Comparison of Vectorization Capabilities of Different Compilers for X86 and ARM CPUs

Nazmus Sakib*, Tarun Prabhu†, Nandakishore Santhi†, John Shalf‡, Abdel-Hameed A. Badawy*
*Klipsch School of ECE, New Mexico State University, Las Cruces, NM 88003, USA
†Los Alamos National Laboratory, Los Alamos, NM 87545, USA
‡Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
*{nsakib6, badawy}@nmsu.edu, †{tarun, nsanthi}@lanl.gov, ‡jshalf@lbl.gov

***Abstract*—Most modern processors contain vector units that simultaneously perform the same arithmetic operation over multiple sets of operands. The ability of compilers to automatically vectorize code is critical to effectively using these units. Understanding this capability is important for anyone writing compute-intensive, high-performance, and portable code. We tested the ability of several compilers to vectorize code on x86 and ARM. We used the TSVC2 suite, with modifications that made it more representative of real-world code. On x86, GCC reported 54% of the loops in the suite as having been vectorized, ICX reported 50%, and Clang, 46%. On ARM, GCC reported 56% of the loops as having been vectorized, ACFL reported 54%, and Clang, 47%. We found that the vectorized code did not always outperform the unvectorized code. In some cases, given two very similar vectorizable loops, a compiler would vectorize one but not the other. We also report cases where a compiler vectorized a loop on only one of the two platforms. Based on our experiments, we cannot definitively say if any one compiler is significantly better than the others at vectorizing code on any given platform.**

I. Introduction

Early supercomputers, like the Cray machines, had vector units to take advantage of the data parallelism common to many computationally intensive scientific applications. Such vector units exist on most processors.

Maximizing the utilization of these vector units requires the appropriate use of vector instructions. However, programming in a high-level language like C with platform-specific vector intrinsics or writing assembly by hand is cumbersome, error-prone, not portable, and unlikely to result in optimal performance unless done by an expert.

High-quality vectorizing compilers are more likely to produce correct, high-performance code. Some also support generating code for different hardware platforms, allowing a programmer to obtain vectorized code for any platform supported by the compiler from a single code base.

GCC [1] and Clang [2] are widely used open-source compilers that can generate code for X86 and ARM (among other platforms). The Intel oneAPI DPC++/C++ Compiler [3] and ARM Compiler for Linux (ACFL) [4] are proprietary, vendor-provided compilers for X86 and ARM, respectively. In this paper, we compare the ability of these compilers to vectorize on two widely used hardware platforms and compare the performance of the resulting code. We also perform a detailed analysis of the code generated by these compilers in cases where one significantly outperforms the others. Prior studies [5], [6] have compared the size and relative performance of the code generated by some of these compilers but not their vectorization abilities. Others [7], [8], [9], [10], [11] have studied the compiler's ability to vectorize, but none evaluated the same compiler on different hardware platforms. Pohl *et al.* [12] have studied the accuracy of speedup prediction by compilers on different platforms, but they did not compare the compilers' ability to vectorize.

II. Evaluation

In this section, we discuss our choice of compilers and hardware platforms.

A. Compilers

Table I lists the versions of the compilers used in our experiments. For the vendor-provided compilers, we used the latest versions that were available on our test system. Note that the ACFL 22.2 is based on LLVM 13.0.1, which is older than the Clang version that was used. We expected the two vendor-provided compilers, ICX on x86 and ACFL on ARM, to outperform the open-source compilers, GCC and Clang.

B. Benchmark

TSVC (Test Suite for Vectorizing Compilers) [13] is a well-known benchmark suite that is used to assess a compiler's ability to vectorize. This suite consists of 151 loop nests containing a variety of control-flow and memory-access patterns such as conditional branches, non-unit strides, reverse array accesses, indirect memory accesses, etc. A variant of TSVC is TSVC2 [14] which utilizes modern C features and prevents function inlining.

In TSVC2, each loop nest is contained within a function with exactly one loop nest per function. Since every function contains exactly one loop nest, we use the name of the containing function to refer to a loop nest.

TABLE I
Compilers and their versions

Name	Version
GCC	14.1.1
Clang	18.1.8
ICX	2024.0.2
ACFL	22.2

TABLE II
HARDWARE SPECIFICATION

Name	Architecture	Model	Vendor
Intel	x86_64	Xeon(R) Gold 6152	Intel
ARM	aarch64	A64FX	Fujitsu

The loop nests generally perform 32-bit floating point operations on one or more arrays. An outer loop wraps the nest, resulting in redundant computations. This is done to minimize the effect of noise and jitter on the timings and accommodate systems that lack high-resolution timers.

In TSVC2, the arrays operated on by the loops are global with sizes that are known at compile time. The trip counts of the loop are also compile-time constants. This is not representative of scientific applications where the arrays are usually dynamically allocated and the trip counts are often input-dependent. In order to obtain a more realistic assessment of the compilers i.e. how they performed on real-world code, the code in TSVC2 was modified so the array sizes and the loop trip counts would not be compile-time constants [15]. This was achieved by bundling them together in a `struct` which was then passed to the functions. This ensured that the compiler would have to perform more sophisticated analyses to ensure the safety and accurately estimate the profitability of optimizations such as vectorization and loop unrolling. Siso *et al.* [16] demonstrated the effect of withdrawing some compile-time information such as globally known array bounds. Here, we withdraw all global information.

C. Hardware

Details about the hardware platforms on which we carried out the experiments are provided in Table II.

III. RESULTS AND ANALYSIS

A. X86

GCC and Clang were passed `-O3 -march=skylake-avx512 -mprefer-vector-width=512` when compiling the test-suite. The last option was replaced with `-qopt-zmm-usage=high` on ICX. The vectorization reports were generated with `-fopt-info-all` on GCC, `-Rpass=loop-vectorize -Rpass-missed=loop-vectorize` on Clang, and `-qopt-report` on ICX.

Figure 1 shows the number of loops vectorized by GCC, Clang, and ICX. Out of the 151 loops, GCC did not vectorize 46%, Clang did not vectorize 54% and ICX did not vectorize 50%. Figure 2 shows the geometric mean of the execution time of code vectorized by all three compilers. The code generated by ICX was fastest for 40% of the loops, GCC was fastest for 39% and Clang was fastest for 21%. Next, we discuss the relative performance of the compilers in greater detail.

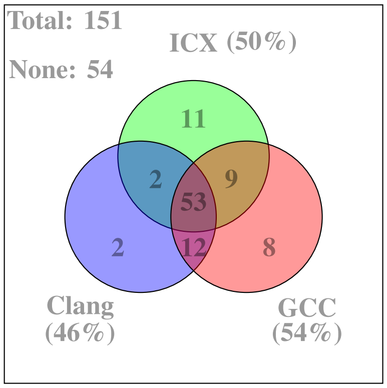

Fig. 1. Loops vectorized by GCC, ICX, and Clang on x86

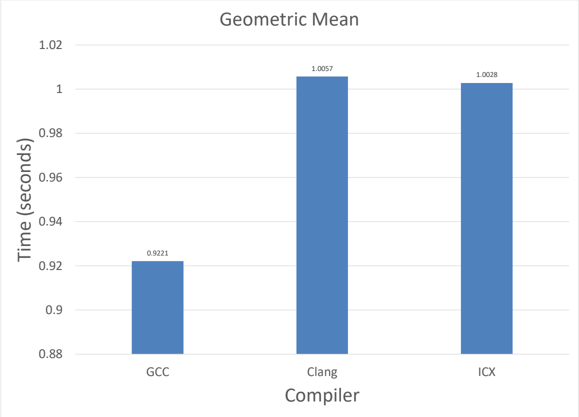

Fig. 2. Geometric Mean of Execution Time

a) GCC: Figure 3 shows the relative execution time of loops reported as having been vectorized by GCC but not Clang or ICX. Out of those 8 loops, GCC was slower than Clang and ICX in 2 cases. One of these loops had a loop carried read-after-write (RAW) dependence which GCC partially vectorized. This optimization did not prove to be beneficial. Some characteristics seen in the 6 loops where the vectorized code generated by GCC was better include:

- Conditional branching
- Non-unit but constant stride memory access
- Reverse array access

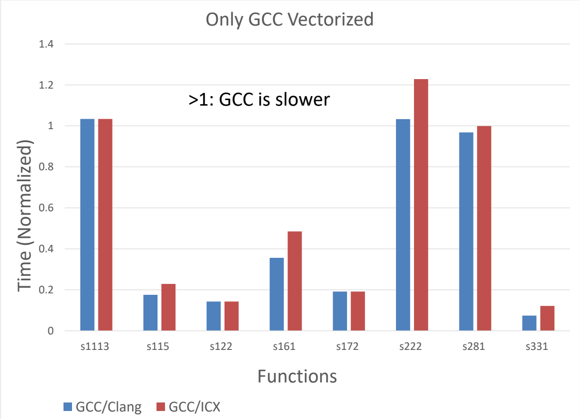

Fig. 3. Execution time of loops vectorized by GCC only

```
1 for (int i = 0; i < lEN_1D; i++)
2   x = a[lEN_1D-i-1] + b[i] * c[i];
3   a[i] = x-(real_t)1.0;
4   b[i] = x;
```

Fig. 4. Loop `s281`

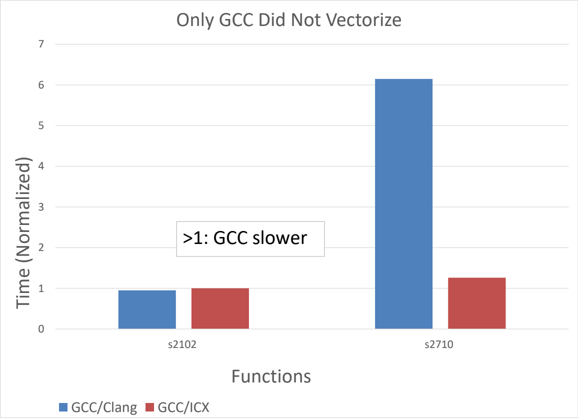

Fig. 5. Execution time of loops not vectorized by GCC only

One such loop is `s281` as shown in Figure 4. GCC used *neg* instruction to ensure reverse order access of `a[]` on line 2. Figure 5 shows the relative execution time of loops reported as having been vectorized by both Clang and ICX, but not GCC. Loop `s2102` creates an identity matrix by setting the diagonal elements to one and everything else to zero. Both Clang and ICX used vector scatter instructions which did not provide any performance improvement over non-vectorized stores.

b) Clang: Figure 6 shows two loops that were vectorized by Clang only.

Figure 7 shows the C code for loop `s1232`. The stride for all 2D arrays are constant (the size of row). Figure 8 is the assembly produced by Clang. The loads and stores are performed using masked gather and scatter instructions, which did not yield any improvement over the unvectorized code produced by GCC and ICX. Figure 9 shows the normalized execution time of loops not vectorized by Clang but vectorized by both GCC and ICX. 7 of these loops perform reductions. The vectorized code generated by GCC was not noticeably faster than the unvectorized code generated by clang. We examined the vectorized code generated by ICX and GCC for one such loop, `s3111`.

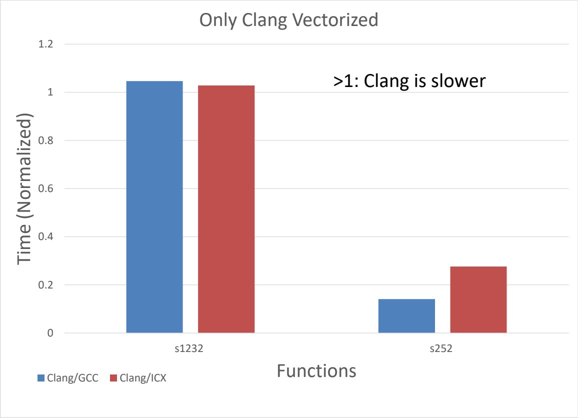

Fig. 6. Execution time of loops vectorized by Clang only

```
for (int j = 0; j < lEN_2D; j++)
  for (int i = j; i < lEN_2D; i++)
    (*aa)[i][j] = (*bb)[i][j] + (*cc)[i][j];
```

Fig. 7. Loop `s1232`

```
vpmullq      %zmm8,%zmm0,%zmm1
kxnorw       %k0,%k0,%k1
vxorps       %xmm2,%xmm2,%xmm2
kxnorw       %k0,%k0,%k2
vgatherqps   (%r10,%zmm1,4),%ymm2{%k1}
vxorps       %xmm3,%xmm3,%xmm3
vgatherqps   (%r11,%zmm1,4),%ymm3{%k2}
vaddps       %ymm3,%ymm2,%ymm2
kxnorw       %k0,%k0,%k1
vscatterqps  %ymm2,(%rbx,%zmm1,4){%k1}
vpaddq       %zmm10,%zmm0,%zmm0
add          $0x8,%r9
jne          30000 <s1232+0x4e0>
```

Fig. 8. Assembly from Clang for loop `s1232`

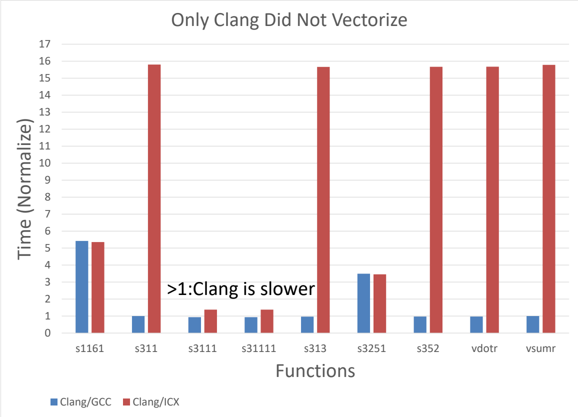

Fig. 9. Execution time of loops not vectorized by Clang only

Figure 10 is the C code for loop `s3111`. GCC partially vectorized this using vector load instruction. ICX, on the other hand, used vector loads, compares and adds. The temporary store also used vector instructions.

c) ICX: Figure 11 shows the loops vectorized by ICX, but not the other compilers. Some common features shared by these loops are:

- Indirect addressing
- Reductions
- Non-unit stride access

```
for (int i = 0; i < lEN_1D; i++)
  if (a[i] > (real_t)0.)
    sum += a[i]
```

Fig. 10. Loop `s3111`

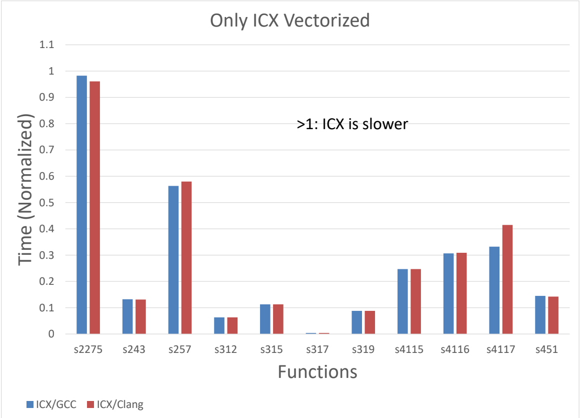

Fig. 11. Execution time of loops vectorized by ICX only

```
1 q = (real_t)1.
2   for (int i = 0; i < lEN_1D/2; i++)
3     q *= (real_t).99
```

Fig. 12. Loop `s317`

Figure 12 is the C code for loop `s317`. The statement in line 3 can be re-written in closed form as $q = 0.99^n$ where n is the loop trip count. Figure 13 shows part of the assembly generated by ICX. Lines 1 to 4 perform the multiplication in line 3 of Figure 12. The initial value of *%zmm0* is set to 1 (not shown in the listing). Using vector multiplication, the number of iterations is reduced from `len2D/2` to `(len2D/2)/16`. The individual values in *%zmm0* are multiplied Lines in 5 to 12 which results in the closed form solution.

Figure 14 shows the normalized execution time of loops not vectorized by ICX only. Nearly 50% of these loops have array subscripts that are linear functions of the loop iterator and some induction variable other than the loop iterator.

B. ARM

We used the following flags when compiling the test-suite on ARM: `-O3 -mcpu=a64fx+sve -msve-vector-bits=512`. The vectorization reports were generated using the same options as x86. Figure 15 shows the number of loops vectorized by GCC, Clang, and ACFL. Out of the 151 loops, GCC did not vectorize 44%, Clang did not vectorize 53% and ACFL did not vectorize 46%. These results are similar to those reported by Bine *et al.* [7].

```
 1 vmulps %zmm1,%zmm0,%zmm0 //%zmm0=q, %zmm1=0.99
 2 add             $0x10,%eax
 3 cmp             %r12d,%eax
 4 jl              4468a0 <s317+0x1a0>
 5 vextractf64x4 $0x1,%zmm0,%ymm1
 6 vmulps          %zmm1,%zmm0,%zmm0
 7 vextractf128  $0x1,%ymm0,%xmm1
 8 vmulps          %xmm1,%xmm0,%xmm0
 9 vshufpd         $0x1,%xmm0,%xmm0,%xmm1
10 vmulps          %xmm1,%xmm0,%xmm0
11 vmovshdup       %xmm0,%xmm1
12 vmulss          %xmm1,%xmm0,%xmm0
```

Fig. 13. Assembly from ICX for loop `s317`

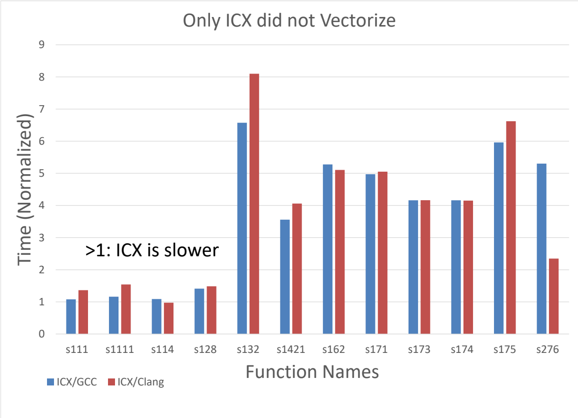

Fig. 14. Execution time of loops not vectorized by ICX only

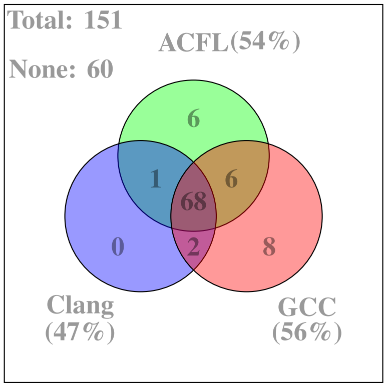

Fig. 15. Loops vectorized by GCC, ACFL, and Clang on ARM

Figure 16 shows the geometric mean of the execution time of code vectorized by all three compilers. Of these, the code generated by Clang was fastest for 65% of the loops, GCC was fastest for 22% and ACFL was fastest for 13%.

a) GCC: Figure 17 shows the normalized execution time of loops that were vectorized by GCC but not by either Clang or ACFL. `s2710` is the only loop that was vectorized by both

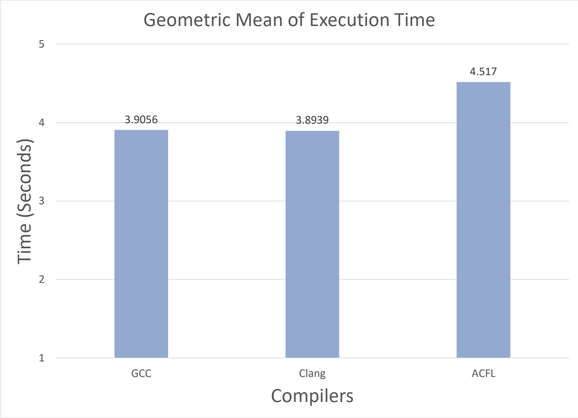

Fig. 16. Geometric Mean of Execution Time

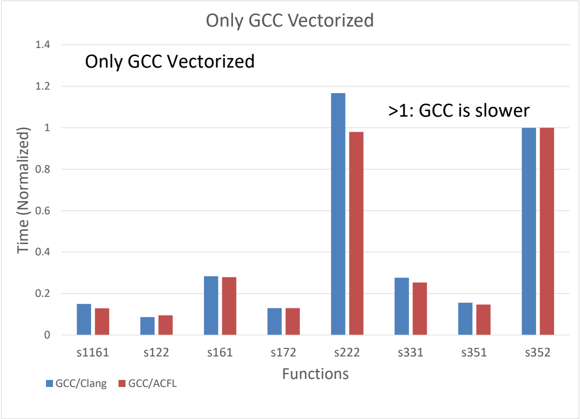

Fig. 17. Execution time of loops vectorized vectorized by GCC only

```
1 for (int i = 1; i < lEN_1D; i++)
2   a[i] += b[i] * c[i]
3   e[i] = e[i - 1] * e[i - 1]
4   a[i] -= b[i] * c[i]
```

Fig. 18. Loop s222

ACFL and Clang, but not GCC. Most of these loops are the same as those in Figure 3.

Despite being vectorized, the performance of loops `s222` and `s352` did not improve. Figure 18 is the C code for loop `s222`. GCC vectorized only the statements in line 2 and 4. The statement in line 3 has a read-after-write (RAW) dependency which cannot be vectorized.

Figure 19 is the C code for loop `s352`. Even though Clang and ACFL did not vectorize it, both unrolled it by factors of 20 and 24 respectively, which might have resulted in the speedup.

b) Clang: Figure 20 shows the normalized execution time of loops that were not vectorized by Clang, but were vectorized by the other compilers. Every loop vectorized by Clang was vectorized by either ACFL or GCC. Some of the features of the 4 loops where Clang was not slower despite not vectorizing are:

- Write-after-read dependence (WAR) for a single iteration
- Reverse data access
- Indirect memory lookup

Figure 21 is the C code for loop `s4115`. Figure 22 is the assembly generated by ACFL. GCC produced similar assembly. The result of the load instruction in line 1 is stored in *z1* which is used in the offset calculation for the gather instruction on line 4.

```
for (int i = 0; i < lEN_1D; i += 5)
  dot = dot + a[i] * b[i] +
        a[i + 1] * b[i + 1] +
        a[i + 2]* b[i + 2] +
        a[i + 3] * b[i + 3] +
        a[i + 4] * b[i + 4]
```

Fig. 19. Loop s352

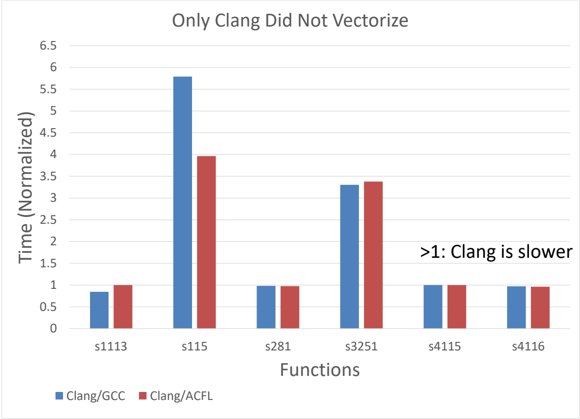

Fig. 20. Execution time of loops not vectorized by Clang only

```
for (int i = 0; i < lEN_1D; i++)
  sum += a[i] * b[ip[i]]
```

Fig. 21. Loop s4115

Clang, on the other hand, did not vectorize the loop, but unrolled it by a factor of 8 instead. Also, the *ldpsw* instruction was used which loads a pair of words. The first set of unrolled instructions is presented in Figure 23. It is not clear why the vector gather instructions were not profitable.

c) ACFL: Figure 24 shows 4 loops that were vectorized by ACFL, but not by any of the other compilers. Figure 25 shows 2 loops that were vectorized by both GCC and Clang, but not ACFL.

Only two vectorized loops were faster. Figure 26 is the C code for loop `s453`. The computations in line 2 and 3 can be re-written as `a[i]=(2*i+2)*b[i]`, which is what ACFL did as shown in Figure 27. Another loop that was vectorized only by ACFL and showed better performance is loop `s442`, which contains a `switch` statement. ACFL was able to vectorize it using *cmpeq* and *sel* instructions and predicate registers which

```
1 ld1w     {z1.s}, p0/z, [x19, x8, lsl #2]
2 ld1w     {z0.s}, p0/z, [x20, x8, lsl #2]
3 add      x8, x8, x24
4 ld1w     {z1.s}, p0/z, [x21, z1.s, sxtw #2]
5 fmul     z0.s, z0.s, z1.s
6 fadda    s2, p0, s2, z0.s
7 whilelo  p0.s, x8, x22
8 b.mi     43466c <s4115+0xa4>
```

Fig. 22. Assembly from ACFL for loop s4115

```
mov      x8, xzr
ldpsw    x11, x12, [x10, #-16]
ldp      s1, s2, [x9, #-16]
sub      x8, x8, #0x8
ldr      s0, [x21, x11, lsl #2]
fmadd    s0, s1, s0, s8
```

Fig. 23. Assembly from Clang for loop s4115

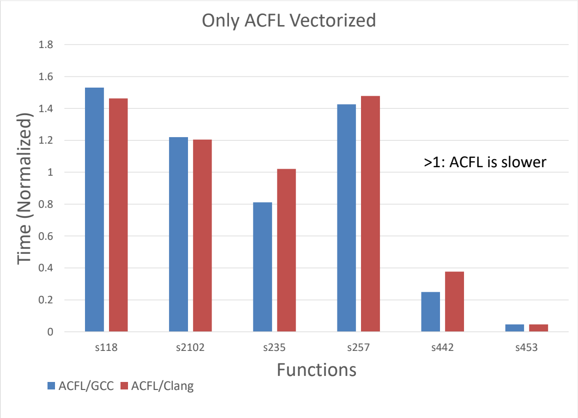

Fig. 24. Execution time of loops vectorized by ACFL only

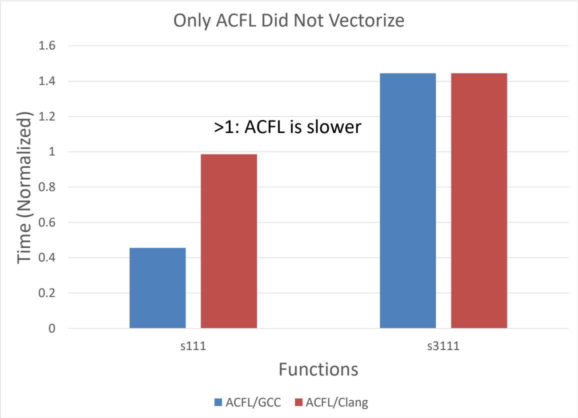

Fig. 25. Execution time of loops not vectorized by ACFL only

can perform loads and other operations conditionally.

C. X86 and ARM

This section summarizes the difference in behavior of GCC and Clang across the two platforms. Almost all loops that were vectorized by GCC or Clang on one platform were vectorized on the other. We found 3 loops that were not vectorized by GCC on X86 which were vectorized on ARM. 2 of them used vector gather instructions. We also found 3 loops vectorized by Clang on x86 but not on ARM. 2 of these had non-unit but constant stride memory accesses. 4 loops were reported vectorized by Clang on ARM but not on x86, all of which performed reductions.

```
for (int i = 0; i < lEN_1D; i++)
  s += (real_t)2.;
  a[i] = s * b[i];
```

Fig. 26. Loop s453

```
scvtf   s0, x8
ld1w    {z1.s}, p0/z, [x21, x8, lsl #2]
fadd    s0, s0, s0
mov     z0.s, s0
fadd    z0.s, z0.s, z2.s
fadd    z0.s, z0.s, z3.s
fmul    z0.s, z0.s, z1.s
st1w    {z0.s}, p0, [x20, x8, lsl #2]
add     x8, x8, x24
whilelo p0.s, x8, x28
```

Fig. 27. Assembly from ACFL for loop s453

```
for (int i = 0; i < lEN_1D; i++)
  a[i] += b[ip[i]] * s;
```

Fig. 28. Loop s4112

Out of 65 loops that were vectorized by both GCC and Clang on x86, GCC outperformed Clang in 34 (52%) of the cases. For ARM, out of 70 loops vectorized by both, Clang outperformed GCC in 51 (73%) of the cases. Of the loops that were vectorized by both GCC and Clang on both x86 and ARM, the code produced by GCC outperformed that produced by Clang in 14 cases, while Clang outperformed GCC in 26 cases.

D. Indirect Memory Access

There are 8 loops in TSVC2 with indirect memory accesses. Both x86 and ARM provide vector gather/scatter instructions. Neither GCC nor Clang were able to utilize them to vectorize these 8 loops on x86. However, ICX was able to vectorize 2 loops. On ARM, GCC and ACFL were able to vectorize the same 2 loops but Clang was not able to vectorize any. We discussed s4115 in Figure 21, which was vectorized on ARM by GCC and ACFL but not Clang. GCC was not able to vectorize it on x86. Figure 28 is the C code for loop s4112. Despite being similar to loop s4115, no compiler, on either platform, vectorized it.

IV. Conclusion and Future Work

We have investigated the ability of several compilers to vectorize on two different hardware platforms. 35% of the loops in the TSVC2 suite were vectorized by all three compilers on x86 and 36% were not vectorized by any of them. For ARM, these numbers are 45% and 40% respectively. GCC reported more loops being vectorized than Clang on both X86 and ARM. However, of the loops vectorized by both GCC and Clang, the code generated by GCC performed better on x86 whereas code generated by Clang performed better on ARM. There were cases where the compilers would vectorize a loop on X86 but not on ARM (and vice versa). There were no immediately obvious, consistent strengths or weaknesses in any one compiler's ability to vectorize. It was also unclear when the code generated by any given compiler would outperform the others. We have reported the few patterns that were apparent to us. In future work, we intend to focus on loops from specific domains, which could help determine if any compiler is particularly suitable for a given domain.

V. Acknowledgment

We would like to thank the anonymous reviewers for their feedback. This work is partially supported by Triad National Security, LLC subcontracts #581326 and #C4975. Also, this work was partially supported by The Regents of the University of California (Lawrence Berkeley National Laboratory) under Subcontract #7558382. Any opinions, findings, or conclusions expressed in this paper do not necessarily represent the views of the DOE or the US Government.

References

[1] GNU, https://gcc.gnu.org/.

[2] LLVM, https://clang.llvm.org/.

[3] Intel, https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html.

[4] ARM, https://developer.arm.com/Tools%20and%20Software/Arm%20Compiler%20for%20Linux.

[5] J.-J. Kim, S.-Y. Lee, S.-M. Moon, and S. Kim, "Comparison of llvm and gcc on the arm platform," in *2010 5th International Conference on Embedded and Multimedia Computing*, 2010, pp. 1–6.

[6] C. Park, M. Han, H. Lee, and S. W. Kim, "Performance comparison of gcc and llvm on the eisc processor," in *2014 International Conference on Electronics, Information and Communications (ICEIC)*, 2014, pp. 1–2.

[7] B. Brank and D. Pleiter, "Assessing the state of autovectorization support based on sve," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, pp. 556–562.

[8] S. Maleki, Y. Gao, M. J. Garzar´n, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 372–382.

[9] O. V. Moldovanova and M. G. Kurnosov, "Auto-vectorization of loops on intel 64 and intel xeon phi: Analysis and evaluation," in *Parallel Computing Technologies*, V. Malyshkin, Ed. Cham: Springer International Publishing, 2017, pp. 143–150.

[10] M. Alvanos and P. Trancoso, "Video simdbench: Benchmarking the compiler vectorization for multimedia applications," in *2016 Euromicro Conference on Digital System Design (DSD)*, 2016, pp. 168–175.

[11] J. G. Feng, Y. P. He, and Q. M. Tao, "Evaluation of compilers' capability of automatic vectorization based on source code analysis," *Scientific Programming*, vol. 2021, no. 1, p. 3264624, 2021.

[12] A. Pohl, B. Cosenza, and B. Juurlink, "Vectorization cost modeling for neon, avx and sve," *Performance Evaluation*, vol. 140, p. 102106, 2020.

[13] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: a test suite and results," in *Supercomputing '88:Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I*, 1988, pp. 98–105.

[14] S. McIntosh-Smith, *TSVC 2*, University of Bristol, 2023, https://github.com/UoB-HPC/TSVC_2.

[15] https://github.com/NMSU-PEARL/tsvc_withArgs.

[16] S. Siso, W. Armour, and J. Thiyagalingam, "Evaluating auto-vectorizing compilers through objective withdrawal of useful information," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–23, 2019.