

An Automatic Superword Vectorization in LLVM

Kuan-Hsu Chen, Bor-Yeh Shen, and Wu Yang

Department of Computer Science

National Chiao Tung University

Hsinchu City 300, Taiwan

{khchen, byshen, wuayang}@cs.nctu.edu.tw

Abstract—More and more modern processors support SIMD instructions for improving performance in media applications. Programmers usually need detailed target-specific knowledge to use SIMD instructions directly. Thus, an auto-vectorization compiler that automatically generates efficient SIMD instructions is in urgent need. We implement an automatic superword vectorization based on the LLVM compiler infrastructure, to which an auto-vectorization and an alignment analysis passes have been added.

The superword auto-vectorization pass exploits data-parallelism and convert IR instructions from primitive type to vector type. Then, in code generator, the alignment analysis pass analyzes every memory access with respect to those vector instructions and generates the alignment information for generate target-specific alignment instructions. In this paper, we use UltraSPARC as our experimental platform and two realignment instructions to perform misaligned access. We also present preliminary experimental results, which reveals that our optimization generates vectorized code that are 4% up to 35% speed up.¹

Index Terms—Auto-vectorization, UltraSPARC T2, VIS Instruction Set.

I. INTRODUCTION

In recent years, in order to improve the performance of multimedia applications, the *multimedia-extension instructions* have been added to most popular general-purpose microprocessors. These instructions operate on fixed-length vectors. Existing multimedia extensions, such as MMX/SSE for X86, Visual Instruction Set (VIS) for UltraSPARC, and AltiVec for PowerPC, are characterized as *single-instruction multiple-data* (SIMD) architectures. SIMD architectures exploit the data-parallelism [1] that exists in many multimedia applications. SIMD typically involves the simultaneous execution of the same instruction sequence on different elements in a large data set. For example, some SIMD instructions

simultaneously operate on a block of 128-bit data that is partitioned into four 32-bit integers. SIMD architectures include not only common arithmetic instructions, but also other instructions, such as data alignment, data type conversion, data reorganization, etc., that are also needed to prepare the data in a proper format for SIMD execution [2].

In an auto-vectorization technique, there are generally four major issues. First, not all code can be vectorized. For example, tasks with complicated control flow would not benefit from SIMD processors. Second, SIMD instructions are commonly used in in-line assembly routines or special library routines, which are usually not portable. Manually preparing in-line assembly routines is costly. The third issue is the alignment problem caused by SIMD load/store instructions. These instructions usually require that a block of data be aligned at machine word boundary. The last issue is that SIMD instructions are always architecture-specific. For example, MMX provides multiply-and add instruction operation ($a = a + (b \times c)$) but VIS does not.

We propose the design and implementation of an automatic superword vectorization based on the LLVM compiler infrastructure, which is widely used in the research community. LLVM supports powerful optimizations and analyses. However, LLVM currently does not yet support auto-vectorization. Therefore, we implement an auto-vectorization pass and an alignment analysis pass in LLVM. Auto-vectorization exploits data-level parallelism in innermost loop and converts IR instructions from primitive type to vector type. Because the operands in a vector operation must be properly aligned, if data is mis-aligned, additional instructions must be used to access correct data. These additional instructions incur extra penalty. Therefore, we use an alignment-analysis pass that detects mis-aligned load/store and reduces realignment instruction use. The alignment information is used by the target code generator to generate target-specific alignment instructions. The optimization and analysis passes operate on LLVM Intermediate Representation (IR).

¹The work reported in this paper is partially supported by National Science Council, Taiwan, Republic of China, under grants NSC 96-2628-E-009-014-MY3, NSC 98-2220-E-009-050, and NSC 98-2220-E-009-051 and a grant from Sun Microsystems OpenSparc Project.

The rest of this article is organized as follows. Related work and an introduction of LLVM are given in section 2. A detailed description of our auto-vectorization and alignment analysis is provided in section 3. The experimental results are presented in section 4. Section 5 is the conclusion and future work.

II. BACKGROUND

This section reviews current auto-vectorization techniques and the alignment mechanisms. In addition, we introduce the LLVM compiler infrastructure briefly.

A. Overview of Auto-Vectorization

Vectorization is a process which translates sequential loops into parallel versions that utilize SIMD instructions [3]. SIMD instructions are normally used within in-line assembly routines or processor-specific libraries. However, this approach is tedious, since a better alternative is to use a compiler that automatically issues SIMD instructions. Many researchers have studied auto-vectorization [1, 3–7]. There are two types of vectorization, loop-based vectorization and basic-block vectorization (i.e., superword level parallelism, SLP).

Loop-based vectorization attempts to explore data-level parallelism from a loop nest. In the first step, a loop-based vectorizer creates a data-dependence graph and identifies strongly connected components in the dependence graph. In a dependence graph, nodes represent statements and edges denote dependences between statements. Note that nodes in the same strongly connected component are not vectorizable because they have cyclic dependences. Next, the vectorizer combines each strongly connected component into a *piblock* [1] and rebuild dependences between operations in different piblocks. The resulting graph contains no cyclic dependences. Next, the vectorizer performs a topological sort of the graph to determine a valid execution order of the piblocks, and then emit a vectorized statement for each vectorizable node. Finally, the vectorizer constructs a loop to execute operations in the piblock of original program order.

The basic-block auto-vectorization (SLP) packs *independent isomorphic statements* in a basic block. Isomorphic statements are those that contain the same operations in the same order [3], for example, Fig. 1 shows the four isomorphic statements. They can be executed in parallel because they are data independent with each other. In addition, the loops and non-iterative programs could benefit from this approach. Besides, this approach unrolls the innermost loop to increase parallelism inside a basic block, but it limits on the unroll factor which relies on the knowledge of target vector register size. For

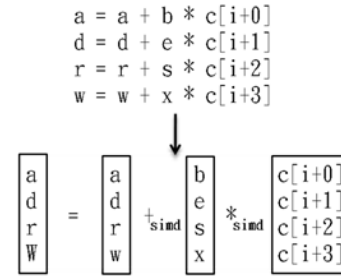


Fig. 1: Isomorphic statements.

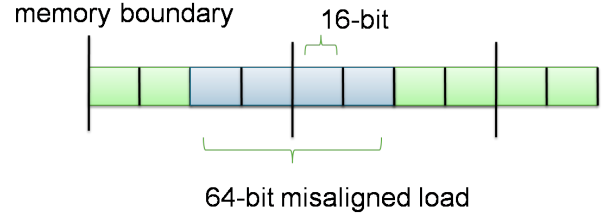


Fig. 2: Misaligned loads.

example, to calculate the unroll factor, the compiler must know the architecture’s vector register size (suppose it is 8-byte) if the array is assumed to contain 2-byte elements in the loop then the unroll factor will be 4 (8/2).

In addition, if the loop’s trip counts is not divisible by unroll factor, the loop would be split which breaks one loop into two, one is a vectorized loop and another one for the remaining iteration.

There are many related techniques that have been proposed in the past. Some techniques focus on vectorizing scalar source [3, 5–9], enable vectorization on non-vectorizable code and build cost models to select the most effective SIMD instructions or target-specific instructions. Others focus on the portable issue [10–12] because each multimedia extension is very different. The usual solution is to design vector instructions in IR. For example, Bocchino and Adve [12] designed a vector LLVA (Low Level Virtual Architecture) based on LLVM, that can transfer IR to native code by avoiding use of hardware-specific parameters and achieve code competitive with hand-coded native assembly, but the programmer should hand-tune vector LLVA code for the program carefully for each target. This method obviously can get more efficient than auto-vectorization. There has been some work focused on generating efficient code that is influenced by the alignment, vector size, and data movement constraints [13].

B. Overview of Alignment Mechanisms

The misalignment problem often occurs in SIMD operations. For example, two SIMD instructions which op-

TABLE I: Alignment Mechanism across different platforms [12, 14]

Target	Unaligned Load	Aligned Load	Realign Load	RT
AltiVec/VMX		lvx	vperm	lvsl
SSE/SEE3	movdqu, lddqu	movdqa		
MIPS-3D		luxcl	alnv.ps	address
MIPS64	ldl, ldr			
alpha		ldq_u	extql, extqh	address
VIS1		ldd	faligndata	alignaddress

1) Original:

```
void add(short *a, short *b, short *c, int n){
    for(i=0; i<n; i++){
        *a++ = *b++ + *c++;
    }
}
```

2) Multi-Versioning:

```
void add(short *a, short *b, short *c, int n){
    if(a&3||b&3||c&3){
        /* at least one of operation pointers is unaligned */
        for(i=0; i<n; i++){
            *a++ = *b++ + *c++;
        }
    } else {
        /* all operation pointers are aligned */
        for(i=0; i<n; i+=3)
            *(a:a+3) = *(b:b+3)+*(c:c+3);
    }
}
```

3) Dynamic Loop Peeling:

```
void add(short *a, short *b, short *c, int n){
    /* pre-loop to align pointer a to 8-byte boundary */
    unsigned offset = (unsigned) a & 7;
    peel = offset ? (8 - offset) / sizeof(short) : 0;
    for(i=0; i<min(n, peel); i++){
        *a++ = *b++ + *c++;
    }
    /* pointer a is aligned */
    for(; i<n; i++){
        *a++ = *b++ + *c++;
    }
}
```

Fig. 3: Software alignment mechanisms (suppose vector length is 8 byte).

erate on 16-bit data elements use one 64-bit load instruction to pack data at once. A misaligned access occurs when the loading address is not at the word boundary. Fig. 2 shows a misalignment example. For a misaligned access, the program would raise the misaligned error if the underlying hardware does not provide unaligned accesses. Although some OS'es support unaligned accesses with software techniques, misaligned accesses are very time-consuming. Therefore, the compiler should deal

with this problem by using software methods to avoid misaligned accesses or enable misaligned accesses by using hardware or software approaches.

In the hardware mechanisms, different architectures have different approaches to handle the misalignment problem. Table. I shows that the Intel SSE ISA and MIPS64 support unaligned accesses, but their latency is longer than that of aligned accesses. The form of the realignment token (RT) can be an address, a bit mask,

a permutation mask or some other value in different platforms. For example, the AltiVec *lvsr* instruction computes a permutation mask which the permutation instruction *vperm* could use to select the correct data. The *alignaddress* instruction in VIS computes a mask and return an aligned address for loading two aligned data to register. Then the *faligndata* instruction uses the mask to select correct data in two registers. Moreover, in the MIPS-3D and alpha platform, the hardware automatically clear the lower-bit of the effective address and returns an aligned address (RT) for shift, rotates, or permute to extract the unaligned data elements.

In the software mechanisms, the common approaches are *dynamic loop peeling* and *multi-versioning* [?, 8, 15]. Multi-versioning is used to avoid misaligned memory accesses by using runtime checks, because not all alignment information can be obtained in static time. Dynamic loop peeling focuses on enforcing aligned accesses. Fig. 3 illustrates those approaches.

C. The LLVM Compiler Infrastructure

The Low Level Virtual Machine (LLVM) is a compiler infrastructure which is composed by reusable modular components (passes). Each pass performs one analysis or one transformation in a certain type (function, loop, etc.). LLVM enable effective program optimizations across the entire lifetime which includes compile time, link time, and run time and provides a powerful Intermediate Representation (IR). The LLVM IR is a RISC-like instruction set but with key higher-level information for effective analysis. This includes type information, explicit control flow graphs, and an explicit dataflow representation (using an infinite, typed register set in the SSA form) [16].

The type system is the most important feature in LLVM. It is low-level, language-independent and is used to implement data types and operations from high-level languages, exposing their implementation behavior to all stages of optimization [16]. IR instructions perform type conversions and low-level address arithmetic while preserving type information [16]. The vector type is the key type used in this paper which represents a vector of elements. A companion code generator in LLVM back-end always emits SIMD instructions for the vector type when the target platform supports SIMD instructions. Note that the LLVM does not support auto-vectorization transformations currently but only provides the vector type in IR which can be generated by intrinsic functions or built-in functions in the source code.

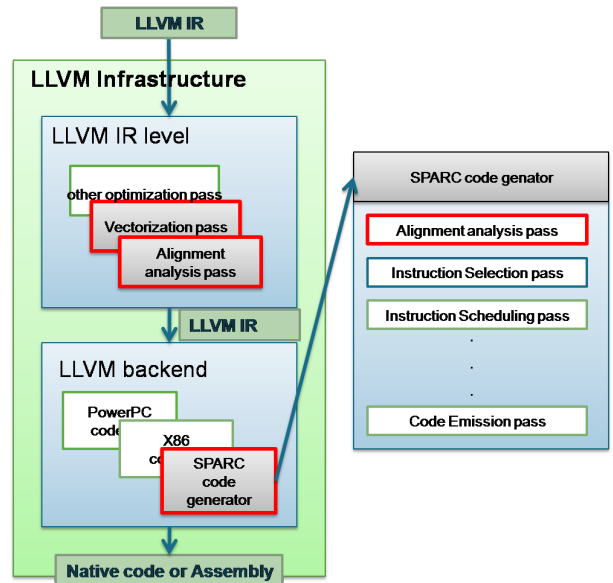


Fig. 4: Our automatic superword vectorization.

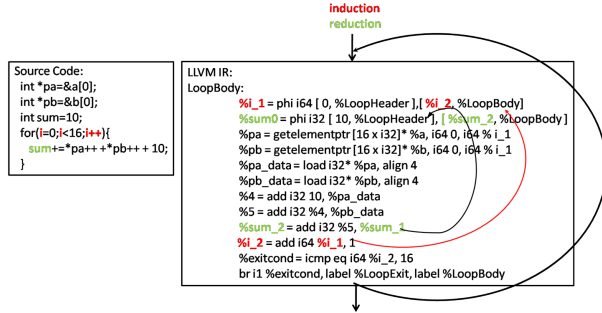
III. OVERVIEW OF AUTOMATIC SUPERWORD VECTORIZATION

This section explains our automatic superword vectorization, which is depicted in Fig. 4. In the front-end, source code is compiled to LLVM IR. LLVM supports many independent and re-useable optimization and transformation passes in the LLVM IR level. We add two passes: auto-vectorization and alignment analysis. The auto-vectorization pass aims to explore data-level parallelism in the innermost loop and converts IR instructions from primitive type to vector type. The alignment analysis is an interprocedural pointer analysis which calculates alignment information for instruction selection. The LLVM back-end provides several code generators for various platforms and supports some multimedia extensions but it does not support the VIS extension on SPARC platform. We modified an existing SPARC code generator in LLVM for generating SIMD instructions and realignment instructions with the help of the alignment analysis pass.

A. Auto-Vectorization

The auto-vectorization pass works on each loop independently. This pass aims to explore data-level parallelism and produces vector-type IR instruction from primitive types. Our basic-block auto-vectorization method is similar to Larsen and Amarasinghe’s approach (SLP) [3], and we assume overflows will not occur. The detail procedure is presented in this section.

1) *Pre-pass*: The pre-pass step performs several LLVM optimization and transformation passes. Those passes include the general SSA-form optimization, con-



(a) An example code

```

LoopBody:
  %i_1 = phi i64 [ 0, %LoopHeader ], [ %i_2.1, %LoopBody ]
  %pa = getelementptr [16 x i32]* %a, i64 0, i64 %i_1
  %pb = getelementptr [16 x i32]* %b, i64 0, i64 %i_1
  %r_gep = getelementptr i32* %r_array, i32 0
  %r_load = load i32* %r_gep
  %pa_data = load i32* %pa, align 4
  %pb_data = load i32* %pb, align 4
  %4 = add i32 10, %pa_data
  %5 = add i32 %4, %pb_data
  %sum_2 = add i32 %5, %r_load
  %i_2 = add i64 %i_1, 1
  store i32 %sum_2, i32* %r_gep
  %pa.1 = getelementptr [16 x i32]* %a, i64 0, i64 %i_2
  %pb.1 = getelementptr [16 x i32]* %b, i64 0, i64 %i_2
  %r_gep.1 = getelementptr i32* %r_array, i32 1
  %r_load.1 = load i32* %r_gep.1
  %pa_data.1 = load i32* %pa.1, align 4
  %pb_data.1 = load i32* %pb.1, align 4
  %4.1 = add i32 10, %pa_data.1
  %5.1 = add i32 %4.1, %pb_data.1
  %sum_2.1 = add i32 %5.1, %r_load.1
  %i_2.1 = add i64 %i_2, 1
  store i32 %sum_2.1, i32* %r_gep.1
  %exitcond.1 = icmp eq i64 %i_2.1, 16
  br i1 %exitcond.1, label %LoopExit, label %LoopBody

```

(b) After loop transformation by unroll factor 2

Fig. 5: A loop transformation example.

stant propagation, dead code elimination, global common subexpression elimination, loop-invariant code motion, and combining redundant instructions and simple loop transformations, etc. Those optimizations ensure the vectorization pass not to explore redundant instructions. The simple loop transformation performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective [17]. That include inserting single loop pre-header, loop exit-blocks, and guarantees that loops will have exactly one back-edge.

2) *Vectorization Analyze and Vectorization Factor Selection*: In this step, the auto-vectorization pass would be terminate if this loop does not satisfied any of the following several conditions: (a) It is the innermost loop with a known loop bound. (b) It has consecutive array data references. (c) There is no if-conversions. After the three checks, whether a loop is vectorizable has not yet been determined. Next, an unroll factor (i.e., vectorization factor) is selected. It scans all statements with data array references and determine the unroll factor which is defined as the vector register size divided by the smallest data size. For example, the unroll factor is 4 (64/16) when a loop contains 16-bit smallest data elements and the vector register size is 64-bit. Finally, if the unroll factor is equal to the loop’s trip count, we do not perform vectorization.

3) *Loop Transformations*: In this step, the goal is to transform loops including loop splitting and loop unrolling. Frequently, a loop cannot be vectorized directly because the loop’s trip count modulo the unroll factor is not zero. In this case, the loop would be split into two loops. The number of iterations in the first loop is equal to some multiple of the unroll factor. The remaining iterations are put in the second loop. After the loop is split, the first loop would be unrolled. When the loop is unrolled, the reduction operand is also be explored. The reduction operand means that the scalar variable will be used as the invariant operand by the next iteration operation, such as summation in the loop. In this case, the reduction operands have anti-dependency after unrolling. To vectorize a reduction operation, an auxiliary array whose size is equal to the unroll factor is added to replace the reduction operand. Lastly, the result can be computed in the loop exit blocks.

Fig. 5 shows an example of loop transformations. In Fig. 5 (a), the sum is reduction variable which has cross iteration dependences with itself, and after transformations in Fig. 5 (b), the loop is unrolled with factor 2, and the reduction operand has been replaced by r_array array. Note that the *getelementptr* instruction is used to get the address of a subelement of an aggregate data structure. It performs address calculation only and does not access memory.

4) *Vectorizable Statement Classification*: This step gathers all statements which have vectorizable operands in the block, such as Add/Sub/Mul/Shift (without induction variables), which is called *candidate statement*. Next we build the data-dependency graph without the *phi* instruction for the candidate statement, which is called *candidate set*. In the data-dependency graph, a node represents an IR instruction, and an edge denotes a dependence between instructions. The dependency includes use-def and def-use relations between two nodes. Note that, after unrolling, the loop dependency can be explored.

For example;

```

for ( i = 0; i < k; i ++ )
  a [ i + 1 ] = a [ i ] + b [ i ];

```

after the loop unrolling would be:

```

for ( i = 0; i < k; i += 2 ) {
  a [ i + 1 ] = a [ i ] + b [ i ];           (1)
  a [ i + 2 ] = a [ i + 1 ] + b [ i + 1 ]; (2)
}

```

Obviously, the variable $a[i+1]$ induces a flow dependence from statement (1) to statement (2), so they cannot be vectorized. In other words, if the dependency graph

two assumptions. First, the compiler must align data. It means that when a program allocates memory, the first element should be aligned. Second, the address is assumed to be misaligned when the alignment information cannot be obtained. The alignment information of memory access is given by:

$$AlignmentInfo = P \bmod N$$

The *AlignmentInfo* is a set of alignment value module N , the P denotes the pointer address, and the N denotes memory boundary in byte (usually, the N is equals to target vector register size). Because alignment information would be changed when the pointer performs arithmetic operation, such as $*(p+i)$. In order to evaluate pointer arithmetic, the transfer function $F : M \times M \mapsto M$ is given by:

$$F(A1 \pm A2) = [(A1 \bmod N) \pm (A2 \bmod N)] \bmod N$$

$$F(A1 \cdot A2) = [(A1 \bmod N) \cdot (A2 \bmod N)] \bmod N$$

M is the set of all possible alignment values modulo N , and $A1$ and $A2$ denote a scalar or a value of *AlignmentInfo* set.

Alignment analysis traverses the IR and propagates the alignment information. It starts from the main function. Each function call is visited sequentially. When a procedure involves a function call, *alignmentinfo* of argument variable would be passed across the function call. Otherwise, *alignmentinfo* would be merged in the different times when a function is involved. In the loop block, the alignment information propagates in each iterator. Consider the following example: the alignment information of b (the vector length is 8) is calculated. Assume the input argument b 's alignment information set is 1. In the first iteration, the alignment information of pointer b is 1, and then the b 's alignment information is 3 ($(1 \bmod 8 + 2 \bmod 8) \bmod 8 = 3$) in the statement $*b+=2$. Next, the alignment information of pointer b is 1,3 by propagation ($1 \cup 3$). Finally, the analysis would stop when the old set equal the new set, the b 's alignment information set is 1,3,5,7.

```
void sum(int *a ,int *b,int *c){
    for (i=0;i<N;i+=2){
        a[i:i+1]= b[i:i+1] + c[i:i+1];
    }
}
main(){
    int A[N],B[N+1],C[N+2];
    sum(&A[0],&B[1],&C[2]);
}
```

In a recursive call, the alignment analysis terminates when no alignment information changes during a call. To use the alignment information pass for back-end code generation, the target must define the target-specific vector register size as N and execute this pass to produce the *alignmentinfo* for each *load/load* instructions.

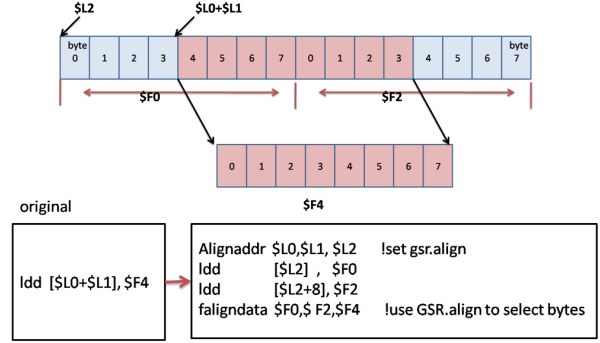


Fig. 9: ALIGNADDR instruction.

When all values in the *alignmentinfo* set are divisible by $N/\text{data element size}$, the address of load/store is aligned. For example, we assume the *alignmentinfo*= 2, 4, 6, the vector register size is 64-bit and the loading data type is 16-bit. Therefore, the load instruction is misaligned because 2, 6 are not divisible by 4 ($64/16$).

IV. EXPERIMENTAL RESULTS

We have implemented the auto-vectorization and alignment analysis passes, and modified the SPARC back-end in the LLVM 2.5. UltraSPARC is a big-endian Load/Store RISC architecture. It provides a multimedia extension, the VIS instruction set [18], which is incorporated into UltraSPARC processor's Floating Point Unit (FPU). The SIMD instruction uses 64-bits floating point register as vector register, and only support 16, 32-bit integer SIMD arithmetic operation. In the code generator, we use parallel arithmetic instructions (i.e., addition, subtraction, multiplication), and two instructions (*ALIGNADDR* and *FALIGNDATA*) for realignment. In addition, because VIS uses three instructions to emulate two 16-bits partitioned multiply (the result is two 32-bit), so we need to use nine instructions to perform four 16-bits partitioned multiply. The realignment instructions can be performed as misalignment load. The *ALIGNADDR* instruction does three things: first, it adds two integer values, and then stores the result with the least significant 3 bits forced to be 0 in the integer register. Third, the least significant 3 bits of the result are stored in a special register *GSR.align*. Therefore, the *FALIGNDATA* instruction can select the correct byte to load by the *GSR.align* register. The concepts is shown in Fig. 9.

We use the DSPstone benchmark [19] and several multimedia kernels which are used in Larsen's thesis [3]. These benchmarks are executed on Sun SPARC Enterprise T5120 Server with SunOS 5.10. Moreover, we replace multiply operation to add operation in matrix1 and matrix2 program for evaluate performance impact of nine instructions for multiply.

In all benchmark programs, we use 1024 data elements since the performance decreases when the number of data elements is smaller than 512 in the SPARC platform. In addition, we modify data type from integer(32-bit) to short(16-bit), so the vectorization factor is 4. All benchmark programs are compiled with four results. -O0 option, pre-pass

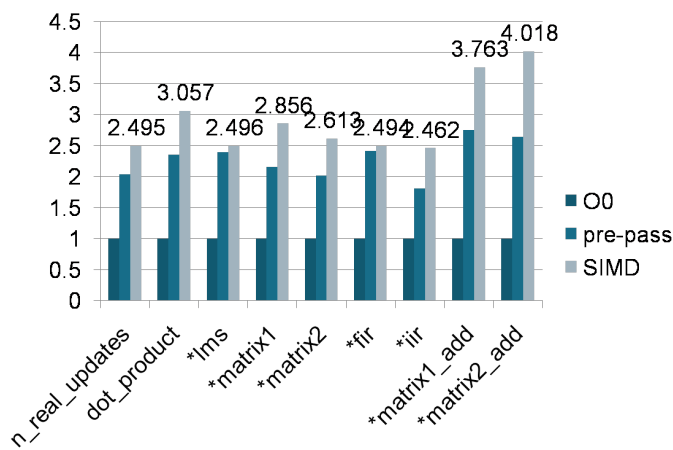


Fig. 10: Pre-pass and Auto-vectorization speedup evaluation.

optimization, auto-vectorization optimization, and disabling alignment analysis result. Note that the disabling alignment analysis always uses realignment instruction to do aligned or misaligned vector access. In the experience, time is measured using the gethrtime() call. The time measured is high resolution in nanoseconds. Speed up rates when compared with -O0 optimization for the test suites are shown in Fig. 10. The results of pre-pass and auto-vectorization are shown in Fig. 11. Fig. 12 shows the comparing result of enabling and disabling alignment analysis that means misaligned instruction would be generated in every vector memory access. In each figure, the start symbol represents the misaligned access in the program.

In certain cases, the speedup rate is lower because the misaligned access (e.g., lms, fir), or multiply always appears in the program (e.g., fir, iir) or the smaller fraction of the benchmark code can be mapped to SIMD instructions (e.g., lms). Otherwise, by using scalar expansion, the auxiliary array would allocate extra memory and use more instructions to access. These would add more penalty. However, the evaluation result shows a speedup of 4% up to 36% and the average performance improvement is up to 17.2% by comparing with pre-pass. If we comparing matrix and matrix_add, the average performance impact is 14.5%, but it still improves 30.35% by comparing with pre-pass.

V. CONCLUSION AND FUTURE WORK

In this paper we design and implement an automatic superword vectorization in LLVM. It can produces SIMD instructions to get performance improved. Our design includes auto-vectorization and alignment analysis passes. Auto-vectorization exploits data parallelism and enables base vectorization. In addition, the alignment analysis pass is used for the back-end to select target-specific realignment instructions to handle the misaligned problem in our experiment environment. Furthermore, the vectorization is effective because LLVM supports high-level information in its IR. Besides, our vectorization is independent of other optimization and analysis passes in LLVM. Programs could also apply other powerful optimizations after the auto-vectorization. In addition, the

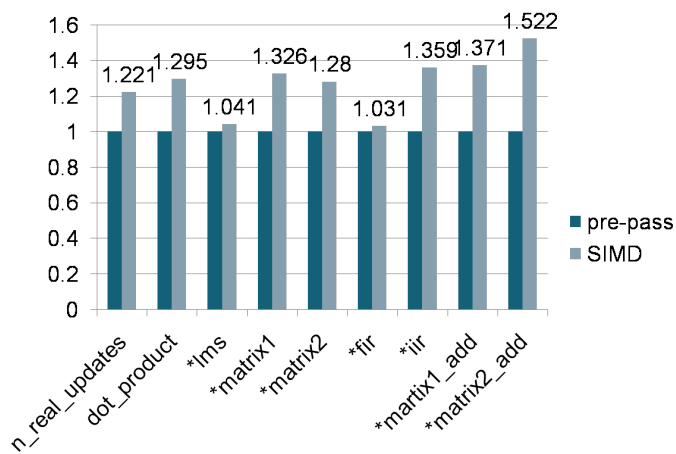


Fig. 11: Auto-vectorization speedup compared with Pre-optimization evaluation.

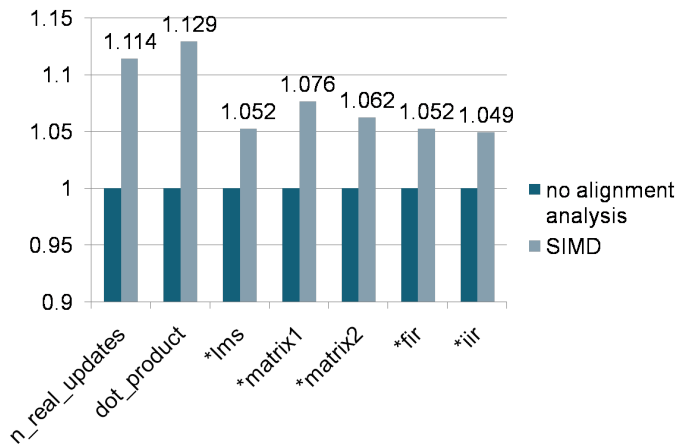


Fig. 12: Auto-vectorization speedup compare with disabling alignment analysis.

vectorized code can generate SIMD instructions for other platforms supported by LLVM.

Future work would focus on enhance vectorization capability and transfer LLVM IR to to vector LLVA [11] which provide rich set of vector instructions based on LLVM IR. (e.g., Vector LLVA provides saturation arithmetic) In addition, because auto-vectorization may introduce some extra codes and costs, the suitable cost model would be added for SPARC on LLVM.

REFERENCES

- [1] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*, San Francisco, CA, USA, 2001.
- [2] A. Shahbahrani and B. Juurlink, "Performance improvement of multimedia kernels by alleviating overhead instructions on SIMD devices," in *APPT '09: Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies*. Berlin, Heidelberg: Springer-Verlag, August 24-25 2009, pp. 389-407.

- [3] L. Samuel and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, June 18-21 2000, pp. 145–156.
- [4] M. J. Wolfe, *High performance compilers for parallel computing*, C. Shanklin and L. Ortega, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [5] D. Naishlos, "Autovectorization in GCC," in *the GCC Developers Summit*, June 2-4 2004, pp. 105–117.
- [6] I. Pryanishnikov, A. Krall, and N. Horspool, "Compiler optimizations for processors with SIMD instructions," in *Software Practice and Experience*, vol. 37, no. 1. New York, NY, USA: John Wiley & Sons, Inc., January 2007, pp. 93–113.
- [7] S. Larsen, R. Rabbah, and S. Amarasinghe, "Exploiting vector parallelism in software pipelined loops," in *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, November 12-16 2005, pp. 119–129.
- [8] A. Krall and S. Lelait, "Compilation techniques for multimedia processors," in *International Journal of Parallel Programming*, vol. 28, no. 4. Norwell, MA, USA: Kluwer Academic Publishers, August 2000, pp. 347–361.
- [9] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao, "An integrated simdization framework using virtual vectors," in *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*. New York, NY, USA: ACM, June 20-22 2005, pp. 169–178.
- [10] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr, "A SIMD optimization framework for retargetable compilers," in *ACM Transactions on Architecture and Code Optimization*, vol. 6, no. 1. New York, NY, USA: ACM, March 2009, pp. 1–27.
- [11] R. L. Bocchino Jr. and V. S. Adve, "Vector LLVA: a virtual vector instruction set for media processing," in *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, June 14-16 2006, pp. 46–56.
- [12] D. Nuzman and R. Henderson, "Multi-platform auto-vectorization," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, March 26-29 2006, pp. 281–294.
- [13] A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD architectures with alignment constraints," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, June 09-11 2004, pp. 82–93.
- [14] Sun Microsystems, Inc., *UltraSPARC T2 supplement to the UltraSPARC architecture 2007*, 2007.
- [15] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, "Automatic intra-register vectorization for the Intel® architecture," in *International Journal of Parallel Programming*, vol. 30, no. 2. Norwell, MA, USA: Kluwer Academic Publishers, April 2002, pp. 65–98.
- [16] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization*. Washington, DC, USA: IEEE Computer Society, March 20-24 2004, pp. 75–86.
- [17] C. Lattner, *The LLVM Compiler Infrastructure Project*. [Online]. Available: <http://llvm.org/>
- [18] Sun Microsystems, Inc., *VIS instruction set user's manual*, 2002.
- [19] V. Zivojnovic, J. M. Velarde, C. Schlager, and H. Meyr, "DSPSTONE: A DSP-oriented benchmarking methodology," in *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*. Dallas, Tex, USA: Miller Freeman, October 1994, pp. 715–720.