UltraPrecise: A GPU-Based Framework for Arbitrary-Precision Arithmetic in Database Systems

Xin Li
Shandong University
Qingdao, China
li.xin@mail.sdu.edu.cn

Mengbai Xiao*
Shandong University
Qingdao, China
xiaomb@sdu.edu.cn

Dongxiao Yu
Shandong University
Qingdao, China
dxyu@sdu.edu.cn

Rubao Lee
Freelance
Columbus, OH, USA
lee.rubao@ieee.org

Xiaodong Zhang
The Ohio State University
Columbus, OH, USA
zhang@cse.ohio-state.edu

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00294

***Abstract*—Fixed-point decimal operations in databases with arbitrary-precision arithmetic refer to the ability to store and operate decimal fraction numbers with an arbitrary length of digits. This type of operation has become a requirement for many applications, including scientific databases, financial data processing, geometric data processing, and cryptography. However, the state-of-the-art fixed-point decimal technology either provides high performance for low-precision operations or supports arbitrary-precision arithmetic operations at low performance. In this paper, we present a design and implementation of a framework called UltraPrecise which supports arbitrary-precision arithmetic for databases on GPU, aiming to gain high performance for arbitrary-precision arithmetic operations. We build our framework based on the just-in-time compilation technique and optimize its performance via data representation design, PTX acceleration, and expression scheduling. UltraPrecise achieves comparable performance to other high-performance databases for low-precision arithmetic operations. For high-precision, we show that UltraPrecise consistently outperforms existing databases by two orders of magnitude, including workloads of RSA encryption and trigonometric function approximation.**

***Index Terms*—database, parallel computing, fixed-point arithmetic, GPU**

I. Introduction

Database systems have long been evolving towards a general and powerful infrastructure for highly complex data analytics tasks [1]–[5]. The widely adopted 32-/64-bit floating-point format following IEEE 754-1985 [6] does not satisfy the requirement of sufficiently accurate or even exact data analytics applications in two ways [7]. First, many simple but critical decimal fractions, such as 0.1, cannot be exactly represented, unsatisfying the basic requirement of preserving the exactness in banking, stock, and many other financing systems. Second, the precision of a 32-/64-bit floating-point number is still limited for many data analytics applications. To completely address the two above-mentioned issues, a fixed-point data type `DECIMAL` has been introduced to represent numbers with decimal fractions. Several major database systems choose to support this data type at arbitrary or nearly arbitrary-precision [8]–[12], adapting themselves to a wide range of applications besides financial systems, such as geographic information systems, scientific computing, geometric computation, and cryptography. For example, evaluating orthogonal

*Mengbai Xiao is the corresponding author.

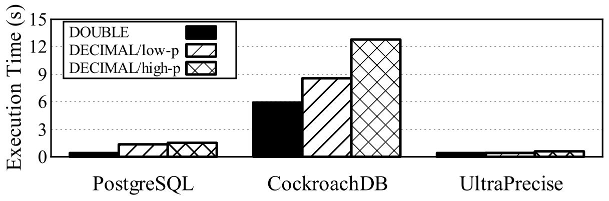

Fig. 1. Execute `SELECT SUM(c1+c2) FROM R` in PostgreSQL, CockroachDB, and UltraPrecise. `R` has 10 million tuples. `c1` and `c2` are set to (1) both `DOUBLE`, (2) `DECIMAL`(17, 5) and `DECIMAL`(14, 2), namely *low-p*, and (3) `DECIMAL`(35, 5) and `DECIMAL`(32, 2), namely *high-p*.

polynomials in statistical analysis requires precision in calculation 4∼5× higher than the standard double-type [13], [14]. In gradient-domain processing, a prodigiously high precision of up to 20,000 digits is necessary for solving a Poisson equation [15], [16]. The arbitrary-precision decimal arithmetic also helps to address challenges in macroeconomic analysis [17]. However, the accuracy of decimal fractions in data analytics gained by fixed-point `DECIMAL` processing is at a cost of high data- and computing-intensive operations.

In databases, `DECIMAL` is associated with p, the *precision*, and s, the *scale*. They are defined by the digital length of the whole decimal number and the part after the decimal point, respectively. The representation scope of `DECIMAL`(p, s) is determined by the underlying word length. A 32-bit word stores a decimal number at the highest precision of 9, and a 64-bit word only raises that to 19. Thus, `DECIMAL` must be extended to a multi-word representation for a higher precision. However, the arithmetic operations of a multi-word representation have to be implemented by software other than by hardware, incurring non-trivial overhead because code that properly handles carries, overflows, etc., should be added. In PostgreSQL, more than 10K lines of C code are added to realize the arbitrary-precision arithmetic of `DECIMAL` [18].

To quantitatively understand the performance and precision issues of multi-word arithmetic operations, we execute a SQL query using both the `DECIMAL` and `DOUBLE` data types on PostgreSQL and CockroachDB, both of which declare to support unlimited precision `DECIMAL` operations. The results, shown in Figure 1, demonstrate that the execution of data type

DOUBLE is very fast but produces incorrect results. Furthermore, we found that the DOUBLE execution results from the two databases are inconsistent. Using the DECIMAL data type, the results are correct and consistent, but the execution time is 3.00× (PostgreSQL) and 1.45× (CockroachDB) slower. Increasing the precision further prolongs the execution time.

The performance of extending fixed-point decimals to high precision is poor. Although PostgreSQL completes the job in Figure 1, its long execution time is unacceptable for many applications. To address the trade-off between high-performance fixed-point operations with low precision and low-performance fixed-point operations with arbitrary-precision, we have explored the opportunity of amortizing the computation cost via massive parallelism. In database systems, the same expression is evaluated for every tuple without dependency on each other. This indicates the arithmetic operations of DECIMAL are highly regular and friendly to the parallel architecture. Even for aggregations, a unique opportunity of parallel processing is still available for the acceleration of the fixed-point operations involved. In this paper, we design and implement a powerful framework called UltraPrecise for fixed-point decimals based on arbitrary-precision arithmetic in GPU databases. Figure 1 shows that in UltraPrecise, executing a query with DECIMAL at low-precision is only 1.04× slower than that with DOUBLE.

The arithmetic operations of DECIMAL appear in both tuple-based expression evaluation and column-based aggregation. For evaluating an expression having DECIMAL, a just-in-time (JIT) compilation engine is installed to generate GPU kernels. Without JIT techniques, the evaluation code is either inefficient or too cumbersome. Our framework accelerates fixed-point arithmetic by optimizing the data representation, arithmetic operators, and expression scheduling. We design both compact and non-compact decimal representation that efficiently stores data in memory and expands only in computation. We also exploit PTX instructions to optimize the code processing carries and to speed up time-consuming divisions. Optimization techniques are also developed at the expression level. The costly alignment operations in additions and subtractions are reduced by rewriting the expression. Additionally, evaluating the portion of an expression with only constants and converting the constants to the DECIMAL type are migrated to the compilation stage, eliminating redundant computation at runtime. We also implement the multi-threading expression evaluation and aggregation, which amortizes the resources of each calculation instance among threads. We implement UltraPrecise in RateupDB, a high-performance GPU database [19], and carry out extensive experiments to justify our designs. When running TPC-H Q1, UltraPrecise achieves comparable performance as that of high-performance databases functional only in this precision range, like HEAVY.AI, RateupDB, and MonetDB. As the precision increases, UltraPrecise outperforms PostgreSQL by up to 41.28×. In more computation-intensive workloads such as Rivest-Shamir-Adleman (RSA) encryption and trigonometric function approximation, similar performance trends are observed, where UltraPrecise outperforms PostgreSQL, H2, and CockroachDB, all featuring arbitrary-precision arithmetic of DECIMAL, by two orders of magnitude. Our contributions are summarized as follows:

- We present a GPU computation framework that supports arbitrary-precision arithmetic operations. To the best of our knowledge, UltraPrecise is the only GPU database realizing DECIMAL at arbitrary-precision.
- We design optimization methods from data representation to expression scheduling that accelerates DECIMAL arithmetic at arbitrary-precision, which have not been considered to enhance the arithmetic precision and performance in existing database systems.
- We implement and evaluate UltraPrecise in a real-world database with extensive experiments. The results justify the generality of our design, and UltraPrecise is the fastest among databases supporting arbitrary-precision arithmetic.

We present a preliminary introduction in Section II. Section III shows the design of UltraPrecise and Section IV extensively evaluates our framework. Section V discusses related work and Section VI concludes our work.

II. Preliminary

A. Number Representations

Binary format *vs.* **decimal format**. A number n is defined as

$$n = \pm i \times b^s, \tag{1}$$

where i is a positive integer, b is the base, and s is the scale. Numbers in computer systems are commonly represented in the *binary format* or the *decimal format*, i.e., set b to 2 or 10. A bit string represents the same number if s is 0, e.g., 11_2 is 3 for both formats. But if s is set to -1, 11_2 means 1.5 in the binary format and 1.1 in the decimal format. If we want to represent digits at the left side of the radix point, two formats make no difference because they are integers that can be represented by the same i with $s = 0$. For the digits on the other side, the binary format fails to represent certain numbers like 0.1 that could be exactly represented in the decimal format, while the decimal format could represent all numbers in the binary format. The decimal format still cannot exactly represent some numbers like 1/3, but it is practical enough since it is compatible with the numeric system in human reality.

Fixed-point *vs.* **floating-point**. If the scale s in Equation 1 is stored with the number and can be varied, the representation is called *floating-point*. A representation is *fixed-point* if s is determined at the compilation time. In database systems, DECIMAL is a fixed-point representation so that only integers are stored as the data, and the scale is the column property. The fixed-point arithmetic between the operands with the same scale is essentially integer arithmetic. With different scales, the arithmetic operations become complicated and details are discussed in Section II-B.

Arbitrary-precision *vs.* **limited-precision**. The modern arithmetic logic units (ALUs) can only support arithmetic operations at *limited-precision*, which is commonly 32-/64-/128-

bit length. The *arbitrary-precision* arithmetic is realized by software and the practical limit should only be imposed by the available memory. For general-purpose computation, we expect not to put an upper bound on the precision p of DECIMAL (p, s) though, in practice, a couple of databases only support small p values for high performance.

B. Fixed-point Arithmetic Operations

In this work, we are more interested in designing an efficient framework for fixed-point arithmetic other than delving into specific arithmetic algorithms. In this section, we briefly introduce the basic arithmetic operations of fixed-point numbers.

Additions/subtractions. The scales of two fixed-point operands must be aligned before the addition or subtraction. For example, 1.23 in DECIMAL(4, 2) is represented as an integer 123 in the memory while 0.1 in DECIMAL(3, 1) is stored as an integer 1. Directly adding them leads to a wrong result so we need to convert 0.1 with a scale of 1 to 0.10 with a scale of 2, which is represented by an integer of 10. In the decimal format, aligning a scale of s_1 to s_2 is realized by $\times 10^{s_2 - s_1}$. If $s_1 > s_2$, the alignment operation is realized as $\div 10^{s_1 - s_2}$. The division operation not only incurs higher computation overhead but also lowers the intermediate precision. Thus, aligning a smaller scale with a larger one is always preferred. Moreover, positives and negatives both exist in the database. In a function implementing the addition, the signs of operands determine whether two numbers are added or one number is subtracted from the other. Numbers are compared before the subtraction to decide the minuend and the subtrahend. For addition and subtraction, numbers are calculated from the least significant word to the most significant word, where carries should be properly handled. For the comparison, numbers are compared from the most significant word to the least significant one. The result is derived once two words differ.

Multiplications. Multiplying two DECIMAL numbers is essentially multiplying two multi-word integers, and the basic algorithm is the *elementary school algorithm*. For two operands that each have N words, their product is $2N$ words. The k-th word of the product is derived by multiplying the words from two operands: if the i-th word from the first operand and the j-th word from the second satisfies $i + j = k$, their product is added to the k-th word of the result, and all such pairs are calculated. The carry-out of the accumulation is added to the (k+*1*)-th word of the result. The *Karatsuba algorithm* [20] is an advanced multiplication algorithm with a lower complexity of $O(N^{\log_2 3})$ than that of the elementary school algorithm, which is $O(N^2)$. But in practice, the Karatsuba algorithm is not as fast as the basic one for a small N. The *Schönhage-Strassen algorithm* [21] has even lower complexity than the Karatsuba algorithm, but it outperforms the latter only if N is sufficiently large.

Divisions. The division operation of DECIMAL numbers is converted to a sequence of multiplications and additions for high performance. The basic division algorithm is similar to the *long division* method taught in elementary school.

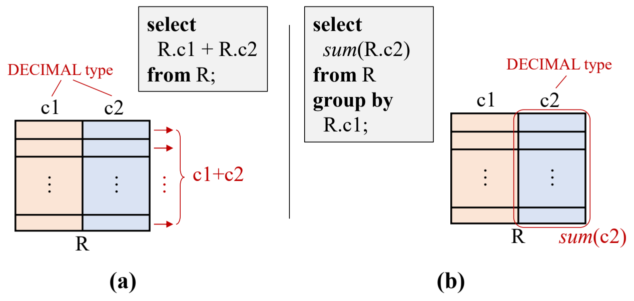

Fig. 2. The fixed-point arithmetics in the query execution. (a) Adding DECIMALs in tuples. (b) Aggregating DECIMALs in columns.

The algorithmic steps are as follows: 1) Derive the quotient range. If the digit length of the divisor is l, the quotient range is then $(D/10^l, D/10^{l-1}]$, where D is the dividend. 2) Find the exact quotient. The quotient is the number that times the divisor equals the dividend and could be located with a binary search. An advanced algorithm is the *Newton-Raphson algorithm* [22], where the quotient is calculated by $D \times d^{-1}$, where d is the divisor. d^{-1} could be approximated by iterations of multiplication, where $d_{i+1}^{-1} = d_i^{-1}(2 - d \cdot d_i^{-1})$. The *Goldschmidt algorithm* [23] calculates the quotient according to $\frac{D}{d} \cdot \frac{F_1}{F_1} \cdot \frac{F_2}{F_2} \cdots$. Once the divisor approximates 1, the dividend approximates the quotient.

III. Framework Design

A. Overview

The fixed-point arithmetic happens in two places when executing a SQL query: evaluating an expression and aggregation. This indicates that the computation occurs in tuples or columns. Two illustrative cases are presented in Figure 2.

We implement UltraPrecise in RateupDB [19], a high-performance GPU database system. Figure 3 shows how the expression evaluation is embedded in the database query execution pipeline. A SQL query with expressions having DECIMAL is parsed into a logical plan tree, where the expressions are associated with relational operators and are parsed into expression trees. The logical plan tree is turned into a physical plan tree when the expression trees are also optimized. The optimized expression trees are evaluated in a just-in-time (JIT) compilation engine, where GPU kernels are generated and compiled. In the end, the executor executes the query plan tree in a bottom-up manner, during which the GPU kernels are launched for evaluating the expressions.

The DECIMAL values are aggregated in rounds for exploiting massive parallelism on GPU. The arithmetic operations are realized in functions instead of instructions. For the tuples grouped according to DECIMAL columns before aggregation, we implement the comparison operators of DECIMAL that help sort the numbers to achieve grouping.

Simply parallelizing the expression evaluation and aggregation is still slow. To significantly improve the performance of our framework, we have made efforts to effectively optimize data representation, fine-tuned operator implementations, and a

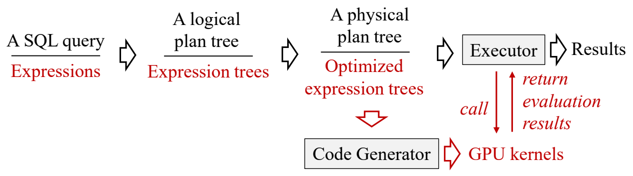

Fig. 3. Evaluating expressions with DECIMAL in a database system

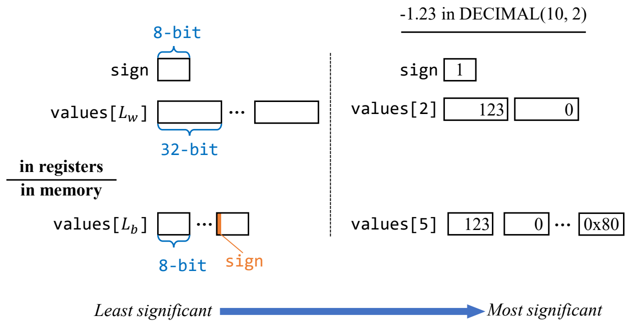

Fig. 4. Fixed-point representations vary in registers and memory. The left side shows the general representations and the right side shows how -1.23 in DECIMAL(10, 2) is stored.

few techniques manipulating expressions. To further accelerate the computation, we have also applied multi-threading for an instance of expression and aggregation.

B. *JIT-based Representation*

If the code of DECIMAL representations and arithmetic operations is pre-implemented, there are three options: 1) A large enough space is always defined for a decimal so that any length of numbers could be contained, which is not practical; 2) We allocate memory at runtime according to the input data, but the memory management cost could easily surpass the time of calculation in simple expressions; 3) We compile the database with DECIMAL representation templates. However, this quickly inflates the code size because it has to instantiate not only the representations with all possible data lengths but also arithmetic operators having arbitrary length combinations.

To make our arithmetic framework general and efficient, the DECIMAL representations are generated by the JIT engine. A number in DECIMAL(p, s) is represented as an integer stored in an array of 32-bit words, and its sign is held in a byte. For example, 1.23 is stored as 123. The precision and scale are contained in the metadata of the relation because values in a data column have the same DECIMAL definition. We can directly use them as constants when generating the GPU kernel that carries out the fixed-point arithmetic operations. The length of the value array L_w is also determined in code generation, which follows

$$L_w = \left\lceil \frac{p \cdot \log_2 10}{32} \right\rceil .$$

The L_w values of varying p are pre-computed and stored in a key-value table. The top of Figure 4 illustrates our representation and gives an example of representing -1.23 in DECIMAL(10, 2), which takes 9 bytes in total.

The value array is word-aligned because this is efficient in the calculation. For example, the PTX instruction addc that adds two numbers with the carry-in operates on 32-bit operands at least. However, storing DECIMAL in a word-aligned manner consumes extra space in the memory and disk. So the fixed-point decimals are stored in more compact byte-aligned arrays before being read to the processors. We keep the sign in the byte array using 1 bit, and thus the array length is L_b bytes calculated by $\left\lceil \frac{1+p\log_2 10}{8} \right\rceil$. In this way, less space is allocated in memory for the decimals, and it also speeds up the calculation if the workload is dominated by memory accesses. The bottom of Figure 4 is the representation in memory and gives an example as well, which stores -1.23 in 5 bytes.

1) An alternative representation: A fixed-point number could also be represented in an array where the decimal point only appears between the array elements, which is the design adopted by PostgreSQL [18] and RateupDB [19]. For example, if we want to represent 1.23 in a word-aligned array, two words must be allocated. One word stores 1, and the other word represents 0.23 (it stores 230,000,000 in the 32-bit word). To realize this representation, a 32-bit word to the right of the decimal point is only allowed to represent 10^9 numbers.

The advantage of this design is that it cancels the alignment operation between two DECIMAL numbers with different scales in additions and subtractions. We show an example of the alternative representation in Figure 5. Though this representation saves computation, it also introduces extra storage costs. Especially when the precision is low, double space is required. We eventually discard this design because we notice that compared to the align operations, reading data from the memory dominates the execution time of additions and subtractions. A compact representation benefits the calculation.

2) A Code Example: The expression evaluation follows three steps: 1) Read the compact decimals, and expand them to non-compact ones; 2) Evaluate the expression; And, 3) write the results back to the memory in the compact format. We present an example in Listing 1. The generated code evaluates an expression of DECIMAL(4, 2) + DECIMAL(4, 1).

```
__global__ void calc_expr_1(ColIter *input,
                            int tupleNum, char *output) {
  int stride = blockDim.x * gridDim.x;
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  for(int i = tid; i < tupleNum; i += stride) {
    Decimal<1> c1_4_2((cDecimal*)(input[0][i]), 2);
    Decimal<1> c2_4_1((cDecimal*)(input[1][i]), 2);
    Decimal<1> result = (c1_4_2 + c2_4_1 << 1);
    result.toCompact(output + i * (size_t)3, 3);
  }
}
```

Listing 1. A code example is generated for evaluating DECIMAL(4, 2) + DECIMAL(4, 1). input is the iterator array of accessing columns, where input[0] points to the DECIMAL(4, 2) column and input[1] points to the other. cDecimal is the compact representation of decimals which is a byte array. Decimal<N> is word-aligned and N is L_w. toCompact converts a decimal to a compact representation.

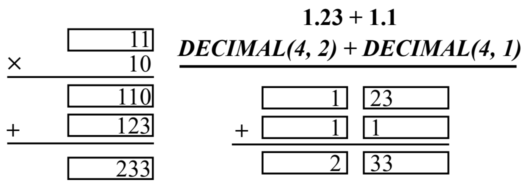

Fig. 5. Different representations change the realization of fixed-point arithmetic operations. Putting the decimal point between array elements introduces additional storage cost but cancels alignment in additions and subtractions.

When generating the GPU kernel, we calculate the lengths of the word array and the byte array. As the precision is 4, L_w is thus 1, and L_b is 2. We use C++ templates to help generate code for decimals with varying precisions. In the code templates, + and `<<n` are overloaded to implement adding two `DECIMAL` numbers and multiplying one value by 10^n, respectively. In this example, two numbers have different scales, so an alignment operation is required. To avoid potential overflows in the computation, we expand the precision of the results to 6, which leads to L_w of still 1 and L_b of 3.

3) Intermediate Precision: Using the JIT engine to evaluate expressions has the advantage of determining the precision of intermediate results and designating their size at compile time. This eliminates the need to set the size of the data container to a large enough value or allocate them at runtime to avoid overflow. We can infer the precision of all intermediate nodes in a bottom-up manner from an expression tree parsed. Different binary operators have different rules for the precisions and scales of two operands. For addition or subtraction, the result is defined as `DECIMAL`$(\max(p_1, p_2 + s_1 - s_2) + 1,\ s_1)$ if $s_1 \geq s_2$. For multiplication, the result is $(p_1 + p_2,\ s_1 + s_2)$.

For division, if the dividend is $(p_1,\ s_1)$ and the divisor is $(p_2,\ s_2)$, the result is guaranteed to have the scale of $s_1 + 4$ in our framework. To achieve this, we multiply the dividend by 10^{s_2+4} before the division. We also need to determine the precision of the quotient. Since the integer part of the quotient has a maximum length of $(p_1-s_1)-(p_2-s_2)+1$, we can safely define the quotient as `DECIMAL`$(p_1-p_2+s_2+5,\ s_1+4)$ without causing the overflow. For the modulo operator, the precision of the result is equal to p_2, and the scale is 0 because only the integer modulo is supported.

We also need to determine the precision of the results for the aggregation operators. For `MAX` and `MIN`, the precision of the result is equal to the precision of the expression being aggregated. For `SUM`, we define the result precision as $p+\lceil \log_{10} N \rceil$, where N is the number of tuples. For `AVG`, we determine the result precision by following the rules of `SUM` and division, where the divisor is converted to `DECIMAL`$(\lfloor \log_{10} N \rfloor + 1,\ 0)$.

C. Optimizations in Operators

Parallel thread execution (PTX) instructions of NVIDIA GPUs are assembly-like, and they could be embedded in the kernel code to gain performance comparable to native devices. We proficiently utilize these instructions to accelerate the arithmetic operations of `DECIMAL` in our framework.

1) Carry-in and carry-out: We have to handle carry-in and carry-out when performing multi-word arithmetic operations. Additions and subtractions have the instructions `addc` and `subc`. Listing 2 shows an example of using `addc` to add two decimals. `madc` is also tested for multiplications, but it is slower than the code implemented by CUDA only.

```
asm volatile ("add.cc.u32 %0, %1, %2;" :
    "=r"(v[0]) : "r"(other.v[0]), "r"(v[0]));
#pragma unroll
for(int i = 1; i < N; i++)
  asm volatile ("addc.cc.u32 %0, %1, %2;" :
   "=r"(v[i]) : "r"(other.v[i]), "r"(v[i]));
```

Listing 2. The code appears in the function overloading operator "+". The instruction `add.cc.u32` adds two operands in unsigned 32-bit type and sets a single flag bit if the result has carry-out. `addc.cc.u32` additionally adds the carry-in set in the previous instruction. `v[N]` is the array storing `DECIMAL` values.

2) Divisions: The division is the most computationally intensive arithmetic operator. To calculate $a \div b$, we follow a straightforward algorithm: 1) Understand the quotient range by comparing the dividend and the divisor, which is derived from the most significant position of 1 in a and b. For example, if a is $1xxxxx_2$ and b is $1xxx_2$, the quotient must be in the range from 10_2 to 111_2. Locating the most significant 1 of operands is accelerated by a PTX instruction `bfind`; 2) Use the binary search to test numbers in the quotient range until a number multiplies the divisor to equal the dividend.

We test two cases to circumvent the costly division algorithm. First, if the dividend and divisor could be contained in a 64-bit word, respectively, the instruction `div` is used to accomplish the division directly. The test is after the precision expansion of the dividend. Second, if the divisor is only a 32-bit word, we divide the dividend from the most significant word to the least with the `div` instruction.

D. Optimizations in Expressions

1) Alignment scheduling: As mentioned above, two `DECIMAL` values with different scales must be aligned before additions and subtractions. This requires more cycles than the `add` or `sub` instructions because a multiplication operation is introduced. In an extreme case, we have to align decimals in every step of evaluating an expression. For example, in `DECIMAL`(4, 1) + `DECIMAL`(4, 2) + `DECIMAL`(4, 1), the first decimal must be aligned to the second one, resulting in a sum with a scale of 2. Another alignment is required because the scale of the third decimal is 1.

Before evaluation, we could reduce the alignment operations by rewriting the expression. If we add two `DECIMAL`(4, 1) numbers first in the previous example, only 1 alignment is required in the evaluation. To achieve this, we rewrite the expression with the following steps: 1) An arbitrary expression is first parsed into a binary expression tree, where the intermediate nodes are operators and the leaf nodes are operands; 2) The subtractions are changed to additions, where

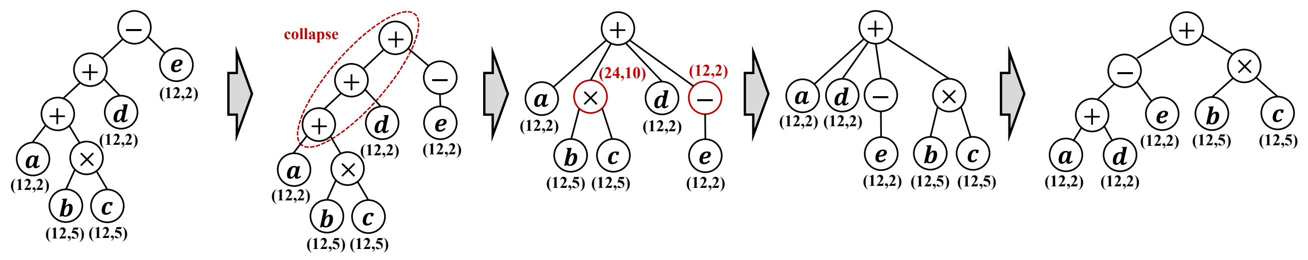

Fig. 6. An expression of $a + b \times c + d - e$ is scheduled to minimize the alignments from 3 to 1. The precision and scale of operands are shown below the leaf nodes. When sorting the operands of addition, the scale of an operator is determined by its type. "$\times$" sums the scale of its operands and the unary negation "$-$" inherits the scale.

the subtrahend is converted into a two-level subtree with the unary negation operator as its root; 3) The binary expression tree is converted into an n-ary tree by collapsing the addition operators at neighboring levels; 4) The order of leaf nodes at the same level is scheduled according to their scales; 5) The n-ary tree is converted back to a binary tree for code generation. Figure 6 shows an example of alignment scheduling.

2) Constants in expressions: Constants, integers, and floating-point numbers involved in fixed-point arithmetic need to be converted to `DECIMAL` first. In the expression evaluation, the constant conversion has to repeat itself for each tuple. To eliminate the computation redundancy, we migrate the constant conversion to the compilation from runtime execution. Furthermore, a portion of an expression with only constants is evaluated before the code generation.

Specifically, we optimize the constant arithmetic after the expression is transformed into an n-ary tree. Before the alignment scheduling, we order the nodes at the same level according to their types, where the constant nodes are put together. Then, it is trivial to calculate the arithmetic results of the constant nodes. Once the calculation is done, i.e., at most one constant remains at each level, we iteratively search if any calculation shortcuts are in the expression tree. A shortcut is a subtree that could be evaluated immediately, e.g., $+a$, $0 + a$, and $1 \times a$. After all, the remaining constants are converted to `DECIMAL` based on their value. For example, 1.23 is `DECIMAL`(3, 2) and 10 is `DECIMAL`(2, 0). The `DECIMAL` constants are scheduled with their scales. A constant is aligned to the minimum of the nodes having a greater or equal scale. We present an example in Figure 7.

E. Multi-threading Arithmetic

1) Expression evaluation: In tuple-based expression evaluation, it is natural to assign one tuple to one thread. The registers required and the memory access cost per thread increase as the precision expands and the expression becomes more complex. To manage the processor resources at a reasonable scale and hide the memory access latency, the arithmetic operations could also be accomplished in multi-threading.

We implement the multi-threading arithmetic operations based on Cooperative Groups Big Numbers (CGBN) [24], [25], a GPU library that realizes multiple precision arithmetic of unsigned integers. We extend the CGBN library to support our `DECIMAL` representations and signed operands. A group of threads is arranged to evaluate an instance of expression, where the group size is *threads per instance* (TPI).

The `DECIMAL` values are loaded into a thread group collaboratively. When generating the GPU kernel, we need to determine how the operands in the compact representation are read. Since processing carries between threads incurs inter-thread communication, when the word length of a decimal exceeds TPI, we direct a thread to read neighboring data to minimize this overhead. For each thread in the group, the words it reads are calculated as $l_t = \lceil \frac{L_b}{4 \cdot \mathrm{TPI}} \rceil$. Since the compact representation is not word-aligned, a trailing thread in the group reads data less than or equal to l_t.

```
int g_tid = threadIdx.x & 3; // TPI − 1 = 3
int tid = (blockIdx.x * blockDim.x + threadIdx.x) / 4;
if(tid >= tupleNum) return;

uint32_t v[2]; // l_t = 2
if(g_tid < 3) // L_b/(l_t · 4) = 3
  memcopy(v, input[0][tid] + g_tid * 8, 8);// l_t · 4 = 8
// No following branch if L_b/(l_t · 4) == TPI.
else if(g_tid == 3)
  memcopy(v, input[0][tid] + g_tid * 8, 3);// L_b%(l_t · 4) = 3
```

Listing 3. The code is generated to load a number in `DECIMAL`(64, 32) to a group of threads. The number is stored in an array of 27 bytes, i.e., $L_b = 27$. As TPI is set to 4, each thread except the last one loads $l_t = 2$ words. The branch code is not generated if the compact representation is aligned to TPI.

Listing 3 presents an example of loading a `DECIMAL` value into a group of threads. Depending on the arithmetic types, the loaded data are directly involved in computation (additions/subtractions) or are broadcast to other threads in the group (multiplications/divisions), piecing up the complete results. After evaluation, the results are also collaboratively written to memory by the thread group.

Additionally, we enhance the CGBN library to support full-fledged `DECIMAL` operations. For additions, the signs of operands are shared among group threads, and the actual calculation is converted to subtraction according to the signs, where the alignment and comparison operations are also added. For multiplications and divisions, the processing of signs, precisions, and scales are added to the implementations as discussed in Section III-B3. To the end, we extend CGBN to support the arithmetic of arbitrary-precision `DECIMAL` and be compatible with UltraPrecise.

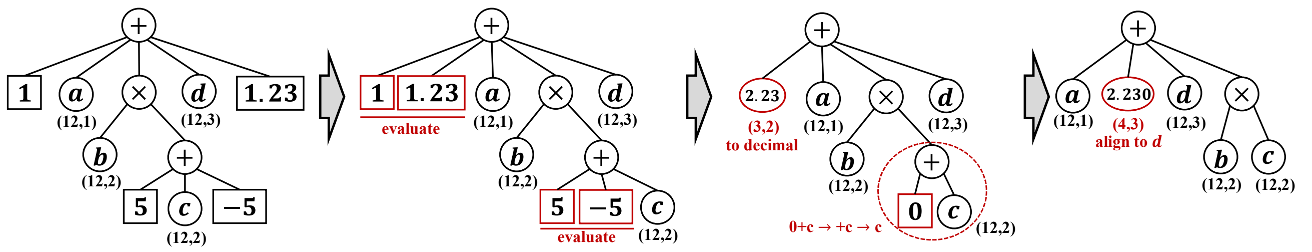

Fig. 7. The constants in an expression of $1 + a + b \times (5 + c - 5) + d + 1.23$ are optimized. The constant arithmetics are processed at each level individually. The subtree representing $0 + c$ is found a shortcut and could be optimized to a single node c. After the alignment scheduling, 2.23 in `DECIMAL`(3, 2) is scaled to `DECIMAL`(4, 3), which saves redundant computation in runtime.

2) Multi-threading Aggregation: We implement multi-threading aggregation operators using *moderngpu* [26]. In these operators, we replace the underlying comparison, addition, and division with multi-threading versions, and adjust how the workload is arranged accordingly. The aggregation is accomplished in several passes. For each pass, the `DECIMAL` values are arranged into several thread blocks, with each thread block calculating an aggregation result. The aggregation results are collected together for the next pass until we can process all the data in one thread block.

To fully exploit the computational capacity of streaming multiprocessors (SMs), the decimals to be aggregated must be properly arranged. Specifically, to aggregate n `DECIMAL` values, each expanded to L_w words in the calculation, we determine n_T, the number of values processed in a thread block, and n_t, the number of values processed in a thread. In a thread block, the `DECIMAL` values are first read into the shared memory and then aggregated. The aggregation is first carried out inner-thread and then inter-thread. Thus, n_T and n_t are determined by T_{max}, the maximum number of threads that can be launched in a thread block, and S, the size of shared memory. Since TPI threads are grouped to calculate an arithmetic instance, the number of thread groups in a thread block is $N_g = T_{max}/\text{TPI}$. Due to the shared memory limitation, we derive that $n_t = \left\lfloor \frac{S}{N_g(4L_w+1)} \right\rfloor$. We thus have $n_T = n_t \cdot N_g$ and need to launch $\lceil N/n_T \rceil$ thread blocks.

IV. Evaluation

Hardware configuration. All experiments are performed on a server equipped with high-performance hardware. The server has two CPUs of Intel Xeon Gold 6130H with a frequency of 2.10 GHz, featuring 16 cores and a 22MB Last-Level Cache. The GPU is an NVIDIA Quadro RTX A6000, which has 48 GB GDDR6 memory and is connected through a PCIe 4.0 bus. The machine is equipped with 128 GB DDR4 DRAM and has mirrored 1TB SSDs. The server runs on Ubuntu 20.04 LTS operating system and the experiments were conducted using CUDA Toolkit 11.6.

Workloads. To comprehensively evaluate the system performance, we vary the precision of decimals used in the experiments. This changes the size of the array used to hold decimal data in both compact and non-compact representations. If not specified, we fix the precision of evaluation results of expressions to 18/38/76/153/307, which means 2/4/8/16/32 words are used to store the results. So, the precisions of `DECIMAL` columns in different experiments are determined by the rules in Section III-B3. When reporting the performance, `LEN` represents the word length. The scales are set to different values in experiments. The relations used in the experiments contain 10 million tuples unless otherwise specified, and the data in `DECIMAL` columns are randomly generated.

Peer systems. We select a couple of representative databases to compare with UltraPrecise. For CPU databases, **PostgreSQL** v14.4 and **MonetDB** v11.46.0 are selected. PostgreSQL supports arbitrary-precision arithmetic toward fixed-point decimals. MonetDB is a high-performance in-memory database but only supports `DECIMAL` at limited precision. For GPU databases, we select **HEAVY.AI** v6.3.0 and an open-source version of **RateupDB** for academic partners, both using GPU to accelerate the query execution but do not support high-precision `DECIMAL`. RateupDB supports the max precision of 36 while HEAVY.AI only supports 18. We additionally evaluate two synthesized workloads, i.e., RSA encryption and sine functions, on **CockroachDB** v23.1.0 and **H2** v2.1.214. The two databases claim to support `DECIMAL` at arbitrary-precision as PostgreSQL.

A. Decimal Representation

We first evaluate the representation design with Query 1, where the relation `R1` has three columns of decimals. The only expression of the query is composed of three columns in a `DECIMAL` type with the same precision and scale. Thus, no alignment scheduling and constant optimization are introduced. We generate 5 versions of `R1` with increasing precision. For all relations, the columns have the same scale of 2. We disable the multi-threading arithmetic in the experiments.

```
SELECT c1+c2+c3 FROM R1;
```

Query 1

The results are reported in Figure 8, where the performance numbers are shown in groups of bars. Each group represents the varying result precision, from 2 to 32 words. The y-axis is the execution time in seconds. The execution time includes

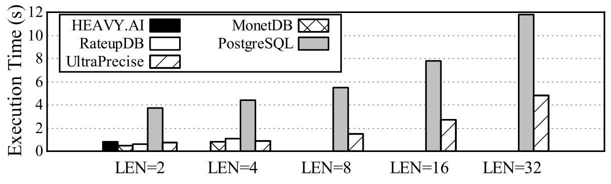

Fig. 8. Performance of various databases when executing the Query 1

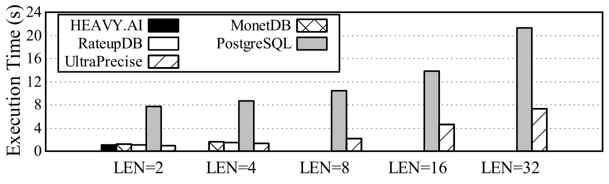

Fig. 9. Performance of various databases when executing Query 2

the disk I/Os except for MonetDB, which is designed as an in-memory database. The execution time of GPU databases, i.e., HEAVY.AI, RateupDB, and UltraPrecise, also includes the PCIe transfer time. For HEAVY.AI, it executes the query successfully only when the decimals can be contained in two 32-bit words, since it uses a 64-bit word to represent the `DECIMAL` type, no matter how the precision and scale are defined. MonetDB fails to execute the query as well when `LEN` exceeds 4 because only two 64-bit words are used in it. RateupDB has the same problem in that internally at most 5 32-bit words are used to represent the decimals. When the precision is low, MonetDB is the fastest database because disk I/Os are excluded. It uses 461 ms and 800 ms to execute the query as `LEN` is 2 and 4, respectively. UltraPrecise is slower than RateupDB when `LEN` is 2 because we introduce the JIT engine to evaluate the decimal expression. UltraPrecise uses 714 ms and RateupDB uses 622 ms. When `LEN` increases to 4, UltraPrecise becomes faster than RateupDB, which is 902 ms versus 1055 ms. The reason is that the compilation time in UltraPrecise is almost the same but the computation time increases. As our representation design benefits the computation, our system becomes faster. Surprisingly, HEAVY.AI is the slowest one among GPU databases, which takes 800 ms even though its decimal arithmetic is evaluated as integers. PostgreSQL accomplishes all query executions as ours but with a quite slower speed. Due to the GPU acceleration and the representation, we speed up the execution by up to 5.24×.

We further execute a more complicated query shown in Query 2, where `R2` is a relation of 8 columns. We still generate 5 versions of `R2`, where `c1−c4` are constantly defined as `DECIMAL`(6, 2) and `c5−c8` are defined with increasing precision. As a result, the first expression always outputs the decimal result in one word and the second expression outputs the result with increasing `LEN`. In our implementation, two GPU kernels are generated in query execution.

```
SELECT c1+c2+c3+c4, c5+c6+c7+c8 FROM R2;
```

Query 2

The performance of executing Query 2 in various databases is shown in Figure 9. As the expressions are more computation intensive, UltraPrecise outperforms other databases and is the fastest in all cases. When `LEN` is 2, UltraPrecise takes 969 ms while HEAVY.AI, RateupDB, and MonetDB take 1.09 s, 1.02 s, and 1.27 s to finish the execution. When `LEN` is 4, UltraPrecise, RateupDB, and MonetDB spend 1.32 s, 1.55 s, and 1.69 s. HEAVY.AI and RateupDB are faster than MonetDB because the GPU databases are more advantageous in computation-intensive workloads. PostgreSQL is still the slowest. Our scheme outperforms PostgreSQL by up to 8.02×.

We use NVIDIA Nsight Compute [27] to profile kernels of evaluating $a + b$ and $a \times b$, where a and b are `DECIMAL` columns. The profiling results show that for additions, the SM utilization is 4.14% if `LEN` is 8 even though the warp occupancy is 100% already. As `LEN` increases to 32, the SM utilization decreases to 2.31% because more registers are required by a thread and the warp occupancy becomes 50%. As for multiplication, similar profiling results are observed. When `LEN` increases from 8 to 32, the SM utilization decreases from 3.70% to 3.23% while the warp occupancy reduces to 33% from 100% as well. The profiling results confirm that simple arithmetic operations are memory-intensive workloads, and our compact representation design accelerates the computation.

B. *Optimizations in Expressions*

1) Alignment scheduling: We evaluate the alignment scheduling by generating GPU kernels from three expressions: 1) $a+b+a$, 2) $a+b+a+a+a$, and 3) $a+b+a+a+a+a+a$, where both a and b are `DECIMAL`. b is `DECIMAL`(17, 11) when `LEN` is 2 or `DECIMAL`(18, 11) otherwise. a has increasing precisions and a constant scale of 1. With the alignment scheduling, the operand b is moved to the end of the expressions because of its large scale, and the alignment operations are reduced to 1 from 2, 4, and 6 times, respectively. We report the experimental results in Figure 10. We can observe the trends from the experimental results that more time is saved with the growing precision and the longer expression length. When the expression is $a + b + a + a + a + a + a$ and `LEN` is 32, the alignment scheduling improves the execution time by 34%. For $a + b + a$ and `LEN` is 2, the saving is 16.5%.

2) Constant construction: One of the techniques for optimizing the decimal expression is constant construction, i.e., instead of converting the constants to `DECIMAL` in runtime, the conversion is moved to the compile time. In this experiment, we generate GPU kernels to evaluate the expression of $1+a$, where a is a `DECIMAL` with increasing precision and the same scale of 10. When generating the code, the constant 1 is converted to a `DECIMAL` with the precision 10 and the scale 10, which aligns with the scale of a. The experimental results are shown in Figure 11. We can observe that with increasing

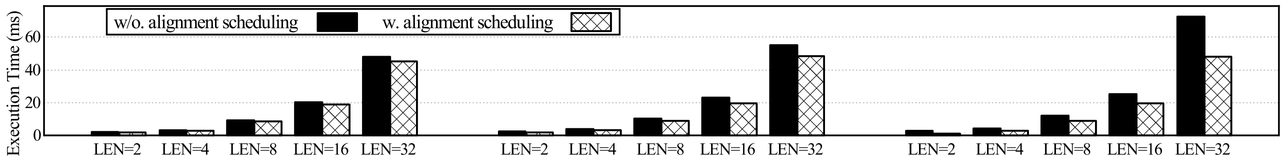

Fig. 10. Performance of GPU kernels evaluating various expressions. From left to right, the expressions are $a+b+a$, $a+b+a+a+a$, and $a+b+a+a+a+a+a$.

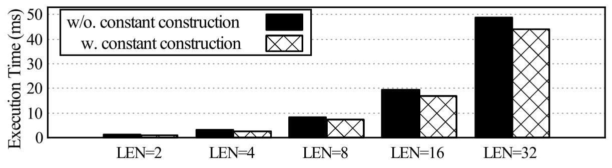

Fig. 11. Performance of GPU kernels evaluating $1 + a$

precision, we save more execution time of GPU kernels due to the higher alignment overhead. When the size of the array storing a increases from 2 to 32, it speeds up the execution by 1.33×, 1.25×, 1.14×, 1.14×, and 1.11×.

3) Constant pre-calculation: We test the optimization technique of constant pre-calculation, which calculates the sub-expression with constants only at the compile time. We generate the GPU kernels of $1 + a + 2 + 11$, $1 + a + 2 - 3$, and $0.25 \times (a+b) \times 4$, where a and b are `DECIMAL` with increasing precision and the same scale of 10. Without the constant pre-calculation, three GPU kernels are generated, and the first two need to process 3 additions or subtractions. The last kernel processes 2 multiplications and 1 addition. If the constant pre-calculation is enabled, $1+a+2+11$ is transformed to $14+a$, and in the GPU kernel, 14 is represented as `DECIMAL`(12, 10). We reduce 3 additions to 1 addition in the code. For $1+a+2-3$, the expression is reduced to a, and no GPU kernel is generated. For $0.25 \times (a+b) \times 4$, we actually evaluate $a+b$. The results are shown in Figure 12, where the execution time is saved up to 62.55%, 100.00%, and 62.50%, respectively.

C. Multi-threading Arithmetic

1) Expression evaluation: We also conduct experiments to measure the kernel execution time if an expression is evaluated in the multi-threading manner, i.e., the arithmetic operations are calculated by a group of threads. We generate kernels for 3 expressions (since the addition and the subtraction are almost the same): 1) $a + b$, 2) $a \times b$, and 3) $a \div b$, where a and b are `DECIMAL` types. In the experiments, we set TPI to 1, 4, 8, 16, and 32. When `LEN` exceeds TPI, every thread in the group processes `LEN`/TPI words of data. The kernels are executed 3 times and we report the average results in Figure 13.

Additions and multiplications have a similar performance trend. When the precision is low, the performance of single-threading arithmetic operations is similar to the multi-threading ones. For `LEN` is 4, the single-threading and the 4-threading kernels both take 3.67 ms to add two decimals. However, the multi-threading implementations are more efficient as the operand precision grows. When `LEN` is 32, the single-threading kernel takes 49.67 ms and 45.00 ms in the additions and the multiplications, respectively. In contrast to this, the multi-threading implementations take as fast as 23.67 ms (8-threading) for additions and 23.33 ms (8-threading) for multiplications. The multi-threading implementations are more efficient because the memory accesses to a value array are coalesced in a thread group. For the divisions, different algorithms are used in the single-threading kernel and the multi-threading kernel. In the single-treading kernel, the result is calculated using binary search in a range of potential quotients. In the multi-treading kernel, we follow the implementation in CGBN [24], [25], which is based on the Newton-Raphson algorithm. However, the implementation has a restriction that `LEN`/TPI must be less than or equal to TPI, so no data is presented when executing the 4-threading kernel and `LEN` is 32.

```
SELECT SUM(c1) FROM R3;
```

Query 3

2) Multi-threading aggregation: We also implement the aggregation operations of `DECIMAL` in multi-threading. We measure the execution time of Query 3 in various database systems, where `R3` has only one column of decimals. We vary the precision and scale of `c1` in (11, 7), (29, 11), (65, 31), (137, 51), and (281, 101) so that the final aggregation results are stored in the 32-bit word-aligned array with the sizes of 2, 4, 8, 16, and 32. The TPI is 8, and the other databases also execute the aggregation operation in parallel.

The experimental results are shown in Figure 14(a). As in the expression evaluation, HEAVY.AI supports the aggregation only if `LEN` is 2, while MonetDB and RateupDB support aggregations if `LEN` is less than or equal to 4. MonetDB is the fastest database when `LEN` is 2 and 4, which takes 0.017 s and 0.019 s to execute the query. The reason is that the disk I/Os are not included in the results. Compared to RateupDB, UltraPrecise reduces the execution time by 33.33% (`LEN`=2) and 12.50% (`LEN`=4) because of the multi-threading implementation and the compact data representation. As for HEAVY.AI, it takes the longest time to execute the query when `LEN` is 2, which is 0.47 s. PostgreSQL could accomplish the query execution like UltraPrecise but with slower speeds. When `LEN` is 8, 16, and 32, PostgreSQL needs 112.12%, 67.24%, and 29.25% more time.

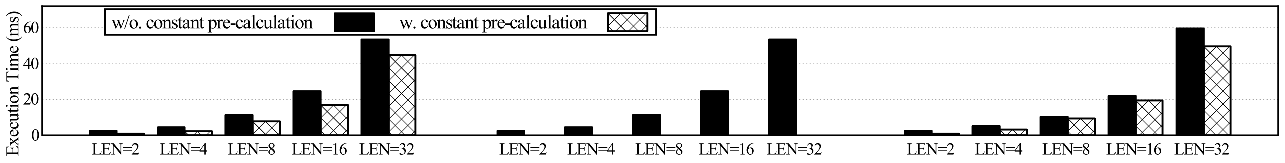

Fig. 12. Performance of GPU kernels evaluating various expressions. From left to right, the expressions are $1+a+2+11$, $1+a+2-3$, and $0.25\times(a+b)\times4$.

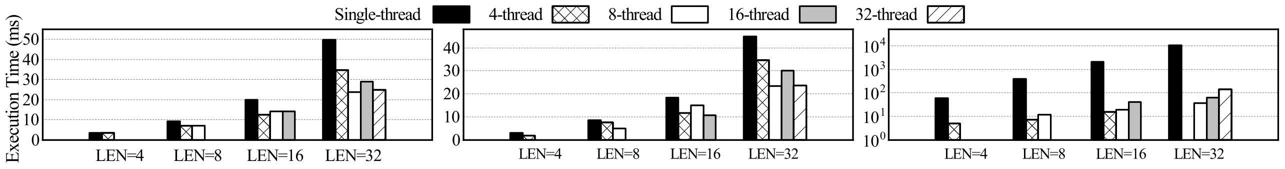

Fig. 13. Performance of GPU kernels evaluating single arithmetic operators. From left to right, the expressions are $a+b$, $a\times b$, and $a\div b$.

D. Synthesized Workloads

1) TPC-H Q1: We execute TPC-H Q1 in various databases, where 4 columns, `l_quantity`, `l_extendedprice`, `l_discount`, and `l_tax`, are defined as decimal types. In the query execution, we need to evaluate 2 expressions and 7 aggregations with decimals. To more comprehensively test our system, we extend the precision of `l_quantity` and `l_extendedprice` in experiments and guarantee that the results can be stored in the 32-bit word array with lengths of 2, 4, 6, and 8. For `l_discount` and `l_tax`, they are always defined as `DECIMAL`(3, 2) because their values are between 0 and 1. We also test the original version of TPC-H Q1, where all columns are `DECIMAL`(12, 2). In the experiments, we exclude the scan time from all database systems.

Figure 14(b) shows the results. Among all methods, our system is only slower than HEAVY.AI which is optimized for low precision. For the original Q1 and `LEN`=2, HEAVY.AI executes the query in 489.00 ms and 642.33 ms while our system takes 684.67 ms and 685.00 ms. But HEAVY.AI fails to support decimal type at higher precision. When `LEN` is 4, 8, 16, and 32, UltraPrecise spends 754.67 ms, 1135.33 ms, 2610.33 ms, and 6164.33 ms to execute the query. MonetDB and RateupDB execute Q1 slower than UltraPrecise. MonetDB is 1.64× (orignal Q1), 1.17× (`LEN`=2), and 1.52× (`LEN`=4) slower. RateupDB is 1.70× (orignal Q1), 1.52× (`LEN`=2), and 1.61× (`LEN`=4) slower. PostgreSQL takes the most execution time. UltraPrecise is 41.28× (original Q1), 39.55× (`LEN=2`), 38.56× (`LEN=4`), 28.09× (`LEN=8`), 14.46× (`LEN=16`), and 7.70× (`LEN=32`) faster than PostgreSQL. The compilation and execution of UltraPrecise are individually measured for Q1. With the `LEN` increasing from 2 to 32, the time proportion of compilation decreases from 47% to 7%, though the absolute compilation time increases from 320 ms to 423 ms due to the longer code generated.

We have also evaluated the potential benefit of the compression technique of frame-of-reference (FOR) [28] to UltraPrecise for Q1 as a case study. We generate `l_quantity` and `l_extendedprice` with different distributions so that the `DECIMAL` columns are compressed into different sizes. We decompress the values before the calculation in the kernel. The experimental results show that when `LEN` is 4, 8, 16, and 32, the execution time including PCIe transferring could be accelerated by 1.38×, 2.01×, 3.36×, and 4.80×, which depends on how much data we could compress. If an advanced scheme featuring little decompression overhead [29] is incorporated, the performance of UltraPrecise could be further improved.

2) Other TPC-H queries: We execute other TPC-H queries with UltraPrecise and RateupDB to see if the performance of queries without `DECIMAL` is impaired. We set the scale to 10, and the experimental results are shown in Table I. For all queries except Q18 and Q20, the performance results of the two databases are consistent and comparable, affirming the stability of UltraPrecision. By looking at the code, the longer execution time of two queries results from subqueries returning `DECIMAL` values. In RateupDB, delivering results of subqueries to the outer query is not JIT-based and our efficient representation cannot be applied.

```
SELECT c1 * c1 % N * c1 % N FROM R4;
```

Query 4

3) RSA: Rivest-Shamir-Adleman (RSA) [30] is one of the most widely deployed cryptosystems, which uses a pair of keys to encrypt and decrypt messages. To encrypt a message X, we need to calculate $X^e \mod N$, where N is a product of two big prime numbers and (e, N) is the encryption key. With the increasing size of N, the encrypted message gets harder to exploit. In this experiment, we implement RSA encryption in SQL. Query 4 encrypts the messages stored in the column `c1`, which is decimal type. 4 versions of `R4` are generated and the precision of `c1` is 17, 35, 71, and 143. The scale of `c1` is always 0. We set e as 3 and N is presented also as decimal type constants with the precisions and scales of (18, 0), (36, 0), (72, 0), and (144, 0). In the end, we execute Query 4 to accomplish the encryption of data stored in `c1`.

TABLE I
EXECUTION TIME (MS) OF TPC-H QUERIES IN RATEUPDB AND ULTRAPRECISE

	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
RateupDB	160	278	68	409	71	562	301	612	490	120	70	106	81	227	97	400	447	94	367	551	42
UltraPrecise	169	271	67	400	57	538	314	614	503	136	67	100	72	226	95	332	690	99	476	586	46

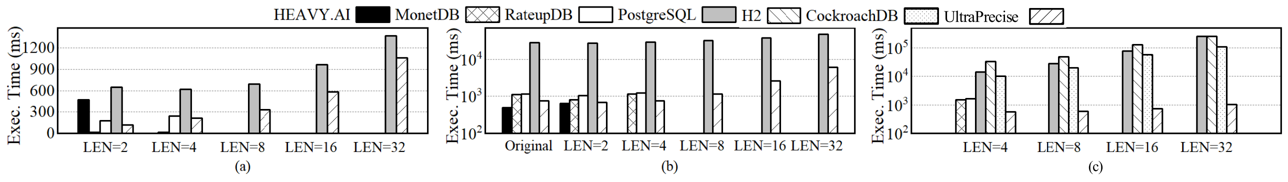

Fig. 14. Performance of various databases executing (a) Query 3, (b) TPC-H Q1, and (c) Query 4

HEAVY.AI fails to execute this query because it does not support the modulo operator of the decimal type. We measure the execution time including the scan operation for all databases. The results are shown in Figure 14(c). We can observe that UltraPrecise is the most efficient solution. It finishes the encryption in 574.67 ms, 601.00 ms, 738.33 ms, and 1018.67 ms when `LEN` is 4, 8, 16, and 32. MonetDB and RateupDB take 1520.67 ms and 1628.00 ms to execute the query when `LEN` is 4. PostgreSQL encrypts the messages 22.22×, 47.55×, 106.19×, and 247.59× slower than UltraPrecise. H2 and CockroachDB are even slower than PostgreSQL for these queries.

```
SELECT c1 - c1*c1*c1/6 + c1*c1*c1*c1*c1/120
FROM R5;
```

Query 5: A query of approximating $\sin(x+\epsilon)$. We vary `c1` to `c2` or `c3` for different input x and append more terms for higher precision results.

4) Trigonometric functions: Trigonometric functions are used extensively in various fields, e.g., calculating distances on geospatial datasets. An effective approach to deriving high-precision results of trigonometric functions is using Taylor Series. An example is the sine function: $\sin(x)$ is approximated with a polynomial $x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$, where x is an input radian. In the experiments, we compose SQL queries to calculate $\sin(\cdot)$ at high precision, and we show one of the queries using 3 terms in Query 5.

The queries are executed on a relation `R5` that has three columns `c1`, `c2`, and `c3`, storing the input radian values. All the columns are `DECIMAL`(9, 8). The radian values of `c1` are randomly generated that follow the normal distribution $N(0.01, 0.01^2)$, representing the extremely small angles close to 0. The radian values of `c2` and `c3` follow the normal distributions of $N(0.78, 0.01^2)$ and $N(1.56, 0.01^2)$, which are close to $\pi/4$ and $\pi/2$, respectively. In the queries, we expand the polynomial from 2 terms to 11 terms, expecting to calculate increasingly precise results. We verify the results via GMP [43], by which we calculate the ground truth results until 287 digits after the decimal point. The queries are executed in PostgreSQL, CockroachDB, H2, and UltraPrecise. We report the execution time in milliseconds as the y-axis and the mean absolute error (MAE) as the x-axis in Figure 15, where each point is one time of execution. For all queries, UltraPrecise has the lowest execution time from 505.67 ms to 1668.33 ms, which is almost two orders of magnitude faster compared to other databases. UltraPrecise is also more scalable. For example, when extending the length of the polynomial that calculates $\sin(0.78+\epsilon)$, i.e., more `DECIMAL` arithmetic operations are added, UltraPrecise increases its execution time by only 1131.33 ms. But the execution time is increased by 134.51 s, 191.27 s, and 385.57 s in PostgreSQL, H2, and CockroachDB, respectively. When calculating $\sin(0.01+\epsilon)$, the precision of results saturates after 4 or 5 terms appended because `DECIMAL`(9, 8) cannot provide enough precision to extremely small values. Though we follow the rules discussed in Section III-B3, only 4 digits can hardly protect the division from underflow in the cases. Other databases also have such a problem except for H2. The reason is that H2 adds 20 additional digits in `DECIMAL` divisions. But this would also incur high computation overhead in a workload with intensive divisions. PostgreSQL executes the query faster when appending the 10th term, reducing the execution time by 48.89 s, 54.87 s, and 67.76 s, respectively. This is due to that PostgreSQL started to enable parallel scans in its query plan.

V. RELATED WORK

`DECIMAL` in databases. Fixed-point decimal arithmetic systems are databases dependent, with varying precision and scale ranges. PostgreSQL and YugabyteDB support up to 131,072 digits before and 16,383 digits after the decimal point, while Greenplum claims to impose no limits. Vertica and PolarDB support a maximum precision of about 1,000. Another group of databases incorporates external libraries to support arbitrary-precision `DECIMAL`. SparkSQL and H2 use `java.math.BigDecimal`, although SparkSQL stopped supporting arbitrary-precision since version 1.5. Hive initially used the same Java library, but they later developed their decimal implementation, supporting up to a maximum

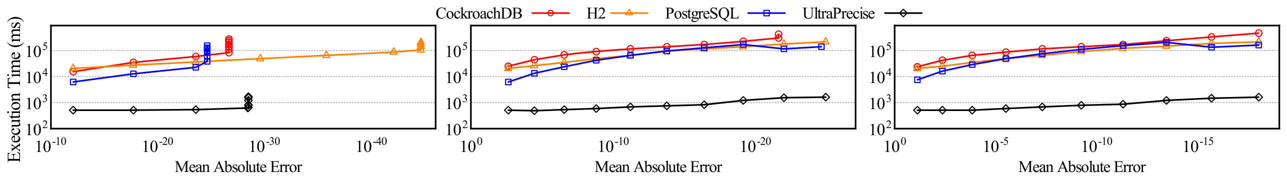

Fig. 15. Performance of various databases when executing SQL queries that approximate $\sin(x+\epsilon)$ with different input x and varying polynomial lengths from 2 to 11. From left to right, the functions are $\sin(0.01+\epsilon)$, $\sin(0.78+\epsilon)$, and $\sin(1.56+\epsilon)$.

TABLE II
THE DECIMAL PRECISIONS IN A GROUP OF DATABASES

Database	Max (p, s)	Database	Max (p, s)	Database	Max (p, s)	Database	Max (p, s)
PostgreSQL [8]	(147,455, 16,383)	Greenplum [10]	no limit	PrestoDB [31]	(38, 18)	RateupDB [19]	(36, 36)
YugabyteDB [9]	(147,455, 16,383)	CockroachDB [11]	no limit	SQL Server [32]	(38, 38)	Hive [33]	(38, 38)
H2 [12]	(100,000, 100,000)	Vertica [34]	(1,024, 1,024)	HEAVY.AI [35]	(18, 18)	Oracle [36]	(38, 127)
MongoDB [37]	double and string	SparkSQL [38]	(38, 38)	MonetDB [39]	(38, 38)	MySQL [40]	(65, 30)
PolarDB [41]	(1000, 1000)	Google Spanner [42]	(38, 9)				

precision of 38. CockroachDB initially used a Go external library for arbitrary-precision arithmetic but later developed its customized library. To store `DECIMAL` data efficiently and precisely, MongoDB stores two fields: a string type for storing the exact value and a floating-point type for quick arithmetic operations. In other databases, the `DECIMAL` type of arbitrary-precision arithmetic is not feasible because of its inferior performance. For instance, Google Spanner supports up to a maximum precision of 38 and recommends using the string type beyond this limit. HEAVY.AI is the most limited database, only supporting `DECIMAL` data at the maximum precision of 18. RateupDB supports a max precision of 36, which is the platform for our system implementation. Oracle deviates from the convention of scale $\leq$ precision. The precisions of `DECIMAL` supported in a group of databases are in Table II.

Arbitrary-precision arithmetic. Arbitrary-precision arithmetic (APA) is widely supported by programming languages and general arithmetic libraries. The supports to APA are found in languages like Python, Java, C#, and MATLAB, where `java.math.BigDecimal` in Java and the `decimal` module in Python are two examples. Several individual libraries also support APA, among them the most widely used one is the GNU Multiple Precision Arithmetic Library [43]. Class Library for Numbers [44] is another representative one. Cooperative Groups Big Numbers (CGBN) [24], [25] is developed by NVIDIA that realize arithmetic operations for big positive integers on GPU. APA is also used in geometric computation like Computational Geometry Algorithms Library (CGAL) [45] and GeometricTools [46]. OpenSSL [47] and Crypto++ [48] also require APA for cryptography operations.

GPU databases. CoGaDB [49] uses an "operator-at-a-time" approach to query execution, with each operator having a corresponding kernel function. GPUDB [50] and HippogriffDB [51] strive to alleviate the transmission overhead between GPU and host. Kernel Weaver [52], GPL [53], and HorseQC [54] optimize data movements on GPU when executing kernels. DogQC [55] proposes to balance divergence effects during the execution of the fusion pipeline. The heterogeneous designs [56]–[59] have also been explored.

JIT in query processing. System R [60] is the first to propose code generation to optimize queries. DBToaster [61] generates efficient C++ code for view maintenance queries, while Cloudera Impala [62] focuses on code generation for data parsing and expression calculation. HIQUE [63] uses C++ templates to generate code for all operators and compile them into a dynamic link library. Hyper [64] uses LLVM IR code to speed up compilation. ROF [65], Tupleware [66], and LegoBase [67] apply different optimization strategies. Zhang et al. [68] propose to generate code dynamically for data chunks. Zeuch et al. [69] provide an accurate cost model for a JIT-enabled database. Other approaches [70], [71] adaptively switch between translation and interpretation for faster compilation.

VI. CONCLUSION

Fixed-point arithmetic operations in a database with exactness and high precision are desirable but challenging. We have achieved both high precision and high performance for fixed-point arithmetic operations through effective GPU acceleration. UltraPrecise offers comparable performance to high-performance databases that are in a low-precision range. However, its arithmetic operations are faster by two orders of magnitude compared to the databases that support arbitrary-precision `DECIMAL`.

ACKNOWLEDGMENT

This work is supported in part by the National Natural Science Foundation of China, No. 62102229, Natural Science Foundation of Shandong Province, China, No. ZR2022ZD02, and the National Science Foundation under grants CCF-2005884, CCF-2210753, OAC-2310510, and CCF-2312507.

References

[1] H. Wang, C. Zaniolo, and C. R. Luo, "ATLAS: A small but complete SQL extension for data mining and data streams," in *Proceedings of the VLDB Endowment*, 2003, pp. 1113–1116.

[2] M. Blacher, J. Giesen, S. Laue, J. Klaus, and V. Leis, "Machine learning, linear algebra, and more: Is SQL all you need," *Conference on Innovative Data Systems Research*, pp. 1–6, 2022.

[3] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for In-RDBMS analytics," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 325–336.

[4] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, "In-RDBMS hardware acceleration of advanced analytics," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, 2018.

[5] J. V. D'silva, F. De Moor, and B. Kemme, "AIDA: Abstraction for advanced in-database analytics," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1400–1413, 2018.

[6] IEEE, "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985.

[7] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.

[8] PostgreSQL, "PostgreSQL Documentation: Datatypes," https://www.postgresql.org/docs/current/datatype-numeric.html, 2023, accessed on July 22, 2023.

[9] YugabyteDB, "YugabyteDB Documentation: Datatypes," https://docs.yugabyte.com/preview/api/ysql/datatypes/type_numeric/, 2023, accessed on July 22, 2023.

[10] GreenPlum, "GreenPlums Documentation: Datatypes," https://gpdb.docs.pivotal.io/6-9/ref_guide/data_types.htmlW, 2023, accessed on July 22, 2023.

[11] CockroachDB, "CockroachDB Documentation: Datatypes," https://www.cockroachlabs.com/docs/stable/decimal.html, 2023, accessed on July 22, 2023.

[12] H2, "H2 Documentation: Datatypes," https://www.h2database.com/html/datatypes.html, 2023, accessed on July 22, 2023.

[13] D. H. Bailey, R. Barrio, and J. M. Borwein, "High-precision computation: Mathematical physics and dynamics," *Applied Mathematics and Computation*, vol. 218, no. 20, pp. 10 106–10 121, 2012.

[14] N. Vuković, M. Petrović, and Z. Miljković, "A comprehensive experimental evaluation of orthogonal polynomial expanded random vector functional link neural networks for regression," *Applied Soft Computing*, vol. 70, pp. 1083–1096, 2018.

[15] D. H. Bailey and J. M. Borwein, "High-precision arithmetic in mathematical physics," *Mathematics*, vol. 3, no. 2, pp. 337–367, 2015.

[16] D. H. Bailey, J. M. Borwein, R. E. Crandall, and I. J. Zucker, "Lattice sums arising from the poisson equation," *Journal of Physics A: Mathematical and Theoretical*, vol. 46, no. 11, p. 115201, 2013.

[17] Z. Qu and D. Tkachenko, "Using arbitrary precision arithmetic to sharpen identification analysis for DSGE models," https://open.bu.edu/handle/2144/41799, 2020.

[18] PostgreSQL, "PostgreSQL Database Management System," https://github.com/postgres/postgres/blob/master/src/backend/utils/adt/numeric.c, 2023, accessed on July 22, 2023.

[19] R. Lee, M. Zhou, C. Li, S. Hu, J. Teng, D. Li, and X. Zhang, "The art of balance: A RateupDB™ experience of building a CPU/GPU hybrid database product," *Proceedings of the VLDB Endowment*, pp. 2999–3013, 2021.

[20] A. Karatsuba, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, vol. 7, 1963, pp. 595–596.

[21] A. Schonhage, "Schnelle multiplikation grosser zahlen," *Computing*, vol. 7, pp. 281–292, 1971.

[22] Wikipedia, "Newton's method," https://en.wikipedia.org/wiki/Newton%27s_method, 2023, accessed on July 22, 2023.

[23] R. E. Goldschmidt, "Applications of division by convergence," Ph.D. dissertation, Massachusetts Institute of Technology, jun 1964. [Online]. Available: http://hdl.handle.net/1721.1/11113

[24] NVIDIA Corp, "CGBN: CUDA accelerated multiple precision arithmetic (big num) using cooperative groups," https://github.com/NVlabs/CGBN/, 2023, accessed on July 22, 2023.

[25] N. Emmart, J. Luitjens, C. Weems, and C. Woolley, "Optimizing modular multiplication for nvidia's maxwell GPUs," in *2016 IEEE 23nd symposium on computer arithmetic*, 2016, pp. 47–54.

[26] S. Baxter, "Moderngpu 2.0," 2016, unpublished. [Online]. Available: https://github.com/moderngpu/moderngpu/wiki

[27] NVIDIA Nsight Compute, https://developer.nvidia.com/nsight-compute, 2023, accessed on Nov 8, 2023.

[28] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes," in *Proceedings 14th International Conference on Data Engineering*, 1998, pp. 370–379.

[29] A. Shanbhag, B. W. Yogatama, X. Yu, and S. Madden, "Tile-based lightweight integer compression in gpu," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1390–1403.

[30] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[31] PrestoDB, "PrestoDB Documentation: Datatypes," https://prestodb.io/docs/current/language/types.html, 2023, accessed on July 22, 2023.

[32] SQL Server, "SQL Server Documentation: Datatypes," https://learn.microsoft.com/en-us/sql/t-sql/data-types/, 2023, accessed on July 22, 2023.

[33] Hive, "Hive Documentation: Datatypes," https://cwiki.apache.org/confluence/download/attachments/27362075/Hive_Decimal_Precision_Scale_Support.pdf, 2023, accessed on July 22, 2023.

[34] Vertica, "Vertica Documentation: Datatypes," https://www.vertica.com/docs/9.3.x/HTML/Content/Authoring/SQLReferenceManual/DataTypes/Numeric/NUMERIC.htm, 2023, accessed on July 22, 2023.

[35] HEAVY.AI, "HEAVY.AI Documentation: Datatypes," https://docs.heavy.ai/sql/data-definition-ddl/datatypes-and-fixed-encoding, 2023, accessed on July 22, 2023.

[36] Oracle, "Oracle Documentation: Datatypes," https://docs.oracle.com/en/database/oracle/oracle-database/21/odpnt/DecimalMembers.html, 2023, accessed on July 22, 2023.

[37] MongoDB, "MongoDB Documentation: Datatypes," https://www.mongodb.com/docs/v3.0/tutorial/model-monetary-data/, 2023, accessed on July 22, 2023.

[38] SparkSQL, "SparkSQL Documentation: Datatypes," https://spark.apache.org/docs/2.2.0/sql-programming-guide.html, 2023, accessed on July 22, 2023.

[39] MonetDB, "MonetDB Documentation: Datatypes," https://www.monetdb.org/documentation-Sep2022/user-guide/sql-manual/data-types/base-types/, 2023, accessed on July 22, 2023.

[40] MySQL, "MySQL Documentation: Datatypes," https://dev.mysql.com/doc/refman/8.0/en/numeric-type-syntax.html, 2023, accessed on July 22, 2023.

[41] PolarDB, "PolarDB Documentation: Datatypes," https://www.alibabacloud.com/help/en/polardb-for-oracle/latest/numeric-type, 2023, accessed on July 22, 2023.

[42] Google Spanner, "Google Spanner Documentation: Datatypes," https://cloud.google.com/spanner/docs/storing-numeric-data/, 2023, accessed on July 22, 2023.

[43] The GNU Multiple Precision Arithmetic Library, https://gmplib.org/, 2023, accessed on July 22, 2023.

[44] B. Haible and R. B. Kreckel, "CLN - Class Library for Numbers," https://www.ginac.de/CLN/, 2023, accessed on July 22, 2023.

[45] A. Fabri and S. Pion, "CGAL: The computational geometry algorithms library," in *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, 2009, pp. 538–539.

[46] D. Eberly, "Geometric tools," https://www.geometrictools.com/, 2023, accessed on July 22, 2023.

[47] OpenSSL Cryptography and SSL/TLS Toolkit, http://www.openssl.org/, 2023, accessed on July 22, 2023.

[48] Crypto++ Library 8.7, https://www.cryptopp.com/, 2023, accessed on July 22, 2023.

[49] S. Breß, "The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS," *Datenbank-Spektrum*, vol. 14, pp. 199–209, 2014.

[50] Y. Yuan, R. Lee, and X. Zhang, "The Yin and Yang of processing data warehousing queries on GPU devices," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 817–828, 2013.

[51] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, "Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics," *Proceedings of the VLDB Endowment*, vol. 9, no. 14, pp. 1647–1658, 2016.

[52] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, “Kernel Weaver: Automatically fusing database primitives for efficient GPU computation,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 107–118.

[53] J. Paul, J. He, and B. He, “GPL: A GPU-based pipelined query processing engine,” in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1935–1950.

[54] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner, “Pipelined query processing in coprocessor environments,” in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1603–1618.

[55] H. Funke and J. Teubner, “Data-parallel query processing on non-uniform data,” *Proceedings of the VLDB Endowment*, vol. 13, no. 6, pp. 884–897, 2020.

[56] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, “Relational query coprocessing on graphics processors,” *ACM Transactions on Database Systems*, vol. 34, no. 4, pp. 1–39, 2009.

[57] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl, “Generating custom code for efficient query execution on heterogeneous processors,” *The VLDB Journal*, vol. 27, pp. 797–822, 2018.

[58] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, “HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines,” *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 544–556, 2019.

[59] B. W. Yogatama, W. Gong, and X. Yu, “Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS,” *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2491–2503, 2022.

[60] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu *et al.*, “A history and evaluation of system R,” *Communications of the ACM*, vol. 24, no. 10, pp. 632–646, 1981.

[61] Y. Ahmad and C. Koch, “DBToaster: A SQL compiler for high-performance delta processing in main-memory databases,” *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1566–1569, 2009.

[62] S. Wanderman-Milne and N. Li, “Runtime code generation in cloudera impala.” *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 37, no. 1, pp. 31–37, 2014.

[63] K. Krikellas, S. D. Viglas, and M. Cintra, “Generating code for holistic query evaluation,” in *2010 IEEE 26th International Conference on Data Engineering*, 2010, pp. 613–624.

[64] T. Neumann, “Efficiently compiling efficient query plans for modern hardware,” *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.

[65] P. Menon, T. C. Mowry, and A. Pavlo, “Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last,” *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 1–13, 2017.

[66] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik, “An architecture for compiling UDF-centric workflows,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1466–1477, 2015.

[67] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi, “Building efficient query engines in a high-level language,” *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 853–864, 2014.

[68] W. Zhang, J. Kim, K. A. Ross, E. Sedlar, and L. Stadler, “Adaptive code generation for data-intensive analytics,” *Proceedings of the VLDB Endowment*, vol. 14, no. 6, pp. 929–942, 2021.

[69] S. Zeuch, H. Pirk, and J.-C. Freytag, “Non-invasive progressive optimization for in-memory databases,” *Proceedings of the VLDB Endowment*, vol. 9, no. 14, pp. 1659–1670, 2016.

[70] A. Kohn, V. Leis, and T. Neumann, “Adaptive execution of compiled queries,” in *2018 IEEE 34th International Conference on Data Engineering*, 2018, pp. 197–208.

[71] A. Krolik, C. Verbrugge, and L. Hendren, “r3d3: Optimized query compilation on GPUs,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization*, 2021, pp. 277–288.