floating point numbers cannot exactly represent every decimal value, and their approximation and rounding can lead to accumulation of errors that may be unacceptable in accounting calculations, for example. Moreover, adding very large and very small floating-point numbers can result in truncation errors. For these reasons, many currency and accounting computations are implemented using fixed-point decimal arithmetic, which stores a fixed number of fractional digits. Depending on the range required, fixed-point arithmetic may need a larger number of bits.

NVIDIA GPUs do not implement fixed-point arithmetic in hardware, but a GPU-accelerated software implementation can be efficient. In fact, RAPIDS cuDF library has provided efficient 32- and 64-bit fixed-point decimal numbers and computation for a while now. But some users of RAPIDS cuDF and GPU-accelerated Apache Spark need the higher range and precision provided by 128-bit decimals, and now NVIDIA CUDA 11.5 provides preview support of the 128-bit integer type(`int128`) that is needed to implement 128-bit decimal arithmetic.

In this post, after introducing CUDA’s new int128 support, we detail how we implemented decimal fixed-point arithmetic on top of it. We then demonstrate how 128-bit fixed-point support in RAPIDS cuDF enables key Apache Spark workloads to run entirely on GPU.

Introducing CUDA __int128

In NVIDIA CUDA 11.5, the NVCC offline compiler has added preview support for the signed and unsigned `__int128` data types on platforms where the host compiler supports it. The `nvrtc` JIT compiler has also added support for 128-bit integers, but requires a command-line option, `--device-int128`, to enable this support. Arithmetic, logical, and bitwise operations are all supported on 128-bit integers. Note that DWARF debug support for 128-bit integers is not available yet and will be available in a subsequent CUDA release. With the 11.6 release, cuda-gdb and Nsight Visual Studio Code Edition have added support for inspecting this new variable type.

NVIDIA GPUs compute integers in 32-bit quantities, so 128-bit integers are represented using four 32-bit unsigned integers. The addition, subtraction, and multiplication algorithms are straightforward and use the built-in PTX addc/madc instructions to handle multiple-precision values. Division and remainder are implemented using a simple O(n^2) division algorithm, similar to Algorithm 1.6 in Brent and Zimmermann's book *Modern Computer Arithmetic*, with a few optimizations to improve the quotient selection step and minimize correction steps.One of the motivating use cases for 128-bit integers is using them to implement decimal fixed-point arithmetic. 128-bit decimal fixed-point support is included in the 21.12 release of RAPIDS libcudf. Keep reading to find out more about fixed-point arithmetic and how `__int128` is used to enable high-precision computation.

Fixed-point Arithmetic

Fixed-point numbers represent real numbers by storing a fixed number of digits for the fractional part. Fixed-point numbers can also be used to "omit" the lower-order digits of integer values (i.e. if you want to represent multiples of 1000, you can use a base-10 fixed-point number with scale equal to 3). One easy way to remember the difference between fixed-point and floating point is that with fixed-point numbers, the decimal "point" is fixed, whereas in floating-point numbers the decimal "point" can float (move).

The basic idea behind fixed-point numbers is that even though the values being represented can have fractional digits (aka the 0.23 in 1.23), you actually store the value as an integer. To represent 1.23, for example, you can construct a `fixed_point` number with `scale = -2` and value 123. This representation can be converted to a floating point number by multiplying the value by the radix raised to the scale. So in our example, 1.23 is produced by multiplying 123 (value) by 0.01 (10 (radix) to the power of -2 (scale)). When constructing a fixed-point number, the opposite occurs and you "shift" the value so that you can store it as an integer (with the floating point number 1.23 you would divide by 0.01 if you were using scale -2.

Note that fixed-point representations are not unique because you can choose multiple scales. For the example of 1.23, you can use any scale less than -2, such as -3 or -4. The only difference is that the number stored on disk will be different; 123 for scale -2, 1230 for scale -3 and 12300 for scale -4. When you know that your use case only requires a set number of decimal places, you should use the *least precise* (aka largest) scale possible to maximize the range of representable values. A 32-bit decimal fixed-point with scale -2 has a range of roughly -20 to +20 million (with two decimal places), whereas with scale -3 the range is roughly -2 to +2 million (with three decimal places). If you know you are modeling money and you don't need three decimal places, scale -2 is a much better option.

Another parameter of a fixed-point type is the base. In the examples in this post, and in RAPIDS cuDF, we use base 10, or decimal fixed point. Decimal fixed point is the easiest to think about because we are comfortable with decimal (base 10) numbers. The other common base for fixed-point numbers is base 2, also known as binary fixed point. This simply means that instead of shifting value by powers of 10, the scale shifts a value by powers of 2. You can see some examples of binary and decimal fixed-point values later in the "Examples" section.

Fixed point vs floating point

Fixed Point	Floating Point
Narrower, static range	Wider, dynamic range
Exact representation avoids certain truncation and rounding errors	Approximate representation leads to certain truncation and rounding errors
Keeps absolute error constant	Keeps relative error constant

Table 1: Comparison of fixed point and floating point representations.

Absolute error is the difference between the real value and its computer representation (in either fixed or floating point). Relative error is the ratio of the absolute error to the represented value.

To demonstrate issues with floating point representations that fixed point can address, let's look at exactly how floating point is represented. A floating point number cannot represent all values exactly. For instance, the closest 32-bit floating point number to value 1.1 is 1.10000002384185791016 (see float.exposed to visualize this). The trailing "imprecision" can lead to errors when performing arithmetic operations. For example, 1.1 + 1.1 yields 2.20000004768371582031.

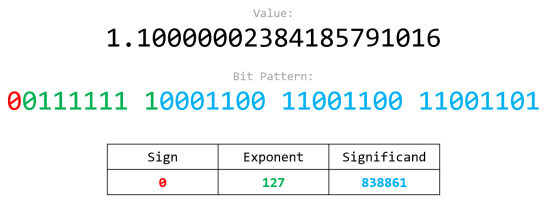

Figure 1: Visualization of 1.1 in floating-point.

In contrast, when using fixed-point representations, an integer is used to store the **exact** value. To represent 1.1 using a fixed-point number with a scale equal to -1, the value 11 is stored. Arithmetic operations are performed on the underlying integer, so adding 1.1 + 1.1 as fixed-point numbers simply adds 11 + 11 yielding 22, representing the value 2.2 exactly.

Why is fixed-point arithmetic important?

As shown in the example preceding, fixed-point arithmetic avoids the precision and rounding errors inherent in floating point numbers while still providing the ability to represent fractional digits. Floating point provides a much larger range of values by keeping relative error constant. However, this means it can suffer from large absolute (truncation) errors when adding very large and very small numbers and run into rounding errors. Fixed-point representation always has the same **absolute** error at the cost of being able to represent a reduced range of values. If you know you need a specific precision after the decimal/binary point, then fixed point allows you to maintain accuracy of those digits without truncation even as the value grows, up to the limits of the range. If you need more range, you have to add more bits. Hence decimal128 becomes important for some users.

Lower Bound	Upper Bound	
decimal32	-21474836.48	21474836.47
decimal64	-92233720368547758.08	92233720368547758.07
decimal128	-1701411834604692317316873037158841057.28	1701411834604692317316873037158841057.27

Table 2: Ranges for decimal32 with scale = -2.

There are many applications and use cases for `fixed_point` numbers. You can find a list of actual applications that use `fixed_point` numbers on Wikipedia

fixed_point in RAPIDS libcudf

Overview

The core of the RAPIDS libcudf `fixed_point` type is a simple class template.

```
template <typename Rep, Radix Rad>
class fixed_point {
  Rep _value;
  scale_type _scale;
}
```

The `fixed_point` class is templated on:

- `Rep`: the representation of the `fixed_point` number (for example, the integer type used)
- `Rad`: the `Radix` of the number (for example, base 2 or base 10)

The `decimal32` and `decimal64` types use `int32_t` and `int64_t` for the Rep, respectively and both have `Radix::BASE_10`. The `scale` is a strongly typed run-time variable (see Run-Time Scale and Strong Typing subsections below, etc).

The `fixed_point` type has several constructors (see Ways to Construct subsection below), explicit conversion operators to cast to both integral and floating point types, and a full complement of operators (addition, subtraction, etc.).

Sign of Scale

In most C++ fixed-point implementations (including RAPIDS libcudf's), a **negative scale** indicates the number of fractional digits. A **positive scale** indicates the

multiple that is representable (for example, if `scale = 2` for a `decimal32`, multiples of 100 can be represented).

```
auto const number_with_pos_scale = decimal32{1.2345, scale_type{-2}}; // 1.23
auto const number_with_neg_scale = decimal32{12345,  scale_type{2}}; // 12300
```

Constructors

The following (simplified) constructors are provided in `libcudf`:

```
fixed_point(T const& value, scale_type const& scale)
fixed_point(scaled_integer<Rep>  s) // already "shifted" value
fixed_point(T const& value)         // scale = 0
fixed_point()                       // scale = 0, value = 0
```

Where `Rep` is a signed integer type and `T` can be either an integral type or floating-point number.

Design and motivation

There are a number of design goals for libcudf's `fixed_point` type. These include:

- Need for a run-time scale
- Consistency with potential standard C++ fixed_point types
- Strong typing

These design motivations are detailed below.

Run-time scale and third-party fixed-point libraries

We studied eight existing fixed-point C++ libraries during the design phase. The primary reason for not using a third-party library is that all of the existing fixed-point types/libraries are designed with the `scale` being a compile-time parameter. This does not work for RAPIDS `libcudf` as it needs scale to be a run-time parameter.

While RAPIDS `libcudf` is a C++ library that can be used in C++ applications, it is also the backend for RAPIDs cuDF, which is a Python library. Python is an interpreted (rather than compiled, like C++) language. Moreover, cuDF must be able to read or receive fixed-point data from other data sources. This means that we do not know the `scale` of the fixed-point values at compile time. Therefore we need to have the `fixed_point` type in RAPIDS `libcudf` that has a run-time scale parameter.

The main library we referenced was CNL, the Compositional Numeric Library by John McFarlane that is currently the reference for an ISO C++ proposal to add fixed-point types to the C++ standard. We aim for the RAPIDS `libcudf fixed_point` type to be as similar as possible to the potentially standardized type. Here's a simple comparison between RAPIDS `libcudf` and CNL.

CNL (Godbolt Link)

```
using namespace cnl;
auto x = fixed_point<int32_t, -2, 10>{1.23};
```

RAPIDS libcudf

```
using namespace numeric;
auto x = fixed_point<int32_t, Radix::BASE_10>{1.23, scale_type{-2}};
```

Or alternatively:

```
using namespace numeric;
auto x = decimal32{1.23, scale_type{-2}};
```

The most important difference to notice here is the `-2` as a template (aka compile-time parameter) in the CNL example versus the `scale_type{-2}` as a run-time parameter in the RAPIDS libcudf example.

Strong typing

Strong typing has been incorporated into the design of the `fixed_point` type. Two examples of this are:

- Using a scoped enumeration with `direct-list-initialization` for scale_type
- Using a scoped enumeration for `Radix` instead of an integer type

RAPIDS libcudf adheres to strong typing best practices and strongly typed APIs because of the protection and expressivity strong typing provides. I won't go into the rabbit hole of weak compared to strong typing, but if you would like to read more about it there are many great resources, including Jonathon Bocarra's Fluent C++ post on How typed C++ is, and why it matters.

Adding support for `decimal128`

RAPIDS `libcudf` 21.12 adds `decimal128` as a supported `fixed_point` type. This required a number of changes, the first being the addition of the `decimal128` type alias that relies on the `__int128` type provided by CUDA 11.5.

```
using decimal32  = fixed_point<int32_t,    Radix::BASE_10>;
using decimal64  = fixed_point<int64_t,    Radix::BASE_10>;
using decimal128 = fixed_point<__int128_t, Radix::BASE_10>;
```

This required a number of internal changes, including updating type traits functions, `__int128_t` specializations for certain functions, and adding support so that `cudf::column_view` and friends work with `decimal128`. The following simple examples demonstrate the use of libcudf APIs with `decimal128` (note, all of these examples work the same for `decimal32` and `decimal64`).

Examples

Simple currency

A simple currency example uses the `decimal32` type provided by `libcudf` with scale -2 to represent exactly $17.29:

```
auto const money = decimal32{17.29, scale_type{-2}};
```

Summing large and small numbers

Fixed point is very useful when summing both large and small values. As a simple toy example, the following piece of code sums the powers of 10 from exponent -2 to 9.

```
template <typename T>
auto sum_powers_of_10() {
    auto iota = std::views::iota(-2, 10);
    return std::transform_reduce(
        iota.begin(), iota.end(),
        T{}, std::plus{},
```

```
        [](auto e) -> T { return std::pow(10, e); });
}
```

Comparing 32-bit floating-point and decimal fixed point shows the following results:

```
sum_powers_of_10<float>();        // 1111111168.000000
sum_powers_of_10<decimal_type>(); // 1111111111.11
```

Where `decimal_type` is a 32-bit base-10 fixed-point type. You can see an example of this using the CNL library on Godbolt here.

Avoiding floating-point rounding issues

An example of where floating-point values run into issues (in C++) is the following piece of code (see in Godbolt):

```
std::cout << std::roundf(256.49999) << '\n'; // prints 257
```

The equivalent code in RAPIDS libcudf will not have the same issue (see on Github):

```
auto col    = // decimal32 column with scale -5 and value 256.49999
auto result = cudf::round(input); // result is 256
```

The value of 256.49999 is not representable with a 32-bit binary float and therefore rounds to 256.5 before the std::roundf function is called. This problem can be avoided with fixed-point representation because 256.49999 is representable with any base-10 (decimal) type that has five or more fractional values of precision.

Binary versus decimal fixed point

```
// Decimal Fixed Point
using decimal32 = fixed_point<int32_t, Radix::BASE_10>;
auto const a    = decimal32{17.29, scale_type{-2}};  // 17.29
auto const b    = decimal32{4.2,   scale_type{ 0}};  // 4
auto const c    = decimal32{1729,  scale_type{ 2}};  // 1700

// Binary Fixed Point
using binary32 = fixed_point<int32_t, Radix::BASE_2>;
auto const x    = binary{17.29, scale_type{-2}};  // 17.25
auto const y    = binary{4.2,   scale_type{ 0}};  // 4
auto const z    = binary{1729,  scale_type{ 2}};  // 1728
```

decimal128

```
// Multiplying two decimal128 numbers
auto const x = decimal128{1.1, scale_type{-1}};
auto const y = decimal128{2.2, scale_type{-1}};
auto const z = x * y;  // 2.42 with scale equal to -2

// Adding two decimal128 numbers
auto const x = decimal128{1.1, scale_type{-1}};
auto const y = decimal128{2.2, scale_type{-1}};
auto const z = x + y;  // 3.3 with scale equal to -1
```

DecimalType in RAPIDS Spark

DecimalType in Apache Spark SQL is a data type that can represent Java BigDecimal values. SQL queries operating on financial data make significant use of the decimal type. Unlike the RAPIDS libcudf implementation of fixed-point decimal numbers, the maximum precision possible for DecimalType in Spark is limited to 38 decimal places. The scale, which is defined as the number of digits after the decimal point, is also

capped at 38. This definition is the negative of the C++ scale. For example, a decimal value like 0.12345 has a scale of 5 in Spark but a scale of -5 in libcudf.

Spark closely follows the Apache Hive specification on precision calculations for operations and provides options to the user to configure precision loss for decimal operations. Spark SQL is aggressive about promoting precision of the result column when performing operations like aggregation, windowing, casting and so on This behavior in and of itself is what makes decimal128 extremely relevant to real-world queries and answers the question: “Why do we need support for high-precision decimal columns?”. Consider the example below, specifically the hash aggregate, which has a multiplication expression involving a decimal64 column, price, and a non-decimal column, quantity. Spark first casts the non-decimal column to an appropriate decimal column. It then determines the result precision, which is greater than the input precision. Therefore, it is quite common for the result precision to be a decimal128 value even if decimal64 inputs are involved.

```
scala> val queryDfGpu = readPar.agg(sum('price*'quantity))
queryDfGpu1: org.apache.spark.sql.DataFrame = [sum((price * quantity)): decimal(32,2)]

scala> queryDfGpu.explain
== Physical Plan ==
*(2) HashAggregate(keys=[],
functions=[sum(CheckOverflow((promote_precision(cast(price#19 as decimal(12,2))) * promote_precision(cas
+- Exchange SinglePartition, true, [id=#429]
   +- *(1) HashAggregate(keys=[],
functions=[partial_sum(CheckOverflow((promote_precision(cast(price#19 as decimal(12,2))) * promote_preci
        +- *(1) ColumnarToRow
        +- FileScan parquet [price#19,quantity#20] Batched: true,DataFilters:
[], Format: Parquet, Location:
InMemoryFileIndex[file:/tmp/data.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<p
```

With the introduction of the new decimal128 data type in libcudf the RAPIDS plug-in for Spark is able to leverage higher precisions and keep computation on the GPU where previously it needed to fall back to the CPU.

As an example, let’s look at a simple query that operates on the following schema.

```
{
    id          :   IntegerType         // Unique ID
    prodName :   StringType     // Product name will be used to aggregate / partition
    price       :   DecimalType(11,2)   // Decimal64
    quantity :   IntegerType    // Quantity of product

}
```

This query computes the unbounded window over totalCost, which is the sum(price*quantity). It then groups the result by the prodName after a sort and returns the minimum totalCost.

```
// Run window operation
val byProdName = Window.partitionBy('prodName)
val queryDfGpu = readPar.withColumn(
    "totalCost",
    sum('price*'quantity) over byProdName).sort(
         "prodName").groupBy(
                 "prodName").min(
                         "totalCost")
```

The RAPIDS Spark plug-in is set up to run operators on the GPU only if all the expressions can be evaluated on the GPU. Let’s first look at the following physical plan for this query without decimal128 support.

Without decimal128 support every operator falls back to the CPU because child expressions that contain a decimal 128 type cannot be supported. Therefore, the

containing exec or parent expression will also not execute on the GPU to avoid inefficient row-to-column and column-to-row conversions.

```
== Physical Plan ==
*(3) HashAggregate(keys=[prodName#18], functions=[min(totalCost#66)])
+- *(3) HashAggregate(keys=[prodName#18],
functions=[partial_min(totalCost#66)])
   +- *(3) Project [prodName#18, totalCost#66]
      +- Window [sum(_w0#67) windowspecdefinition(prodName#18,
specifiedwindowframe(RowFrame, unboundedpreceding$(), unboundedfollowing$()))
AS totalCost#66], [prodName#18]
        +- *(2) Sort [prodName#18 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(prodName#18, 1), true, [id=#169]
               +- *(1) Project [prodName#18,
CheckOverflow((promote_precision(cast(price#19 as decimal(12,2))) *
promote_precision(cast(cast(quantity#20 as decimal(10,0)) as
decimal(12,2)))), DecimalType(22,2), true) AS _w0#67]
               +- *(1) ColumnarToRow
               +- FileScan parquet [prodName#18,price#19,quantity#20]
Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex[file:/tmp/data.parquet], Pa
```

The query plan after enabling decimal128 support shows that all the operations can now run on the GPU. The absence of ColumnarToRow and RowToColumnar transitions (which show up for the collect operation in the query) enables better performance by running the entire query on the GPU.

```
== Physical Plan ==
GpuColumnarToRow false
+- GpuHashAggregate(keys=[prodName#18], functions=[gpumin(totalCost#31)]),
filters=ArrayBuffer(None))
   +- GpuHashAggregate(keys=[prodName#18],
functions=[partial_gpumin(totalCost#31)]), filters=ArrayBuffer(None))
        +- GpuProject [prodName#18, totalCost#31]
        +- GpuWindow [prodName#18, _w0#32, gpusum(_w0#32, DecimalType(32,2)) gpuwindowspecdefinition(pro
            +- GpuCoalesceBatches batchedbykey(prodName#18 ASC NULLS FIRST)
            +- GpuSort [prodName#18 ASC NULLS FIRST], false, com.nvidia.spark.rapids.OutOfCoreSort$@3204
               +- GpuShuffleCoalesce 2147483647
               +- GpuColumnarExchange gpuhashpartitioning(prodName#18, 1),
 true, [id=#57]
                     +- GpuProject [prodName#18,
gpucheckoverflow((gpupromoteprecision(cast(price#19 as decimal(12,2))) * gpupromoteprecision(cast(cast(q
decimal(12,2)))), DecimalType(22,2), true) AS _w0#32]
                     +- GpuFileGpuScan parquet
[prodName#18,price#19,quantity#20] Batched: true, DataFilters: [], Format:
Parquet, Location: InMemoryFileIndex[file:/tmp/data.parquet],
PartitionFilters: [], PushedFilters: [], ReadSchema: struct<prodName:string,price:decimal(11,2),quantity
```

For the multiplication operation, the quantity column is cast to decimal64 (precision = 10) and the price column, which is already of type decimal64 is casted up to precision of 12 making both columns of the same type. The result column is sized to a precision of 22, which is of type decimal128 since the precision is greater than 18. This is shown in the GpuProject node of the plan above.

The window operation over the sum() also promotes the precision further to 32.

We use NVIDIA Decision Support (NDS), an adaptation of the TPC-DS data science benchmark often used by Spark customers and providers, to measure speedup. NDS consists of the same 100+ SQL queries as the industry standard benchmark but has modified parts for dataset generation and execution scripts. Results from NDS are not comparable to TPC-DS.

Preliminary runs of a subset of NDS queries demonstrate significant performance improvement due to decimal128 support, as shown in the following graph. These were run on a cluster of eight nodes each with one A100 GPU and 1024 CPU cores, running executors with 16 cores on spark 3.1.1. Each executor uses 240GiB in memory. The

queries show excellent speedup of nearly 8x, which can be attributed to operations that were previously falling back to the CPU now running on the GPU, thereby avoiding row-to-column and column-to-row transitions and other associated overheads. On average the end-to-end run time of all the NDS queries shows 2x improvement. This is (hopefully) just the beginning!

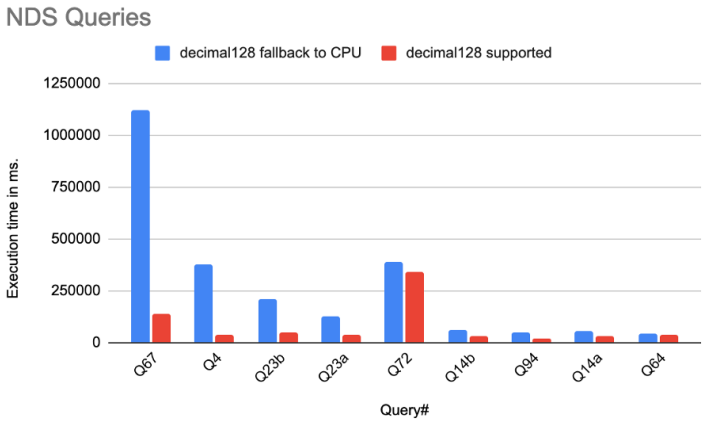

Figure 2: Performance evaluation of a subset of NDS queries.

With the 21.12 release of the RAPIDS plug-in for Spark, decimal128 support is available for the majority of operators. Some special handling of overflow conditions to maintain result compatibility between CPU and GPU is necessary. The ultimate goal of this effort is to allow retail and financial queries to fully benefit from GPU acceleration through the RAPIDS for Spark Plugin.

Summary

fixed_point types in RAPIDS libcudf, the addition of DecimalType, and decimal128 support for the RAPIDS plug-in for Spark enable exciting use cases that were previously only possible on the CPU now to be run on the GPU. If you want to get started with RAPIDS libcudf or the RAPIDS plug-in for Spark, you can follow the links below: