Refactoring Loops with Nested IFs for SIMD Extensions Without Masked Instructions

Huihui Sun[1,2](✉), Sergei Gorlatch[1], and Rongcai Zhao[2]

[1] University of Münster, Münster, Germany
{huihuisun,gorlatch}@uni-muenster.de
[2] National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China

Abstract. Most CPUs in heterogeneous systems are now equipped with SIMD (Single Instruction Multiple Data) extensions that operate on short vectors in parallel to enable high performance. Refactoring programs for such systems relies on vectorization, i.e., transforming into a form with SIMD-instructions. We improve the state of the art in refactoring loops with nested IF-statements that are notoriously difficult to vectorize. For IF-statements whose conditions are independent of the loop variable, we improve the classical *loop unswitching* method, such that it can tackle nested IFs. For IF-statements whose conditions change with loop iterations, we develop a novel *IF-select transformation* method: (1) it can work with arbitrarily nested IFs, and (2) while previous methods rely on either masked instructions or hardware support for predicated execution, our method works for SIMD extensions without such operations (as found, e.g., in IBM Power8 and ARM Cortex-A8). Our experimental evaluation for the SPEC CPU2006 benchmark suite is conducted on an SW26010 processor used in the Sunway TaihuLight supercomputer (#2 in the TOP500 list); it demonstrates the performance advantages of our implemented approach over the vectorizer of the Open64 compiler.

Keywords: SIMD extensions · Nested IF-statements
Loop vectorization · Loop unswitching · IF-select transformation

1 Motivation and Related Work

Most modern processors are equipped with SIMD (Single Instruction Multiple Data) extensions that operate on short vectors in parallel to enable high performance. To use this performance potential, programs must be refactored to a form with SIMD instructions; this is traditionally called *vectorization*. Manual vectorization via hand-written instrinsics is tedious, error-prone and unportable. Therefore, automatic vectorization is an indispensable part of most modern compilers, such as the commercial compiler ICC [11], as well as open-source compilers Open64 [3], GCC [6], and LLVM [14].

G. Mencagli et al. (Eds.): Euro-Par 2018 Workshops, LNCS 11339, pp. 769–781, 2019.
https://doi.org/10.1007/978-3-030-10549-5_60

There are three classic vectorization approaches: (1) loop vectorization [18] combines multiple occurrences of a scalar operation across consecutive loop iterations into one SIMD instruction, (2) basic block or SLP (Superword Level Parallelism) vectorization [13] transforms a group of isomorphic operations into one SIMD instruction, and (3) WFV (Whole Function Vectorization) [12] converts multiple instances of a kernel into SIMD instructions. These approaches are restricted: in particular, IF-statements lead to the control flow divergence that makes vectorization difficult.

Several methods were suggested to overcome this restriction, but they work only in special cases. The *loop unswitching* method [19] requires that the IF-condition remains the same across loop iterations. The *IF conversion* method [2] targets vector computers with explicit hardware support for predicated execution, where instructions from both paths of the branch are executed speculatively, and each instruction is then associated with a dedicated predicated register that determines whether this instruction should modify processor state. In this paper, we develop vectorization methods for processors with SIMD extensions that do not have explicit hardware support for predicated execution. Shin et al. [16] extend the classic SLP method to work in the presence of IF-statements. Our approach is similar to [16], except that we extend loop vectorization to work in the presence of arbitrarily nested IF-statements; we discuss further differences below. In comparison with the WFV method [12], we vectorize loops rather than functions in data-parallel languages (like CUDA or OpenCL). The state-of-the-art compilers such as LLVM depend on masked instructions to vectorize IF-statements, and need to fall back on IF-cascades on architectures without masked instructions, which makes automatic vectorization futile on such architectures. In our recent work [8], we extend the WFV vectorizer for SIMD extensions without masked instructions. Also Smith et al. [17] describe using masked vector instructions for vectorization, while we target architectures without masked instructions.

Summarizing, we aim at improving the state-of-the-art methods of refactoring by vectorization, that currently cannot generate efficient SIMD code for loops with arbitrarily nested IF-statements without hardware support for predicated execution or masked instructions. We cover two cases depending on whether the IF-condition changes across the loop iterations: (1) for loop-independent IF-statements, we extend the *loop unswitching* method to arbitrarily nested IF-statements; (2) for loop-dependent IF-statements, we develop a novel *IF-select transformation* method which works for loops with arbitrarily nested IF-statements on SIMD extensions without hardware support for predicated execution and masked instructions. We integrate our approach into the Open64 compiler [3] and evaluate it on an SW26010 processor [7] with a 256-bit SIMD extension as used in the Sunway TaihuLight supercomputer (#2 in the TOP500 list [20]). Experiments on a set of benchmarks from SPEC CPU2006 [9] with loops containing IF-statements confirm the efficiency of our approach.

In the remainder of the paper, Sect. 2 introduces the background on refactoring via vectorization and our target architecture model. Section 3 presents our vectorization approach for loops with nested IF-statements. Experimental results are presented in Sect. 4, and Sect. 5 concludes the paper.

2 Background: SIMD Extensions and Vectorization

We target modern heterogeneous systems that comprise CPUs with SIMD extensions, but without masked instructions, such as IBM Power8 [10], ARM Cortex-A8 [4], and SW26010 [7]. While the existing frameworks like FastFlow [1] and REPARA [5] can distribute workload among different cores using manual refactoring based on parallel patterns, we aim at automated refactoring within one core using vectorization. We use the SW26010 processor as our example: each core of it employs a 256-bit SIMD extension that works on 256 bits in parallel: it can be one long int (256-bit) operation, or 8 integer operations, or 4 floating point operations. Without loss of generality, we work in this paper with 64-bit floating point values, i.e., 4 operations can be executed simultaneously on such values.

Figure 1 illustrates a simple example of refactoring via vectorization: Fig. 1(a) shows a loop with regular computations, so it is straightforwardly vectorizable. Figure 1(b) shows the vectorization result using SIMD intrinsics, i.e., C-style functions providing access to SIMD instructions. For simplicity, we call these intrinsics *instructions*. A SIMD extension executes a loop iteration in Fig. 1(b) in parallel as follows: load the operands from memory to vectors, add the two vectors, and store the result vector into memory.

```
for(i=0;i<1024;i++)
{
  c[i]=a[i]+b[i];
}
```

(a)

```
for(i=0;i<1024;i=i+4)
{
  simd_load(v_a,&a[i]);
  simd_load(v_b,&b[i]);
  v_c=simd_vaddd(v_a,v_b);
  simd_store(v_c, &c[i]);
}
```

(b)

Fig. 1. (a) An easily vectorizable loop; (b) The loop after vectorization

Table 1 shows the SIMD instructions used in this paper, with the names as used in the SW26010 processor. We only list the instructions for double precision floating point parameters; the vector type `doublev4` means 4 packed 64-bit double elements.

An important feature of our target architecture model is that we do not assume the existence of dedicated predicate registers, while many previous approaches to vectorization (e.g., [2]) rely on these registers and the corresponding predicated execution modus. Such registers can be found, e.g., in conventional vector processors, but not in modern CPUs with SIMD extensions. We also do not require from our target SIMD extensions to provide masked instructions

Table 1. Specific SIMD instructions used in this paper

Instruction	Operation	Input	Output	Functional description
simd_load	Load	doublev4 va, double *addr	void	Load 4 double elements into vector va from contiguous memory starting from *addr
simd_store	Store	doublev4 va, double *addr	void	Store 4 elements of vector va into contiguous memory starting from *addr
simd_vaddd	Addition	doublev4 va, vb	doublev4	Add 4 elements of va with 4 elements of vb element-wise, return the result
simd_vsubd	Subtraction	doublev4 va, vb	doublev4	Subtract 4 elements of va from 4 elements of vb element-wise, return the result
simd_vseleq	Select	doublev4 va, vb, vc	doublev4	Test the value of va element-wise: if it equals 0, then return the element of vb, otherwise return the element of vc
simd_vfcmplt	Comparison	doublev4 va, vb	doublev4	Compare the value of va and vb element-wise; if va < vb, then the element of vc is assigned 1.0, otherwise 0

that are present, e.g., in the Intel AVX extension and used in some vectorization methods [17]. Summarizing, we aim at covering a broader class of target architectures than most of previous approaches.

3 Vectorization of Loops with IF-statements

Figure 2 shows the overall structure of our vectorization approach. For clarity, we assume that there is only a single, probably nested, IF-statement in the loop. For multiple IF-statements, we process them ordinally.

The first step in Fig. 2, *SIMD preanalysis*, checks whether vectorization can be applied to the loop legally. We mainly rely on the traditional four criteria of legal vectorization: (1) there are no dependence cycles between the statements in the loop body; (2) the loop is countable [15], i.e., the number of iterations of the loop is known before entering the loop body; (3) there is only one exit from the loop; (4) the loop is the innermost loop.
Note that the IF-statement may be nested, such that either the THEN or ELSE block or both have at least one IF-statement. Each IF-statement in a candidate loop is put into one of two categories:

- *a loop-independent IF-statement*, if its condition remains the same across loop iterations;
- *a loop-dependent IF-statement*, if its condition changes with loop iterations.

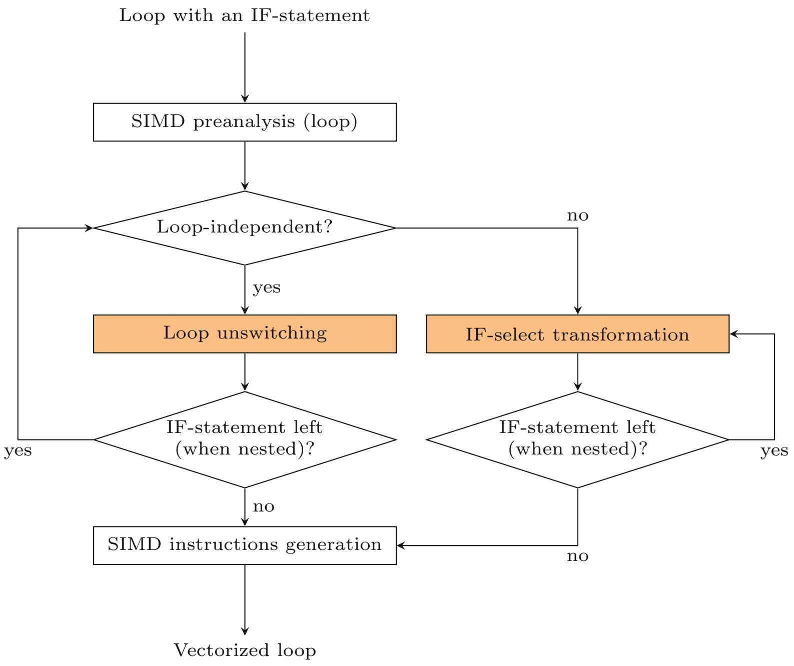
Loop with an IF-statement
SIMD preanalysis (loop)
Loop-independent?
no
yes
Loop unswitching
IF-select transformation
IF-statement left (when nested)?
yes
IF-statement left (when nested)?
yes
no
SIMD instructions generation
no
Vectorized loop

Fig. 2. Overview of our vectorization approach for a loop with a single, possibly nested IF-statement

According to these two cases, we apply two vectorization methods in Fig. 2:

– If the IF-statement is loop-independent, we apply our improved *loop unswitching* method (described in Sect. 3.1) to move the condition testing of the IF-statement outside of the loop. If the IF-statement is nested, we first tackle the outermost IF-statement, then tackle the inner IF-statement in the THEN or ELSE block and so on, until there is no IF-statement left, or until we encounter a loop-dependent IF-statement.
– If the IF-statement is loop-dependent, we apply our novel *IF-select transformation* (described in Sect. 3.2) that converts control dependences (IF) into data dependences (select). If the IF-statement is nested, we first tackle the innermost IF-statement in the corresponding THEN or ELSE block, then tackle the outer IF-statement and so on, until there is no IF-statement left.

The output of our method is the loop without IF-statements, for which equivalent SIMD instructions can be generated straightforwardly like in Fig. 1. The following two subsections describe the two core methods (the highlighted parts in Fig. 2) in detail.

3.1 Vectorizing Loop-Independent IFs: Loop Unswitching

The *loop unswitching* method, originally proposed in [19], is applied to a loop with a loop-independent IF-statement: the idea is to move the condition testing of the IF-statement outside of the loop; the original loop is duplicated, and a copy of it is placed inside of both THEN and ELSE blocks of the resulting IF-statement. Note that, besides enabling vectorization, loop unswitching also optimizes the program, because the testing of the IF-condition is performed only once outside of the loop, rather than repetitively in each loop iteration.

Our modification of the original loop unswitching method [19] allows it to be applied to arbitrarily nested IF-statements within a loop if all of them are loop-independent: we first apply loop unswitching to the outermost IF-statement, and then we apply loop unswitching iteratively to the both copies of the loop with respect to their outermost IF-statements, and so on, see Fig. 2. However, repetitive loop unswitching may lead to an exponential increase of code size, thus hindering the compiler to do other optimizations. We empirically impose a limit of 4 passes of this transformation for nested IF-statements, which is found to be a good solution via experimental evaluation.

3.2 Vectorizing Loop-Dependent IFs: IF-select Transformation

As described above, the classical loop unswitching method is only applicable to loop-independent IF-statements. For a loop-dependent IF-statement, we follow the idea of [16] to transform the IF-statement into *select* statements. However, our approach proceeds very differently from [16], where the original IF conversion [2] is applied to transform a program with IF-statements into an equivalent program with predicated statements, which are then transformed into select statements. This transformation relies on the PHG (Predicate Hierarchy Graph) representing the nesting relations among predicates. Our approach generates select statements directly, without generating predicated statements: we also avoid building and analyzing the PHG.

The idea of our approach is that we generate select statements by matching the statements in the THEN block with the statements in the ELSE block and combining each pair of matched statements into a select statement. We say that statements are matched if they define the same variable. For example, in the statement `if(cond){dst=val1;} else{dst=val2;}`, the statements in the THEN and ELSE blocks both define `dst`, so they are matched, and we can combine them into one select statement `dst=select(cond,val1,val2)`. In contrast, if there are no matched statements for the current statement in the THEN or the ELSE block, then we assume that there is a fictitious statement `dst=dst` to match with the current statement, and we combine the original statement with the fictitious statement into one select statement. For example, in the statement `if(cond){dst=val1;}`, there is no ELSE part and thus no matched statement, therefore, for this single statement we generate the select statement `dst=select(cond,val1,dst)`. We denote the former case that generates a select statement for two matched statements as *Rule 1*, and the latter case that generates a select statement for a single unmatched statement as *Rule 2*.

Algorithm 1 shows the pseudocode of our *IF-select transformation* method applied to an IF-statement in a loop. We first create a new block `sel_wn` to store the newly generated select statements (line 2). Then we sequentially traverse the statements in the THEN and ELSE blocks (line 6): we initialize the flag `matched` as FALSE (line 7) at the beginning of each traversal pass, and then we try to match the statements in the THEN and ELSE blocks and generate corresponding select statements according to Rule 1 and Rule 2 (line 8–44). Eventually, if `sel_wn` is not empty (line 45), we replace the original IF-statement with `sel_wn` (line 46), otherwise, we leave the IF-statement unchanged.

We describe in the following how we match statements and generate select statements, especially when there are flow dependences in the block. We begin with traversing the ELSE block from the current statement and looking for a matching statement (line 10–14) for the current statement in the THEN block. If there is no matching statement (Case 1), then we generate a select statement according to Rule 2 (line 16) and we turn to the next statement in the THEN block (line 17). If we find a matching statement that is the current statement in the ELSE block (Case 2), then we combine these two statements and generate a select statement according to Rule 1 (line 20), and we turn to the next statements in the THEN and ELSE blocks (line 21–22).

Otherwise, if the matching statement is not the current statement in the ELSE block (Case 3), then we reset flag `matched` to FALSE (line 24), and then we turn to looking for a matching statement in the THEN block (line 26–30) for the current statement in the ELSE block. If there is no matching statement in the THEN block (line 31), then we generate a select statement according to Rule 2 (line 32), and we turn to the next statement in the ELSE block (line 33). If a matching statement for the current statement is found, then it means that the order of these two statements is different in the THEN and ELSE blocks: e.g., `dst1` is defined before `dst2` in the THEN block and after `dst2` in the ELSE block. In this case, we check whether there is a flow dependence between the memory accesses in these two statements (line 35). If no flow dependence is found from `then_stmt` to `then_iter`, then we change the order of these two statements in the THEN block by moving `then_stmt` after `then_iter`. Likewise, if no flow dependence is found from `else_stmt` to `else_iter`, then we change the order of these two statements in the ELSE block by moving `else_stmt` after `else_iter`. Otherwise, we retain the IF-statement unchanged, ignore all select statements generated before, and return (line 41). After detecting flow dependences and reordering statements, we generate select statements according to Rule 1 (line 36) and turn to the next statements in the THEN and ELSE blocks (line 37–38). Note that case 3 enables us to generate select statements even when there is a flow dependence between the statements in the THEN or ELSE block. If we would simply add all matched statements to `sel_wn` and perform an analysis for detecting a cyclic dependence afterward, we may end up with inconsistent semantics by ignoring flow dependences.

Algorithm 1. IF-select Transformation

```
 1 Function IF-selectTransformation(IF)
 2     build a new block sel_wn ;                 // store the generated select statements
 3     get the Array_Dependence_Graph as ADG;
 4     then_stmt=get_first(IF.then);    // initiate the current statement in the THEN block
 5     else_stmt=get_first(IF.else);    // initiate the current statement in the ELSE block
 6     while then_stmt!=NULL | else_stmt!=NULL do
 7         BOOL matched = FALSE;
 8         if then_stmt != NULL then
 9             else_iter=else_stmt;
10             while else_iter!=NULL & matched ==FALSE do
11                 if else_iter is matched with then_stmt then
12                     matched = TRUE ;                     // find the matched else_iter
13                 else
14                     else_iter=get_next(else_iter);
15             if matched==FALSE then                                          // Case 1
16                 generate select statement (Rule 2) and insert it into sel_wn;
17                 then_stmt=get_next(then_stmt);
18             else
19                 if else_iter == else_stmt then                              // Case 2
20                     generate select statements (Rule 1) and insert it into sel_wn;
21                     then_stmt=get_next(then_stmt);
22                     else_stmt=get_next(else_stmt);
23                 else                                                        // Case 3
24                     matched = FALSE;
25                     then_iter=then_stmt;
26                     while then_iter!=NULL & matched ==FALSE do
27                         if then_iter is matched with else_stmt then
28                             matched = TRUE ;             // find the matched then_iter
29                         else
30                             then_iter=get_next(then_iter);
31                     if matched==FALSE then
32                         generate select statement (Rule 2) and insert it into sel_wn;
33                         else_stmt=get_next(else_stmt);
34                     else
35                         if Forward_Motion(then_stmt, then_iter, ADG) |
                             Forward_Motion(else_stmt, else_iter, ADG) then
36                             generate select statements (Rule 1) and insert it into sel_wn;
37                             then_stmt=get_next(then_stmt);
38                             else_stmt=get_next(else_stmt);
39                         else
40                             sel_wn = NULL;
41                             return;
42         else if else_stmt != NULL then                                      // Case 4
43             generate select statement (Rule 2) and insert it into sel_wn;
44             else_stmt=get_next(else_stmt);
45     if sel_wn != NULL then
46         replace IF with sel_wn;
```

If we are done with all statements in the THEN block and there are still statements in the ELSE block (Case 4), then for the current statement in the ELSE block we generate a select statement according to Rule 2 (line 43), and we turn to the next statement in the ELSE block (line 44), until we are also done with all statements in the ELSE block.

We further extend our IF-select transformation method (Algorithm 1) to handle nested loop-dependent IF-statements: we tackle the IF-statements starting from the innermost one and moving to the outermost, see Fig. 2.

```
for(i=0;i<1024;i++)
{
 if(a[i]<b[i])
 {
  if(a[i]<10)
    c[i]=a[i]+b[i];
  else
    c[i]=a[i]-b[i];
  }
}
```

(a)

```
for(i=0;i<1024;i++)
{
 if(a[i]<b[i])
 {
   c[i]=select(a[i]<10,a[i]+
        ↪ b[i],a[i]-b[i]);
  }
}
```

(b)

```
for(i=0;i<1024;i++)
{
 c[i]=select(a[i]<b[i],select(a[i]<10,
      ↪ a[i]+b[i],a[i]-b[i]),c[i]);
}
```

(c)

```
for(i=0;i<1024;i=i+4)
{
 simd_load(v_a,&a[i]);
 simd_load(v_b,&b[i]);
 v_add=simd_vaddd(v_a,v_b);
 v_sub=simd_vsubd(v_a,v_b);
 v_cond1=simd_vfcmplt(v_a,v_10);
 v1=simd_vseleq(v_cond1,v_sub,v_add);
 simd_load(v_c,&c[i]);
 v_cond2=simd_vfcmplt(v_a,v_b);
 v2=simd_vseleq(v_cond2,v_c,v1);
 simd_store(v2,&c[i]);
}
```

(d)

Fig. 3. (a) A loop with a nested loop-dependent IF-statement; (b) Apply IF-select transformation to the innermost IF-statement; (c) Apply IF-select transformation to the outermost IF-statement; (d) Vectorized code with SIMD instructions

Figure 3(a) illustrates how we vectorize a nested loop-dependent IF-statement. According to Algorithm 1, we first transform the innermost IF-statement to a select statement (Rule 1), with the result in Fig. 3(b). Then we transform the outermost IF-statement to a select statement (Rule 2), with the result in Fig. 3(c). Finally, we generate SIMD instructions as shown in Fig. 3(d).

4 Experimental Evaluation and Results

We integrated our presented vectorization approach for loops with nested IF-statements into the Open64 compiler [3] by adding to it our improved methods of loop unswitching (Sect. 3.1) and IF-select transformation (Sect. 3.2). The SIMD preanalysis and the generation of SIMD instructions shown in Fig. 2 have been slightly adapted in order to exploit our proposed vectorization methods.

Table 2. Benchmark kernels with IF-statements from SPEC CPU2006

Program	Kernel	Kernel runtime (%)	Application category	IF-stmt type
429.mcf	primal_bea_mpp	49.95	Combinatorial optimization	Nested
456.hmmer	P7Viterbi	99.53	Search gene sequence database	Nested
464.h264ref	SetupFastFullPelSearch	40.93	Video compression	Nested
454.calculix	e_c3d	69.12	Structural mechanics	Nested
482.sphinx3	vector_gautbl_eval_logs3	38.67	Speech recognition	Single
458.sjeng	std_eval	15.11	Pattern recognition	Nested
462.libquantum	quantum_toffoli	63.41	Physics and quantum computing	Nested

We conduct our experiments on the programs with IF-statements from the SPEC CPU2006 benchmark suite [9], listed in Table 2. Out of 29 programs in SPEC CPU2006, the 7 programs in the table contain IF-statements in their most time-consuming loops; 6 of these programs have nested IF-statements within loops. Our experimental platform is an SW26010 processor with a 256-bit dedicated SIMD extension, running under Linux Redhat Enterprise 5.

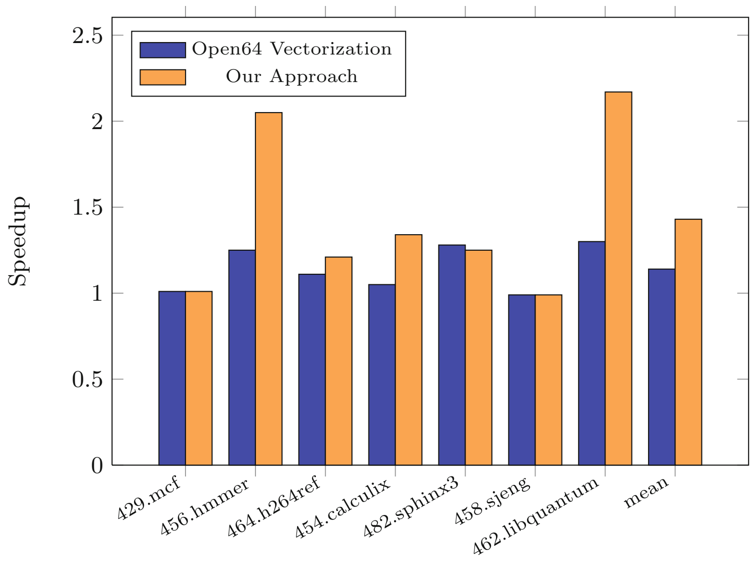

Fig. 4. Kernel speedups: our approach compared with the Open64 vectorization

For the seven benchmarks listed in Table 2, both kernel and whole-program speedups are presented. We compare two vectorization approaches: the Open64 compiler vectorization (performing loop unswitching and IF conversion) and our approach. All programs are compiled with the same flags: -O3, -LNO:simd=1. The execution time of a kernel or program is measured as the average of 20 runs. The results are within a few percent over each run. The speedups are calculated as compared with the execution on the same SW26010 processor, but without vectorization, i.e., when the SIMD extension is not used.

Figure 4 shows the kernel speedups. The mean kernel speedup achieved by our approach is 1.43x compared to the non-vectorization baseline and 1.25x compared to the Open64 vectorization. Our approach outperforms Open64 vectorization for 4 out of 7 programs and matches it for 3 remaining programs. We attribute the performance gains as follows. For 454.calculix, our loop unswitching method is applied twice to the two-level nested loop-independent IF-statement. For 456.hmmer, there is a two-level nested IF-statement: firstly, our loop unswitching method is applied to the outermost loop-independent IF-statement, and then our IF-select transformation is applied to the innermost loop-dependent

IF-statement, the same is done for 464.h264ref. For 462.libquantum, our IF-select transformation is applied to the two-level nested loop-dependent IF-statement. Our approach achieves a speedup similar to the Open64 vectorizer for 482.sphinx3, because its IF-statement is not nested. The remaining 2 programs which show no improvement are 429.mcf and 458.sjeng: they are not vectorized. For 429.mcf, its IF-statement contains pointers where dependence cycles are conservatively assumed and, therefore, the surrounding loop is excluded from vectorization. For 458.sjeng, there is a three-level nested loop-dependent IF-statement, however, the dependence cycles between the indirected arrays exclude the loop from vectorization in the SIMD preanalysis phase.

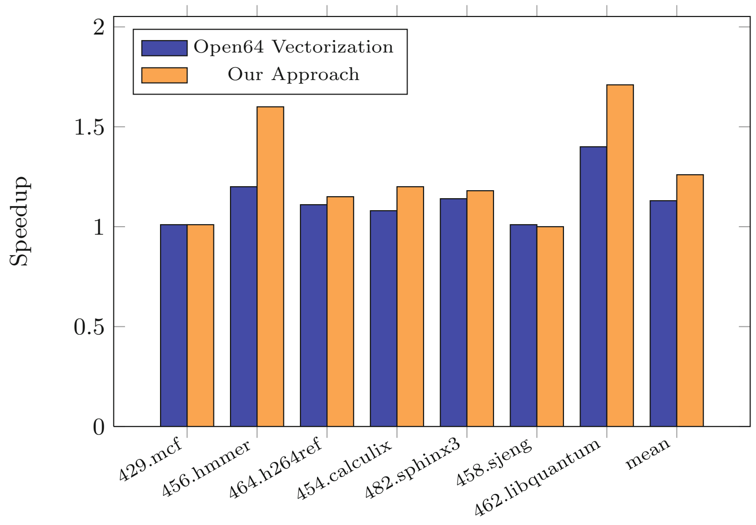

Fig. 5. Whole-program speedups: our approach compared with the Open64 vectorization

Figure 5 shows the whole-program speedups. The mean whole-program speedup achieved by our approach is 1.26x compared to the non-vectorization baseline and 1.11x compared to the Open64 vectorization. In most cases, the achieved whole-program speedups are consistent with the cumulative speedups of the most-time consuming kernels.

5 Conclusions

In this paper, we present an approach to refactoring loops with nested IF-statements by vectorizing them. Our new contributions to the state of the art in program vectorization are as follows:

- for loop-independent IF-statements, our modified loop unswitching method extends previous work to the case of arbitrarily nested IF-statements;

– for loop-dependent IF-statements, we develop a novel *IF-select transformation* method for targeting arbitrarily nested IF-statements and for SIMD extensions without predicated execution and masked instructions.

We integrate our approach into the Open64 compiler and we experimentally confirm its advantages using SPEC CPU2006 benchmarks on the SW26010 processor used in the Sunway TaihuLight supercomputer (#2 in the TOP500 list).

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with FastFlow. In: Proceedings of the 17th International Conference on Parallel Processing (Euro-Par), pp. 170–181 (2011). https://doi.org/10.1007/978-3-642-23397-5_17
2. Allen, J.R., Kennedy, K., Porterfield, C., et al.: Conversion of control dependence to data dependence. In: Proceedings of the Symposium on Principles of Programming Languages (POPL), Austin, Texas, USA, pp. 177–189 (1983). https://doi.org/10.1145/567067.567085
3. AMD: Using the x86 Open64 Compiler Suite (2012). For x86 Open64 version 4.5.2
4. ARM. https://developer.arm.com/products/processors/cortex-a/cortex-a8. Accessed 24 Sept 2018
5. Danelutto, M., Garcia, J.D., Sanchez, L.M., Sotomayor, R., Torquati, M.: Introducing parallelism by using REPARA C++11 attributes. In: 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 354–358 (2016). https://doi.org/10.1109/PDP.2016.115
6. Free Software Foundation: Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc/. Accessed 24 Sept 2018
7. Fu, H., Liao, J., Yang, J., et al.: The Sunway TaihuLight supercomputer: system and applications. Sci. China Inf. Sci. **59**, 1–16 (2016)
8. Haidl, M., Moll, S., Klein, L., Sun, H., Hack, S., Gorlatch, S.: PACXXv2 + RV: an LLVM-based portable high-performance programming model. In: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, pp. 7:1–7:12 (2017). https://doi.org/10.1145/3148173.3148185
9. Henning, J.L.: SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput. Arch. News **34**, 1–17 (2006)
10. IBM. https://www.ibm.com/systems/power/hardware/power8/. Accessed 24 Sept 2018
11. Intel: Intel C++ Compiler Developer Guide and Reference (2017). Version 18.0
12. Karrenberg, R., Hack, S.: Whole-function vectorization. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), Chamonix, France, pp. 141–150 (2011). https://doi.org/10.1109/CGO.2011.5764682
13. Larsen, S., Amarasinghe, S.P.: Exploiting superword level parallelism with multimedia instruction sets. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, pp. 145–156 (2000). https://doi.org/10.1145/358438.349320
14. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), San Jose, CA, USA, pp. 75–88 (2004)

15. Naishlos, D.: Autovectorization in GCC. In: Proceedings of the GCC Developers Summit, Ottawa, Ontario, Canada, pp. 105–118 (2004)
16. Shin, J., Hall, M.W., Chame, J.: Superword-level parallelism in the presence of control flow. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), San Jose, CA, USA, pp. 165–175 (2005)
17. Smith, J.E., Faanes, G., Sugumar, R.A.: Vector instruction set support for conditional operations. In: Proceedings of the International Symposium on Computer Architecture (ISCA), Vancouver, BC, Canada, pp. 260–269 (2000)
18. Sreraman, N., Govindarajan, R.: A vectorizing compiler for multimedia extensions. Int. J. Parallel Program. **28**, 363–400 (2000)
19. Thomas, J., Allen, F., Cocke, J.: A Catalogue of Optimizing Transformations. Prentice-Hall, Englewood Cliffs (1971)
20. TOP500. https://www.top500.org/lists/2018/06/. Accessed 24 Sept 2018