Rephrasing Polyhedral Optimizations with Trace Analysis

Hugo Thievenaz, Keiji Kimura, Christophe Alias

Rephrasing Polyhedral Optimizations with Trace Analysis

Hugo Thievenaz
Laboratoire de l'Informatique du Parallélisme
CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon
Lyon, France
Hugo.Thievenaz@inria.fr

Keiji Kimura
Department of Computer Science and Engineering
Waseda University
Tokyo, Japan
keiji@waseda.jp

Christophe Alias*
Laboratoire de l'Informatique du Parallélisme
CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon
Lyon, France
Christophe.Alias@inria.fr

ACM Reference Format:
Hugo Thievenaz, Keiji Kimura, and Christophe Alias. 2022. Rephrasing Polyhedral Optimizations with Trace Analysis. In *Proceedings of 12th International Workshop on Polyhedral Compilation Techniques (IMPACT'22).* ACM, New York, NY, USA, 3 pages.

1 Introduction

Compilers must restructure the application to use as best as possible computing and storage resources. In general, compiler optimizations are fragile and highly depend on the shape of the source code. The polyhedral model [5, 6, 9, 10] provides the mathematical foundations to develop *domain-specific* compiler optimizations focusing on compute-intensive scientific kernels with affine loop kernels manipulating arrays. If polyhedral optimizations are precise and powerful, they often *lack of scalability*. For instance, max-fusion loop tiling [4] freezes when applied directly to large loop nests, like those implementing deep learning algorithms [7]. Also, array contraction [1] do not scale when applied to HLS, where hundreds of buffers needs to be allocated [2]. An attentive analysis show that most of the time is spent in static polyhedral operations – projections, parametric ILP, subtractions or combinations of piecewise affine mappings.

In this paper, we defend the iconoclast idea that polyhedral optimizations might be computed *without those operations*, simply by applying a *lightweight analysis on a few off-line execution traces*. The main intuition being that, since polyhedral transformation are expressed as affine mappings, only a few points are required to infer the general mapping. Our hope is to compute those points from a few off-line execution traces thanks to a lightweight algorithms. We focus on array contraction, a well known technique to reallocate temporary arrays thanks to affine mappings $A[\vec{i}] \mapsto \hat{A}[\sigma_A(\vec{i}, \vec{N})]$ so the array size is reduced; and whose scalability needs to be improved. Our contributions to this problem are the following:

- We outline an off-line trace analysis to infer a contraction mapping of the form $\sigma_A(\vec{i}, \vec{N}) = \vec{i} \mod \vec{b}(\vec{N})$, which reproduces the results of the successive minima method [8].
- In particular, we describe a liveness algorithm from an execution trace, and another to compute the maximum number of variables alive alongside a dimension, from which we get our scalar modular mappings. We show that a simple interpolation allow to infer the modulo mapping $\vec{N} \mapsto \vec{b}(\vec{N})$.

This paper is structured as follows. Section 2 illustrates the array contraction problem. Section 3 outlines our trace analysis approach. Section 4 presents our preliminary results. Finally, Section 5 concludes this paper and draws future research directions.

2 Focus: array contraction

Array contraction consists in finding a mapping $A[\vec{i}] \mapsto \hat{A}[\sigma_A(\vec{i}, \vec{N})]$ reducing the size of A, where $\vec{N}$ is the vector of structure parameters. We illustrate array contraction on the Blur filter kernel:

```
for(y=0; y<2; y++)
 for(x=0; x<N; x++)
  blurx[x,y] = in[x,y] + in[x+1,y] + in[x+2,y];
for(y=2; y<N; y++)
 for(x=0; x<N; x++) {
  blurx[x,y] = in[x,y] + in[x+1,y] + in[x+2,y];
  out[x,y] = blurx[x,y-2] + blurx[x,y-1] + blurx↩
      [x,y];
 }
```

This kernel applies two consecutive elementary convolutions on the input signal *in*. The first convolution is bufferized to the array *bluxr*, in turn processed by the next convolution. With the canonical sequential schedule θ, *blurx* might be contracted with the mapping $(x, y, N) \mapsto (x \bmod N, y \bmod 3)$.

So far, that mapping was derived in two steps. First (step 1) an *array liveness analysis* is applied to compute the *conflict relation* $\bowtie_\theta$, relating array cells whose liveness intervals intersect. Then (step 2), the conflict relation is analyzed to infer a minimum-range mapping such that conflicting arrays

*Corresponding author

IMPACT'22, June 20, 2022, Budapest, Hungary
2022.

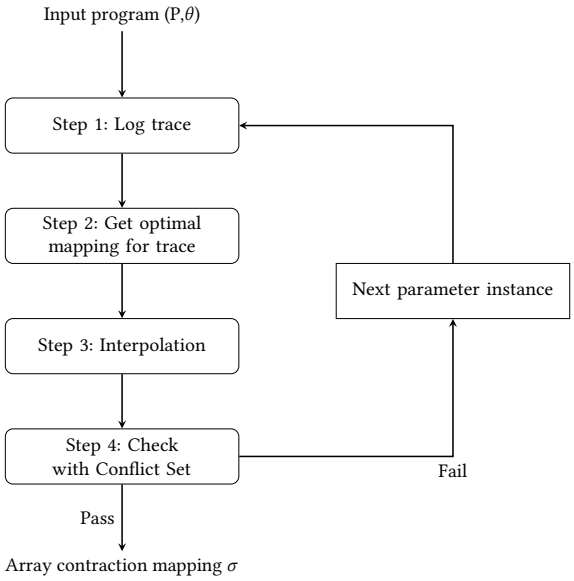
cells are mapped to different places:

$$blurx(\vec{i}) \bowtie_\theta blurx(\vec{j}) \wedge \vec{i} \neq \vec{j} \Rightarrow \sigma_{blurx}(\vec{i}, \vec{N}) \neq \sigma_{blurx}(\vec{j}, \vec{N})$$

Both steps require polyhedral manipulations. Step 1 requires to check the emptiness of integer polyhedra. Depending on the contraction algorithm, step 2 may require rational projections ([8]), mixed-LP ([1]), or parametric lexico-minimization of integer polyhedra ([3]). Both steps are expensive. In this paper, we use an oracle which assumes the precomputation of the conflict relation. Thus, we focus on improving step 2.

3 Our approach

The following diagram outlines our approach.



We execute the input kernel on small parameter values (step 1). Then, we apply a liveness dataflow analysis on the trace to infer the conflict relation, summarized as a conflict difference set $\mathcal{D}_a = \{\vec{i} - \vec{j} \mid a(\vec{i}) \bowtie_\theta a(\vec{j})\}$ (step 2). Since $\mathcal{D}_a$ is finite, the successive minima might be *instanciated* and computed with a small cost. Finally, we try to infer a parametrized modulo using an affine interpolation. If the mapping interpolated is correct (step 4), it is returned. Else we iterate on the next parameter value. Many open problems need to be investigated. How the parameters should be chosen? So far we considered programs with a single parameter N. In case several parameters are considered, one need to select linearly independant parameters. Other point, the contraction mapping is usually piecewise affine. With our method, we actually seek for the dominating value (on the biggest piece, not on the corners). How to guess parameters on that piece is still an open problem. Finally, we still depend on an oracle computed by means of polyhedral operations. We believe we can get rid of that oracle, providing we actually apply an *instance* of a polyhedral algorithm (here the successive minima).

4 Preliminary results

This section presents the preliminary results obtained with our approach. Our method have been implemented and evaluated on the following kernels, whose code is detailed in [11]:

- **pc-1D** is a simple producer/consumer through a 1D-array. The producer and the consumer access the array cells in the same order with a phase shift of two steps.
- **pc-2D-single** is a perfect loop nest operating over a 2D array with dependence vectors (1,0) and (0,1).
- **pc-2D-single** is a perfect loop nest operating over a 2D array with dependence vectors (1,0) and (0,1). Only the last array column is live-out.
- **blur** is the motivating example.

Our analyzer and the kernels have been compiled using gcc 9.3.0, the timings are obtained on an Intel Core i5-1135G7 CPU running at 2.40GHz. The results are depicted on the following table.

Kernel	Array	Size	Mapping found	It.	Runtime
pc-1D	A	N	$i \bmod 2$	2	1.04 ms
pc-2D	A	$N \times N$	$i \bmod N, j \bmod N$	2	13.0 ms
pc-2D-single	A	$N \times N$	$i \bmod 2, j \bmod N - 1$	2	14.2 ms
blur	blurx	$N \times N$	$x \bmod N, y \bmod 3$	2	13.1 ms

For each kernel and each array we provide the original size, the mapping inferred from our algorithm. *It.* is the number of iterations before interpolating a correct mapping. For all kernel we tried $N = 2$, then $N = 3$, except for the blur kernel, which required $N = 3$ and 4, since $N = 2$ is a corner case. Must of the time is lost in the oracle. This must still be improved to provide competitive execution times.

5 Conclusion

In this paper, we have presented a first step toward a trace-based polyhedral model through array contraction. We have outlined a method able to infer a modulo mapping given a canonical basis from a few off-line executions traces. We have also shown that interpolation of the parameters from a subset of the possible traces is possible, and we have applied it to our examples. We have shown results that reproduce the original successive modulo method and answered positively to the question of generalization from a subset of traces.

In the future, we seek to apply the same process to more benchmarks of the polyhedral community, as our experimentations were limited by time. We also look forward to apply finer analysis on the modulo parametrization, like heuristics to choose parameter values that are relevant to both the conflict set checking and the interpolation, and more generally a finer approach to the choice of basis for any array allocation problem. The first parameter will also be investigated. Finally, we plan to get rid of the oracle and focus on pure trace analysis.

References

[1] Christophe Alias, Fabrice Baray, and Alain Darte. 2007. Bee+ Cl@ k: An implementation of lattice-based array contraction in the source-to-source translator Rose. *ACM SIGPLAN Notices* 42, 7 (2007), 73–82.

[2] Christophe Alias and Alexandru Plesco. 2021. Data-aware process networks. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 1–11.

[3] Somashekaracharya G Bhaskaracharya, Uday Bondhugula, and Albert Cohen. 2016. Automatic storage optimization for arrays. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38, 3 (2016), 1–23.

[4] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.

[5] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multi-Dimensional Time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420.

[6] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*. 1581–1592.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (may 2017), 84âĂŞ90. https://doi.org/10.1145/3065386

[8] Vincent Lefebvre and Paul Feautrier. 1998. Automatic storage management for parallel programs. *Parallel computing* 24, 3-4 (1998), 649–671.

[9] Patrice Quinton and Vincent van Dongen. 1989. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology* 1, 2 (1989), 95–113.

[10] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. 1986. On synthesizing systolic arrays from Recurrence Equations with Linear Dependencies. In *Foundations of Software Technology and Theoretical Computer Science*, Kesav V. Nori (Ed.). Lecture Notes in Computer Science, Vol. 241. Springer Berlin Heidelberg, 488–503.

[11] Hugo Thievenaz, Keiji Kimura, and Christophe Alias. 2021. *Towards Trace-Based Array Contraction.* Research Report. Inria ; Waseda University. https://hal.inria.fr/hal-03482055