Guiding Polyhedral Scheduling for Vectorization through Constraints Generated from an SLP Algorithm

Tom Hammer
Stéphane Genaud
Vincent Loechner
ICube, Université de Strasbourg, and Inria CAMUS
Illkirch, France

Abstract

Polyhedral schedulers present well established techniques to extract parallelism, improve data locality, and generate tiled code for statically analyzable loops. However, as the polyhedral model abstracts programs in a mathematical representation detached from language, architectural, and hardware specific constraints, encoding vectorization in an affine form can prove challenging.

In this paper, we present an approach to integrate information on vectorization decisions made by an SLP algorithm (Autovesk) into a polyhedral compiler (Pluto) through the addition of constraints to the schedule. We execute the SLP vectorization algorithm preserving annotated statement instance information. From its output, we create a set of constraints aiming to enforce vectorization. These optional constraints are injected during the scheduling process of the polyhedral compiler. We evaluate the performance and make use of hardware counters to check the relevancy of our method on the Polybench/C suite.

ACM Reference Format:
Tom Hammer, Stéphane Genaud, and Vincent Loechner. January 28th, 2026. Guiding Polyhedral Scheduling for Vectorization through Constraints Generated from an SLP Algorithm. In *Proceedings of 16th International Workshop on Polyhedral Compilation Techniques (IMPACT).* ACM, New York, NY, USA, 9 pages.

1 Introduction

The polyhedral model enables the representation of computational kernels made up of statically analyzable nested loops presenting static control parts through mathematical abstractions. The representation is composed of several elements, aiming to completely describe a program through a set of parametrized polyhedra. More precisely, individual statements within a program made up of nested loops with static control parts (SCoPs) are represented by their domains, memory accesses, and scheduling, from which additional properties such as data dependencies can be extracted. This mathematical representation can then be manipulated to produce transformations on the initial program that can be the result of explorative composition of transformations [9], or from the formulation and solving of Integer Linear Programming (ILP) problems [7].

Since many computational applications used in research contain computational kernels that can be represented in the polyhedral model, this approach has remained a leading method for automatic program optimization since it was introduced.

Today, there are several widely used implementations of schedulers using the polyhedral model. Their benefits include the extraction of thread level parallelism and data locality optimization, like in Pluto [6], and transformations for GPU execution, as in PPCG [22]. The implementation of such scheduling algorithms relies on other tools for input program parsing [2, 23], internal representation [3, 21], or code generation [1]. Polyhedral schedulers can be implemented in a high-level source to source workflow, or directly into low-level compilers, through the use of Intermediate-Representations (IR) [8, 14]. In both cases, kernels are scheduled into a transformed program, and further optimizations such as vectorization are often delegated to the back-end compiler, sometimes guided by the polyhedral scheduler. In this paper, we will present an approach to integrate insights on vectorization opportunities generated from an SLP algorithm into a polyhedral scheduler through the addition of constraints to the ILP formulation.

All recent architectures offer some implementation of vector instructions. To exploit these instruction sets, vectorization algorithms are being implemented in modern compilers. These algorithms can be divided into two categories:

- Loop vectorizers. When looking at programs structured by loops, it is possible to perform a dependency analysis between iterations of the same statement. If deemed possible, the loop is then stripmined by the available vector length, and consecutive iterations are grouped into vector instructions. This approach enables coarse-grained optimizations when there are no dependencies between consecutive iterations of a loop, but is limited by the fact it attempts only to group instances of the same instruction together. This approach is used in most polyhedral compilers when attempting to extract vectorization. In Polly [8] loops that are trivially vectorizable are emitted as vector instructions, and in Pluto [6] vector directives are generated whenever the innermost loop can be parallelized.

- Superword Level Parallelism (SLP) vectorizers. In the case of more complex programs, grouping instances of the same instruction may be difficult due to dependencies or to the structure of the program. To tackle this, another family of vectorizers performs optimizations on sequences of instructions. SLP vectorizers [11, 13, 16, 17, 19] aim to group scalar instructions that perform arithmetic operations into the corresponding vector instructions by packing. Here, values may need to be gathered/scattered in vector registers before executing a vector instruction. This results in an average lesser gain in execution time than loop-vectorizers because of the overhead of gathering/scattering values, but can enable vectorization where loop vectorizers fail.

In practice, both types of vectorizers are implemented within compilers, with a loop-based vectorizer performing a first coarse-grained generation of vector instructions. Then, an SLP algorithm is run on the remaining scalar instructions. Such a system is implemented in, e.g., GCC and Clang [18].

The main reason leading SLP vectorizers to be implemented alongside loop vectorizers within compilers is scalability. While SLP vectorizers can extract more vectorization opportunities, executing them on large programs can lead to long compilation times. As they take as input a sequence of instructions, all the control flow must be removed from the input program. In practice, this means that we need to unroll all loops of a program before executing such an algorithm, leading to a large number of instructions. This, in turn, leads to a combinatorial explosion, with complexity growing exponentially with the number of instructions in the original program. Instead, the loop vectorizers perform all possible transformations within their framework, and the remaining scalar instructions are handled by an SLP algorithm. Furthermore, the optimizations produced by SLP vectorizers are dependent on the fixed parameters of the program. Being able to generalize such transformations in a parametrized context through the polyhedral model would alleviate the computational constraints of straight-line vectorizers. Such an approach was theorized by Bastoul et al. [4].

As mentioned above, polyhedral schedulers are well tailored to extract thread level parallelism, but either rely on the algorithms implemented within compiler vectorization passes, or can only extract loop vectorization. As such, transformations generated by polyhedral techniques are mostly agnostic to low-level optimizations. Let us consider the trisolv kernel from the PolyBench/C [25] as shown in fig. 1. The kernel, when scheduled with Pluto as shown in fig. 1b, presents dependencies on its innermost loop and therefore cannot be vectorized. Instead, if we schedule the code as shown in fig. 1c, interchanging the i and j accesses, we obtain a loop that can be vectorized along the new j dimension.

```
for (i = 0; i < N; i++)
  x[i] = b[i];                    //S1
  for (j = 0; j < i; j++)
    x[i] -= L[i][j] * x[j]      //S2
  x[i] = x[i] / L[i][i]         //S3
```

(a) Original code

```
for (i = 0; i < N; i++)
  x[i] = b[i]
x[0] = x[0] / L[0][0]
for (i = 1; i < N ; i++)
  for (j = 0; j < i; j++)
    x[i] -= L[i][j] * x[j]      //S2
  x[i] = x[i] / L[i][i]
```

(b) Code generated by Pluto

```
for (i=0; i < N; i++)
  x[i] = b[i]
for (i = 0; i < N-1 ; i++)
  x[i] = x[i] / L[i][i]
  for (j = i+1; j < N; j++)
    x[j] -= L[j][i] * x[i]      //S2
x[N-1] = x[N-1] / L[N-1][N-1]
```

(c) Code generated by our approach

Figure 1. Example: trisolv from the PolyBench/C suite

We propose an approach to integrate information extracted from the result of an SLP algorithm to constrain the solution space of a polyhedral scheduler. As mentioned above, SLP algorithms are hardly scalable, and as such, we execute a vectorizer on kernels parametrized with small problem sizes. The results are then converted into scheduling constraints for a polyhedral compiler, with the aim of generalizing the transformation to a parametrized model. We will present the approach and evaluate the relevant metrics with the aim of transforming programs to generate more vector instructions once compiled. In carrying out this study, this work brings the following contributions:

- The extension of an SLP vectorizer to be able to track how vectorized instructions relate to the original statement instance instructions.
- An algorithm to select the preferred vectorization dimensions for each statement.
- A modified version of the Pluto algorithm with extra constraints to take into account the vectorization dimension when possible.

- An analysis of how prioritizing vectorization in Pluto's schedule modifies execution time performance, which we experiment on the PolyBench/C benchmarks.

The rest of the paper is organized as follows. In section 2, we present the context and related work. In section 3, we explain our method to detect the vectorization opportunities and how it can be integrated into the Pluto scheduling process. Section 4 is devoted to the evaluation, and section 5 to the discussion of the results.

2 Context & Related work

2.1 Polyhedral schedulers

There exist many implementations of polyhedral schedulers ranging from source to source applications [6, 24], runtime systems [12] and implementations within compilation flows [8, 14]. In the general landscape of polyhedral schedulers, Pluto has remained a state of the art optimizer, enabling tiling and loop-fusion heuristics through its set of constraints. It also provides a comprehensive library to implement further modifications. Hence, we have chosen it as a base for our implementation, with the objective to preserve its benefits.

2.2 Modeling vectorization within polyhedral schedulers

Beyond general purpose polyhedral compilers, some work has focused on integrating additional scheduling constraints into the ILP formulations specifically for vectorization. The first way to add additional constraints is through objective functions, representing properties of the program, that are then minimized or maximized when solving the ILP. Such an approach has been proposed by Kong. et al. [10] and enforces parallelism on the innermost loop while maximizing stride-0/1 references. Another approach, proposed by Zinenko et al. [26], includes objective functions to model temporal and spatial locality as separate objective variables while including non-linear decisions on the ordering of constraints. Trifunovic et al. [20] proposes a modeling of the impact of loop transformations on vectorization through a cost model, used to direct loop transformations.

Bastoul et al. [5] expand on these ideas by abstracting transformations in the form of a tree encoding explorative constraint injection and relaxation. The framework supports objective functions, as well as simple linear constraints generated from non-linear optimizers, in this case a cost function modeling load/store vectorization, strides, and thread distribution for GPU.

While our approach shares similarities with these previous projects, it differs in some key aspects. First, we delegate decisions on vectorization to an SLP optimizer. Furthermore, the constraints built from the output of the SLP algorithm only model vectorization, and take the form of simple linear constraints instead of objective functions. Finally, we only enforce additional constraints within the scope of the existing ILP formulation and do not modify the other constraints.

2.3 SLP algorithms

As mentioned in the previous section, SLP algorithms work by aggregating instructions into groups to form vector instructions. There exists a number of algorithms in this family, stemming from the work of Larsen and Amarasinghe [11], and several other implementations have been developed including LSLP [17], taking into account commutative operations, and SN-SLP [16], considering the inverse of arithmetic operations. goSLP [13] provides cost-modeling of vector packing operations through ILP formulations. An SLP algorithm is included in LLVM/Clang within the vectorization passes, inspired by the implementation present in GCC [18]. However, this implementation can prove difficult to work with as the SLP algorithm works conjunctly with a loop vectorizer, rendering isolation of the SLP pass tedious.

2.4 Autovesk

Autovesk [19] is a stand-alone auto-vectorizer project that fits into the SLP family. Autovesk works by aggregating all instructions that perform the same operation and that are independent with respect to each other. It eliminates the consideration of variables and control through its representation as a flow of operations stemming from loads and leading to stores. The programs are represented through graphs of instructions generated by templates and operator overloading, this effectively removes all control flow. Nodes in the graph, representing operations, are aggregated when they perform the same operation and are independent. The aggregated nodes can contain any number of operations, and are subsequently split along the vector size using a cost model aiming to minimize the total number of nodes in the graph. While Autovesk has been tested and yielded good results on simple and short kernels, it has not been tested on more complex polyhedral benchmarks. It also suffers from a high computational complexity which prevents its use on large programs. However, we will see that this limitation does not hinder our approach. In the following work, we have used Autovesk for SLP vectorization, taking advantage of the extensibility it offers through its template-based design.

3 Approach

The proposed approach can be split into three steps/components:

1. A modified version of the Autovesk vectorization algorithm that enables tracking of original statement instance information through instruction annotations;
2. A decision algorithm that selects preferred vectorization dimensions and generates configuration files identifying the preferred dimension for each statement;

3. A modified version of the Pluto algorithm with added constraints to the ILP formulation, directed by the configuration files.

3.1 Using Autovesk to generate program traces

Because Autovesk creates its instruction graphs by running the kernels, generating a node at each operation encountered, it effectively removes all the control flow from an input program. Because of the complexity issue mentioned in section 1, and to keep execution times low, we run the vectorizer on a reduced with all parameters set to eight. As we want to formulate constraints for a polyhedral scheduler, we want to retain some information about the original loops during the vectorization process. A statement in a loop nest is executed several times, and the polyhedral model treats these executions as occurrences, or instances, of the same instruction at different points in the iteration space. To store information about the original control flow, we annotate the nodes corresponding to each instance of a statement through additional overloading of Autovesk's templates. Each statement instance is marked with its corresponding statement number and iterator values. It is possible that a statement, as considered when analyzing the source code with a Scop extraction tool, may comprise several arithmetic operations. When this is the case, we attach the annotations on the last executed operation.

Once all statements have been annotated, we run the vectorization algorithm, and the annotations are propagated to the generated vector nodes from the scalar nodes. In the end, we obtain a graph of vector instructions, with some of the nodes including annotations on the original instructions of the input program. Instead of generating intrinsics, as originally implemented in Autovesk, we only output the annotated statement instances, which yields a vectorized execution trace, as shown in fig. 2a. We see iterator values and statement numbers corresponding to statement instances of the original program. Instances that are vectorized in the same vector instruction are shown enclosed by `Vec Node` and `End` comments.

3.2 Extracting intra-node information on vectorization decisions

Autovesk's output takes the form of a directed acyclic graph, whose nodes represent arithmetic, vector, or memory operations, and edges encode dependencies. This graph only provides a partial order on the execution of the instructions of the program. When generating intrinsics, the last step of Autovesk is to schedule the graph using a simple algorithm minimizing register load. Since we are only interested in vectorization opportunities, i.e. the innermost scheduling dimension, we will only take into account the intra-node information, and stop Autovesk before this step. To this end, we propose a simple algorithm that selects the preferred vectorization dimension.

```
// Vec node 1:
S1 5 1 0
              S1: [+1, +0, +0]
S1 6 1 0
              S1: [+1, +0, +0]
S1 7 1 0
              S1: [+0, +1, +0]
S1 0 2 0
// End
// Vec node 2:
S2 1 0
              S2: [+1, +0]
S2 2 0
S1 3 1 0
              S1: [+1, +0, +0]
S1 4 1 0
// End
```

(a) Sample trace

```
S1: [3, 1, 0]
S2: [1, 0]
```

(b) Sum of increments for each dimension

```
S1
1 0 0
S2
1 0
```

(c) Configuration file generated

Figure 2. Generation of configuration files from program traces

To illustrate the algorithm, let us consider the example in fig. 2. Figure 2a shows a sample partial trace of an arbitrary program, containing two vector nodes made up of two statements S1 and S2. For every consecutive instance of the same statement within a vector node, we count the occurrences of increments by one for each iterator. In this case, the first vector node presents two instances of an increment in the first iterator and one instance in the second iterator. For the second node, there is an increment of one in the first dimension of S2 and S1. We add the total number of increments for each statement dimension across all nodes, as shown in fig. 2b. The dimension with the highest count, i.e. the one that is the most frequently recurring in the vector nodes is selected to be the vectorization dimension for that statement. Figure 2c shows the configuration file generated by the algorithm, a value of one in a dimension indicates that we will try to vectorize it for this statement, as explained in the next subsection.

The extraction algorithm selects at most one dimension per statement. This constitutes a naïve approach, as it does not take into account potential interleaving of the statements within the vector nodes. Considering that we only want to direct the polyhedral scheduling process through simple constraints, and that complex interleaving patterns may not be expressible by linear constraints, we keep this simple representation with the objective that the other Pluto constraints will improve data locality and parallelism.

3.3 Generating constraints with Pluto

The Pluto algorithm works by iteratively solving an ILP formulation comprising a set of constraints, namely the legality constraint and the volume bounding constraint. The same ILP formulation is solved several times while adding linear independence constraints. This creates a permutable band of several scheduling dimensions representing loops that can be interchanged one with another. Once there are no longer any linearly independent solutions to the ILP, satisfied dependencies are removed from the problem and the process starts over. If there is no solution to the problem, a heuristic operates a splitting of the dependency graph into its strongly connected components.

Based on the configuration file generated in the previous step, we want to ensure that the preferred dimension for vectorization is scheduled as the innermost loop. To ensure this, we can add the following constraint to the ILP formulation at each step.

Let us consider a single statement S of dimensionality m, its schedule θ_S can be written as follows:

$$\begin{aligned} \theta_S(\vec{i_S}) = (c_1^S, c_2^S, c_3^S, \ldots, c_m^S)(\vec{i_S}) + c_0^S \\ \vec{i_S} \in \mathbb{Z} \end{aligned} \quad (1)$$

where the c^S are the scheduling coefficients of S. If dimension k is constrained by our configuration file, we enforce:

$$c_k^S = 0 \quad (2)$$

for the first $m-1$ linearly independent solutions to the ILP formulation. This ensures that the last coefficient in the schedule for S will present a component in the desired dimension k.

Due to the way we generate the configuration files, the preferred dimension for vectorization can prove impossible to enforce, for example when many nodes remain scalar due to dependencies. This is for instance the case with the seidel-2d kernel as shown in fig. 3.

Here, the only vector nodes generated by Autovesk appear between iterations of the outermost loop (as shown in fig. 3b), the other statement instances remaining scalar. The last iteration of the `i,j` loops for a given `t` is packed with the first `i,j` iteration at `t+1`. The algorithm that generates the configuration files treats this as a preferred vectorization dimension in `t` as shown in fig. 3c. When trying to apply this configuration file to all iterations of this statement, Pluto is unable to solve the ILP, as dependencies prevent finding of any valid schedule.

When encountering this problem, we apply a constraint relaxation algorithm. When Pluto yields no solution because the generated ILP cannot be solved, we remove the vectorization constraints one by one, in the order of the statements, until a solution is found. Once a solution is found, constraints

```
for (t = 0; t <= T_STEPS - 1; t++)
  for (i = 1; i <= N - 2; i++)
    for (j = 1; j <= N - 2; j++)
      A[i][j] = (A[i-1][j-1]+A[i-1][j]+
        A[i-1][j+1]+A[i][j-1]+A[i][j]+
        A[i][j+1]+A[i+1][j-1]+A[i+1][j]+
        A[i+1][j+1])/SCALAR_VAL(9.0);
```

(a) Seidel-2d kernel

```
// Vec node:
S1 0 4 1
S1 1 1 4
// End
S1 1 2 2
S1 2 1 1
S1 0 2 6
S1 0 3 4
// Vec node:
S1 0 4 2
S1 1 1 5
// End
```

(b) Partial trace generated by Autovesk

```
S1
1 0 0
```

(c) Generated configuration file

Figure 3. Configuration file generation for the Seidel-2d benchmark

that were relaxed are reintroduced until there is no longer any solution again. This solution converges towards a good approximation of the maximal number of constraints we can add, while eliminating those that render the ILP solution empty.

4 Evaluation

4.1 Experimental setup

We ran our experiments on an Intel 12th gen i7-12700H processor, with 6 P-Cores and 8 E-cores and 64GB of RAM. Our experiments were run on a single thread mapped to a P-Core, with 80KB, 1.25MB and 24MB of L1, L2 and L3 cache sizes, respectively. The largest vector instruction available set is AVX2.

The approach was evaluated on the kernels from the Polybench/C, excluding nussinov and deriche, which uses operations that cannot be modeled in Autovesk.

As mentioned in section 2.4, running Autovesk on large kernels is untractable due to its excessive complexity. Our objective being the generalization of non-parametrized transformations, we run the vectorizer on kernels with very small

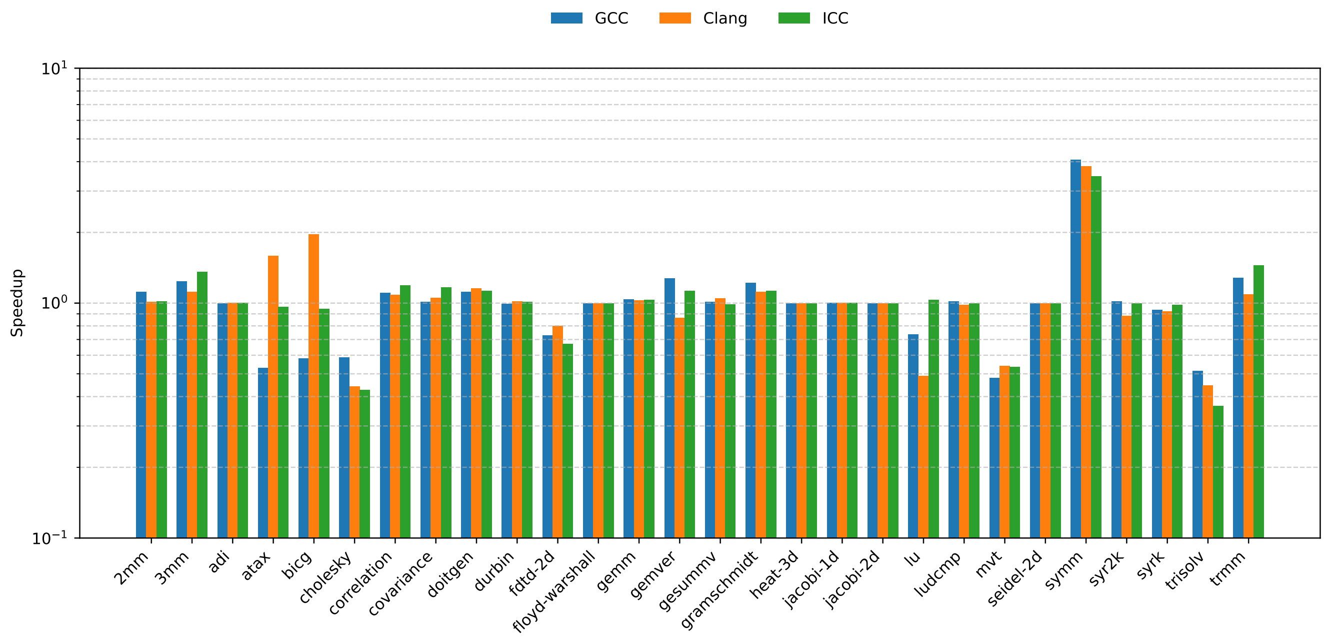

Figure 4. Sequential speedup of Pluto-vec over Pluto

problem sizes. We set all parameters of the kernels to 8, which, given a desired vector size of 4, guarantees that statements need at least two vector nodes per iteration dimension to be completely scheduled.

After generating the configuration files, we ran Pluto with and without the configuration files using the options `-noparallel -tile -nounrolljam -prevector -smartfuse` to generate scheduled kernels for comparison between our approach and vanilla Pluto.

We used a modified dataset, with sizes adjusted up from the EXTRALARGE dataset in order to obtain execution times greater than a second. The kernels were run using double precision floats. We compiled with Clang, GCC and ICC using the options `-O3 -march=native -mprefer-vector-width=256 -ffast-math`. The cache is flushed before running each benchmark. The time measurement utility packaged with PolyBench was used to gather execution times, running each benchmark five times, eliminating the lowest and highest, and averaging the three remaining measurements.

To gather additional information on vector and scalar arithmetic operations, cache misses, and loads/stores, we used the perf-cpp [15] library that enables profiling on specific parts of programs through wrappers on kernel functions.

4.2 Results

The speedups of our modified Pluto algorithm with additional constraints compared to the unmodified version are shown in fig. 4. For clarity, we will differentiate between the base version of Pluto and call our implementation Pluto-vec. When running the schedulers on our benchmark suite, two kernels, adi and ludcmp could not be scheduled by either version of Pluto and fell back to the original schedule. Eight kernels produce the same schedule in Pluto-vec as Pluto without added constraints. Namely: durbin, floyd-warshall, gesummv, heat-3d, jacobi-1d, jacobi-2d, seidel-2d, and syr2k. We will analyze the eighteen remaining kernels where the code generated by Pluto-vec differs from Pluto.

Let us examine the number of arithmetic vector operations generated for the kernels as shown in fig. 5. We observe that for cholesky and trisolv, the Pluto-vec transformed code does enable GCC to generate a significant number of vector instructions, while Pluto does not. However, despite this vectorization, no speedup is observed on these kernels. In other benchmarks, the number of vector instructions generated remains the same but yields contrasted results regarding execution time speedup. For instance, for mvt which no compiler can vectorize, Pluto-vec transformed code shows about half the performance of the Pluto optimized code. By constrast, we get a 4x speedup on symm without any change in the number of arithmetic vector operations executed. We observe that, in this experiment, the number of vector arithmetic operations is not directly correlated with the program performance.

To delve further into this analysis, the gains and losses in performance can be explained by looking at other metrics, namely cache misses and loads/stores. If we look at the

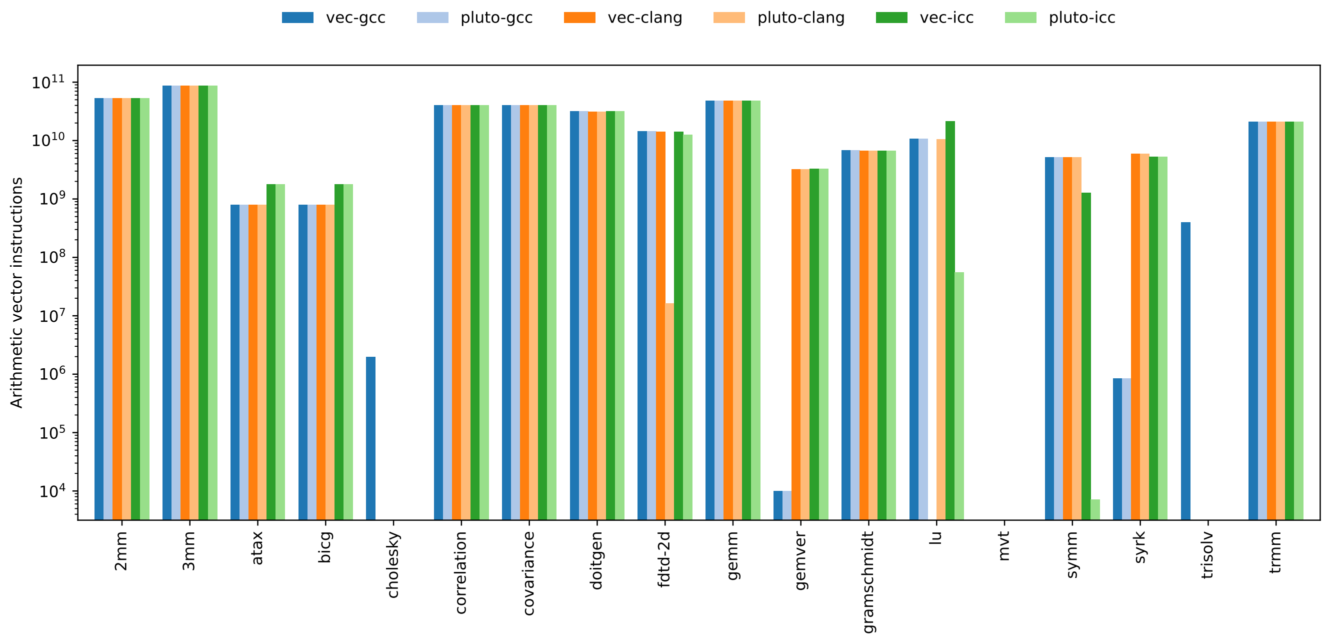

Figure 5. Number of arithmetic vector operations generated

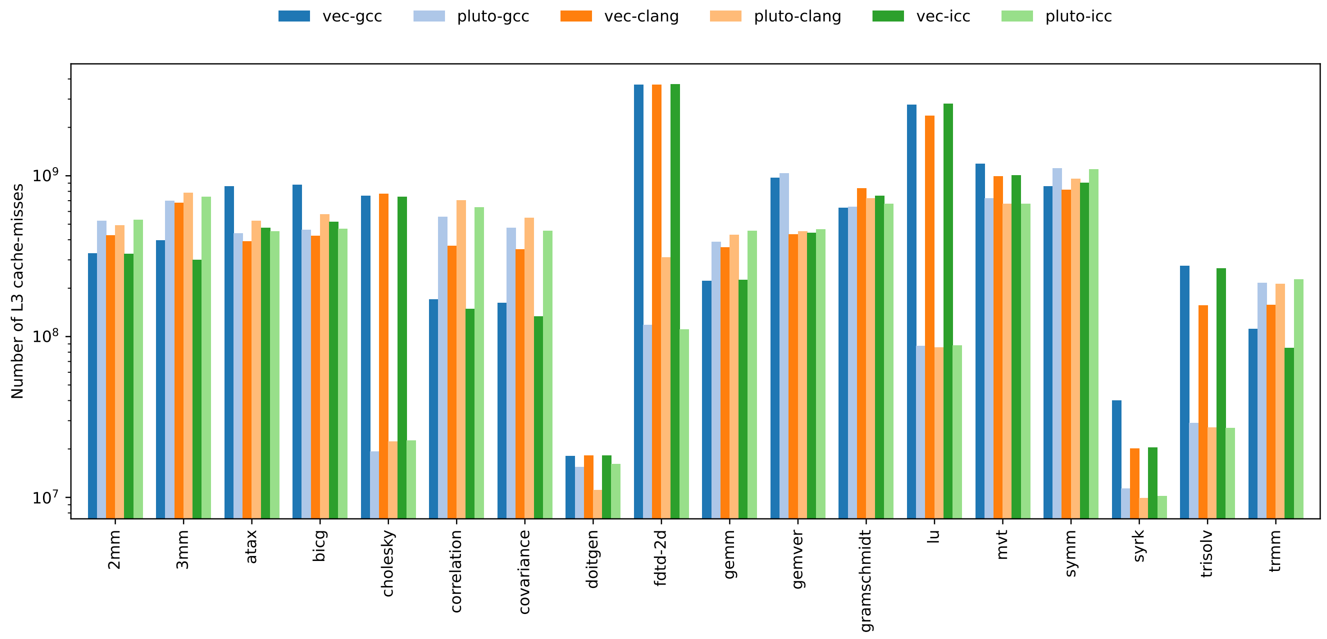

Figure 6. Number of cache misses

number of cache misses as shown in fig. 6, we observe that cache misses are often correlated to performance. Kernels producing fewer cache misses with Pluto-vec yield speedups while those producing more yield slowdowns. Above a certain threshold, cache misses negatively impact execution time. This is for instance the case in cholesky and trisolv, where our approach yields vectorization but the produced schedule leads to a higher cache miss count.

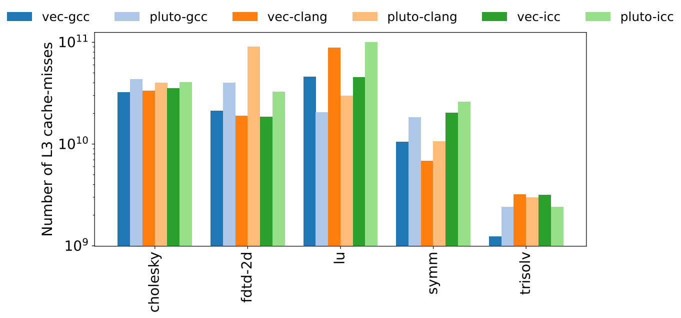

Figure 7. Number of loads and stores

From our performance analysis, we can also extract the number of loads and stores. As there are no separate hardware counters for vector and scalar memory operations, we can only infer a higher vectorization count if the number of loads and stores varies between Pluto and Pluto-vec. Figure 7 shows the data on five kernels that present significant differences in total memory operations. cholesky and trisolv present a lower count of memory operations for GCC, which is consistent with the previous results on the number of vector operations considering we have enabled vectorization for those kernels. symm performs fewer memory operations across all compilers with our approach, which, considering it also produces fewer cache misses, explains the speedup. In lu we generate fewer memory operations with ICC, which is consistent with the number of arithmetic vector operations generated. In fdtd-2d we generate more memory vector operations but because our transformed code incurs a very high number of cache misses, the overall effect is still a slowdown.

5 Discussion

Overall, our approach improves the number of vector instructions generated on many kernels. It enables the generation of more arithmetic vector operations or more memory vector operations, or both. However, as shown in the previous section, the number of vector instructions generated is not directly correlated with performance gains. On the eighteen kernels analyzed in the previous section, ten yield a speedup and eight result in a slowdown. And the remaining kernels yield the same schedule as Pluto.

Even when Pluto-vec enables compilers to generate more vector instructions in the final compiled code, losses in data locality can outweigh the benefits. Vectorizing statements along dimensions that produce cache-misses lead to performance drops. Adding additional constraints to the scheduling dimensions with the goal of improving vectorization can disrupt data locality. As data locality is not taken into account by our approach, we obtain speedups where Pluto can preserve it and slowdowns where the added constraints hinder it.

In the trisolv example from section 1, shown in fig. 1c, our Pluto-vec transformation enables vectorization on the `j` dimension, but, as our results show, this does not improve performance. The reason is that the inner instruction reads one value from a different row of array `L` each time, potentially requiring the invalidation and reloading the corresponding cache line at each iteration of the `j` loop, hence voiding vectorization benefits.

6 Conclusion & Perspectives

This paper introduces a way of adding schedule constraints inferred from the results of a low level vectorization algorithm into the constraint system of polyhedral schedulers.

We have demonstrated its efficiency in terms of increase in the number of an vector instructions generated when compiling the output of polyhedral schedulers. However, the benefits of vectorization can be outweighed by losses in data locality. Also, our approach does not ensure preservation of the benefits on data locality resulting from Pluto's constraints.

In some cases, our approach yields the same results as Pluto, which does not directly enforce vectorization during the scheduling process. Instead, it relies on post-processing of the detected permutable loops. This means that our additional constraints were able to enforce decisions improving vectorization during the iterative scheduling process, and not at a post-processing step once the schedule has already been found.

Going further in this work, the approach would benefit from the implementation of finer-grained injected constraints. These constraints could better represent the vector packing found by SLP algorithms, for example presenting scalar dimensions enforcing potential interleaving of the statements through fusion of the innermost loops.

Furthermore, our approach only takes into account vectorization and, as we have shown, other factors such as cache locality or memory layout also influence the performance. We aim to expand our constraint generation algorithm to take into account such factors.

References

[1] Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*. Juan-les-Pins, France, 7–16.

[2] Cédric Bastoul. 2014. CLAN - A Polyhedral Representation Extraction Tool for C-Based High Level Languages. http://icps.u-strasbg.fr/~bastoul/development/clan/index.html.

[3] Cédric Bastoul. 2014. OpenScop - A Specification and a Library for Data Exchange in Polyhedral Compilation Tools. http://icps.u-strasbg.fr/~bastoul/development/openscop/index.html.

[4] Cédric Bastoul, Alain Ketterlin, and Vincent Loechner. 2023. Superloop Scheduling: Loop Optimization via Direct Statement Instance Reordering. In *IMPACT 2023*. Toulouse, France. https://inria.hal.science/hal-04393522

[5] Cedric Bastoul, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Adilla Susungi, Javier de Juan, Etienne Filhol, Baptiste Jarry, Gianpietro Consolaro, and Renwei Zhang. 2022. Optimizing GPU deep learning operators with polyhedral scheduling constraint injection. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 313–324. doi:10.1109/CGO53902.2022.9741260

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. doi:10.1145/1375581.1375595

[7] Paul Feautrier. 1997. Some efficient solutions to the affine scheduling problem Part II Multidimensional time. *International Journal of Parallel Programming* 21 (01 1997). doi:10.1007/BF01379404

[8] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Process. Lett.* 22, 4 (2012). https://doi.org/10.1142/S0129626412500107

[9] W. Kelly and W. Pugh. 1995. A unifying framework for iteration reordering transformations. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, Vol. 1. 153–162 vol.1. doi:10.1109/ICAPP.1995.472180

[10] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël 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* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 127–138. doi:10.1145/2491956.2462187

[11] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not.* 35, 5 (May 2000), 145–156. doi:10.1145/358438.349320

[12] Juan Manuel Martinez Caamaño, Aravind Sukumaran-Rajam, Artiom Baloian, Manuel Selva, and Philippe Clauss. 2017. APOLLO: Automatic speculative POLyhedral Loop Optimizer. In *IMPACT 2017 - 7th International Workshop on Polyhedral Compilation Techniques*. Stockholm, Sweden, 8. https://inria.hal.science/hal-01533692

[13] Charith Mendis and Saman Amarasinghe. 2018. goSLP: globally optimized superword level parallelism framework. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 1–28. doi:10.1145/3276480

[14] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 45–59. doi:10.1109/PACT52795.2021.00011

[15] Jan Mühlig. 2025. *perf-cpp: Effortless Hardware Performance Monitoring for C++ Applications*. https://github.com/jmuehlig/perf-cpp Accessed: 2025-12-04.

[16] 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. doi:10.1109/CGO.2019.8661192

[17] 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 '18)*. Association for Computing Machinery, New York, NY, USA, 163–174. doi:10.1145/3168807

[18] Ira Rosen, Dorit Nuzman, and Ayal Zaks. 2007. Loop-aware SLP in GCC. 131–142. GCC and GNU Toolchain Developers' Summit 2007.

[19] Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. *ACM Trans. Archit. Code Optim.* 21, 1, Article 4 (Dec. 2023), 25 pages. doi:10.1145/3631709

[20] 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. doi:10.1109/PACT.2009.18

[21] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model, Vol. 6327. 299–302. doi:10.1007/978-3-642-15582-6_49

[22] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. 9, 4, Article 54 (jan 2013), 23 pages. doi:10.1145/2400682.2400713

[23] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. doi:10.13140/RG.2.1.4213.4562

[24] Sven Verdoolaege and Gerda Janssens. 2017. *Scheduling for PPCG*. Technical Report. Department of Computer Science, KU Leuven. doi:10.13140/RG.2.2.28998.68169

[25] Tomofumi Yuki and Louis-Noël Pouchet. 2016. PolyBench/C 4.2. https://sourceforge.net/projects/polybench/.

[26] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the conflicting demands of parallelism and Temporal/Spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC '18)*. Association for Computing Machinery, New York, NY, USA, 3–13. doi:10.1145/3178372.3179507