Autovesk: Automatic vectorized code generation from unstructured static kernels using graph transformations

Hayfa Tayeb, Ludovic Paillat, Bérenger Bramas

Autovesk: Automatic vectorized code generation from unstructured static kernels using graph transformations

HAYFA TAYEB, ICube Lab, France, Inria, France, and University of Strasbourg, France
LUDOVIC PAILLAT, ICube Lab, France, Inria, France, and University of Strasbourg, France
BÉRENGER BRAMAS, ICube Lab, France, Inria, France, and University of Strasbourg, France

Leveraging the SIMD capability of modern CPU architectures is mandatory to take full advantage of their increased performance. To exploit this capability, binary executables must be vectorized, either manually by developers or automatically by a tool. For this reason, the compilation research community has created several strategies to transform a scalar code into a vectorized implementation. However, most existing automatic vectorization techniques in modern compilers are designed for regular codes, leaving irregular applications with non-contiguous data access patterns at a disadvantage. We present in this paper a new tool, Autovesk, that automatically generates vectorized code from scalar code, specifically targeting irregular data access patterns. We describe how our method transforms a graph of scalar instructions into a vectorized one using different heuristics to reduce the number or cost of the instructions. Finally, we demonstrate the effectiveness of our approach on various computational kernels using Intel AVX-512 and ARM SVE. We compare the speedups of Autovesk vectorized code over GCC, Clang LLVM and Intel automatic vectorization optimizations. We achieve competitive results in linear kernels and up to 11x speedups in irregular ones.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**.

Additional Key Words and Phrases: Automatic vectorization, code generation, graph transformations

1 INTRODUCTION

Vectorization is a feature of modern CPUs that consists in applying a single instruction to multiple data (SIMD) [9]. It is one of the hardware features that have increased CPU performance despite the stagnation of the clock frequency. To exploit this functionality, a binary must explicitly use vector instructions, and this requires the program to be either vectorized by the back-end compiler when it issues the machine code, or at runtime [11].

HPC experts tend to vectorize their code manually [4] in assembly or with intrinsic functions [1]. This allows them to fine-tune and control the behavior of their computational kernels. However, this also requires significant expertise and can imply re-implement new kernels for each instruction set architecture (ISA). To mitigate this drawback, several abstraction libraries have been created [3, 8, 10, 15, 16, 24, 29]. They allow writing vectorized code with an abstract vector type which is fixed at compile time depending on the target architecture. This clearly facilitates maintainability while still asking for a moderate understanding of the vectorization concept. The main limitation of this approach is that the operations supported by the library should exist in all underlying ISA or, when it is not the case, the desired behavior should be implemented with more instructions, leading to difficulty to have a single kernel implementation efficient everywhere. From another research perspective, the compilation community proposed different strategies for automatically transforming a scalar code into a vectorized equivalent. The resulting mechanisms introduced in modern compilers work well when the code is regular, i.e. with contiguous data accesses and affine loops. However,

[1] An intrinsic function is a function that is usually converted into a single instruction by the compiler, allowing to use some instructions explicitly while remaining at a high level

Authors' addresses: Hayfa Tayeb, ICube Lab, 300 Bd Sébastien Brant, Illkirch, France and Inria, Nancy, France and University of Strasbourg, France, hayfa.tayeb@inria.fr; Ludovic Paillat, ICube Lab, 300 Bd Sébastien Brant, Illkirch, France and Inria, Nancy, France and University of Strasbourg, France, ludovic.paillat@etu.unistra.fr; Bérenger Bramas, ICube Lab, 300 Bd Sébastien Brant, Illkirch, France and Inria, Nancy, France and University of Strasbourg, France, berenger.bramas@inria.fr.

when the code to vectorize is irregular or with complex dependencies between the variables, the compilers tend to fail. Nevertheless, this strategy allows having a single source code clear and portable and still obtain good performance thanks to the compiler. This paper presents a new strategy to vectorize irregular and unstructured computing kernels. Our method aims to complement existing mechanisms by focusing on non-contiguous data access patterns. It allows non-expert programmers to benefit from vectorization and experts to delegate complex vectorization implementations that require many data transformations. Indeed, our tool attempts to minimize the number of instructions at the cost of complex vector transformations (permutations, extractions, merging, etc.) that are usually tedious to manage with intrinsics or in assembly.

The contribution of our work is the following:

- to provide a new automatic vectorization strategy. Our approach operates on an instruction graph, which is transformed to minimize the total number of instructions;
- to decompose the problem into several sub-problems, and provide different heuristics to solve them;
- to study the performance of the resulting vectorized kernels against automatic vectorization in modern compilers, namely GCC and icpx Intel, and state-of-the-art techniques in LLVM's vectorizer;
- to propose a new instruction ordering strategy to reduce the number of register spilling, i.e. the transfers between the stack and the registers.

The paper is organized as follows. In Section 2 we detail the background on vectorization and automatic vectorization. Then, we discuss the related work in Section 3. We present our method in Section 4. Finally, we evaluate the benefit of our method in Section 5.

2 PROBLEM DESCRIPTION

2.1 Vectorization

The principle of vectorization, as developed in modern CPUs, can be described as a technique that allows applying a single instruction on one or several vectors of native data types. The implementation of this concept in modern CPUs comes with several constraints. For instance, the size of the vector is fixed and hardware dependent, even though it is possible to pad the vector with arbitrary values to work on smaller sizes. The vectors are CPU registers, such that load and store operations must be used to exchange data with the main memory. The operation set supported by a processing unit to manipulate the vectors is hardware dependent. More precisely, it is tied to the ISA and each ISA may have operations that are not supported by others. For example, some can allow shuffling of the data inside the vectors, whereas others might only support shuffling inside half of the vectors. Some might support non-contiguous memory accesses with gather/scatter operations, but others might not. However, we can surely consider that they all support arithmetic/logical operations, which are usually applied term by term on two vector operands. In our approach, we consider that vector transformation instructions are available to shuffle the data, extract elements, and merge two vectors in one. When considering performance, it is usually more efficient to loads/stores contiguous elements in memory and use aligned memory accesses (i.e. the starting address pointer is a multiple of a given value). Predication is a feature supported by some ISAs, such as ARM SVE, which allows operations to be applied only to certain active elements using predicate vectors. Other ISAs, namely x86, can achieve similar functionality but with a different implementation. It generates mask vectors that are used with binary operations to eliminate inactive elements.

2.2 Automatic vectorization

Automatic vectorization consists of converting a scalar source code or binary into a vectorized equivalent without requiring explicit guidance from the developers. This is a difficult problem because the original program can be written in a way that makes it difficult to vectorize. The main reasons are the use of data structures/accesses that are not adequate, or when it is difficult to predict the dependencies leading to the impossibility to generate a vectorized kernel before the execution. Moreover, even if vectorization can be applied, the result can be inefficient and take more time than the original version to execute. Indeed, if we suppose that we put a single scalar value (or a few of them) inside each vector, it seems straightforward to obtain a naive vectorized program. However, the waste in the vectors and the fact that vectorized operations can have a higher cost (either in latency or throughput) would lead to poor implementation. This is why a good metric to predict the performance of a vectorized kernel is to count the number of instructions, with potentially a cost associated with each of them, and consider that the lower is the better. A more refined optimization can also predict the optimal use of hardware units by considering that some operations applied to disjoint data can be performed concurrently.

In the current study, we focus on static kernels, which allows us to know the original number of scalar instructions. Also, some non-static kernels could be seen as a dynamic repetition of a static kernel, such that our method is also useful in this case. To illustrate the difficulty of the problem, we will provide a rough complexity to obtain an optimal solution if we attempt to compute all possibilities to find the optimum.

Consider we have a direct acyclic graph $G(V, E)$ where the V vertices correspond to the N scalar operations of the original program, and the E edges are the dependencies. We state that the number of possibilities is bounded by

$$\prod_{i=0}^{\lceil N/k \rceil} \frac{(N - i \times k)!}{(N - k \times i - k)!} \tag{1}$$

which is equal to the number of possibilities to group the elements in subsets of k among N. We can lower this number by considering that N is composed of L loads, S stores and O different operations each of I scalar operations (the operations of the same type that are at different depth in the graph can be considered of different type). However, this limit considers all vectors as full, which might not be the case, increasing, even more, the number of possibilities. Therefore, a brute force approach to find an optimal vectorial graph seems impossible and motivates the use of heuristics.

3 RELATED WORK

Auto-vectorization transforms scalar operations into SIMD operations that can process multiple data elements in parallel, leading to significant performance improvements on modern processors. There are two main strategies for automatic vectorization in modern compilers: loop vectorization and Superword Level Parallelism (SLP).

Loop vectorization is a technique that leverages parallelism in loop iterations to transform scalar operations into SIMD operations. This process involves expanding each operation in the loop from a scalar type to a vector type, which is a straightforward transformation for simple loops. However, many computations require more advanced loop transformations to be in a vectorizable form and preferably without dependencies to ensure consistency. One of the challenges of loop vectorization is dealing with non-contiguous access patterns to the operation data. To address this, Nuzman et al. [17] introduced an extension to a classical loop-based vectorizer that handles non-unitary computations and data accesses, particularly data with powers of 2 on CPUs. This extension helps to exploit spatial locality and address interleaved data access, but remains limited to regular strided access patterns.

Loop vectors are mostly dealing with the data parallelization aspect directly related to the SIMD architecture. As a counterpart, there is the SLP algorithm, originally introduced in [14]. It performs instruction-oriented vectorization. It is commonly used to convert linear code into vector code, which complements traditional loop-based vectorizers. SLP algorithm analyses the input code by scanning the compiler-generated intermediate representation (IR) looking for instruction groups that can be combined into vectors. It replaces these selected instruction groups with the corresponding vector instructions. Rosen et al. [23] proposed the Bottom-Up SLP algorithm implemented in the well-known GCC and LLVM compilers. The main contribution of the algorithm is its ability to handle sequential code anywhere in the program, not just in loops. It can also vectorize code inside loops if the loop vectorizer fails. Our proposed tool has a similar objective but is currently limited to working with code that uses C++ templates, as it has not yet been integrated into a compiler at the IR level. While the evaluation focuses on vectorization opportunities within loops, which are generally more common, it still provides valuable insights into the potential of our techniques. The SLP approach suffers from some limitations. Mainly, it operates locally or even atomically on individual SLP graphs which can be limiting in the case of consecutive graphs sharing data. In these cases, an overestimation of the vectorization costs may occur and lead to missed opportunities, even in a simple code. The SLP vectorizer may miss several opportunities for vectorization in loops. Rocha et al. [22] presented Vectorization Aware Loop Unrolling (VALU), a new heuristic accompanying the compiler in searching for loop-unrolling opportunities to allow auto-vectorization of sequential code. VALU checks whether the loop unrolling will be cost-efficient for the SLP vectorizer and identifies the loop unrolling factor that can maximize the use of the vector units in the target architecture. In Autovesk, we fully unroll the kernels as we create the scalar instruction graph, thus allowing more instruction grouping possibilities, i.e. better vectorization of the code. Porpodas et al. [20] presented Look-Ahead SLP (LSLP), an improved vectorization algorithm based on SLP that mainly focuses on commutative operations. It is implemented in LLVM compiler. LSLP's main contributions are first, to be able to extend SLP graph data structure to form multi-nodes of chains of commutative operations of the same type. Secondly, it introduces a more powerful operand reordering scheme that makes informed decisions based on instructions deeper in the code. Our method enables chain reduction for improved vectorization efficiency by transforming the scalar instruction graph and grouping and annotating commutative operation nodes. Porpodas et al. [19] go further and present Super-Node SLP (SN-SLP), a new algorithm similar to LSLP with multi-nodes, including both commutative operations and their corresponding inverse elements, for example, addition and its inverse subtraction. SN-SLP leverages the algebraic properties of these operators to allow additional transformations, increasing the possibilities of vectorization. It is implemented in LLVM. Several works have proposed heuristics to optimize instruction ordering and code regions for vectorization in SLP-based algorithms. Nevertheless, our new approach is competitive and offers additional flexibility by allowing the vectorization of kernels with irregular data access patterns through instruction grouping and reordering transformations and heuristics.

As vectorization can be seen as a parallelization at the instruction level, the work on automatic parallelization using the polyhedral model can be used [2, 18, 28]. Polyhedral compilers are designed specifically to support calculations in the affine domain in which the loop boundaries and the subscript expressions can be expressed as integer linear functions of both the loop indices and the loop constants.

Irregular applications such as graph algorithms, particle simulation codes and sparse matrix codes that use compact representations [1, 6, 7, 13] are now being considered for acceleration using SIMD capabilities. Although, the initial focus of SIMD devices was on speeding up regular applications such as dense matrix multiplication and image processing. Vectorization becomes particularly challenging when dealing with irregular access patterns. Another challenge is dealing with memory alignment, which has been largely ignored in previous research focusing on properly aligned memory

references. Existing vectorization approaches for such irregular applications include inspector/executor and conflict masking. Inspector-executor transformations involve loop and data layout transformations that require a runtime component due to unresolved indirect array accesses. The Sparse Polyhedral Framework (SPF) extends the capabilities of the Polyhedral Framework by incorporating uninterpreted functions to represent non-affine array accesses, non-affine loop boundaries, and inspector-executor transformations [25, 30]. Conflict masking is an approach to resolving data conflicts in SIMD vectors by identifying non-conflicting lanes and writing only to them in each round of execution. A conflict detection instruction (vpconflict) has been added to the Intel AVX-512 instruction set to facilitate conflict masking. The vectorization performance of conflict-masking is dependent on the input distribution. If conflicts arise frequently in the input, conflict-masking will result in low SIMD utilization and thus poor vectorization performance. Jiang and Agrawal [12] propose a code transformation called in-vector reduction that can efficiently vectorize a class of associative irregular applications. Further research aims to support aggressive SIMD vectorization of important loops, even when their bodies contain complex control flow and the entire computation cannot be fully parallelized such as speculative execution [26, 27]. Beyond this, Chen et al. [5] propose VeGen, an extendable framework that targets non-SIMD vector instructions by introducing a new model of vector concurrency called lane-level concurrency (LLP). This new model goes beyond traditional SLP by allowing instructions to perform multiple non-isomorphic operations, and operations on any output lane can use values from arbitrary input lanes.

4 AUTOVESK

4.1 Transformation flow

We provide in Figure 1 an overview of the internal organization of our engine[2]. First, the process starts by obtaining a graph of scalar instructions of a given computational core (Section 4.2). In the second step, the graph can be transformed to exploit reduction when relevant (Section 4.3). In the third step, the instructions are grouped to obtain a meta-graph (Section 4.4). Next, the groups are split to match the desired vector size denoted by VEC_SIZE (Section 4.5). We first divide the loads/stores, then the arithmetic/logical operations. Afterward, the order of the elements in each group is set, and the necessary permutations/extractions are performed (Section 4.6). Finally, the resulting vector operation graph is ready to be translated into a vector instruction list by our backend (Section 4.8).

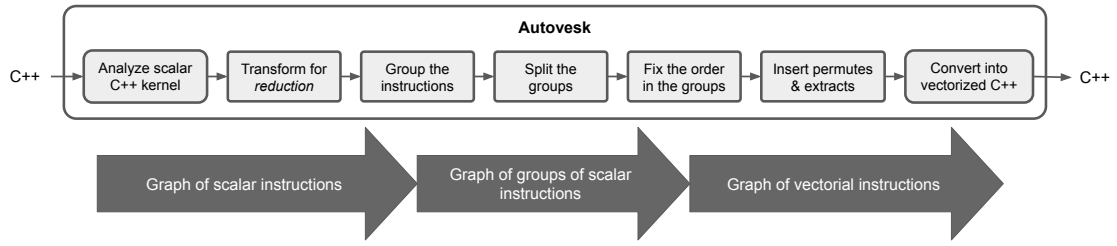


Fig. 1. Overview of the transformation flow in Autovesk. Starting from a C++ code, the tool uses successively a graph of scalar instructions, a graph of groups, and a graph of vectorial instructions.

To illustrate these different mechanisms, we will consider the vectorization of two kernels given in Figure 2 under the names KA and KB.

[2]https://gitlab.inria.fr/bramas/autovesk

```
1  void Kernel_A(const double* inValue1, const double* inValue2, double* outValue, const long size){
2      for(long int idx = 0 ; idx < size ; ++idx){
3          outValue[idx] = inValue1[(idx+2)%size] * inValue2[(idx*2)%size];
4      }
5  }
6  void Kernel_B(const double* inValue1, const double* inValue2, double* outValue, const long size){
7      double x = 0;
8      for(long int idx = 0 ; idx < size ; ++idx){
9          x += inValue1[idx] + inValue2[idx];
10     }
11     outValue[0] = x;
12 }
```

Fig. 2. Two example functions to be vectorized which we denote by KA and KB. A static version of these functions is obtained by fixing the *size* variable. In this case, the loops can be fully unrolled and vanish.

4.2 Scalar graph generation

In our current engine, we generate a direct acyclic graph of scalar instructions using a custom C++ tool based on templates and operator overloading, which builds the graph during the execution of a compiled program. This stage will disappear when Autovesk is ported into an existing compiler, but it allows us to validate the transformation steps. We have four types of scalar nodes that compose our graph:

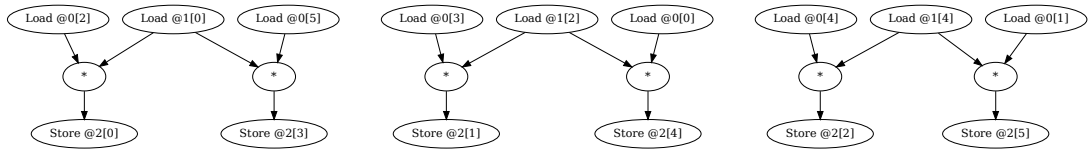
- set: is the assignment of a variable with a given value, for example, Line 7 in Figure 2.
- load: is the access to memory using an address to read a value, for example, Lines 3 and 9 in Figure 2 (right hand side).
- operation: is the use of one or several elements as input and the creation of an output. The arithmetic operations belong to this category.
- store: is the access to memory using an address to write a value, for example, Lines 3 and 11 in Figure 2 (left hand side).

What we get is a graph and not a tree since for a given node, the sub-graphs of its successors are not necessarily disjoint. For instance, if we have the instructions $a = x$, $b = a * a$, $c = a + a$ and $d = b + c$, the nodes that represent b and c will have the same predecessors and successors. In our representation, the scalar graph describes operations that are applied from loads/sets to stores without having the notion of variables. Thus, some instructions are invisible, such as copying a variable to a local variable, because they do not affect memory. The graph for KA and KB are given in Figure 3.

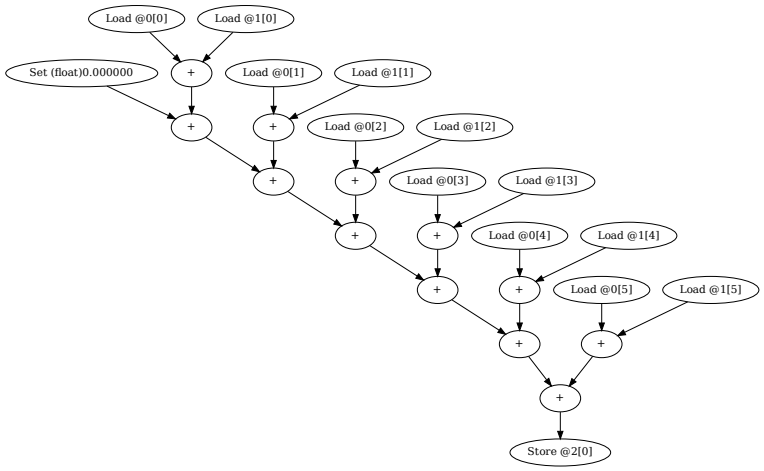
At this stage, we can already remove the duplicate nodes. Two nodes are considered equivalent if they perform the same operation and have the same input. If the operation is commutative, we consider that the input order is irrelevant. Therefore, we iterate on the graph from loads/sets to stores and remove the duplicate node wherever possible.

4.3 Commutative operations and reductions

In vectorization, the reduction pattern consists in changing the order of commutative operations to maintain multiple intermediate results that are then reduced into one, opening the possibility to vectorize more efficiently. In our case, we enable the reduction by transforming the scalar instruction graph and annotating the corresponding nodes to group them (Section 4.4). The transformation consists in finding a path in the graph of similar consecutive commutative operations and splitting this path such that we obtain independent VEC_SIZE sub-paths. Therefore, we iterate over all the nodes of the graph and when we find a commutative operation, we evaluate if it is the starting point of a reduction



(a) Scalar graph for KA



(b) Scalar graph for KB

Fig. 3. Graphs of scalar instructions for KA and KB for kernel size equals 6. The arrays given in the parameters are numbered from 0 to 2 in order.

path. Then, we check if the length of the path is enough to hope to benefit from the reduction, i.e. if there are more than VEC_SIZE nodes. If it is the case, we split this path into chunks of VEC_SIZE and connect them with reduction nodes. We provide the graph for KB after the reduction transformation in Figure 4. Unlike KB, there is no opportunity for reduction in the KA example, i.e. there is no chain of commutative operations that are reduced in an output result.

4.4 Instruction grouping

In this step, we generate a meta-graph where the nodes contain lists of scalar operations that could potentially be vectorized together. To perform this transformation, we group nodes of the same type. Thus, we group the loads related to the same array, and we proceed in the same way with the stores. Then, we put together the same operations that are at the same distance from the leaves of the graph (i.e., loads and stores). By proceeding this way, we ensure that there are no dependencies between the nodes of a group because we are managing it by construction. At this stage, groups can contain more scalar nodes than the size of the SIMD vector can handle (VEC_SIZE), and the internal order of scalar operations in the list is unspecified. We provide the group for KA and KB in Figure 5.

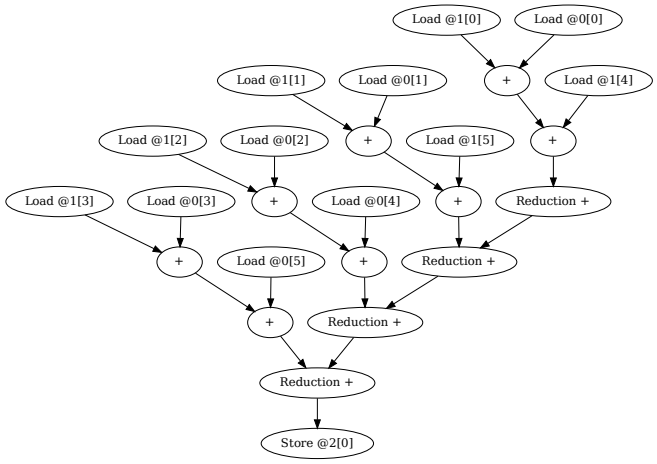


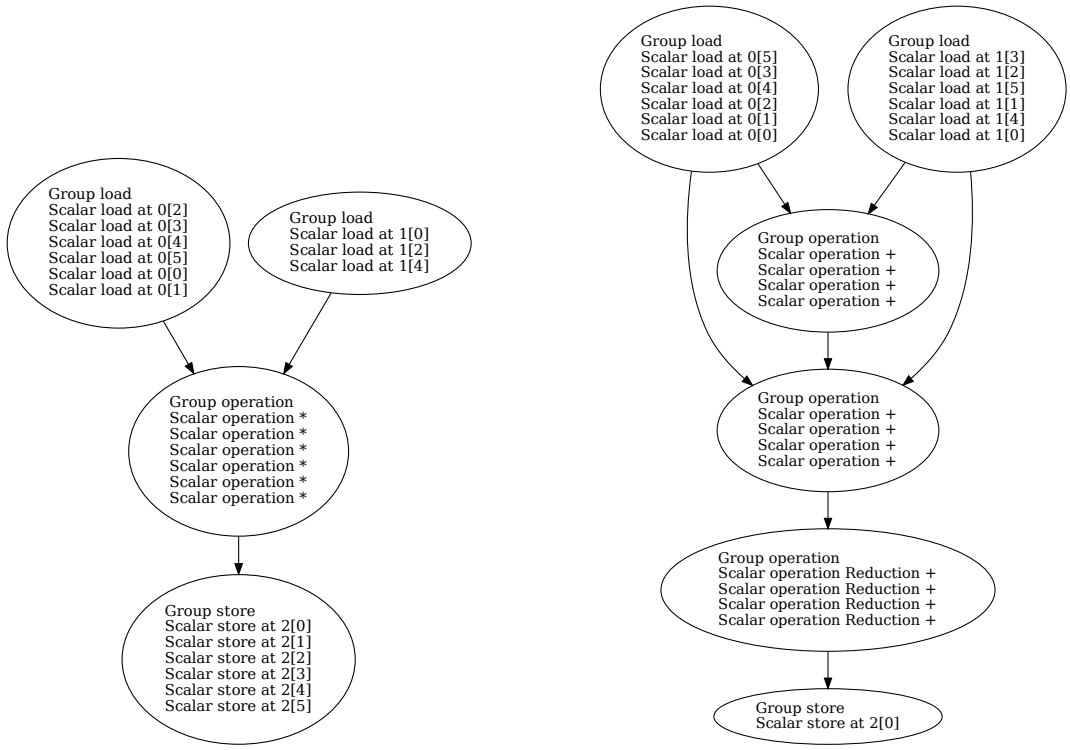
Fig. 4. Graph of scalar instructions for KB for kernel size equals 6 and $VEC\_SIZE = 4$. We can observe that the operations just before the store node were replaced by reduction nodes which materialize the reduction instruction that will be generated later. We can also see that the sources of these nodes are 4 reduction chains (as much as VEC_SIZE) that happen to be isomorphic so they can be vectorized.

4.5 Group divisions

Once we have the meta-graph, we need to decide how to split the groups that contain more than VEC_SIZE instructions. We do this in two steps, first on loads/stores, then on operations.

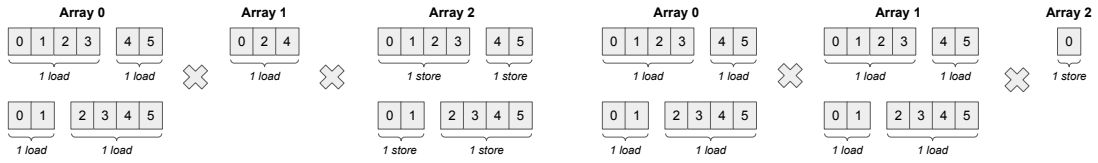
Loads/stores divisions. To divide the groups of load operations, we follow three rules. First, we need to load the values in a contiguous order. Second, we want to minimize the number of memory accesses. Finally, there should be no more than one vector that is not fully filled, i.e. all used vectors are filled with VEC_SIZE values and at most, one vector has less than VEC_SIZE values. Suppose we have an array of L scalar loads, where $L = k \times VEC\_SIZE + r$, with $0 \leq r < VEC\_SIZE$. If L is divisible by VEC_SIZE without leaving a remainder (i.e. $r = 0$), there is no choice: loads are in order and all the vectors will be full. However, when we have $r > 0$, we need to decide which vector will not be fully filled, i.e. how to distribute the scalar loads across the vectors with respect to the contiguous order. This decision affects the other stages of vectorization because it determines the initial state of the vectors, which in turn affects the number of transformations required in subsequent stages. This can have an impact on the final quality of the vector implementation.

Therefore, we tackle this problem by generating all the possible combinations satisfying the three above rules and picking the combination that leads to the vectorial graph with fewer nodes. If we consider that the number of vectors needed to load L values is V (with $V = k + 1$ when $r \neq 0$, or $V = k$ otherwise), we have V possibilities. This is because only one vector selected among the V vectors should be incomplete. If there are A arrays accessed in the kernel, either by load or store operations, the total number of possibilities is given by $C = \prod_{i=1}^{i=A} V_i$, with V_i the number of vectors for array i. Concretely, if we have two arrays as input and one array as output, and load ten vectors from each, C will be equal to one thousand. The different combinations for K are shown in Figure 6.



(a) KA (b) KB

Fig. 5. Graphs of groups for KA and KB for a kernel size equal to 6. An edge between two nodes indicates that at least one instruction from each of the two groups are connected.



(a) KA (b) KB

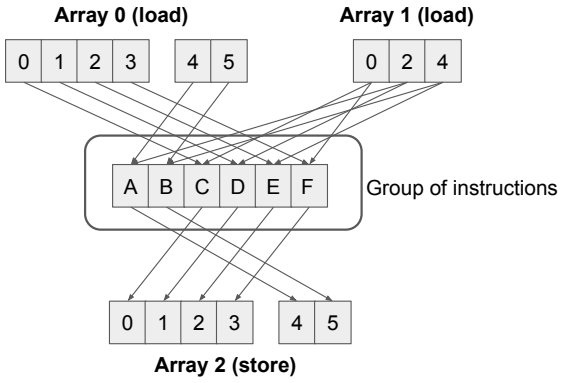
Fig. 6. All the combinations of loads/stores for the two kernels KA and KB considering size equals 6 and $VEC\_SIZE = 4$. There are 4 possibilities for KA ($2 \times 1 \times 2$) and KB too ($2 \times 2 \times 1$).

Operation divisions. After splitting the load/store groups, we split the operation groups. We iterate over the operation groups from loads to stores. By going through the operation groups this way, we can be sure that the inputs to the operations are already divided (or if the inputs are loads, they are also already fixed). For each group g we compute a score-matrix d of dimension $|g| \times |g|$ that aims to express which of the elements of g should be in the same vectorial operation, i.e. $d(i, j)$ describes the expected interest of putting the scalar operations i and j in the same vectorial

operation. The calculation of this score is defined in Algorithm 1. The score $d(i, j)$ increases with the number of common predecessors/successors between i and j. If they are two separate nodes, we increment the score by the number of sources in common. For each destination group: if it is a *store* group, we increment the score by 1; if it is an *operation*, we increment by 1 in the case of a group of size less than or equal to VEC_SIZE, or we increment by the inverse of the size of the destination in the case of a group of size greater than VEC_SIZE, so the larger the size of the destination, the less value we bring to this group. We show an example of a score-matrix in Figure 7.

Algorithm 1: Compute the matrix entry $d(A, B)$ for elements A and B of the same group.

```
score ← 0;
if A ≠ B then
    // Increment score by 1 or 2 for same sources
    score += (in_same_vector(A.src1,B.src1) + in_same_vector(A.src2,B.src2));
else
    // For the diagonal, add the number of sources, i.e. 1 or 2
    score += (A.src1 != null) + (A.src2 != null);
// Increment score if they are used at the same place (destination)
for d in Intersection(A.dest,B.dest) do
    if d is a Store Group then
        score += 1;
    else
        // d is an Operation Group
        if d.size ≤ VEC_SIZE then
            score += 1;
        else
            score += (1 /d.size);
```



	A	B	C	D	E	F
A	3	3	1	1	1	1
B	3	3	1	1	1	1
C	1	1	3	3	3	3
D	1	1	3	3	3	3
E	1	1	3	3	3	3
F	1	1	3	3	3	3

Fig. 7. Example of score matrix d for KA to split the scalar instruction group. The groups of loads and stores are already divided, and the Algorithm 1 is used to compute the matrix. The given matrix is found out for this specific configuration of loads/stores and could be different otherwise.

In order to split the group of operations, we propose two main strategies based on dividing the matrix d:

(1) a clustering strategy where we create initial sub-groups and then aggregate the elements using the best score remaining so far;
(2) a partitioning strategy where we split the matrix at the point with the lowest score.

We know we will need $V = \lceil |g|/VEC\_SIZE \rceil$ vectors. The partitioning strategy is provided in Algorithm 5 in Appendix A. We start with a partition that contains all the elements. We find in this partition the two elements with the lowest score in the matrix (line 14) and initiate two new partitions (line 15). Then we aggregate the remaining elements on these two partitions while ensuring that no partition exceeds a given size. This is done either by finding the highest accumulated score among the remaining elements to move them one by one (line 24) or by placing each element in the partition where it has the heaviest weight overall, i.e. the maximum accumulated sum of the scores of an element relative to all the others.

The clustering strategy is provided in Algorithm 6 in Appendix B. We start by finding the two elements i and j with the lowest scores in the matrix, i.e. those with the least number of predecessors and successors in common, and use them to initiate two sub-groups (line 2). Then, we find $V - 2$ other elements, one by one, by selecting the elements having the lowest scores compared with the already fixed elements (line 6). We end up with V sub-groups that each contain one element. We finish by processing the remaining elements. We compute, for each of them, the sum of the scores of the nodes of all subgroups respectively and choose the sub-group having the best score (line 16).

4.6 Fixing the order of vector's elements

At this stage, we have sub-groups that contain at most VEC_SIZE scalar instructions. The order of these instructions is fixed for the loads and stores but not for the operations. That is the aim of this layer. We have to find the correct positions to make the operations coincide with their predecessors and successors; thus, we will ensure the consistency of the kernel. Fixing the order will imply using permutations, merges and extractions; therefore, the objective is to find each group order to minimize the need for additional instructions. We note that using an *Extract* instruction is costly in terms of execution time for the vectorized code. We also note that if we have to extract a single element through a vectorial instruction, this is equivalent in terms of execution time to extracting several elements less than or equal to the size of the vector instruction.

In a sub-group, for VEC_SIZE scalar operations, we have $VEC\_SIZE!$ possibilities to order them, leading to the impossibility to compare all of them greedily. That said, in our case, an additional layer of constraints is added, which reduces the number of such possibilities. We have the connections between a sub-group of operations and its predecessors and successors, the order of the predecessors is already fixed (the order of a successor is fixed if it is a store). Therefore, taking into account the order of a successor, for example, leads to being forced to use extract/permute instructions to retrieve the data required for the operation from other groups and ensure kernel consistency. To illustrate the impact of a good order of operations, we present the examples in Figure 8.

It is enough that the positions of the operations in the group of instructions are changed that we find ourselves forced to add instructions *Extract* to have the vectorial instructions coincide. A good order prevents this. To address this issue, we propose Algorithm 2 which attempts to minimize the number of *Extract* instructions needed for the consistency of a given graph. If for a given operation node, we have the indices of its two successors equal then we can say that we have the right order and we set the position of this index. On the other hand, if the indices are different, we no longer have a choice, we set the position of one and admit that the other needs an *Extract*.

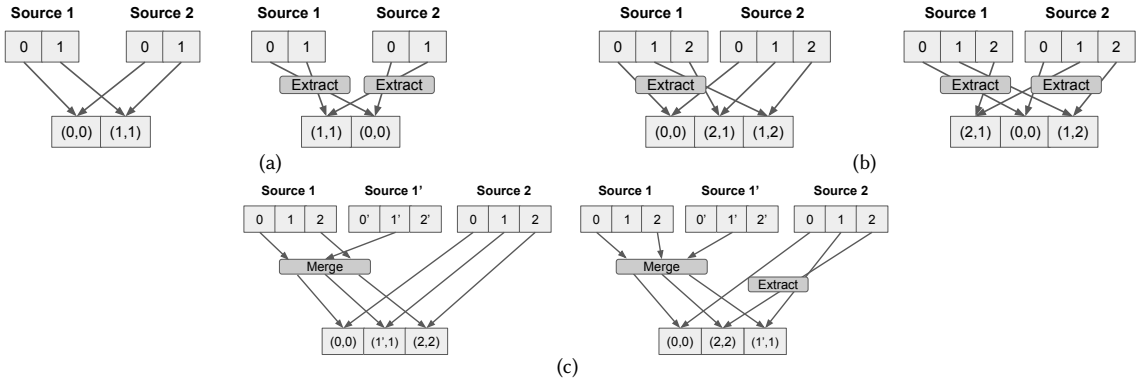


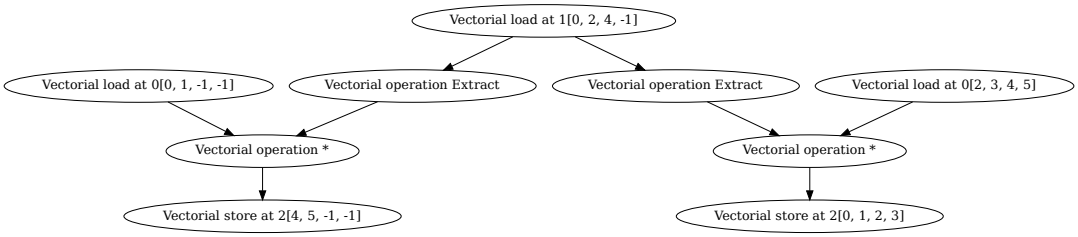
Fig. 8. Three examples with different order of the elements of the output vector. An *Extract* operation is used to select some elements and/or permuting elements. A merge operation is used to select elements from two vectors and store them into a single one.

Algorithm 2: Set the order of operations by minimizing the Extracts

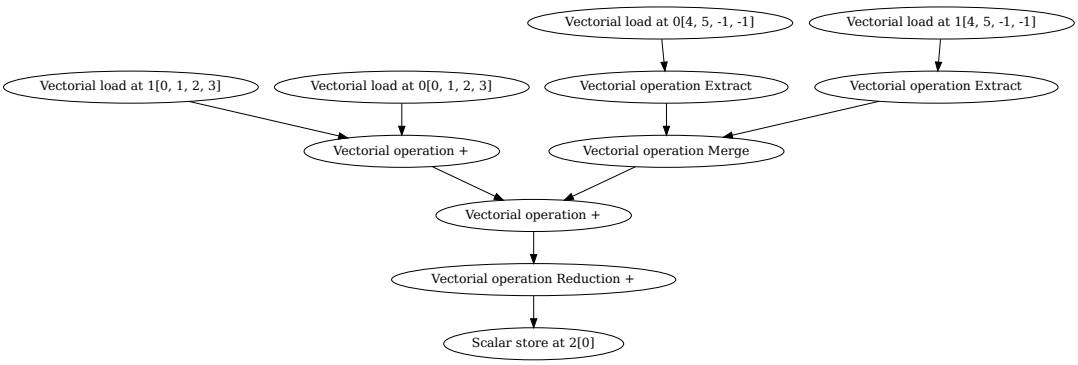
```
1  allNodes ← mapNodesWithIdNode();
   // Add nodes that have connections between each other
2  for node in allNodes do
3      for op in allOperations do
4          node.depIds ← addDepNode(op);
   // Connect nodes if they are linked by operations (including self-connect)
5  for op in allOperations do
6      for idx1 in op.depIds do
7          for idx2 in op.depIds do
               // Not the same index, so fixing one implies extracting to the other
8              if idx1 ≠ idx2 then
9                  ConnectDependantNodes();
   // Sort the nodes by number of constraints
10 sort(allNodes);
11 unusedPositions ← init();
12 unusedOperations ← init();
   // Fix order of operations that they don't need an Extract
13 for node in allNodes do
14     needExtractAnyway ← findIfNeedExtractAnyway();
15     if not needExtractAnyway then
16         fixPosition(unusedPositions, node.operation);
17         unusedPositions.erase(node.position);
18         unusedOperations.erase(node.operation);
   // Put operations that will need an Extract anyway in the vacant positions
19 while unusedPositions not empty do
20     fixPosition(unusedPositions, unusedOperations);
```

4.7 Prospecting for the best configuration using a greedy strategy

In the previous sections, we introduced various strategies for group partitioning, element ordering and optimization. However, some of these transformations are exclusive and cannot be used jointly, such as choosing between clustering and partitioning algorithms for operation group splitting. Similarly, only one of the split configurations of the loads/stores can be used. In addition, not all kernels benefit from reduction transformations, adding to the difficulty of predicting the best approach. To meet this challenge, we propose to test all the strategies with a greedy algorithm and select the one that produces the graph with the least number of instructions. We provide the final vectorial graph we obtain for KA and KB in Figure 9.



(a) KA



(b) KB

Fig. 9. Graphs of vectorial instructions for KA and KB for kernel size equals 6 and $VEC\_SIZE = 4$.

In cases where a kernel is complex, exhaustive testing of all transformation configurations may be impractical and time-consuming. However, users can guide our tool and explicitly configure the transformations. This option allows them to explore the search space selectively and thus save time. Determining the optimal configuration depends on the user's understanding of the problem and the specific characteristics of the kernel, such as potential reductions, contiguous load accesses, and the need for advanced group partitioning techniques. By considering these factors, users can effectively navigate the transformation options and improve the vectorization process. In summary, the greedy selection algorithm can be useful to further assist in identifying the most efficient transformation configuration.

4.8 Backend

Our backend takes as input the graph of vectorial instructions and generates a C++ program by converting each instruction into an intrinsic function call. The conversion is straightforward and no low-level optimization is performed at this stage. However, because we fully unroll and inline the kernels we can potentially obtain a significant code portion which can put pressure on the registers, leading to a register spilling. Therefore, to help the compiler that will compile the generated C++, we propose a reordering algorithm that aims at reducing the save/restore of intermediate variables.

To this end, we propose the Algorithm 3. It is a two-stage algorithm. In the first one, we find disjoint parts in the instruction list, i.e. parts of the graph that are not connected. Each of these parts is then treated separately in the second step. Our second step consists in iterating over the graph as if it was a graph of tasks. We maintain a list of ready tasks (the instructions that can be computed), select among them the task that will be computed next and then release the dependencies, with the possibility to move new tasks to the list of ready tasks. This approach offers several advantages. It has low complexity and does not require managing the correctness of moving instructions before/after the instructions they depend on (it prevents generating cycles). In addition, all the core part is then in the selection of the next task among the ones in the ready list.

Algorithm 3: Schedule a list of instructions l.

```
1  instructions ← ∅ ;
2  ready_list ← roots(l);
3  invdepth ← distance_from_leaves(l);
4  while ready_list is not empty do
5      costs ← compute_costs(ready_list);
       // Sort the instructions using their costs first and their invdepth if tie
6      ready_list ← sort(ready_list, costs, invdepth);
7      next_i ← ready_list.front;
8      instructions.insert(next_i);
9      ready_list.pop_front();
10     ready_list ← manage_dependencies(next_i);
11 return instructions;
```

Our strategy consists in sorting the list based on a register benefit and depth to the leaves of the graph. We present in the Algorithm 4 the computation of the cost of scheduling an instruction as the next one. The benefit is based on the number of registers that will be released if the instruction is computed. If we consider that there are no unused variables, a load or an operation has a cost of 1 because the result of the instruction will have to be stored somewhere. Then for each instruction, we iterate on its predecessors (input) and see what will be the gain of computing it. For this purpose, we do the sum of the average of the successors of the predecessors. We provide an example in Figure 10. We use the longest distance to the leaves to sort the instructions only when the benefit is equal between several tasks. Consequently, our algorithm has a local view of the graph and focuses on a minimum local.

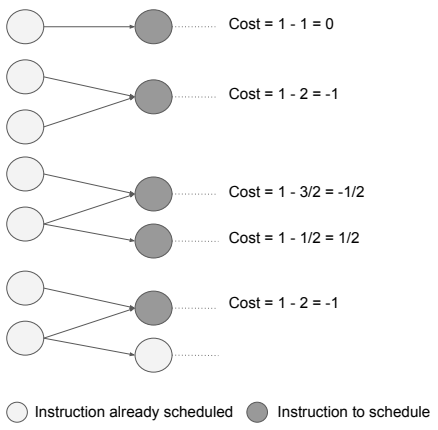


Fig. 10. Example of costs to schedule the instructions.

Algorithm 4: Compute the cost of scheduling instruction *i* next.

```
1 cost ← 0;
2 if i has successors then
      // We need a register to store the result
3     cost += 1;
4 for p in i.predecessors do
      // Considered p is done already
5     for sp in p.successors do
6         if sp is not done then
7             counter += 1;
8     cost -= 1/counter;
9 return cost;
```

5 PERFORMANCE STUDY

5.1 Configurations

We evaluate our tool on two platforms:

- INTEL-AVX512: is an Intel Xeon Skylake Gold 6240 with 2x 18 cores at 2.6GHz and 512-bit AVX, i.e. a vector can contain eight double floating-point values. The node has 192 GB memory and each core has 64KB private L1 cache, 1MB private L2 cache and 1.375MB private L3 cache. The OS is CentOS Linux release 7.6.1810 (Core). We use the GNU compiler 10.2.0, Clang the LLVM-based compiler 15.0.1 and icpx the Intel compiler 2022.0.0.
- ARM-SVE: is an ARMv8.2 A64FX - Fujitsu with 48 cores at 1.8 GHz and 512-bit SVE, i.e. a vector can contain eight double floating-point values. The node has 32 GB HBM2 memory arranged in four core memory groups (CMGs) with 12 cores and 8GB each, 64KB private L1 cache, and 8MB shared L2 cache per CMG. The OS is Red Hat 8.3.1-5. We use the GNU compiler 10.3.0 (20210408).

We compile both the scalar (original programs) and the vectorized ones (output of Autovesk) with the optimization flags `-march=native -mtune=native -O3 -ffast-math`. The executions are pinned to a single core using *taskset*.

5.2 Test cases

We use 10 custom computational kernels (Set-CK) and 4 common numerical kernels (Set-NK) to evaluate Autovesk's performance in vectorizing.

Set-CK kernels are specifically designed to cover a wide range of patterns. They include kernels with scalar input/outputs (denoted 1) and those with arrays (denoted *N*). The access patterns to the elements of the arrays vary and can be either contiguous, random or shifted (circular access). These kernels are in the form of for loops on data arrays. They represent a variety of scenarios, including linear contiguous access, irregular access, and circular access. Each of the 10 kernels in the set consists of two inputs and one output. Some of these kernels are expected to be vectorized well by modern compilers.

The Set-CK kernels are:

- (N,N) → N: $dest[i] = op(src0[i], src1[i]), \forall i \in [O, N[$.
  This kernel is similar to Blas *axpy*.
- (N,N) → 1: $dest+ = op(src0[i], src1[i]), \forall i \in [O, N[$.
  This kernel includes KB and is similar to a dot product.
- (N,1) → N: $dest[i] = op(src0[i], src1), \forall i \in [O, N[$
- (N,1) → 1: $dest+ = op(src0[i], src1), \forall i \in [O, N[$
- (r(N),N) → N: $dest[i] = op(src0[r(i)], src1[i]), \forall i \in [O, N[$
- (N,N) → r(N): $dest[r(i)]+ = op(src0[i], src1[i]), \forall i \in [O, N[$
- (r(N),N) → 1: $dest+ = op(src0[r(i)], src1[i]), \forall i \in [O, N[$
- (r(N),1) → N: $dest[i] = op(src0[r(i)], src1), \forall i \in [O, N[$
- (r(N),1) → 1: $dest+ = op(src0[r(i)], src1), \forall i \in [O, N[$
- (s(N),s(N)) → N: $dest[i] = op(src0[s(i)], src1[s(i)]), \forall i \in [O, N[$.
  This kernel includes KA.

The shift function $s(x)$ is defined as

$$s(x) = (x + 2) \pmod N. \tag{2}$$

The random access function $r(x)$ is defined as

$$r(x) = (x\ XOR\ 0x55555555) \pmod N. \tag{3}$$

A binary operation *op* is either a sum or a multiplication.

We evaluate each kernel from size equal to 4 to 128, and an increasing step equal to 4. An evaluation consists in executing a kernel on 21 distinct arrays 10000 times. Consequently, in a run, we use at most 21×3 arrays of 128 double floating point values, which takes $63KB$, such that the data fit in the L1 cache for both hardware configurations.

The kernels from Set-NK come from real applications:

- *bcucof* (size 4): *bcucof* routine [21] computes a table for bi-cubic interpolation. The bi-cubic algorithm is frequently used for scaling images and video for display.
- *bcucofx2f*: It is two consecutive calls to *bcucof* (size 4) on different data that are vectorized together.
- *weight* (size 7): *weight* routine [21] computes differentiation matrices for pseudo-spectral collocation, which are essential in spectral methods. Spectral methods are a powerful tool widely used for solving partial differential equations (PDEs) due to their high accuracy and efficiency.
- *convolution* (sizes 8 and 16) Discrete convolution is a mathematical operation that combines two discrete functions f and g to produce a new function f*g representing their combined effect. It is commonly used for filtering, smoothing and feature detection in various fields including audio and image processing.

For all the test cases, the data are initialized before calling the kernels, such that they are already in the cache before the first execution. When showing the speedup, we obtain it by comparing the scalar version automatically vectorized vs. the version vectorized by Autovesk both compiled with the same compiler.

5.3 Results

Set-CK. Our performance study compared the automatically vectorized C++ code generated by Autovesk with the automatic vectorization techniques of scalar kernels produced by the GCC compiler, the Clang LLVM-based compiler,

and the Intel icpx compiler. Figure 11 shows the execution times of the 10 kernels with different problem sizes for both the INTEL AVX512 and ARM-SVE configurations. The comparison is made between the vectorized code generated by Autovesk and the automatic vectorization of GCC compiler on scalar codes. The results show that vectorized kernels provide a noticeable speedup over scalar kernels, as indicated by the speedup shown at some points. In Figure 12, we provide a detailed breakdown of the number and types of graph nodes for the different kernels and problem sizes in our study. There are scalar graph nodes, including loads, stores and operations, and vectorial graph nodes for the generated vectorized code, including loads, stores, operations and data transformations. Importantly, the data transformation operations exist only for the vectorized kernels, as they apply to vectors. Although the number of resulting binary instructions may differ, it is highly correlated with the number of graph nodes shown. These results provide insights into the complexity of the vectorization process and the types of operations involved in generating efficient vectorized code from scalar code.

From the execution times, we observe that our approach provides a significant speedup for all kernels on both configurations, except for kernels $(N, 1) \rightarrow N$ and $(N, N) \rightarrow N$. For these kernels, the compiler is able to vectorize, such that our approach obtains very similar execution times. The kernels $(N, 1) \rightarrow 1$ and $(N, N) \rightarrow 1$ can also be vectorized easily, but it requires a reduction. We can see, Figures 11a and 11c, that our approach is faster for all sizes on ARM-SVE and after a fairly large size on INTEL-AVX512. When using random access ($r(N)$) or a shift ($s(N)$) our method can provide a significant speedup of 12. The execution times then can vary and we can understand this behavior by looking at Figure 12. For instance, for the kernel $(r(N), 1) \rightarrow 1$, execution time Figure 11f and operations Figure 12f, we can see that the data transformation operations vary significantly depending on the size. This is because we do not always use a multiple of 8 (the length of the SIMD vector), which leads to different best instruction graphs. This effect is visible in Figures 12a, 12g and 12c but with less impact on the performance. As expected for the kernel $(N, N) \rightarrow N$, Figure 12d, there are no data transformation operations. Among the strategies to fix the order of the vector's elements, 76% of the best configurations were obtained by leaving the elements in their original order, which is not surprising due to the types of kernels we test. Then, 16% were obtained with the partitioning strategy and 8% with the clustering strategy.

Figure 13 shows the speedups achieved by Autovesk compared to the Intel icpx and Clang compilers for the 10 kernels selected in our study. To evaluate performance, we ran each kernel as we did for the comparison with GCC, i.e. with different problem sizes, but instead of showing all the details, we calculated the average speedup along with the minimum and maximum variation over the automatic vectorization techniques of the compilers. The evaluation showed that the performance of the vectorized kernels using Autovesk is above the compiler versions for most kernels. For the cases "(N,N) → N" and "(N,1) → N" using Intel icpx, see Figure 13b, we obtain similar performance, which is what we expect since the data accesses are contiguous and linear. However, Clang did not deliver the same performance, see Figure 13a.

Set-NK. Figure 14 provides the results for the numerical kernels. Our study highlights the relative advantages of Autovesk over modern compilers in optimizing the numerical computations presented. In particular, Autovesk achieves significant speedups over Clang and Intel compilers, exceeding the speedups achieved over GCC. For the "bcucof" kernel, Autovesk runs 1.8 times faster than GCC. It also outperforms Clang LLVM by 3.8x with an execution time of 0.0182. Autovesk achieves a remarkable 2x speedup over GCC for the "bcucofx2f" kernel. Compared to Intel icpx, Autovesk achieves a significant speedup of 4.2x, reducing the execution time to 0.0134. Autovesk performs well on the "weight" kernel and excels on the "convolution" kernel with speedups of 2.1x over GCC, 3.6x over LLVM and 2.9x over

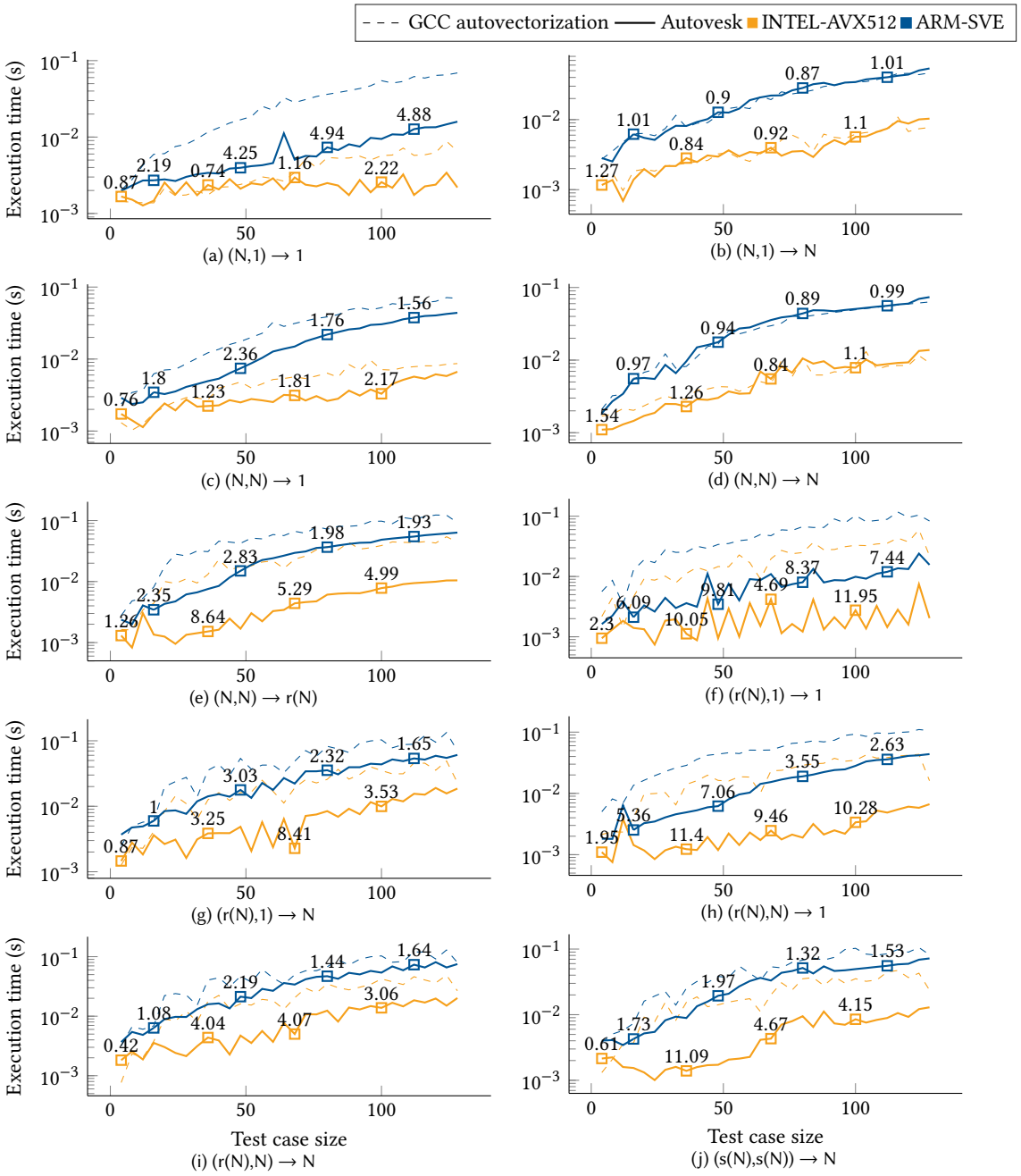


Fig. 11. Execution times for the different kernels and different problem sizes for the INTEL-AVX512 and ARM-SVE configurations. The speedup of the vectorized kernels by Autovesk against the automatic vectorization techniques of GCC compiler on scalar kernels is shown for some points.

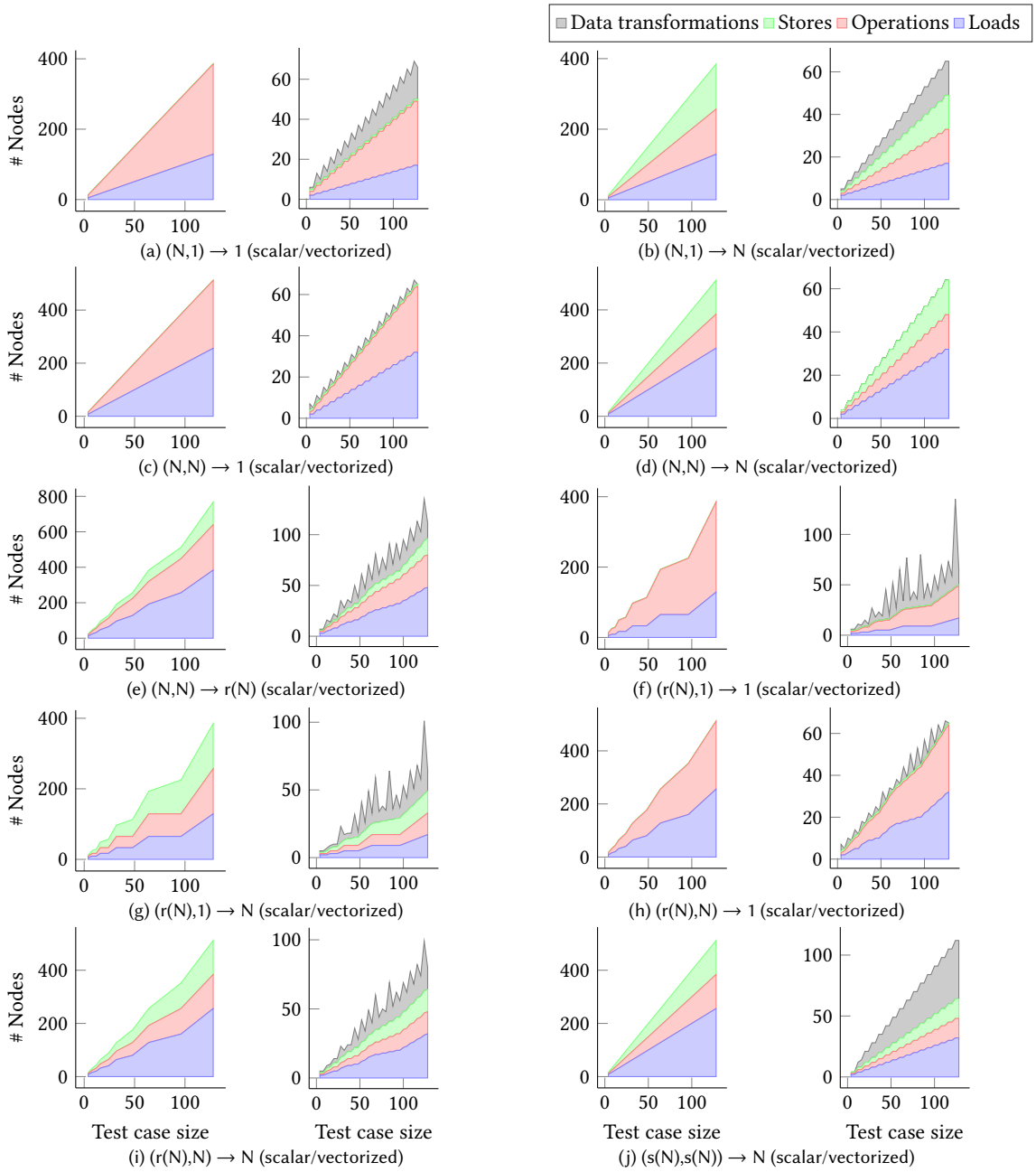
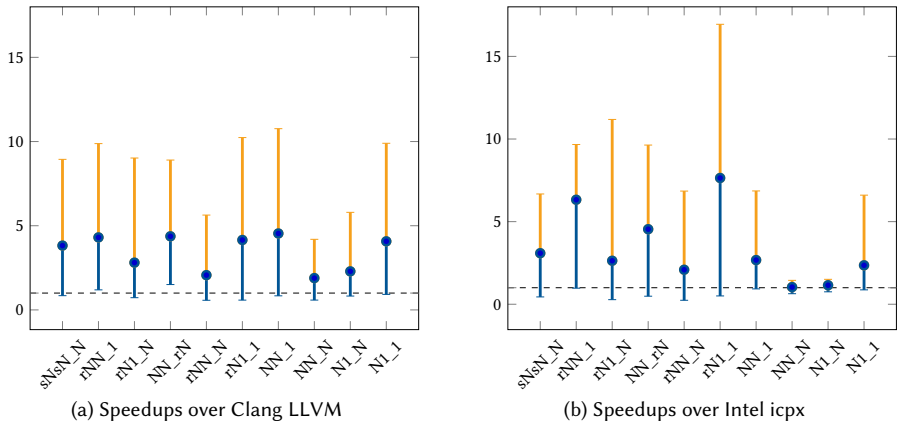


Fig. 12. Number and types of graph nodes for the different kernels and different problem sizes. The data transformation operations only exist for the vectorized kernels because they apply to vectors. The number of resulting binary instructions can differ but will be highly correlated to the numbers shown.



(a) Speedups over Clang LLVM (b) Speedups over Intel icpx

Fig. 13. Comparison of Autovesk's automatic vectorization speedups against the Clang LLVM-based and Intel icpx compilers on 10 selected kernels. Each kernel was executed with different problem sizes. The minimum, average and maximum speedup over the compilers' auto-vectorization techniques are shown.

Intel icpx, reducing the execution time to 0.0276. These consistent and significant performance improvements highlight the effectiveness of Autovesk in optimizing numerical kernels. Further investigation is required to explore its potential in broader application domains.

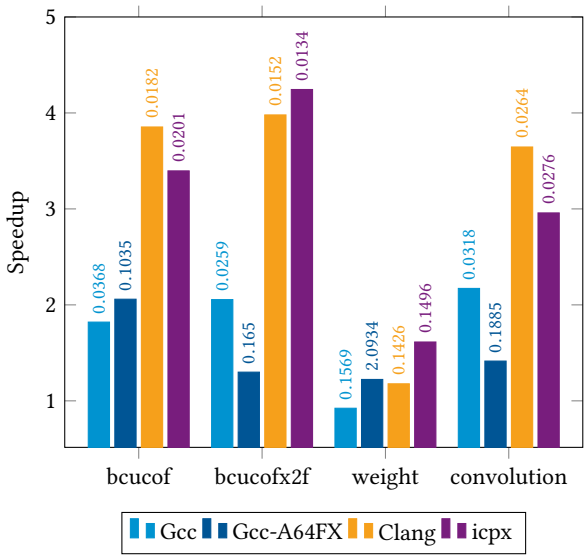


Fig. 14. Comparison of Autovesk's automatic vectorization speedup against the GNU compiler (GCC) on x86 architecture and A64FX 64-bit ARM architecture, Clang LLVM-based compiler and the Intel (icpx) compiler for 4 different numerical kernels. We provide the execution times of the Autovesk versions above each bar.

6 STACK/REGISTER EXCHANGE MINIMIZATION

To alleviate the pressure on the registers and minimize stack/register exchange, we proposed a reordering algorithm for the generated C++ code in Section 4.8.

6.1 Test cases

To evaluate the effectiveness of our proposed reordering algorithm, we generated random instructions using our test case generator, Pred*X*. This generator creates random instructions that have dependencies between variables. The *X* in Pred*X* represents, for a given variable, the maximum number of predecessors. To generate the test cases, we start with a single variable and link it to previous variables with a jump between 1 and 10, meaning that the first 9 variables could have zero predecessors. A variable with zero predecessors is considered a load and a variable with zero successors is considered a store. We generated AVX512 code, where each variable is an AVX512 vector, and simply added the input. This allowed us to test the performance of our reordering algorithm on randomly generated code of varying complexity. Our test case generator is referred to as Pred*X*, where *X* means that a variable is computed using from 1 to *X* variables (i.e. in the dependency graph, a variable can have up to *X* predecessors).

6.2 Results

Using our test case generator Pred*X*, we evaluated the effectiveness of our proposed reordering algorithm by generating random instructions for two different test case configurations: Pred*4* and Pred*10*, while varying the problem sizes. We provide the results in Figure 15, showing the number of accesses to the stack (push, pop or direct accesses) with and without the reordering algorithm. We can observe that reordering the instructions with our strategy always provide a benefit. More precisely, for Pred*4*, the compiler can avoid using the stack for all problem sizes but will do 68 registers spinning with the original order. For Pred*10*, the gain is around 15%.

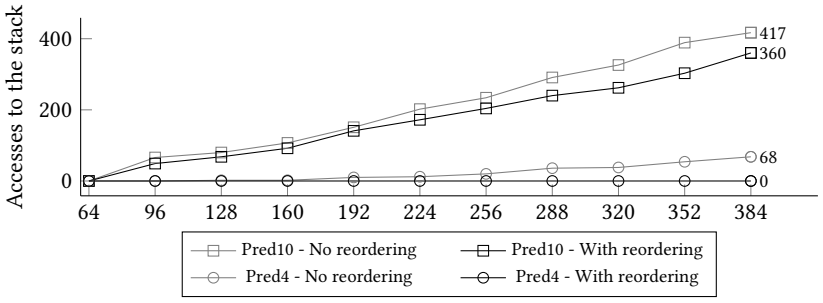


Fig. 15. Number of accesses to the stack (push, pop or direct accesses) for different test case sizes. Pred*X* refers to a test case where a new variable is constructed using up to X other variables as input.

7 LIMITATIONS AND FUTURE WORKS

Our research has focused on automatically vectorizing unstructured static kernels with irregular data access patterns, but there are several limitations and opportunities for future improvement. One area we intend to address is extending the applicability of Autovesk to cases where non-static loops can be expressed as repetitions of static ones. Indeed, non-static kernels can be divided into those that can be described by the polyhedral model, for which our approach is

not beneficial and those that cannot. For the latter, there are non-static kernels that require alternative optimization methods, and those that can be viewed as repetitions of static kernels, where we aim to apply our method. Although our tool automatically generates vectorized code, at this stage, it still requires the user to provide input code in a specific format, as we are using C++ templates to extract the graph of instructions. To eliminate user intervention and extend the range of codes that can be automatically vectorized, we plan to integrate Autovesk's transformations into an existing compiler. In addition, we can investigate incorporating a cost model for each strategy in the graph transformations. By considering the potential benefits and overheads associated with vectorization, we can make informed decisions and avoid unnecessary vectorization where it may not provide significant performance gains. These directions aim to improve Autovesk's usability and effectiveness in more applications.

8 CONCLUSION

We introduced Autovesk, an automatic vectorization tool designed for complex computational kernels. We presented our strategy that relies on heuristic-based algorithms to avoid paying the price of finding an optimal solution. Our method generates several solutions and returns the one with the lowest number of operations. To assess the vectorization performance of Autovesk, we a comparative evaluation against three leading compilers: GCC, LLVM, and Intel, using a set of proposed kernels, including both custom and real-world numerical kernels. We demonstrate that Autovesk can provide a significant speedup on several kernels. It remains competitive even in cases where general compilers can vectorize. Furthermore, Autovesk includes a procedure to reorder the instructions to reduce register spilling. While Autovesk shows promising results for the automatic vectorization of unstructured static kernels, there is still room for improvement and extension. Our ongoing research will continue to explore these possibilities.

ACKNOWLEDGMENTS

This work used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, which is an EPSRC project (EP/T022078/1). We also used the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (https://www.plafrim.fr).

REFERENCES

[1] Sara S. Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. 2016. FlexVec: Auto-Vectorization for Irregular Loops. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 697–710. https://doi.org/10.1145/2908080.2908111

[2] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. 2004. Putting Polyhedral Loop Transformations to Work. In *Languages and Compilers for Parallel Computing*, Lawrence Rauchwerger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–225.

[3] Berenger Bramas. 2017. Inastemp: A novel intrinsics-as-template library for portable simd-vectorization. *Scientific Programming* 2017 (2017), 1–18.

[4] Bérenger Bramas and Pavel Kus. 2018. Computing the sparse matrix vector product using block-based kernels without zero padding on processors with AVX-512 instructions. *PeerJ Computer Science* 4 (April 2018), e151. https://doi.org/10.7717/peerj-cs.151

[5] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[6] Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. 2022. Vectorizing Sparse Matrix Computations with Partially-Strided Codelets. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41404.2022.00037

[7] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 82–93. https://doi.org/10.1145/996841.996853

[8] Joël Falcou and Jocelyn Serot. 2004. Application of template-based metaprogramming compilation techniques to the efficient implementation of image processing algorithms on SIMD-capable processors. In *Advanced Concepts for Intelligent Vision Systems*.

[9] Michael J. Flynn. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.* C-21, 9 (1972), 948–960. https://doi.org/10.1109/TC.1972.5009071

[10] Matthias Gross. 2016. Neat SIMD: Elegant vectorization in C++ by using specialized templates. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*. 848–857. https://doi.org/10.1109/HPCSim.2016.7568423

[11] Nabil Hallou, Erven Rohou, and Philippe Clauss. 2017. Runtime Vectorization Transformations of Binary Code. *International Journal of Parallel Programming* 8, 6 (June 2017), 1536 – 1565. https://doi.org/10.1007/s10766-016-0480-z

[12] Peng Jiang and Gagan Agrawal. 2018. Conflict-Free Vectorization of Associative Irregular Applications with Recent SIMD Architectural Advances. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 175–187. https://doi.org/10.1145/3168827

[13] Peng Jiang, Linchuan Chen, and Gagan Agrawal. 2016. Reusing Data Reorganization for Efficient SIMD Parallelization of Adaptive Irregular Applications. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/2925426.2926285

[14] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[15] Roland Leißa, Immanuel Haffner, and Sebastian Hack. 2014. Sierra: A SIMD Extension for C++. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing* (Orlando, Florida, USA) *(WPMVP '14)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/2568058.2568062

[16] Ralf Möller. 2016. Design of a low-level C++ template SIMD library. *Computer Engineering Group, Bielefeld University: Bielefeld, Germany* (2016).

[17] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-Vectorization of Interleaved Data for SIMD. *SIGPLAN Not.* 41, 6 (jun 2006), 132–143. https://doi.org/10.1145/1133255.1133997

[18] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. 2006. GRAPHITE: Polyhedral analyses and optimizations for GCC. *Proceedings of the GCC Developers' Summit 2006*.

[19] Vasileios Porpodas, Rodrigo C. O. Rocha, Evgueni Brevnov, Luís F. W. Góes, and Timothy Mattson. 2019. Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 206–216. https://doi.org/10.1109/CGO.2019.8661192

[20] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. 2018. Look-Ahead SLP: Auto-Vectorization in the Presence of Commutative Operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/3168807

[21] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing* (3 ed.). Cambridge University Press, USA.

[22] Rodrigo C. O. Rocha, Vasileios Porpodas, Pavlos Petoumenos, Luís F. W. Góes, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Vectorization-Aware Loop Unrolling with Seed Forwarding. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3377555.3377890

[23] Ira Rosen, D. Nuzman, and A. Zaks. 2007. Loop-aware SLP in GCC. *GCC Developers' Summit* (01 2007), 131–142.

[24] P Souza, L Borges, C Andreolli, and P Thierry. 2015. OpenVec portable SIMD intrinsics. In *Second EAGE Workshop on High Performance Computing for Upstream*, Vol. 2015. European Association of Geoscientists & Engineers, 1–5.

[25] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721

[26] Majedul Haque Sujon, R. Clint Whaley, and Qing Yi. 2013. Vectorization past dependent branches through speculation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 353–362. https://doi.org/10.1109/PACT.2013.6618831

[27] Aravind Sukumaran-Rajam and Philippe Clauss. 2015. The Polyhedral Model of Nonlinear Loops. *ACM Trans. Archit. Code Optim.* 12, 4, Article 48 (dec 2015), 27 pages. https://doi.org/10.1145/2838734

[28] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-Model Guided Loop-Nest Auto-Vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 327–337. https://doi.org/10.1109/PACT.2009.18

[29] Haichuan Wang, Peng Wu, Ilie Gabriel Tanase, Mauricio J. Serrano, and José E. Moreira. 2014. Simple, Portable and Fast SIMD Intrinsic Programming: Generic Simd Library. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing* (Orlando, Florida, USA) *(WPMVP '14)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/2568058.2568059

[30] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-Iteration. *ACM Trans. Archit. Code Optim.* 20, 1, Article 16 (dec 2022), 26 pages. https://doi.org/10.1145/3566054

A PARTITIONING STRATEGY

Algorithm 5: Converting a group into a set of V sub-groups, where each group has at most VEC_SIZE elements.

```
   // Given a list of elements e and a matrix score d
   // We start with one partition containing all the elements
1  wip_partitions.push(new_partition(e)) ;
2  final_partitions ← ∅ ;
3  while |final_partitions| ≠ V do
4      || p ← wip_partitions.pop() ;
5      if |p| ≤ VEC_SIZE then
6          final_partitions.insert(p);
7          continue;
       // We compute the desired partition sizes
8      vecs_in_p ← (|p| + VEC_SIZE − 1)/VEC_SIZE;
9      margin ← vecs_in_p ×VEC_SIZE − |p|;
10     size_max_p1 ← (|p|/2) × VEC_SIZE ;
11     size_max_p2 ← (|p|/2) × VEC_SIZE ;
12     size_min_p1 ← size_max_p1 - margin ;
13     size_min_p2 ← size_max_p2 - margin ;
       // Find the elements in p with the lowest score
14     || (i, j) ← find_min_score(p , d);
15     || sub_p1 ← i;
16     sub_p2 ← j;
17     p.remove(i, j);
18     while |p| ≠ 0 do
19         if |sub_p1| == size_max_p1 then
20             sub_p2.insert(p.pop());
21         else if |sub_p2| == size_max_p2 then
22             sub_p1.insert(p.pop());
23         else
               // Find the element n that has the highest accumulated score either with sub_p1 or sub_p2
24             || [n, score1, score2] ← find_highest_score(p, sub_p1, sub_p2);
25             p.remove(n);
26             if score1 ≥ score2 OR (score1 == score2 AND size_max_p1 − |p1| ≥ size_max_p2 − |p2|) then
27                 sub_p1.insert(n);
28             else
29                 sub_p2.insert(p.pop());
30     wip_partition.insert(sub_p1, sub_p2);
```

B CLUSTERING STRATEGY

Algorithm 6: Converting a group into a set of V sub-groups, where each group has at most VEC_SIZE elements.

```
// Given a list of elements e and a matrix score d
// We initiate the sub-groups
subgroups[V] ← ∅ ;
|| (i,j) ← find_min(d);
e ← remove(e, i, j);
subgroup[0].add(i);
subgroup[1].add(j);
|| for v from 2 to V do
    worst_idx ← 0;
    worst_score ← +inf;
    for idx in e do
        score ← ∑_{i=0}^{v} d(subgroup[i][0], idx);
        if score < worst_score then
            score = worst_score;
            worst_idx = idx;
    e ← remove(e, worst_idx);
    subgroup[v].add(worst_idx);
|| while e is not empty do
    best_idx ← 0;
    best_subgroup ← 0;
    best_score ← −inf;
    for v from 0 to V do
        for idx in e do
            score ← ∑_{i=0}^{v} d(subgroup[i][:], idx);
            if score > best_score then
                score = best_score;
                best_idx = idx;
                best_subgroup = v;
    e ← remove(e, best_idx);
    subgroup[best_subgroup].add(best_idx);
```