VEGEN: A Vectorizer Generator for SIMD and Beyond

Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe
Massachusetts Institute of Technology

1. Motivation

Vector instructions are ubiquitous in modern commodity microprocessors. Every few years, processor vendors introduce new instruction sets that either increase the data width of these instructions or introduce instructions with complex computation patterns. Initially, most of these instructions were single instruction multiple data (SIMD), but vendors have introduced more non-SIMD vector instructions to accelerate computation patterns occurring in performance-critical domains such as digital signal processing, image processing, and machine learning (e.g., Intel's VNNI extension). For example, the `addsubpd` instruction from the Intel SSE3 instruction set performs additions and subtractions in alternative vector lanes. This is an example of Single Instruction Multiple Operation Multiple Data (SIMOMD) instructions [1]. In addition to SIMOMD instructions, modern vector extensions also support non-SIMD vector instructions with complex operations that gather values from multiple lanes. Intel's VNNI instruction, `vpdpbusd`, for instance, multiplies *signed* and *unsigned* bytes, reduces every four elements, and accumulates the results into 32-bit vector lanes.

Supporting non-SIMD vector instructions requires significantly more effort than SIMD ones. For instance, the initial support LLVM commit that landed support for the `addsub` instruction family required three coordinated changes to LLVM: refactoring LLVM's SLP vectorizer to support alternating opcodes, changing the target-specific cost model to recognize a special case use of vector blending, and modifying the backend lowering logic to detect the pattern the special patterns generated by the SLP vectorizer.

2. Limitations of the State of the Art

Existing vectorizers cannot directly emit non-SIMD instructions because non-SIMD instructions violate a basic invariant assumed by vectorizers: Vector instructions perform simple, isomorphic, and vertical (element-wise) operations. Existing vectorization algorithms do not recognize the type of parallelism supported by non-SIMD instructions and, in the case that they do vectorize, emits excessive overhead shuffle instructions instead of directly exploiting more non-SIMD instructions. The existing methodology to targeting non-SIMD vector instructions thus relies primarily on writing backend instruction selection (peephole) rules to combine simpler vector instructions into more efficient non-SIMD ones.

There are two limitations to this methodology. First, it is burdensome and error-prone. Second, it prevents vectorizers from systematically consider non-SIMD instructions in their search space. The peephole rules targeting non-SIMD instructions assume that the input program is already in vector form and are therefore ineffective if the vectorizer does not emit vector instructions in the first place.

As a corollary to the second point, previous research on automatically generating instruction selection/peephole rules [2, 4] is insufficient. However robust and powerful these local optimizations might be, they alone do not extract implicit, fine-grained parallelism from a scalar program.

3. Key Insights & Overview

To support these instructions, we first broaden the type of parallelism supported by vectorizers. We term this Lane Level Parallelism (LLP). In contrast to superword-level parallelism (SLP), in which groups of isomorphic operations executing independently on individual vector lanes, LLP relaxes the constraints on the parallel operations in two aspects. (1) The operations need not be isomorphic. (2) The operations on each lane can use the values of arbitrary input lanes.

The key to VEGEN's vectorizer, which exploits LLP, comes from the observation that a vectorizer only requires two pieces of information to extract LLP:

- Which stream of scalar IR instructions can a given vector lane execute?
- Once multiple streams are *packed* together, which values are required as vector operands?

VEGEN uses a *vector instruction description language* to model the instruction semantics and generates vectorizer components that can answer these two types of queries. To target a new vector instruction set, VEGEN only requires the compiler writers to describe the semantics of each instruction in VEGEN's instruction description language. For the first type of query, VEGEN generates pattern matching code to recognize scalar operations performed on each vector lane. For the second type of query, VEGEN generates summaries of how the live-ins of such matched operations are bound to input vector lanes.

Within our framework, a target-independent vectorization heuristic uses the generated target-specific pattern matching code to recognize and pack operations supported by the given vector instructions and uses the lane-binding summary to sort out which vector values are required by the selected vector instructions. We discuss two LLP vectorization heuristic: (1) a compile-time efficient bottom-up dynamic programming algorithm and (2) a more aggressive heuristic based on beam search.

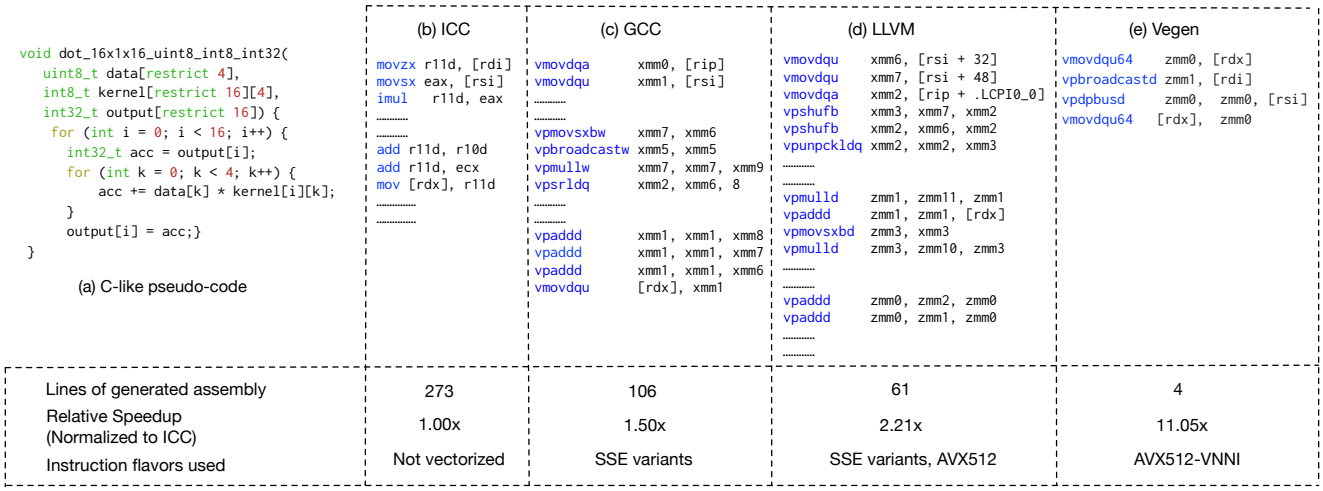

Figure 1: Dot kernel in TVM's 2D convolution computation (a) in C-like pseudo-code. Compiler generated assembly and statistics for (b) Intel's compiler ICC (c) GCC (d) LLVM and (e) the Vegan generated vectorizer

4. Main Artifacts

VEGEN takes Intel's Intrinsic Guide[1] as input and generates the target-specific components of its vectorizer. The full vectorizer is an LLVM pass that automatically use the instructions documented by the Intrinsic Guide.

5. Key Results and Contributions

On several performance-critical kernels, we show that VEGEN can automatically use non-SIMD instructions in non-trivial ways and, in some cases, rediscover the low-level algorithms used by expert programmers.

Figure 1 shows an example of the vector code emitted by VEGEN. The source program is a kernel used by TVM's [3] 2D convolutional layers.[2] Figure 1(a) shows the reference C code of their kernel. Figures 1(b)-(e) show the assembly output of ICC 19.0.1, GCC 10.2, LLVM 10.0, and the VEGEN generated vectorizer, respectively. All code generators were configured to target the AVX512-VNNI extension. VEGEN does not use any hand-coded instruction information in its implementation.

We make the following contributions in this paper:

- We introduce Lane Level Parallelism, which captures the type of parallelism implemented by both SIMD and non-SIMD vector instructions.
- We present a vectorizer generator that systematically use complex non-SIMD instructions, using only their documented semantics as input.
- We describe a code-generation framework that jointly performs vectorization and vector instruction selection while maintaining the modularity of traditional target-independent vectorizers designed for SIMD instructions.
- We show that on several performance-critical image and signal processing kernels from FFmpeg and x265, VEGEN is able to automatically use non-SIMD instructions and attain speedup up to 3.4× (compared to LLVM's vectorizer).

6. Why ASPLOS

We present a system that combines programming languages and computer architecture. Our system demonstrates the possibility of automating code generation support for non-SIMD instructions and systematically integrating these instructions into a vectorizer. As hardware becomes more complicated to accelerate compute-intensive domains, we believe, this is a valuable contribution to the ASPLOS community.

7. Citation for Most Influential Paper Award

This is the first paper that describes the design and implementation of an automatic code generator generator for non-SIMD vector instructions, which have important applications in performance-critical domains such as DSP, image processing, and machine learning. In addition to reducing the effort required to develop vectorizing compilers, VEGEN is the first system that systematically considers non-SIMD instructions during vectorization, while previous work approaches rely primarily on peephole rules. This work presented a simple design to decouple the target-dependent components from a target-independent vectorization algorithm while still exposing the relevant information to make vectorization effective.

References

[1] Leonardo Bachega, Siddhartha Chatterjee, Kenneth A Dockser, John A Gunnels, Manish Gupta, Fred G Gustavson, Christopher A Lapkowski, Gary K Liu, Mark P Mendell, Charles D Wait, et al. A high-performance simd floating point unit for bluegene/l: Architecture, compilation, and algorithm design. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 85–96. IEEE, 2004.

[2] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[2] We transcribed this reference implementation from the kernel's source comment, which contains the pseudocode.

Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, page 300–313, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[4] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *CoRR*, abs/1711.04422, 2017.