



Memory latency optimizations for the elementary functions on the Sunway architecture

Bei Zhou¹ · Yongzhong Huang² · Jinchen Xu¹ · Shaozhong Guo¹ · Hongyuan Qi¹

Published online: 22 January 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

As fundamental software of high-performance computers, elementary functions have a significant impact on the performance of the high-level applications. Benefiting from the Chinese-designed manycore system consisting of processing cores and auxiliary cores, the Sunway TaihuLight supercomputer is considered as one of the fastest supercomputers in the world, having ranked on the top of the TOP500 supercomputer list several times. The processing cores of the Sunway architecture are coupled using a shared memory strategy, leading to high latency of memory accesses and performance degradation of the elementary functions where a variety of memory accesses exist. To address this issue, we propose a set of optimizations for memory latency of the Sunway processing cores. Firstly, we obtain a reduced data table in the context of guaranteed accuracy by optimizing underlying algorithms, grouping and mapping, removing error compensations, etc. Secondly, we perform data movement from the global memory shared by all processing cores to the scratchpad memory of individual processing cores, significantly reducing the memory latency. Finally, we convert the memory accesses that cannot be localized due to the limited space of the scratchpad memory into equivalent immediate loads and/or shift operators, further improving the performance. In addition, we automate the algorithm by carefully selecting the most suitable data conversion approach and table-lookup algorithm, mitigating the code explosion issue effectively. We implement our method and evaluate the effectiveness of the optimizations by conducting experiments on the Sunway architecture. The experimental results show that exponential functions can achieve performance improvements by 91 and 86.2% from the data movement and data conversion strategies.

Keywords Exponential functions · Memory latency optimization · Scratchpad memory · Data reduction · Data movement · Data conversion

✉ Yongzhong Huang
2389483289@qq.com

Extended author information available on the last page of the article

1 Introduction

Elementary functions are the fundamental software of high-performance computers, being integrated into the standard library of a large number of programming languages and usually tightly coupled with the dedicated processors of the manufacturers. For example, the two CPU giants, Intel and AMD, have both released their own dedicated numerical libraries for fully exploiting the performance of the processors. Since the Sunway TaihuLight supercomputer peaked the TOP500 list in 2016, the 260-core SW26010 manycore processor composed of 256 processing cores and 4 auxiliary cores has been one of the hot topics in the realm of high-performance computing. Similarly, the Sunway architecture also needs a dedicated library of mathematical functions, but the performance of the existing numerical library of the SW26010 manycore processor is still far from expectation, calling for a fully optimization of the library on the architecture.

Transcendental functions like trigonometric functions, exponential functions, logarithmic functions, hyperbolic functions constitute the majority of a numerical library, typically implemented by the reduction–approximation–reconstruction approach which is supposed as being of near-perfect accuracy and high performance. By combining the table-lookup algorithm and Taylor series, the method caches the intermediate results into a table, which in turn is retrieved by memory access operations, improving the efficiency while not hampering the accuracy. As a result, memory access operations are at the core of the reduction–approximation–reconstruction algorithm, implying optimizing memory accesses would benefit the performance of elementary functions significantly.

SW26010 has four clusters of 64 processing cores plus 1 auxiliary core. A processing core deviates from an auxiliary core in terms of instruction sets, register usage, instruction latency, etc. The memory access latency on auxiliary cores is close to that of elementary operations, usually varying between 3 and 5 cycles. On the contrary, the processing cores are coupled using a shared memory manner for accessing the main memory, leading to a much higher memory access latency by 100 cycles or more. A surprising result of such strategy is the heavy performance degradation of the reduction–approximation–reconstruction algorithm suffered on the Sunway processing cores, greatly different from that of the same implementation on auxiliary cores. For example, the *sin* function takes 1360 cycles when executed on the Sunway processing cores, while it only needs 90 cycles on auxiliary cores. The intolerable memory access latency is therefore at the core of performance degradation of elementary functions on the Sunway processing cores; worse yet, almost all of the computation tasks from the high-level applications are executed on the Sunway processing cores, meaning that the performance of elementary functions on the Sunway processing cores may determine the performance of the entire architecture.

Instruction scheduling is a typical optimization for memory access latency, improving the performance by hiding the latter behind the execution of computation. On the one hand, the improvement of this strategy is still far from the optimal performance since instruction scheduling is heavily influenced by the

implementation of the target functions and the data dependences between the functions. On the other hand, the memory access latency is usually more than 100 cycles on the Sunway processing cores, making the accesses cannot always be overlapped with computations and hereby calling for new optimization techniques for tackling with this issue.

We introduce a set of optimizations for the memory latency of elementary functions on the processing cores of SW26010. To fully benefit from the high-speed but limited scratchpad memory of the Sunway processing cores, we first reduce the size of the cached data for the target functions and obtain a reduced data table in the context of guaranteed accuracy by optimizing underlying algorithms, grouping and mapping, removing error compensations, etc. We perform data movement from the global memory to the scratchpad memory when the reduced data do not exceed the size of the scratchpad memory, minimizing the memory access latency on processing cores. Otherwise, we convert the memory access operations into equivalent but more efficient instructions like immediate loads and/or shift operators, thereby achieving data conversion. With regard to the side effect caused by data conversion, i.e., code explosion, we also propose an effective, automatic data conversion approach and a table-lookup method to reduce cache misses. We implement the presented optimization techniques and evaluate their effectiveness by conducting experiments on the Sunway architecture. The experimental results show that exponential functions used in the evaluation can achieve performance improvements by 91 and 86.2% from the data movement and data conversion strategies.

While addressing the inefficiency issue of the elementary functions on the processing cores of the Sunway architecture, our optimizations also open the door for further improving the performance of the elementary functions on supercomputers. The main contributions of our method are as follows.

- We propose an optimizing approach with the portability to different accuracy and performance requirements.
- We present an automatic, easy-to-use data conversion technique for optimizing the memory access latency of elementary functions.
- We validate our technique on the Sunway architecture, followed by a discussion of the portability to those beyond the SW26010 manycore processor.

The paper is organized as follows. The next section explains the existing implementation of transcendental functions on the processing cores of the Sunway architecture and its limitation. Section 3 introduces our method, followed by the experimental results in Sect. 4 and related work in Sect. 5. Concluding remarks are presented at the end of the paper.

2 Background

2.1 The reduction–approximation–reconstruction algorithm

Transcendental functions like trigonometric functions, exponential functions, logarithmic functions can only be approximated by elementary operations, i.e., the approximation approach, e.g., Taylor series, table-lookup [1–4] and CORDIC. Tang [5–7] proposed a high-performance implementation of transcendental functions by integrating the Taylor series and table-lookup algorithms, achieving the tradeoff between the two approaches. The main idea of the implementation could be summarized as follows.

Given a function $f(x)$, the implementation first setups breakpoints by extracting N points from the input, each of which is represented as $C_k (1 \leq k \leq N)$. The value of $f(x)$ at each breakpoint C_k is stored in the data table, represented using T_k . As a result, the $f(x)$ function could be approximated by (1) reduction, mapping any input x to its closest breakpoint $C_x (1 \leq x \leq N)$, with the result represented as $r = R(x, C_x)$ where R represents the reduction function; (2) approximation, approximating $f(r)$ using a given approximation function $p(r)$; and (3) reconstruction, reconstructing $f(x)$ using a well-defined reconstruction function S by taking $f(C_x), f(r), T_x$ and $p(r)$ as input.

While keeping itself away from the pointwise convergence of Taylor series by reducing a given input to a fast-convergence interval, the reduction–approximation–reconstruction algorithm also enhances the accuracy of the approximation of a given function. By combining the benefits of Taylor series and table-lookup algorithms, the reduction–approximation–reconstruction approach is equipped with a high-accuracy, high-performance property, widely used to approximate elementary functions.

2.2 The SW26010 processor

SW26010 is the manycore processor used on the Sunway TaihuLight supercomputer, implementing the Sunway architecture by coupling four clusters of 64 processing cores and a single auxiliary core, with each cluster equipped with an 8 GB global memory space. A processing core can access the global memory space randomly either in the *gld/gst* manner or in batch via its scratchpad memory. The scratchpad memory could also be inspected as local data memory, represented as LDM throughout the figures of this paper. Each processing core features 64 kB of scratchpad memory for data and 16 kB for instructions. Figure 1 depicts the architecture of one cluster of the SW26010 processor, while Table 1 lists the memory access latency of the processing cores and auxiliary cores.

2.3 Limitations of the implementation on the Sunway processing cores

On Sunway architecture, the performance of the reduction–approximation–reconstruction algorithm varies with its execution on different cores. The

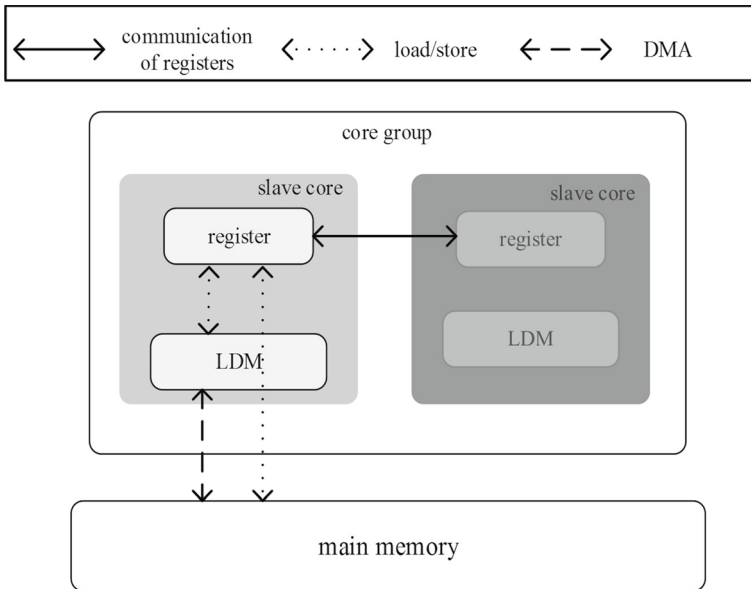


Fig. 1 The sketch of one cluster of the SW26010 processor (slave core has the same meaning as processing core)

Table 1 A summary of the access latency of SW26010 processor

Content	Cycle
Auxiliary cores to global memory	4
Processing cores to global memory	177
Processing cores to scratchpad memory	4

implementation of the algorithm can achieve high accuracy and high performance on auxiliary cores but suffers from a dramatic performance degradation on processing cores. The performance comparison of some representative elementary functions on auxiliary and processing cores is shown in Fig. 2.

We take the *pow* function as an example for illustrative purpose. The function may take 146 cycles on auxiliary cores, while it goes up to 2616 cycles on processing cores, making the execution of the function on the processing cores impractical. Such conclusion can also apply to the remaining functions shown in Fig. 2. While the execution latency of elementary operations stays the same on both kinds of cores, we may conclude that the memory access latency should be the main source of performance degradation on processing cores, which can also be validated from our experimental results. As a result, we argue that optimizing the memory access latency on the Sunway processing cores may be essential for improving the performance of elementary functions on Sunway architecture.

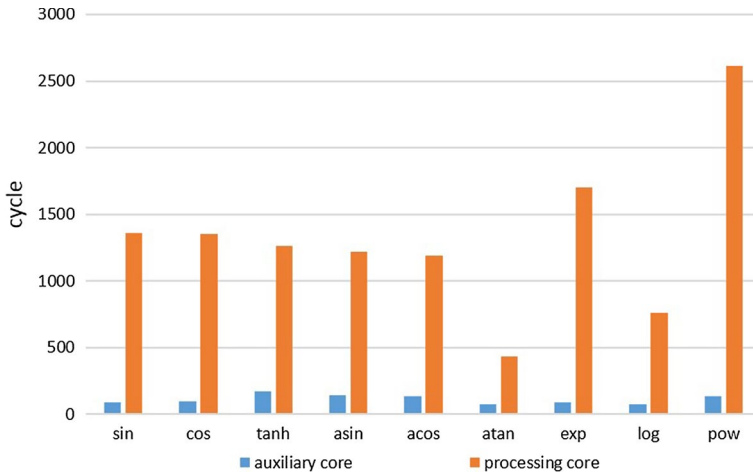


Fig. 2 Performance comparison of some representative elementary functions on auxiliary and processing cores

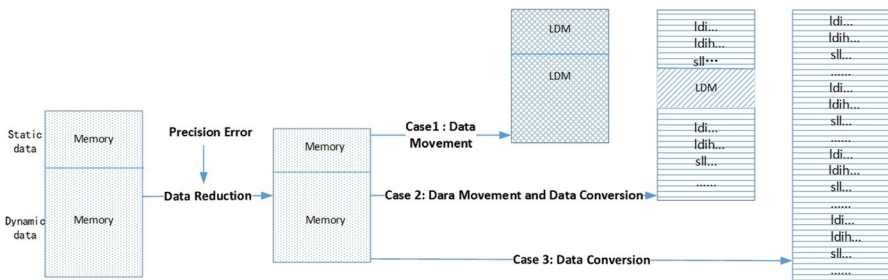


Fig. 3 An overview of our approach

3 Our approach

We first give an overview of our approach in Fig. 3. Generally speaking, our approach can be divided into three steps. In the first place, we are trying to reduce the size of the lookup table in the context of guaranteed accuracy by eliminating redundant data, optimizing underlying algorithms, grouping and mapping, removing error compensations, etc. In the second place, we transform global memory accesses into local scratchpad memory accesses by localizing the data from the main memory. Last, we convert the memory accesses that cannot be localized due to the limited space of the scratchpad memory into equivalent immediate loads and/or shift operators. Data movement and conversion could be used independently or in conjunction, with each case depicted in Fig. 3. We would introduce each case in the following context.

An alternative way of the Sunway processing core to access data in the global memory is to resort to the scratchpad memory or loading the data on the global

memory to its scratchpad memory via so-called DMA instructions, both expecting for a plenty space of scratchpad memory for storing the lookup table. The total size of lookup table may greatly exceed the scratchpad memory size, i.e., 64 kB, since the size of a single lookup table for each elementary function usually varies between 4 and 8 kB. Worse yet, the very limited space of the scratchpad memory is always reserved to the high-level applications, making the availability of the scratchpad memory for system software like numerical library an undeterminable problem. Meanwhile, it is impractical and impossible to store the entire lookup table for all elementary functions into the scratchpad memory, meaning that we have to reduce the size of the lookup table first. As a result of the localization of the reduced data tables, one may reduce the memory latency of elementary functions on the processing cores or resort to immediate loads and/or shift operators for equivalent instructions substitution.

In practice, we are allowed to handle the functions with the same lookup table in a similar way. For example, the elementary functions including *expm1*, *exp2*, *pow*, *sinh*, *cosh*, *tanh*, *erf*, *erfc*, etc. have the same lookup table with the *exp* function, we can therefore group all of these functions as exponential functions and handle them in the same way. In the following context of the section, we will take the exponential functions as example for illustrating our approach.

3.1 Data reduction

As explained before, we first need to reduce the size of lookup tables. The data stored in the lookup tables of elementary functions can generally be divided into two groups, with one representing the constants used in computation, e.g., the constant $1/\ln 2$ of the *exp* function, referred to as static data, and the other for dynamic data, i.e., the intermediate results caused by different inputs. While static data are insensitive to the input of a function, the size of dynamic data stored in lookup tables varies significantly to the access pattern of the input. We mainly focus on dynamic data as it dominates the lookup table of an elementary function.

Data reduction may also be divided into two categories according to its impacts on the accuracy of elementary functions. The first category minimizes lookup tables by eliminating redundant data and enhancing data reuse, usually supposed as inefficient although being side-effect free on the accuracy of elementary functions. On the contrary, the second category is recognized as more efficient since this kind of approaches updates lookup tables by recomputing, equivalent reasoning, algorithmic reforming, etc., even though sacrificing the accuracy to some extent.

Generally speaking, performance usually comes before accuracy from the view of users, especially for high-performance computers like the Sunway TaihuLight supercomputer, and approaches achieving high performance by sacrificing accuracy are therefore widely used in the field of high-performance numerical library, with introduced errors constrained to 3–5 ulp (unit in the last place) when comparing with MPFR. We therefore use a maximum allowance of 3 ulp in our method, minimizing the side effect caused by such kind of approaches by forcing the reduction to be invalid when exceeding such maximum allowance.

3.1.1 Analyzing lookup tables

According to the description of Tang [5–7], we may summarize the implementation of the exp function as below.

1. Given an integer L such that $L \geq 1$. To reduce an input x to r where r lies in the interval $[-\log \frac{2}{2^{L+1}}, \log \frac{2}{2^{L+1}}]$, i.e., $r \in [-\log \frac{2}{2^{L+1}}, \log \frac{2}{2^{L+1}}]$, we can let x be equal to $(m2^L + j)\log \frac{2}{2^L} + r$ with m and j being integers such that $j = 0, 1, 2, \dots, 2^L - 1$.
2. $exp(r)$ can be approximated by resorting to the polynomial expansion technique. In other words, we can compute $exp(r) - 1$ using $exp(r) - 1 = p(r) = r + a_1 r^2 + a_2 r^3 + \dots + a_n r^{m+1}$.
3. The result of applying the exp function to x can be reconstructed using a given expression, that is, $exp(x) = 2^m(2^{\frac{j}{2^L}} + 2^{\frac{j}{2^L}} p(r))$.

One may conclude that the number of breakpoints and dynamic data varies with the integer L . More specifically, the converge rate of the polynomial expansion $p(r)$ increases when the value of the integer L raises, and the performance of the function will therefore be higher; it may also bring about the growth of the resulting data, hereby improving the accuracy. Due to the limited space of memory, one may have to choose an appropriate value for the integer L , not only for guaranteeing the performance and accuracy, but also for reducing memory space. There would be 512 dynamic data generated in the data table, one half for representing j of $2^{\frac{j}{256}}$ and the other for error compensation for the round error, when the integer L is set to 8.

3.1.2 Reduction of static data

Reducing the static data is straightforward. One may improve static data reuse by reusing a single read access rather than multiple read accesses, or substituting each occurrence of the accessing to the results of existing data with the computation of such existing data.

3.1.3 Reduction of dynamic data

On the contrary, reducing dynamic data is a bit complicated. We therefore propose three different strategies for this issue.

Strategy 1: Eliminating the error compensation

The number of dynamic data for the function exp would reach 512, with one half used for error compensation and constituting the final data table of the function. To obtain a reduced data table, one may eliminate such error compensations when the accuracy difference between the two results, one with error compensation and the other not, of the function can meet the requirement of the user.

Strategy 2: Reducing the number of breakpoints by tuning the reduction interval

Table 2 The relationship among L , the reduction interval and the number of breakpoints in the \exp function

Value of L	Reduction interval	Number of breakpoints
8	$[- 1/512, 1/512]$	256
7	$[- 1/256, 1/256]$	128
6	$[- 1/128, 1/128]$	64
5	$[- 1/64, 1/64]$	32

It is straightforward to conclude from the implementation of the \exp function that the reduction interval of the input x is also determined by the integer L , and we may summarize the relation in Table 2.

Still, one may decrease the value of the integer L to obtain a reduced data table. In addition, one should also tune the reduction interval by considering the accuracy of functions and the size of data table. For example, the value of the integer L should be set to 7 rather than 6 when the latter cannot guarantee the accuracy of functions but the former does.

Strategy 3: Grouping, mapping and computing

Part of dynamic data could be the result of the computation of some existing data. We still consider the \exp function as an example and suppose the integer L be set to 8. As a result, there would be 256 breakpoints corresponding to each variable j of 2^{256} ($j = 0, 1, 2, \dots, 255$), with each consecutive pair of variables differing from each other by $2^{\frac{1}{256}}$. An effective solution to obtain a reduced data table is grouping the data, with each group using a basis of the data to represent all the remaining data, i.e., mapping all the data of one group to this basis.

We illustrate the grouping and mapping strategy by explaining one concrete implementation that uses 64 groups. There would be 64 dynamic data in this implementation, meaning each four of the original 256 data are filtered into one group, reducing $\frac{3}{4}$ quarters of the original data size. We may set the first element of each group, $2^{\frac{j}{256}}$ ($j = 0, 4, 8, \dots, 252$) as the basis to represent the original data set composed of $2^{\frac{i}{256}}$ ($i = 0, 1, 2, \dots, 255$) and map the data $2^{\frac{j}{256}}$, $2^{\frac{j+1}{256}}$, $2^{\frac{j+2}{256}}$ and $2^{\frac{j+3}{256}}$ to the basis of each group j . The reasons why we choose the first element as the basis are twofold: On the one hand, each pair of the data differs by a factor of $2^{\frac{1}{256}}$; on the other hand, a multiplication operation is always preferred when compared with a division operation. One may obtain an approximation of each element in a group by multiplying the basis with $2^{\frac{1}{256}}$ for 0, 1, 2 or 3 times, respectively. The underlying principle of the grouping and mapping strategy is to compensate the loss of accuracy caused by such approach with computations, achieving high accuracy by sacrificing performance. As a result, we refer to this strategy as grouping, mapping and computing and implement it with the following steps for a given number of breakpoint *COUNT*.

Step 1 Grouping Partition the original data set into N groups, with each including $COUNT / N$ data.

Step 2 Mapping Select one element from each group as the basis by considering the effectiveness and performance of computing the remaining data element, and put it into the data table.

Step 3 Computing Compute an approximation for each reduced data element by taking into account the difference between the real value of this data element and the basis.

The size of the reduced data table and the number of multiplications are determined by the value of N , which in turn influences the performance and accuracy of functions. More specifically, increasing the value of N may bring about the raise of the size of data table and the decrease in the number of computations, thereby mitigating the influence on the performance and accuracy of functions; on the contrary, decreasing the value of N may result in the decline of the size of data table and the increase in the number of computations, hereby impacting the performance and accuracy more significantly. As a consequence, we have to make a tradeoff between the grouping strategy and the requirement of performance and accuracy of functions, by taking into consideration practical results. One may try to reduce the size of the data table by guaranteeing the accuracy and the performance to a maximum extent.

Remarks on the strategies

The proposed strategies may reduce the size of data table of the *exp* function and could be combined for high-performance purpose. We therefore summarize some remarks on these strategies in this subsection.

In the first place, Strategy 1 may reduce the half of the original size of data table and have a slight impact on the accuracy of functions. The performance improvement of Strategy 1 may be small scale due to the elimination of error compensation.

In the second place, Strategy 2 reduces the size of data table by optimizing the underlying algorithm. While this strategy may have no impact on performance, it may lead to a slight loss of accuracy.

In the third place, a grouping–mapping–computing–based approach is used in Strategy 3 to reduce the size of data table, which may result in the decrease in both performance and accuracy. As we explained in the previous section, one may have to choose the grouping strategy very carefully to guarantee the performance and accuracy.

In summary, all the strategies are trying to minimize the size of data table by considering the requirement of accuracy of the function. One may also use any combinations of such strategies in different cases.

We still use the *exp* function as an illustrative example. We could eliminate the error compensation of the function according to Strategy 1 and then reduce the size of data table by tuning the reduction interval according to Strategy 2, followed by an analysis of the possibility of further reducing the size of data table by means of the grouping approach in Strategy 3.

3.2 Localizing data on scratchpad memory

One may now localize the data on the scratchpad memory after obtaining a reduced data table. The space size of the scratchpad memory for elementary functions is determined by high-level applications, represented as LDM_SIZE . One may also specify the remaining available space to LDM_SIZE .

One may also have different versions of the elementary functions with a varying size specified to LDM_SIZE , like 1 kB, 2 kB, 4 kB, 8 kB. The data on the scratchpad memory could also be accessed by other functions. For example, functions like $exp2$, pow , $sinh$, $cosh$, $tanh$ may also access the data of the exp function that stored on the scratchpad memory, avoiding multiple loading operations for these functions.

We provide two manners for using the scratchpad memory of processing cores. One is the static manner, with which the memory space allocated to elementary functions on the scratchpad memory would not be released before the user application is finished; the other is the dynamic manner, with which the memory space on the scratchpad memory for elementary functions would be released when the function is returned.

Ideally, we expect the capacity of scratchpad memory may be large enough for loading the reduced data table, but it is usually impossible in practice due to the limited space. We may therefore have to convert the data in this case.

3.3 Data conversion

Data conversion refers to process of replacing the time-consuming memory access operations with equivalent but more efficient instructions like immediate loads and/or shift operators. A data table could be released when all its data have been converted.

One may obtain dynamic data by referencing their offsets in data table before data conversion, or by inspecting the instructions that access such dynamic data after data conversion. As a result, data conversion is facing challenges from two aspects, one replacing memory access operations with their equivalent instructions and the other seeking the corresponding piece of code for dynamic data.

3.3.1 Eliminating memory access operations

To illustrate how to eliminate memory access operations, we may first have to explain how a memory access operation is represented. Let $fdd R_i, offset(R_j)$ be a memory access operation, it refers to read the data at address (addressed by a register R_j plus an offset) and store it into a register R_i .

We may use the following instructions in the process of data conversion.

$ldi R_i, offset(R_j)$ is used to sign extend the 16-bit offset and assign the summation of the result and R_j to R_i .

$ldih\ R_i, offset(R_j)$ represents a similar process, with only a 16-bit left-shifting operation introduced to the 16-bit offset.

$sll\ R_j, \#a, R_i$ refers to shift the value stored in register R_j to left by $\#a$ bits and assign the result to register R_i .

Let D_i be a 64-bit double-precision number. To benefit from immediate loads, one may represent D_i using four components as $0xA_3A_2A_1A_0$ with each $A_j(j = 0, 1, 2, 3)$ represented using 16 bits, denoted in a 4-digit hexadecimal representation.

As a result, an instruction that attempts to read the number D_i may be converted into its equivalence by (1) loading the higher 32 bits, A_3A_2 , with ldi and $ldih$, (2) shifting the result to left by 32 bits with sll and (3) loading the lower 32 bits, A_1A_0 , with ldi and $ldih$.

There may be many different implementations for the above process in practice, but we only introduce two representative ones as below.

Implementation 1: Encoding an immediate value using the hexadecimal representation

We take a concrete hexadecimal number $0x3ff8123456752563$ as an example for illustrative purpose and show the code for data conversion and storing the result in register R_i in Fig. 4.

Notice that the higher 32 bits might be affected by sign extension operators when the value at the 32nd bit is 1. To illustrate our solution, we use another

Fig. 4 The code for data conversion of $0x3ff8123456752563$ using Implementation 1

```
ldih Ri,0x3ff8 (zero)
ldi Ri,0x1234 (Ri)
sll Ri,32, Ri
ldih Ri,0x5675 (Ri)
ldi Ri,0x2563 (Ri)
```

Fig. 5 The code for data conversion of $0x3fe3c21ff5156423$ using Implementation 1

```
ldih Ri,0x3fe3(zero)
ldi Ri,0xc21f(Ri)
sll Ri,32, Ri
ldih Ri,0x7515(Ri)
ldi Ri,0x6423(Ri)
mov l, Rj
sll Rj,31, Rj
or Rj, Ri, Ri
```

Fig. 6 The code for data conversion of $0x3fe3c21ff5156423$ using Implementation 2

```
ldi Ri, -15840(zero)
ldih Ri, 16356(Ri)
sll Ri, 32, Ri
ldi Ri, 25635(Ri)
ldih Ri, -2795(Ri)
```

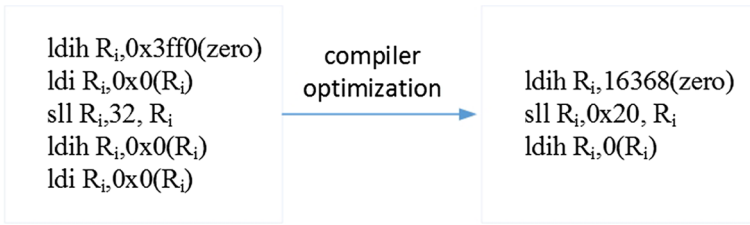


Fig. 7 The code for data conversion of 0x3ff0000000000000 after compiler optimizations

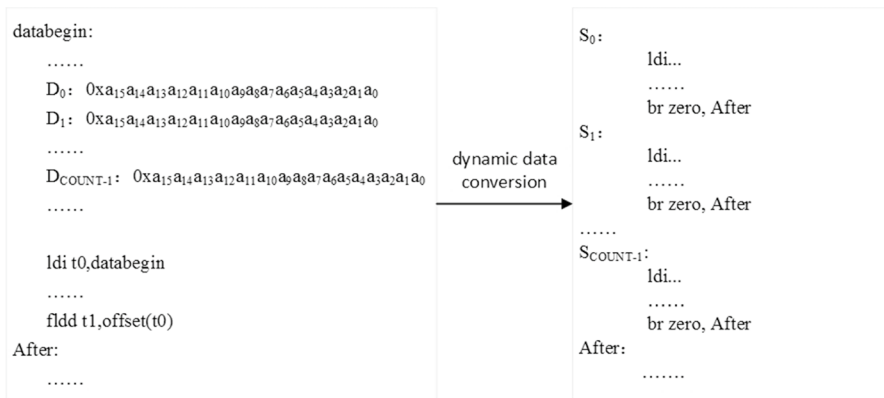


Fig. 8 The code before and after dynamic data conversion

hexadecimal number 0x3fe3c21ff5156423 as an example, and the code for data conversion in this case is shown in Fig. 5.

Implementation 2: Encoding an immediate value using the decimal representation

When encoding D_i in the hexadecimal representation, i.e., $0xA_3A_2A_1A_0$, each $A_j(j = 0, 1, 2, 3)$ could also be viewed as the complement of a decimal number, implying that we may also encode D_i using the decimal representation. However, one may have to encode D_i in the lower to higher order due to the presence of sign extension. In other words, A_{i+1} has to be first increased by 1 when the value at the leftmost bit of A_i is equal to 1, and then, it can be encoded in the decimal representation. We still use the number 0x3fe3c21ff5156423 as an example and obtain a set of decimal numbers $(25635)_{10}$, $(-2795)_{10}$, $(-15840)_{10}$ and $(16356)_{10}$, for the hexadecimal numbers, $(6423)_{16}$, $(f515)_{16}$, $(c220)_{16}$ and $(3fe4)_{16}$ used in the hexadecimal representation. The code of this implementation is shown in Fig. 6.

While the code size may vary when given different input numbers with Implementation 1 although it is easy to put into practice, the code size of Implementation 2 stays unchanged (5 instructions) but has to resort to decimal conversion. Unfortunately, the code size after data conversion would grow proportionally with the number of converted data, with either implementation. As a result, Implementation 2 is preferred when there is a strict limitation on the code size. Both

implementations can bring about performance improvement over the memory access-based implementation, since the instruction cycle of an immediate load and that of a logical shift would both be 1, meaning that Implementation 1 would take 5 to 8 cycles, while Implementation 2 would only take 5 cycles instead.

In addition, some redundant instruction of the two implementations may be eliminated by compilers under some specific input numbers, e.g., 0x3ff0000000000000. Figure 7 depicts the code after compiler optimizations for this example.

3.3.2 Code positioning

Each dynamic number $D_i (i = 0, 1, 2, \dots, COUNT - 1)$ would be replaced by a piece of code S_i after the data conversion step. The code before and after the replacement is shown in Fig. 8.

To read the value of this number, one may have to be aware of the offset of a dynamic data D_i in the data table when given the code before data conversion. This can be achieved by resorting to the *fdd* instruction. However, it may be non-straightforward if one attempts to position the corresponding piece of code, S_i , in the code after data conversion. To tackle with it, we abstract the issue as a searching problem and may resort to searching algorithms like the sequential search, the binary search, the block-matching search. In common, such searching algorithms would always try to find the given input by multiple attempts of comparing, but may differ in terms of the number of comparisons and their implementations. For example, the sequential search may return S_0 when it compares the value of i with 0 and proves the equality, or continues the comparing process by proceeding to the next value until it can prove the equality. Unlike the sequential search, the binary search would first compare the value of i with that of the number at $COUNT / 2$ and decide to proceed the comparing process in one of the two parts divided by $COUNT / 2$ according to the result of the first comparison. However, the comparison introduced by such searching algorithms may not only decline the performance of the functions after data conversion but also makes code explosion issue worse.

Suppose there exists a linear table, each of i th element represented as a_i . One may access a_i with the following expression

$$Loc(a_i) = Loc(a_0) + sizeof(a_i) * (i - 1) \quad (1)$$

where $sizeof(a_i)$ represents the memory size taken by a_i and $Loc(a_i)$ represents the address of a_i .

Similarly, one could find the location of S_i in the code by abstracting the latter as a linear table and S_i as an element, meaning the searching of S_i could be finished within $O(1)$ time. The only requirement for this abstraction is that S_i has to be stored sequentially which is a straightforward task. To achieve a constant memory size taken by each S_i , one may only have to ensure that there would the same number of instructions in each S_i since the memory size allocated to each instruction remains the same on the Sunway architecture, that is, 32 bits (4 bytes). We use M to represent the number of instructions in each S_i , and then, the address of S_i could be computed with

```

1. String Conversion_D_I(D) /* the return value is a code segment Si */
2. {
3.     i=0;
4.     while(i<4)
5.     {
6.         if (Ai's high bit is 1) {
7.             if (i<3) Ai-1 = Ai-1 + 1;
8.             Ai = (Ai - 0x1) ^ (0xffff);
9.         }
10.        //the function of Decimal_Conversion(Ai) is to get Ai's decimal representation
11.        Bi = Decimal_Conversion(Ai);
12.        i++;
13.    }
14.    String result="1di Ri, B2(zero)n";
15.    result=result + "1di Ri, B3(Ri)n";
16.    result=result + "s1 Ri, 32, Ri)n";
17.    result=result + "1di Ri, B0(Ri)n";
18.    result=result + "1di Ri, B1(Ri)n";
19.    return result;
20.}

```

Fig. 9 The code used for illustrating the automation of the data conversion

$$Loc(S_i) = Loc(S_0) + M * 4 * (i - 1) \quad (2)$$

where $Loc(S_0)$ and M are both constants. We refer to this method as quick search.

The benefits of quick search are twofold. On the one hand, it could mitigate the code explosion issue caused by the introduced comparisons of data conversion; on the other hand, it is also able to bring about performance improvement by decreasing the overhead of the searching algorithm.

However, the hypothesis of quick search is also a little strict. One should not only guarantee that the number of instructions M stays unchanged for each S_i but also have to minimize this number. As a result, Implementation 2 is preferred as there would always be 5 instructions plus an unconditional branch instruction, minimizing M by specifying it with a value of 6. One may use *NOP* as complementary instructions when the number of instructions is less than 6 due to compiler optimizations.

3.3.3 Automating the conversion

Both the elimination of memory access operations and the code positioning process could be automated, with the following steps.

1. Elimination of memory access operations. One may represent each D_i in the form of $0xA_3A_2A_1A_0$ with each $A_i (i = 0, 1, 2, 3)$ represented by a 4-digit hexadecimal

number. The automation of this process could be explained using the code shown in Fig. 9.

2. Code positioning. The position of the dynamic number D_i could be automatically computed using the following expression for a direct jump to the location of S_i .

$$Loc(S_i) = Loc(S_0) + 6 * 4 * (i - 1) \quad (3)$$

Note that we may introduce additional *NOP* instructions for guaranteeing the constant number of instructions in each S_i , for avoiding the impact of compilation optimizations.

3.3.4 Code explosion

The price we have to pay for converting data tables is the code explosion issue introduced when replacing memory access operations with immediate loads and/or shift operators. The number of instructions would rise up to $COUNT * M$ after the optimization, while there only exist $COUNT$ before the transformations. This may result in the performance degradation since code explosion may increase cache misses. To minimize the impact of the code explosion issue, we always perform the conversion of data tables after data reduction and data movement, since the latter two optimizations could minimize the number of instructions before the conversion optimization.

4 Evaluation

4.1 Experimental setup and methodology

To validate the effectiveness of our technique, we conduct experiments on the Sunway TaihuLight supercomputer comprised of 40,960 Chinese-designed SW26010 manycore 64-bit RISC processors based on the Sunway architecture. Equipped with a 16 kB L1 cache, each processing core runs at a clock speed of 1.5 GHz, peaking 12GFLOGS for double-precision floating-point operations and 13.5GIPS for single-precision fixed-point operations.

We conduct experiments on exponential functions to validate the effectiveness of the proposed optimizations, including data reduction, data movement and data conversion. More specifically, we validate the effectiveness of data reduction optimization by comparing the accuracy of the exponential functions before and after such optimization, with a maximum tolerance of 3 ulp for the error. We validate the effectiveness of the remaining optimizations by comparing the performance before and after these optimizations. We extract a random number from the usual input range of an exponential function as input and report the average speedup by running each function 400 times.

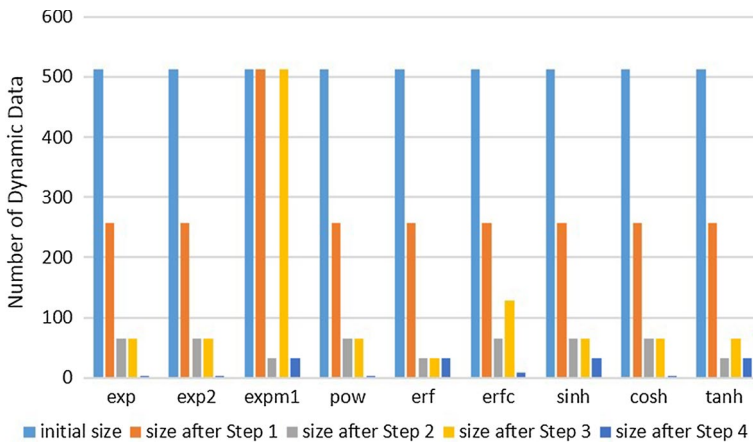


Fig. 10 The experimental results for reducing data tables

4.2 Experimental results of reducing data tables

We first conduct experiments to validate the effectiveness of data reduction including Strategy 1 (eliminating error compensations), Strategy 2 (reducing the number of breakpoints by tuning the reduction interval) and Strategy 3 (grouping, mapping and computing) with the following steps.

Step 1 Validate whether the introduced error is within 3 ulp by applying Strategy 1.

Step 2 Following Step 1, validate whether the accuracy of each function guarantees the predefined requirement by applying Strategy 2.

Step 3 Still following Step 1, validate whether the accuracy of each function guarantees the predefined requirement under each grouping strategy by applying Strategy 3.

Step 4 Validate whether it is possible to further reduce the data table of a function by synthesizing Strategy 1 to 3.

We select 21 million data by covering all the combinations of the sign bit and the exponent bits, and most combinations of the fraction bits, from the usual input range of each function, covering all the breakpoints of the original data table.

We show the experimental results of reducing data tables in Fig. 10 by following the above steps.

- We always have 512 dynamic, double-precision data for exponential functions. The *pow* function and *tanh* function may have additional dynamic data, but we only focus on those shared with other functions.
- With Step 1, we can reduce the data size by 50% except for the *expm1* function. The reason why the *expm1* function cannot be reduced is because the error compensation used in this function has a significant impact on the per-

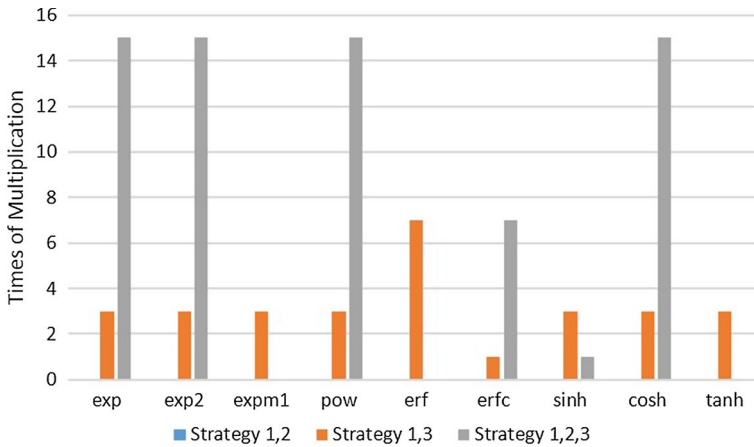


Fig. 11 Maximum number of multiplication from different reduction methods

formance of the function and we may not be able to eliminate the error compensation in this function.

- A variety of functions, including *exp*, *exp2*, *pow*, *erfc*, *sinh* and *cosh*, can benefit from Step 2 as the items can be reduced to 64, while the items can be reduced to 32 for the functions including *expm1*, *erf* and *tanh*.
- The reduction result of the grouping–mapping–computing–based strategy is diverse, with some functions including *exp*, *exp2*, *pow*, *erf*, *sinh* and *cosh* performing similarly to Step 2 and others including *expm1*, *erfc* and *tanh* falling behind. The reason why Step 3 has no impacts on the *expm1* function is because there is no geometric progression on the error compensation of this function.
- With Step 4, we may reduce the data size of functions *expm1*, *erf*, *sinh* and *tanh* to 32, while the size can be reduced to less than 10 for all the remaining functions.

Accordingly, we have the following conclusions.

- Each method of data reduction may have different effects when given different functions. The reason is because the *exp* function constitutes only part of some functions, leading to the variation of the accuracy of different functions. For example, Strategy 1 may be ineffective for the *expm1* function, but it can reduce 50% of the data table of the remaining functions.
- A combination of different strategies is always better than a single one, as can be validated by the experimental results.

As Strategy 3 may introduce additional computations and therefore hamper the performance improvement, one may have to make a tradeoff between the reduced result and introduced computations. Figure 11 shows the maximum numbers of introduced multiplication operations caused by these steps.

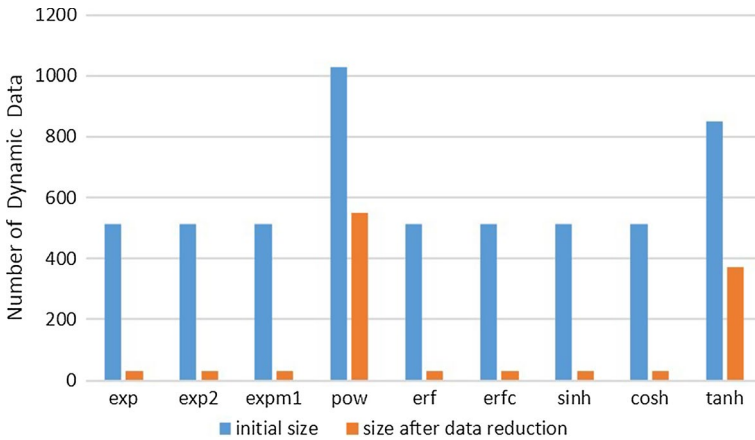


Fig. 12 The final result of reductions of exponential functions

Table 3 The numbers of dynamic data and the introduced multiplications

Function	Before data reduction			After data reduction		
	Data number	LDM (K)	Multiplication times	Data number	LDM (K)	Multiplication times
<i>exp</i>	512	4	0	32	0.25	1
<i>exp2</i>	512	4	0	32	0.25	1
<i>expm1</i>	512	4	0	16	0.25	0
<i>pow</i>	512 + 516	8	0	32 + 516	4.28	1
<i>erf</i>	512	4	0	32	0.25	0
<i>erfc</i>	512	4	0	32	0.25	1
<i>sinh</i>	512	4	0	32	0.25	1
<i>cosh</i>	512	4	0	32	0.25	1
<i>tanh</i>	512 + 339	7	0	32 + 339	2.90	0

While Strategy 3 may introduce multiplication operators, Strategy 1 and Strategy 2 would not have to pay for such price. There would be a large number of multiplications after synthesizing Strategy 1 to 3 although data reduction seems effective on some functions like *exp*, *exp2*, *pow*, *erfc* and *cosh*. As a consequence, one may have to make a tradeoff between the reduced data and the performance of functions, guaranteeing the latter by selecting the strategies that may introduce as few multiplications as possible, e.g., by setting the maximum number of introduced multiplications as 1 or 2. Accordingly, we have the results shown in Fig. 12, with a detailed analysis shown in Table 3.

As shown in Table 3, each function can benefit from data reduction, with the data of the *exp* part of each function reduced significantly. The number of data after reduction would not be greater than 32, much less than the number 512 before

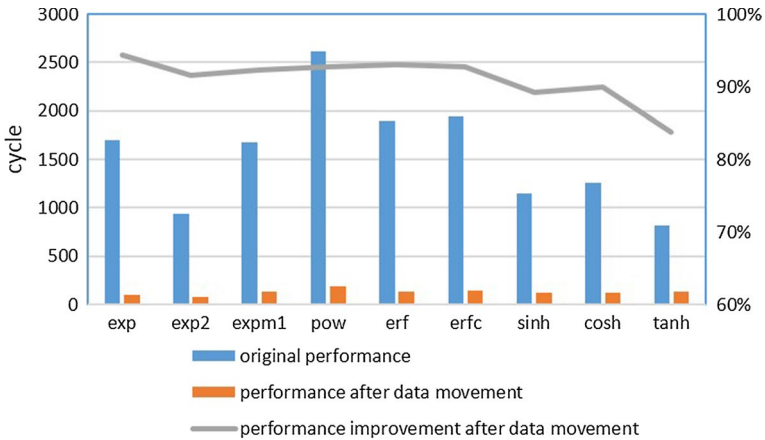


Fig. 13 Performance improvement of exponential functions after data movement

reduction, decreasing the memory footprint of the functions on the scratchpad memory from 4 to 0.25 kB. The experimental results validated the effectiveness of the data reduction optimization.

4.3 Performance of data movement

While the memory footprint of all the remaining functions was reduced from 4 to 0.25 kB by means of reduction optimization, it falls from 8 to 4.3 kB and 7 to 3 kB for the *pow* function and *tanh* function, respectively. One may localize these exponential functions on the scratchpad memory given that the size of the latter is 4 kB. Figure 13 shows the performance comparison before and after data movement.

As can be seen in Fig. 13, the performance of the exponential functions is obviously improved, with an average improvement by 91%.

4.4 Performance of data conversion

However, one may have to resort to the data conversion strategy when the scratchpad memory cannot hold the reduced data. As we introduced in the last section, data conversion may improve the performance by eliminating memory access operations, but it may bring about code explosion issue which in turn may hamper the performance improvement. We therefore conduct experiments by considering the performance improvement as well as the code explosion issue. More specifically, we conduct experiments by (1) comparing the performance of functions and code size under different searching algorithms used for code positioning, (2) comparing the performance of static data conversion and dynamic data conversion, and (3) comparing the performance of functions and code size before and after data reduction.

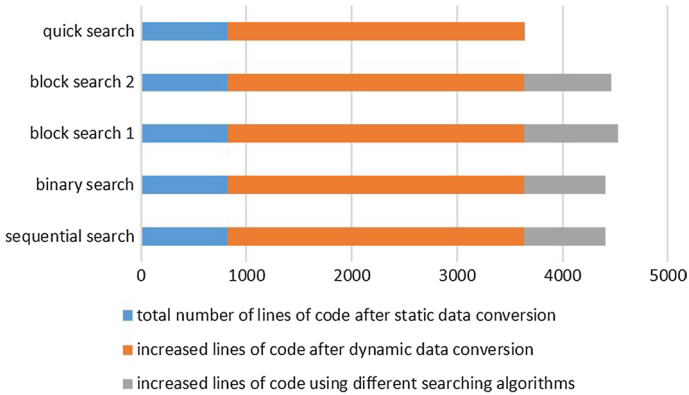


Fig. 14 Code explosion for the *exp* function when using different searching algorithms

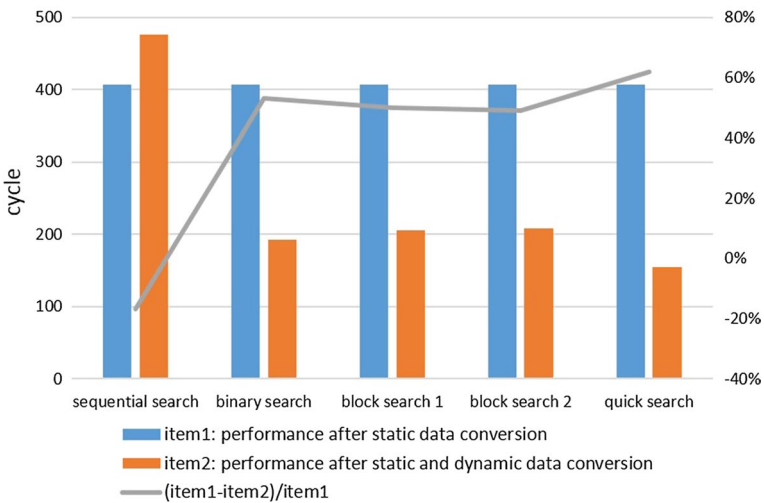


Fig. 15 Performance comparison of the *exp* function when using different searching algorithms

4.4.1 Performance of different searching algorithms

We take the *exp* function before data reduction as an example, and use different searching algorithms, including the sequential search, the binary search, the block-matching search with different implementations with one represented as block search 1 with 8 breakpoints in each block and the other represented as block search 2 with 16 breakpoints in each block, and the quick search, for code positioning. The

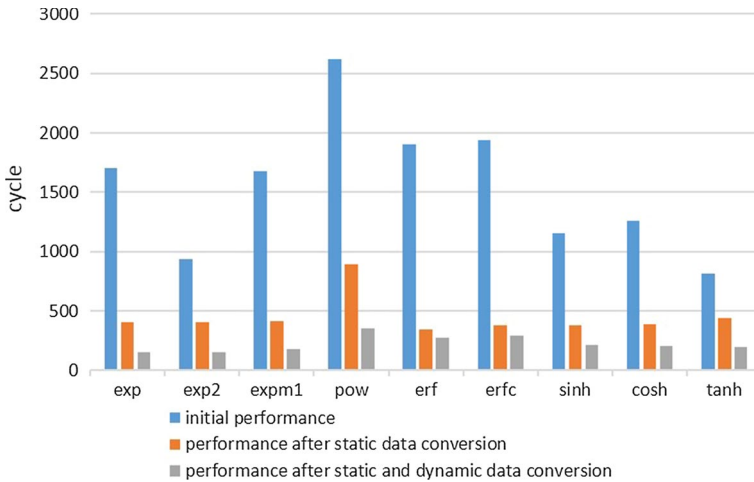


Fig. 16 Performance improvement of static data conversion and dynamic data conversion on exponential functions

code explosion effect with different searching algorithms is shown in Fig. 14, followed by the performance comparison shown in Fig. 15.

We can make the following conclusions from Fig. 14.

- Static data conversion has a slight impact on code explosion, while dynamic data conversion is the main source of the code explosion issue.
- The quick search contributes least to the code explosion issue among all the searching algorithms used in the experiment, while the remaining algorithms aggravate the issue by introducing various operations like comparison, branching, jumping.

The sequential search suffers from a performance degradation due to the numerous introduced searching operations, while the binary search and the block-matching search behavior similarly. The quick search outperforms the other algorithms thanks to its own properties as we introduced above.

As a summary, we have the following conclusions from Figs. 14 and 15.

- The quick search is better than the remaining algorithms in terms of both code explosion and performance improvement, validated by our theoretical analysis and the experimental results shown in this subsection.
- The sequential search, the binary search and the block-matching search behave similarly in terms of code explosion, but the latter two algorithms may outperform the first one in improving the performance.

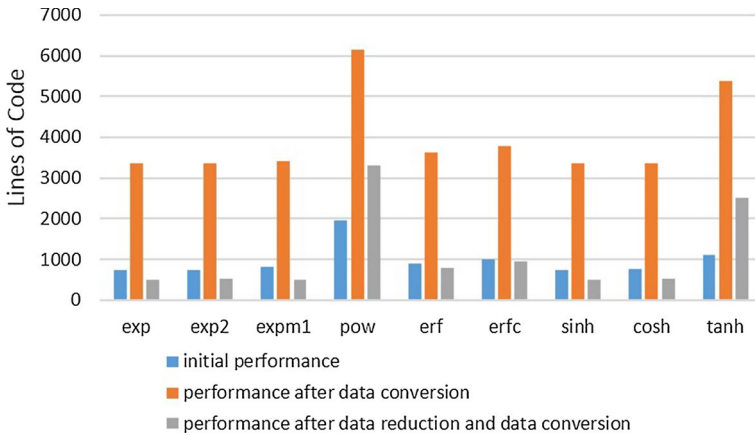


Fig. 17 Effect of data reduction on code size in data conversion

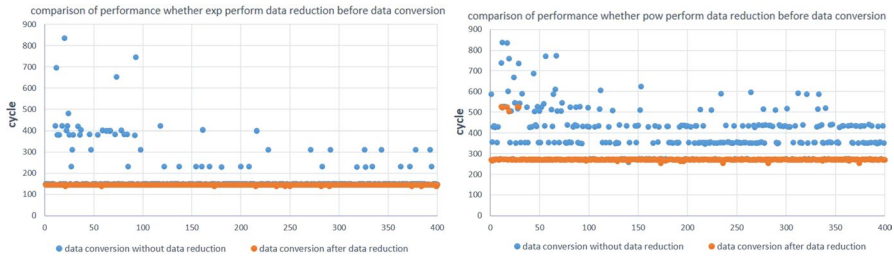


Fig. 18 Performance comparison of the *exp* function and *pow* function under data conversion, with and without data reduction

4.4.2 Performance of data conversion

We compare the performance of data conversion by analyzing the effectiveness of static data conversion and dynamic data conversion. Figure 16 shows the performance improvement of exponential functions when enabling different data conversions.

The following conclusions can be inferred from the results shown in Fig. 16.

- Both static data conversion and dynamic data conversion may improve the performance of exponential functions.
- Static data conversion contributes more to the performance improvement, since all static data would be accessed by a function, each of which would be accessed by a single instruction. On the contrary, only partial of the dynamic data would be accessed by a function. Generally speaking, an increase in the number memory access operations may lead to a proportional improvement of the performance when enabling data conversion. As a result, the performance improvement of static data conversion is more significant than that of dynamic

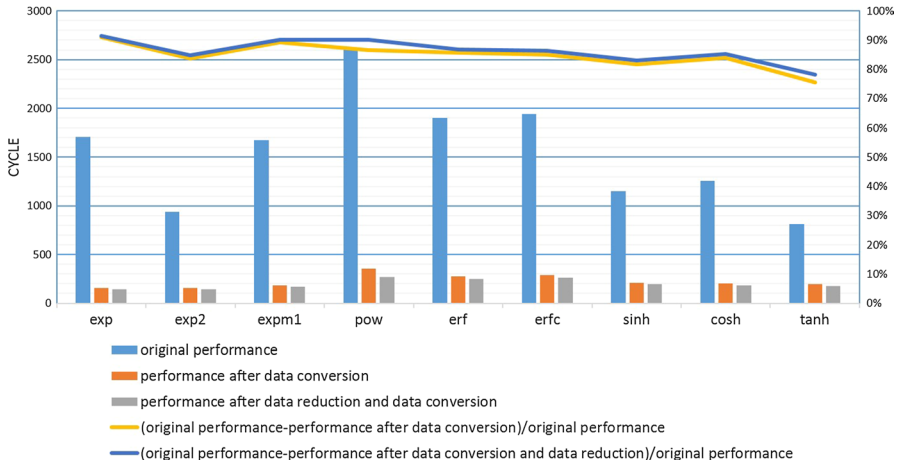


Fig. 19 Performance of exponential functions after optimization

data conversion as the former holds a greater number of memory access operations.

- With the advantages in code growth and performance, static data should be converted first and then dynamic data.

4.4.3 Performance of data reduction and its impact on code explosion

Data conversion may not change the accuracy of exponential functions when data reduction is disabled. Figure 17 shows the code explosion result of data conversion with and without data reduction.

Code explosion caused by data conversion could be much heavier by increasing the code size by 4 to 5 times, but it could be mitigated by data reduction. The sizes of the data tables of the *pow* function and *tanh* function are still much larger than expected since only the data of the *exp* part are reduced, leading to a heavier code explosion, while the size of data table of the remaining functions could be reduced from more than 500 to dozens, leaving out code explosion even with the data conversion optimization. As a result, data reduction is an effective solution to mitigate the code explosion issue and it is well recommended to apply this optimization before data conversion. Figure 18 shows the performance the *exp* function and *pow* function under the data conversion optimization, with and without data reduction.

We run each function 400 times and report the execution cycles. The performance trends of both functions stay flat when applying both data conversion and data reduction, while they become fluctuated when data reduction is disabled. More specifically, the code explosion issue of the *pow* function is much heavier due to the larger size of data, leading to a higher probability of cache misses. As a result, we may conclude that the performance improvement could benefit from the data reduction optimization.

We also evaluate the performance of exponential functions under data conversion together with data reduction, with the performance results shown in Fig. 19.

As can be seen from Fig. 19, the performance of all exponential functions has been improved significantly when data conversion is turned on, leading to an improvement by 85% over the version without data conversion. The performance is further improved by up to 86.2% when data reduction is enabled before applying data conversion.

As a result, data conversion is an effective optimization for memory access. The performance of such optimization may be inferior under some input cases due to the introduced cache misses caused by data conversion, as shown in Fig. 18, but it may bring about significant performance improvement in most cases.

4.4.4 Comparison of data conversion and data movement

We also show the performance of different (combinations of) optimization strategies, including those without any optimizations, data conversion, data reduction and movement, data reduction and conversion, in Fig. 19. Comparing with the version without any optimizations, the combination of data reduction and movement may bring about a performance improvement by 91%, while it may reach 86.2% when both data reduction and data conversion are turned on, with the contribution of data conversion to performance improvement by 85%. As a consequence, data movement is preferred for better performance when there is enough space on the scratchpad memory, or data conversion can be an effective, alternative optimization.

5 Related work

Optimizations of memory access latency have always been one of the hot topics of computer architecture, evolving into two main categories in the research world.

The first category improves the performance of memory accesses by upgrading the underlying structures. Ruliang Ma [8] proposed a novel queue design strategy by reexecuting load instructions and implementing a so-called storage fragile window algorithm, in the context of queue design of high-performance processors. The strategy further reduces the frequency of accesses to main memory by fully exploiting the data locality. Hongyan Wang [9] designed a new mapping approach for improving the parallelism and data locality of memory accesses. More specifically, this approach reduces cache misses of memory access instructions and the switching delay of read/write instructions by designing a multilayer memory access scheduler.

The second category improves the performance by hiding memory access latency [10–16], including the so-called larger instruction window technique, out-of-order execution, multi-threading, software pipelining, speculative precomputation, data prefetching, etc. We did not resort to hiding memory access latency since it may not bring about the expected performance improvement on the Sunway architecture due to the long memory access latency of the processing cores.

Gal and Bachelis [4] designed a shared data table for the *sin* function and *cos* function, implementing the shared data table with division operators in the *atan2* part, thereby optimizing the overall memory bandwidth and/or the memory footprint in cache of elementary functions. Christopher [17] also implemented a similar a shared-table-based algorithm for integrating the $\log(x)$ and $\log(1+x)$ functions, and also for the integration of the *exp*(*x*) and *expm1*(*x*) functions, without a special handle to the accuracy issue of such functions with the inputs around zero, significantly improving performance of the functions on architectures with SIMD extension and/or time-consuming data-dependent branching instructions. The idea of shared data table is similar to the data reduction strategy of our work, but our approach has a wider applicability to elementary functions, unlike the work for some specific functions, e.g., the optimizations for fast Fourier transform [18].

Jinchen Xu [19] studied the optimizations for hiding memory access latency and reducing cache misses of elementary functions in the context of Sunway processing cores, proposing an instruction scheduling algorithm by considering the local optimality of memory accesses on heterogeneous architectures, hiding the memory access latency by overlapping with computation. They also discussed the dynamic allocation strategies of the scratchpad memory, storing part of the data of functions on the scratchpad memory to improve the performance of this part. Compared with the work of Xu [19], our approach is more general with a better performance improvement by bypassing hiding memory latency. Our work takes into the scratchpad memory of the Sunway architecture into consideration, but the kernel fusion strategy [20] that was proposed for memory latency optimizations on distributed memory architecture is missing.

6 Conclusion and future work

We proposed some efficient optimizations for memory access latency of elementary functions on the Sunway processing cores. With the proposed optimizations include data reduction, data movement and data conversion, our approach can significantly optimize the memory access latency on the Sunway architecture and improve the performance of such functions. The optimizations could also be used in a different combination manner for a better performance purpose, with the guidance of the following aspects.

- Data conversion could be applied when the predefined requirement of accuracy of the functions is strict, i.e., one may not change decrease the accuracy, with which the exponential functions can achieve an average performance improvement by 85%.
- Data conversion could be applied but has to come after data reduction when the accuracy could be changed without exceeding the maximum allowance, while the scratchpad memory cannot hold all the data. In our experiments, the exponential functions can achieve an improvement by 86.2% on average.

- Data reduction followed by data movement should be preferred when both the accuracy could be changed and the space on the scratchpad memory for the data of elementary functions could be guaranteed, validated by the average performance improvement of the exponential functions by 91%.

Our approach could be easily extended to handle transcendental functions, applicable to all cases of heavy latency caused by memory accesses. In addition, one may also view the accelerators like GPU, FPGA of heterogeneous architectures as the Sunway processing cores and implement the proposed strategies on other heterogeneous architectures.


Our future work is to exploit the allocation strategies of the scratchpad memory of Sunway processing cores, balancing the limited but high-speed local buffers to both elementary functions and user applications, further improving the performance of the applications on the Sunway architecture.

References

1. Muller J-M (1999) A few results on table-based methods. *Reliab Comput* 5(3):279–288
2. Muller J-M (2006) *Elementary functions: algorithms and implementation*, 2nd edn. Birkhauser, Basel
3. Burden Richard L, Douglas Faires J (2010) *Numerical analysis*, 9th edn. BROOKS/COLE CENGAGE Learning, Boston
4. Gal S, Bachelis B (1991) An accurate elementary mathematical library for the IEEE floating point standard. *ACM Trans Math Softw* 17(1):26–45
5. Tang PTP (1991) Table-lookup algorithms for elementary functions and their error analysis. In: Kornerup P, Matula DW (eds) *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, Los Alamitos, CA, pp 232–236
6. Tang PTP (1999) Table-driven implementation of the logarithm function in IEEE Floating-point arithmetic. *ACM Trans Math Softw* 16(4):378–400
7. Tang PTP (1990) Accurate and efficient testing of the exponential and logarithm functions. *ACM Trans Math Softw* 16(3):185–200
8. Ma RL (2012) *Design and optimization of key load store technology in high performance processor*. Shanghai Jiao Tong University, Shanghai
9. Wang HY (2012) *The optimization of memory controller for high performance CPU*. National University of Defense Technology, Changsha
10. Zhou H, Conte TM (2003) Performance modeling of memory latency hiding techniques. Technical report. ECE Department, State University, NC
11. Mowry T (2009) *Tolerating latency through software controlled data prefetching*. In: PhD Thesis. Stanford University, Stanford
12. Gornish E, Granston E, Veidenbaum A (2009) Compiler-directed data prefetching in multiprocessors with memory hierarchies. In: *International Conference on Supercomputing*
13. Liu W, Ma S, Huang L, Wang Z (2017) The design of NoC-side memory access scheduling for energy-efficient GPGPUs. *Int J Parallel Program* 46:1–14
14. Rau BR, Fisher JA (1993) Instruction-level parallel processing: history, overview, and perspective. *J Supercomput* 7(12):9–50
15. Naderan-Tahan M, Sarbazi-Azad H (2014) Adaptive prefetching using global history buffer in multicore processors. *J Supercomput* 68(3):1302–1320
16. Torrents M, Martnez R, Molina C (2016) Facing prefetching challenges in distributed shared memories for CMPs. *J Supercomput* 72(4):1453–1476

17. Anand CK (2010) Unified tables for exponential and logarithm families. *ACM Trans Math Softw* 37(3):28
18. Carlson DA (1991) Using local memory to boost the performance of FFT algorithms on the CRAY-2 supercomputer. *J Supercomput* 4(4):345–356
19. Xu JC (2014) Access optimization technique for mathematical library of slave processors on heterogeneous many-core architectures. *Comput Sci* 41(6):12–17
20. Filipovi J, Madzin M, Fouse J, Matyska K (2015) Optimizing CUDA code by kernel fusion: application on BLAS. *J Supercomput* 71(10):3934–3957

Affiliations

Bei Zhou¹  · Yongzhong Huang² · Jinchun Xu¹ · Shaozhong Guo¹ · Hongyuan Qi¹

Bei Zhou
13653970052@163.com

¹ State Key Laboratory of Mathematical Engineering and Advanced Computing, No. 62, Science Avenue, High-Tech Zone, Zhengzhou 450001, China

² Guilin University of Electronic Technology, Guilin 541004, China