Supporting Extended Precision on Graphics Processors

Mian Lu
Hong Kong University of Science and Technology
lumian@cse.ust.hk

Bingsheng He
The Chinese University of Hong Kong
he.bingsheng@gmail.com

Qiong Luo
Hong Kong University of Science and Technology
luo@cse.ust.hk

ABSTRACT

Scientific computing applications often require support for non-traditional data types, for example, numbers with a precision higher than 64-bit floats. As graphics processors, or GPUs, have emerged as a powerful accelerator for scientific computing, we design and implement a GPU-based extended precision library to enable applications with high precision requirement to run on the GPU. Our library contains arithmetic operators, mathematical functions, and data-parallel primitives, each of which can operate at either multi-term or multi-digit precision. The multi-term precision maintains an accuracy of up to 212 bits of signifcand whereas the multi-digit precision allows an accuracy of an arbitrary number of bits. Additionally, we have integrated the extended precision algorithms to a GPU-based query processing engine to support efficient query processing with extended precision on GPUs. To demonstrate the usage of our library, we have implemented three applications: parallel summation in climate modeling, Newton's method used in nonlinear physics, and high precision numerical integration in experimental mathematics. The GPU-based implementation is up to an order of magnitude faster, and achieves the same accuracy as their optimized, quadcore CPU-based counterparts.

1. INTRODUCTION

New generation GPUs (Graphics Processing Units) have become a powerful and cost-effective co-processor for scientific applications [24]. With the advance on the support of general-purpose computation, GPUs have supported IEEE-compliant native double precision floating point numbers. However, data types with higher precision or extended precision are required in scientific applications [3] in two aspects. First, it is adopted to improve numerical reproducibility and stability. In some applications, computation with native floating point numbers can cause large rounding errors and produce incorrect final results, e.g., the climate modeling [19]. Second, some applications inherently require high precision support, such as high precision numerical integral in experimental mathematics [7]. Without extended precision support, such applications cannot be handled by modern processors. In this paper, we present the design and implementation of high-performance and high-precision algorithms on GPUs. Moreover, we demonstrate the performance of query processing with extended precision and related applications on GPUs.

Extended precision libraries, such as ARPREC [6], QD [20] and GMP [11], are provided to facilitate programming the extended precision on CPUs. Unfortunately, the computation with extended precision is much more expensive than native precision due to additional computation overhead introduced. For example, an arithmetic operator with *quad-double* precision (around 62 decimal digits), which consists of hundreds of native precision arithmetic instructions, is 25x slower than a native precision arithmetic operator on the CPU [3]. While the computational overhead can be a killer for the CPU-based applications, there are opportunities on the GPU to support fast computation with extended precision. This is because the GPU has over an order of magnitude higher computational power and memory bandwidth. For example, the NVIDIA 280 with 240 cores achieves over 1 Tera FLOPS (Floating Point Operations per Second) and over 100 GB per second memory bandwidth.

With superb raw hardware performance, we develop an extended precision library that is optimized with the GPU hardware features. In particular, the library includes arithmetics, mathematical functions and data parallel primitives. We have implemented two basic extended precision formats, namely *multi-term* [20] and *multi-digit* [6]. We show that an efficient implementation on the GPU requires an optimized memory layout, and memory accesses with a good temporal locality.

Considering GPUs have been shown as a promising platform for efficient query processing [18, 17], and modern scientific applications may rely on data management systems to maintain their data sets [15, 8], we integrate our extended precision library into an existing GPU-based query processing engine [17] to enable data management with extended precision for scientific applications on GPUs. This way, scientific applications requiring extended precision support may directly take advantage of this extended query processing engine on the GPU. For example, since the native precision causes large rounding errors in the global summation of climate modeling [19], we develop a user-defined aggregate function with extended precision in our query engine to implement the summation.

Based on our library and GPU-based query processing en-

Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010). June 7, 2010, Indianapolis, Indiana.

gine with extended precision support, we have implemented three real-world applications to demonstrate the effectiveness and efficiency. Specifically, the first application adopts the multi-term format to maintain the numerical reproducibility in a climate modeling application [19]. The second application computes a 42-digit physical constant [5] through Newton's method using the multi-term format. In the third application, the multi-digit format is adopted in the *numerical integration* algorithm [4] in experimental mathematics. Compared with the parallel CPU implementations on quad-cores, our GPU-based implementations are around 3-12x faster than their CPU counterparts, meeting the same accuracy requirement.

The remainder of the paper is organized as follows. The background and related work are presented in Section 2. We describe the implementation of high precision algorithms on GPUs in Section 3. Section 4 shows the experimental results on the efficiency of operations implemented in our library and database operations. Section 5 demonstrates the performance of three applications with extended precision on the GPU. Finally, we conclude in Section 6.

2. PRELIMINARY AND RELATED WORK

In this section, we introduce the preliminary on extended precision formats, and then review the related work on their applications in scientific computing.

2.1 Extended Precision Formats

Depending on how an extended precision number is represented, there are two basic kinds of formats, namely *multi-term* and *multi-digit*.

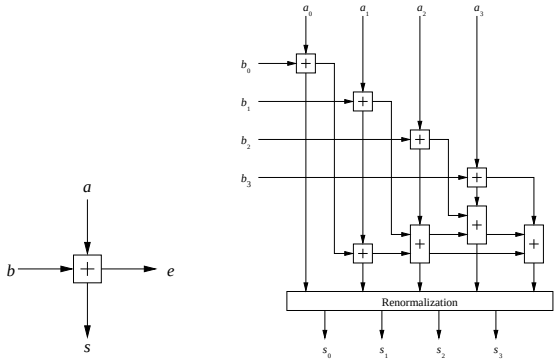
Multi-term. The multi-term format represents an extended precision number as a sum of multiple native floating point numbers, each of which has its own significand and exponent. The QD library [20] has adopted this approach and implemented *double-double* (around 31 decimal digits) and *quad-double* (around 62 decimal digits) precision, which use two and four native double precision numbers to represent an extended precision number, respectively.

Multi-digit. The multi-digit format stores an extended precision number with a sequence of digits coupled with a single exponent. ARPREC [6] is a representative high precision library using such method. In ARPREC, an extended precision number (denoted as A) is represented as an array of 64-bit words (the ith word is denoted as w_i): $A = \pm 2^{48 w_3} \sum_{k=0}^{n-1} w_{k+4} 2^{-48k}$, where $\pm = sign(w_2)$, $n = |w_2|$, and w_1 stores the number of words allocated for this array. The multi-digit format can represent an arbitrary precision number at run-time. However, the computation cost is usually higher than multi-term when the same precision is required.

Algorithm 1 TWO-SUM(a, b)

$s \leftarrow a \oplus b$;
$v \leftarrow s \ominus a$
$e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
return (s, e)

For the detailed mathematical background and algorithms for these extended precision formats, we refer readers to previous studies on extended precision techniques [9, 26, 20]. We show the example of an addition with quad-double precision. For any binary operator $\cdot \in \{+, -, \times, /\}$, and a and b with native double precision, we use $fl(a \cdot b) = a \odot b$ to denote the double precision result of $a \cdot b$, and define $a \cdot b = fl(a \cdot b) + err(a \cdot b)$. Algorithm 1 computes $s = fl(a+b)$ and $e = err(a+b)$. This algorithm is also illustrated in Figure 1(a). Let (a_0, a_1, a_2, a_3) and (b_0, b_1, b_2, b_3) represent two quad-double precision numbers, and the sum of these two numbers is represented as (s_0, s_1, s_2, s_3). Figure 1(b) shows how an addition is performed on two quad-double precision numbers. An additional Renormalization routine is used to normalize the five-item expansion to four components.



(a) TWO-SUM (b) Quad-double + quad-double

Figure 1: The algorithm of an addition with quad-double precision [20]. The addition of two quad-doubles numbers is implemented using the addition with doubles.

2.2 Related Work

Nowadays, extended precision is widely used in various scientific applications of different domains, such as physics, chemistry, and applied mathematics [3].

Extended precision has been used to improve numerical reproducibility and stability. He et al. [19] and Lake et al. [21] have presented such scenarios and extended precision solutions in climate modeling and planetary orbit calculations, respectively. Moreover, XBLAS [28] has adopted extended precision internally to improve the accuracy of results for basic linear algebra routines.

Extended precision is also used to support the computation requiring arbitrary precision. For example, the arbitrary precision is suitable for a few applications from experimental mathematics, such as *numerical integration* [7] and *integer relation detection* [2]. More recently, Gunnels et al. [16] have adopted the arbitrary precision library ARPREC to solve discrete optimization problems.

GPGPU(General-Purpose Computation on Graphics Processors) has become a fruitful research area in recent years. We refer readers to the survey on GPGPU [24]. Previous studies [18, 13, 10] show that the performance of database operations can be improved significantly with the GPU acceleration. There are several studies [14, 27] providing simple extended precision functions on GPUs using the multi-term format based on single precision. Moreover, the extended precision technique is adopted in some GPU-based applications [12]. To our best knowledge, there is neither a public, complete library for general purpose high preci-

sion computation on GPUs nor a comprehensive study for database operations with extended precision on GPUs.

3. EXTENDED PRECISION ON GPUS

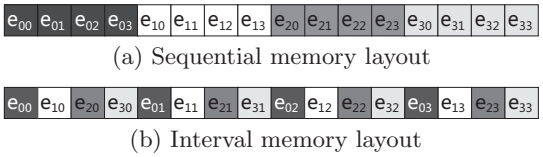
3.1 Library Overview

We have implemented both multi-term and multi-digit extended precision algorithms on the GPU using CUDA [22]. Since we have mainly adopted the algorithms implemented in the QD [20] and ARPREC [6] library on CPUs, we named our GPU-based multi-term and multi-digit implementations as GQD and GARPREC, respectively. Extended precision algorithms in those CPU libraries are designed based on IEEE-compliant double floating point numbers. Similar to the CPU-based extended precision libraries, our GPU-based library adopts native double floating point numbers for our implementations on GPUs. Both GQD and GARPREC have implemented arithmetic operators and important mathematical functions including square root, exponential and logarithm functions, and trigonometric functions. GQD consists of double-double and quad-double precision, which can achieve the accuracy of around 31 and 62 decimal digits, respectively. GARPREC is suitable for arbitrary precision computation. All arithmetic operators and mathematical functions are implemented as GPU kernel functions to facilitate further developments. Moreover, we have also implemented data parallel primitives with extended precision, including *map*, *scatter*, *gather*, *reduction*, *prefix-sum*, and *sorting* [18]. These primitives can be used to construct high level applications effectively.

3.2 Memory Optimizations

Memory optimizations are used to improve the access locality of the extended precision computation. We carefully design the data layout in order to exploit the *coalesced access* feature of the GPU. In CUDA, when each 16 threads in a thread group access consecutive memory addresses, these memory accesses are coalesced into one access. Moreover, we put the frequently accessed data, such as intermediate results, into the *local memory* of the GPU, which is a small user-programmable on-chip fast memory.

Coalesced access optimization. Although algorithms for extended precision computation implemented on GPUs are the same as traditional algorithms, an extended precision number requires multiple native precision numbers for the representation. To simplify our presentation, a native precision number used in extended precision formats is denoted as an *element*. Suppose m elements are required for each extended precision number. Given an array that contains n extended precision numbers, there are $(n \times m)$ elements used in this array. There are two methods to logically organize these elements. For extended precision libraries on CPUs, a *sequential* memory layout is adopted in QD and ARPREC. In this approach, m elements used for the same extended precision number are stored sequentially in a linear memory space. Suppose a thread accesses the jth element in the ith extended precision number, it actually accesses the $(i \times m + j)$th element in the array. Due to the SIMD nature of GPUs, the memory addresses accessed by the threads in a thread group are not consecutive. Therefore, these accesses do not match the coalesced access pattern of GPUs. Instead, the *interval* layout stores the elements of the same extended precision number at a regular interval, n. Then the jth element in the ith extended precision number is stored in the position of $(j \times n + i)$ in the array. In this way, memory addresses accessed by the threads in a thread group are consecutive, matching the coalesced access pattern of the GPU. Thus the interval memory layout is more suitable than the sequential memory layout on the GPU, and is adopted in our implementations. Specifically, we record the memory address of first element for each extended precision number for the representation. Other elements in an extended precision number can be accessed according to the address of first element and the number of extended precision number stored in the array. Figure 2 shows an example of different memory layouts for four extended precision numbers, and each one consists of four elements. Through evaluations, the implementation employing the interval memory layout is up to 3x faster than the sequential memory layout.



(a) Sequential memory layout

(b) Interval memory layout

Figure 2: Different memory layouts of extended precision numbers on GPUs. e_{ij} refers to the jth element used in the ith extended precision number. Elements in the same color belong to the same extended precision number.

Local memory optimization. Since an extended precision number is represented as multiple native numbers, the memory pressure of extended precision computation is much higher than native precision. Registers are usually used in the native precision computation on GPUs to hold necessary intermediate results. However, due to the large size of intermediate results, e.g., around 70 double precision numbers used for a 1000-digit extended precision number in GARPREC, registers of a multiprocessor on GPUs may be exhausted. Therefore, we use the local memory on GPUs with a larger size but also a low memory access latency to hold intermediate results. Moreover, the global memory is further used if the local memory is also not sufficient to store necessary intermediate results.

4. PERFORMANCE EVALUATION

This section presents the experimental results on the efficiency of extended precision operations on the GPU, in comparison with their CPU-based counterparts.

4.1 Experimental Setup

We conduct our experiments on a commodity PC running Fedora 10 Linux with a 2.40 GHz four-core CPU (Intel Q6600), 2 GB main memory and a NVIDIA GeForce GTX 280 GPU (G280). The theoretical bandwidth of G280 and CPU are 141.7 GB/sec and 10.4 GB/sec, respectively. G280 has 240 processor cores and 1 GB device memory. G280 natively supports IEEE-compliant double precision floating point numbers. Our implementations are developed using NVIDIA CUDA 2.0 [22]. QD [20] and ARPREC [6] are adopted as the CPU counterpart for the multi-term and multi-digit formats. Additionally, all CPU-based implementations are compiled with the -O3 optimization option and

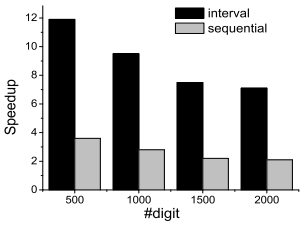


Figure 3: Performance comparison of the exponential function in GARPREC for different memory layouts with the precision varied.

developed using OpenMP [23] to take the advantage of multiple cores on the CPU.

We use the map primitive employing different functions to measure the performance of various extended precision computation operations in our library. The default data sets used for the multi-term and multi-digit formats are randomly generated 8 million and 1 million extended precision numbers, respectively. For both formats, speedups of typical arithmetic operations and mathematical functions on the GPU are reported. We also study the performance of three basic database operations with extended precision on GPUs, namely *selection*, *sort*, and *non-indexed nested-loop join* (NINLJ). We have excluded the time of memory transfer between the main memory and GPU memory for all evaluations.

While the native double precision on modern GPUs are IEEE-compliant, it does not necessarily mean that all extended precision computation on the GPU produces the exactly same result as that on the CPU. First, the IEEE floating point standard has not described the required error bounds or any implementation details for mathematical functions. A mathematical function with native precision on the GPU may produce slightly different results compared with the function on the CPU [22], and consequently cause the result difference for extended precision. Second, algorithms for most mathematical functions with extended precision do not have theoretical error bounds. For example, the last word in ARPREC is not reliable for some functions [1], such as the exponential function. Through our extensive tests, the result on the GPU is different from that on the CPU with up to the last two digits for mathematical functions, but the exactly same as that on the CPU for arithmetic operators. Moreover, the computation flow also affects the result value for a specific application.

4.2 Evaluation Results

4.2.1 Computation Performance

We first study the performance impact from the optimized memory layout. We show the comparison result of multi-digit format with the sequential and interval memory layouts. Figure 3 has shown the speedup of GPU-based implementations with the sequential and interval memory layouts for the exponential function in GARPREC compared with their CPU counterparts. Overall, the implementation employing the interval memory layout is around 3x faster than the sequential memory layout, and 8-12x faster than the CPU-based parallel implementation. The performance improvement from the interval memory layout is due to the coalesced access pattern adopted to utilize the high memory bandwidth on the GPU. Through our further evaluations, the real memory bandwidth with and without the coalesced access pattern on G280 are around 110 GB/sec and 25 GB/sec, respectively. Considering the computation and branch divergency that may offset the memory bandwidth improvement, the difference of bandwidth efficiency roughly agrees with the overall performance difference between the implementations with an interval and sequential memory layout on the GPU.

Table 1: Speedups of arithmetics and typical mathematical functions in GQD compared with their CPU counterparts.

	double-double	quad-double
+	20	16
×	21	11
/	16	12
sqrt	21	12
exp	27	13
log	27	13
sin	13	10

Table 2: Speedups of arithmetics and typical mathematical functions in GARPREC compared with their CPU counterparts.

	500 digits	1000 digits	1500 digits	2000 digits
+	12	12	12	11
×	11	9	8	8
/	13	10	10	10
sqrt	13	11	10	10
exp	12	9	8	7
log	12	10	9	9
sin	8	7	7	7

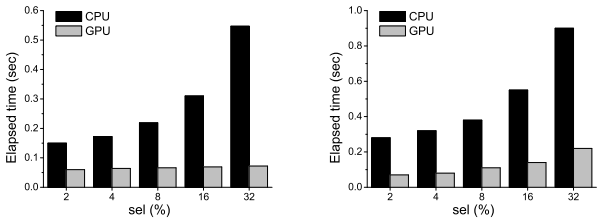
Table 1 and 2 have summarized speedups of arithmetic operators and typical mathematical functions in GQD and GARPREC compared with their CPU counterparts. The computation with double-double and quad-double precision in GQD achieves the speedup of up to 27 and 16 times, respectively. Arithmetics and mathematical functions in GARPREC are 7-13x faster than their CPU counterparts. Since the algorithm of multi-digit is more complex than that of multi-term, it may decrease the speedup of GPUs, e.g., more statement branches can hurt the overall performance due to the SIMD feature of GPUs. Therefore, the speedup of GARPREC is less significant than that of GQD. Nevertheless, GPUs have been shown to accelerate the computation with extended precision greatly.

4.2.2 Database Operations

We have integrated extended precision algorithms into an existing GPU-based query processing engine [17] to support data management with extended precision. We show the performance of three basic database operations with double-double and quad-double precision: selection, sort, and non-indexed nested-loop join. Since aggregation is simply a parallel reduction or summation, we show its performance in

the context of user-defined aggregation functions in scientific computing in Section 5.

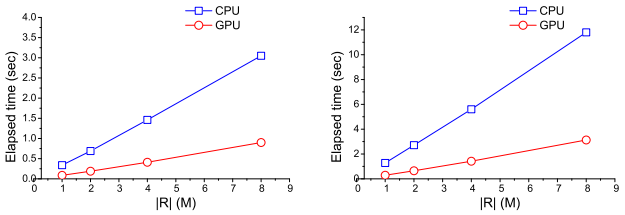
Selection. We consider the range selection with selectivity varied to study the selection performance with extended precision. The relation is fixed to 16 million tuples. Figure 4 shows the GPU-based selection with extended precision is around 2-8x faster than their CPU counterparts.



(a) double-double (b) quad-double

Figure 4: Performance comparison of selection with double-double and quad-double precision on the GPU and CPU when the selectivity is varied.

Sort. We have integrated the multi-term format to the GPU-based bitonic sort. The sort on the CPU is a parallel quick sort implemented with OpenMP. Figure 5 illustrates the elapsed time of GPU- and CPU-based sort algorithms with extended precision when the relation size is varied. Our GPU-based sort outperforms the quick sort on the CPU by up to 3.1 and 3.8 times for the double-double and quad-double precision, respectively.



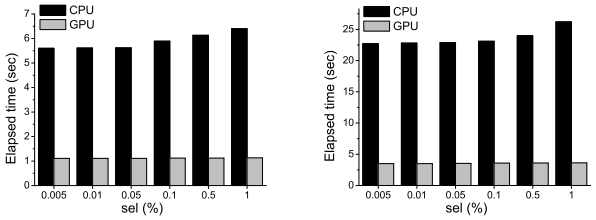
(a) double-double (b) quad-double

Figure 5: Performance comparison of sort with double-double and quad-double precision on the GPU and CPU when |R| is varied, where |R| is the number of tuples.

NINLJ. For two given relations R and S, we investigate the non-equijoin $R.val \leq S.val \leq R.val+\delta$, where δ is varied in our evaluations to examine the performance with different join selectivity values. R and S are fixed to 32,000 tuples. Figure 6 shows the performance of NINLJ with extended precision on the GPU and CPU. The GPU-based NINLJ is around 6-7x faster than its parallel CPU counterpart.

Overall, the speedups of three operations are similar to or slightly higher than the comparison result with native precision reported by He et al. [18], which shows the GPU has a similar capability to accelerate database operations with extended precision. This indicates the effectiveness of our optimization techniques for extended precision.

5. APPLICATIONS



(a) double-double (b) quad-double

Figure 6: Performance comparison of NINLJ with double-double and quad-double precision on the GPU and CPU when the selectivity is varied.

In this section, we demonstrate three real-world scientific applications that require extended precision support. We compare the performance and accuracy of GPU-based implementations with their CPU counterparts. We use the experimental setup the same as that in Section 4. All CPU implementations are also developed using OpenMP to unlock the power of the multi-core CPU.

5.1 Parallel Summation in Climate Modeling

In this application, the multi-term extended precision is used to improve the numerical reproducibility and stability in scientific computing. He et al. have identified the numerical reproducibility issue raising in a parallel climate modeling simulation [19]. They have observed that the same simulation performed on different numbers of processors produced very different results. They further found that the major error was generated from a global summation. Although only two decimal digits are required for final results in this application, the double precision is insufficient to provide the correct result due to large rounding errors. The authors have addressed this issue by adopting the double-double precision in the QD library.

As an example, we adopt the data of sea surface height (SSH) including 7680 floating point numbers used in the previous study of climate modeling [19] to demonstrate the accuracy and performance of our GQD library. We focus on the summation step that may produce incorrect results when double precision is used. The summation is implemented as a user-defined aggregate function *sum* in the GPU-based query processing engine with extended precision. The correct result of summation for this data set is 0.35798583924770355. Table 3 shows summation results employing different levels of precision on the GPU and CPU. Either the GPU or the CPU produces an incorrect result when only the native double precision is used. In contrast, double-double precision is sufficient for this application since both the GPU and the CPU have generated the correct result.

To study the performance, we have randomly sampled the original SSH data set to generate a new large data set in order to scale the data size. We demonstrate the performance of summation employing double-double precision only since it is sufficient to produce correct results. Figure 7 shows the elapsed time of the CPU- and GPU-based summation with double-double precision as a function of the number of input elements. The GPU-based implementation significantly outperforms the CPU-based parallel implementation, with a peak speedup of around 10x.

Table 3: Results of summation employing different levels of precision on the GPU and CPU. The correct value is 0.35798583924770355.

	CPU	GPU
Native double	34.414768218994141	0.2500000000000000
Double-double	0.35798583924770355	0.35798583924770355
Quad-double	0.35798583924770355	0.35798583924770355

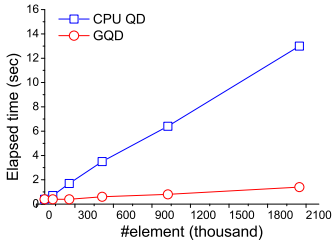


Figure 7: Performance comparison of the summation with double-double precision on the GPU and CPU when the number of elements is varied.

5.2 Numerical Analysis of Nonlinear Physics

In this application, one of the most fundamental numerical analysis algorithms, namely *Newton's method*, is used to find an approximation for a given function [5]. The quad-double precision is adopted to help scientists find a closed-form formula based on the 42-digit result. Without the extended precision, the analytic solution would be difficult to identify and validate.

As an emerging science, the spontaneous synchronization of oscillators usually occurs in the natural world, such as the chorusing of crickets and the synchronous applause of concert audiences. Recently, Quinn et al. [25] have studied this process using a summation expression shown in Equation 1.

$$0 = \sum_{i=1}^{N} (2\sqrt{1 - s_N^2(1 - 2\frac{i-1}{N-1})^2} - \frac{1}{\sqrt{1 - s_N^2(1 - 2\frac{i-1}{N-1})^2}}) \quad (1)$$

In Equation 1, N is the population size being considered in the model, and s_N is the N-dependent equation root representing how far in or out of synchronization a group is. Consequently, s_N is a crucial parameter that can be used to study and understand the model [25]. After obtaining solutions of Equation 1 for different values of N, the scientists further found that $s_N \sim 1 - c_1 N^{-1}$, where c_1 is a constant with the value 0.6054436..., which is known as the QRS constant. Scientists wanted to find an analytic formula to represent this constant. Bailey et al. have solved this problem through a method of experimental mathematics [5]. As the first effort, they attempted to obtain a 42-digit value to help identify the analytic solution. They developed the algorithm on 64 CPUs using the quad-double precision in QD to calculate this 42-digit QRS constant.

We use the GPU-based library to accelerate the computation of this 42-digit QRS constant. The major time-consuming step is to apply Newton's method to Equation 1 with N varied until two successive values are differed no more than 10^{-52} [5]. Specifically, 15 values of N are used, where $N = 4^m$ and m is ranged from 1 to 15. Newton's method is expensive in this application since N can be very large. We focus on using the GPU to improve the performance of Newton's method for Equation 1 when N is large.

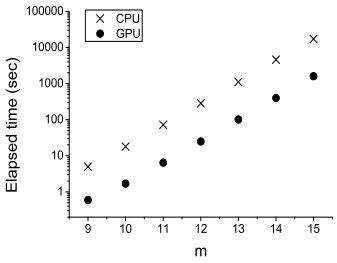


Figure 8: Performance comparison of Newton's method for the QRS constant computation on the GPU and CPU with N varied, where $N = 4^m$.

We have developed the GPU-based Newton's method with quad-double precision. The main components of the algorithm are implemented as a data parallel *map* and *reduction*. The *map* is used to compute each item of the summation in Equation 1, and the *reduction* is applied to obtain the sum of all items. Translated into database operations, the evaluation of Equation 1 is done through a selection with the predicate on the range of the iterator and a user-defined aggregation for the summation. With the required accuracy, the result of Newton's method on the GPU is the exactly same as that on the CPU for a given N. Additionally, our parallel CPU-based implementation is 3.2x faster than the sequential one.

Figure 8 shows the elapsed time of the GPU- and the CPU-based Newton's method for the QRS constant computation with N varied. The GPU-based implementation is 8-12x faster than the CPU counterpart. For all calculations, when m is varied from 1 to 15 in this application, our GPU-based implementation takes about half an hour in total. Bailey et al. have reported that their implementation based on 64 CPUs required 25 minutes to perform the same computation task [5]. The same result with the similar elapsed time is obtained, but only one GPU is used in our evaluation, which indicates the efficiency of our GPU-based implementation.

5.3 High Precision Numerical Integration

High precision numerical integration (or called *quadrature* in several literatures) [7] is a typical application in experimental mathematics employing arbitrary precision. Without extended precision support, this application cannot be handled by modern processors. Combined with the integer relation detection algorithm [2], it can be used to discover previously unknown analytic solutions for integrals.

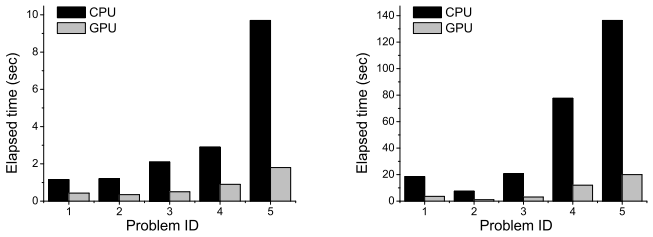
Tanh-sinh is an effective algorithm for high precision numerical integration [7]. There are several computation levels to perform function evaluations corresponding to a group of abscissa-weight pairs. The algorithm is done when the estimated accuracy is met or the defined number of levels is exhausted. Bailey et al. [4] have implemented the parallel tanh-sinh algorithm various integrals, and achieved significant performance speedup. We implement the GPU-based tanh-sinh algorithm using GARPREC to support arbitrary

precision. Function evaluation for a large number of input elements is a data parallel primitive *map* on the GPU, which corresponds to a *selection* in the database engine.

Following the previous study [7], we evaluate the GPU-based numerical integration using two different precision levels: 400, and 1000 decimal digits, respectively. We have excluded the time to generate the abscissa-weight table since it can be initialized only once and is used for any problems for a given precision. Bailey et al. have categorized different problems of numerical integration that they have encountered into five types [7]. We choose the most expensive problem in each category for our evaluations. These five problems are summarized in Table 4. When the target accuracy is achieved, all of our evaluated problems on both the GPU and the CPU consume the exactly same number of computation levels as shown in the previous work. Moreover, our OpenMP-based CPU implementation is 2.5-3.3x faster than the sequential implementation included in ARPREC.

Table 4: Test sets of the numerical integration.

ID	Problem	Solution
1	$\int_0^{\pi/2} e^t \cos t dt$	$(e^{\pi/2}-1)/2$
2	$\int_0^1 \sqrt{t} \log t dt$	$-4/9$
3	$\int_0^{\pi/2} \log(\cos t) dt$	$-\pi log 2/2$
4	$\int_0^{\infty} e^{-t^2/2} dt$	$\sqrt{\pi/2}$
5	$\int_0^{\infty} e^{-t} \cos t dt$	$1/2$



(a) #digit = 400 (b) #digit=1000

Figure 9: Performance comparison of high precision numerical integration on the GPU and CPU for the five problems when the precision is 400 and 1000 decimal digits.

Figure 9 shows the elapsed time of GPU-based numerical integration compared with the CPU counterpart for the 400- and 1000-digit precision. Overall, the GPU-based implementation is 3-9x faster than the parallel CPU-based implementation.

6. CONCLUSION

As the GPU becomes a promising parallel platform for scientific computing, we implement a GPU-based high-performance extended precision library to enable numeric applications that require a high precision for GPU-acceleration. Our library consists of basic arithmetic operators, mathematic functions, as well as data-parallel primitives with extended precision. It supports both multi-term and multi-digit extended precision formats. In addition, to support scientific data management on the GPU, we have integrated our extended precision library to a GPU-based query processing engine. The experimental study shows that our library on G280 provides a performance speedup of up to an orders of magnitude over existing CPU-based extended precision libraries on a quad-core CPU and has a similar performance improvement to native precision on the GPU for database operations. Moreover, we have also implemented three real-world scientific applications employing extended precision to demonstrate the efficiency of our library. Compared with the CPU-based parallel implementation, the GPU-based implementation achieves the same accuracy requirement and a speedup of 3-12x.

The code base and report are maintained at: http://code.google.com/p/gpuprec/ .

Acknowledgement

We thank the anonymous reviewers for their comments on the paper. This work was supported by Grant 616808 from the Hong Kong Research Grants Council.

7. REFERENCES

[1] D. H. Bailey. Algorithm 719: Multiprecision translation and execution of fortran programs. *ACM Trans. Math. Softw.*, 19(3):288–319, 1993.

[2] D. H. Bailey. Integer relation detection. *Computing in Science and Engineering*, 2:24–28, 2000.

[3] D. H. Bailey. High-precision floating-point arithmetic in scientific computation. *Computing in Science and Engg.*, 7(3):54–61, 2005.

[4] D. H. Bailey and J. M. Borwein. Highly parallel, high-precision numerical integration. Technical Report LBNL-57491, April 2008.

[5] D. H. Bailey, J. M. Borwein, and R. E. Crandall. Resolution of the quinn-rand-strogatz constant of nonlinear physics. *Experimental Mathematics*, 18:107–116, 2009.

[6] D. H. Bailey, Y. Hida, X. S. Li, and O. Thompson. ARPREC: An arbitrary precision computation package. Technical Report LBNL-53651, 2002.

[7] D. H. Bailey, K. Jeyabalan, and X. S. Li. A comparison of three high-precision quadrature schemes. *Experimental Mathematics*, 14:317–329, 2004.

[8] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of SciDB: a science-oriented DBMS. In *VLDB*, 2009.

[9] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971.

[10] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo. Frequent itemset mining on graphics processors. In *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware*, 2009.

[11] GNU Multiple Precision Arithmetic Library. http://gmplib.org/.

[12] D. Göddeke, R. Strzodka, and S. Turek. Accelerating double precision FEM simulations with GPUs. In *18th Symposium on Simulations Technique (ASIM'05)*, 2005.

[13] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, 2004.

[14] G. D. Gracca and D. Defour. Implementation of float-float operators on graphics hardware. In *7th conference on Real Numbers and Computers*, 2006.

[15] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Rec.*, 34(4):34–41, 2005.

[16] J. Gunnels, J. Lee, and S. Margulies. Efficient high-precision dense matrix algebra on parallel architectures for nonlinear discrete optimization. Technical Report IBM Research RC24682, 2008.

[17] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):1–39, 2009.

[18] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, 2008.

[19] Y. He and C. H. Q. Ding. Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. *J. Supercomput.*, 18(3):259–277, 2001.

[20] Y. Hida, X. Li, and D. Bailey. Algorithms for quad-double precision floating point arithmetic. *IEEE Symposium on Computer Arithmetic*, pages 155–162, 2001.

[21] G. Lake, T. Quinn, and D. C. Richardson. From sir isaac to the sloan survey: calculating the structure and chaos owing to gravity in the universe. In *SODA*, 1997.

[22] NVIDIA CUDA. *http://developer.nvidia.com/object/cuda.html*.

[23] OpenMP. *http://openmp.org/*.

[24] J. D. Owens, D. Luebke, N. K. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, 2005.

[25] D. D. Quinn, R. H. Rand, and S. H. Strogatz. Singular unlocking transition in the Winfree model of coupled oscillators. *Physical Review*, 75(3):036218–+, 2007.

[26] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18:305–363, 1997.

[27] A. Thall. Extended-precision floating-point numbers for GPU computation. In *ACM SIGGRAPH Research posters*, 2006.

[28] XBLAS - Extra Precise Basic Linear Algebra Subroutines. http://www.netlib.org/xblas/.