Algebraic Tiling facing Loop Skewing

Clément Rossetti
ICube, Inria
Strabourg, France
crossetti@unistra.fr

Alexis Hamon
ICube, Inria
Strasbourg, France

Philippe Clauss
ICube, Inria
Strasbourg, France
philipe.clauss@inria.fr

Abstract

Last year at Impact 2023, we presented an ongoing work of a new tiling technique called *algebraic tiling*. With algebraic tiling, tiles are defined by their volume (the number of iterations) instead of the size of their edges. This way tile of quasi-equal volumes are generated at runtime, whatever are the original loop bounds. This has many advantages, particularly it addresses load-balancing when parallelizing loops. However, algebraic tiling poses particular challenges when the tiled loops require a final skewing transformations of the tiles in order to exhibit parallel loops. In this paper, we focus on this challenge and propose a solution that makes algebraic tiling applicable in this context too.

Keywords: loop tiling, loop skewing, parallel loop, data locality, optimizing compilers

1 Introduction

Last year at Impact 2023, a new tiling technique called algebraic tiling was introduced [6]. It is a novel approach of rectangular, parametric and fully dynamic loop tiling based on the tile volumes instead of their edge sizes or shapes. When the tiled loops exhibit at least one inter-tile parallel loop, without requiring any further skewing transformation of the tiles, it has been shown in [6] that algebraic tiling provides a well-balanced multi-threaded parallel program: threads perform almost the same number of iterations, whenever the shape of the original iteration domain (rectangular, triangular, ...). However, when a final skewing transformation of the tiles is required in order to exhibit a parallel inter-tile loop implementing wavefront parallelism, algebraic tiles of arbitrary edge sizes pose some difficult challenges related to data dependences among the tiles. In this paper, we focus on this challenge and propose a solution making algebraic tiling also applicable when skewing of tiles is required.

Our application steps for algebraic tiling are the same as those used by the polyhedral compiler Pluto for generating a parallel tiled program [2] :

1. Check if the original loops can be tiled as a valid serial tiled program regarding data dependences.
2. If not and if possible, apply a skewing transformation such that the resulting loops can be tiled as a valid tiled serial program.
3. Check if there is at least one outer inter-tile loop that can be parallelized regarding data dependences.
4. If not and if possible, apply a skewing transformation impacting the two outermost inter-tile loops such that the second outermost loop in the resulting program does not carry any data dependence and thus can be parallelized.

Note that additional skewing transformations may be applied separately to the inner intra-tile loops in order to improve intra-tile data locality. With algebraic tiling, the same can be achieved without any specific difficulty.

We show in Section 3 that when a last skewing transformation is required to exhibit an inter-tile parallel loop, this last step poses a particular challenge to algebraic tiles, due to their arbitrary edge sizes.

The paper is organized as follows. In the next section we remind how algebraic tiling is applied on a given loop nest. Then in section 3, we describe how we had to adjust this technique when transformations are required to exhibit parallelism on the inter-tile loops. In section 4 we present some experimental results that we obtained by comparing algebraic tiling to standard rectangular tiling.

2 How algebraic tiling works

Algebraic tiling was introduced in [6]. Here is a reminder of how this technique works. Algebraic tiling is a new tiling approach based on the volumes of the tiles, i.e., the number of iterations contained in each tile, instead of the sizes of standard (hyper-)rectangular tiles, i.e., the sizes of the edges of the tiles. In the proposed approach, tiles are dynamically generated and have almost equal volumes. The iteration domain is well covered by a minimum number of tiles that are all almost full. Since the bounds of the generated tiles are not linear and defined by algebraic mathematical expressions, we call this loop tiling technique algebraic tiling.

One major difference with standard rectangular tiling is that bounds of algebraic tiles are computed dynamically. To do so, we use Trahrhe expressions. Those expressions are derived from the inversion of Ranking Ehrhart polynomials. Ehrhart polynomials are integer-valued and express the exact number of integer points contained in a finite multi-dimensional convex polyhedron which depends linearly on integer parameters [3].

Ranking Ehrhart polynomials are similar to Ehrhart polynomials. Those polynomials compute the rank of a given iteration I_0 of a loop nest of depth n, which is equal to the

IMPACT 2024, January 17, 2024, Munich, Germany
2024.

number of iterations that are executed before I_0 (included), i.e., the number of tuples $I = (i_1, i_2, ..., i_n)$ inside the iteration domain which are lexicographically less than or equal to I_0. Inverting those ranking polynomials allows to compute for a given rank at what iteration I_0 it would be computed.

In the case of algebraic tiling, we can compute at what iteration I_0 a given volume (or rank) will be reached, thus resulting in the bounds of the algebraic tiles.

In order to perform algebraic tiling on an iteration domain of dimension $n = (d_0, ..., d_n)$, we divide each dimension d_i in a given number of slices DIV_i, from the outermost to the innermost dimensions. Then for each dimension d_i, the total volume must be computed using the corresponding Ehrhart polynomial, which depends on the bounds of the slices of the previous dimensions. We have $vmax_i = Ehrhart_i(lb_0, ub_0, ...lb_{i-1}, ub_{i-1})$, where lb_k and ub_k denotes the slice lower bound, respectively the slice upper bound, of dimension k. We define the target volume that must be approached as closely as possible for each slice i as being $target\_vol_i = vmax_i / DIV_i$

To compute the bounds of the slices, we instantiate the Trahrhe expressions corresponding to each dimension. For each dimension d_i, the bounds of s contiguous slices must be dynamically computed, with $0 \leq s < DIV_i$. The lower bound ${lb_i}^s$ and upper bound ${ub_i}^s$ are computed as follows:

$$ {lb_i}^s = trahrhe_i(target\_vol_i \times s) $$
$$ {ub_i}^s = trahrhe_i(target\_vol_i \times (s+1)) - 1 $$

where $trahrhe_i(r)$ computes the value of loop index i of the tuple whose rank is r.

3 Adapting algebraic tiling to skewing tiles

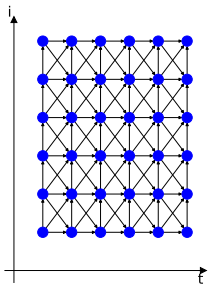
When algebraic tiling was introduced in [6], we explained that it has the the same validity requirements as standard rectangular tiling regarding data dependences. So whenever standard rectangular tiling is a valid transformation (i.e that result in a correct program), algebraic tiling is also valid. However, following the four main steps sketched in Section 1, the fourth step may not result in a valid program with algebraic tiles.

Indeed, if a skewing transformation of the inter-tile loops is required to exhibit a parallel inter-tile loop, the particular shapes of algebraic tiles may yield some dependence violations.

Let us consider the seidel-2d stencil computation from the polybench benchmark suite [5]. In this program, a grid of points of size $N \times N$ is updated $\_PB\_TSTEPS$ times as shown in Figure 1a. Each point is the average value of its neighbors. Therefore there is a dependence carried by index t: to update a point, it is required that all its neighbors have been updated at iteration $t-1$. There are also dependences carried by loops i and j as neighbors of a point that are lexicogaphically less must have been computed before.

```
1 for (t = 0; t <= _PB_TSTEPS - 1; t++)
2     for (i = 1; i <= _PB_N - 2; i++)
3         for (j = 1; j <= _PB_N - 2; j++)
4             A[i][j] = (A[i - 1][j - 1]
5                 + A[i - 1][j] + A[i - 1][j + 1]
6                 + A[i][j - 1] + A[i][j]
7                 + A[i][j + 1] + A[i + 1][j - 1]
8                 + A[i + 1][j] + A[i + 1][j + 1])
9                 / SCALAR_VAL(9.0); // S1
```

(a) Seidel-2d loop kernel.



(b) Iteration domain representation, with dependence vectors.

Figure 1. Seidel-2d kernel.

If we try to tile the original program, it will end up breaking data dependences, thus creating an incorrect program. So a skewing transformation must be applied before any type of tiling. The skewing indicated by Pluto's heuristic is the following:

$$ (t, i, j) \rightarrow (t, t+i, 2t+i+j) $$

This skewing results in the code shown in Figure 2.

Once the skewing, we can perform a tiling, either rectangular or algebraic as illustrated in Figure 3.

But in this program, there is no parallel inter-tile parallel loop. An additional skewing transformation is required to exhibit a parallel inter-tile loop. Pluto suggests the following skewing transformation:

$$ (zt_0, zt_1, zt_2, t_0, t_1, t_2) \rightarrow (zt_0 + zt_1, zt_1, zt2, t_0, t_1, t_2) $$

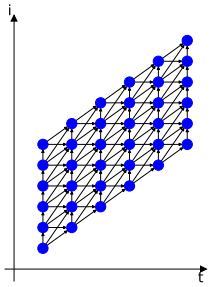
where the zt_i's denote the inter-tile loops and the t_i's are the initially skewed dimensions (i.e. $t_0 = t, t_1 = t+i, t_2 = 2t+i+j$).

But when applying the same skewing on algebraic loops, we break some data dependences as illustrated in Figure 4a. On this figure, tiles that have the same color should be executed concurrently. This ends up in an incorrect program because almost all the tiles computed in parallel have a dependence among each other (a $(1, 0)$ dependence vector).

Unlike standard rectangular tiles, algebraic tiles are not aligned and do not have the same height, thus making it difficult to determine a valid skewing transformation on the inter-tile loops.

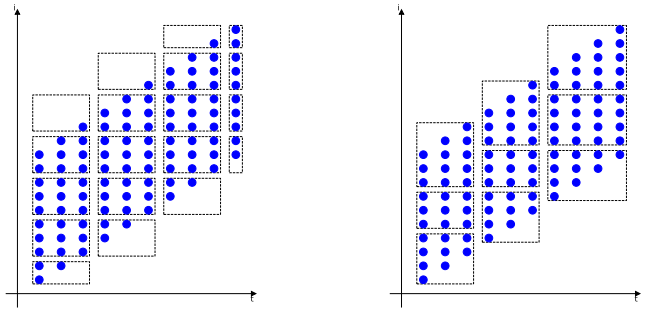
```
1 for (t0 = 0; t0 <= _PB_TSTEPS - 1; t0++) {
2   for (t1 = t0+1; t1 <= t0 + _PB_N-2; t1++) {
3     for (t2=t0+t1+1; t2 <= t0+t1+_PB_N-2; t2++) {
4       S1(t0, -t0+t1, -t0 - t1 + t2);
5     }
6   }
7 }
```

(a) Loop kernel.



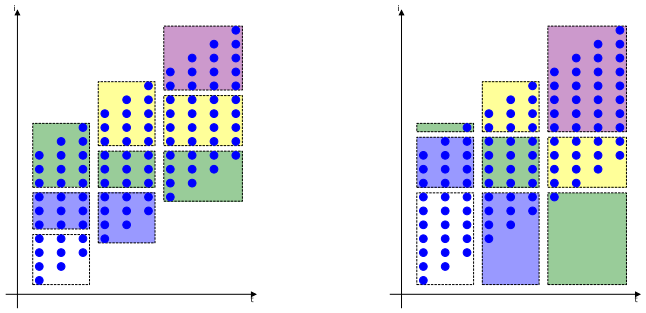
(b) Iteration domain representation, with dependence vectors.

Figure 2. Seidel-2d after the first skewing transformation.



(a) Rectangular tiling. **(b)** Algebraic tiling.

Figure 3. Applying tiling on seidel-2d.



(a) incorrect. **(b)** correct.

Figure 4. Seidel-2d algebraic tiling dependence violation. Tiles of same color should be executed concurrently.

In order to generate valid algebraic tiled loops in such cases, we use the following strategy: when tiling loops after the first skewing transformation (step 2), the second inter-tile loop is tiled independently to the algebraic bounds of the outermost inter-tile loop. In this way, slices are performed on the whole dimension of this second loop, and not slice per slice regarding the outermost dimension. The resulting tiled program can then support a final skewing transformation of the inter-tile loops which is valid, similarly to standard rectangular tiling. Thus, the same skewing transformation as the one suggested by Pluto for standard tiling can now be applied for algebraic tiling.

For seidel-2d, it is shown on Figure 4b the result of this strategy, where each concurrent tile does not break any dependence anymore, leading to a correct program.

Figure 5 presents the final algebraically tiled seidel-2d code. A skewing of the inter-tile loops was achieved so the second inter-tile loop can be parallelized using OpenMP directive. Note that this code implements a hybrid tiling, since the innermost dimension uses a standard rectangular tile approach. This was made necessary by the very high complexity of the Ehrhart ranking polynomials and Trahrhe expressions associated to this dimension. Thus DIV_i denotes either dividers, for both outermost dimensions, or a tile size, for the innermost dimension.

Some statements are also added to compute the bounds of the tiled loops:

lines 1-3: the upper bounds ubt_i of each inter-tile loop are computed. If the loop is tiled using the standard approach, then the upper bound is computed by dividing the number of iteration by the tile size. If the loop is tiled algebraically, then it is the number of slices.

lines 4-7: Variable t_0_pcmax is the total number of iterations of the loop nest. This volume is computed by calling the Ehrhart function. By dividing this quantity by divider DIV_0, we dynamically compute a target volume for each of the slices over the outermost dimension, stored in variable $TARGET\_VOL\_L0$. Each slice volume will be as close as possible to this volume. The same process is achieved for the next inner slices with variables $t1_pcmax$, $TARGET\_VOL\_L1$ and divider DIV_1.

lines 12-20: upper and lower bounds of the two outermost slices are computed at every iteration of the second outermost loop.

4 Experiments

Some experiments were conducted on five stencil programs from the polybench benchmark suite using the data size `EXTRALARGE`.

Details about the machine used for those experiments can be found in Table 1. Runs were performed using 64 threads after having set the environment variable `OMP_PROC_BIND=true`

```
ubt0 = DIV0 - 1;
ubt1 = DIV1 - 1;
ubt2 = ceild(_PB_N - 2, DIV2);
t0_pcmax = t0_Ehrhart(_PB_TSTEPS, _PB_N);
TARGET_VOL_L0 = t0_pcmax / DIV0;
t1_pcmax = t1_Ehrhart(_PB_TSTEPS, _PB_N);
TARGET_VOL_L1 = t1_pcmax / DIV1;
for (zt0 = 0; zt0 <= ubt0 + ubt1 - 1; zt0++) {
#pragma omp parallel for private(lb0, ub0, lb1, ub1, lb2, ub2, zt2, t0, t1, t2)
    for (zt1 = max(0, zt0 - ubt0); zt1 <= min(ubt1, zt0); zt1++) {
        lb0 = t0_trahrhe_t0(max((zt0 - zt1) * TARGET_VOL_L0, 1), _PB_TSTEPS, _PB_N);
        ub0 = t0_trahrhe_t0(min((zt0 - zt1 + 1) * TARGET_VOL_L0, t0_pcmax), _PB_TSTEPS, _PB_N) - 1;
        if (zt0 - zt1 == DIV0 - 1)
            ub0 = _PB_TSTEPS - 1;
        lb1 = t1_trahrhe_t1(max(1, zt1 * TARGET_VOL_L1), _PB_TSTEPS, _PB_N);
        ub1 = t1_trahrhe_t1(min(t1_pcmax, (zt1 + 1) * TARGET_VOL_L1), _PB_TSTEPS, _PB_N) - 1;
        if (zt1 == DIV1 - 1)
            ub1 = _PB_N - 2 + ub0;
        for (zt2 = 0; zt2 < ubt2; zt2++) {
            lb2 = zt2 * DIV2 + lb0 + lb1 + 1;
            ub2 = (zt2 + 1) * DIV2  + lb0 + lb1;
            for (t0 = max(0, lb0); t0 <= min(_PB_TSTEPS - 1, ub0); t0 += 1)
                for (t1 = max(t0 + 1, lb1); t1 <= min(ub1, _PB_N + t0 - 2); t1 += 1)
                    for (t2 = max(t0 + t1 + 1, lb2); t2 <= min(_PB_N + t0 + t1 - 2, ub2); t2 += 1)
                        S1(t0, t1 - t0, t2 - t1 - t0);
        }
    }
}
```

Figure 5. Algebraic tiling on seidel-2d loop kernel.

Table 1. Details of architecture used for experiments.

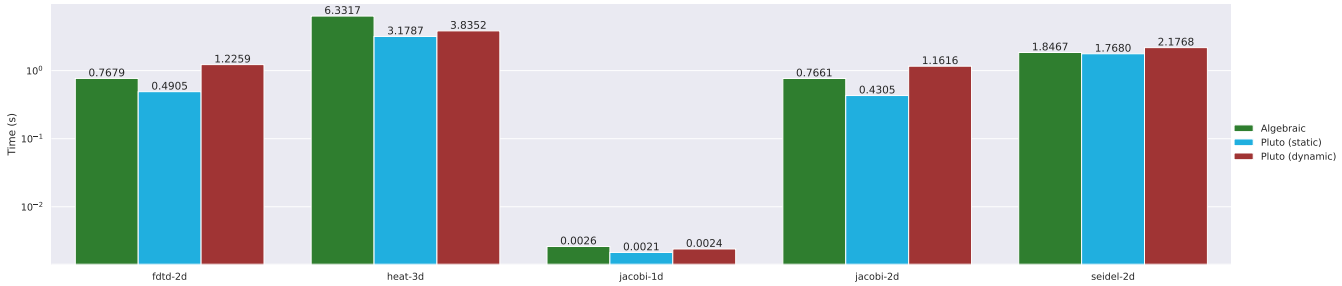
	AMD EPYC 7502
Micro-architecture	Zen2
Clock Speed	2.5Ghz
Cores / socket	32
Total cores	64
L1 cache / core	32kB
L2 cache / core	512kB
L3 cache / core	128MB
Compiler	gcc 11.4.0
Compiler flags	-O3 -fopenmp -march=native
Linux kernel	5.15.0
Pluto version	0.11.4
Pluto flags	–tile –parallel –nounroll –noprevector

to avoid thread migration and bind the threads to processor cores.

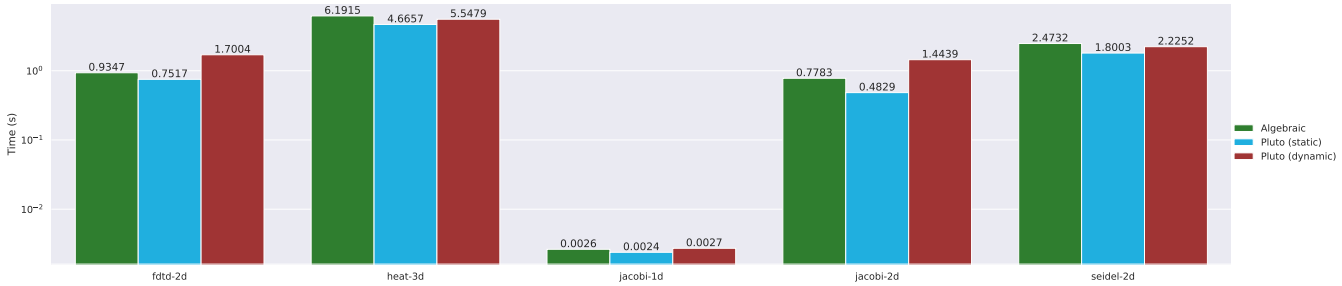
Standard (hyper-)rectangular tiling and OpenMP parallelization were automatically applied thanks to Pluto. Note however that four of the five target programs may be handled using non-rectangular tiling techniques as diamond tiling [1, 4], that do not require any skewing transformation. But our goal in this paper is to discuss exclusively the issues related to the skewing of rectangular tiles, without competing with other tiling techniques.

For each benchmark, the program has been run five times, and the average execution time of the three median runs has been retained. If standard deviation is above 5% for a given set of instantiated algebraic tiling dividers, then the result has not been retained. The tile sizes resulting in the fastest codes were found through an exhaustive search among combinations of powers of 2 from 2 to 128. Similarly, the best dividers for algebraic tiling were also selected through an exhaustive search among combinations of powers of 2 from 256 to 2048.

The results of our experiments are presented in Figure 6a with automatic vectorization activated (a) and deactivated (b). Execution times are displayed in seconds (lower is better) using a logarithmic scale. Algebraically tiled benchmarks are displayed in green, while red and blue bars correspond to standard rectangular tiling. The blue bars correspond to programs using the OpenMP static schedule while the red ones correspond to the use of the OpenMP dynamic schedule.



(a) (with vectorization)



(b) (without vectorization)

Figure 6. Performance of algebraic tiling vs rectangular tiling (lower is better).

Programs with algebraic tiling have their execution times always higher than those obtained with standard tiling, however very close in some cases (e.g. jacobi-1d and seidel-2d). Although the presented strategy allows to generate valid parallel programs with algebraic tiles when skewing of inter-tile loops is required, it leads to a loss of the load balance that algebraic tiling is supposed to provide: dividers DIV_i are no more directly related to the number of parallel threads, and partial algebraic tiles are generated.

This means that for better performance, another strategy similar to diamond tiling [1] or split tiling [4] should be investigated, in order to get better control on the number, the volume and the density of the parallel algebraic tiles.

5 Conclusion

We have proposed a strategy for applying algebraic tiling when a skewing transformation is required, in order to exhibit a parallel inter-tile loop. Although valid programs are generated with this strategy, the resulting runtime performance is not satisfactory enough. Indeed, the main advantages of algebraic tiling related to load balancing are lost with the skewing of inter-tile loops.

In the near future, another strategy must be elaborated, where parallel algebraic tiles are defined precisely regarding the potential wavefront parallelism of the skewed loops.

References

[1] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. 2017. Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (2017), 1285–1298. https://doi.org/10.1109/TPDS.2016.2615094

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.* 43, 6 (jun 2008), 101–113. https://doi.org/10.1145/1379022.1375595

[3] Philippe Clauss. 1996. Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs. In *Proceedings of the 10th International Conference on Supercomputing* (Philadelphia, Pennsylvania, USA) *(ICS '96)*. New York, NY, USA, 278–285.

[4] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. 2013. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (Houston, Texas, USA) *(GPGPU-6)*. ACM, New York, NY, USA, 24–31. https://doi.org/10.1145/2458523.2458526

[5] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench* 437 (2012), 1–1.

[6] Clément Rossetti and Philippe Clauss. 2023. Algebraic Tiling. In *Proceedings of the 13th International Workshop on Polyhedral Compilation Techniques.*