# An SIMD Code Generation Technology for Indirect Array

Pengyuan Li, Rongcai Zhao, Qinghua Zhang, and Lin Han

***Abstract*—Due to disjoint memory references and non-aligned memory references, existing SIMD compilers can't vectorize loops containing indirect array utilizing SIMD (single instruction multiple data) instructions. However, addressing this problem is inevitable, since many important applications extensively use this program pattern to reduce memory and computation requirement. In this paper, we propose a new efficient code generation technique for indirect array. For an irregular indirect array access, we adopt two separately registers to store the array base and the index address. It significantly contributes to the performance improvement by vectorizing more loops and reducing the overheads. We also developed this method in our auto-vectorization compiler SW-VEC. The experimental results show that the proposed method can translate applications within direct array access into high-performance targeted vectorized codes, thereby advancing the execution efficiency adequately.**

***Index Terms*—Code generation, indirect array, vectorization.**

## I. Introduction

Most modern processor architectures employ SIMD (single instruction multiple data) units. By using SIMD instructions, processors can simultaneously execute the same operation on multiple data packed into one register as illustrated in Fig. 1. For programmers, the SIMD processor, with a low power and low complexity processor design, is very effective in executing programs containing large data-level parallelism such as multimedia applications [1], [2].

In recent years, several auto-vectorizing compilers such as IBM's XL compiler, INTEL's icc compiler, ARM's RealView compiler, GNU's Open64 compiler and GCC compiler have been introduced for efficient SIMD code generation [3]-[7]. Various impactful techniques have been applied to automatically generate SIMD code and to address the difficulties during vectorizing such as data permutations [8], interleaved data [9], etc. However, the optimizing approaches employed by those compilers still cannot drastically eliminate the irregular and non-aligned obstacles.

Consider an indirect array reference $X[idx[i]]$, where the accesses to array $X$ are dictated by the value computed by array $idx$. The actual element of array $X$ which is accessed with an index variable $i$ is unknown at compile time. As a consequence, compilers are hardly able to infer any useful properties of indirect reference pattern such as alignment, adjacency, and dependence information.

In our proposed efficient code generation method for loops containing indirect memory references, the array base and array index are separately packed into two vector register variable, and conduct a vector with the input operands of the base register and the index register variable by one special SIMD ADD operation.

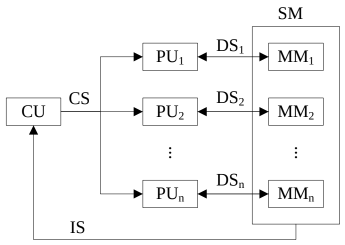


Fig. 1. Structure of SIMD unit.

The rest of this paper is organized as follows. Section II briefly describes the related work. Then explaining a new efficient code generation technique detailed in Section III. Section IV syllabify bewrites one benefit model to guarantee the basic yield. The performance evaluation results are shown in Section V. Finally concluding remarks are made in Section VI.

## II. Related Works

There have been several research works on optimizing loops with indirect array reference. One SIMD compilation method is employed to address the problems discussed in the previous sections in [10]. This proposed method examines the data-flow graph (DFG) of the loop body to exploit parallelism with Superword Level Parallelism (SLP) [11] algorithm and is designed to replace unnecessary gather and scatter operations by scalar operations. It is often use full when the data reorganization costs are lower than the SIMD instruction benefits. Another efficient method is applied in [12] by utilizing a special hardware to improve the performance of SIMD processors. This SIMD processor is only designed with the hardware support on ARMv4 architecture [13] without universality. Xin *et al.* [14] try to vectorize an irregular reduction kernel with a large number of gather and scatter operations. Wu *et al.* [15] try to resolve a coalesced memory access problem within the context of the GPU architecture.

Recently, the SIMdD (Single Instruction Multiple disjoint Data) architecture contains a multi-port memory unit which allows accessing disjoint data [16]. Although the SIMdD supports non-aligned and irregular data access efficient, it is based on costly multi-port memory.

In addition, several SIMD processors provide pack, permutation and shuffle instructions to arrange data within vector registers in various patterns. The methods for

Manuscript received November 3, 2014; revised March 2, 2015.
The authors are with State Key Laboratory of Mathematical Engineering and Advanced Computing, China (email: {pengyuan_li, pengyuan_li, strollerlin}@163.com, rczhao126@126.com).

generating pack or permutation instructions are presented in paper [8].

## III. An Efficient SIMD Code Generation Techology

In this section, we propose an SIMD code generating method to address the problems discussed in previous sections. Fig. 2 shows the SW_VEC compiler's code generation infrastructure. First, we rely on the front_end to parse the source code, perform pre_optimization and emit the WHIRL intermediate representation (IR) [17], [18] which is binary tree form [19]. Then, turn to each innermost loop, we analysis the loop form to search the illegal loop and test the data dependency relations among loop iterations that might prohibit vectorization. After that, we analysis the scalar and SIMD operations based on the targeted SIMD instructions. Next, the benefit analysis phase is performed to determine whether the code generating operation is profitable or not. At last, compiler translates the scalar WHIRL IR into SIMD WHIRL IR automatically.

In this work, the validity and correctness of SIMD codes which are generated with a better method is our aim. Thereby we study the benefit analysis phase and the vectorization transform phase with most energy. Here, how to transform the scalar code to SIMD code is described in detail. The other phase will be bewrited in next section.

In order to vectorize loops containing indirect array references, we employ one effective code generating method during vectorization transform phase. The foundational principle of translating SIMD programs is shown in Fig. 3. For an irregular data access array such as $X[idx[i]]$, we store the base address in a base_register which is one vector variable, and store the array index into the index_register, then conduct the object vector through an special SIMD ADD instruction.

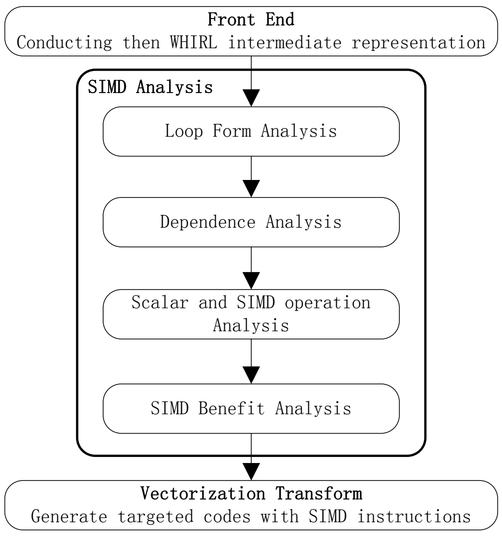

Front End
Conducting then WHIRL intermediate representation
SIMD Analysis
Loop Form Analysis
Dependence Analysis
Scalar and SIMD operation Analysis
SIMD Benefit Analysis
Vectorization Transform
Generate targeted codes with SIMD instructions

Fig. 2. SIMD code generation infrastructure in SW_VEC.

When the SIMD ADD instruction is executed, each data element of the base_register and the index_register execute the add operation parallel, the corresponding values in memory are extracted at the same time as well. It is success to get the vector unit of $X[idx[i]]$.

## IV. Benefit Analysis

In this section, we will use one benefit model to make sure the profit of our method. Comprehensive consideration of various factors in the processing of SIMD program, we describe the model as follows:

$$C_{simdopt} = \sum C_{vec\_intr} + \sum (C_{vec\_load} + C_{vec\_store}) + C_s + C_a \quad (1)$$

In this model, $C_{simdopt}$ indicates the total income which is obtained from the vectorization, $C_{vec\_intr}$ represents the benefits getting from each vector intrinsic which is not load or store and $C_{vec\_load}$ represents the vector load's profits and $C_{vec\_store}$ is SIMD store's benefits.

In addition, $C_s$ expresses the costs of data regrouping for disjoint memory accesses, $C_a$ indicate the cost of align optimization for non-aligned accesses. We hypothesized that $C_{vload}$ expresses the overhead of regrouping one load intrinsic and $C_{vstore}$ expresses the overhead of recomposing one store intrinsic, the cost of shift and merge operation for vectorizing non-aligned codes are indicated with $C_{vshift}$ and $C_{vmerge}$. So, $C_a$ and $C_s$ can be described as:

$$C_s = \sum (C_{vload} + C_{vstore}) \quad (2)$$

$$C_a = \sum (C_{vshift} + C_{vmerge}) \quad (3)$$

If there are no discrete accesses and non aligned accesses, $C_s = 0$ and $C_a = 0$.

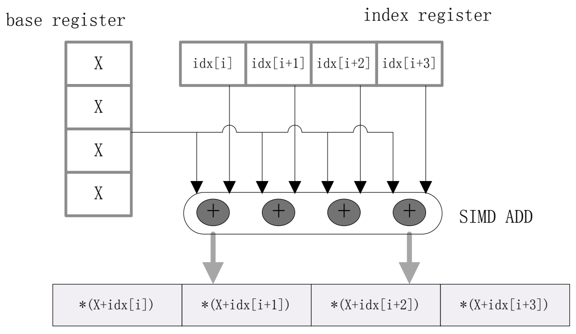


Fig. 3. SIMD Code generation principle for indirect array reference.

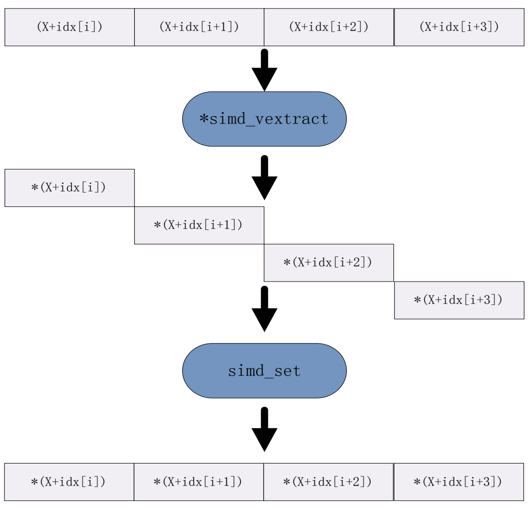


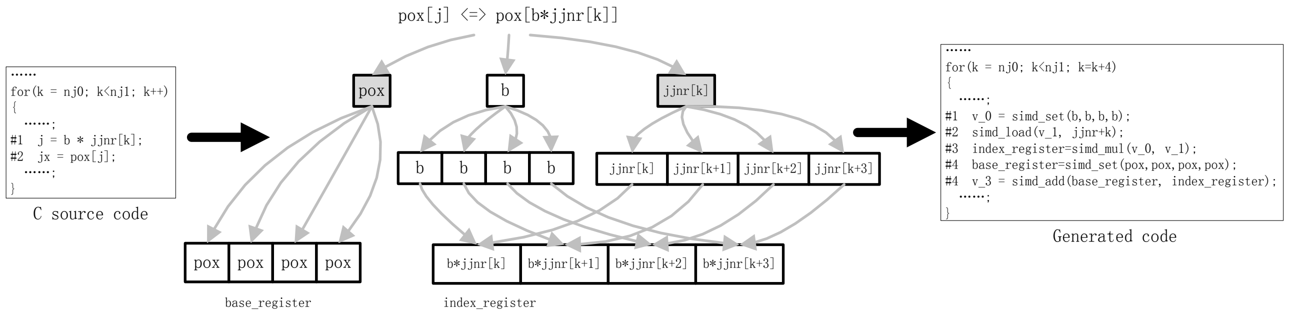
Fig. 4. Special SIMD ADD using regrouping instruction.

## V. Experimental Results

In this section, we will test the performance of this SIMD code generation method. We implement this method on the SW_VEC vectorizing tool and run the program on Red hat Enterprise AS 5.0. Runtime environment is Sunway BlueLight Server, and the processor is SW-1600. We compile three benchmarks from the SPEC2006 [21] benchmark unites.

### A. SPEC2006 Benchmarks

Table I lists the benchmarks used in this experiment. Since this work addresses challenges due to array indirections, we only collect programs whose inner-most loops contain array indirections, by examining the most time-consuming functions of the floating applications from SPEC CPU2006 [21]. In Table I, each kernel has several indirect references for both read and write accesses. The nature of the indirect references in the kernels is similar to the example in Fig. 5. We named the kernels after their enclosing functions. The third column of Table I shows the fraction of each function's execution time. Except for FORMS, each function consists only of a single loop nest that contains the extracted kernel. In case of FORMS, the function has nine loop nests in similar form.



```
……
for(k = nj0; k<nj1; k++)
{
  ……;
#1  j = b * jjnr[k];
#2  jx = pox[j];
  ……;
}
```

```
……
for(k = nj0; k<nj1; k=k+4)
{
  ……;
#1  v_0 = simd_set(b,b,b,b);
#2  simd_load(v_1, jjnr+k);
#3  index_register=simd_mul(v_0, v_1);
#4  base_register=simd_set(pox,pox,pox,pox);
#4  v_3 = simd_add(base_register, index_register);
  ……;
}
```

pox[j] <=> pox[b*jjnr[k]]
pox
b
jjnr[k]
b b b b
jjnr[k] jjnr[k+1] jjnr[k+2] jjnr[k+3]
pox pox pox pox
b*jjnr[k] b*jjnr[k+1] b*jjnr[k+2] b*jjnr[k+3]
base_register
index_register
C source code
Generated code

Fig. 5. SIMD code generation flow for indirect array reference.

TABLE I: The SPEC2006 Benchmark Kernels

| Application | Kernel | Extracted From(Function) | Exec. Time | General Category |
|---|---|---|---|---|
| 435.GROMACS | INL | INL1130() | 66.11% | CHEMISTRY/MOLECULAR DYNAMICS |
| 444.NAMD | CPEF | CALC_PAIR_ENERGY_FULLELECT() | 10.76% | STRUCTURAL BIOLOGY |
| 416.GAMES | FORMS | FORMS() | 16.09% | QUANTUM CHEMICAL COMPUTATIONS |

### B. Speedups from Our Framework

Fig. 7 shows the speedup factors achieved by the proposed SIMD optimization technique that is compared with the conventional SIMD code generation method.

But, modern SIMD units can't execute this SIMD ADD by an easy SIMD instruction as previous said. We must do that with indirect ways. Regrouping instructions are widely implemented in Most SIMD processors such as extracting and inserting instructions. Some special regrouping operations such as pack and unpack is mapped directly to vec_pack and vec_unpack on AltiVec platform [8], [20]. In Fig. 4, we use a set of extracting instruction to resolve this problem and improve the applicability.

With this proposed way, the compilers can even vectorize loops containing irregular indirect array references. The vectorization procedure of this access pattern is described in Fig. 5. Firstly, the compiler searches for the statements in loop that contain the indirect access pattern. Secondly, compiler generates base_register and index_register for the indirect array. Thirdly, we conduct the SIMD codes with these special registers.

Moreover, the algorithm of this advanced technique is characterized in Fig. 6.

- SRC: Compiler execute each benchmark's source code.
- Conventional SIMD: Conventional SIMD code generation method is applied when compiler execute each benchmark.
- Special SIMD: The proposed SIMD code generation technique applied when compiler execute each benchmark.

```
producedure SIMDCodeGeneration(G = (N))
  N <- {n | n ∈ N, n is one node in loop}
  IA    //represent the indirect Array
    Vci <- ∅  //Vci candidates to SIMD for indirect array access nodes
    Vcr <- ∅  //Vcr candidates to SIMD for formal nodes
    for n ∈ N do
      if n ∈ IA then
        Vci <- Vci∪{n}
      else
        Vcr <- Vcr∪{n}
      end if
    end for
    for n ∈ N do
      if n ∈ Vci then
        // use the proposed code generation method
        SCALAR LOAD -> SIMD LOAD
        SCALAR STORE -> SIMD STPRE
        SCALAR COMPUTE -> SIMD COMPUTE
      else if n ∈ Vcr then
        //use the conventional code generation method
        SCALAR LOAD -> SIMD LOAD
        SCALAR STORE -> SIMD STPRE
        SCALAR COMPUTE -> SIMD COMPUTE
      else
        continue
      end if
    end for
end producedure
```

Fig. 6. Algorithm for this optimization technology.

### C. Evaluation

To evaluate the performance of the proposed technique, we measure the RPCC of each benchmark. Table II shows the test result. Data in Table II show, compared to the conventional SIMD vectorization method, the proposed technique can obviously improve the performance of INL, CPFE and FORMS by 160%, on average. However, in Fig. 7, the speedup of each benchmark is lower than the corresponding kernel in Table II. The reasons of this actuality can be summarized as follows:

- Each kernel is only part of the corresponding benchmark.

- The proposed technique may reduce the performance of some part of the benchmark.
- The outer loops consume maybe balance out the score of inner-most loop.

In addition, the implementation of this proposed method is controlled by a transformation option -vectorize_indirect in compiler.

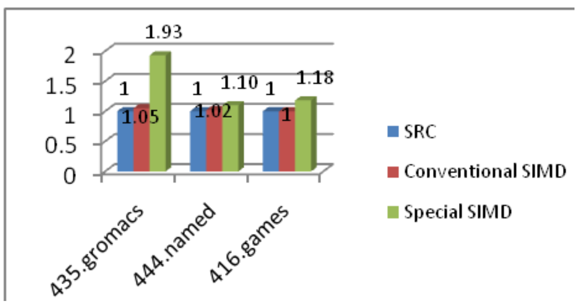


Fig. 7. Speedup factors achieved for three benchmarks.

TABLE II: THREE KERNELS'S RPCC TEST RESULT

| KERNEL | SRC | CONVENTIONAL SIMD | SPECIAL SIMD | SPEEDUP |
|---|---|---|---|---|
| INL | 1430420 | 1430420 | 397388 | 3.59 |
| CPFE | 1862646 | 1862646 | 980340 | 1.90 |
| FORMS | 304076 | 304076 | 126699 | 2.39 |

## VI. CONCLUSION AND FUTURE WORK

Array indirection causes several important challenges for SIMD compilation including disjoint memory references, unknown alignment, etc. Due to those challenges, most modern auto-vectorization algorithm is hardly able to achieve a certain performance improvement in the presence of array indirection. Several code generation methods are proposed to address the challenges arisen from array indirection [12], [14].

In this work, we proposed a new SIMD code generation technique to address the proposed challenges. The indirect array is separated into two independent parts, the base part and the index part. Then, the two different parts are corresponding packed into two vector registers, the base register and the index register. After that, we regroup the result of one SIMD add operation, and conduct the ideal codes. Our experiments on SW-1600 show that our proposed technique can improve the performance of several kernels in SPEC CPU2006 with the average speedup of 2.60.

The proposed framework can further be improved by integrating the other optimizing techniques such as alignment optimization [22], data permutation optimization [8] and loop transformation optimizations [23]. Since those techniques can reduce some obstacles that impede the exploration of SIMD parallelism.

## REFERENCES

[1] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proc. International Meeting on High Performance Computing for Computational Science*, *Lecture Notes in Computer Science*, 2011, vol. 6449, pp. 1-25.

[2] L. Bachega, S. Chatterjee, K. A. Dockserz, *et al*., "A high-performance SIMD floating point unit for BlueGene/L: Architecture, compilation, and algorithm design," in *Proc. the 13th International Conference on Parallel Architecture and Compilation Techniques*, September 2004, pp. 85-96.

[3] J. Lorenz, S. Kral, F. Franchetti, and C. W. Ueberhuber, "Vectorization techniques for the blue Gene/L double FPU," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 437-446, 2005.

[4] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, "Automatic intra-register vectorization for the intel architecture," *International Journal of Parallel Programming*, vol. 30, no. 2, pp. 65-98.

[5] R. Cravotta, "ARM updates realview development suit; Adds cortex-m1 Support,*" EDN Network*, 2007.

[6] B. Chapman, O. Hernandez *et al*., "An open64-based interactive program analysis tool for large application," in *Proc. the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2003.

[7] M. Levy, "Autovectorization for GCC compiler," *EDN Network*, vol. 7, 2007.

[8] G. Ren, P. Wu, and D. Padua, "Optimizing data permutations for SIMD devices," in *Proc. the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 118-131.

[9] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for SIMD," in *Proc. the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 132-143.

[10] S. Kim and H. Han, "Efficient simd code generation for irregular kernels," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 55-64, 2012.

[11] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proc. the Conference on Programming Language Design and Implementation*, 2000, pp. 145-156.

[12] H. Chang and W. Sung, "Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware," in *Proc. the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 167-176.

[13] Cortex-A8 Technical Reference Manual, *ARM*, 2007.

[14] X. Huo, B. Ren, and G. Agrawal, "A programming system for Xeon Phis with runtime SIMD parallelization," in *Proc. the 28th ACM international Conference on Supercomputing*, 2014, pp. 283-92.

[15] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu," in *Proc. the SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[16] D. Naishlos, M. Biberstein, A. Zaks *et al*., "Vectorizing for a SIMdD DSP architecture," in *Proc. the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, San Jose, California, USA, ACM.

[17] M. Kong, R. Veras, and K. Stock, "When polyhedral transformations meet SIMD code generation," in *Proc. the 2013 Conference on Programming Language Design and Implementation*, 2013.

[18] Overview of the open64 compiler infrastructure [EB/OL]. [Online]. Available: http://open64.sourceforge.net

[19] T. Hafer and W. Thomas, "Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree," *Automata, Languages and Programming*, 1987.

[20] J. Stewart, "An investigation of SIMD instruction sets," University of Ballarat School of Information Technology and Mathematical Sciences, 2005.

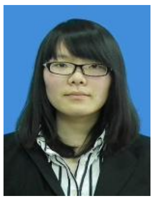
[21] Silicon Graphic International. (2010). WHIRL intermediate language specification [EB/OL]. [Online]. Available: http://cdnetworks-kr-l.dl.sourceforge.net/project/open64/open64/Documentation/whirl.pdf

[22] S. Larsen, E. Witchel, and S. P. Amarasin, "Increasing and detecting memory address congruence," in *Proc. the 2002 International Conference on Parallel Architectures and Compilation Techniques*, 2002, pp. 18-29.

[23] J. L. Henning, "SPEC CPU2006 benchmark," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1-17, 2006.

**Pengyuan Li** was born on 18 December, 1989 in China. He received the undergraduate degree in information warfare from Nanjing University of Science & Technology in 2012. Now, he is studying for the master's degree. He is working in the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research area is advanced compilation technology.

**Rongcai Zhao** was born in 1957. His main research fields are in parallel compilation, high performance computing and decompile technology. He is a tutor, a professor, and the senior member of CCF. Now, he is working in the State Key Laboratory of Mathematical Engineering and Advanced Computing.



**Qinghua Zhang** was born on 19 October, 1991 in China. She received the undergraduate degree in Qingdao University of Science and Technology. Now, she is studying for the master's degree. She is working in the State Key Laboratory of Mathematical Engineering and Advanced Computing. Her research area is advanced compilation technology.

**Lin Han** was born in 1978. His main research field is advanced compilation technology, and he is working in the State Key Laboratory of Mathematical Engineering and Advanced Computing.