On eliminating if-conversions

Florian Freitag
e11908096@student.tuwien.ac.at
TU Wien

Abstract

Most computer architectures feature SIMD (Single Instruction Multiple Data) instruction. Since their introduction, compilers attempted to exploit them through auto-vectorization to generate faster code. If-conversion is one established technique they apply to vectorize loops with conditions. In this paper, we explore modern attempts to eliminate if-conversions to further improve performance.

1 Introduction

SIMD instructions (Single Instruction Multiple Data) allow to perform the same arithmetic operation on multiple operands. Utilizing these instructions improves performance [1]. These instructions often operate on a bit length between 128 and 512 bits.

To reverse an ASCII string, 1.5 move instructions are needed per character (three instructions to swap two characters). With the 128bit `pshufb` instruction on the intel x86 architecture [1], it is possible to reverse 16 bytes at once in three instructions (the `pshufb` instruction requires a load to and store from a 128bit xmm register), which would otherwise need 24 instructions. Therefore the instructions needed per character can be reduced from 1.5 to 0.1875.

While most architectures have some form of SIMD instructions, the exact instructions can differ quite a lot and even on the same architecture, not all SIMD instructions might be supported by all processors. Therefore, implementing a program by hand is not only error-prone and requires sophisticated knowledge of the architecture but also reduces portability.

Compilers often employ many optimizations, either on the flow graphs and data flow graphs or on the generated instructions directly [2].

One of these optimizations can be to search for patterns that qualify for SIMD instructions for the target architecture in a step called auto-vectorization. Therefore, programmers can write their code in a not-vectorized manner and can rely on the compiler to choose the best SIMD instruction.

Auto-vectorization promises great performance improvements and has therefore been a research topic since the introduction of SIMD instructions. The reason for the extensive research is also that SIMD instructions introduce some restrictions. Since vectorized code executes multiple iterations at once it is difficult and often impossible to vectorize loops where one iteration depends on a side effect of the previous iteration. Moreover, SIMD instructions execute the same operations on all operands which makes loops with non-linear control flow difficult to vectorize.

2 If-Conversion

If-conversion is a technique to rewrite control dependency as data-dependency that was first developed to allow auto-vectorization to vectorize loops with control-flow [3].

Control-dependency means here that a statement will be executed, depending on the control flow. Whereas data-dependency describes a statement that depends on other data, that must be computed first. While control-dependent code cannot simply get vectorized, no such problem exists for data dependencies.

While this technique was initially developed, for auto-vectorization, it can also be utilized simply because of its property reduce branching code. For example, if-conversion have gained new importance with the introduction of branch prediction in CPUs, as branches that are hard to predict can be entirely eliminated and therefore miss prediction by branch predictors can be avoided [4]. Some recent research even used this technique to prevent the exploitation of side-channel vulnerabilities (timing attacks) in programs [5].

```
if (x > 0) {
  y += 3
}
```

Listing 1: Branching code

```
char c = x > 0;
char o = 3 * c;
y += o;
```

Listing 2: Unvectorized if-conversion

In Listing 1 we can see a simple control-flow dependency, where the value of `y` only gets modified if `x` is a non-zero positive integer.

Listing 2 archives the same effect without branching. In this particular example, we exploit that conditions in C are mapped to 1 or 0 and that 1 is the neutral element of multiplication and zero is the neutral of addition.

In most cases, this transformation on its own wouldn't bring any speedups, but mostly reduce the codes performance, especially since modern hardware and their branch prediction is quite good [4].

However, this transformation allows us to utilize SIMD instructions. To apply this for loop vectorization we need to replace all operations from Listing 2 with SIMD versions. The condition result will be replaced by a vector that holds multiple condition results and is often referred to as a mask. With that mask, we can then execute a vectorized version of the multiplication and addition. Some SIMD instructions are designed with that mask and will conditionally execute depending on a mask like Intel's `vmaskmov` [1].

While if-conversion is a useful tool and can lead to performance improvements [3], it replaces jumps with arithmetic operations which are not free in terms of computation. Even more performance can be gained by eliminating the if-conversion entirely.

2.1 Evaluation

Evaluating, the performance impact of, if conversions are not straight forward as the authors of the original paper did not do any evaluation by themself. This was partly because their compiler wasn't finished yet and could only linearize small code blocks [3].

Since many modern compilers do apply if-conversions it is fair to assume that they still impact performance [6]. However, to actually evaluate their performance further work is required, in which if-conversions are disabled in a state of the art compiler, such as LLVM.

3 Static Uniformity

One property some modern attempts exploit is uniformity [7, 6]. We differentiate between static and dynamic uniformity, the latter will be discussed in the next chapter. A branch is considered static uniform if it can be proven at compile-time that all iterations will take the same branch.

```
int sum(int* data, int n, bool pos) {
  int res = 0;
  for (int i=0; i < n; i++) {
    int element = data[i];
    if (pos) {
      abs(&element)
    }
    res += element;
  }
  return res;
}
```

Listing 3: Static uniform

Listing 3 shows a loop where depending on the function's arguments all iterations will either call the `abs` function or jump over its invocation.

Linearizing, this loop with if-conversion would enable the compiler to vectorize the loop but would also introduce an unintended overhead, as the condition will be true for all iterations but will anyway calculated every iteration.

Some recent research was aimed at reducing such counterproductive conversions by only partially linearizing the CFG (control flow graph) [7].

3.1 Evaluation

In their implementation, the authors can outperform the if-conversion implemented in LLVM and can archive an average speedup of 146% over `clang -O3` [7].

4 Dynamic Uniformity

Often, we might not be able to prove uniformity during compilation but still encounter uniformity during runtime.

```
// Pretend data is read at runtime
int[] data = [0, 0, 0, 0, 1, 1]

for (int i = 0; i < 6; i++) {
   if (data[i])  {
     x[i] += 3
   }
}
```

Listing 4: Dynamic uniformity

In Listing 4 we cannot prove uniformity statically, but if we vectorize the loop by executing two iterations at once, we would find that either we would take the same branch for all pairs.

Researchers at IBM and the University of Toronto found a novel, called VecRC approach to, eliminate in these dynamic in some uniform environments [6]. They archive this by inserting additional checks at runtime and can therefore avoid if conversion on uniform iterations and only need to do that transformation on divergent branches.

```
for (int i = 0; i < 6; i+=2) {
    vec m = data[i]
    if (all(m)) {
      // Condition body
      simd_add(x[i], 3)
    } else if (none(m)) {
      // Do nothing
      // (because no else)
    } else {
      // If conversion
      vec o = simd_mul(m, 3)
      simd_add(x[i], o)
   }
}
```

Listing 5: Vectorized dynamic uniformity

Listing 5 shows a vectorized version of Listing 4 with the uniformity optimizations from VecRC. The example itself is pseudocode but reflects the algorithm described in the paper.

First, the algorithm checks if all conditions of the current iteration are true, if that is the case, a vectorized version of the condition body can be executed without any if-conversion. Next, a condition checks if all conditions are false, in this example nothing has to be done, but if the condition had an `else` it could execute a vectorized version of the else block. Finally, if the branches are divergent, a classic if-conversion is executed.

Obviously, these additional run-time checks, while less expensive than if -conversion, are also not free in terms of computation. In the worst case of divergent branches we still execute an if-conversion but have introduced two additional checks, which will deteriorate instead of improving performance. Therefore the authors of the papers introduced a cost model, which is used to decide if the additional runtime checks should be inserted.

The introduced cost model evaluates the penalty of inserting these checks on divergent branches and weights it against the gains in uniform ones. To accurately predict this cost, their implementation in LLVM makes use of LLVM's internal branch predictor.

For simplicity, Listing 5 shows a vectorization of only two elements per iteration. This metric is called the vector-factor and is chosen in optimally to minimize the cost. On the one hand, higher vector-factors can use fewer SIMD instructions for each calculation and therefore decreases the number of iterations. On the other hand, more conditions must be uniform, decreasing the probability of uniformity, and increasing the probability of divergent branches, the worst case scenario.

```
for (int i = 0; i < n; i!++) {
  if (cond[i]) {
    j++;
  }
  dst[j] = a[i] + b[i];
}
```

Listing 6: Loop carried data-dependency

The algorithm described in the VecRC paper can also vectorize some loops that were previously impossible to vectorize [6]. Listing 6 shows the loop carried data-dependency `j`, which prevents state-of-the-art vecotorizers to vectorize the loop. This is because the vectorized version would execute multiple iterations at once but the iterations are not independent from each other. With the newly introduced checks for dynamic uniformity, however, it is is possible to vectorize the loop if the condition is uniform, because if it is we can either always or never increment j in the vectorized version.

4.1 Evaluation

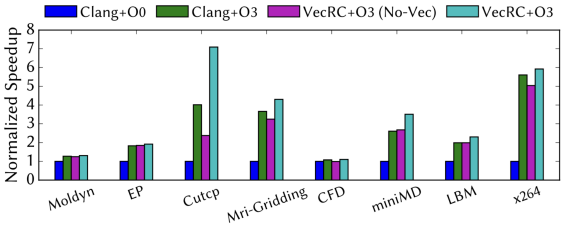

Figure 1: VecRC results on Intel Skylake

VecRC could archive a speedup over LLVM in most benchmarks, and on average had a performance improvement of 118% over `clang -O3`.

5 Related Work

If conversions are a technique to linearize control flow, which was invented to and is still used by compilers to auto-vectorize control flow [3].

The technique had quite a few renaissances since its introduction four decades ago, with the introduction of branch predicting microchips[4] and the wide availability and utilization of GPUs [8].

Only, partially linearizing the control flow can improve performance in cases where uniformity can be statically proven by avoiding unnecessary if-conversion [7].

Exploiting dynamic uniformity at runtime can further improve performance, but poses more challenges as it introduces runtime penalties that can outweigh the benefit [6].

Branching code executes different instructions based on the chosen branch. In some cases, an attacker can measure the execution time of a program, and depending on the time spent, they can predict which branch was chosen. This might reveal sensitive information. By linearizing these branches, we can force the CPU to always execute the same instructions and prevent these attacks [5].

One of the most relevant books in the world of compiler engineering is *Compilers: Principles Techniques and Tools*, often just referred as the *Dragon Book* because of the red dragon on the cover. It explains the backgrounds of a complete compiler from lexing to instruction-level optimization like auto-vectorization [2].

6 Conclusion

In this paper, we compared the initial paper introducing if-conversions to some modern attempts that further attempt to optimize code by better utilizing SIMD instructions.

Especially to VecRC, a novel solution that exploits dynamic uniformity in data during runtime, and can archive performance improvements of 1.18x over speedups `clang -O3` by avoiding if-conversions [6].

Another similar solution is exploiting static uniformity, which archived a higher speedup because no runtime checks are required. Furthermore, the authors of VecRC argue that static and dynamic uniformity exploitation does not obstruct each other and it should be possible to implement both to complement each other.

References

[1] A. Fog, 2022. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proc. 10th ACM SIGACT-SIGPLAN Symp. Princ. Program. Languages* in Popl '83, Austin, Texas, 1983, p. 177, doi: 10.1145/567067.567085. [Online]. Available: https://doi.org/10.1145/567067.567085

[4] E. Quinones, J.-M. Parcerisa, and A. Gonzailez, "Improving branch prediction and predicated execution in out-of-order processors," in *2007 IEEE 13th Int. Symp. High Perform. Comput. Architecture*, vol. 0, 2007, pp. 75–84, doi: 10.1109/HPCA.2007.346186.

[5] L. Soares, M. Canesche, and F. M. Q. Pereira, "Side-channel elimination via partial control-flow linearization," *ACM Trans. Program. Lang. Syst.*, May 2023, doi: 10.1145/3594736. [Online]. Available: https://doi.org/10.1145/3594736

[6] B. Liu, A. Laird, W. H. Tsang, B. Mahjour, and M. M. Dehnavi, "Combining run-time checks and compile-time analysis to improve control flow auto-vectorization," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.* in Pact '22, Chicago, Illinois, 2023, p. 439, doi: 10.1145/3559009.3569663. [Online]. Available: https://doi.org/10.1145/3559009.3569663

[7] S. Moll, and S. Hack, "Partial control-flow linearization," in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Implementation* in Pldi 2018, Philadelphia, PA, USA, 2018, p. 543, doi: 10.1145/3192366.3192413. [Online]. Available: https://doi.org/10.1145/3192366.3192413

[8] G. Anantpur Jayvant
and R., "Taming control divergence in gpus through control flow linearization," in *Compiler Construction*, Berlin, Heidelberg, 2014, pp. 133–153.