Binary Analysis for Missed Vectorization Opportunities Detection

AMOS HERZ and ALESSANDRO LEGNANI, ETH Zürich, Switzerland

1 INTRODUCTION

Modern compilers are often able to automatically vectorize code using SIMD instructions. Take, as an example, the following code snippet:

```
void copy(long *restrict a, long *restrict b, unsigned long n) {
    for (unsigned long i = 0ul; i < n; i++) {
        a[i] = b[i];
    }
}
```

Listing 1. copy.c

We can compile it with the following set of compiler flags

- -O3: tells the compiler to use the highest level of optimization available.
- `-fno-tree-loop-distribute-patterns`: prevents replacing the loop with a call to `memcpy`
- `-fno-tree-vectorize`: prevents vectorization

Which will produce the following assembly code:

```
.L3:
    movq  (%rsi,%rax,8), %rcx
    movq  %rcx, (%rdi,%rax,8)
    addq  $1, %rax
    cmpq  %rax, %rdx
    jne   .L3
```

Listing 2. copy.c compiled with vectorizations disabled

However, by compiling without the `-fno-tree-vectorize` flag, the compiler will produce the following vectorized code (note the use of wider instructions and registers):

```
.L4:
    movdqu  (%rsi,%rax), %xmm0
    movups  %xmm0, (%rdi,%rax)
    addq    $16, %rax
    cmpq    %rcx, %rax
    jne     .L4
```

Listing 3. copy.c compiled with vectorizations enabled

The main goal of this project is to develop a method for identifying missed opportunities for vectorization in existing code. That is, given an existing binary, we want to identify pieces of code and procedures that could have been vectorized by the compiler but were not.

2 RELATED WORK

Autovectorization is *difficult*, compilers tend to miss many optimizations (as shown by Feng et al. [3]) , and more than often vectorizing a small piece of code requires large changes to the whole code base (as was done for example by Chen et al. [2]).

Compilers have been shown to widely miss out on vectorization opportunities. As an example, Maleki et al. [4] report that in their research only 45-71% of the loops in a benchmark they developed

Authors' address: Amos Herz, amherz@ethz.ch; Alessandro Legnani, alegnani@ethz.ch, ETH Zürich, Zürich, Switzerland, 8092.

and only a few loops from the real applications are vectorized by the compilers they evaluated, which include widely popular compilers such as gcc (version 4.7.0).

Auto-vectorization is still an open field of research: attempts have been made at "fixing" the compiler work by post-processing compiled code (Porpodas and Ratnalikar [6]) or by applying Machine Learning to produce improved vectorization schemes (Mendis et al. [5]).

3 APPROACH

3.1 Dataflow Analysis

Let's look at Listing 2: there, we can realize that the `load` and `store` instructions are completely independent (from other instructions of the same type) and could, theoretically, be computed in parallel, therefore vectorized. This can be concluded by simply looking at the dataflow, an example on how a dataflow graph could look like is given in Figure 1.

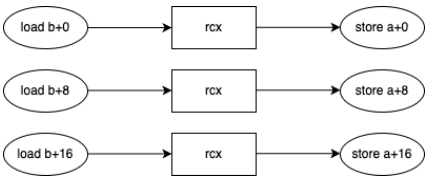

Fig. 1. Example of dataflow graph for `copy.c`

3.2 Dynamic Analysis

In order to build a dataflow graph we will be using the dynamic analysis framework Intel Pin[1]. Intel Pin allows us to instrument single instructions in the assembly to generate the appropriate trace information to construct our dataflow graph.

We opted to use dynamic analysis in our approach as it simplifies the approach of finding missed vectorization when compared to static analysis. Dynamic analysis allows us to generate the trace of a given program, containing all of its memory accesses, which can easily be converted into a dataflow graph. However, when performing dynamic analysis, we will need to be careful not to falsely report code with complex control flow as non-optimized.

```
#include <stdint.h>
void dyn(float *a, float *b, uint8_t *c, unsigned int n) {
    for (int i = 0; i < n; ++i) {
        if (c[i]) {
            a[i] = 2 * b[i];
        } else {
            a[i] = 4 * b[i];
        }
    }
}
```

Listing 4. dyn function

As an example, assume that in an execution of function `dyn` from Listing 4 we have that every element of array `c` is 0. In this case, we would observe that `a = 4 * b` and vectorization would therefore bt trivial. Therefore, if the code were not to be vectorized, we would flag this code as missing a vectorization opportunity, without considering the control flow operation. Note that this example is actually trivially vectorizable if we were to employ masked vector operations. However, this is only an illustrative example and one can consider even more complex control flow operations

[1]https://software.intel.com/sites/landingpage/pintool/docs/98830/Pin/doc/html/index.html

that make the code impossible to be vectorized. To fix this issue we can either use static analysis or consider the control flow operation in our dataflow graph. We plan on using the second approach as it is less complex to implement and are aware of the loss in precision this could result in.

All in all, we foresee that building a (correct) dataflow graph will likely be the greatest challenge this project poses. An alternative, would this challenge turn out to be too big of a one, could be to simply analyze the program trace directly, to track down for example repeating accesses to increasing (or decreasing) memory addresses that would be vectorizable but are not.

3.3 Real world examples of missed vectorization opportunities

By searching on the different bugs and issues trackers of the `clang` and `gcc` compilers we found two examples where the latest version of a compiler does not generate vectorized code. For this we are considering the latests versions of both `gcc` (13.2) and `clang` (18.1.0).

3.3.1 Example 1 (`gcc`). The following example has been taken from the `gcc` bugs tracker[2].

```
#include <stdint.h>

void ex1(int8_t *v, int8_t x, const uint64_t *bits, unsigned n) {
  int num_words = (n + 64 - 1) / 64; // round up to nearest quad-word
  for (int i = 0; i < n; ++i) {
    const uint64_t word = bits[i];
    for (int j = 0; j < 64; ++j) {
      v[i * 64 + j] += x * (bool)(word & (1UL << j));
    }
  }
}
```

Listing 5. Piece of code that `gcc` fails to vectorize

One would expect the compiler to generate a mask from the `word` variable and perform masked multiplication to vectorize the code. Compiling this code with the `-O3` flag on `gcc` yields no vectorized code, whereas `clang` unrolls the inner loop and, as expected, makes extensive use of vector instructions. A side-by-side comparison of the two compilers' work can be found at the following link: https://godbolt.org/z/W8v4Pc463.

3.3.2 Example 2 (`clang`). The following example has been taken from the `clang` issues tracker[3].

```
void cvt_8_32(uint32_t* op, uint8_t const* ints)
{
    // When this is outlined it generates correct code
    uint32_t out[16];
    uint8_t in[16];
    memcpy(in, ints, sizeof(in));
    for (int i = 0; i < 16; ++i) {
        out[i] = in[i];
    }
    memcpy(op, out, sizeof(out));
}
```

Listing 6. Piece of code that `clang` fails to vectorize

It appears that when the function in Listing 6 is inlined, `clang` fails at correctly vectorizing it. However `gcc` correctly vectorizes the code without any issue. A side-by-side comparison of the two compilers' work can be found at the following link: https://gcc.godbolt.org/z/x6nEh8oqe.

[2]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96888
[3]https://github.com/llvm/llvm-project/issues/74380

3.4 Test Inputs

To test our results, we use code snippets from an already existing autovectorization test suite: TSVC[4], first described by Callahan et al. [1]. Other random program generators such as YARPGen[5] could be used to generate more, less specific, test inputs. However, given that such less specific inputs hardly contains code that can be vectorized but is not, we opted not to utilize them.

4 IMPLEMENTATION AND EXPERIMENTAL SETTINGS

4.1 Project Pipeline

As mentioned previously, we test our implementation (implemented using the latest version of Intel Pin) using the latests versions of both `gcc` (13.2) and `clang` (18.1.0). In order to do so, we use pieces of code from the TSVC test suite that are known to be vectorizable, pass them to `gcc` and `clang` preventing them to vectorize the resulting binary and then test if our implementation can actually spot the (induced) missed vectorization opportunities. A diagram representing the pipeline of this project is provided in Figure 2.

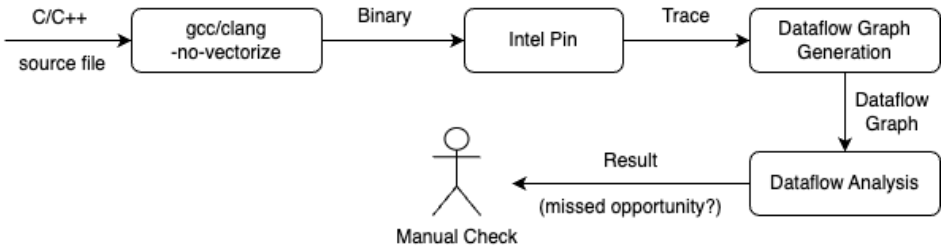

Fig. 2. Sketch of the pipeline of the whole project

4.2 Trace Generation (PinTool)

In order to trace our test program snippets we leverage the Intel Pin framework. Initially we only traced reads and writes to memory that are not performed on stack addresses. This can be used to detect strided accesses to certain memory areas which might be performed inside loops which, in turn, might be vectorizable. Having only information about read and writes is not sufficient to check whether or not a program has been correctly vectorized. Therefore, we added tracing for various operations like `mov` and `add`. The Pin framework makes it easy to instrument these instructions and trace the mnemonic of the operation performed. We encountered issues when tracing memory operations where the memory operand is neither an immediate or a register, like `add %rcx, 0x8(%rcx, %rax, 8)`, as there are multiple operands making up the memory one. In these cases our traces had some registers being flagged as invalid because they were being accessed by the Pin tool like a standard operand. We weren't able to solve these issues directly in Pin but managed to find a workaround whilst processing the trace (Section XXX). Having the ability to trace memory accesses as well as operations performed on registers (including registers from extensions like AVX) we are able to generate a dataflow graph precise enough to make conclusions about vectorization. An example trace of the program in Listing 7 is provided in Listing 8.

```
1 static int add(int *a, int *b, int *c, int n) {
2   for (int i = 0; i < n; ++i) {
3     c[i] = a[i] + b[i];
```

[4]https://github.com/UoB-HPC/TSVC_2
[5]https://github.com/intel/yarpgen

```
  }
  return 0;
}
```

Listing 7. add.c

```
0x4011f8: READ 0x7ffd344637e0 ecx
0x4011fb: READ 0x7ffd344637d0 ecx
0x4011fb: BinOp ADD *invalid* ecx ecx
0x4011fe: WRITE ecx 0x7ffd344637f0
0x401201: BinOp ADD #4 rax rax
0x4011f8: READ 0x7ffd344637e4 ecx
0x4011fb: READ 0x7ffd344637d4 ecx
...
```

Listing 8. Partial trace of add.c

4.3 Dataflow Graph Generation

To generate the dataflow we leverage the information contained in the trace, generated in the step before. The graph consist of a directed graph composed of nodes, representing an operand such as a register, an immediate value or a value at a memory location, and edges, representing the operation being performed on these operands. Listing 9 contains a high level overview of our proposed algorithm for graph generation.

```
for operation in trace {
  match operation {
     MemoryOperation(src, dst) => {
        label = (dst is register) ? "Read" : "Write"
        add edge from src to dst with label
     }
     Operation([srcs], dst) => {
        for src in srcs {
           label = mnemonic of operation
           add edge from src to dst with label
        }
     }
  }
}
```

Listing 9. Algorithm used to generate dataflow graphs starting from a program trace.

It should be noted that each subsequent access to an operand that has been written to can effectively be considered a different operand. Therefore the implementation of the algorithm makes sure to always connect edges using the node corresponding to the latest snapshot at which the operand has been written to.

We made sure that only information that is critical to the data flow of the program is encoded in the graph, through a step of reduction. As an example, a register like `%rax` could be used as a "counter" inside a `for` loop, and we must make sure that we do not pollute our graph with operations irrelevant to its possible vectorization. As the analysis is performed on a trace, the program's control flow is already directly embedded in te dataflow and don't therefore need any of the side-computations. Images **??** and **??** provide a concrete example: they are the two graphs generated from the `add.c` function from Listing 7

To remove this unnecessary information we determine the connected components of the dataflow graph and remove any connected components that do not result in memory being read and then written. This works incredibly well for isolated functions that do not return values. In bigger programs, we might have to conserve the connected components that end with a write to `%rax` (the return register), by instrumenting the return instruction as well.

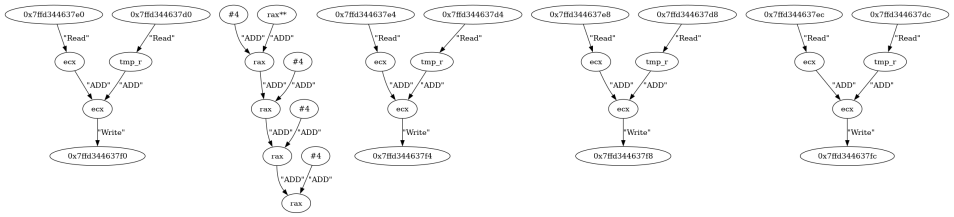

Fig. 3. Complete graph generated from add.c, not how the second tree from the left is simply the loop index computation and therefore irrelevant for vectorization

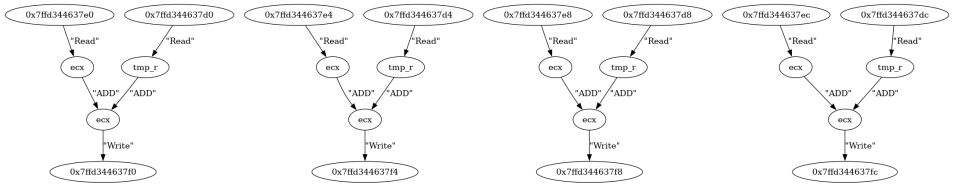

Fig. 4. Reduced graph generated from add.c, note how the loop index computation isn't present

4.4 Dataflow Analysis

As the possibility space of different kinds of missed vectorization opportunities is enourmous we decided to focus on some specific patterns. Initially our goal was be to be able to recognize basic missed vectorizations, which don't appear "in the wild" anymore, like the one shown in Listing 2. To do so more effectively, we started from the top of the code snippets collection from TSVC[6] that contains various vectorizable pieces of code of increasing complexity, ranging from trivial sum of vectors to more intricated and convoluted operations.

The idea is that TSVC should contain vectorization patterns that are commonly found in every program, and by successfully being able to tell whether one of those snippets was vectorized or not, we believe that our analysis could be expanded to bigger and more complicated programs.

Afterwards we focused on pattern-matching the access patterns found in the two examples described in Section 3.3, which is non-trivial work. For this we will be using a our own PinTool to extract the trace and our program `vecspot` to generate the dataflow graph and find missed vectorizations.

The following section will focus on analyzing, step-by-step how we approached some of the examples from the TSVC test suite, and the titles of the sections will refer to the specific functions contained in it, for an easier reference.

4.4.1 `s000` *(Linear Dependence Testing, Vectorizable).*

4.4.2 `s111` *(Linear Dependence Testing, Vectorizable).*

4.4.3 `s113` *(Linear Dependence Testing, Vectorizable).*

4.4.4 `s116` *(Linear Dependence Testing, Vectorizable).*

[6] https://github.com/UoB-HPC/TSVC_2/blob/master/src/tsvc.c

REFERENCES

[1] D. Callahan, J. Dongarra, and D. Levine. 1988. Vectorizing compilers: a test suite and results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing* (Orlando, Florida, USA) *(Supercomputing '88)*. IEEE Computer Society Press, Washington, DC, USA, 98–105.

[2] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All you need is superword-level parallelism: systematic control-flow vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3519939.3523701

[3] Jing Ge Feng, Ye Ping He, and Qiu Ming Tao. 2021. Evaluation of Compilers' Capability of Automatic Vectorization Based on Source Code Analysis. *Scientific Programming* 2021 (30 Nov 2021), 3264624. https://doi.org/10.1155/2021/3264624

[4] Saeed Maleki, Yaoqing Gao, María Garzarán, Tommy Wong, and David Padua. 2011. An Evaluation of Vectorizing Compilers. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, 372–382. https://doi.org/10.1109/PACT.2011.68

[5] Charith Mendis, Cambridge Yang, Yewen Pu, Dr.Saman Amarasinghe, and Michael Carbin. 2019. Compiler Auto-Vectorization with Imitation Learning. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/d1d5923fc822531bbfd9d87d4760914b-Paper.pdf

[6] Vasileios Porpodas and Pushkar Ratnalikar. 2021. PostSLP: Cross-Region Vectorization of Fully or Partially Vectorized Code. In *Languages and Compilers for Parallel Computing*, Santosh Pande and Vivek Sarkar (Eds.). Springer International Publishing, Cham, 15–31.