Custom High-Performance Vector Code Generation for Data-Specific Sparse Computations

Marcos Horro*
marcos.horro@udc.es
Universidade da Coruña, CITIC
A Coruña, Spain

Louis-Noël Pouchet
pouchet@colostate.edu
Colorado State University
Fort Collins, Colorado, USA

Gabriel Rodríguez
gabriel.rodriguez@udc.es
Universidade da Coruña, CITIC
A Coruña, Spain

Juan Touriño
juan.tourino@udc.es
Universidade da Coruña, CITIC
A Coruña, Spain

Abstract

Sparse computations, such as sparse matrix-dense vector multiplication, are notoriously hard to optimize due to their irregularity and memory-boundedness. Solutions to improve the performance of sparse computations have been proposed, ranging from hardware-based such as gather-scatter instructions, to software ones such as generalized and dedicated sparse formats, used together with specialized executor programs for different hardware targets. These sparse computations are often performed on read-only sparse structures: while the data themselves are variable, the sparsity structure itself does not change. Indeed, sparse formats such as CSR have a typically high cost to insert/remove nonzero elements in the representation. The typical use case is to not modify the sparsity during possibly repeated computations on the same sparse structure.

In this work, we exploit the possibility to generate a specialized executor program dedicated to the particular sparsity structure of an input matrix. It creates opportunities to remove indirection arrays and synthesize regular, vectorizable code for such computations. But, at the same time, it introduces challenges in code size and instruction generation, as well as efficient SIMD vectorization. We present novel techniques and extensive experimental results to efficiently generate SIMD vector code for data-specific sparse computations, and study the limits in terms of applicability and performance of our techniques compared to state-of-practice high-performance libraries like Intel MKL.

CCS Concepts

• **Software and its engineering → Source code generation**; • **Computing methodologies → Vector / streaming algorithms**.

---

*With AMD at the time of publication.

Keywords

1 Introduction

Sparse data structures, such as sparse matrices, are ubiquitous in modern computing to represent nonzero-valued regions within a dense coordinate system. This avoids storing useless zero values, and bypasses useless computations (e.g., multiplication by zero) [5, 35]. Sparse tensors, including sparse matrices, are widely used in physics simulation, graph analytics [45], or may occur after sparsification of a neural network weight matrix [4, 19].

A typical use case is to create a sparse representation from the set of nonzero coordinates in a data structure by inspection and compression into multiple arrays, as exemplified with the Compressed Sparse Row (CSR) format [35]. Such formats are not amenable to fast insertion/deletion of a nonzero element in the structure: one or more array reallocations and, more importantly, shifting of all the data, is typically required. Sparsity at the data structure level is often immutable, and only the values associated to the nonzero coordinates evolve before another phase of compression into, e.g., CSR is done again. A generic executor code, applicable on any sparse matrix of a given format, is typically deployed and optimized for a particular target [3, 5, 6, 10, 14, 18, 38, 44, 46]. However, this generality comes at the cost of using indirection arrays, i.e., arrays that encode the original sparsity structure and that are used to index the compressed structures, and creating challenges for efficient SIMD vectorization. The state of practice is typically to select a sparse format delivering good performance for the considered sparsity pattern [36], or more simply to rely on a vendor library such as Intel MKL [40].

In this work, we particularly focus on optimizing sparse-immutable data structures, exploiting *data-specific compilation*: we generate code that is specialized (unique) to one sparse structure. Augustine et al. [4] developed a system for automatically building sets of regular subcomputations by mining regularly strided subregions on the irregular data structure, i.e., on a sparse tensor.

```
for (int i = 0; i < N; ++i) {
    y[i] = 0.0;
    for(int j = row_ptr[i]; j < row_ptr[i + 1]; ++j)
        y[i] += A[j] * x[cols[j]];
}
```
Listing 1: Standard SpMV kernel using CSR format.

However, their work is limited to reconstructing loop-based codes for sparse structures, without further emphasis on generating specialized SIMD programs for these particular small-loop computation patterns. *Their approach trades off large indirect accesses for larger code size using direct accesses only, but does not provide solutions to generate specialized high-performance multi-core SIMD implementations for such reconstructed programs. We present the MACVETH system (Multi-Architectural C-VEcTorizer for HPC applications) to address this problem.* Section 2 details the performance challenges in optimizing such codes, and overviews MACVETH before details are presented in Section 3. MACVETH is a Clang-based compiler, which has been used to generate extensive experimental analysis in Sec. 4.

MACVETH implements a systematic search of functionally equivalent SIMD implementations of packing and reduction operations, typically at compiler installation time. It uses an SMT solver approach to reduce the search space size. We support the fusion of independent reductions on the same SIMD vector(s) to improve SIMD occupancy for small reductions, in a fully automated way. As extensively studied in Section 4, MACVETH consistently improves performance over Intel MKL (single-core and multi-core) for the 200+ sparse matrices we evaluated from the SuiteSparse repository [12], with diminishing returns with the sparse matrix size, indicating the limits of our current techniques. We make the following contributions:

- We introduce MACVETH, the Multi-Architectural C-VEcTorizer for HPC applications, for synthesis of high-performance SIMD implementations of regular strided-access reductions or map loops, as typically occurring in the reconstruction of sparse matrices in data-specific compilation [4];
- MACVETH is a Clang-based source-to-source compiler, with the ability to automatically discover functionally equivalent ASM implementations, profiling them to build machine-specific recipes for the compiler to use for SIMDization. It embeds several problem-specific optimization strategies, such as the fusion of independent reductions to improve SIMD vector occupancy;
- We present extensive experimental results on 200+ sparse matrices of less than 20M nonzeros, in single-core and multi-core for two modern processors (Intel Alder Lake and AMD Zen3) demonstrating consistently better performance obtained with MACVETH versus Intel MKL.

2 Data-Specific Compilation of Sparse Structures

In this work we target the optimization of sparse (tensor) computations, exemplified with the Sparse Matrix - dense Vector product (SpMV) kernel using the CSR format shown in Listing 1.

In typical practice, a discrete representation of the set of stored values (i.e., the matrix sparsity structure) is assumed to be known a priori. Then, a compressed sparse representation can be built,

```
void kernel_spmv_fragment_0(float *__restrict A,
                             float *__restrict x,
                             float *__restrict y) {
    register int i0;
    for (i0 = 0; i0 <= 1; ++i0)
        y[1] += A[i0] * x[i0];
    for (i0 = 0; i0 <= 2; ++i0)
        y[2] += A[i0 + 2] * x[i0];
    for (i0 = 0; i0 <= 1; ++i0)
        y[3] += A[i0 + 5] * x[i0 + 1];
    for (i0 = 0; i0 <= 1; ++i0)
        y[4] += A[i0 + 7] * x[i0 + 1];
    y[5] += A[9] * x[1];
    for (i0 = 0; i0 <= 1; ++i0)
        y[5] += A[i0 + 10] * x[2]; }
```
Listing 2: Code generated for matrix JGD_Kocay/Trec5

using a variety of formats trading off memory footprint for the compressed structure versus ease to produce high-performance code scanning the nonzero elements of the structure. We note the cost of modifying the sparsity is typically prohibitive for most sparse formats (e.g., CSR, COO, etc.) as inserting or removing a nonzero element implies growing/reducing, and then shifting, the arrays used to index elements. For example, adding one nonzero element to a CSR representation requires updating A, row_ptr, and cols. Therefore many practical use cases repeat computations over the same sparsity pattern, only modifying the fields (i.e., the data values in the arrays) which are computed on. Generic executors, as in Listing 1, present the advantage to be sparsity-independent (the code is the same for any sparse matrix) and therefore generic [5, 6, 10, 18, 36, 46], but at the cost of using indirection arrays, limiting the performance.

2.1 Reconstruction by Codelets

In order to avoid the use of indirection arrays, Augustine et al. [4] developed a system for automatically building sets of regular subcomputations by mining regularly strided subregions on the irregular data structure, i.e., on a sparse tensor. This approach generates *data-specific* code for each input sparse structure, such as in those included in the SuiteSparse collection [12]. An example of the output of this system for the input sparse matrix JGD_Kocay/Trec5 is depicted in Listing 2. At compile-time, the nonzero coordinates are inspected and a code scanning the exact same coordinates is produced, but without any indirection array. In essence, the nonzero coordinates stored in row_ptr and cols are replaced by their actual values in the code generated, making it specialized to this specific sparsity pattern.

Listing 2 depicts a very simple and intuitive example of code fragment that is reconstructed using rectangular template shapes being mined over the full set of nonzero coordinates [4]. This set of nonzero coordinates (i, j) in the 2D sparse matrix leading to such code is, for the first loop, $((1, 0); (1, 1))$, then $((2, 0); (2, 1); (2, 2))$ for the second loop, etc. Taking into account the A data vector, where non-zero *values* are typically stored in contiguous fashion, is required to emit correct code manipulating the CSR representation. 3D codelets are mined for, for the (i, j, A_pos) coordinates. Precisely in the example above, we have $(1, 0, 0); (1, 1, 1)$ for the first loop, $(2, 0, 2); (2, 1, 3); (2, 2, 4)$ for the second, etc.

With these coordinates known at compile-time, the above code is produced by mining for the existence of (hyper-)rectangles containing regularly strided coordinates (i.e., the values indexing y, x, and A above) that can be expressed as simple affine functions of loop iterators [4]. Note that more complex loop shapes may be used to further compress sparse coordinates by using a geometric approach to build a $\mathcal{Z}$-polyhedron that fits a given set of points, instead of mining for the existence of a given shape (eTRE) [4]. In this work, we focus on codelet-based reconstruction, which appears to provide a good trade-off between code size and eventual performance [4].

2.2 Performance Trade-offs

The simple loop-based codes above present a difficult performance trade-off to navigate: on the one hand, more specialized code, with regular loops and no indirection arrays, should be amenable to high-performance SIMD implementation. On the other hand, the program remains entirely functionally equivalent to Listing 1 if it is invoked on the JGD_Kocay/Trec5 sparse matrix, up to a reordering of the iterations.

In general, even after reordering, it does perform non-consecutive (scattered) memory accesses, especially along the dense vector x, following the sparsity pattern of the input matrix. *Furthermore, generated loops typically have a very small trip count, insufficient to pack a full SIMD vector, and contain numerous short reductions.*

Augustine et al. noted several sources of ineffectiveness of these data-specific codes. First, and most importantly, *code size explosion, as the sparsity information is now encoded in the form of specialized loop nests.* **This triggers performance issues related to instruction cache prefetching (or lack thereof)**: as code is typically not reused, for good performance one shall implement a form of software prefetching for the code itself [4]. They also relied on existing auto-vectorizers for the generated code, which limits performance due to generic cost models and the small size of the loops to optimize. Compilers' auto-vectorizers synthesize machine-specific assembly code exploiting SIMD units of the target processor. Most of these techniques are conservative and are only applied if certain patterns are found in the code, and if a cost model has assessed their profitability. For instance, GNU GCC and Clang/LLVM use both Loop-Level Vectorization (LLV) and Superword-Level Parallelism (SLP) [17, 27, 29].

When accessing scattered (non-consecutive) memory addresses, gather is a convenient but complex x86 macro-instruction, introduced in AVX2 for loading random data points given a starting address and a set of indices. This instruction has been reported to deliver variable latencies [1], depending on the source and destination operands. Hofmann et al. [20] demonstrate the performance variability of the gather instruction for the Intel Knights Corner and Intel Haswell architectures, depending on the number of elements fetched by the gather instruction from a cache line. A single gather instruction may be decoded into many micro-operations, a number that varies depending on the architecture, e.g., in modern AMD architectures, such as Zen2 and Zen3, gather may be decoded into more than 30 micro-operations[1], depending on the vector width. Implicitly, this controls the ASM code size (a single gather) while enabling a complex computation (numerous executed μ-operations).

[1]According to values reported in https://uops.info/table.html ([1, 15]).

As such, gather *can be a good idiom for code-size-aware applications, but it may not be the best solution in terms of performance when it comes to throughput.* Actually, even though a gather instruction can save machine code bytes for the L1 instruction cache, it could take up more space in the μop-cache (or L0 cache) than other solutions due to its decoding phase. The μop-cache is a specialized cache for storing pre-decoded micro-operations [37], improving the performance in the decoding phase. It is present in modern architectures and it has a significant impact on performance [9], as it allows the decoder to idle when reusing micro-operations.

In this work, we tackle the problem of automatically generating machine-specific SIMD implementations for loop-based codes coming from the reconstruction of sparse structures as in Listing 2. We specifically focus on codes reconstructed for matrices in the SuiteSparse collection [12] and deliver a system optimized for SpMV-style computations, generating multi-core (OpenMP) and AVX2/SSE SIMD implementations of such programs. *Combining Augustine et al.'s long-distance rescheduling approach by codelet mining with the specialized short-distance SIMD optimizer we introduce in this work leads to a complete, hierarchical scheduling and packing approach to generate efficient SIMD programs for sparse-immutable computations.*

The optimized programs we generate systematically outperform Augustine et al.'s approach [4], as well as Intel MKL and a reference CSR implementation, as extensively studied in Section 4. In particular, **we present a fully automated synthesizer of ASM-based "software emulation" of arbitrary gather instructions, with the aim to improve performance over gather for random vector packing operations**. By profiling a variety of alternate implementations for the same type of access pattern once at installation time, machine-specific performance profiles are exploited. They are deployed in a novel Clang-based packing and SIMD vector code generation tool for reduction codelets, that is, for small loops computing reductions (e.g., loops in Listing 2).

Other related work Little prior work focused on exploiting the sparsity information to emit data-specific codes. EGGS implements sparsity-specific code generation [39] but does not exploit loop-based compression of the sparse coordinates [4]. The TACO compiler [11, 23] generates format/problem-specific efficient implementations of sparse tensor operations, which can also lead to very large binaries, even without data-specific compilation. Sparse formats and associated executors have been proposed with a focus on SIMD potential [14, 25, 38], including more recent work specifically targeting better utilization of SIMD units [7, 8, 26, 43]. However, to the best of our knowledge none exploit the actual sparsity information to generate specialized and simplified codes at compile-time, nor have the ability to substitute a gather instruction by higher-performing operations for a particular memory load pattern as we do in the present work.

Numerous prior works on automatic SIMD vectorization are directly applicable on such reconstructed programs, due to the simplicity of the loops generated. SLP vectorization may be applied (e.g., after unrolling) [30, 31] and several techniques for auto-vectorization of strided accesses have been developed [28, 34], including problem-specific ones [24]. To the best of our knowledge, while these techniques are applicable off-the-shelf to the

highly regular codes we generate, none is implementing the random vector packing strategies we present in Section 3, nor fusion of reductions for better SIMD occupancy in such an automated system as MACVETH. The Spiral system illustrates very well the ability to mine for effective machine-specific SIMD implementations [16, 24, 33] albeit via an algebraic system to search for equivalent implementations. In contrast, we develop data packing SIMD recipes that are tuned to the non-regular strided accesses in reconstructed loops, by first automatically characterizing their performance profile on the target machine, avoiding the limitations of static cost models.

2.3 Overview of MACVETH

In this work we take an aggressive approach to pattern-specific and data-specific auto-vectorization by developing MACVETH. This Clang-based source-to-source compilation framework targets the automatic vectorization of code regions (delimited by pragmas).

We implement vectorization using a SIMD-Intrinsics-style approach, to facilitate portability to a variety of concrete SIMD ISAs. We develop platform-aware cost-driven algorithms to efficiently pack arbitrary operands and operations into SIMD vectors. For this purpose, *we have also developed MRKVS (Mega-Random Kernel Vector SMT), a tool for generating candidate combinations of instructions to efficiently pack random elements into vector registers given a concrete ISA and a subset of instructions to consider.* Equipped with this model, *MACVETH supports vector packing across multiple distinct loop nests to maximize vector occupancy, in particular when loops have a very small trip count,* targeting operations such as reductions. These codes may typically be found in sparse tensor computations.

MRKVS is our proposed SMT-based model and system for generating combinations of instructions given an ISA to gather and pack random memory positions within a vector register. *The goal is to emulate the behavior of a gather instruction, by combining other available ASM instructions to achieve the same functionality, eventually leading to performance trade-offs such as (binary) code size versus performance.* Indeed, gather may be replaced by several ASM instructions, growing the binary size.

We also develop a platform-specific cost model derived from the candidates generated by the MRKVS system. This model is built by accurately micro-benchmarking all these candidates for each possible case using the MARTA framework [22], an automated tool for micro-benchmarking and building cost models from data, and selecting the most promising and profitable candidate for each platform. This profiling phase is reused across the compilation of different sparse structures on the same machine, and is typically done once at installation time.

Our implementation of MACVETH works as a Clang AST-based source-to-source compiler for vectorizing codes reconstructed from sparse structures. *This compiler is able to vectorize multiple reductions within the same vector register, and to fuse independent reductions using the same vector operations, and even across multiple vectors.* This solution also includes the platform-aware random packing combinations described above, for efficiently packing random operands in the same vector.

The only input to the system is a concrete C/C++ file with marked regions to be considered for vectorization. The output is a SIMD version of that code, if the cost model predicts its profitability; otherwise it just emits scalar code. *For simplicity, we focus in this paper on x86 architectures with AVX2 only, but the same approach can be applied to other architectures and ISAs*: we develop machine-independent techniques to produce automatically machine-specific codes.

3 SIMD Code Synthesis

We now outline our approach for SIMD synthesis of strided codelets, that is fully implemented in MACVETH. We first generate a collection of semantically-equivalent micro-programs that pack scalar operands stored at arbitrary addresses in memory into a contiguous SSE / AVX vector, for all possible data packing situations (e.g., same or different cache lines, etc.).

The performance of these implementations is then characterized, by actual measurement on the host machine, as well as using cost models such as LLVM-MCA, to find the best performing versions. This forms a set of *SIMD packing code templates* for MACVETH, to be used each time a compilation using MACVETH is to be performed. For the actual compilation of programs to SIMD with MACVETH, we operate on a DAG-based representation of the computation, obtained after fully unrolling loops in the code region to vectorize. MACVETH packs operands and operations into vector form, tiling this DAG by instantiating the proper templates and replacing scalar code by these instantiations in the input program. Similarly SIMD operations are also generated, following the SIMD operands being packed. Additional optimizations for performance, such as fusing two independent and short reductions on the same SIMD vector, are implemented by MACVETH to improve vector occupancy and eventual performance.

3.1 Generation of Data Packing Recipes

A major performance problem to address is finding the most efficient (wall-clock time) machine-specific code to pack randomly placed operands in memory into a single SIMD vector. This random vector packing phase can be implemented with a gather instruction. Our objective is first to create *specialized* code that emulates the gather semantics, for every possible situation (i.e., locations of the scalars to pack) it may be called on. We do so because maximal performance may be achieved with very different ASM instructions *depending on where the operands to pack are placed in memory*. As operands may be mostly randomly placed in memory after codelet reconstruction, we must implement a high-performance solution for each possible packing scheme.

Intuitively, we can define each of the load instructions as a function that, for a given virtual address p, returns a set of contiguous positions in memory using a little-endian format: $f(p) := \{v_{n-1} = f(p[n-1]), ..., v_0 = f(p[0])\}$. For instance, _mm_loadu_ps(p) and _mm_loadu_ss(p) are represented as:

$$loadu\_ps(p) := \{v_3 = p[3], v_2 = p[2], v_1 = p[1], v_0 = p[0]\}$$
$$load\_ss(p) := \{v_3 = \emptyset, v_2 = \emptyset, v_1 = \emptyset, v_0 = p[0]\}$$

Representing swizzle instructions in our approach can be done, e.g., for the _mm_shuffle_ps(a,b,m) instruction:

$$shuffle\_ps(a, b, m) := \{v_3 = f(b, m[7:6]), v_2 = f(b, m[5:4]),$$
$$v_1 = f(a, m[3:2]), v_0 = f(a, m[1:0])\}$$
$$where \; f(src, c) := src[31 + 32 * c : 32 * c]$$

In this case, the output of this instruction depends on the value of the mask m, so we must compute all possible mask values (256 different values since the mask width is 8 bits), e.g.,

$$shuf\_000\_ps(a, b) := \{v_3 = b[0], v_2 = b[0], v_1 = a[0], v_0 = a[0]\}$$
$$shuf\_001\_ps(a, b) := \{v_3 = b[0], v_2 = b[0], v_1 = a[0], v_0 = a[1]\}$$
$$\dots$$
$$shuf\_255\_ps(a, b) := \{v_3 = b[3], v_2 = b[3], v_1 = a[3], v_0 = a[3]\}$$

A packing strategy that needs to be synthesized can be expressed in a similar notation, e.g. `pack_001(a[0],a[2],b[0],b[1]) := {v_3 = a[0], v_2 = a[2], v_1 = b[0], v_0 = b[1]}` making the problem amenable to constraint solving. We look for a combination of instructions in the set $\mathcal{I}$ of instructions considered, such that their composition matches the semantics of the desired Packing Class, i.e., of the `pack_xxx` prototype.

Note that the space created by generating all possible combinations of these instructions would grow exponentially and quickly become intractable. Exploring all combinations of memory addresses and their possible packing candidates for getting the performance-optimal recipe is an NP-complete problem, so purely brute force will not scale. On the other hand, it is possible to tackle this issue by carefully defining the exploration space to traverse, and applying heuristics to prune the set of candidates to combine in each step as shown below.

3.2 MRKVS: Mega-Random Kernel Vector SMT

One of the issues when enumerating all the candidates is the combinatorial explosion of the number of variants of a single instruction according to its masks or control value. So instead of generating and testing all these possible combinations, a smarter strategy would be to check whether there is any value that for a combination of instructions is able to meet the packing conditions, i.e., the packing of memory points into a vector register in a certain order.

Wegner developed x86-sat [42], a system for building an auto-generated formal model of x86 Intrinsics by interpreting the pseudo-code in the official documentation, and transforming it into a valid model for the Z3 SMT solver [13]. This tool is mainly written in Python, and it can help assess the equivalence of two different ways of permuting values, i.e., to find equivalent implementations, or to find the values of some variables in a formula.

x86-sat works as follows. A set of assertions or conditions describing the behavior of each instruction according to the documentation are added to the solver using the Z3 library (a custom parser is used to automate this process [41]). Next, the check function tests if those conditions can be satisfied for those variables or instructions and, in that case, the system also returns a model containing the values required. This system avoids testing all different control value combinations for a concrete instruction. The system is also interesting for performing sanity checks given a set of instructions

---

Algorithm 1: High-level approach of the MRKVS system.

Input: Instructions $\mathcal{I}$, PackClass S_l, int $max\_candidates$
Result: Set of Candidates

1   candidates = {};
2   max_ins = compute_max_ins(S_l);
3   load_ins = prune_load_instructions(S_l, $\mathcal{I}$);
4   **for** *load in load_ins* **do**
5   **if** *check(load, S_l)* **then**
       // Base case: only a load required,
       // no need to explore
6   candidates.append(load);
7   continue;
8   **if** *new_candidate = recursive_search(load, $\mathcal{I}$, S_l, max_ins)* **then**
9   candidates.append(new_candidate);
10   **if** *size(candidates) >= max_candidates* **then**
11   break;
12   **end**
13   **return** candidates;

---

and the conditions to be satisfied, or even for finding bugs in the documentation. In addition, this system can be easily extended with any other desirable instruction by just using the same syntax as in the Intel Intrinsics documentation.

MRKVS searches the solution space as described in Algorithm 1. Our approach follows a depth-first fashion, with a limited number of levels, and where the level is determined by the number of chained instructions used. This algorithm is a modification of the naive brute force approach, where the explosion of new combinations is minimized by applying heuristics to prune the set of new candidates, i.e., new instructions to consider, and the use of the SMT system to parameterize and check the satisfiability of the chain of instructions. The system has also a variable stop condition when the number of candidates found has reached, at least, $max\_candidates$.

We note that candidate instructions are pruned depending on their type and number in each recursive step, in order to reduce the node explosion as we create new levels in the exploration space. These pruning techniques are ad hoc and dependent on the set of considered ASM instructions, which in our case is a small subset of the AVX2 ISA. Some of these techniques involve limiting the maximum number of instructions of a type to consider in a combination, avoiding the appearance of costly instructions (such as masked loads or blends) more than once for each candidate, or avoiding the use of load instructions to load only one element more than twice.

Random vector packing templates format: In order to make these candidates portable to any memory address, we developed a template-based format to capture their semantics and be input agnostic. Any system can then fill any of the candidates generated by MRKVS with the desired input memory addresses or vector registers for packing the operands. These templates are in MACVETH Random Template format (`.mrt`). MACVETH leverages these templates to pack random operands filling the input values with the corresponding memory addresses from the code.

```
;; Candidate 0
vmovss     xmm2, DWORD PTR [r12 + 0x40]
vinsertps  xmm1, xmm2, DWORD PTR [rcx], 0x14
vinsertps  xmm0, xmm1, DWORD PTR [rdx], 0x68

;; Candidate 1
vblendps   xmm1, xmm3, XMMWORD PTR [rsi], 0x0b
vinsertps  xmm2, xmm1,  DWORD PTR [rcx], 0x18
vinsertps  xmm0, xmm2,  DWORD PTR [rdx], 0x28
```

Listing 3: Example of generated assembly code

```
__vop2 = _mm256_loadu_ps(&z[0]);
__vop0 = _mm256_hadd_ps(__vop0, __vop2);
__mv_lo128 = _mm256_castps256_ps128(__vop0);
__mv_hi128 = _mm256_extractf128_ps(__vop0, 0x1);
__mv_lo128 = _mm_add_ps(__mv_lo128, __mv_hi128);
__mv_hi128 = _mm_shuffle_ps(__mv_lo128, __mv_lo128,
                            0b00110001);
__mv_lo128 = _mm_add_ps(__mv_lo128, __mv_hi128);
tmp0 = tmp0 + __mv_lo128[0];
tmp1 = tmp1 + __mv_lo128[2];
```

Listing 4: Example of synthesis in MACVETH for the fusion of two independent reductions of 8 32-bit elements each, in two different vectors

3.3 Search Space and its Evaluation

The next step after having built a set of candidate implementations is to evaluate their performance on the target machine. This stage is typically performed once on the host machine, at installation time of MACVETH (or when the system has been modified, such as when using a different compiler/version).

In this work, we use MRKVS to generate random vector packing formulas for AVX2 with floats as data type. However the approach can be easily extended to other ISAs and data types. The output of the SMT-based system is a set of candidate implementations for each equivalent functionality (i.e., identical layout in the destination register from the input registers). Each equivalence class, referred to as a "packing class", is defined here by the contiguity of the memory addresses to pack, vector width and data type.

For example, the compilation of two candidate implementations for packing three non-contiguous elements is shown in Listing 3. In the Zen3 processor we experimented with, both choices retire the same number of micro-operations, and the cycles consumed are on average the same. Regarding a Cascade Lake processor we experimented with, both candidates also have identical performance in terms of cycles, but the number of micro-operations retired is lower for the first candidate. These are the two metrics considered for building our cost model: in the first place the number of execution cycles or cycles consumed, and in case of identical performance (within an error margin due to measurement errors), the number of micro-operations retired. Results obtained in LLVM-MCA using MARTA for these candidates confirm the values reported by our empirical measurements. The recently released uiCA tool [2] also reports better reciprocal throughput (i.e., cycles per instruction) for the first candidate.

Once we have chosen the best candidates for each platform and for each packing class, in order to assess their quality, we compare their performance with that of the equivalent gather instruction. According to our measurements, for Intel Cascade Lake, most of the candidates proposed by our automated system outperform `gather` in terms of number of execution cycles. However, for less than 15% of the cases, `gather` outperforms by 10-15% the latency of our approach. In contrast, according to LLVM-MCA all candidates outperform the gather instruction, showing the need for in-situ measurements. Our cost model is driven by these measurements and will therefore use the gather instruction for those equivalence classes where there is no speedup from the candidates generated, to favor also reducing the final binary size.

3.4 Fusion of Independent Reductions

In the middle-end of MACVETH, the packing cost model tries to maximize the vector occupancy for reductions. MACVETH considers two forms of fusing independent reductions: using the same vector register (intra-register), and using multiple vector registers (inter-register). For the first case, the back-end just performs a partial reduction on the register to be reduced. In the second case, the compiler uses the same operations to simultaneously compute both independent reductions. This approach has a limitation: the number of values in each independent reduction must be the same, and the values must be placed contiguously. This is why the packing cost model must pack, typically, a multiple of 2 reductions together. Following the example, packing 5 reductions on `tmp0` and 3 on `tmp1` cannot be done with the approach proposed here. We have developed fully automated algorithms to detect opportunities (and profitability of) packing independent reductions together. For the sake of space saving we limit to displaying Listing 4 to illustrate the code we can generate for fusing such reductions.[2]

3.5 SIMD Code Generation

MACVETH relies on the Clang AST for parsing the input code. Instead of lowering this abstraction to LLVM IR, our compiler rewrites the original code using SIMD directives in an Intrinsics style whenever profitable, thanks to the Clang's LibTooling library, which supports rewriting the original source code. The high-level picture of the system's architecture is depicted in Figure 1. We logically divide our source-to-source compiler architecture into front-end, middle-end and back-end.

MACVETH uses different abstraction levels and IRs in order to facilitate the vectorization process. The input AST is obtained with Clang. From there, MACVETH generates three-address code in SSA form, which itself facilitates the creation of a Directed Acyclic Graph (DAG) for the computation, after unrolling loops as needed. Typically MACVETH operates on a window of the AST at a time, to limit the size of the DAG manipulated, especially for large programs.

This DAG structure is suitable for finding patterns in the code such as reductions and long-distance load sharing opportunities. The operations and operands in this DAG are packed when possible to generate vector operations, based on the measured profitability of the corresponding vector packing recipes. These are generated in the SIMD back-end. Then, using the Clang framework, the front-end rewrites the original source code synthesizing the SIMD code generated in the back-end.

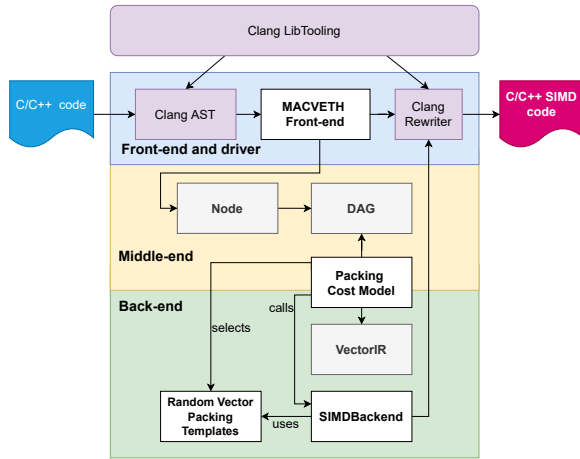[2]Full details and all algorithms are available in Chapter 5 of [21]

Figure 1: High-level diagram of MACVETH's architecture showing the different IRs used by the system.

4 Experimental Results

MACVETH was used to vectorize the data-specific programs generated from the sparse matrix – vector multiplication of the 200 matrices by Augustine et al. [4][3]. These are selected from the full SuiteSparse [12] by applying a sieve process in which matrices are classified according to the decile they belong to in terms of matrix size and the percentage of points that are issued as micro-codelets. This sieve yields 100 categories. Inside each category, k-means clustering is used to select representative matrices. The number of representatives per cluster is selected so that the probability density of the sample matches the original one.

Experiments were executed on an Intel Core i9 12900K (Alder Lake) with 128 GiB of RAM memory. All runs were repeated 10 times, reporting the best performance achieved for each experiment after removing outliers, identified as those measurements that deviate more than 3σ from the mean. The CPU frequency was fixed at the base of 3.2 GHz to prevent thermal constraints affecting experimental variability. The data and code segments are stored into 2 MiB hugepages. The data-specific codes implementing SpMV for each selected matrix were synthesized using the DSCG tool that applies the shape-based mining approach by Augustine et al. [4] (763 different hyper-rectangular shapes with increasingly large sizes and strides). Codes were compiled with GCC v11.2.0 with `-Ofast -march=alderlake`. All vectorization flags were enabled when compiling baseline codes. The PolyBench [32] and MARTA testing harnesses were used for performance measurements. Prefetching of the text segment was included in the linking process [4]. Other CPUs and compilers were used for sensitivity studies, detailed in Section 4.7. Using these same basic setup and running on the 16 logical P-cores of this machine, Intel Linpack reports an average raw performance of 364.2 GFLOPS, with a peak of 499.6 GFLOPS. The memory-bound Intel HPCG benchmark reports a raw performance for SpMV computations of 5.5 GFLOPS, which can be used to contextualize our results.

---

[3]The full list of matrices used in these experiments can be double-blindly consulted at https://pastebin.com/kqMABFUQ.

```
#pragma omp for private(j) nowait
for(i = 0; i < N; ++i) {
  y[i] = 0.0;
  for(j = row_ptr[i]; j < row_ptr[i+1]; ++j)
    y[i] += A[j] * x[cols[j]];
}
```

Listing 5: CSR executor employed as a baseline throughout the experimental section. The start of the `omp parallel` region is left out of the timed scope, to avoid measuring thread-creation overheads. OpenMP is disabled for single-threaded experiments.
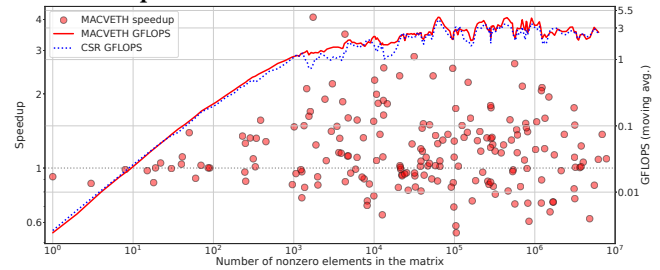

Figure 2: Speedup of TSC-based MACVETH version w.r.t. CSR (left) and raw performance (right). Log axes.

4.1 Using TSC as a Predictor of Performance

We first explore the performance of the 12900K processor on 2568 recipes generated for packing up to 8 single-precision floating-point elements into a 256-bit vector register. We micro-benchmark the performance of each packing recipe, selecting for each packing class the one which runs the fastest, as measured by the Time Stamp Counter (TSC), a 64-bit register that counts the number of CPU cycles since reset. We select 256 packing recipes (corresponding to the 256 packing classes in a 256-bit vector). MACVETH is then used to vectorize the data-specific SpMV codes for the 200 matrices in the experimental set using these recipes. In all cases, the MRKVS-generated recipes are faster, according to the TSC register values, than using the generic AVX2 gather instruction. The codes are executed under cold cache conditions: each SpMV operation is run once for each performance counter measured, to avoid multiplexing effects. The cache is flushed between executions using `clflush` instructions, taking special care to flush the entire text segment to avoid unfair advantages for DSCG and MACVETH codes.

Performance results versus the irregular CSR executor shown in Listing 5 are detailed in Figure 2. The geometric mean of the speedups with respect to the CSR and DSCG versions are 1.16 and 1.14, respectively. We find that the speedups obtained by DSCG with respect to the CSR executor are significantly smaller when compared to the ones reported by Augustine et al. [4]. The reason is that newer compilers feature a much more efficient optimization of the CSR code. While ICC19 vectorized 44% of the total floating-point operations in these codes, all of them employing 128-bit vector operations only, the more recent GCC v11.2.0 vectorizes 79% of them, and 40% of the total operations are executed using 256-bit vector operations, resulting in a 15% reduction of the total number of executed instructions.

We also observe that the SLP vectorizer in GCC is not capable of vectorizing DSCG codes to the same degree as the CSR executor.

Only 49% of the total FLOPs are vectorized, and only 14% of the total correspond to 256-bit vector operations. This figure is similar to the one obtained by the older compilers, and appears to suggest that no relevant improvement in free-code SLP vectorization has been achieved by more recent compilers.

In contrast, the custom optimization employed by MACVETH **vectorizes 93% of all FLOPs**, and 82% of the total is issued using 256-bit vector operations. MACVETH achieves a **3.5x reduction on the number of executed instructions** with respect to CSR, which represents an additional 1.26x reduction on top of the 2.8x achieved by data-specific codes compiled by GCC. Note that, in the CSR version, branch instructions alone account for 12.3% of the total instruction count. This figure does not include arithmetic instructions involved in branch computations. While the reduction in DSCG codes comes mostly from the elimination of control-flow instructions, the additional reduction provided by MACVETH is solely due to the improved vectorization.

4.2 Using Instruction Count as a Predictor of Performance

In order to assess whether the performance obtained by the TSC-based approach can be improved upon, we carefully analyze the results of the matrices for which MACVETH performs poorly. One such matrix is `GHS_psdef/apache2`, one of the largest ones in the experimental set, but which includes a high percentage of codelets: 99% of its 9.6 MFLOPs are captured by the DSCG using affine loops, with an average 9.14 FLOPs per loop. The raw performance in this case is 0.8 GFLOPS, which corresponds to 1.6x and 1.3x slowdowns for the MACVETH code with respect to the CSR and DSCG codes, respectively. The reasons for this slowdown are significant increases in the number of L3 misses, which are mostly due to code blocks. In fact, while the size of the CSR code is negligible, the DSCG and MACVETH codes take up 84 and 110 MB, respectively. Note how the increase in code size corresponds precisely with the slowdown obtained by the MACVETH version with respect to the DSCG version. We observe this effect consistently across the experimental set, and in fact the Pearson correlation coefficient between speedup and reduction in code sizes for the DSCG and MACVETH versions is $R^2 = 0.91$.

We revise the methodology for generating MACVETH recipes. Instead of considering the TSC cycles as the driver of performance, we choose the best recipe for each packing class as the one which contains the fewest number of instructions[4]. Furthermore, in this second version of the packing recipes we consider the performance of the native AVX2 gather instruction as well, selecting it when it results in fewer micro-operations than the recipes generated by MRKVS. In total, 129 out of the 255 packing classes are issued as MRKVS recipes, while the remaining 126 packing classes are issued using `_mm256_i32gather_ps`. We stress that these results have been generated by micro-benchmarking the Alder Lake architecture, and will not generalize to others.

Figure 3 shows the speedups obtained for this new MACVETH version. The new geometric mean speedup relative to the CSR and DSCG versions is 1.5x, with a geometric mean reduction in the number of executed instructions of 4.7x and 1.8x, respectively. With

---

[4]As measured by the `INSTRUCTIONS_RETIRED` performance counter in Alder Lake.

```
sparse_matrix_t M;
mkl_sparse_s_create_csr(&M, ...); // Inspector phase

polybench_start_instruments; // Starts timed scope
mkl_sparse_s_mv(...); // Executor phase
polybench_stop_instruments; // Stops timed scope
```
Listing 6: MKL executor.

respect to the DSCG version, the number of L2 misses attributable to code blocks is reduced by 3.6x, and the total L3 misses by 1.7x. For the particular case of the `GHS_psdef/apache2` matrix, the new instruction count-based version achieves now a raw performance of 3.0 GFLOPS, a 3.8x speedup relative to the TSC-based version, which represents a speedup of 1.2x and 1.5x with respect to the CSR and DSCG versions, respectively.

MACVETH achieves good results even for matrices with virtually no operations recognized as codelets. E.g., `Mittelmann/fome13` features an SpMV kernel with 570K FLOPs, out of which 99.8% are written as scalar operations in DSCG codes. MACVETH manages to execute 86% of them as vector operations (versus 78% by the CSR executor and 18% by the GCC-compiled DSCG code), achieving a reduction in the number of executed instructions of 4.5x and 1.5x with respect to the CSR and DSCG versions, respectively, and a speedup of 2.5x and 1.6x. The raw performance achieved by MACVETH increases from 1.67 GFLOPS using TSC-based packing recipes to 2.40 GFLOPS using instruction count-based recipes. This exemplifies how the performance improvements obtained by MACVETH are **not dependent on the regularity of the sparsity patterns** exhibited by the input matrix.

Note that, when using instruction count-based packing recipes, MACVETH generates vector code for exactly the same operations as in Section 4.1, but using packing instructions that reduces executable size. We compared the results achieved by the instruction count-based version with the ones obtained by a version which only uses `_mm256_i32gather_ps` for data packing. The version employing MRKVS recipes achieves a 1.12x geometric mean speedup with respect to the gather-only version, with a 1.18x reduction in the number of executed instructions.

4.2.1 Hot cache Finally, we analyze the performance obtained by these codes under hot cache conditions. For this, we execute the same SpMV operation 100 times, without flushing the cache after each repetition. Note that these are the usual experimental conditions when computing tensor operations on batches of data, as in, e.g., the inference of neural networks. Likewise, these are the experimental conditions for the Intel HPCG benchmark that provided the reference performance of 5.5 GFLOPS. As expected, we observe significantly increased benefits for the DSCG and MACVETH codes with respect to the CSR baseline, as the main bottleneck for these codes, i.e., text segment sizes, is now alleviated by the 30 MiB LLC cache in the 12900K processor. The speedups and raw performance are detailed in Figure 4. Geometric mean speedups are now 2.0x and 1.2x with respect to the CSR and DSCG baselines, respectively, while, as was to be expected, the relationship between the number of executed instructions among the different code versions remains identical as under cold cache conditions.
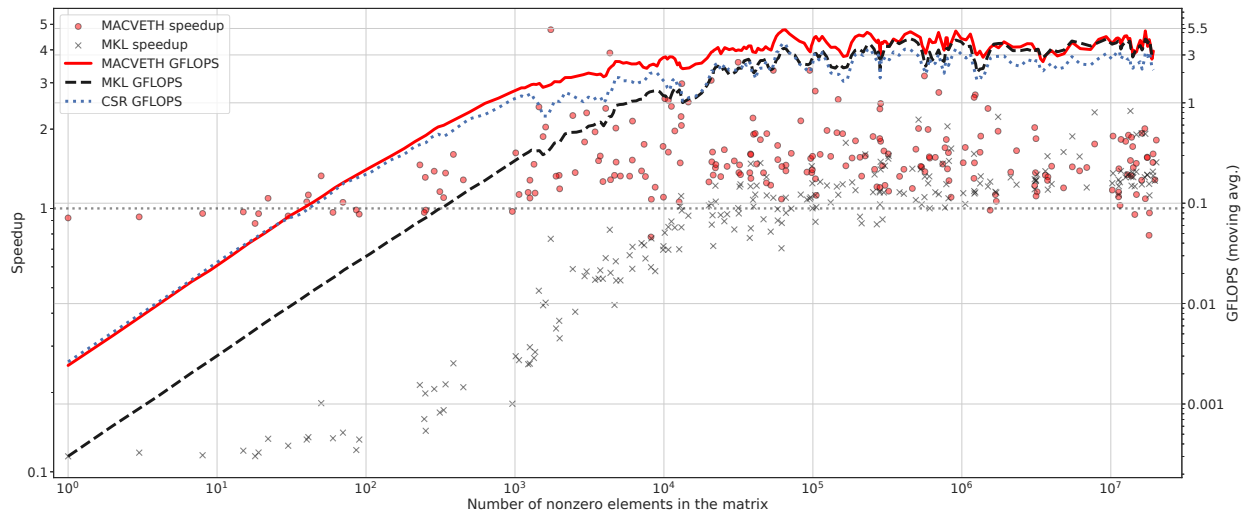
Figure 3: Speedup of instruction count-based MACVETH w.r.t. CSR and MKL (left) and raw performance (right). Log axes.
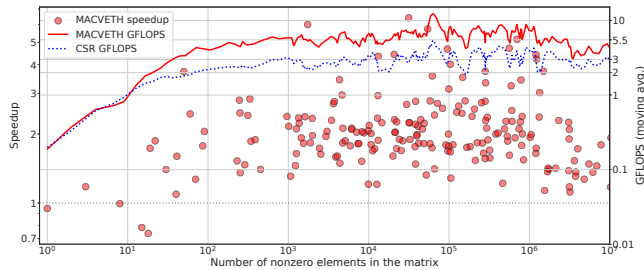

Figure 4: Hot cache speedup of MACVETH w.r.t. CSR (left) and raw performance (right). Log axes.

Table 1: GFLOPS for each SpMV version. Columns labeled "$> X$" are geometric means restricted to matrices with more than X nonzeros.

Cache	Version	Performance (GFLOPS)				
		>1	>10K	>1M	> 10M	Peak
Cold	CSR	1.43	2.15	2.27	2.39	5.26
	DSCG	1.42	2.03	2.00	2.07	4.55
	MKL	1.15	2.56	3.07	3.29	5.31
	MACVETH	2.16	3.41	3.30	3.37	7.91
Hot	CSR	2.55	2.70	2.48	2.50	6.80
	DSCG	3.81	3.45	2.29	2.12	11.48
	MKL	3.29	3.81	3.33	3.31	7.13
	MACVETH	4.91	5.27	3.92	3.53	17.48

4.3 Intel Math Kernel Library

While the previous results are informative of the performance of the DSCG codes compiled with MACVETH with respect to a default CSR executor, the current state-of-practice standard is to employ the Intel Math Kernel Library (MKL) [40], a C++-based library designed to enable HPC. In particular, MKL provides a set of SparseBLAS routines, including SpMV. We used Intel MKL v2022.0.2 in our experiments. The SparseBLAS part of the library works in an inspector-executor fashion. The inspection part was left out of the timed scope to ensure fair comparisons. The MKL code employed in these experiments, linked against the single-threaded MKL libraries, is detailed in Listing 6. Multithreaded results are presented in Section 4.4.

Since only 12 out of 200 matrices in the original set had code sizes above the LLC size of 30 MiB, we added 30 matrices in between 10M and 20M nonzeros. These new matrices were selected using the same original notion of fairly sampling the SuiteSparse domain according to matrix sizes and regularity of their sparsity structure.

Figure 3 details the results obtained in these experiments. MKL achieves a geometric mean speedup of 0.8x with respect to the CSR executor, i.e., a slowdown. This is due to significant initialization overhead of the executor function, which is not offset until matrix

sizes are significantly large, approximately above 10K nonzeros. If we consider only that subset, then MKL achieves a geometric mean speedup of 1.19x. The speedup of the MACVETH codes with respect to this subset is 1.33x. Generally speaking, in single-threaded executions MACVETH keeps a significant lead over MKL for matrices below 2M nonzeros. After that point, it depends on the particular characteristics of each input matrix, with MACVETH leading by a geometric mean speedup of just 1.02x, being faster in 27 out of 47 matrices. The performance of each version for a particular matrix depends on how the complex execution trade-offs play off for a given matrix size and sparsity structure. MACVETH achieves a reduction of 6.07x in the number of instructions executed, while MKL features a better execution schedule that incurs 1.33x less L2 misses and 1.50x less L3 misses than the CSR version. Table 1 summarizes raw performance in single-threaded executions for each experimental version, including the larger matrices and performance under hot cache conditions, measured in a similar way as in Section 4.2.

We ran additional experiments comparing the deprecated MKL (dMKL) version to the newer inspector-executor MKL (ieMKL). For the cold cache setup, we found dMKL to be, on average, 1.51x slower than ieMKL. In our tests, dMKL is slower for 100% of the matrices. It executes 16.8% less instructions, and presents roughly the same number of L2 and L3 misses, although it reduces the
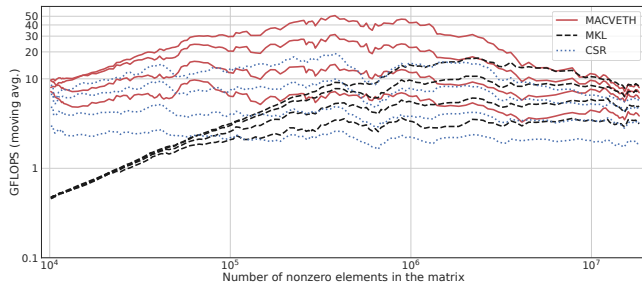
Figure 5: Hot cache multithreaded performance. Each of the four series represents the speedup for 1/2/4/8 threads. Higher thread count yields higher performance. Log axes.

Table 2: Performance of multithreaded codes, hot cache.

Threads	Version	Performance (GFLOPS)				
		>1	>10K	>1M	> 10M	Peak
1	CSR	2.01	2.05	1.87	1.82	3.65
	DSCG	2.94	3.29	2.23	2.10	10.86
	MKL	0.65	2.03	2.87	2.98	5.79
	MACVETH	3.87	5.00	3.79	3.47	16.40
2	CSR	2.82	3.61	3.25	3.08	7.07
	DSCG	4.35	6.10	3.93	3.56	18.49
	MKL	0.81	2.87	4.79	4.79	8.21
	MACVETH	5.25	8.61	6.44	5.70	30.07
4	CSR	4.08	6.30	5.40	4.60	13.64
	DSCG	5.80	9.96	5.16	3.97	38.63
	MKL	1.00	3.89	7.64	6.99	15.68
	MACVETH	6.87	13.88	9.17	6.81	58.78
8	CSR	5.11	9.67	7.81	5.74	26.47
	DSCG	6.57	14.32	6.48	4.40	65.83
	MKL	1.12	4.66	10.67	8.62	26.37
	MACVETH	7.75	19.14	12.31	7.82	100.87
16	CSR	4.54	10.59	9.12	6.16	32.95
	DSCG	5.24	13.37	6.82	4.58	66.45
	MKL	1.13	4.84	11.83	9.10	34.49
	MACVETH	6.42	18.17	12.51	7.85	105.60

number of D1 misses by 1.7x. It executes 1.7x more scalar floating point operations, but it issues 60.8% of the total FLOPs using 256-bit vector instructions, something that the ieMKL version never does (it uses 128-bit vector ops for 82.2% of the total FLOPs, and scalar ops for the rest).

As for the hot cache setup, dMKL is still 1.06x slower on average, but in this case it is faster than ieMKL for 41 matrices. The average speedup for these 41 matrices is 1.12x. The general stats are similar to the ones presented above for cold cache. When we restrict ourselves to the 41 matrices for which dMKL is faster, the reduction in the number of instructions reaches 2.1x. The increase in scalar operations is reduced to 1.57x. The share of FLOPs that is issued using 256-bit vector ops reaches 85.4%. In order to keep plots simple, the figures in this section include only results for the newer, better performing, inspector-executor MKL version.

4.4 Multithreaded Results

We parallelize codes using OpenMP in order to evaluate the scaling of the speedups observed for single-threaded versions of these codes. We target the hot cache setup, with 100 repetitions of the multiplication kernel, in order to ensure that computations are substantial enough to benefit from parallelization. The CSR baseline is parallelized by performing a static block distribution of the sparse matrix rows among the different threads, as seen in Listing 5[5]. The DSCG and MACVETH codes are parallelized by dividing the number of FLOPs fairly among the different threads. Note that neither approach guarantees fair schedules: heterogeneous distribution of nonzeros among the matrix rows will cause load imbalance between different threads of the CSR executor. Similarly, for the DSCG and MACVETH codes the number of FLOPs does not drive performance, which depends essentially on the access patterns performed by each thread, determining the vectorization recipes and size of its code block. We observe a difference of 1% between the average standard deviations in both parallelization approaches, highlighting that the experimental setup is fair. MKL schedules computations across threads according to the inspection phase.

We observe noticeable performance differences between the parallel codes executed with one thread and the single-threaded versions. In the CSR case, this difference is due to GCC disabling

---

[5]We empirically observed the dynamic distribution to be slightly less performant for our experimental set.

vectorization of the irregular loop when compiling with -fopenmp.[6] The number of LLC misses increases by 1.07x, and the executed instructions by 1.21x. As for MKL, there is a 1.06x overhead incurred by the parallel version, driven by a 1.15x increase in the LLC misses. Besides these, all code versions perform significantly worse for smaller matrices due to the overhead of spawning the threads.

We execute the experimental set using 1, 2, 4, 8, and 16 threads, pinned to different physical performance cores in the processor except for the case of 16 threads, where each of the 16 physical cores is assigned 2 threads (one per logical core). Figure 5 and Table 2 show the multithreaded results. Series for 16 threads were removed from the figure to improve readability, as the speedup across the entire experimental set was 1x for MKL, and below 1x (a slowdown) for the CSR, DSCG and MACVETH versions, but are included in Table 2. The MACVETH version scales the best for matrix sizes below 8M nonzeros. From that point on, the best performance is offered either by MACVETH or MKL depending on the particular trade-offs of each input matrix regarding instruction count and memory performance. The best geometric mean speedup is offered by MKL using 4 or more threads. However, MACVETH is the superior choice for many of the largest matrices in the experimental set, e.g., MACVETH achieves a 1.66x speedup versus MKL using 16 threads for Mazaheri/bundle_adj, the largest matrix in the experimental set with 20.2M nonzeros, while it presents a 0.77x speedup (a slowdown) for DIMACS10/hugetric00010, the second largest matrix with 19.8M nonzeros. The reason for the reduced scaling in the DSCG and MACVETH versions for some matrices appears to be the need for very aggressive code prefetching to feed instructions to the processor front-end. When using 16 threads the memory is not capable of providing code at the rate necessary to keep the front-end running.

4.5 Analysis and Code-Generation Performance

The code generation toolchain used in these experiments includes three phases: i) matrix inspection and DSCG code generation; ii)

---

[6]It is not possible to turn vectorization on again using ivdep or other pragmas.

MACVETH source-to-source optimization; and iii) GCC compilation and linking. The current version of i) is implemented in C in a single-threaded fashion. MACVETH is implemented in C++, and can be run in parallel on different files. For many small matrices, the entire process can be run in seconds. For the largest matrices these times increase to a few minutes. Generally speaking, the codes in the "sweet spot" single-threaded performance region close to the peak of 5.5 GFLOPS, with sizes around 10M nonzeros, can be processed and compiled in 5 to 10 minutes. For codes around 1M nonzeros the processing time is around 1 minute. Processing time scales mostly linearly with the number of nonzeros, with constant terms that depend on the regularity of the matrix structure.

The time to directly compile the DSCG codes, using autovectorization as implemented in GCC, is roughly equivalent to the time to run source-to-source optimization using MACVETH and then compiling the result using GCC. Since the MACVETH-generated code generates mainly intrinsics and assembly code, the final compilation step is much faster than when deploying autovectorization.

4.6 Code Statistics

We statistically analyzed the MACVETH-generated codes to extract information about how the scalar operations are being vectorized in terms of the usage of MACVETH recipes vs. generic AVX2 gathers.

The entire set of matrices on which we experiment contains approximately 554M nonzeros. Out of these, 16M operations are issued using scalar instructions in the MACVETH codes, which leaves roughly 538M nonzeros to be processed using vector operations. The MACVETH codes also contain 46M vector FMAs (1M in 128-bit mode and 45M in 256-bit mode), and 25M vector MULs (1M in 128-bit mode and 24M in 256-bit mode). The remaining operations are swizzles, casts, gather-like operations (whether implemented through gathers or through custom recipes), and reductions through adds and horizontal adds, but do not perform basic matrix-vector computation. As such, the 71M combined vector FMAs and MULs perform an average 7.6 useful scalar multiplications each.

In order to feed these multiplication operations, the code emits 151M gather-like operations, i.e., each FMA or MUL fuses together an average 2.1 independent reductions. Out of these 151M gather-like operations, 97M (64%) are issued through custom recipes discovered by microbenchmarking using MRKVS and MARTA, while the remaining 54M are executed through gathers, and more precisely through `_mm256_i32gather_ps()`. Considering that, out of the 255 packing recipes considered by MACVETH, only 129 (50.5%) are statically issued through custom recipes, this means that the targeted packing classes are the ones which most frequently appear in the experimental set.

4.7 Sensitivity Analysis

The results presented in the previous section depend on architectural and software factors. For instance, the LLC cache size determines how well the hot cache results will scale with matrix size. Similarly, the speedup of MACVETH with respect to the CSR executor and the DSCG versions depends on how well the compiler vectorizes irregular (CSR) and fully-unrolled (DSCG) codes.

In order to assess the impact of our architectural and compiler choices in the previous results, we experimented on an AMD Ryzen9

Table 3: Performance on Zen3, hot cache.

Cache	Version	Performance (GFLOPS)				
		>1	>10K	>1M	> 10M	**Peak**
Hot	CSR	2.02	2.13	1.98	2.04	4.10
	DSCG	2.68	2.36	1.89	1.77	9.92
	MKL	3.04	3.56	3.15	3.10	6.80
	MACVETH	3.16	3.32	2.67	2.37	10.76

5950X (Zen3 architecture), and we introduced Clang v13.0.1 and ICC v2022.0.1 as alternate compilers. Note that MACVETH optimization on the Zen3 machine required re-evaluating the MRKVS-generated recipes to find again which packing classes benefit from using ad-hoc recipes instead of the AVX2 gather instruction.

There is no significant performance difference (below 5%) for the CSR, MKL, and DSCG versions of the single-threaded kernels when compiled with ICC versus GCC. The reason appears to be less efficient AVX2 code produced from the MACVETH-generated intrinsics. Whereas the GCC version generated 4.8%, 9.1%, and 86.1% of vector operations as scalars, 128-bit, and 256-bit vector operations, respectively, the ICC version changes this mix to 6.6%, 4.4%, and 89%. In this process, it generates 1.14x more instructions, and consequently 1.13x more LLC misses. When using ICC, MACVETH still improves versus CSR (1.35x), DSCG (1.40x) and MKL (1.06x). The Clang-generated MACVETH codes perform identically to the ones generated by ICC, with a negligible performance difference under 1%. DSCG codes, however, present a 1.27x slowdown, derived from a 1.40x increase in the number of floating-point operations due to a less aggressive vectorization, and consequent 1.14x increase in the number of total instructions executed, together with a 1.28x increase in LLC misses.

Performance on the Zen3 machine is lower than on Alder Lake in our experiments, for all code versions. Note that, in fact, the reference Intel HPCG benchmark reports a raw SpMV performance of 4.3 GFLOPS in Zen3, versus 5.5 GFLOPS in Alder Lake. The geometric mean speedup of MACVETH with respect to the CSR and DSCG versions is 1.37x and 1.33x, respectively. The same trade-offs involving number of instructions and memory performance can be observed for the larger matrices. The raw performance results for this architecture are provided in Table 3. Besides the weaker raw performance across the board, we observe decreased scaling of the benefits of the DSCG and MACVETH versions for large matrices. The fundamental factor appears to be the processor front-end: the Zen3 legacy decoder is 4-wide, versus the 5-wide one in Alder Lake. This factor impacts data-specific codes the hardest: being fully unrolled, explicit codes, they cannot take advantage of the more efficient μ-instruction caches in both architectures, from which 6 instructions can be decoded per cycle. The smaller CSR and MKL kernels benefit from this improved issue rate.

5 Concluding Remarks

Optimizing sparse-immutable data structures by exploiting data-specific compilation trades off large data size and indirect accesses for larger code size but using only direct accesses. We presented the MACVETH system (Multi-Architectural C-VEcTorizer for HPC applications) to address the synthesis of high-performance SIMD implementations for such regular strided-access reduction or map

loops. Experimental results on 230 sparse matrices demonstrate the performance benefits, and limitations, achieved with MACVETH versus Intel MKL on modern multi-core processors.

Acknowledgments

References

[1] A. Abel and J. Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Intl. Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS.* Providence, RI, USA, 673–686.

[2] A. Abel and J. Reineke. 2022. uiCA: Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing, ICS.* Virtual Event, USA, 33:1–33:14.

[3] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. 2014. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In *Intl. Conference for High Performance Computing, Networking, Storage and Analysis, SC.* New Orleans, LA, USA, 781–792.

[4] T. Augustine, J. Sarma, L.-N. Pouchet, and G. Rodríguez. 2019. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI.* 625–639.

[5] N. Bell and M. Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

[6] N. Bell and M. Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *ACM/IEEE Conference on High Performance Computing, SC.* Portland, OR, USA.

[7] H. Bian, J. Huang, R. Dong, L. Liu, and X. Wang. 2020. CSR2: a new format for SIMD-accelerated SpMV. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID.* Melbourne, Australia, 350–359.

[8] X. Chen, P. Xie, L. Chi, J. Liu, and C. Gong. 2018. An efficient SIMD compression format for sparse matrix-vector multiplication. *Concurrency and Computation: Practice and Experience* 30, 23 (2018), e4800:1–10.

[9] Chips and Cheese. 2021. *How Zen 2's Op Cache Affects Performance.* [Accessed: 01-03-2022].

[10] J.W. Choi, A. Singh, and R.W. Vuduc. 2010. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP.* Bangalore, India, 115–126.

[11] S. Chou, F. Kjolstad, and S. Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 123.

[12] T. A. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38 (2011), 1–25. Issue 1.

[13] L. de Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS.* 337–340.

[14] E.F. D'Azevedo, M.R. Fahey, and R.T. Mills. 2005. Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. In *Intl. Conference on Computational Science, ICCS.* Atlanta, GA, USA, 99–106.

[15] A. Fog. [n. d.]. *4. Instruction Tables. Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD, and VIA CPUs.* [Accessed: 01-03-2022].

[16] F. Franchetti, Y. Voronenko, P. A. Milder, S. Chellappa, M. R. Telgarsky, H. Shen, P. D'Alberto, F. de Mesmay, J. C. Hoe, J. MF Moura, et al. 2008. Domain-specific library generation for parallel software and hardware platforms. In *Intl. Symposium on Parallel and Distributed Processing, IPDPS.* 1–5.

[17] GNU GCC. [n. d.]. *Auto-Vectorization in GCC: Using the Vectorizer.* [Accessed: 01-03-2022].

[18] J. Godwin, J. Holewinski, and P. Sadayappan. 2012. High-performance Sparse Matrix-vector Multiplication on GPUs for Structured Grid Computations. In *5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU.* London, UK, 47–56.

[19] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste. 2021. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.

[20] J. Hofmann, J. Treibig, G. Hager, and G. Wellein. 2014. Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips. In *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing, WPMVP.* Orlando, Florida, USA, 57–64.

[21] M. Horro. 2022. *Manycore Architectures and SIMD Optimizations for High Performance Computing.* Ph. D. Dissertation. Universidade da Coruña, Spain.

[22] M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. 2022. MARTA: Multi-configuration Assembly pRofiler and Toolkit for performance Analysis. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS.* Singapore, 79–89.

[23] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 77:1–29.

[24] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI.* 127–138.

[25] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423.

[26] Y. Li, P. Xie, X. Chen, J. Liu, B. Yang, S. Li, C. Gong, X. Gan, and H. Xu. 2020. VBSF: a new storage format for SIMD Sparse Matrix–Vector multiplication on modern processors. *The Journal of Supercomputing* 76, 3 (2020), 2063–2081.

[27] LLVM. [n. d.]. *Auto-Vectorization in LLVM.* [Accessed: 01-03-2022].

[28] D. Nuzman, I. Rosen, and A. Zaks. 2006. Auto-vectorization of interleaved data for SIMD. *ACM SIGPLAN Notices* 41, 6 (2006), 132–143.

[29] A. Pohl, B. Cosenza, and B. Juurlink. 2019. Portable Cost Modeling for Auto-Vectorizers. In *Proceedings of the 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS.* Rennes, France, 359–369.

[30] V. Porpodas. 2017. Supergraph-SLP auto-vectorization. In *26th International Conference on Parallel Architectures and Compilation Techniques, PACT.* 330–342.

[31] V. Porpodas, R. CO Rocha, and L. FW Góes. 2018. Look-ahead SLP: Auto-vectorization in the presence of commutative operations. In *Proceedings of the Intl. Symposium on Code Generation and Optimization, CGO.* 163–174.

[32] L.-N. Pouchet. 2011. PolyBench: The Polyhedral Benchmarking suite, version PolyBench/C 4.2.1. http://polybench.sf.net. Last accessed: May 2017.

[33] M. Puschel, J. MF Moura, J R Johnson, D. Padua, M. M Veloso, B. W Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[34] I. Rosen, D. Nuzman, and A. Zaks. 2007. Loop-aware SLP in GCC. In *GCC Developers Summit.*

[35] Y. Saad. 1990. SPARSKIT: A basic tool kit for sparse matrix computations. (1990).

[36] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In *Proceedings of the 29th ACM on Intl. Conference on Supercomputing, SC.* 99–108.

[37] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, and R. Ronen. 2001. Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA. In *Proceedings of the Intl. Symposium on Low Power Electronics and Design, ISLPED.* Huntington Beach, CA, USA, 4–9.

[38] W.T. Tang, R. Zhao, M. Lu, Y. Liang, H.P. Huynh, X. Li, and R.S.M. Goh. 2015. Optimizing and Auto-tuning Scale-free Sparse Matrix-vector Multiplication on Intel Xeon Phi. In *13th Annual IEEE/ACM Intl. Symposium on Code Generation and Optimization, CGO.* San Francisco, CA, USA, 136–145.

[39] X. Tang, T. Schneider, S. Kamil, A. Panda, J. Li, and D. Panozzo. 2020. EGGS: Sparsity-Specific Code Generation. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 209–219.

[40] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. 2014. Intel Math Kernel Library. In *High-Performance Computing on the Intel Xeon Phi.* Springer, 167–188.

[41] Z. Wegner. [n. d.]. *SPRDPL: Simple Python Recursive-Descent Parsing Library.* [Accessed: 01-03-2022].

[42] Z. Wegner. [n. d.]. *x86-sat.* [Accessed: 01-03-2022].

[43] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang. 2018. Cvr: Efficient vectorization of spmv on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO.* Vösendorf / Vienna, Austria, 149–162.

[44] S. Yan, C. Li, Y. Zhang, and H. Zhou. 2014. yaSpMV: Yet Another SpMV Framework on GPUs. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP.* ACM, Orlando, FL, USA, 107–118.

[45] C. Yang, A. Buluç, and J D Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software, TOMS* 48, 1 (2022), 1–51.

[46] X. Yang, S. Parthasarathy, and P. Sadayappan. 2011. Fast Sparse Matrix-vector Multiplication on GPUs: Implications for Graph Mining. *Proc. VLDB Endow.* 4, 4 (2011), 231–242.