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 loops that could be vectorized but are not.

2 RELATED WORK

Autovectorization is *difficult*, compilers tend to miss many optimizations (as shown by Feng et al. [2]) , 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. [1]).

Compilers have been shown to widely miss out on vectorization opportunities. As an example, Maleki et al. [3] 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 [5]) or by applying Machine Learning to produce improved vectorization schemes (Mendis et al. [4]).

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.

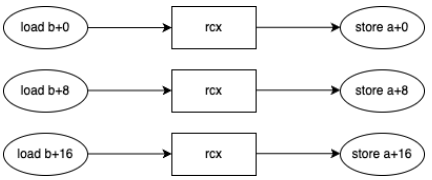

Fig. 1. Example of dataflow graph for `copy.c`

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 an approach using static analysis. When using dynamic analysis one needs to be careful not to falsely report code with complex control flow as non-optimized only considering the memory accesses observed by dynamic analysis.

```
#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. Example of XXX

Assuming 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 is trivial using the `addps` instruction. Therefore, we would flag this code as missing vectorization without considering the control flow operation (this example is trivially vectorizable using masked vector operations but one can consider complex control flow that can't be vectorized). To fix this issue we can either use

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

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 lost precision this results in.

All in all, we foresee that building the 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 Examples of missed vectorization

By searching on the different bug and issue trackers of the clang and gcc compiler 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. The following examples has been taken from the gcc bug tracker[2].

```c
#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. `example1.c`

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 as shown in Listing 6, whereas clang unrolls the inner loop and, as expected, makes extensive use of vector instructions[3].

```
ex1(signed char*, signed char, unsigned long const*, unsigned int):
        mov     r9d, ecx
        mov     rax, rdi
        mov     r10, rdx
        mov     edi, esi
        test    r9d, r9d
        je      .L1
        mov     rcx, rax
        xor     r8d, r8d
.L4:
        mov     rsi, QWORD PTR [r10+r8*8]
        xor     eax, eax
.L3:
        bt      rsi, rax
        setc    dl
        neg     edx
        and     edx, edi
        add     BYTE PTR [rcx+rax], dl
        add     rax, 1
        cmp     rax, 64
        jne     .L3
        add     r8, 1
        add     rcx, 64
        cmp     r9, r8
        jne     .L4
.L1:
        ret
```

[2] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96888
[3] https://godbolt.org/z/W8v4Pc463

Listing 6. `example1.c` compiled with `gcc -O3`

3.4 Approach XXX: change name

Our goal in the project is to be able to test for missed vectorization opportunities. As the possibility space of different kinds of missed vectorization opportunities is enourmous we will be focusing on some specific patterns. Initially our approach will be to be able to recognize basic missed vectorizations, which don't appear in the wild anymore, like the one shown in 2. Afterwards we will be focusing on pattern-matching the access patterns found in the two examples described in 3.3, which is non-trivial work. For this we will be using a combination of Pin's built in Dynamic Control-flow Graph Generation[4] and our own PinTool to extract relevant traces.

3.5 Test Inputs

To test our results, we will use an already existing autovectorization benchmark such as `TSVC`[5]. Other random program generators such `YARPGen`[6] could be used to generate more, less specific, test inputs.

4 IMPLEMENTATION AND EXPERIMENTIAL SETTINGS

To ease our work, and build a first prototype of our project, we focused on a specific version of a specific compiler, namely `gcc 13.2`.

REFERENCES

[1] 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

[2] 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

[3] 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

[4] 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

[5] 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.

[4] https://www.intel.com/content/www/us/en/developer/articles/technical/pintool-dcfg.html
[5] https://github.com/UoB-HPC/TSVC_2
[6] https://github.com/intel/yarpgen