Partial Elements Reuse of Vector Register in SIMD Mathematical Functions

Lei Wang, Zhang Chun-Yan, Yong-Zhong Huang, Jin-Chen Xu

Zhengzhou Information Science and Technology Institute, wanglei1167@gmail.com
School of Information Science and Engineering, Henan University of Technology, zcy_zz@163.com

Abstract

In order to achieve high performance in Single Instruction Multiple Data (SIMD) mathematical functions, efficient approach of memory accessing and register reuse is necessary. This paper proposes an optimizing approach of partial elements reuse of vector register (PERVR) in SIMD mathematical functions, which takes advantage of partial elements of vector to reduce register pressure, redundant computation and memory access. In addition, we present implementation to improve the effectiveness of PERVR and analyze the benefits and applicability can be applied in a general Reduced Instruction Set Computing (RISC) architecture with SIMD extensions. The experimental results show that our approach is 14.19% performance improvement on average compared with vector pack/unpack approach (VPU).

Keywords: SIMD, Partial Elements Reuse, Mathematical Function

1. Introduction

Most of the current Instruction Set Architecture (ISA) of mainstream processor support Single Instruction Multiple Data (SIMD) extensions, such as Intel’s MMX/SSE/SSE2/SSE3, AMD’s 3DNow!, IBM’s VMX, SUN’s VIS, MIPS’ MDMX and so on, which can perform the same operation on multiple data simultaneously. SIMD extensions have already become an effective way to improve the performance of both multimedia and non-multimedia applications. For some applications, performance can be greatly enhanced by using SIMD extensions that take advantage of data-level parallelism.

The SIMD architecture usually provides limited number of fixed-length vector registers, such as Intel's Pentium IV has only eight 128-bit vector registers, usually in addition to the scalar registers. Thus data needs to be frequently load/store between the vector registers and system cache. Accessing data from vector registers, versus a cache or main memory, has two advantages [1]: The most obvious advantage is lower latency of accesses; another is taking advantage of data-level parallelism in order to eliminate memory access instructions and reduce the number of instructions to be issued. Efficient use of vector register resources will always be an important design consideration of compiler and assembler programming, especially designing architecture-based low-level algorithm or implementing SIMD mathematical functions.

There has been previous work on value reuse, register reuse, and register locality. These previous studies focus both on optimizing uses of scalar register [2,3,4] or vector register [5], and some of them attempt to integrate instruction scheduling and register allocation as single stage of compiler [6]. Rakvic, R. Rakvic and J. Gonzalez [7] propose a fusion of threads to help reduce energy consumption and the number of register file access. Rubén González and Adrián Cristal [8] present a new integer register file organization to reduce access time of registers accomplished by partial value locality. Liem Tran and Nicholas Nelson [9] evaluate two register sharing techniques for reducing register usage. The first technique dynamically combines physical registers having the same value, and the second technique combines the demand of several instructions updating the same logical register and share physical register storage among them. Xipeng Shen and Yutao Zhong [10] extend reuse distance histograms method by more than two training inputs using regression analysis, and using multi-model prediction to overcome the limitation due to small training inputs or coarse-grain data collection.

However, there has been much less attention paid to partial elements reuse of register to reduce the overall average demand of physical registers and unnecessary redundant computation in SIMD mathematical functions. We have observed that the allocation of vector register is not well optimized
only by compiler such as GCC or Intel C++ compiler when using assembly language. Many SIMD instructions split into more than one instruction such as scalar load/store, vector pack/unpack and so on, which are expense of a serious performance penalty and redundant amount of wasteful memory access.

In this paper, we present an optimizing approach of partial elements reuse of vector register (PERVR) in SIMD mathematical functions, which takes advantage of partial elements of vector to reduce register pressure, redundant computation and memory access. We further show the benefits and applicability of PERVR. This research is distinguished from previous work on exploiting reuse of registers, because it considers not only temporal but also partial elements reuse, especially focus on optimization of SIMD mathematical functions. PERVR can be implemented along with register file organization or instruction scheduling, since orthogonal and synergistic effects can be obtained.

Performance evaluation indicates that our optimization approach is significantly improving the performance of SIMD mathematical functions compared with vector pack/unpack approach (VPU). In our simulations, we demonstrate 14.19% performance improvement on average across a set of double precision floating-point functions benchmark and dramatically reduce the number of memory accesses, register pressure and redundant computation.

The rest of this paper is organized as follows: Section 2 presents the architecture of our experimentation. A simple example of VPU approach and PERVR approach is presented in Section 3. We present an experimental evaluation of PERVR and VPU in Section 4. We conclude and discuss future work in Section 5.

2. Vector Registers and Instructions of Improved SimAlpha Simulator

In this section, we introduce a RISC-based processor architecture of our experimentation. We use improved prototype SimAlpha [11] simulator with SIMD extensions. The simulator has 32 floating-point vector registers, each of which is 256-bit width that holds four 64-bit double/single precision floating point elements. For simplify the presentation, we use a smaller subset of the assembly language in our experiments that will suffice for presentation purposes in our experiments. The vector registers are used for different purpose. Table 1 indicates the usage conventions for the vector registers. Only floating point register $f31$ is always equal to 0.0. The remainder are partitioned by convention into return values ($f0–f1$), callee saved ($f2–f9$), temporaries ($f10–f15$ and $f22–f30$), and procedure arguments ($f16–f21$)[12].

<table>
<thead>
<tr>
<th>Register name</th>
<th>Usage</th>
<th>Preserved across procedure call (Saved)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f0–f1$</td>
<td>Used to hold function results.</td>
<td>No</td>
</tr>
<tr>
<td>$f2–f9$</td>
<td>Saved registers.</td>
<td>Yes</td>
</tr>
<tr>
<td>$f10–f15$</td>
<td>Temporary registers used for expression evaluation.</td>
<td>No</td>
</tr>
<tr>
<td>$f16–f21$</td>
<td>Used to pass the first six actual arguments.</td>
<td>No</td>
</tr>
<tr>
<td>$f22–f30$</td>
<td>Temporary registers used for expression evaluation.</td>
<td>No</td>
</tr>
<tr>
<td>$f31$</td>
<td>Always has the value 0.0</td>
<td>No</td>
</tr>
</tbody>
</table>

SimAlpha supports 4*64-bit floating point vector. The common scalar and SIMD vector instructions of SimAlpha are ldi, fld, fst, pack, unpack, vload, vstore, vsllow etc.. For example, the operations of pack and unpack instructions can be expressed as ra, vb, $x$, vc and va, $y$, rb respectively. Note that $x$ and $y$ is 0, 1, 2 or 3. The pack instruction indicates that the $x$'th element of vector register vb is replaced by the value of register ra, and save the result into vector registers vc. The unpack instruction take the $x$'th element out of vector register va and save into register rb.

3. Partial Elements Reuse of Vector Register

3.1 General Operations in SIMD Registers

We present a simple general example to further motivate our work. SIMD mathematical functions such as simd_sin, simd_cos and so on, which can perform the same operation on four input data
simultaneously. However, mathematical functions are rather difficult to fully vectorized by SIMD extensions. The main reason is that four input data may be in different intervals of function domain, therefore computation run into different code branches. On the other hand, the operations performed on four elements may be different, thus the functions cannot calculate four input data simultaneously. As shown in Figure 1, \(a_0, a_1, a_2, a_3\) are four elements of vector register \(f_{14}\), and \(b_0, b_1, b_2, b_3\) are four elements of vector register \(f_{15}\), and \(c_0, c_1, c_2, c_3\) are four elements of vector register \(f_{16}\). The four elements of vector register \(f_{14}\) perform partial different operations and store results into \(f_{15}\) and partial results into \(f_{16}\).

**Figure 1.** A simple example of different operations performed on four elements of vector register

From an implementation standpoint, four elements of vector register should be unpacked before executing corresponding operation, and the results will be packed into vector register. The procedure of traditional VPU consists of the following steps:

1) The vector register \(f_{14}\) executes unpack operations by four times, and stores the corresponding elements \(a_0, a_1, a_2, a_3\) into four temporary registers.

2) \(a_0, a_1, a_2, a_3\) execute \(OP1\) operation by three times and one \(OP0\) operation, the results will be saved into \(b_0, b_1, b_2, b_3\) respectively.

3) \(b_0, b_1, b_2, b_3\) store into four temporary registers, then execute pack operations by four times into vector register \(f_{15}\).

From the above method, we can see that vector register \(f_{14}\) and \(f_{15}\) have three save operations which can be optimized by SIMD instructions. Furthermore, \(f_{15}\) and \(f_{16}\) have two elements of same values which can be partial reused.

The above method is expansive memory access and limited to the number of vector registers. The procedures perform pack/unpack operations eight times and need eight temporary registers totally. The simple example mentioned above is very frequent in the implementation of SIMD mathematical functions. The redundant operations and temporary registers can result in a serious performance penalty. Furthermore, if many operations performed on vector registers are different, register spilling may be occurred. The implementation of PERVR and applicability is proposed to solve the problem in the following sections.

### 3.2 Our approach

In this section, we describe a general method for reusing shared values in the vector register and reducing the number of physical registers requirements. As illustrated in Figure 2, the right side shows memory access scheme in SIMD sin function, and corresponding assembly code of VPU method generated by compiler is on the left side. The procedure has two similar phases, and uses three temporary registers to generate results packed into vector register \(f_{14}\) and \(f_{15}\).
PERVR consists of two steps. First, the data dependency is checked in order to reduce memory access. Second, shared values in the vector register are reused by using logical and shift operations.

The source code on the left side in Figure 2 comprises two phases. There are totally eight memory accesses, and the data are packed into vector register $f14$ and $f15$ respectively. As the phase 1 and phase 2 have no data dependency of memory access, we can directly load memory data into vector registers by using two SIMD memory access instructions instead. As shown in Figure 3(a), the code of register reuse eliminates redundant memory access and vector pack operations.

From the Figure 3(a) we can further find that the elements data of vector register $f14$ and $f15$ are similar. The traditional approach to solve the problem has two methods. The first method is to choose the overlap length of two registers as a temporary register, and then merge the results after memory access [13]. The second method is to control the elements of registers which are involved in computation by using mask register [14]. The two methods can adapt well to traditional vector machines because the vector length is very long and variable. However, the vector of SIMD architecture is usually known as short vector, and the vector length is usually fixed. Both of the two methods result in more redundant codes to guarantee correctness and consistency.

In the implementation of SIMD mathematical functions, a significant number of instructions create values that already exist in the vector registers. In order to reduce redundant memory access, we use a general method for reusing shared values in the vector register by using logical and shift operation in Figure 3(b).

In this process, the target vector register $f15$ reuse data of the first, second and third elements in vector register $f14$. If we adopt VPU shown in Figure 1, eight memory access operations, eight register pack operations and four registers are needed. PERVR only need one SIMD memory access operation, one scalar memory access operation, one vector shift operation and two registers totally.

From the above discussion, PERVR can be summarized as following five steps:

**Step 1:** Use $v_{load}$ / $v_{store}$ SIMD operations to reduce duplicated memory access.

**Step 2:** Assume the elements of vector $v[a3$, $a2$, $a1$, $a0]$ in vector register $V$ have two or more different operations $OP0, OP1$ and so on.

**Step 3:** Find the sub-vector $V$ which has same operation $OPi$, and transform $OPi$ to SIMD operation $VOPi$

**Step 4:** The result of SIMD operation is saved into corresponding elements of target register, and other elements of vector $V(V-V')$ are computed by using scalar operations.

**Step 5:** The results of scalar operations are saved into corresponding elements of target register.
This method can also be applied to vectors with variant length. \( OP_1, OP_2, OP_3 \) denote the vector logical shifting operation, floating point number accessing operation and vector inserting operation respectively. Assuming that there are \( n \)-dimension-length vectors \( v[1][A, B, \ldots, Z], v[2][G, H, \ldots, W] \) and a temporary vector register \( v[3] \). The number of the repeatable data is \( i (i=0, 1, \ldots, n) \) with the length of single data is \( k \) bits. So the PERVR general reusing method can be described as following three steps:

**Step 1:** If \( i=n \), which means \( v[1].rx=v[2].rx \), then assign the value of \( v[1] \) to \( v[2] \) directly. It just is the complete data reusing operation.

**Step 2:** If \( i=0 \), data reusing operation is invalid, the \( n \) access data immediately.

**Step 3:** Shift the value of \( v[1] \) by \( (n-i)*k \) bits to right with the vector logical shifting operation \( OP_1 \) and then assign the result to the lower \( i \) \( k \)-bit-length double precision floating point numbers of the temporary vector \( v[3] \). Access the remaining \( n-i \) double precision floating point numbers from main memory by floating point number accessing operation \( OP_2 \), and then insert them to the corresponding position of target register \( v[2] \) by vector inserting operation \( OP_3 \).

### 3.3 Analysis on Applicability of Partial Elements Reuse of Vector Register

The part of vector register that reused is the same as the part of elements with same operations. For example, there are three elements of four elements vector register have the same operation, so target vector register can reuse the three elements. If all the four elements have different operations, only one element can be reused, PERVR is turned into VPU. For this reason, we should analyze the applicability of our approach.

The model of applicability for our approach is similar to a realistic state-of-the-art processor. To simplify the presentation, we make following four definitions:

1. Vector \( v[a_0, a_1, a_2, \ldots, a_i, a_n] \) has \( m \) elements, and in these elements there are \( n \) different operations: \( OP_i \), \( OP_j \), \( OP_k \). The CPU cycles of these operations are \( r_i, r_j, r_k \). There are \( m_k \) elements have the \( k \)th operation.
2. The vector pack/unpack operations consume the same CPU cycles denoted by \( r \).
3. The operation \( OP_i \) consumes \( r_i \) CPU cycles, and the SIMD operation \( SIMD.OP_i \) consumes \( n r_i \) CPU cycles.
4. \( m_i \) is the average elements of the same operation \( OP_i \).

First, we analyze the performance of VPU. The vector \( v[a_0, a_1, a_2, \ldots, a_i, a_n] \) has \( m \) elements, need \( m \) pack and \( m \) unpack operations that consume \( 2m*r \) CPU cycles. The CPU cycles consumed by \( m \) elements of \( n \) different operations are:

\[
\sum_{k=1}^{n} m_k \times r_k
\]

As a result, the total CPU cycles consumed by VPU are:

\[
\tau_{\text{VPU}} = 2m \times r + \sum_{k=1}^{n} m_k \times r_k
\]

Second, we analyze the performance of PERVR. The vector \( v[a_0, a_1, a_2, \ldots, a_i, a_n] \) has \( m \) elements, and there are \( m_i \) elements have the same operations \( OP_i \). Therefore, this part of elements can use PERVR and no need to pack/unpack operations. The rest \( m-m_i \) elements still need pack/unpack operations, and total CPU cycles are \( 2(m-m_i)*r \).

The operation \( OP_i \) implement by SIMD extensions, the total CPU cycles are \( nr_i \). Other elements totally consume CPU cycles:

\[
\sum_{k=1}^{n} m_k \times r_k + \sum_{k=1}^{n} m_k \times r_k
\]

As a result, the total CPU cycles consumed by PERVR are:

\[
\tau_{\text{PERVR}} = 2(m-m_i)*r + \sum_{k=1}^{n} m_k \times r_k + nr_i
\]

If PERVR is an applicable approach for implementation in SIMD mathematical functions in general situations, the CPU cycles of our approach is less than the CPU cycles of VPU. Let \( \tau_{\text{PERVR}} < \tau_{\text{VPU}} \). From equation (2) and equation (4), we can express the following equation:
In order to quantitatively analyze the Applicability of PERVR on SimAlpha simulator, assuming that $r_i = 1$, $v_{sum} = 4$ and $v_{s} = 1$, we can calculate the result of average elements of the same operations $OP_i$ should be satisfying the condition: $m > 1.33$. This condition means that in the vector register of four elements, our approach can obtain better performance than VPU if there are at least two elements have the same operation. Furthermore, we can see that the VPU’s utilization of register is $2m$ and the PERVR’s is $2(m - m_i)$, which is related to the average elements of the same operations in a vector.

### 4. Experimental Evaluation

To evaluate our approach, we use a combination of benchmarks from the SIMD mathematical functions library. We take 9 SIMD mathematical functions (sin, cos, tan etc.) for example, and evaluate the performance of PERVR and VPU respectively. Our experiments were carried out using SimAlpha simulator with SIMD extensions, the SIMD functions were compiled with the Compaq C compiler using the standard -O2 optimization level. The 32 SIMD registers of SimAlpha are 256-bit double/float precision floating-point vectors, and each of them has four 64-bit double/float precision floating point elements. Note that the letter ‘P’ denotes PERVP, and ‘V’ denotes VPU.

<table>
<thead>
<tr>
<th>SIMD Func</th>
<th>Total number Ops committed</th>
<th>Temporary registers used</th>
<th>Saved vector registers used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instructions</td>
<td>Memory accesses</td>
<td>P</td>
</tr>
<tr>
<td>sin</td>
<td>866339</td>
<td>869221</td>
<td>0.33%</td>
</tr>
<tr>
<td>cos</td>
<td>757884</td>
<td>763007</td>
<td>0.67%</td>
</tr>
<tr>
<td>tan</td>
<td>910234</td>
<td>916953</td>
<td>0.73%</td>
</tr>
<tr>
<td>acos</td>
<td>1024931</td>
<td>1031614</td>
<td>0.65%</td>
</tr>
<tr>
<td>sqrt</td>
<td>458212</td>
<td>462762</td>
<td>0.98%</td>
</tr>
<tr>
<td>cbrt</td>
<td>713959</td>
<td>716753</td>
<td>0.39%</td>
</tr>
<tr>
<td>log</td>
<td>871923</td>
<td>879742</td>
<td>0.89%</td>
</tr>
<tr>
<td>pow</td>
<td>951279</td>
<td>954660</td>
<td>0.35%</td>
</tr>
<tr>
<td>exp</td>
<td>904833</td>
<td>903135</td>
<td>0.60%</td>
</tr>
</tbody>
</table>

Table 2. Evaluation of memory access and register pressure

Figure 4. (a) Reduction of Registers (b) Reduction of instructions and memory accesses

Table 2 depicts the results for register pressure and memory access of different SIMD functions. The total number of operations committed to simulator consists of instructions and memory accesses. The number of instructions and memory accesses include our test-code of PERVR as same as VPU. The fourth and seventh columns indicate the average percentage of PERVP instructions is reduced by 0.62% compare with VPU, and memory access is reduced on average by 7.14%, respectively. The less instructions and memory accesses potentially allow a shorter register pressure. As shown in Table 3, The eighth and ninth column present the number of temporary and saved floating point vector registers use, indicating the percentage reduction with respect to VPU.
Figure 4(a) and Figure 4(b) indicate that temporary registers use of PERVP is reduced on average by 13.8%, and saved vector registers use is reduced on average by 43.1%. It is notable that using less registers could lead to a reduction of the number of instructions and memory accesses in the implementation of SIMD mathematical functions, and hence to a reduction in the number of execution cycles shown in Table 3.

As described in Section 3.3, there are $n$ different operations, and $m_i$ is the average elements of the same operations. We can express the following equation of each SIMD functions:

$$m_i = \sum_{k=1}^{m} m_i / n$$

Table 3 presents the summary of the performance evaluation of PERVR and VPU. Performance was measured by the cycle counters.

<table>
<thead>
<tr>
<th>Table 3. Evaluation of memory access and register pressure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SIMD Func</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>acos</td>
</tr>
<tr>
<td>tan</td>
</tr>
<tr>
<td>cbrt</td>
</tr>
<tr>
<td>pow</td>
</tr>
<tr>
<td>log</td>
</tr>
<tr>
<td>cos</td>
</tr>
<tr>
<td>sin</td>
</tr>
<tr>
<td>sqrt</td>
</tr>
<tr>
<td>exp</td>
</tr>
</tbody>
</table>

We show two sets of results which indicate the performance improvements of PERVR for each individual SIMD mathematical functions. In Figure 5(a), we show the PERVR CPU cycles of SIMD functions compared with VPU. On average, PERVR is increased performance of 14.19% compared with VPU. In Figure 5(b), we measure the effect on speedup improvement by different $m_i$ of SIMD functions. Not surprisingly, the average elements of the same operations can nearly lead speedup to linear performance improvement. From the Figure 5, while the average elements of the same operations is increasing we can see that the improvement of speedup from PERVR is additive. In summary, the PERVR presented in this paper dramatically reduce the number of memory accesses, register pressure and redundant computation, and yield significant performance improvements across SIMD mathematical functions.

Overall, we have shown that the PERVR is indeed very important and effective that delivers significant performance improvement in SIMD mathematical functions.
5. Conclusions

This paper presents an optimizing approach of partial elements reuse of vector register in SIMD mathematical functions, which takes advantage of partial elements of vector in order to reduce register pressure, redundant computation and memory accessing. We further show the benefits and applicability of PERVR. The prototype implementation of our approach can be used in LLVM and other compiler frameworks [15].

We implemented our approach in improved RISC-based prototype SimAlpha simulator with SIMD extensions. Our experimental results show that the speedup ranges are from 11.75% to 17.31% and an average of 14.19% on the nine SIMD mathematical functions, and most memory accesses and register pressure are removed. We can deduce that an optimizer that integrates the SIMD parallelism and partial elements reuse techniques could be even more effective. For example, in a memory access intensive or computation intensive algorithm, selection of load/store to parallelize by reuse vector register could also take partial elements reuse technique into account. A memory access intensive or computation intensive algorithm is the subject of future work.

References
