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:

```c
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 on Figure 1.

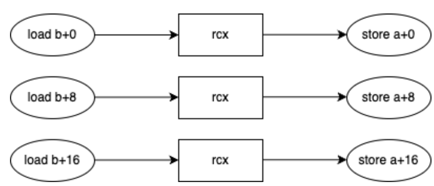

Fig. 1. Example of dataflow graph for `copy.c`

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. For finding vectorization opportunities with complex control-flow it is still necessary to use a hybrid approach in order to check for false-positives (or negatives?). Imagine the case of a masked add/sub operation on vectors. If running with the array c = 0 or 1, we will observe that this is a trivial vectorization, when, instead, it is more involved then optimizing it with a vaddps. XXX mettere codice e inserire questo sotto Given the trace of execution of a program (instructions and memory accesses), one can build a dataflow graph, on which a dataflow analysis can be performed to identify “parallel” computations and thus, vectorization opportunities.

However, 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

To find examples of missed vectorization opportunities we conducted a small research throughout various online resources, namely:

- The `llvm` forum (https://discourse.llvm.org/)
- The `llvm` issue tracker (https://github.com/llvm/llvm-project/issues)
- The gcc bug tracker (https://gcc.gnu.org/bugzilla/)

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

This research aided us into refining the focu sof this project by making us aware of real and existing challenges that widely used compiler face when vectorizing code. On top of that, it also helped us better understand which patterns are more prone to go "unvectorized". We hereby report two indicative examples of the collection of unvectorized pieces of code we compiled.

3.3.1 *Example 1.* https://godbolt.org/z/K6T3TG5Wb XXX spiegare differenza tra gcc e clang

3.3.2 *Example 2.* https://godbolt.org/z/Mh8Y9d41G

3.4 Test Inputs

To test our results, we will use an already existing autovectorization benchmark: `TSVC`[2]. Other random program generators such `YARPGen`[3] 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 11.4`.

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.

[2]https://github.com/UoB-HPC/TSVC_2

[3]https://github.com/intel/yarpgen