PAPER

# Insufficient Vectorization: A New Method to Exploit Superword Level Parallelism

**Wei GAO**[†a)], **Lin HAN**[†], **Rongcai ZHAO**[†], **Yingying LI**[†], *Nonmembers*, *and* **Jian LIU**[††b)], *Member*

**SUMMARY** Single-instruction multiple-data (SIMD) extension provides an energy-efficient platform to scale the performance of media and scientific applications while still retaining post-programmability. However, the major challenge is to translate the parallel resources of the SIMD hardware into real application performance. Currently, all the slots in the vector register are used when compilers exploit SIMD parallelism of programs, which can be called sufficient vectorization. Sufficient vectorization means all the data in the vector register is valid. Because all the slots which vector register provides must be used, the chances of vectorizing programs with low SIMD parallelism are abandoned by sufficient vectorization method. In addition, the speedup obtained by full use of vector register sometimes is not as great as that obtained by partial use. Specifically, the length of vector register provided by SIMD extension becomes longer, sufficient vectorization method cannot exploit the SIMD parallelism of programs completely. Therefore, insufficient vectorization method is proposed, which refer to partial use of vector register. First, the adaptation scene of insufficient vectorization is analyzed. Second, the methods of computing inter-iteration and intra-iteration SIMD parallelism for loops are put forward. Furthermore, according to the relationship between the parallelism and vector factor a method is established to make the choice of vectorization method, in order to vectorize programs as well as possible. Finally, code generation strategy for insufficient vectorization is presented. Benchmark test results show that insufficient vectorization method vectorized more programs than sufficient vectorization method by 107.5% and the performance achieved by insufficient vectorization method is 12.1% higher than that achieved by sufficient vectorization method.

***key words:*** *SIMD extension, SIMD parallelism, vector register, insufficient vectorization*

## 1. Introduction

The need to increase performance and power efficiency in modern processors has led to a wide adoption of SIMD (single-instruction multiple-data) vector units. All major vendors support vector instructions and the trend is pushing them to become wider and more powerful [1]. SIMD instruction set extensions are quite common today in both high performance and embedded microprocessors [2]. However, writing code that makes efficient use of these units and leads to platform-specific implementations is rather difficult [3]. Compiler-based automatic vectorization is one of the solutions to this problem. There are two main types of vectorization algorithms. Loop-based algorithms can convert multiple iterations of a loop into a single iteration of vector instructions [4]. However, these algorithms require that the loop has well-defined induction variables, usually affine, and that all inter-loop and intra-loop dependences are statically analyzable. On the other hand, algorithms that target straight-line code operate on repeated sequences of scalar instructions outside a loop [5]. They do not require sophisticated dependence analysis and have more general applicability. However, vectorization is often thwarted when the original scalar code does not contain enough isomorphic instructions to make conversion to vectors profitable [6].

Automatic vectorization techniques have proven quite effective at extracting large levels of data-level parallelism (DLP). However, vectorization is often much less effective for applications which have low trip count loops, complex control flow, and non-uniform execution behavior [7]. As a result, SIMD lanes remain idle due to insufficient DLP [8]. SIMD widths have been following an upward trend: the 128-bit Streaming SIMD Extensions (SSE) of x86 architectures has been augmented by 256-bit Advanced Vector Extensions (AVX); the new Intel Many Integrated Core (MIC) architecture supports 512-bit SIMD. For the high-performance computing (HPC) industry, effective utilization of SIMD on current hardware – and preparing for potentially wider SIMD in the future – are crucial [9]–[11]. Though the vector register provided by SIMD extension is becoming wider, performance achieved by it is not scaling. When exploiting the SIMD parallelism of programs, compiler will make full use of slots that vector register provides, which can be called sufficient vectorization. Sufficient vectorization means all the data in the vector register is valid. Because all the slots must be used, the chances of vectorizing programs with low SIMD parallelism are given up by sufficient vectorization method. In addition, the speedup achieved by full use of vector register sometimes is not as large as that achieved by partial use. However, partial use of vector register has its own advantages including the following aspects:

- When the number of isomorphic statements in a straight-line code is less than vector factor, insufficient vectorization method is needed. With the length of vector register becomes longer, vector factor is getting bigger. Take IMCI as an example, it can deal 8 double or 16 float simultaneously. Therefore, 8 double isomor-

Manuscript received May 31, 2016.
Manuscript revised August 15, 2016.
Manuscript publicized September 29, 2016.

[†]The authors are with State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450000, People's Republic of China.
[††]The author is with College of Communication and Information Engineering, Nanjing University of Posts and Telecommunications, Xinmofan Road #66, Nanjing 210003, China.
a) E-mail: yongwu22@126.com
b) E-mail: by.liujian@gmail.com

DOI: 10.1587/transinf.2016EDP7236

phic statements are needed when sufficient vectorization method is used in IMCI. But there are rarely 8 double isomorphic statements naturally in a straight-line code. Though more isomorphic statements can be gained through loop unrolling, sometimes it is illegal to unroll a loop because of dependence or non-standard loop. Therefore, insufficient vectorization method is needed at this time.

- When loop-based method is used to exploit SIMD parallelism of inter-iteration for loops, if the iteration number of the loop which is the most suitable for vectorization is smaller than vector factor, sufficient vectorization cannot vectorize it and insufficient vectorization method is needed at this time. This scenario is very common in real-world applications. Loop nest may have many levels and the iteration number of the most suitable vectorization one is often smaller than vector factor.
- Vectorization is often impeded by the SIMD memory architecture, which typically provides access to contiguous memory items only, often with additional alignment restrictions. Computations, on the other hand, may access data elements in an order that is neither contiguous nor adequately aligned. Bridging this gap efficiently requires careful use of special mechanisms including permute, pack/unpack, and other instructions that incur additional performance penalties and complexity. Besides, these mechanisms differ widely from one SIMD platform to another. The permutation ability of different platforms is not the same, some platforms support flexible shuffle modes, others support fixed shuffle mode and even unsupported. It is difficult for fixed or unsupported platform to vectorize non-stride memory access. Furthermore, the cost of shuffle instruction is expensive. It is NP-hard problem to obtain the optimal shuffle mode [28]. These are all needed to be considered when adopting sufficient vectorization method. However, when using insufficient vectorization method, there is no need to consider permutation. Though it is not making full use of vector register, the performance is better than sufficient vectorization method when shuffle cost is large. Besides, it provides a method to vectorize non-stride memory access.

Therefore, insufficient vectorization method is proposed, the methods of computing inter-iteration and intra-iteration SIMD parallelism for loops are put forward, then according to the relationship between the parallelism and vector factor a method is established to make the choice of vectorization method in order to vectorize programs as much as possible. Finally, code generation strategy for insufficient vectorization is considered.

The contributions of this paper are threefold:

- The adaptation scene of insufficient vectorization is analyzed.
- The computation methods of inter-iteration and intra-iteration SIMD parallelism for loops are given. In addition, they are used to guide the choice of vectorization method.
- Loop unrolling is reconsidered for vectorized loop.

The rest of this paper is organized as follows: Sect. 2 describes insufficient vectorization method in detail. In Sect. 3, we give a brief overview of the GCC compiler used and its vectorizer infrastructure. Section 4 presents our approach of vectorizing loops, which is guided by SIMD parallelism. In Sect. 5 we demonstrate performance results of applying our approach compared to sufficient vectorization method. Section 6 introduces related work and Sect. 7 concludes.

## 2. Insufficient Vectorization Method

Vector register can deal with multiple data simultaneously. Currently, most of the vector registers are used entirely, which means all the data are useful in the slots. The status lasts from vector loading to computation then to vector storing and data in the slots are all useful during these operations. However, partial use of vector register is needed when SIMD parallelism is low, which means some slots hold valid data while others hold invalid data. Using manners of vector register can be divided into four kinds, as is shown in Fig. 1. Using entirely is shown in Fig. 1(a) and Fig. 1 (b) means one part of the vector register is invalid and the other is valid. Figure 1 (c) shows both ends are invalid and the middle part is valid. Discontinuous use is shown in Fig. 1(d).

Insufficient vectorization method means not all the data in the register is valid. Take vectorizing statement $S1$ : $c[2i] = a[2i]+b[2i]$ as an example, which is shown in Fig. 2. In the vector register $Va$ and $Vb$, because data is loaded to the vector register in the continuous manner, the even positions are valid and the odd positions are invalid. Insufficient vectorization means computing $Va$ and $Vb$ directly, then storing the result to $Vc$ and ignoring the data in the odd position and meanwhile the results of even positions will be stored to memory using extract instruction. Sufficient vectorization method will load twice and shuffle once to get a vector register which is full of valid data. After computation, shuffling is also needed when the results are stored.

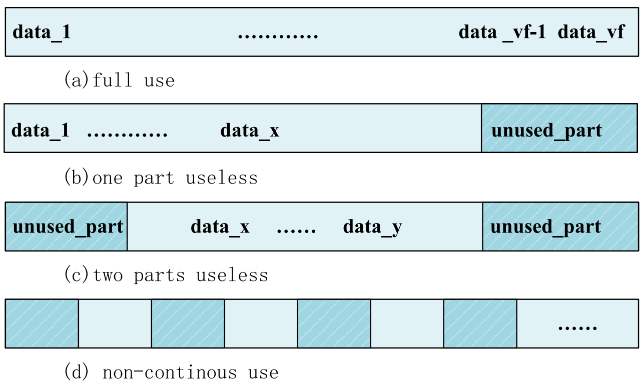


**Fig. 1** Use manners of vector register.

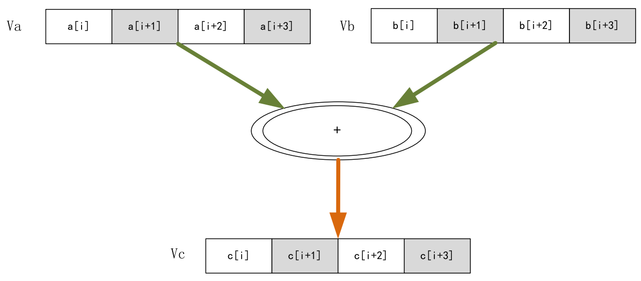


**Fig. 2** An example to illustrate insufficient vectorization method.

For convenience, we call slots that hold valid data valid slots and those hold invalid data invalid slots. Assume the vector factor is 4. When compilers vectorize statement $S1$, the theoretical speedup is 4 in the case that sufficient vectorization is used and is 2 in the case that insufficient vectorization is used. Though insufficient vectorization method cannot make full use of vector register, it has advantages in several cases.

Insufficient vectorization applies in the following two scenarios: one is when the SIMD parallelism is low, as is shown in Fig. 1(b) and Fig. 1 (c); the other is for non-stride memory access, as is shown in Fig. 1(d).

## 2.1 Low SIMD Parallelism

When the intra-iteration SIMD parallelism is low, insufficient vectorization is needed. As is shown in Fig. 3(a), it is the most time-consuming loop in 435.gromacs of SPEC2006, which is called inl1130 and spends 75 percent of the time. There are 3 isomorphic statements in the loop. However, the number of isomorphic statements cannot scale through loop unrolling because it may be illegal to unroll the loop as indirect memory exists in the basic block. Therefore, when the vector factor of a platform is greater than 3, insufficient vectorization is needed because of the low intra-iteration SIMD parallelism. This case often appears in real-world applications. Another example is shown in Fig. 3(b). It is the most time-consuming loop in183.equake of SPEC2000, which is called smvp and spends 72 percent of the time. There are 3 isomorphic statements in this loop. Loop unrolling is illegal because indirect memory access exists in this while-do loop. Therefore, when the vector factor of a platform is greater than 3, insufficient vectorization is needed.

When the inter-iteration SIMD parallelism is low, insufficient vectorization is needed. As is shown in Fig. 4(a), it is the most time-consuming loop in 454.calculix of SPEC2006, which is called E_c3d and spends 69 percent of time. The outermost *i1-loop* is suitable for vectorization, but its iteration number is only 3. This case often appears in real-world applications. As is shown in Fig. 4(b), it is the second time-consuming loop in BT of NPB, which is called compute_rhs and spends 16 percent of time. The innermost *m-loop* is suitable for vectorization and the number of iteration is only 5. Therefore, when the vector factor of a plat-

```
for (k= nj0 ; k< nj1 ; k ++) {
j = 3* jjnr [k];

jx = pos [j];
jy = pos [j +1];
jz = pos [j +2];

dx = ix -jx;
dy = iy -jy;
dz = iz -jz;
rsq = dx*dx + dy*dy + dz*dz;

rinv = 1.0/ sqrt ( rsq );

rinvsq = rinv * rinv ;
rinvsix = rinvsq * rinvsq *
rinvsq ;
vnb6 = c6* rinvsix ;
vnb12 = c12 * rinvsix * rinvsix
;
fs = (vnb12 - vnb6 + vcoul )*
rinvsq ;

fac [j] -= dx*fs;
fac [j +1] -= dy*fs;
fac [j +2] -= dz*fs;
}
```

(a) The most time-consuming loop of 435.gromacs in SPEC2006

```
while (Anext < Alast) {
    col = Acol[Anext];
    sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] + A[Anext][0][2]*v[col][2];
    sum1 += A[Anext][1][0]*v[col][0] + A[Anext][1][1]*v[col][1] + A[Anext][1][2]*v[col][2];
    sum2 += A[Anext][2][0]*v[col][0] + A[Anext][2][1]*v[col][1] + A[Anext][2][2]*v[col][2];

    w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] + A[Anext][2][0]*v[i][2];
    w[col][1] += A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][2][1]*v[i][2];
    w[col][2] += A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] + A[Anext][2][2]*v[i][2];
    Anext++;
  }
```

(b) The most time-consuming loop of 183.equake in SPEC2000

**Fig. 3** Examples of low intra-iteration SIMD parallelism.

```
do i1=1,3
           do j1=1,3
             vo(i1,j1)=0.d0
             do k1=1,nope
               vo(i1,j1)=vo(i1,j1)+shp(j1,k1)*voldl(i1,k1)
             enddo
           enddo
         enddo
```

(a) The most time-consuming loop of 454.calculix in SPEC2006

```
do j = 1, grid_points(2)-2
    do i = 3,grid_points(1)-4
       do m = 1, 5
          rhs(m,i,j,k) = rhs(m,i,j,k) - dssp *
    >                (  u(m,i-2,j,k) - 4.0d0*u(m,i-1,j,k) +
    >                6.0*u(m,i,j,k) - 4.0d0*u(m,i+1,j,k) +
    >                u(m,i+2,j,k) )
       enddo
    enddo
 enddo
```

(b) The second most time-consuming loop of BT in NPB

**Fig. 4** Examples of low inter-iteration SIMD parallelism.

```
for(i=0; i<reg->size; i++){
    if(reg->node[i].state & (1 << control1)){
        if(reg->node[i].state & (1 << control2)){
            reg->node[i].state ^= (1 << target);
        }
    }
}
```

(a) The most time-consuming loop of 462.libquantum in SPEC2006.

```
do  i1=1,mm1-1
   u(2*i1-1,2*i2-1,2*i3-1)=u(2*i1-1,2*i2-1,2*i3-1)
   >         +z(i1,i2,i3)
   u(2*i1,2*i2-1,2*i3-1)=u(2*i1,2*i2-1,2*i3-1)
   >         +0.5d0*(z(i1+1,i2,i3)+z(i1,i2,i3))
enddo
```

(b) The second most time-consuming loop of BT in NPB

**Fig. 5** Examples of non-stride memory access.

form is greater than 5, insufficient vectorization is needed.

In addition to the above two cases, there are several cases resulting in low SIMD parallelism. For example, the number of iteration epilogue loop is smaller than vector factor after a loop is peeled for vectorization. When there is flow dependence in the loop and the dependence distance is smaller than vector factor, insufficient vectorization method is also needed to ensure the correctness of vectorization.

### 2.2 Non-Stride Memory Access

Non-stride memory access kernels are often used in real-word applications. As is shown in Fig. 5(a), it is the most time-consuming loop in 462.libquantum of SPEC2006, which is called toffoli and spends 64 percent of time. In 172.mgrid of SPEC2006 the third time-consuming loop is shown in Fig. 5(b), which is called interp and spends 13 percent of the time. Furthermore, programs related to complex number often contain non-stride memory access.

(1) When the platform does not support shuffling or the shuffle mode is fixed, insufficient vectorization is needed. Vector registers can hold all valid slots through permutation for non-stride memory access programs. But permutation is restricted to the ability of shuffling instruction. If the platform does not support shuffle or the shuffle mode is fixed, the cost of permutation is expensive or even offset the benefit from vectorization. However, insufficient vectorization does not need permutation, which provides a manner to implement the vectorization of non-stride memory access programs.

(2) When the performance achieved by insufficient vectorization method is better than that achieved by sufficient vectorization, insufficient vectorization is needed. Sufficient vectorization method needs more than one load and shuffle in order to obtain a vector register which is full of valid data. Shuffling is also needed when the result is written to memory after computation. Besides, shuffling is expensive and it is an NP-hard problem to obtain the optimal shuffle mode [28], and thus sufficient vectorization may get no benefit. But for insufficient vectorization method, though it doesn't make full use of vector register, sometimes the speedup achieved may be greater than that achieved by sufficient vectorization.

## 3. Vectorizer Overview

We use GCC to implement insufficient vectorization. In this section, we describe the infrastructure of GCC that is relevant to our work. In the next section, we describe how this infrastructure was extended to support insufficient vectorization.

GCC uses multiple levels of Intermediate Languages in the course of translating the original source language to assembly. Our focus is on GIMPLE, which supports Static Single Assignment and retains enough information from the source language to facilitate advanced data-dependence analysis and aggressive high-level optimizations, including auto-vectorization. From the high-level target-independent GIMPLE IL, statements are translated to the low-level instructions of the RTL, where target-dependent optimizations, such as instruction scheduling and register allocation, is applied.

Two mainstream vectorization methods include loop-based method which is oriented to inter-iteration and SLP which is oriented to intra-iteration. Both loop-based and SLP are implemented in GCC. The GCC auto-vectorizer derives from the classic approach for vectorization, which focuses on loops, and attempts to detect data parallelism across different iterations of the loop. After some general properties of the entire loop are analyzed, each statement is analyzed and vectorized independently. For each statement, the vectorizer tries to group together VF occurrences of that statement from VF different consecutive iterations, where VF is the vectorization factor. This is what we refer to one-to-one substitution approach. Each scalar statement is mapped to one vector statement that performs the respective operation on VF data elements from VF consecutive iterations of the loop. Additional handling beyond the one to-one substitution is provided to generate code before or after the loop: constants and loop invariants require that vectors be initialized at the loop pre-header; reduction and induction computations require special epilog code after the loop. In other cases, like access to unaligned or non-unit stride data and operations on mixed data types, single scalar operation is replaced by more than one vector operation. The loop bound is transformed to reflect the new number of iterations, and if necessary, an epilog scalar loop is created to handle the remaining loop iteration. Using if conversion, the framework is also extended to handle more sophisticated loop body. The cost model is also improved and loop transformations oriented to vectorization with a heuristic method is implemented, such as loop interchange, loop distribution. Outer-loop can also be vectorized in our framework.

The SLP vectorization approach on the other hand groups VF statements from the same iteration into a vector statement. It looks at flat code sequences. So it is in fact not aware of the loop context, and can be applied to basic-blocks anywhere in the program. The SLP approach starts by looking for groups of accesses to adjacent memory addresses, attempting to pack them together into vector load/store operations. These groups of adjacent memory references are used as the seed to an analysis that, starting from this seed, following the def-use chains between statements, in searching for computation chains that can be vectorized. Loop rerolling can regroup isomorphic statements into a loop and then vectorize them using the loop-based method.

Loop-aware SLP is derived from SLP. It gains more isomorphic statements through loop unrolling. SLP is applied when the number of isomorphic statements in a basic block is greater than vector factor. Therefore, determining unroll factor is the key of loop-aware SLP. When the unroll factor is 1, loop-aware SLP method can be called pure SLP, and when the unroll factor equals to vector factor, loop-aware SLP is loop-based method. When the statement number in the loop is less, loop unrolling can convert inter-iteration SIMD parallelism into intra-iteration SIMD parallelism. Hence, loop-aware SLP is restricted to loop unrolling. When it is illegal to unrolling a loop, such as the dependence inhibits unrolling or it is a non-standard loop and even the inter-iteration SIMD parallelism of the loop is also low, loop-aware SLP method cannot vectorize it. At this time, insufficient vectorization is needed.

## 4. Extending the Vectorizer to Handle Insufficient Vectorization

### 4.1 The SIMD Parallelism Computation Method of Loops

Note that not only inter-iteration SIMD parallelism but also intra-iteration SIMD parallelism can be exploited for loops, and thus the SIMD parallelism can be calculated from these two perspectives.

The computation method of intra-iteration SIMD parallelism is introduced first. As intra-iteration SIMD parallelism of loops is an inherent property, its value is fixed. But for the limitation of exploiting method, the values obtained from different methods obtained may not be the same. We take SLP as an example to illustrate how to calculate intra-iteration SIMD parallelism (IAP) for short) of loops. The first requirement of SLP is that packed statements must be isomorphic. SLP takes the adjacent memory access array as the seed, then extends pack set from the def-use chain and use-def chain, and finally schedules pack set according to the dependence. Using def-use chain and use-def chain to extend pack set can reduce the opportunity of unpacking. Taking the adjacent memory access array as the seed can improve the performance of SIMD code largely. Therefore, three limitations of computation intra-iteration SIMD parallelism are as follows: statements must be isomorphic; they must have adjacent memory access; these statements satisfy the dependence of vectorization. The number of statements satisfying the above three conditions is IAP.

The inter-iteration SIMD parallelism (IEP for short) of loops refers to the number of a statement can be SIMD executed. The legality of SIMD executing is that of statement reordering in fact, while the legality of statement reordering depends on the dependence analysis. If the dependence of loop does not hinder statement reordering, then the loop has inter-iteration SIMD parallelism. In a traditional vector machine, since the length of vector register is considered unlimited, a loop can be executed in one time, the loop cannot have loop-carried dependence. While the length of SIMD extension is fixed, say 128-bit or 256-bit, it is vectorizable if a loop has loop-carried dependence, but the dependence distance must be considered. If the distance is larger than the vector factor, all the slots can be used. But if the distance is smaller than the vector factor, it is illegal to use all the slots and insufficient vectorization method must be used. Therefore, two conditions have to be considered when we calculate inter-iteration SIMD parallelism: one is the iteration number, and the other is the dependence distance. In specific, if the dependence distance is 0, which means it is loop-independent dependence, IEP equals to *iter*, which stands for the iteration number; if the dependence distance is *dep*, which is larger than 0, IEP equals to the minimum between *dep* and *iter*.

### 4.2 Vectorization Method Guided by SIMD Parallelism for Loops

The essence of loop-aware SLP is converting inter-iteration SIMD parallelism into intra-iteration SIMD parallelism through loop unrolling when the intra-iteration SIMD parallelism of a loop is low. Hence, the inter-iteration SIMD parallelism of the loop must be enough. But loop-aware SLP cannot exploit in the following two cases: one is that when loop unrolling is illegal, such as if the dependence inhibits or it is a non-standard loop; the other is the inter-iteration SIMD parallelism of a loop is not enough with the increasing vector register, which is more likely to occur. Hence, we propose a new vectorization method for loops that is guided by SIMD parallelism (VMSP for short) to solve the disadvantages of loop-aware SLP. VMSP method chooses appropriate vectorization method according to the relationship between SIMD parallelism and vector factor, as shown in Table 1.

When the intra-iteration SIMD parallelism is larger than the vector factor, i.e., $IAP \geq VF$, SLP can be applied directly because the number of the isomorphic statements is enough to pack.

When the intra-iteration SIMD parallelism is smaller than the vector factor but is bigger than 1, i.e., $1 < IAP < VF$, loop unrolling is needed at this time. The unrolling times $UF$ can be expressed as $\left\lceil \frac{\mathrm{VF}}{\mathrm{IAP}} \right\rceil$, which is the upper bound of the quotient of $VF$ and $IAP$. It means the inter-iteration SIMD parallelism is enough if $IE \geq UF$, then the loop is unrolled UF times and SLP is applied to pack. If $1 \leq IEP < UF$, which means the inter-iteration SIMD parallelism is also not enough. Insufficient vectoriza-

**Table 1** Vectorization method guided by SIMD parallelism for loops

| $IAP \geq VF$ | $1 < IAP < VF$ | | $IAP = 1$ | | |
|---|---|---|---|---|---|
| SLP | $IEP \geq UF$ | $1 \leq IEP < UF$ | $IEP \geq VF$ | $1 < IEP < VF$ | $IEP = 1$ |
| | loop-aware | insufficient vectorization method based SLP | Loop-based | insufficient vectorization based loop-based | execute serially |

tion method based SLP works after the loop is unrolled $UF$ times.

When the intra-iteration SIMD parallelism equals to 1, i.e., $IAP = 1$, only inter-iteration SIMD parallelism can be applied at this time. In specific, if the inter-iteration SIMD parallelism is larger than the vector factor, the loop-based method is applied. If the inter-iteration SIMD parallelism is smaller than the vector factor but it is bigger than 1, i.e., $1 < IEP < VF$, insufficient vectorization method based loop-based works. If $IEP = 1$, the loop has to be executed serially.

In the Fig. 4 (b), The $IEP$ of innermost *m-loop* is 5, and the $IAP$ of innermost *m -loop* is 1. Assuming the platform is intel's AVX, we can choose loop-based method to achieve best vectorization performance according to Table 1, when $IAP = 1$, $IEP = 5$ and $VF = 4$. Assuming the platform is intel's MIC, we can choose insufficient vectorization based loop-based to achieve the best vectorization performance according to Table 1, when $IAP = 1$, $IEP = 5$ and the $VF = 8$.

After exploiting intra-iteration SIMD parallelism by SLP, loop-based method can also be applied to exploit inter-iteration SIMD parallelism. Because there may be scalar codes after using SLP method, vector codes have to be unrolled if compiler continues vectorizing the loop using the loop-based method.

Loop unrolling can not only improve instruction-level parallelism but also increase the opportunity of data reuse. Three differences exist between scalar loops and vector loops in unrolling. First, scalar registers have more uses than vector registers. Scalar registers may be used for loop index, a base address register, stack register and so on. Vector registers are only used for memory access and computation. Second, unrolling for vector loops do not need to consider the cost of control flow, because there is no if statement after if-conversion. Third, some registers have been owned by global variables, but there is rarely global vector variable when compiler vectorize a loop. Unrolling too many times will lead to overflow in the instruction buffer, so the determination of unroll factor is crucial. As there is no need to consider the number of vector register and branch statement, only the number of operations is concerned about vectorized loop unrolling. According to the number of operations and the size of instruction buffer, the upper bound of unroll factor $UFU$ equals to *SL/N_sum*, where $SL$ is the size of instruction buffer and *N_sum* stands for the number of operations.

When a loop is vectorized, the iteration number will decrease $VF$ times, where $VF$ is the vector factor. Therefore, the vectorized loop is suitable for unrolling completely if $VF$ is very big. Unrolling completely can reduce the cost of loops, which is very significant for loop nest. But unrolling immoderately will also lead to code inflating and decrease the efficiency of unrolling. If the number of iteration is known when compiling, unrolling completely for the loops whose number of iteration *simd_ite* is smaller than *unroll_times + 1*. Because when *simd_ite* equals to *unroll_times + 1*, the loop will be unrolled unroll_times and generate an epilog, the statements number of unrolling completely is the same with before unrolling. We add an option vec_unroll to control loop unrolling for vectorized loops, unrolling flow is as follows:

(1) if *vec_unroll=0*, jump to step 4 as it means turning off this optimization. If *vec_unroll =1*, unroll times is obtained by the compiler at this time and jump to step 2. If *vec_unroll* is bigger than 1, unroll times is designated by *vec_unroll*, jump to step 3.

(2) determine the upper bound of unroll times according to the formula *UFU=SL/N_sum*. In order to reduce the number of prefetching, unroll times will be adjusted to the pow of 2, which is $2^{\lfloor \log_2^{UFU} \rfloor}$

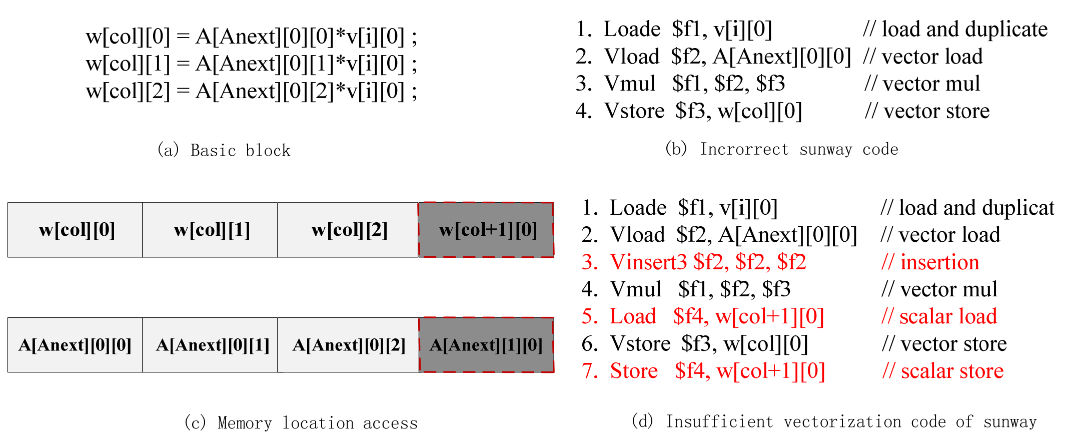
(3) unroll the vectorized loop unroll times, if the iteration number is known when compiling and is less than *unroll times +1*, unroll completely for the vectorized loop.

(4) finish unrolling.

## 4.3 Code Generation for Insufficient Vectorization Method

Both SLP and loop-based vectorization methods have the passes of pre-analysis, dependence analysis and code generation. There are a few differences between insufficient vectorization method and sufficient vectorization method in pre-analysis and dependence analysis. How to generate code for insufficient vectorization method correctly is a crucial problem. In order to guarantee the correctness of insufficient vectorization, three aspects have to be considered: first is valid slots and invalid slots have to be marked when data is loaded from memory; second is to ensure the corresponding of the valid slots when computing data in the vector register; third is how to avoid the result of invalid slots written to memory and guarantee the valid results written to memory.

When SLP is used, load instruction is generated according to the packs. Continuous slots can be used in the vector register. If the array access memory is aligned, compiler can generate aligned load, otherwise unaligned load is generated. The valid slots are marked simultaneously. Because the statements are isomorphic, we get the corresponding valid slots when computing. The valid slots and invalid ones all participate in the computation. Valid results must be written to memory to ensure the correctness, and the results in invalid slots must be abandoned. Code generation for loop-based method is similar with SLP. Valid slots are determined using stride and offset of arrays in the loop, and



```
w[col][0] = A[Anext][0][0]*v[i][0] ;
w[col][1] = A[Anext][0][1]*v[i][0] ;
w[col][2] = A[Anext][0][2]*v[i][0] ;
```

(a) Basic block

```
1.  Loade  $f1, v[i][0]             // load and duplicate
2.  Vload  $f2, A[Anext][0][0]  // vector load
3.  Vmul   $f1, $f2, $f3            // vector mul
4.  Vstore  $f3, w[col][0]          // vector store
```

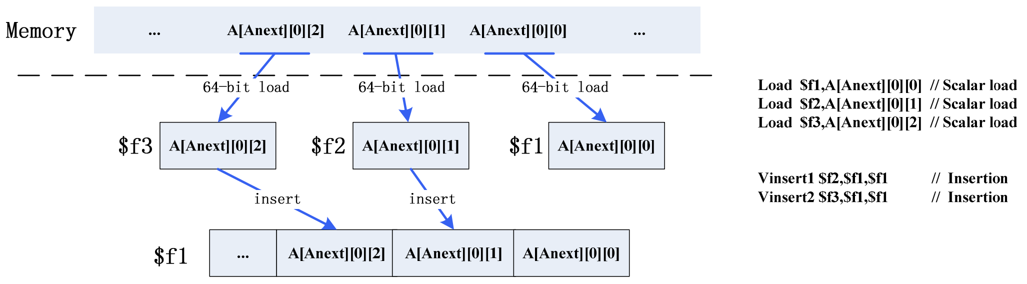
(b) Incrorrect sunway code

(c) Memory location access

```
1.  Loade  $f1, v[i][0]             // load and duplicat
2.  Vload  $f2, A[Anext][0][0]  // vector load
3.  Vinsert3 $f2, $f2, $f2         // insertion
4.  Vmul   $f1, $f2, $f3            // vector mul
5.  Load    $f4, w[col+1][0]      // scalar load
6.  Vstore  $f3, w[col][0]          // vector store
7.  Store   $f4, w[col+1][0]      // scalar store
```

(d) Insufficient vectorization code of sunway

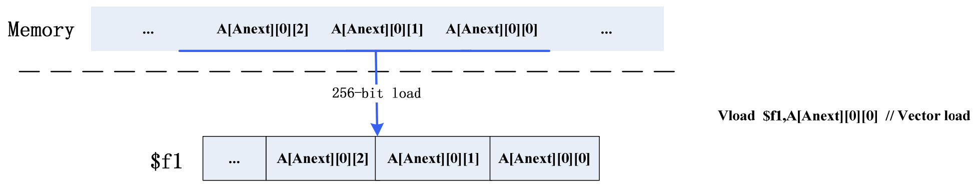
**Fig. 6** An example to illustrate code generation of insufficient vectorization



```
Load  $f1,A[Anext][0][0]  // Scalar load
Load  $f2,A[Anext][0][1]  // Scalar load
Load  $f3,A[Anext][0][2]  // Scalar load

Vinsert1 $f2,$f1,$f1        // Insertion
Vinsert2 $f3,$f1,$f1        // Insertion
```

**Fig. 7** Data insertion for loading A



```
Vload  $f1,A[Anext][0][0]  // Vector load
```

**Fig. 8** Widened vector load for A

if the strides of arrays are the same, the valid slots of vector register are corresponding. When the strides of arrays in the loop are not the same, the valid slots are not corresponding and code generation is difficult. Therefore, we only consider the same stride in loops. We take an example basic block to illustrate the insufficient vectorization code generation, which is simplified from Fig. 3 (b).

As shown in Fig. 6(a), we discuss how to vectorize 3-way double precision operations on sunway, which has a 256-bit SIMD extension. The sunway register can be viewed as a 256-bit vector register or a 64-bit scalar register, which depends on operating by vector instruction or scalar instruction [31]. Figure 6 (b) gives a code generation method in terms of sufficient vectorization method on sunway. However, the vectorized code is incorrect since the two load instructions access the locations that are not originally intended, which are highlighted by the dashed boxes in Fig. 6(c). The two load instructions may cause potentially wrong output, memory errors, or even program crashes. In addition, the multiply instruction may trigger new numeric exceptions on the unused slots. We introduce a number of techniques to ensure their correct and efficient execution on SIMD extension. We describe how to perform partial vector loads, partial vector computations, and partial vector stores correctly and efficiently.

1) Partial Vector Loads

As shown in Fig. 7, data insertion is a straightforward approach to achieve insufficient vector load, which is accomplished by scalar and vector loads followed by packing. Some platform supported masked vector loads, such as Intel's AVX2 and Intel's IMCI. With an masked load instruction "vmaskmovpd $f[k][0]$, mask, ymm2", *f[k][0], f[k][1]* and $f[k][2]$ is loaded to the vector register.

We can also use a widened vector load instead of insertion or mask load. As shown in Fig. 8, one single widened vector load suffices if the elements loaded are consecutive in memory. However, a widened vector load may touch locations that are not accessed originally at the end of an array, causing potentially memory protection errors. This can be avoided by adding dummy elements via tail padding. For a static array, this can be done at compile time. For a dynamic array, its original size can be increased in a call to, e.g., allocate.

2) Partial Vector Computations

Unused slots do not perform any computation in the original program. However, no platform supports masked arithmetic instructions currently. Appropriate data values have to be assigned to unused slots to avoid introducing numeric exceptions. As shown in Fig. 9, we replicate the computational behavior of an arbitrarily selected used slot in every unused slot by simply replicating the values loaded initially into the used slot. Such initial values can be available in a vector initialized from memory or data packing. As a result, no new numeric exceptions will arise. Replication is a general approach that works on any data type. The replication is unnecessary if all numeric exceptions are masked.

3) Partial Vector Stores

As shown in Fig. 10, data extraction is a straightforward approach to achieve insufficient vector store, which is accomplished by scalar and vector stores followed by packing. Some platform supported masked vector stores, such as Intel's AVX2 and Intel's IMCI. With an masked store instruction "vmaskmovpd *ymm*3, *mask*, $x[k][0]$", $x[k][0]$, $x[k][1]$ and $x[k][2]$ is stored to the memory.

Just like the case of partial vector loads, a widened vector store can also be used instead of extraction or mask store. However, a widened vector store must also avoid introducing not only memory protection errors as before but also memory corruption errors. As shown in Fig. 11, by appending a few dummy elements via tail padding to avoid memory protection errors, which is similar with partial vector loads. Furthermore, we must also avoid memory corruption errors due to the writes that do not exist originally. In this example, the full vector writes in line 2 may corrupt the value in $w[col+1][0]$. We propose to apply a backup and recovery mechanism to correct such a corrupted value. As illustrated in Fig. 8, the value in $w[col+1][0]$ is first backed up and then recovered, after it may have been corrupted previously.

Insufficient vectorization code is generated using widened load. Compared with sufficient vectorization method, three extra instructions are added for insufficient vectorization method. As shown in Fig. 6 (d), we add an instruction insertion to ensure the correctness of partial vector computation. We also add a scalar load and a scalar store to ensure the correctness of partial vector store. Different SIMD extensions may offer different memory access methods, we can classify them to three kinds, which are insertion/extraction, mask load/store, and widened vector load/store. Every method has its own advantages and disadvantages. We will discuss the performance differences due to different memory access methods in the evaluation.

## 4.4 Cost Model for Insufficient Vectorization Method

Insufficient vectorization provides a method to vectorize stride memory access programs. When the platform does not support permutation flexibly, insufficient vectorization method can make vector registers work and obtain speedup possibly. When the platform supports permutation, the compiler can choose a better way to vectorize stride memory access programs between using insufficient vectorization method and permutation. If benefit obtained by insufficient vectorization method is larger, insufficient vectoriza-

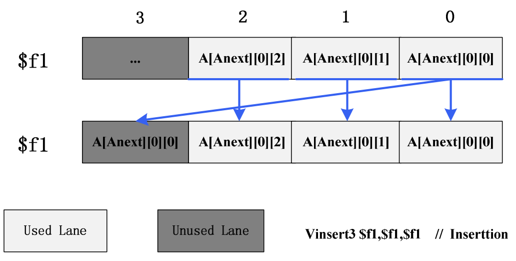


**Fig. 9** Safe execution of partial vector computations

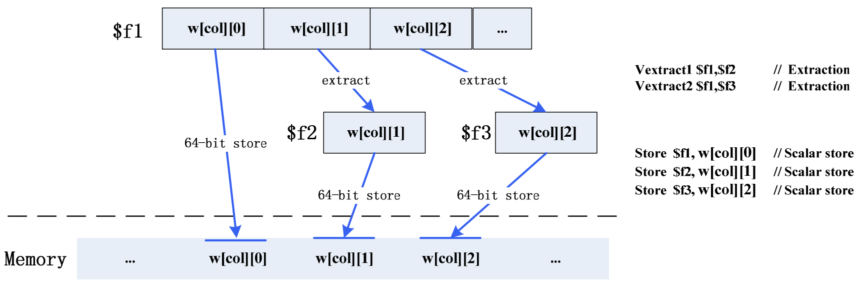

$f1
w[col][0]
w[col][1]
w[col][2]
...
extract
extract
$f2
w[col][1]
$f3
w[col][2]
64-bit store
64-bit store
64-bit store
Memory
...
w[col][0]
w[col][1]
w[col][2]
...
Vextract1 $f1,$f2 // Extraction
Vextract2 $f1,$f3 // Extraction
Store $f1, w[col][0] // Scalar store
Store $f2, w[col][1] // Scalar store
Store $f3, w[col][2] // Scalar store

**Fig. 10** Data extraction for storing w

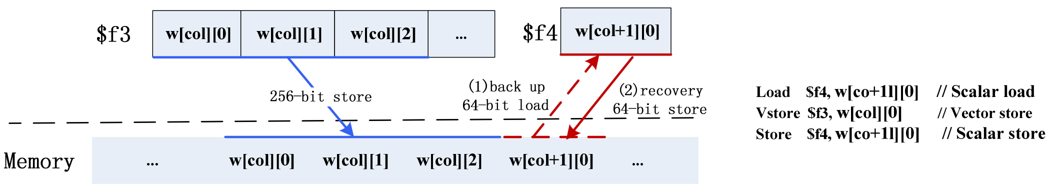


**Fig. 11** Widened vector stores for w

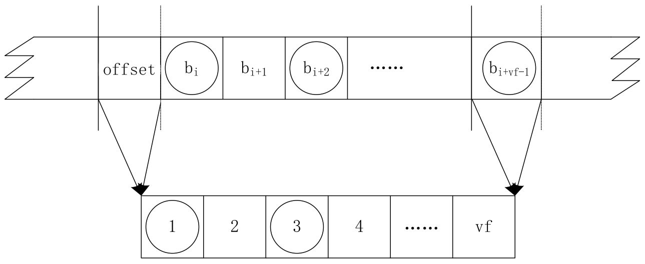


**Fig. 12** Speedup of insufficient vectorization method.

tion method can be chosen. Therefore, a cost model is needed to guide which vectorization method to be chosen.

Because SLP exploits intra-iteration SIMD parallelism, adjacent memory addresses are needed, so we only consider vectorizing stride memory access programs by loop-based method. Insufficient vectorization based on loop-based method focuses on inter-iteration SIMD parallelism, which can perform the respective operation on $N$ data elements from $N$ consecutive iterations of the loop, and thus $N$ is the theoretical speedup. Obviously, $N$ is related to stride. As the length of a vector register is fixed, $N$ will be smaller if the stride is large. In specific, analyzing the relationship between stride and $N$ from align load and unaligned load, as is shown in Fig. 12.

An unaligned load will be used if the first slot of vector register is a valid one. If stride is smaller than $VF$, the speedup will be achieved because data of two consecutive iterations can be filled in the vector register at least. Now $N$ equals to $\left\lceil \frac{\text{VF}}{\text{stride}} \right\rceil$.While an aligned load is used, $offset$ stands for the offset. The first offset slots will be invalid slots. Because remaining $VF$ - $offset$ slots can be used, the speedup of N equals to $\left\lceil \frac{\text{(VF–offset)}}{\text{stride}} \right\rceil$.

Three aspects have to be considered when computing vectorization cost. They are the cost of grouping vector, computation and writing computation results to memory respectively.$\sum c_{gather}$ is the cost of grouping vector and $\sum c_{compute}$ is the cost of computation.$\sum c_{scatter}$ is the cost of writing computation results to memory. For a statement $S$, its cost of sufficient vectorization based on loop-based method is

$$Cost_S = \frac{|D^S|}{VF}(\sum c_{compute} + \sum c_{gather} + \sum c_{scatter})$$

Its cost of insufficient vectorization based on loop-based method is

$$Cost_S = \frac{|D^S|}{N}\left(\sum c_{compute} + c_{load} + c_{store}\right)$$

Where $|D^S|$ is iteration domain, $VF$ is the theoretical speedup of sufficient vectorization method and N is the theoretical speedup of insufficient vectorization method. The key problem of memory access is alignment and consecutive. The platform may supply several ways to deal with alignment, such as it supports unaligned instruction, or can use shift or shuffle instruction when permutation. At the same time, there also may be several ways to vectorize stride memory access programs, such as permutation, extract_odd/extract_even, gather/scatter or read/write memory with the mask. The cost of each scheme can be calculated according to the cost of instructions, finally a scheme of the lowest cost will be applied.

## 5. Experiment and Analysis

There are three aspects in the experiment, including recognition test, kernel test and full program performance test. We implement our insufficient vectorization method in GCC(version 4.9.1). Evaluations are carried on two platforms, i.e., one is on E5-2680 and the other is on KNC. E5-2680 is used to evaluate SSE and AVX. KNC is used to evaluate IMCI. The vector width on IMCI is 512 bits and the VPU may operate on 8 double-precision or 16 single precision data elements. AVX can deal with 4 double-precision or 8 single precision data elements. SSE can deal with 2 double-precision or 4 single precision data elements. The vector width on sunway is 256 bits, which can deal with 4 double-precision or 8 single precision data elements. Sunway is the CPU of Sunway TaihuLight supercomputer, which is number one in the latest Top 500 rank of supercomputer [31]. We use option -ftree-vectorize to generate insufficiently vectorized codes and option -ftree-no-vectorize to generate non-vectorized codes. Run vectorized codes to obtain vectorized time and non-vectorized codes to obtain sequence time. Speedup is the result of sequence time diving vectorized time. Therefore, in the relative performance graphs, the base performance refers to the performance that obtained by the non-vectorized programs. In Fig. 13 to 17, x-axis stands for the kernel that is selected to the evaluation, and y-axis stands for the speedup that the kernel achieves.

### 5.1 Recognition Test

We choose programs with high SIMD parallelism as samples, which are from SPEC2000 and NPB3.3. Compare the recognition rate of loop-aware and VMSP on KNC. Results of SPEC2000 are shown in Table 2 and results of NPB3.3 are shown in Table 3.

In SPEC2000, 6 programs are selected. The recognition of VMSP and loop-aware are nearly the same in 171.swim, 173.applu, 187.facerec and 301.apsi. But in 183.equake and 191.fma3d, the recognition of VMSP is far more than that of loop-aware, this is because programs have low SIMD parallelism and loop-aware cannot exploit effectively. The intra-iteration SIMD parallelism in the kernel of 183.equake is 3 and it is a while-do loop and cannot be rewritten to a for loop, and thus unrolling cannot be applied to convert inter-iteration SIMD parallelism to intra-iteration SIMD parallelism. Therefore, loop-aware fails to exploit it. The same situation occurs in 191.fma3d, whose kernel is as follows,

**Table 2** The recognition of two methods in SPEC2000

| Benchmark | Loop numbers | VMSP | loop-aware | Improved |
|---|---|---|---|---|
| 171.swim | 29 | 11 | 10 | 10% |
| 173.applu | 26 | 5 | 5 | 0% |
| 183.equake | 50 | 26 | 5 | 420% |
| 187.facerec | 30 | 6 | 6 | 0% |
| 191.fma3d | 190 | 21 | 5 | 320% |
| 301.apsi | 20 | 5 | 5 | 0% |
| average | | | | 125% |

**Table 3** The recognition of two methods in NPB

| Benchmark | Loop numbers | VMSP | loop-aware | Improved |
|---|---|---|---|---|
| BT | 23 | 15 | 5 | 200% |
| FT | 19 | 9 | 6 | 50% |
| MG | 62 | 15 | 15 | 0% |
| LU | 26 | 14 | 7 | 100% |
| SP | 30 | 12 | 6 | 100% |
| average | 32 | 13 | 8 | 90% |

```
DO N = 1,NUMRT
   MOTION(N)%Vx = MOTION(N)%Vx + DTaver * MOTION(N)%Ax
   MOTION(N)%Vy = MOTION(N)%Vy + DTaver * MOTION(N)%Ay
   MOTION(N)%Vz = MOTION(N)%Vz + DTaver * MOTION(N)%Az
 ENDDO
```

$Ax$, $Ay$, $Az$ are the three members of $MOTION$. $MOTION$ is a structure, so the compiler has to vectorize array of structure. Three isomorphic statements have the adjacent memory access in the loop body, and thus SLP is more suitable than loop-based for this kernel. As its intra-iteration SIMD parallelism degree is 3, loop unrolling is needed. Since there are other members in the structure, memory access is not continuous between two consecutive iterations after unrolling. Loop-aware cannot exploit it while VMSP can vectorize it successfully.

Take 183.equake to illustrate the computation method of recognition, VMSP recognizes 26 loops successfully and the number of loop-aware is 5, and thus the improved ratio is (26-5)/5 = 420%. In SPEC2000, VMSP recognizes 125% more than loop-aware averagely.

We choose 5 programs from NPB. The recognition of VMSP and loop-aware are nearly the same in MG and FT. But in BT, SP and LU the recognition of VMSP is far more than that of loop-aware. The first kernel of BT is binvcrhs. We can get the following statements after forward substitution and statement reordering are appiled,

```
A basic block that SIMD parallelism degree is 4
    lhs(2,2) = lhs(2,2) - lhs(2,1)*lhs(1,2)
    lhs(3,2) = lhs(3,2) - lhs(3,1)*lhs(1,2)
    lhs(4,2) = lhs(4,2) - lhs(4,1)*lhs(1,2)
    lhs(5,2) = lhs(5,2) - lhs(5,1)*lhs(1,2)
A basic block that SIMD parallelism degree is 3
    lhs(3,3) = lhs(3,3) - lhs(3,2)*lhs(2,3)
    lhs(4,3) = lhs(4,3) - lhs(4,2)*lhs(2,3)
    lhs(5,3) = lhs(5,3) - lhs(5,2)*lhs(2,3)
A basic block that SIMD parallelism degree is 2
    lhs(4,4) = lhs(4,4) - lhs(4,3)*lhs(3,4)
    lhs(5,4) = lhs(5,4) - lhs(5,3)*lhs(3,4)
```

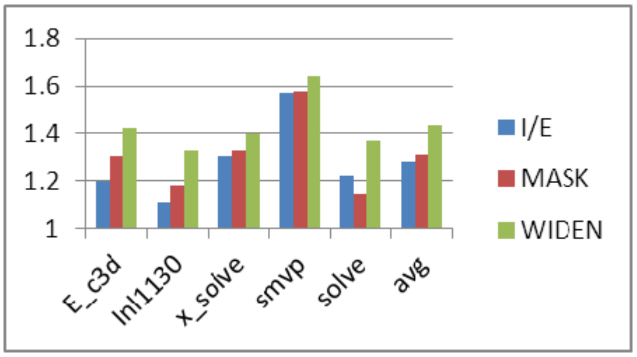


**Fig. 13** Speedups of WIDEN, I/E, and MASK on Intel' AVX

The SIMD parallelism of binvcrhs is 2, 3 and 4 respectively. The second kernel in BT is shown in Fig. 4(b). The vector width on IMCI is 512 bits and vector register can operate 8 double-precision data elements, and thus insufficient vectorization method can exploit successfully while loop-aware cannot. The similar situations occur in LU and SP. VMSP recognizes 90% more than loop-aware averagely in NPB.

Because VMSP recognizes 125% more than loop-aware averagely in SPEC2000 and VMSP recognizes 90% more than loop-aware averagely in NPB. Therefore, VMSP recognizes 107.5% more than loop-aware in selected benchmark averagely.

## 5.2 Evaluating of Different Memory Access Methods

We discuss the performance differences due to different memory access methods. I/E means that data insertion is applied to perform partial vector loads and data extraction is applied to perform partial vector stores. Instead of insertion and extraction, MASK means that masked vector loads and stores are used. WIDEN means that widened vectors are used to perform partial vector loads and stores. We choose five kernels to illustrate the evaluation, which are all double precision Insufficient vectorization method must be used for these five kernel on Intel' AVX. Speedups of different memory access methods are shown in Fig. 13.

Figure 13 reveals the superiority of WIDEN over I/E and MASK. WIDEN achieves speedups that are better than or similar to I/E and MASK for the 5 kernels. The average speedups achieved by WIDEN are better. WIDEN is faster than I/E by 12.0% and is faster than MASK by 9%. WIDEN achieves performance improvements, particularly in the case of E_c3d and Inl1130, because it has avoided the high packing/unpacking and memory access overheads incurred by I/E and the masking overheads incurred by MASK. On average, MASK is slightly better than I/E. I/E suffers from packing/unpacking overhead and more memory access overhead due to more loads and stores issued.

In contrast, MASK avoids PAUN's packing/unpacking overhead but ends up with some other performance pitfalls, such as increased uops, reduced load/store throughput, and ineffective store-load forwarding. A masked vector load has

**Table 4** Examples of low intra-iteration SIMD parallelism

| kernel name | Program | percentage | SIMD parallelism degree | reasons why only intra-iteration SIMD parallelism can be exploited |
|---|---|---|---|---|
| inl1130 | 435.gromacs | 75% | 3 | dependence hinders unrolling |
| smvp | 183.equake | 72% | 3 | non-standard loop |
| solve | 191.fma3d | 33% | 3 | memory access is not continuous between two consecutive iterations. |
| binvcrhs | BT | 23% | 2,3,4 | straight-line code |
| buts | LU | 19% | 2,3,4 | straight-line code |
| z_solve | SP | 18% | 2,3,4 | straight-line code |

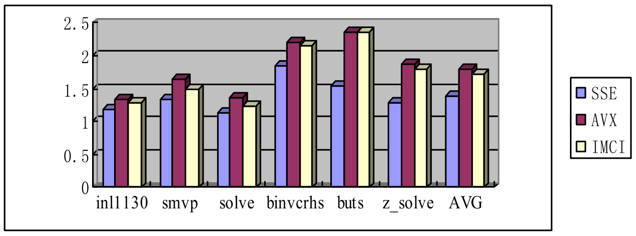


**Fig. 14** Result of low inter-iteration SIMD parallelism.

a similar latency as a normal vector load. However, their reciprocal throughputs are different: 0.5 for a normal load but 2 for a masked load. MASK has managed to achieve better or similar speedups as I/E for all kernels except for solve. The main reason is that MASK uses longer vectors than I/E and it suffers three times as many cache-line splits as I/E.

## 5.3 Evaluating the Kernels of Low Intra-Iteration SIMD Parallelism

We test the kernels which loop-aware cannot vectorize and VMSP do in the previous section. The evaluation is divided into two parts, i.e., one is evaluating kernels of intra-iteration SIMD parallelism is low, and the other is evaluating kernels of inter-iteration SIMD parallelism is low. Three SIMD extension instructions are utilized, which are SSE, AVX and IMCI. We compare performance of the kernels and evaluate these three SIMD extensions.

In the real-world application, loops of low intra-iteration SIMD parallelism are very common. We choose kernels which take large execution time as examples, which are shown in Table 4. It lists kernel names, from which program, percentage, SIMD parallelism degree. Though unrolling can convert inter-iteration SIMD parallelism into intra-iteration SIMD parallelism, sometimes unrolling is illegal. In the last column, reasons why only intra-iteration SIMD parallelism can be exploited are listed. The first reason is it is straight-line code and no loop exists. The second is it is a while-do loop. The third is dependence hinders unrolling. The fourth is that memory access is not continuous between two consecutive iterations.

Examples are all double, so vector factor is 2, 4 and 8 respectively for SSE, AVX and IMCI. Results are shown in Fig. 14. Sufficient vectorization method is applied for SSE because its vector factor is 2. The speedup of inl1130, smvp and solve are 1.19, 1.34 and 1.14 respectively. The speedup of binvcrhs, buts and z_solve are 1.86, 1.54 and 1.28 respectively.

Vector factor is 4 for AVX, so inl1130, smvp and solve are all exploited by insufficient vectorization method. The speedup of them is 1.33, 1.64 and 1.37 respectively. Because in binvcrhs the SIMD parallelism degree is 2, 3, 4 respectively, so both insufficient vectorization method and sufficient vectorization method are used, buts and z_solve are the same case. The speedup of them is 2.20, 2.36 and 1.88 respectively.

Vector factor is 8 for IMCI, and thus only insufficient vectorization method is exploited. The speedup of inl1130, smvp and solve are 1.29, 1.50 and 1.24 respectively. The speedup of binvcrhs, buts and z_solve are 2.15, 2.36 and 1.81 respectively.

The last bar of Fig. 14 is the average speedup for SSE, AVX and IMCI, which are 1.39, 1.80 and 1.73 respectively. The largest speedup of loop-aware is 1.20 for SSE because it exploits sufficient vectorization. The largest speedup of VMSP is AVX, which is 1.80.

**Table 5** Examples of low inter-iteration SIMD parallelism

| Kernel | Program | Percentage | SIMD parallelim |
|---|---|---|---|
| E_c3d | 454.calculix | 69% | 3 |
| x_solve | SP | 17% | 3 |
| mat_times_vec | 410.bwaves | 30% | 5 |
| compute_rhs | BT | 16% | 5 |
| Rhs | LU | 24% | 5 |

## 5.4 Evaluating the Kernels of Low Inter-Iteration SIMD Parallelism

Loops of low inter-iteration SIMD parallelism are also very common in the real-world application. We choose kernels that large execution time are taken as examples, which are shown in Table 5. It lists kernel names, from which program, the percentage of execution time and SIMD parallelism degree. They are all the kernels of programs, and vectorizing them can be highly improved performance. Low inter-iteration SIMD parallelism caused by true dependence is seldom in real-world application, and examples we chose are all because of small iteration numbers.

Examples are all double precision data elements. Therefore vector factor is 2, 4 and 8 respectively for SSE, AVX and IMCI. Sufficient vectorization method is applied

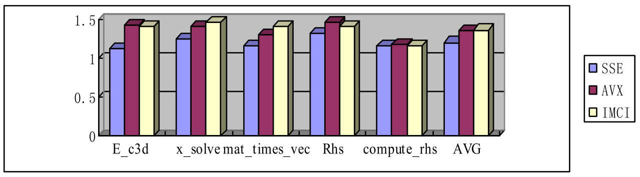


**Fig. 15** Result of low inter-iteration SIMD parallelism.

for SSE because its vector factor is 2. Insufficient vectorization method is applied for IMCI because its vector factor is 8. Insufficient vectorization method is applied to E_c3d and x_solve for AVX because the inter-iteration SIMD parallelism degree of E_c3d and x_solve are 3, while other kernels are exploited by sufficient vectorization method for AVX. Results are shown in Fig. 15. The speedup of E_c3d and x_solve for AVX is 1.42 and 1.40 respectively, 1.40 and 1.46 for IMCI, 1.12 and 1.24 for SSE.

Sufficient vectorization method can be applied on SSE and AVX for mat_times_vec, compute_rhs and Rhs. The speedup of AVX is 1.30, 1.18 and 1.45 respectively. The speedup of SSE is 1.15, 1.15 and 1.32 respectively. Insufficient vectorization method is applied on IMCI, where mask writing is used. The speedup of these three kernels is 1.40, 1.15 and 1.40 respectively.

The last bar in Fig. 15 is the average speedup of these five kernels, which is 1.20, 1.35 and 1.36 for SSE, AVX and IMCI respectively. The largest speedup of loop-aware is 1.20 for SSE, because it exploits sufficient vectorization. The largest speedup of VMSP is IMCI, which is 1.36.

The theoretical speedup of AVX is double of SSE, besides AVX is compatible with SSE. When speedup achieved by insufficient vectorization method for AVX is lower than SSE, the lower 128-bit can be used which is like SSE, and hence the performance achieved by AVX will not be lower than SSE. When sufficient vectorization method is used for SSE, the performance cannot be larger than AVX, so vector length is the key factor that restricts SIMD performance when SIMD parallelism degree is larger than vector factor. From the comparison of AVX and IMCI, some kernels performance for AVX is larger than IMCI, though the vector length of IMCI is larger than AVX. This is because IMCI is not compatible with AVX, and IMCI do not support the flexible partial use of vector register. Hence, for longer vector register, like more than 512 bit, supporting flexible partial use of vector register is the key to SIMD performance.

### 5.5 Evaluating the Kernels of Non-Unit Stride Memory Access

Real-world applications often have non-unit stride memory access in their kernels. Eight kernels are chosen for our evaluation. In SPEC2006, we select toffoli, C-not and sigma_x from 462.libquantum. In SPEC2000, we select interp and rprj3 from 172.mgrid. Furthermore, we choose complex dot, complex multiplication and FFT which are typical kernels in muti-media field.

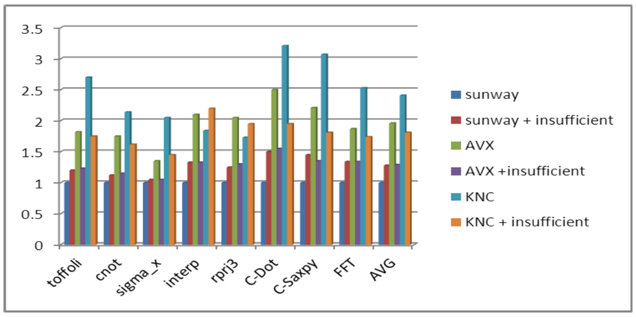


**Fig. 16** Results of non-unit stride memory kernels.

Stride of the examples are all 2, so theoretic speedup of insufficient vectorization method is half of sufficient vectorization method. Test results are shown in Fig. 16. Because kernels are all double or long, it is useless when insufficient vectorization method is applied for SSE. We take sunway, AVX and IMCI as test platforms. In Fig. 16, AVX stands for speedup obtained by sufficient vectorization method, and AVX+ insufficient is speedup of insufficient vectorization method, and the others are the same.

Sunway only provides extract and insert instructions, which do not support flexible permutation, and hence it is difficult to implement sufficient vectorization method. Though extract and insert instructions can be used to generate vectorized code, the efficiency is still lower than scalar code. In our work, we consider the speedup of sufficient vectorization method as 1 on sunway. Though insufficient vectorization method is not able to make full use of vector register, the theoretic speedup will be half of sufficient vectorization method because stride is 2. When insufficient vectorization method is applied, the speedup of toffoli, cnot and sigma_x is 1.20, 1.12 and 1.05 respectively. The speedup of interp and rprj3 is 1.33 and 1.25 respectively. The speedup of C-Dot, C-Saxpy and FFT is 1.50, 1.45 and 1.28 respectively. In sunway the average speedup is 1.28. Hence, when the platform do not support shuffle, insufficient vectorization method can exploit SIMD parallelism of non-unit stride memory access kernels.

Extract and insert instructions are also used to implement insufficient vectorization method for AVX. The speedup of toffoli, cnot and sigma_x is 1.23, 1.15 and 1.05 respectively. The speedup of interp and rprj3 is 1.33 and 1.30 respectively. The speedup of C-Dot, C-Saxpy and FFT is 1.55, 1.41 and 1.35 respectively. In AVX the average speedup is 1.29, which is higher than sunway. This is because AVX supports loadu instruction by hardware and sunway has to use shift to implement unaligned load, we can see that our cost model works. As fixed shuffle is supported, the performance for AVX is better when sufficient vectorization method is applied. The speedup of toffoli, cnot and sigma_x is 1.82, 1.75 and 1.35 respectively. The speedup of interp and rprj3 is 2.10 and 2.05 respectively. The speedup of C-Dot, C-Saxpy and FFT is 2.50, 2.21 and 1.87 respectively. From the comparison of sufficient vectorization method and insufficient vectorization method on AVX, we can see that

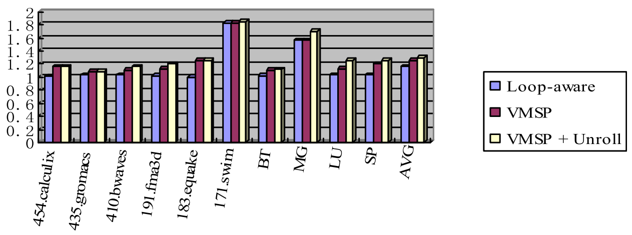


**Fig. 17** Comparison of speedups for loop-aware, VMSP and VMSP with unrolling.

when powerful permutation is supported, sufficient vectorization method can obtain better performance than insufficient vectorization method.

The vector length of IMCI is double than sunway and AVX, and thus theoretic speedup is also double. Mask writing memory is supported by IMCI, which is of great benefits for insufficient vectorization method. The speedup of toffoli, cnot and sigma_x is 1.75, 1.62 and 1.45 respectively. The speedup of interp and rprj3 is 2.20 and 1.95 respectively. The speedup of C-Dot, C-Saxpy and FFT is 1.95, 1.81 and 1.74 respectively. For sufficient vectorization method, the speedup of toffoli, cnot and sigma_x is 2.70, 2.14 and 2.05 respectively. The speedup of interp and rprj3 is 1.84 and 1.73 respectively. The speedup of C-Dot, C-Saxpy and FFT is 3.21, 3.07 and 2.53 respectively. From the results of interp and rprj3, we can see that the speedups achieved by insufficient vectorization are higher than those of sufficient vectorization method. This is due to sophisticated permutation modes are needed by interp and rprj3. If compiler generates codes by permutation more instructions are needed. However, writing to memory with mask makes insufficient vectorization method efficiently.

The insufficient vectorization method of IMCI is the same as sufficient vectorization method of AVX, for double it is 4. When permutation mode is complex, the speedup of insufficient vectorization method on IMCI is higher than that of sufficient vectorization method on AVX, such as sigma_x and interp. While sufficient vectorization method on AVX obtains better performance for kernels whose permutation model is simple, such as toffoli and cnot. Because vector register length of IMCI is longer than that of AVX and sunway, writing to memory with mask is also very efficient on IMCI. Therefore the performance achieved by insufficient vectorization method on IMCI is much higher than that of on AVX and sunway.

### 5.6 Full Program Performance Test

In this section, full program performance is implemented to compare VMSP with loop-aware on KNC. Programs are selected from SPEC2006, SPEC2000 and NPB. The reference input is used for SPEC benchmark and Class B input is used for NPB. Results are shown in Fig. 17. We test loop-aware, VMSP and VMSP with unrolling. Unroll times is generated heuristically by compiler.

Because loop-aware method cannot vectorize 454.calculix, 435.gromacs and 410.bwaves effectively, the speedup obtained is 1.01, 1.03 and 1.03 respectively. While VMSP can exploit the SIMD parallelism of these programs sufficiently. The speedup is 1.16, 1.08 and 1.10 respectively, which are much higher than that of loop-aware. After unrolling vectorized loops, the speedup of 410.bwaves improves to 1.16. This is because mat_times_vec is a five nest loop, unrolling completely can not only eliminate the cost of loops but also enhance instruction-level parallelism.

The SIMD parallelism degrees of both 191.fma3d and 183.equake are lower than vector factor, and unrolling is illegal which is illustrated in the previous section, so loop-aware cannot exploit successfully and obtains the speedup of 1.02 and 1.0 respectively. VMSP can vectorize successfully, and the speedups achieved by it are larger than loop-aware, which is 1.13 and 1.25. Unrolling is not suitable for 183.equake because the kernels are while-do, but 191.fam3d get a high speedup when unrolling is applied, which is 1.20.

The kernels of BT, LU and SP do not have enough SIMD parallelism, as shown in previous section. VMSP can vectorize more loops than loop-aware, so the speedups of VMSP are much higher than loop-aware. The speedups of loop-aware are 1.02, 1.03 and 1.03 respectively, while the speedups of VMSP before loop unrolling is applied are 1.10, 1.13 and 1.20 respectively. After using loop unrolling, the speedups of VMSP are 1.12, 1.25 and 1.25 respectively because most of the kernels are three levels loop nest. Improvement is not obvious when unrolling is used.

The SIMD parallelism of 171.swim and MG are enough, loops which can be vectorized by insufficient vectorization method are also vectorized by sufficient vectorization method. Therefore, the speedup of two methods achieved is the same, which are 1.83 and 1.56 respectively. When loop unrolling is applied, performance improves to 1.85 and 1.70.

The last bar of Fig. 17 is average speedup of these three methods. The speedup of loop-aware method is 1.16. The speedup of VMSP is 1.26 before unrolling. After unrolling is applied the speedup of VMSP is 1.30. Hence, the speedup achieved by VMSP is 12.1% higher than loop-aware.

## 6. Related Work

Currently, there are two major vectorization algorithms. Loop-based algorithm can combine multiple iterations of a loop into a single iteration of vector instructions. Superword level parallelism (SLP) targets straight-line code.

As SIMD extensions are similar to vector processors, SIMD compilation techniques first originated from traditional vectorization techniques for vector processors [12]. Loop-based algorithms are based on the notion of data dependence along with several classical loop transformations. Strip-mining, scalar expansion, reduction processing, loop distribution and outer-loop vectorization are major loop transformation techniques used to enhance parallelism [13]. A cost model and a loop transformation framework is proposed to extract subword parallelism opportunities and to select an optimal strategy among them, which is based on polyhedral compilation, leveraging its representation of memory access patterns and data dependences as well as its expressiveness in building complex sequences of transformations [14].

SLP has been recently introduced to taking advance of SIMD extensions for the straight-line code. Larsen and Amarasinghe [15] are the first to present an automatic vectorization technique based on vectorizing parallel scalar instructions with no knowledge of any surrounding loop. Other straight-line code vectorization techniques which depart from the SLP algorithm have also been proposed in the literature. A back-end vectorizer in the instruction selection phase based on dynamic programming was introduced by Barik [16]. The approach is different from most of vectorizers as it is close to the code generation stage and can make more informed decisions on the costs involved with the instructions generated. Holewinsky et al. [17] propose a technique to detect and exploit more parallelism by dynamically analyzing data dependences at runtime, and thus guiding vectorization. Liu et al. [5] present a vectorization framework that improves SLP by performing a more complete exploration of the instruction selection space while building the SLP tree.

Control flow is another factor that inhibits exploiting SIMD parallelism. Currently, there are two major vectorization control dependence algorithms. Shin et al. [18] introduces an SLP algorithm with a control-flow extension that makes use of predicated execution to convert control flow into data-flow, thus allowing it to become vectorized. They emit select instructions to perform the selection based on the control predicates. The other code generation technique is vectorizing data flow in each basic block individually, and then maintains data dependency of variables for SIMD operations by inserting assignment statements for the variables [19]. Programs with control flow can be vectorized without modifying control flow structure. In addition, function-level vectorization has also been researched in [20]. Speculative dynamic vectorization algorithm is also presented to reorder speculatively ambiguous memory references to uncover vectorization opportunities [21], [22].

In spite of similarities, there exist several differences between traditional vector machine and SIMD extensions [23]. The most significant differences arise from the weaker memory units of SIMD extensions. In contrast to those of vector processors, the memory units of SIMD extensions usually do not support scatter/gather operations. They only allow to access memory locations that are aligned at vector register length boundaries. Eichenberger et al. proposes a method for vectorizing loops with misaligned stride-one memory references [24]. Ren et al. optimizes a sequence of multiple data reorganization for statically misaligned data [25]. Nuzman et al. extends a loop-based vectorization technique to handle computations with non-unit stride accesses to data, where the strides are the powers of 2 [26]. The methods to address unaligned memory access can be divided into three categories: first is using multiple times of memory accesses alignment and then shifting [23], [27]–[29]; second is using loop transformation [25], [30]; third is that hardware supports for unaligned visit deposit [26].

In [32], to ensure a correct and efficient execution of partial vector operations on SIMD extensions are introduced. In our work, we call it insufficient vectorization. How to generate memory accesses for insufficient vectorization without using special hardware support such as predicated load and store is discussed in our paper. Furthermore, the adaptation scene of insufficient vectorization is classified in detail. We proposed loop SIMD parallelism to guide the choice of vectorization methods, loop unrolling oriented to vectorized loops is also proposed.

## 7. Conclusions

In this paper, we present insufficient vectorization, a novel method to exploit low SIMD parallelism for the growing SIMD width. First, the adaptation scene of insufficient vectorization is analyzed. Then the computation method for intra-iteration and inter-iteration SIMD parallelism is given. Afterward, loop vectorization parallelism guides the choice of which vectorization method to be used in order to fully exploit the parallelism of loops. Finally, loop unrolling oriented to vectorized loops is proposed to improve the instruction-level parallelism further. Experimental results on three platforms show that compared with loop-aware method which is widely used in current compilers, the methods proposed in this paper boost the recognition rate by 107.5% and the performance is improved by 12.1%.

SIMD parallelism in programs can be exploited from multiple granularities, such as loops, basic blocks and so on. In this paper, we present how to exploit the SIMD parallelism of loops completely. How to exploit function-level vectorization efficiently is to be solved. Furthermore, to obtain optimal SIMD performance for a specific application, which one or more granularities to vectorize, is also to be solved.

## Acknowledgments

We would like to acknowledge the anonymous referees for their invaluable comments and suggestions on this paper. Our work is supported by the HEGAOJI Major Project of China under Grant No.2009ZX01036-001-001-2 and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing No. 2013A11.

### References

[1] T. Mytkowicz and M. Marron, "Single-Core Performance is Still Relevant in the Multi-Core Era," SIMD compiling technology review[C].

[2] M. Hassaballah1 and S. Omran, "A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications [J]," The Computer Journal, vol.51, no.6, pp.630–650, 2008.

[3] R. Leißa, S. Hack, and I. Wald, "Extending a C-like Language for Portable SIMD Programming," Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP), New Orleans, Louisiana, USA. pp.65–74, 2012.
[4] D. Nuzman and A. Zaks, "Outer-Loop Vectorization: Revisited for Short SIMD Architectures," Proceedings of the 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp.2–11, 2008.
[5] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir, "A Compiler Framework for Extracting Superword Level Parallelism," Proceedings of the 2012 Conference on Programming Language Design and Implementation (PLDI), pp.347–358, 2012.
[6] V. Porpodas, A. Magni, and T.M. Jones, "PSLP: Padded SLP Automatic Vectorization," Proceedings of the 2015 Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2015.
[7] S. Kim and H. Han, "Efficient SIMD Code Generation for Irregular Kernels," Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP), pp.55–64, 2012.
[8] Y. Park, S. Seo, H. Park, H.K.Cho, and S. Mahlke, "SIMD Defragmenter: Efficient ILP Realization on Data-parallel Architectures," Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.363–374, 2012.
[9] A. Ramachandran, J. Vienne, R. Van Der Wijngaart, L. Koesterke, and I. Sharapov, "Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi," Proceedings of 42nd International Conference on Parallel Processing (ICPP), 2013.
[10] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors," Proceedings of the 2013 International Conference on Supercomputing, pp.273–282, 2013.
[11] X. Huo, B. Ren, and G. Agrawal, "A Programming System for Xeon Phis with Runtime SIMD Parallelization," Proceedings of the 2013 International Conference on Supercomputing, pp.283–292, 2014.
[12] R. Allen and K. Kennedy, Optimizing compilers for modern architectures, Morgan Kaufmann, 2001.
[13] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-Model Guided Loop-Nest Auto-Vectorization," Proceedings of the 2009 International Conference on Parallel Architectures and Compilation Techniques (PACT), 2009.
[14] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When Polyhedral Transformations Meet SIMD Code Generation," Proceedings of the 2013 Conference on Programming Language Design and Implementation (PLDI), pp.127–138, 2013.
[15] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.145–156, June 2000.
[16] R. Barik, J. Zhao, and V. Sarkar, "Efficient Selection of Vector Instructions using Dynamic Programming," Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010.
[17] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Dynamic trace-based analysis of vectorization potential of applications," Proceedings of the Conference on Programming Language Design and Implementation (PLDI), pp.371–382, 2012.
[18] J. Shin, M. Hall, and J. Chame, "Superword-Level Parallelism in the Presence of Control Flow," Proceedings of the 3rd Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), New York, ACM Press, pp.165–175. 2005.
[19] H. Tanaka, Y. Ota, N. Matsumoto, T. Hieda, Y. Takeuchi, and M. Imai, "A New Compilation Technique for SIMD Code Generation across Basic Block Boundaries," Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC), pp.101–106, 2010.
[20] R. Karrenberg and S. Hack "Whole-Function Vectorization," Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2011.
[21] R. Kumar, A. Martínez, and A. Gonzalez, "Speculative Dynamic Vectorization for HW/SW Codesigned Processors," Proceedings of the 2012 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp.459–460, 2012.
[22] M. Haque and Q. Yi, "Past dependent branches through speculation," Proceedings of the 22nd International Conference on Parallel Architecture and Compilation Techniques (PACT), Washington DC: IEEE Computer Society, 2013.
[23] G. Ren, P. Wu, and D. Padua, "A preliminary study on the vectorization of multimedia applications for multimedia extensions," Languages and Compilers for Parallel Computing, vol.2958 of Lecture Notes in Computer Science, pp.420–435, 2004.
[24] A.E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD architectures with alignment constraints," Proceeding of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI), ACM Press, New York, pp.82–93, 2004.
[25] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Performance impact of misaligned accesses in SIMD extensions," Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing, pp.334–342, 2006.
[26] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for SIMD," Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI), pp.132–143, 2006.
[27] H. Chang and W. Sung, "Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware," Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), pp.167–176, 2008.
[28] G. Ren, P. Wu, and D. Padua, "Optimizing data permutations for SIMD devices," Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI), pp.118–131, 2006.
[29] L. Huang, L. Shen, Z. Wang, W. Shi, N. Xiao, and S. Ma, "SIF: Overcoming the Limitations of SIMD Devices via Implicit Permutation," Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA), pp.355–366, 2010.
[30] L. Yuxiang, S. Hui, and C. Li, "Vectorization-oriented local data regrouping," Computer System, vol.30, no.8, pp.1529–1534, 2009.
[31] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, G. Yang, "The Sunway TaihuLight supercomputer: system and applications," Sci. China Inf. Sci., vol.59, no.7, 072001, 2016.
[32] H. Zhou and J. Xue, "A Compiler Approach for Exploiting Partial SIMD Parallelism," ACM Transactions on Architecture and Code Optimization (TACO), vol.13 no.1, April 2016.

**Wei Gao** was born in 1988. Currently he is a Ph.D. candidate in State Key Laboratory of Mathematical Engineering and Advanced Computing (MEAC), Zhengzhou, China. His research interests are high performance computing and advanced compiling, including parallelization, auto-vectorization.

**Lin Han** was born in 1978. Currently he is an associate professor in State Key Laboratory of Mathematical Engineering and Advanced Computing (MEAC), Zhengzhou, China. His research interests are high performance computing and optimizing compiler.

**Rongcai Zhao** was born in 1957. Currently he is an professor in State Key Laboratory of Mathematical Engineering and Advanced Computing (MEAC), Zhengzhou, China. His research interests are high performance computing and optimizing compiler.

**Yingying Li** was born in 1984. Currently she is an associate professor in State Key Laboratory of Mathematical Engineering and Advanced Computing (MEAC), Zhengzhou, China. Her research interests are high performance computing and optimizing compiler.

**Jian Liu** was born in 1990. Currently he is a Ph.D. candidate in College of Communication and Information Engineering at Nanjing University of Posts and Telecommunications (NJUPT), Nanjing, China. His research interests are in the areas of stochastic resonance (SR) and its applications, including signal detection, signal transmission, and digital communication system.