Towards Optimising Programs with Sketch-Guided Polyhedral Compilation

Valeran Maytié
University of Strasbourg
France
maytie@unistra.fr

Reuben Carolan
University of Edinburgh
Edinburgh, United Kingdom

Christophe Alias
Inria, CNRS, ENS de Lyon, UCBL
Lyon, France

Cedric Bastoul
University of Strasbourg
France

Thomas Kœhler
ICube Lab - CNRS
University of Strasbourg - France
thomas.koehler@cnrs.fr

$$\text{In} : A[i] : \{0 \leq i < N\}$$
$$B[i] : \{0 \leq i < N-2\} := \frac{A[i] + A[i+1] + A[i+2]}{3}$$
$$\text{Out} : C[i] : \{0 \leq i < N-4\} := \frac{B[i] + B[i+1] + B[i+2]}{3}$$

Figure 1. Blur 1d specification

```
1 for i {  2
    4 for j {
        5 B[!i + j]; 5
      } 4
    3 C[!i]; 3
    2
  } 1
```

Figure 2. Blur 1d sketch

```
for i in {0 ≤ i < N − 4} {
    for j in {0 ≤ j ≤ 2}
        B[i + j] = (A[i + j] +
                    ...) / 3;
    C[i] = (B[i] + ...) / 3;
}
```

Figure 3. Blur 1d result

Abstract

When programmers use semi-automatic compilers, they typically write optimisation scripts, to guide the compiler towards key optimisations. Instead of writing scripts, it may be preferable to write *sketches* that focus on the desired structure of the optimised code, without worrying about individual transformations.

In this article, we present a new semi-automatic, sketch-guided compilation approach. We introduce a sketch language that enables expressing the result of imperative loop transformations and a new polyhedral algorithm capable of generating code constrained by both a sketch and a computation specification.

1 Introduction

Contemporary advances in image processing, physical simulation, and artificial intelligence demand enormous computational power [2]. Meeting these performance requirements has driven the development of highly optimised programs.

Compilers such as GCC, Clang, and PluTo implement a wide range of optimisations, yet their effectiveness remains limited. In practice, these optimisations often rely on fragile heuristics, which can prevent the discovery and application of transformations that would yield substantial performance improvements [8].

Early attempts to address this limitation without resorting to fully manual optimisation produced semi-automatic approaches. In these systems, users typically guide the compiler through scripting languages that explicitly specify the transformations to apply [1, 6]. However, these scripts must enumerate the transformations directly, making them difficult to write, and reason about [7].

An alternative approach to writing scripts is to write sketches. Sketches enable the programmer to outline the overall structure of the desired program while leaving certain details unspecified, to be completed automatically. In the context of program synthesis for stencils, sketching was introduced by [4]. However, to generate imperative code for relatively small stencil kernels, this approach takes minutes [7]. More recently, sketches have also been explored in the context of functional languages, in combination with automatic term rewriting techniques [3].

In this article, we present a new approach to semi-automatic compilation that combines sketch-guided optimisation with polyhedral methods. Using polyhedral methods allows representing iteration domains and efficiently perform dependency computations, taking advantage of existing polyhedral libraries such as PolyLib or isl .

This work consists of preliminary work towards two major contributions:

1. A new **sketching language** that enables expressing the result of imperative loop transformations.
2. The first **algorithm** completing sketches using polyhedral methods. It takes as input a Systems of Affine Recurrence Equations (SARE) specification of the requested computation and a sketch that constrains the shape of the desired output program. It generates a program that respects the computation specification and the desired sketch.

	Iteration domain	Validation Domain: d_v	Needed: d_n
① = ② = ③		$(), () \rightarrow \{\}$	$\vdash C[k] : 0 \leq k < N-4$
3	$(i), () \rightarrow \{0 \leq i < N-4\}$	"	$i \in \{0 \leq i < N-4\} \vdash B[k] : i \leq k \leq i+2$
④ = ⑤		"	"
5	$(i,j), () \rightarrow \{0 \leq i < N-4 \wedge 0 \leq j \leq 2\}$	"	$\emptyset$
4	$(i), () \rightarrow \{0 \leq i < N-4\}$	"	"
2	$(i), () \rightarrow \{0 \leq i < N-4\}$	"	"
1	$(), () \rightarrow \{\}$	"	"

Table 1. blur 1d, sketch completion

2 Example: Blur 1d

In this section, we detail an example that generates code for the blurring of a one-dimensional array on two stages. This example has the particularity of showing that our algorithm is capable of generating code that performs redundant computations.

To specify the desired computations, we chose to represent them in a functional form, known in the polyhedral community as Systems of Affine Recurrence Equations (SARE) [9]. We chose this representation because it provides a simpler abstraction to manipulate than raw C code and because we care about values more than instructions. In Figure 1, we define the blurring computation for a one-dimensional array A of size N, that consists of computing the average of three surrounding cells. The blur is performed on two stages, first blurring the array A in the array B of size $N-2$ and then reblurring the array B in the array C of size $N-4$.

Our objective is to generate code that computes $B[i]$, $B[i+1]$, and $B[i+2]$ within the same iteration of i. This approach, however, leads to recomputing overlapping portions of B multiple times. To the best of our knowledge, such redundant computations are challenging to implement using conventional automatic polyhedral compilers.

The corresponding code generation sketch is shown in Figure 2, where B is computed inside a `for` loop over j. The purpose of this loop is to allow computing several values of B. Exclamation marks are used to force the indices to be exactly those specified by the user. Without them, the algorithm may introduce existential variables that permit implicit shifts.

Our sketch completion algorithm performs a recursive traversal from the bottom of the sketch to the top in order to propagate all the computations that are needed. In Figure 2, we annotate the sketch with numbers that indicate the order in which the algorithm traverses it; the circles represent the input to each constructor, and the square represents the return value. We have detailed the states of the algorithm of our current examples in the Table 1.

For each constructor, the algorithm returns structured imperative code that corresponds to the code described by the sketch and the specification, as well as an *iteration domain* that represents the iteration domain of the free variables of the returned code. This means that the free variables of the returned code must take all the values in this iteration domain in order to perform the required computations. We write this domain in the following form: $(v_0, \ldots, v_i), (c_0, \ldots, c_j) \rightarrow \{C\}$, where C is a set of linear constraints. Semantically, this corresponds to the set: $\{\exists c_0, \ldots, c_n,\ \forall v_0, \ldots, v_m.\ C\}$

The required computations are recorded in the *needed domain*. This set associates each array identifier from the specification with a computation domain that must be executed to respect the dependency order between arrays. Unlike the other two domains, the needed domain includes a context, which allows local computations to be expressed relative to the surrounding loop structure. We denote this domain as $\Gamma \vdash B[k] : \{C\}$, where Γ is the needed context, B is the array being computed, k is the index variable, and C is the set of associated constraints.

The algorithm also uses a *validation domain* that allows constraints to be accumulated on the existential variables that can be instantiated. In this example, this domain is not used. This domain has the same representation as the iteration domain.

At the beginning of the algorithm, we start by recording the arrays requested in the needed domain. In this example, our objective is to compute array C, so, first we add the necessary domain to the set of calculations we need :

$$\vdash C[k] : 0 \leq k < N-4.$$

In step 3, we compute $C[i]$ for i ranging from 0 to $N-4$ excluded. For each iteration of i, the computation requires the values $B[k]$ with k between i and $i+2$. This yields the needed domain:

$$i \in \{0 \leq i < N-4\} \vdash B[k] : i \leq k \leq i+2.$$

In step ⑤, we want to generate a function f that allows us to calculate the iteration domain of $B[i+j]$, so that the code is correct. To do this, we just need to replace k with $i+j$ without changing the context, so we generate the function:

$$f(i,j) = (i,\ i+j).$$

By computing its preimage, we obtain the iteration domain at line 5 of the of the array in Figure 1, so we do indeed have $i+j$ between i and $i+2$.

```
fn complete(s, spec) → CPoly::Stmt {
  let d_v = (), () → {};
  let d_n = spec.out_to_needed();

  let st, _ = complete_rec(s, d_v, d_n, spec);
  Seq(EVar(d_v.evar, d_v.dom), st)
}

fn complete_rec(s, d_v, d_n, spec) → (Stmt, D_i) {
  match s {
      Arr(id, li) ⇒ c_arr(id, li, d_b, d_n, spec),
      Seq(s1, s2) ⇒ c_seq(s1, s2, d_b, d_n, spec),
      For(fv, sf) ⇒ c_for(fv, sf, d_b, d_n, spec),
  }
}
```

Figure 4. Sketch completion algorithm

At the end of step [2], we want to generate correct code for the sequence. To do this, we union the iteration domains returned by steps [3] and [4]. Since the domains are equal, there is nothing more to do. In other cases, if conditions may need to be generated.

At the end of the algorithm execution, we find the code given in the Figure 3.

3 Algorithm

The entry point of the algorithm is defined in Figure 4. It is the function `complete`, which takes a sketch `s` and a specification `spec` as input parameters—these are the two inputs of the problem described in the previous section. The function initialises two variables: validation domain d_v and needed domain d_n, which represent the state of the algorithm. Once initialised, we call the sketch traversal function `complete_rec`, Which takes the two states as parameters and returns a program `st`, associated with an iteration domain over which all free variables of `st`, must iterate.

Sequence generation The code generation for sequences `s1; s2` is in the function `c_seq(s1, s2, ...)`. The main objective is to propagate the needed computations from bottom to top. To achieve this, we begin by generating the code for `s2`, which will update both the needed domain d_n and validation domain d_v. We can now complete `s1` using the updated set of needed domain. At the end of these two recursive calls, we obtain `st1` and `st2`, which are the codes corresponding to `s1` and `s2`, along with the domains `d1` and `d2`, representing the iteration domains of `s1` and `s2`. If the two domains do not match, we add `if` statements to be able to union the two iteration domains.

For loop generation To complete a sketch that generates a for loop: `for fv { sf }`, we use the function: `c_for(fv, sf, ...)`. We start by generating the code for the body of the loop by adding fv to the domains used by the algorithm. Then, we simply retrieve the iteration domain from traversing the body to obtain the constraints on the loop index. At the end of the function, we return the for loop with the iteration domain that contains only of the constraints on fv. The iteration domain returned is the one returned by the body of the loop, removing the dimension corresponding to the variable fv.

Array computations generation The code for generating the array computations `A[f0]...[fn]` is implemented in the function `c_arr(A, li, ...)`. The objective is to generate the iteration domain, which allows us to specify the domain of variables over which we must iterate in order to compute all the indices of the array we need. To do this, we retrieve the domain from the needed domain and construct a function using the indices described in the sketch. Then, we generate the imperative code by replacing the indices in the specification with those given in the sketch. The algorithm will also generate existential variables such as, $A[i]$ becoming $A[i + s]$ for some s if the indices are not forced (with an exclamation mark).

Once the sketch has been completed and the needed domain is empty, the existential variables generated by the sketch are removed using the `EVar` constructor, which allows constants constrained by the validation domain to be declared. As these variables are constrained by a domain, they can therefore have several values, so we can use heuristics to choose the best values to assign to them. Conversely, if the domain is empty, then the sketch is not valid, as it does not comply with the constraints of the dependencies.

4 Conclusion

This paper lays the foundations towards optimizing programs with sketch-guided polyhedral compilation. We introduce a new sketch language for imperative programs and an algorithm that leverages polyhedral tools to generate code consistent with both the sketch and the computation specification. To make this new technique usable, our next plan is to generate executable C code, and to measure the performance of the generated code on the Polybench benchmark suite [5]. We also plan to develop a frontend to improve usability, enabling both specifications and sketches to be written in a Python-based DSL.

Acknowledgments

This work benefited from government funding managed by the National Research Agency under France 2030 via the ENACT AI Cluster (ANR-23-IACL-0004).

References

[1] Lénaıc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening polyhedral compiler's black box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016.* Björn Franke,

Youfeng Wu, and Fabrice Rastello, (Eds.) ACM, 128–138. doi:10.1145/2854038.2854048.

[2] Paul Barham and Michael Isard. 2019. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*. ACM, 177–183. doi:10.1145/3317550.3321441.

[3] Thomas Kœhler, Phil Trinder, and Michel Steuwer. 2021. Sketch-guided equality saturation: scaling equality saturation to complex optimizations in languages with bindings. (2021). arXiv: 2111.13040 [cs.PL].

[4] A Solar Lezama. 2008. *Program synthesis by sketching*. Ph.D. Dissertation. PhD thesis, EECS Department, University of California, Berkeley.

[5] Louis-Noël Pouchet and Tomofumi Yuki. 2022. PolyBench/C version 4.3.1. https://sourceforge.net/projects/polybench/.

[6] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31, 4, 32:1–32:12. doi:10.1145/2185520.2185528.

[7] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bod'ık, Vijay A. Saraswat, and Sanjit A. Seshia. 2007. Sketching stencils. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Jeanne Ferrante and Kathryn S. McKinley, (Eds.) ACM, 167–178. doi:10.1145/1250734.1250754.

[8] Arun Thangamani, Vincent Loechner, and Stéphane Genaud. 2024. A survey of general-purpose polyhedral compilers. *ACM Trans. Archit. Code Optim.*, (June 2024). Just Accepted. doi:10.1145/3674735.

[9] Tomofumi Yuki, Vamshi Basupalli, Gautam Gupta, Guillaume Iooss, D Kim, Tanveer Pathan, Pradeep Srinivasa, Yun Zou, and Sanjay Rajopadhye. 2012. Alphaz: a system for analysis, transformation, and code generation in the polyhedral equational model. *Colorado State University, Tech. Rep.*