Algebraic Tiling

Clément Rossetti and Philippe Clauss
Inria, ICube
University of Strasbourg, France
{clement.rossetti,philippe.clauss}@inria.fr

Abstract

In this paper, we present an ongoing work whose aim is to propose a new loop tiling technique where tiles are characterized by their volumes – the number of embedded iterations – instead of their sizes – the lengths of their edges. Tiles of quasi-equal volumes are dynamically generated while the tiled loops are running, whatever are the original loop bounds, which may be constant or depending linearly of surrounding loop iterators. The adopted strategy is to successively and hierarchically slice the iteration domain in parts of quasi-equal volumes, from the outermost to the innermost loop dimensions. Since the number of such slices can be exactly chosen, quasi-perfect load balancing is reached by choosing, for each parallel loop, the number of slices as being equal to the number of parallel threads, or to a multiple of this number. Moreover, the approach avoids partial tiles by construction, thus yielding a perfect covering of the iteration domain minimizing the loop control cost. Finally, algebraic tiling makes dynamic scheduling of the parallel threads fairly purposeless for the handled parallel tiled loops.

Keywords Loop tiling, load balancing, parallel loop, data locality, optimizing compilers

1 Introduction

Loop tiling [7, 13, 18, 19, 26] (variously called loop blocking or partitioning) is a well-known loop optimizing transformation often providing significant speed-ups. It explicitly addresses data locality by contracting the accessed data space of inner loops in order to promote good cache reuse. When handling parallel loops, it also provides a way to adjust the grain of parallelism or to exploit vectorization while avoiding register pressure on a single core. Loop tiling is an essential strategy used by compilers and automatic parallelizers.

However, as it was pointed out by Sato *et al.* in [22], traditional loop tiling techniques do not address load balancing explicitly when handling parallel loops, thus yielding bad scalability for parallelism. Moreover, rectangular tiles of constant size may generally not cover perfectly the iteration domain, since tiles overlapping borders of the domain result in partial tiles containing less iterations than the full tiles. Iooss *et al.* in [12] and Renganarayanan *et al.* in [20] address the limitations of fixed size tiling. They point out that the tile sizes have a huge influence on performance, and that parametric tiling in its full generality is known to be non-linear. Allowing tile sizes to be symbolic parameters at compile time has many benefits, including efficient auto-tuning, and run-time adaptability to system variations [12, 20].

In the related literature, load balancing is sometimes addressed, but from a different perspective and for the particular class of loops that implement stencil computations. While stencil loops mostly require a skewing transformation to be amenable to parallelization [1, 18, 27], dedicated techniques for data locality optimization and concurrent start for tiles are proposed. Some noticeable proposals consist in generating tiles with a specific shape (diamond tiling [2], hexagonal tiling [9], or split and overlapped tiling [11, 17]).

We propose a new tiling approach based on the volumes of the tiles, *i.e.*, the number of iterations delimited by the tiles, 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, even if their shape and edge sizes may differ. 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*.

Algebraic tiles are built by successive hierarchical slicing of the initial iteration domain, from the outermost to the innermost depth dimensions of the target loop nest, in a way ensuring that slices have all quasi-equal volumes. The bounds of the loop nests that are handled must be constants, or linear functions of the surrounding loop iterators and of unknown parameters – which are typically related to the data input size. Such loops are also called polyhedral loops since they may be handled using the polyhedral model [8]. Quasi-perfect load balancing is achieved when each parallel loop is sliced using as many slices of quasi-equal volumes as parallel threads, and when most of the iterations have close execution times. Thus, such dynamic slicing strategy makes the resulting parallel loop scalable regarding the number of threads. Good data locality is reached by slicing profitably the non-parallelized loops, and by slicing the parallel loops in a number of parts equal to a multiple of the number of parallel threads. The number of generated slices for each dimension may stay as a parameter at compile-time, making algebraic tiling a parameterized loop tiling technique, allowing the produced code to adapt to the number of parallel threads and data layout.

IMPACT, Jan. 16, 2023, Toulouse, FR
2023.

We are currently implementing algebraic tiling as an extension of the automatic loop optimizer Pluto[1] [3]. Our first experiments show that algebraic tiling outperforms significantly (hyper-)rectangular tiling when parallelizing loops with OpenMP using static scheduling, and mostly provides similar or lower execution times when compared to traditionally tiled loops parallelized using dynamic scheduling of OpenMP. Thus, algebraic tiling makes dynamic scheduling fairly purposeless for the handled loop nests.

The paper is organized as follows. In the next section, we first describe the optimizing strategy of algebraic tiling and how it may be applied to any compliant loop nest. Then in Section 3, we introduce the mathematical background required for computing the bounds of algebraic tiles. Several issues regarding the applicability of algebraic tiling are addressed in Section 4. Our software that automatically generates the functions required for using algebraic tiles is described in Section 5. Experiments showing significant speedups against the (hyper-)rectangular tiling approach are exhibited in Section 6. Related work is addressed in Section 7 and conclusions are given in Section 8.

2 Algebraic tiling in practice

In this section, we describe algebraic tiling regarding its related programming aspects. Its mathematical foundations are presented in the next section.

2.1 Overview

Consider the loop nest shown in Figure 1, which is extracted from program syr2k of the polybench benchmark suite[2] v4.2. This loop kernel defines a triangular iteration domain. When tiling this loop nest using the standard approach, the iteration domain is partitioned into fixed-sized rectangular tiles. When also parallelizing the outermost loop among a given number of threads, each thread is traditionally assigned a slice of the iteration domain, all slices being of the same width, as illustrated in Figure 2. Two main issues can be highlighted with this example: (1) many partial tiles occur at the borders of the iteration domain and (2) slices assigned to the threads are significantly unbalanced regarding their iteration counts. In contrast, algebraic tiling first slices the outermost loop into parts of quasi-equal volumes V_i, as shown in Figure 3a, where $V_0 \simeq V_1 \simeq V_2 \simeq V_3 \simeq V_4$. Then, these slices are sliced in turn along the inner loop direction in parts of quasi-equal volumes V_{ij}, as shown in Figure 3b. Hence quasi-perfect load balancing is reached when parallelizing the outermost loop, while good data locality is achieved thanks to inner slicing and perfect domain covering.

A sample of the algebraic tile bounds that are dynamically generated is shown in Table 1, when slicing the outermost loop in 24 parts, and the innermost in 64 parts, and with

[1]http://pluto-compiler.sourceforge.net

[2]https://sourceforge.net/projects/polybench

```
for(i = 0; i < N;  i++)
 for(j = 0; j <= i; j++)
  for(k = 0; k < M; k++)
   C[i][j] += A[j][k]*alpha
              *B[i][k]+B[j][k]
              *alpha*A[i][k];
```

Figure 1. syr2k loop kernel

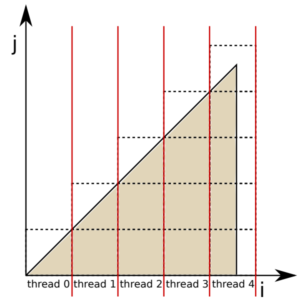
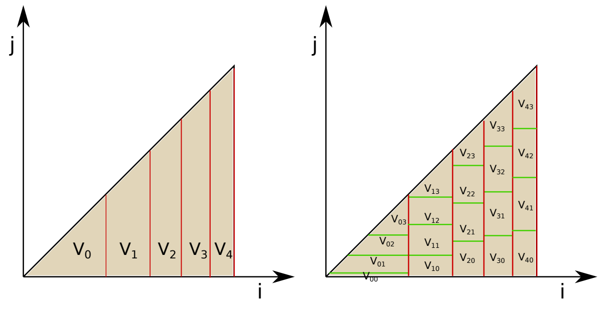


Figure 2. syr2k iteration domain, tiled and parallelized among 5 threads



(a) outermost balanced slicing **(b)** inner balanced slicing

Figure 3. Algebraic tiling applied on syr2k iteration domain

input data size LARGE DATASET ($M = 1000$; $N = 1200$). The first column of each of the three parts of the table shows the 24 outermost slices and their respective lower and upper bounds, lb_i and ub_i, and the resulting slice volumes (iteration count). For each outermost slice, the second column shows a sample of the related 64 inner slices and their respective lower and upper bounds, lb_j and ub_j, and the resulting tile volumes. Each resulting tile contains a number of iterations which is as close as possible to $LTC/24/64 = 469140$, where LTC denotes the total loop trip count of the syr2k loop kernel. Note that all tiles have different sizes and may be non-rectangular along the borders of the iteration domain.

2.2 Volumes of algebraic tiles

The target volumes of algebraic tiles are approached using dividers, DIV_1, DIV_2, etc., that are the target number of slices associated to every loop nest depth. When parallelizing the outermost loop of a loop nest, the outermost divider DIV_1 is typically the number of parallel threads, and the volumes of the outermost slices must be as close as possible to LTC/DIV_1. However, for data locality optimization reasons, it may be beneficial in some cases to set DIV_1 as equal to a multiple of the number of threads. If so, the OpenMP static schedule still yields a balanced distribution of the so-generated smaller slices of quasi-equal volumes. The next dividers for inner slices are selected to reach good cache performance. Note also that nested parallelism may also be managed efficiently by selecting next dividers according to the number of processing units of each hierarchical parallel level.

2.3 Code transformation

On Figure 4, it is shown how the original syr2k loop kernel of Figure 1 is transformed for algebraic tiling. A similar transformation may be applied on any target loop nest in the same way:

- **line 1:** function `i_Ehrhart` computes the loop trip count (*LTC*) of the target loop nest, given the values of the size parameters M and N.
- **line 2:** the target volume `TILE_VOL_L1` of the outermost slices is computed using the given outermost divider DIV_1.
- **lines 3-5:** the outermost loop scanning the outermost slices is parallelized with OpenMP, where private variables are defined appropriately.
- **lines 6-7:** lower and upper bounds of every outermost slice are computed at every iteration of the outermost loop, and such that the resulting slice has a volume as close as possible to `TILE_VOL_L1`. Slice bound values are obtained thanks to function `trahrhe_i`, that computes the value of the original outermost loop iterator, where `it` $\times$ `TILE_VOL_L1` iterations have been completed.
- **line 8:** if the last iteration of the outermost loop scanning the outermost slices has been reached, then the last bound is set to the upper bound of the original outermost loop.
- **line 9:** the loop trip count of the current slice is computed using function `j_Ehrhart`, which takes the current slice bounds as input parameters.
- **lines 10-14:** operations similar to lines 3-8 are performed for the inner depth.
- **lines 15-21:** loops scanning the interior of the tiles are performed, where loop bounds are set using the bounds of the algebraic tiles.

The implementation of algebraic tiling is mainly based on functions that provide the values of loop iterators where a given number of iterations has been completed (`trahrhe_i` and `trahrhe_j` in the current example). Such functions evaluate algebraic expressions using floating-point arithmetic. These algebraic expressions are symbolic roots of multivariate polynomial equations. The automatic generation of such functions is detailed in the next section.

3 Trahrhe expressions

Name «trahrhe» comes from «Ehrhart» with letters in reversed order. Trahrhe expressions are roots of Ehrhart polynomials, and they are also the inverses of particular Ehrhart polynomials which are called *ranking polynomials*. Note that these new mathematical objects were already used to collapse non-rectangular loops in [5].

3.1 Ranking polynomials

Ehrhart polynomials were proposed and extended to program analysis by Clauss in [4]. These integer-valued polynomials express the exact number of integer points contained in a finite multi-dimensional convex polyhedron which depends linearly on integer parameters. They have many applications for the quantitative analysis of loop nests whose loop bounds are linear functions of the surrounding loop indices and integer parameters, and whose statements are referencing multi-dimensional array elements through linear functions of the loop indices and parameters. Such a counting of integer points may translate to the exact number of iterations of a parameterized loop nest, the exact number of memory locations touched by a loop nest, the maximum number of parallel iterations, etc. When considering a d-dimensional polyhedron – as for example the iteration domain of a d-depth loop nest – depending linearly on integer parameters $p_1, p_2, \ldots, p_m$, its Ehrhart polynomial is a polynomial of degree d whose variables are $p_1, p_2, \ldots, p_m$. Ehrhart polynomials can be automatically computed using existing algorithm implementations as the one of the barvinok library [24].

Among their applications, Ehrhart polynomials are used by Clauss and Meister in [6] to reorganize the memory layout of array elements accessed by a loop nest, in order to improve their spatial data locality: array elements are relocated in memory in the same order as they are accessed. In this approach, the new location of an array element is given by the order, or *rank*, of the iteration referencing it.

Such a rank of iterations is given by a polynomial, called a *ranking polynomial*, whose variables are the loop iterators, and whose evaluation results in the number of iterations preceding a given iteration. More formally, the *ranking polynomial* of a d-depth loop nest whose loop iterators, from the outermost to the innermost, are $(i_1, i_2, \ldots, i_d)$, is denoted $r(i_1, i_2, \ldots, i_d)$. Without loss of generality, if $(0, 0, \ldots, 0)$ defines the first iteration of the loop nest, then $r(0, 0, \ldots, 0) = 1$, $r(0, 0, \ldots, 1) = 2$, and so on. If $(N_1, N_2, \ldots, N_d)$ are the indices of the very last iteration, then $r(N_1, N_2, \ldots, N_d)$ is the total number of iterations of the loop nest (*LTC*).

The computation of the ranking polynomial of a loop nest is detailed in [6]. We recall this technique by applying it to the syr2k loop kernel of Figure 1, to compute the associated ranking polynomial $r(i, j, k)$.

The rank of a given iteration (i_0, j_0, k_0) is equal to the number of iterations that are executed before (i_0, j_0, k_0) (included), *i.e.*, the number of triplets (i, j, k) inside the iteration domain which are lexicographically less than or equal to (i_0, j_0, k_0):

Table 1. Generated tile bounds and resulting tile volumes for kernel syr2k with dividers 24 and 64 (sample)

Target: 30,025K					
$lb_i \rightarrow ub_i$	$lb_j \rightarrow ub_j$	$lb_i \rightarrow ub_i$	$lb_j \rightarrow ub_j$	$lb_i \rightarrow ub_i$	$lb_j \rightarrow ub_j$
	Target: 467K		Target: 468K		Target: 466K
0→243 (29,890K)	...	692→733 (29,967K)	0→10 (462K)	979→1008 (29,835K)	0→14 (450K)
	1→2 (484K)		11→21 (462K)		15→30 (480K)
	...		...		...
	63→65 (537K)		434→444 (462K)		621→636 (480K)
	...		...		...
	191→200 (440K)		691→702 (384K)		963→977 (450K)
	...		...		978→1008 (31K)
	Target: 470K		Target: 471K		Target: 480K
244→345 (30,141K)	0→3 (408K)	734→773 (30,180K)	0→10 (440K)	1009→1038 (30,735K)	0→15 (480K)
	...		...		...
	277→284 (496K)		707→718 (480K)		992→1007 (480K)
	...		...		1008→1038 (31K)
	Target: 463K		Target: 471K		Target: 460K
346→422 (29,645K)	0→5 (462K)	774→811 (30,153K)	0→11 (456K)	1039→1066 (29,498K)	0→15 (448K)
	...		...		...
	362→369 (432K)		743→755 (494K)		987→1003 (476K)
	370→379 (440K)		756→767 (456K)		1004→1019 (448K)
	380→392 (403K)		768→780 (416K)		1020→1036 (476K)
	393→422 (30K)		781→811 (31K)		1037→1066 (30K)
	Target: 470K		Target: 467K		Target: 473K
423→488 (30,129K)	0→6 (462K)	812→847 (29,898K)	0→11 (432K)	1067→1094 (30,282K)	0→15 (448K)
	...		...		...
	436→445 (440K)		791→803 (468K)		1047→1063 (476K)
	...		...		...
	Target: 469K		Target: 473K		Target: 467K
489→546 (30,073K)	0→7 (464K)	848→882 (30,310K)	0→12 (455K)	1095→1121 (29,943K)	0→16 (459K)
	...		...		...
	494→503 (440K)		825→837 (455K)		1039→1056 (486K)
	...		...		...
	Target: 465K		Target: 464K		Target: 461K
547→598 (29,822K)	0→7 (416K)	883→915 (29,700K)	0→13 (462K)	1122→1147 (29,523K)	0→16 (442K)
	...		...		...
	537→545 (468K)		871→885 (465K)		1082→1099 (468K)
	...		...		...
	Target: 467K		Target: 466K		Target: 471K
599→646 (29,928K)	0→8 (432K)	916→947 (29,840K)	0→13 (448K)	1148→1173 (30,199K)	0→17 (468K)
	...		...		...
	584→593 (480K)		830→844 (480K)		1125→1142 (468K)
	...		...		1143→1173 (31K)
	Target: 471K		Target: 466K		Target: 482K
647→691 (30,150K)	0→9 (450K)	948→978 (29,884K)	0→14 (465K)	1174→1199 (30,875K)	0→17 (468K)
	10→19 (450K)		15→29 (465K)		18→36 (494K)
	20→30 (495K)		30→44 (465K)		37→54 (468K)
	31→40 (450K)		45→59 (465K)		55→73 (494K)
	41→51 (495K)		60→74 (465K)		74→91 (468K)
	...		...		92→110 (494K)
	638→648 (484K)		948→978 (31K)		...
	...				

$$\forall (i_0, j_0, k_0) \text{ s.t. } 0 \le i_0 < N \text{ and } 0 \le j_0 \le i_0 \text{ and } 0 \le k_0 < M,$$

$$r(i_0, j_0, k_0) = \#\{(i,j,k) \mid (i,j,k) \trianglelefteq (i_0, j_0, k_0), 0 \le i < N, 0 \le j \le i, 0 \le k < M\}$$

where $\trianglelefteq$ denotes the lexicographic order. Since lexicographic inequalities are not linear, the problem is split as the conjunction of three equivalent sets of linear inequalities, according to the definition of the lexicographic order:

$$(i,j,k) \trianglelefteq (i_0, j_0, k_0) \Leftrightarrow (i < i_0) \text{ or } (i = i_0 \text{ and } j < j_0) \text{ or } (i = i_0 \text{ and } j = j_0 \text{ and } k \le k_0)$$

Therefore, the sets whose integer points must be counted can be defined as the union of three disjoint convex polyhedra, and $r(i_0, j_0, k_0)$ as the sum of three Ehrhart polynomials:

$$\begin{aligned} r(i_0, j_0, k_0) &= \#\{(i,j,k) \mid 0 \le i < i_0, 0 \le j \le i, 0 \le k < M\} \\ &+ \#\{(i,j,k) \mid i = i_0, 0 \le j < j_0, 0 \le k < M\} \\ &+ \#\{(i,j,k) \mid i = i_0, j = j_0, 0 \le k \le k_0\} \\ &= \frac{M\, i_0\ (i_0 + 1)}{2} + M\, j_0 + k_0 + 1 \\ &= \frac{2\, k_0 + 2\, M\, j_0 + M\, i_0^2 + M\, i_0 + 2}{2} \end{aligned}$$

One can verify that the rank of the first iteration $(0,0,0)$, $r(0,0,0)$, is equal to 1, the rank of the second iteration $r(0,0,1) = 2$, the rank of the third iteration $r(0,0,2) = 3$ and so on. The rank of the last k-iteration when $i = 0$ and $j = 0$, $r(0,0,M-1) = M$, and the rank of the first iteration when

```
i_pcmax = i_Ehrhart(N, M); /* Loop Trip Count */
TILE_VOL_L1 = i_pcmax/DIV1; /* Outermost Slices' Targeted Volume */
#pragma omp parallel for firstprivate(i_pcmax,TILE_VOL_L1)  \
 private(i,j,k,lbi,ubi,lbj,ubj,jt,j_pcmax,TILE_VOL_L2)
for (it=0;it < DIV1;it++) { /* Loop Scanning the Outermost Slices */
 lbi=trahrhe_i(max(it*TILE_VOL_L1,1),N,M); /* Bounds of the it^th slice */
 ubi=trahrhe_i(min((it+1)*TILE_VOL_L1,i_pcmax),N,M)-1;
 if (it==DIV1-1) ubi=N-1; /* Last slice adjustment */
 j_pcmax=j_Ehrhart(N, M, lbi, ubi); /* Loop Trip Count of the current outermost slice */
 TILE_VOL_L2 = j_pcmax/DIV2; /* Tiles' Targeted Volume */
 for (jt=0; jt < DIV2; jt++) { /* Loop Scanning the Tiles */
  lbj=trahrhe_j(max(jt*TILE_VOL_L2,1),N,M,lbi,ubi); /* Bounds of the jt^th tile */
  ubj=trahrhe_j(min((jt+1)*TILE_VOL_L2,j_pcmax),N,M,lbi,ubi)-1;
  if (jt==DIV2-1) ubj=ubi; /* Last tile adjustment */
  for (i=max(0,lbi); i < min(N,ubi+1); i++) { /* Inner tile loops */
   for (j=max(0,lbj); j <= min(i,ubj); j++) { /* Bounded by the tile bounds */
    for (k=0; k <= M-1; k++) {
     C[i][j]+=A[j][k]*alpha*B[i][k]+B[j][k]*alpha*A[i][k];
    }
   }
  }
 } /* end for jt */
} /* end for it */
```

Figure 4. Algebraic tiling on syr2k loop kernel

$i = 1$ and $j = 0$, $r(1, 0, 0) = M + 1$. The total number of iterations is $LTC = r(N-1, N-1, M-1) = \frac{MN(N+1)}{2}$.

Such a ranking polynomial associates, to each iteration index tuple, a unique integer of a continuous interval of integers starting at 1. This continuous interval is the range of integers between one and the total number of iterations. Conversely, each integer value in the interval is associated to one unique iteration index tuple. The ranking polynomial can also be seen as the one-dimensional polynomial schedule function of the iterations, which is equivalent to the original multi-dimensional linear schedule defined by the nested loops. Another important property is that such a ranking polynomial is monotonically increasing over the integers, from 1 to the total number of iterations, relatively to the lexicographic order of the loop indices. Thus, a ranking polynomial defines a *bijection* between the iteration domain and the interval of successive integers. It implies that in theory, it can be inverted. Such inversions result in expressions called *trahrhe expressions*, which are used to compute the bounds of algebraic tiles.

3.2 Generating trahrhe expressions

For a given rank pc, $trahrhe(pc)$ is the tuple $(t_1, t_2, ..., t_d)$ such that $r(t_1, t_2, ..., t_d) = pc$, *i.e.*, a solution of the latter equation. The definition of function $trahrhe(pc)$ is calculated incrementally by first determining t_1, then propagating it to determine t_2, and so on. At each step, a symbolic uni-variate polynomial equation is solved:

Find t_1: Solve for s: $r(s, 0, ..., 0) - pc = 0$. Depending on the degree d of the ranking polynomial, this equation may have d real or complex solutions $s_1, ..., s_d$. Among these solutions, only one solution s_k is such that $t_1 = \lfloor s_k \rfloor = 0$ when $pc = 1$, which means that for the very first iteration of the d-depth loop nest ($pc = 1$), the value of the outermost loop iterator t_1 is, as expected, 0. This solution is propagated in the next equation.

Find t_2: Solve for s: $r(t_1, s, 0, ..., 0) - pc = 0$. Once again, among the $d - 1$ solutions, only one solution s_l is such that $t_2 = \lfloor s_l \rfloor = 0$ when $pc = 1$. This solution is propagated in the next equation.

Find t_3: Solve for s: $r(t_1, t_2, s, 0, ..., 0) - pc = 0$.

...

Find t_d: Solve for s: $r(t_1, t_2, t_3, ..., s) - pc = 0$. This last equation is obviously linear: $t_d = pc - r(t_1, t_2, ..., 0)$.

As an example, let us compute step by step the trahrhe expressions of the syr2k loop kernel:

1. $r(i, j, k) = \frac{2k+2Mj+Mi^2+Mi+2}{2}$
2. Solve $r(s, 0, 0) - pc = \frac{Ms^2+Ms+2}{2} - pc = 0$. This equation has two solutions:
$$s_1 = -\frac{\sqrt{8Mpc + M^2 - 8M} + M}{2M}, s_2 = \frac{\sqrt{8Mpc + M^2 - 8M} - M}{2M}$$
When $pc = 1$, $s_2 = 0$. Thus $t_1 = \left\lfloor \frac{\sqrt{8Mpc+M^2-8M}-M}{2M} \right\rfloor$
3. Solve $r(t_1, s, 0) - pc = \frac{Mt_1^2+Mt_1+2Ms+2}{2} - pc = 0$. This equation has one solution. Thus $t_2 = \left\lfloor -\frac{Mt_1^2+Mt_1-2pc+2}{2M} \right\rfloor$

4. Finally, $t_3 = pc - r(t_1, t_2, 0) = -\frac{2\,M\,t_2 + M\,t_1^2 + M\,t_1 - 2\,pc + 2}{2}$

One can verify that for rank $pc = 2$, the trahrhe expressions result in $(0, 0, 1)$ as expected:

- $t_1 = \left\lfloor \frac{\sqrt{M\,(M+8)} - M}{2\,M} \right\rfloor = 0$ for any $M > 1$
- $t_2 = \left\lfloor \frac{2}{2\,M} \right\rfloor = 0$ for any $M > 1$
- $t_3 = \frac{2}{2} = 1$

For rank $pc = M$, the trahrhe expressions result in $(0, 0, M - 1)$ as expected:

- $t_1 = \left\lfloor \frac{\sqrt{M(9\,M-8)} - M}{2\,M} \right\rfloor = 0$ for any $M > 1$

  since $0 < \frac{\sqrt{M(9\,M-8)} - M}{2\,M} < \frac{\sqrt{9\,M^2} - M}{2\,M} = 1$
- $t_2 = \left\lfloor 1 - \frac{1}{M} \right\rfloor = 0$ for any $M \geq 1$
- $t_3 = \frac{2\,M-2}{2} = M - 1$

Note that since the degree of ranking polynomials is less or equal to the depth of the target loop nest, their roots may include radicals (square roots, cubic roots, ...), and such radicals may result in complex numbers. However, the imaginary parts of the entire numerical evaluations of trahrhe expressions are always null. Nevertheless, it means that evaluations must be performed using *complex* floating-point arithmetic.

Only polynomial equations whose degree is at most equal to 4 can be solved symbolically, with exact expressions for roots. The handled uni-variate polynomial equations are built from a multi-variate ranking polynomial, where one index i_k is set as the equation unknown, indices $i_1, ..., i_{k-1}$ are set as symbolic parameters, and indices $i_{k+1}, ..., i_d$ are set to their lexicographic minimum values. Thus, to ensure that such a built equation has a degree less than 4, the ranking polynomial must be such that any index i_k, in any of its monomials, has a degree less than 4, *i.e.*, any monomial is of the form: $a\,i_1^{p_1} i_2^{p_2} ... i_d^{p_d}$ where a is a rational number, and every power p_k is such that $0 \leq p_k \leq 4$.

Loop nests yielding such ranking polynomials are such that the maximum number of nested loops, whose loop trip counts depend on a given index i_k, is less than or equal to 4. For example, both outermost loops of the syr2k loop kernel in Figure 1 depend on index i, yielding a ranking polynomial where index i is of power 2 in some monomial.

However, note that the dependence of loop trip counts regarding indices is transitive: if a loop index j depends on a surrounding loop index i, and an inner loop index k depends on j, then k depends also on i: index i is of power 3, in some monomial of the ranking polynomial. Note also that loops may depend simultaneously on several surrounding loops' indices, as for example in `for(k=0;k<i+j;k++)`.

Hence, the loop nests that can be handled by our method may be of any depth, but are such that the number of nested loops that all depend on a given index is less than or equal to 4. This should be quite sufficient for most cases, regarding the usual loop nest depth and complexity of user codes.

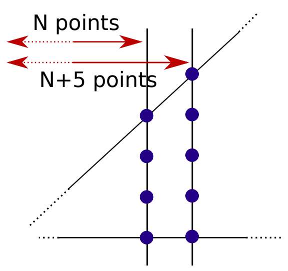


Figure 5. Why exact volumes cannot be reached

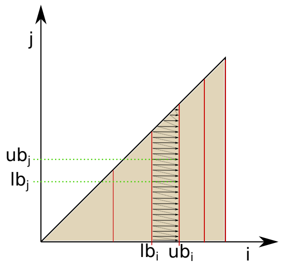


Figure 6. Changed lexicographic order from (i, j) to (j, i) inside slices

This equation solving process has been implemented in our software computing trahrhe expressions. In addition, several issues regarding arithmetic precision and mathematical generalization had to be solved. They are detailed in Section 5.

3.3 Bounds of algebraic tiles

As shown in Section 2, the bounds of algebraic tiles are resulting from slicing the iteration domain into parts of quasi-equal volumes, these latter being sliced in turn into parts of quasi-equal volumes, and so on, from the outermost to the innermost loop dimensions.

The slicing process is driven by a target volume for the resulting parts. Nevertheless, this volume can generally never be reached exactly, since slices are built along one unique dimension, as it is illustrated by Figure 5. Moreover, the target volume cannot be less than the largest cutting hyperplane, since smallest possible slices have their lower bounds equal to their upper bounds. In the proposed approach, the target volume results from dividing the total number of iterations – which is the evaluation of an Ehrhart polynomial – by a chosen divider, as explained in Section 2. Multi-dimensional tiling is achieved dimension per dimension, from the outermost to the innermost loop depth. When handling a given dimension, the associated iterator is considered as if it was the outermost iterator, *i.e.*, as the high-order index regarding the lexicographic order. Thus, a different lexicographic order of the iterator is considered at each step. This is illustrated by Figure 6, where it is shown on the iteration domain of kernel syr2k, that inner slices bounds must be computed "as if" the outer slices were scanned relatively to an interchanged lexicographic order.

More formally, consider a d-depth loop nest whose loop iterators are $(i_1, i_2, \ldots, i_d)$, from the outermost to the innermost. Let us denote by D its iteration domain. First, D is sliced regarding dimension i_1. For this dimension, the lexicographic order defined by the original loop nest is considered. The associated ranking polynomial $r(i_1, i_2, \ldots, i_d)$ is computed and inverted as explained in Subsection 3.2, to get the trahrhe expression, $trahrhe\_i_1(pc)$, defining the value of i_1 where pc iterations have been completed. The target volume, named $TILE\_VOL\_L1$, is computed by dividing the total number of iterations by a given divider DIV_1. Lower and upper bounds of slices are then roughly defined as being:

$$lb_{i_1}(it_1) = trahrhe\_i_1\,(it_1 \times TILE\_VOL\_L1)$$
$$ub_{i_1}(it_1) = trahrhe\_i_1\,((it_1 + 1) \times TILE\_VOL\_L1) - 1$$

where $it_1 \in [0...DIV_1 - 1]$.

Next, each resulting slice characterized by a value it_1 is sliced in turn regarding dimension i_2. However, since each slice it_1 has to be sliced along dimension i_2, i_2 has now to be considered as the outermost iterator regarding the lexicographic order. Hence, for each outermost slice defined by $D \cap \{lb_{i_1}(it_1) \leq i_1 \leq ub_{i_1}(it_1)\}$, the associated ranking polynomial $r(\boldsymbol{i_2}, i_1, i_3, \ldots, i_d)$ is computed and inverted, to get the trahrhe expression, $trahrhe\_i_2(pc, lb_{i_1}(it_1), ub_{i_1}(it_1))$, defining the value of i_2 where pc iterations have been completed. The target volume $TILE\_VOL\_L2$ is computed by dividing the total number of iterations of the current outer slice it_1 by a given divider DIV_2. Lower and upper bounds of slices are then roughly defined as being:

$$\begin{aligned}&lb_{i_2}(it_2, lb_{i_1}(it_1), ub_{i_1}(it_1))\\ &= trahrhe\_i_2\left(it_2 \times TILE\_VOL\_L2, lb_{i_1}(it_1), ub_{i_1}(it_1)\right)\\ &ub_{i_2}(it_2, lb_{i_1}(it_1), ub_{i_1}(it_1))\\ &= trahrhe\_i_2\,((it_2 + 1) \times TILE\_VOL\_L2,\\ &\qquad\qquad lb_{i_1}(it_1), ub_{i_1}(it_1)) - 1\end{aligned}$$

where $it_2 \in [0...DIV_2 - 1]$.

The same process is repeated until the deepest tiling dimension, by considering at each step a different lexicographic order, where the current sliced dimension index is set as the outermost iterator.

4 Applicability of algebraic tiling

Regarding its validity, algebraic tiling follows the same rules as standard tiling: algebraic tiling may be applied when it is valid regarding data dependences. Thus a prior affine transformation of the target loop nest may be required. In practice, we use the dependence analysis process implemented in the automatic loop optimizer Pluto. Moreover, to promote vectorization and data locality, loops may be interchanged, as it is performed by Pluto.

Regarding vectorization of the innermost loops, the non-constant algebraic tile bounds may prevent compilers to generate the fastest vector code. In such cases, a hybrid tiling approach that combines standard and algebraic tiling seems the best strategy for getting better code : the outer dimension that is linked to the innermost vectorizable loop is actually sliced using a constant value (e.g. 32 or 64), as it is achieved with standard tiling, while the other dimensions are sliced with non-constant algebraic bounds. In the future, we expect that algebraic tiling will take advantage of hardware extensions as SVE (Scalable Vector Extension) [23] to result in more efficient vector code.

Algebraic tile bounds are obviously not linear and may stay parameterized by the dividers and the problem size. Parameterized tiling in the standard approach also yields non-linear bounds, but those are polynomials while ours are algebraic expressions. In [12], Iooss *et al.* propose parametric tiling based on one unique parameter, which enables further linear code transformations. By contrast, algebraic tiling prevents any chance for further linear transformations, but opens the field to managed non-linear transformations.

Algebraic tiling is also a parametric tiling approach whose parameters are iteration count dividers rather than tile sizes. The importance of parametric tiling is exemplified by the ATLAS system [25] for empirical tuning of BLAS kernels. ATLAS uses parametrically tiled BLAS kernels that are repeatedly executed on the target architecture for different problem sizes using an empirical search strategy that varies the tile sizes. A similar search could be performed by varying the dividers in place of the tile sizes.

Another main difference compared to standard tiling is the required complex floating-point arithmetic for computing the bounds, which are algebraic expressions. Such computations may require high-order precision in some cases, whenever large integer values are involved – typically large loop bounds related to problem sizes – or very small real values. Such requirements are still essential, even if the final used result is the integer part (floor) of the computed real value. For this purpose, we implemented several related features in our software: multiple-precision complex floating-point computations, dichotomous search of the polynomial roots, adjustment of the solution using the ranking polynomials, verification code, and error handling functions computing trahrhe expressions.

5 The Trahrhe software and the new -atiling flag of Pluto

The Trahrhe software has been written as a bash script with a few parsers in C, that makes intensive use of the computer-algebra system *Maxima*[3] and of *iscc*[4], the interactive interface to the *barvinok* counting library and the *isl* integer set library. It is freely available[5]. It takes as input a parameterized iteration domain in the iscc syntax, and several possible flags as:

- -t*n*: computes the trahrhe expressions required to tile the *n* outermost loops;
- -e: generates a C header file including all the definitions of functions required to apply algebraic tiling to a loop nest, as shown in Figure 4. It may be included by any C/C++ code for algebraic tiling optimization. Complex floating-point operations are performed with C type `long double`. Function definitions include verification

[3]http://maxima.sourceforge.net
[4]https://repo.or.cz/barvinok.git
[5]https://webpages.gitlabpages.inria.fr/trahrhe

instructions that stop the program whenever incoherent values arise in the evaluations of trahrhe expressions; arithmetic precision. This program compares every tuple of original iterators to the corresponding evaluation of the trahrhe expressions. It may be used as a code pattern to algebraic-tile-transform a given target code that includes an equivalent loop nest;

Ehrhart and ranking polynomials are computed by invoking iscc. To solve symbolic polynomial equations, our software invokes the computer algebra system Maxima.

We are currently implementing algebraic tiling in the polyhedral compiler Pluto. By using flag `-atiling`, algebraic tiling is automatically applied to loops identified by the pragma `#pragma scop` in the input C source file. Additionally, the Trahrhe software is then invoked by Pluto to generate the required header files where the relevant trahrhe functions are generated in C.

6 Experiments

Some first experiments were conducted on 8 programs from the polybench benchmark suite v4.2 with data size EXTRALARGE.

All programs were compiled using gcc 10.3.0 with flags `-O3 -march=native -fopenmp -lm`. Standard (hyper-)rectangular tiling and OpenMP parallelization were automatically applied on the 8 programs thanks to Pluto, which performed simultaneously some loop transformations to promote data locality and vectorization. On each hardware platform, tile sizes providing the lowest execution times were selected by performing an exhaustive search among sizes that are powers of 2, from 1 to 2048. When very lower execution times were reached with algebraic tiling, standard tile sizes that are as close as possible to the size of the best performing algebraic tiles were also tested, although algebraic tile sizes are not generally constant.

Note that size 1 for a tiling depth means that the related loop was actually not tiled. The best performing tile sizes were found separately by running programs with schedule static and with schedule dynamic of OpenMP. With standard tiling, schedule dynamic may provide better performance, since it aims to improve load-balancing by dynamic distribution of iterations among the threads. It is particularly significant when the iteration domain is non-rectangular, as it is for kernel syr2k (see Figure 2). However, when the iteration domain is rectangular, schedule dynamic generally yields lower performance than schedule static, due to its underlying time overhead.

Regarding algebraic tiling, dividers providing the lowest execution times were also selected through an exhaustive search among powers of 2, from 1 to 2048, excepting for the dimension of the parallel loop, where the search was performed among dividers that are multiple of the number of threads (up to 10 times the number of threads). The same loop transformations as the ones generated by Pluto for

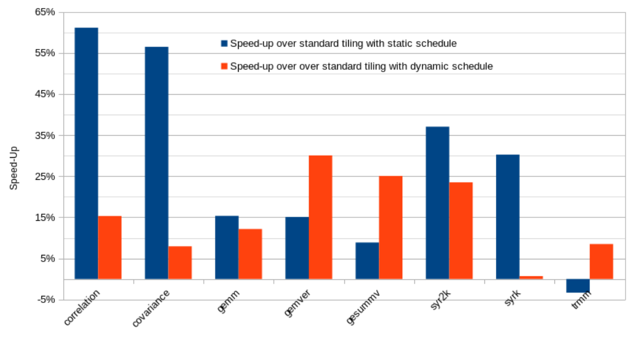


Figure 7. Speed-ups resulting from comparing algebraic tiling vs standard tiling with *non-vectorized codes* (64 threads)

standard tiling (skewing or interchange) were also applied with algebraic tiling. Note that programs with algebraic tiling were exclusively run with schedule static of OpenMP, since quasi-perfect load balancing is obviously expected. Hybrid tiling (standard + algebraic) was applied in some cases, in order to improve vectorization.

The programs were run on 2 × 32-core AMD Zen2 EPYC 7452 @ 2.35 GHz with Turbo-Boost and Hyperthreading deactivated, using 64 parallel threads. Runs were performed after having set the environment variable OMP_PROC_BIND=true to avoid thread migration and bind the threads to processor cores.

The best performing tile sizes and dividers providing the lowest execution times are reported in Table 2. Note that only multiples of 64 are used as the outermost slices' dividers. When tiling is hybrid, symbol «×» precedes the standard tile size that was selected, additionally to the algebraic dividers.

Speed-ups were computed by comparing execution times of programs optimized with algebraic tiling (that include the time spent in computing the algebraic tile bounds) against execution times of programs optimized with standard tiling, using the best performing dividers and tile sizes:

$$speed{-}up = \frac{time\_standard\_tiling - time\_algebraic\_tiling}{time\_standard\_tiling}$$

The resulting speed-ups against standard rectangular tiled programs run using schedule either static or dynamic are reported in Figures 7 and 8. For Figure 7, automatic vectorization was deactivated using flag `-fno-tree-vectorized` of gcc, while it was activated for Figure 8. One can observe that vectorization has a significant influence in the lower speed-up amount that is mostly obtained with algebraic tiling, even when hybrid tiling is applied. It shows that current microprocessor SIMD units and compilers seem not well suited for inner parallel loops of varying sizes, although algebraic tiling mostly outperforms standard tiling, particularly when iteration domains are not rectangular.

Table 2. Best performing tile sizes and trahrhe dividers with size EXTRALARGE_DATASET and 64 threads

Program	without vectorization			with vectorization		
	static	dynamic	algebraic	static	dynamic	algebraic
correlation	$4 \times 2048 \times 8$	$8 \times 64 \times 128$	128, 1, 2048	$16 \times 128 \times 16$	$16 \times 128 \times 8$	128, 1, 2048
covariance	$4 \times 32 \times 64$	$8 \times 2048 \times 1$	64, 1, 32	$16 \times 128 \times 128$	$16 \times 128 \times 128$	64, 1024×128
gemm	$32 \times 32 \times 32$	$32 \times 16 \times 256$	64, 128, 1	$32 \times 64 \times 64$	$32 \times 64 \times 32$	$64 \times 1 \times 256$
gemver	$5 \times 2048 \times 1$	$16 \times 1 \times 1$	384, 1024	$4 \times 2048 \times 1$	$16 \times 1024 \times 1$	128, 128, 64
gesummv	$22 \times 2 \times 1$	$44 \times 1 \times 1$	512, 64, 1	$5 \times 8 \times 1$	$43 \times 1024 \times 1$	192, 1024, 1
syr2k	$16 \times 1 \times 1$	$4 \times 8 \times 1$	128, 64, 1	$32 \times 64 \times 1$	$8 \times 64 \times 1$	128, 32, 16
syrk	$16 \times 16 \times 1$	$4 \times 16 \times 1$	64, 4, 512	$32 \times 128 \times 1$	$8 \times 128 \times 1$	64, 128×16
trmm	$14 \times 64 \times 1$	$8 \times 2048 \times 1$	64, 16, 2	$41 \times 32 \times 16$	$41 \times 16 \times 16$	128, 256×64

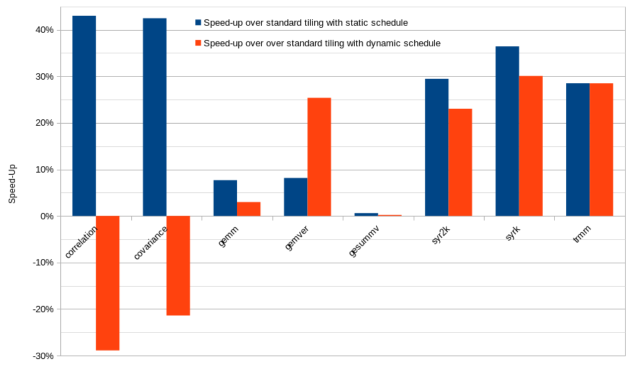


Figure 8. Speed-ups resulting from comparing algebraic tiling vs standard tiling with *vectorized codes* (64 threads)

7 Related work

Tiling has been a very productive research topic for nearly thirty years [7, 13, 18, 19, 26]. However, tiling techniques that have been proposed are all based on partitioning the iteration domain into tiles, that may be (hyper-)rectangular or non-rectangular (hexagonal, trapezoidal, etc.), but always defined by their edge directions and sizes. Thus, their bounds are always expressed as (min or max of) affine functions, or polynomial functions when parameterized. To our knowledge, tiles delimited by algebraic bounds were never proposed.

There have been many proposals regarding stencil computations, which are an important class of programs that occurs in a variety of scientific applications, and for which tiling is particularly difficult. Stencil computations mostly require skewing parallelizing transformations [1, 18, 27] that have to be handled specifically [2, 9, 11, 17].

In [10], Hartono *et al.* propose PrimeTile, a parametric multi-level tiler for imperfect loop nests. Their tiler is “multi-level” as outer tiles may also be tiled at deeper levels. The authors use this approach to distinguish full tiles from partial tiles, and thus to avoid large partial tiles by deeper tiling of areas along borders of the iteration domain. Note that such strategy may be purposeless with algebraic tiling, since (full) algebraic tiles are built by considering the volume of the whole target iteration domain from the beginning.

Jiménez *et al.* [14] propose a code generation technique for register tiling of non-rectangular iteration spaces. They generate code that traverses the bounding box of the tile iteration space to enable parameterized tile sizes. They apply index-set splitting to tiled code to traverse parts of the tile space that include only full tiles. Their approach involves less overhead in the loop nest that visits the full tiles, at the price of significant code expansion.

In their work, Sato *et al.* [22] address explicitly load balancing through an analytic model. They model the load balancing issue analytically and combine it with empirically autotuning the loop tile size for many-core CPUs, through an iterative compilation framework. However, they only address standard rectangular tiling and obtain speed-ups against a baseline which is all tile sizes equal to 32, while in our approach, we obtain significant speed-ups against the best performing rectangular tile sizes.

Sakellariou in [21] proposes a compile-time scheme for partitioning non-rectangular loop nests, where the minimization of load imbalance is based on symbolic cost estimates. In [16], Kejariwal *et al.* present a geometric approach for partitioning N-dimensional non-rectangular iteration spaces. They partition an iteration space along the axis corresponding to the outermost loop to achieve a near-optimal partition. Kafri and Abu Sbeih in [15] focus on static decomposition of perfect triangular iteration spaces to achieve load balancing, by partitioning a triangular iteration space of a loop nest along the outermost loop index.

8 Conclusion

When compared to standard (hyper-)rectangular tiling, algebraic tiling provides significant performance improvements, despite the relatively complex algebraic expressions that have to be evaluated at runtime to determine the tile bounds. It clearly confirms that load balancing among computing hardware units is crucial for performance. Moreover, with this technique, dynamic scheduling becomes purposeless for the handled loops.

Algebraic tiling may be further extended and adapted, particularly for heterogeneous computing platforms by associating performance ratios to the target computing units. More generally, trahrhe expressions open a new field for loop optimizations based on iteration counts. Besides tiling, we plan to investigate other promising directions such as algebraic scheduling.

References

[1] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. 2001. Optimal Semi-oblique Tiling. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01)*. ACM, New York, NY, USA, 153–162. https://doi.org/10.1145/378580.378619

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

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08*. ACM, 101–113.

[4] 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 (ICS '96)*. New York, NY, USA, 278–285.

[5] Philippe Clauss, Ervin Altintas, and Matthieu Kuhn. 2017. Automatic Collapsing of Non-Rectangular Loops. In *Parallel and Distributed Processing Symposium (IPDPS), 2017*. IEEE International, Orlando, United States, 778 – 787. https://doi.org/10.1109/IPDPS.2017.34

[6] Philippe Clauss and Benoît Meister. 2000. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *SIGARCH Comput. Archit. News* 28, 1 (March 2000), 11–19.

[7] Jack Dongarra and Robert Schreiber. 1990. *Automatic Blocking of Nested Loops*. Technical Report. Knoxville, TN, USA.

[8] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[9] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 66, 10 pages. https://doi.org/10.1145/2581122.2544160

[10] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. 2009. Parametric Multi-level Tiling of Imperfectly Nested Loops. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 147–157. https://doi.org/10.1145/1542275.1542301

[11] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A Stencil Compiler for Short-vector SIMD Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2464996.2467268

[12] Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. 2021. Monoparametric Tiling of Polyhedral Programs. *Int J Parallel Prog* 49 (2021), 376–409. https://doi.org/10.1007/s10766-021-00694-2

[13] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 319–329. https://doi.org/10.1145/73560.73588

[14] Marta Jiménez, José M. Llabería, and Agustín Fernández. 2002. Register Tiling in Nonrectangular Iteration Spaces. *ACM Trans. Program. Lang. Syst.* 24, 4 (July 2002), 409–453. https://doi.org/10.1145/567097.567101

[15] Nedal Kafri and Jawad Abu Sbeih. 2008. Simple Near Optimal Partitioning Approach to Perfect Triangular Iteration Space. In *Proceedings of the 2008 High Performance Computing & Simulation Conference.*

[16] Arun Kejariwal, Paolo Alberto, Alexandru Nicolau, and Constantine D. Polychronopoulos. 2005. A Geometric Approach for Partitioning N-dimensional Non-rectangular Iteration Spaces. In *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing (LCPC'04)*. Springer-Verlag, Berlin, Heidelberg, 102–116.

[17] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 235–244. https://doi.org/10.1145/1250734.1250761

[18] Monica S. Lam and Michael E. Wolf. 2004. A Data Locality Optimizing Algorithm. *SIGPLAN Not.* 39, 4 (April 2004), 442–459. https://doi.org/10.1145/989393.989437

[19] J. Ramanujam and P. Sadayappan. 1991. Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/125826.125893

[20] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout, and Sanjay Rajopadhye. 2012. Parameterized Loop Tiling. *ACM Trans. Program. Lang. Syst.* 34, 1, Article 3 (May 2012), 41 pages. https://doi.org/10.1145/2160910.2160912

[21] Rizos Sakellariou. 1997. A Compile-Time Partitioning Strategy for Non-Rectangular Loop Nests. In *Proceedings of the 11th International Symposium on Parallel Processing (IPPS '97)*. IEEE Computer Society, Washington, DC, USA, 633–637.

[22] Yukinori Sato, Tomoya Yuki, and Toshio Endo. 2019. An Autotuning Framework for Scalable Execution of Tiled Code via Iterative Polyhedral Compilation. *ACM Trans. Archit. Code Optim.* 15, 4, Article 67 (Jan. 2019), 23 pages. https://doi.org/10.1145/3293449

[23] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (2017), 26–39. https://doi.org/10.1109/MM.2017.35

[24] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* 48, 1 (2007), 37–66.

[25] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC '98)*. IEEE Computer Society, Washington, DC, USA, 1–27. http://dl.acm.org/citation.cfm?id=509058.509096

[26] Michael Wolfe. 1989. Iteration Space Tiling for Memory Hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 357–361. http://dl.acm.org/citation.cfm?id=645818.669220

[27] David Wonnacott. 2002. Achieving Scalable Locality with Time Skewing. *International Journal of Parallel Programming* 30, 3 (01 Jun 2002), 181–221. https://doi.org/10.1023/A:1015460304860