A Compiler Algorithm to Guide Runtime Scheduling

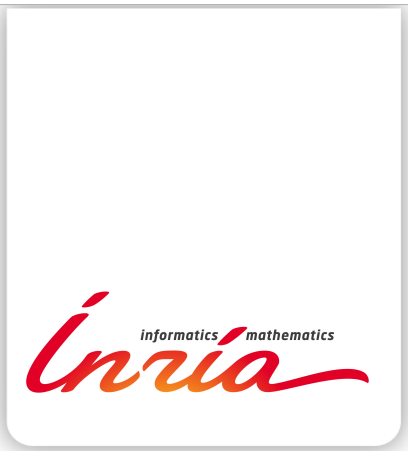
Christophe Alias, Samuel Thibault, Laure Gonnord

A Compiler Algorithm to Guide Runtime Scheduling

Christophe Alias, Samuel Thibault, Laure Gonnord

RESEARCH REPORT
N° 9315
December 2019
Project-Team Cash & Storm

ISSN 0249-6399 ISRN INRIA/RR--9315--FR+ENG



A Compiler Algorithm to Guide Runtime Scheduling

Christophe Alias*, Samuel Thibault†, Laure Gonnord‡

Project-Team Cash & Storm

Research Report n° 9315 — December 2019 — 12 pages

Abstract: Task-level parallelism is usually exploited by a runtime scheduler, after tasks are mapped to processing units by a compiler. In this report, we propose a compilation-centric runtime scheduling strategy. We propose a complete compilation algorithm to split the tasks in three parts, whose properties are intended to help the scheduler to take the right decisions. In particular, we show how the polyhedral model may provide a precious help to compute tricky scheduling and parallelism informations. Our compiler is available and may be tried online at `http://foobar.ens-lyon.fr/kut`.

Key-words: Compilation, automatic parallelization, polyhedral model, runtime scheduling

* Inria/ENS-Lyon/UCBL/CNRS
† Université de Bordeaux / INRIA / CNRS
‡ Inria/ENS-Lyon/UCBL/CNRS

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Un algorithme de compilation pour guider l'ordonnancement dynamique

Résumé : Le parallélisme de tâches est habituellement exploité par un ordonnanceur dynamique, après que les tâches aient été compilées sur les différentes unités d'exécution. Dans ce rapport, nous proposons une nouvelle stratégie d'ordonnancement qui donne une place centrale au compilateur. Nous proposons un algorithme de compilation complet et un compilateur pour diviser les tâches en trois parties, dont les propriétés permettent à l'ordonnanceur de prendre les bonnes décisions. En particulier, nous montrons que le modèle polyédrique permet de construire et de raffiner ce type de stratégie.

Mots-clés : Compilation, parallélisation automatique, modèle polyédrique, ordonnancement dynamique

1 Introduction

Since the end of Dennard scaling, computer architects are striving to increase the computing power under a constant energy budget, power efficiency is the new performance measure. Hence the rise of heterogeneous architectures with multicore processors and power efficient hardware accelerators (typically GPUs). Beside the inherent difficulty of parallel programming, the programmer is left to a myriad of standards with different levels of abstraction, to control both *coarse-grain* and *fine-grain* parallelism. The general philosophy is to exploit coarse-grain parallelism *at runtime* and fine-grain parallelism at *compile-time*: tasks are expressed at the langage-level, then a runtime system schedules the tasks on the processing units and rules the data transfers across processing units. The compiler is then responsible to map properly each task to the processing units. However, runtime scheduling suffers from many drawbacks, notably a lack of global view on the computation – required to exploit to the best the computing capabilities.

In this report, we propose to put the compiler at the center of the process. At compile-time, we analyse the task dependences and we produce a task classification which guide the runtime scheduling. Specifically, this report makes the following contributions:

- A compile-time/runtime strategy to improve task scheduling on CPUs and GPUs.
- A compiler algorithm, which analyses an HPC application and partitions the task instances according to the scheduling strategy and the type of processing units to be used at runtime (CPU or GPU).

This report is structured as follows. Section 2 outlines our compilation/scheduling strategy and introduces the polyhedral model, the compilation framework used to design our compiler algorithm. Section 3 describes our compiler algorithm. Section 4 concludes this report and draws research perspectives. Finally the annex presents the input of our compiler, `kut`.

2 Preliminaries

This section outlines the features of StarPU, the runtime system considered in this report, our compilation/runtime strategy, and then the polyhedral model, the general framework in which we express our compilation algorithm.

2.1 Runtime scheduling

StarPU We focus on StarPU [1], a runtime system providing a high-level, unified execution model tightly coupled with an expressive data management library. StarPU features several scheduling strategies, which can be selected in a simple manner by the programmer. In particular, the area-bound strategy relies on an affine expression of the tasks *bottom-level* to ensure a progression of the tasks close to the critical path. The bottom-level of a task is the time to completion, assuming unbounded ressources. Conceptually, it can be defined as an ASAP schedule, starting from the last task and reverting the dependences. Figure 1.(a) depicts a loop kernel with BLAS tasks COPY (first statement, S), GEMM (second statement, T) and TRSM (third statement, U). The expanded dependence graph is given in (b). The last task ($\circ$ (TRSM), $i = 7, j = 6$) has a bottom-level 0. Assuming latencies $\delta(\text{COPY}) = 1$ $\delta(\text{GEMM}) = 2$ and $\delta(\text{TRSM}) = 2$, all the tasks above the red line have a bottom-level lower than $\ell_{\text{red}} = 8$. One of our contributions is to automate the computation of the bottom-level as an affine function of loop counters i and j. We will see that, on that example, our compilation algorithm computes the bottom-level $\beta_{\text{red}}(S, i) = 4N - 2i - 2, \beta_{\text{red}}(T, i, j) = 4N - 2(i + j) - 4$ and $\beta_{\text{red}}(U, i) = 4N - 4i - 4$.

Compile-time / runtime strategy We follow the following strategy. The compiler analyzes the loop nest, given the CPU/GPU number and latency for each task. Starting from the last task, the compiler schedules ASAP using the GPU task latency until reaching as many parallel tasks as the number of GPUs. This gives the *red frontier* ℓ_{red} depicted on Figure 1.(b). From the red frontier, the compiler schedules ASAP using average latencies CPU/GPU until reaching as many parallel tasks as #CPU + #GPU. This gives the *green frontier* depicted on Figure 1.(b). The compiler produces a function deciding if a task, given the loop counters, is beyond the red frontier. It does the same thing for the green frontier.

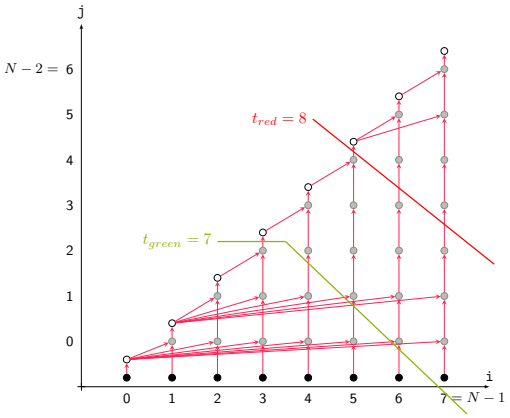
At runtime, at the beginning, the scheduler deals with a few tasks, scheduled with a HEFT-like (dmdas) strategy. When there are more ready tasks than processing units, the scheduler uses an area-bound strategy to maximize the acceleration rather than prioritizing the critical path. The scheduler uses the bottom-level to ensure the progression of the tasks close to the critical path. *Beyond the green frontier*, the tasks are kept to be scheduled later. When there is no more tasks to schedule, the task kept are scheduled with an HEFT-like (dmdas) strategy. *Beyond the red frontier*, (less parallel tasks than GPUs) all the tasks are scheduled on the GPUs.

Computing the bottom-levels and the amount of parallelism at each level requires a precise static analysis of the program which is possible thanks to the *polyhedral model* described in the next section.

2.2 Polyhedral model

for $i := 0$ **to** $N-1$
- $\mathbf{s} := \mathbf{b}_i$;
  for $j := 0$ **to** $i-1$
- $\mathbf{s} := \mathbf{s} - \mathbf{A}_{ij}\mathbf{x}_j$;
- $\mathbf{x} := \mathbf{A}_{ii}^{-1}\mathbf{s}$;

(a) Forward Substitution kernel



(b) Flow dependences

Figure 1: **Motivating example: forward substitution kernel.** (a) depicts a polyhedral kernel (• = copy, • = gemm, ∘ = trsm), then (b) gives the polyhedral representation of loop iterations with $N = 8$ and flow dependences (red arrows).

The polyhedral model [7] is a general framework to design loop transformations, historically geared towards source-level automatic parallelization [7] and data locality improvement [2]. It abstracts loop iterations as a union of convex polyhedra – hence the name – and data accesses as affine functions. This way, precise – iteration-level – compiler algorithms may be designed (dependence analysis [5], scheduling [6] or loop tiling [2] to quote a few) .

Program model The polyhedral model manipulates loop kernels – referred to as *polyhedral programs* – which consist of nested `for` loops and `if` conditions manipulating arrays and scalar variables, such that loop bounds, `if` conditions, and array access functions are *affine expressions* of surrounding loops counters and structure parameters. This way, the control is static, it only depends on the input size (the structure parameters, e.g. N), and may be analysed at compile-time. With polyhedral programs, each iteration of a loop nest is uniquely represented by the vector of enclosing loop counters $\vec{i}$. The execution of a program statement S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i} \rangle$. The set $\mathcal{D}_S$ of iteration vectors is called the *iteration domain* of S. Figure 1 depicts a polyhedral program (a) with the iteration domains (b).

Dependences & scheduling With the polyhedral model, *data dependences* may be computed exactly at compile-time and represented with a Presburger relation $\langle S, \vec{i} \rangle \rightarrow \langle T, \vec{j} \rangle$ iff $(\vec{i}, \vec{j}) \in \Phi_{S,T}$. $\Phi_{S,T}$ is usually abstracted as a rational polyhedron, called *dependence polyhedron*. Dependences are usually classified as flow (write to read), anti (read to write) and output (write to write). The size of the longest dependence path is refered to as $\lambda^*(\rightarrow)$. Figure 1.(b) represents the flow data dependences across the iterations.

A *schedule* maps each operation $\langle S, \vec{i} \rangle$ to a timestamp $\theta_S(\vec{i}) \in (\mathbb{Z}^d, \ll)$. Conceptually, a schedule specifies a program transformation by mapping each iteration to its new iteration in the target program. The latency of a schedule, $\lambda(\theta)$, is the number of steps before completion. Its bounded by the longest dependence path: $\lambda(\theta) \geq \lambda^*(\rightarrow)$. When $\lambda(\theta) = \lambda^*(\rightarrow)$, the schedule is said to be *free* or optimal. In the polyhedral model, schedules are affine mappings and may be derived at compile-time. In particular, the GREEDY algorithm [6] produces an asymptotically optimal schedule [4] – asymptotically optimal $\equiv$ when structure parameters are big enough – and will be used in this report to produce an affine "nearly ASAP" schedule.

3 Our compilation algorithm

Our goal is to derive the red and green frontiers ℓ_{red} and ℓ_{green} and the statement-wise predicates below_red_frontier$(S, \vec{i})$ and below_green_frontier$(S, \vec{i})$ to check whether a frontier has been reached by an operation $(S, \vec{i})$. To do so, we need to compute, in the following order:

1. $\beta_{\text{red}}(S, \vec{i})$, the bottom level from the last operation, assuming GPU latencies δ_{red} – called *red bottom-level*.

2. $\pi_{\text{red}}(\ell) = \text{card}\{x \mid \beta_{\text{red}}(x) = \ell\}$, the number of operations running concurrently at red bottom-level ℓ.

3. $\ell_{\text{red}} = \min_{\ll}\{\ell \mid \pi_{\text{red}}(\ell) \geq \#GPU\}$, the red frontier. There is no way to automate this statically; it is computed from $\pi_{\text{red}}(\ell)$ *at runtime* by enumerating the first values of ℓ. Details are given in Algorithm 2. Hence, ℓ_{red} is considered as an *input parameter* in the next steps.

4. $\beta_{\text{green}}(\ell_{\text{red}}, S, \vec{i})$, the bottom level *from the red frontier*, assuming average CPU/GPU latencies δ_{green} – called *green bottom-level*.

5. $\pi_{\text{green}}(\ell_{\text{red}}, \ell)$, the number of operations *above the red frontier* running concurrently at green bottom-level ℓ, $\pi_{\text{green}}(\ell_{\text{red}}, \ell) = \text{card}\{x \mid \beta_{\text{green}}(x) = \ell \wedge \beta_{\text{red}}(x) \gg \ell_{\text{red}}\}$

6. $\ell_{\text{green}}(\ell_{\text{red}}) = \min_{\ll}\{\ell \mid \pi_{\text{green}}(\ell_{\text{red}}, \ell) \geq \#CPU + \#GPU\}$, the green frontier. Again, this is compute at runtime, once ℓ_{red} is known. Details are given in Algorithm 2.

Except steps 3 and 6 (computing the final red and green frontiers), all these steps are computed at *compile-time* by the Algorithm 1 described in Section 3.1. Then, Section 3.2 describes the Algorithm 2 to generate the code computing the red and green frontiers at runtime and defining the final predicates below_red_frontier$(S, \vec{i})$ and below_green_frontier$(S, \vec{i})$.

3.1 Bottom-levels and parallelism volume

Algorithm 1 computes the bottom levels $\beta_{\text{red}}, \beta_{\text{green}}$ and the parallelism volume $\pi_{\text{red}}, \pi_{\text{green}}$ from the polyhedral dependence graph $\rightarrow$ and the dependence delays δ_{red} (GPU) and δ_{green} (average CPU/GPU). By definition, a bottom-level is obtained by scheduling with an ASAP strategy (ASAP scheduling) the reverse dependence graph $\rightarrow^{-1}$. Reverting a dependence graph is direct: it suffices to swap the source and the target operations (Step 1). The polyhedral counterpart of ASAP scheduling is the GREEDY scheduling algorithm [6] described in Section 2.

Red part It suffices to feed the greedy scheduling algorithm with the reverted dependences $\rightarrow_{\text{red}}$ and the GPU delays $\delta_{\text{red}} = \delta_{\text{gpu}}$ to obtain β_{red} (Step 2). Here, with $\delta_{\text{gpu}}(S) = 1$ and $\delta_{\text{gpu}}(T) = \delta_{\text{gpu}}(U) = 2$, we obtain: $\beta_{\text{red}}(S, i) = 4N - 2i - 2, \beta_{\text{red}}(T, i, j) = 4N - 2(i + j) - 4$ and $\beta_{\text{red}}(U, i) = 4N - 4i - 4$. We point out that we obtain a monodimensional mapping, since the free latency is linear: $\lambda^*(\rightarrow) = 4N$. However, when $\lambda^*(\rightarrow)$ is more than linear (e.g. $O(N^2)$), we get a multidimensional bottom-level. Our algorithm does not make any assumption on the dimension of the bottom-level, and is thus general enough to handle this case.

Then, we compute $\pi_{\text{red}}(\ell)$, the number of operations running concurrently at bottom-level ℓ, $\pi_{\text{red}}(\ell) = \text{card}\{x \mid \beta_{\text{red}}(x) = \ell\}$ (Step 5). Writing $\mathcal{F}_{\text{S}}(\ell)$ the instances of statement S at bottom-level ℓ, $\mathcal{F}_{\text{S}}(\ell) = \{\vec{i} \mid \beta_{\text{red}}(S, \vec{i}) = \ell \wedge \vec{i} \in \mathcal{D}_S\}$, we have:

$$\pi_{\text{red}}(\ell) \quad = \sum_S \text{card}\, \mathcal{F}_{\text{S}}(\ell) \tag{1}$$

For each statement S, $\mathcal{F}_{\text{S}}(\ell)$ is an integer polyhedron parametrized by ℓ (and the program parameters, here N), whose volume $\text{card}\, \mathcal{F}_{\text{S}}(\ell)$ might be computed with [3]. The result is a piecewise Ehrhart polynomial:

$$\begin{array}{ll}
\text{card}\, \mathcal{F}_{\text{copy}}(\ell) = [1\ 0]_\ell & \forall 2N \leq \ell \leq 4N - 2 \\
\text{card}\, \mathcal{F}_{\text{gemm}}(\ell) = \left\{ \begin{array}{l} [0\ 0\ \frac{1}{2}\ 0]_\ell + [\frac{1}{4}\ 0]_\ell \ell \\ [-1\ 0\ -\frac{1}{2}\ 0]_\ell + [-\frac{1}{4}\ 0]_\ell \ell + [1\ 0]_\ell N \end{array} \right. & \begin{array}{l} \forall 2 \leq \ell \leq 2N - 2 \\ \forall 2N - 2 < \ell \leq 4N - 6 \end{array} \\
\text{card}\, \mathcal{F}_{\text{trsm}}(\ell) = [1\ 0\ 0\ 0]_\ell & \forall 0 \leq \ell \leq 4N - 4
\end{array}$$

The notation $[a_0 \dots a_{n-1}]_\ell$ is called a *periodic number*. It is interpreted as $a_{\ell \mod n}$, the coefficient selected rotates depending on ℓ. Likewise, the final summation $\pi_{\text{red}}(\ell)$ is a piecewise Ehrhart polynomial obtained by separating the pieces common to each mapping. There is no way to compute statically the red frontier $\ell_{\text{red}} = \min_{\ll}\{\ell \mid \pi_{\text{red}}(\ell) \geq \#GPU\}$. Even though, it is much more cheaper and easier to compute it dynamically, simply by enumerating the red bottom levels increasingly. Again, the details are given in the Algorithm 2 described in the next section.

Green part The green bottom-level is computed from the red frontier ℓ_{red}, which is only known at runtime. Hence, we have to keep ℓ_{red} as a parameter and let all the intermediate results depend on it. Step 3 substracts from the original dependence graph $\rightarrow$ the operations below the red frontier. Then, the green bottom-level is computed from this new dependence graph with average CPU/GPU latencies δ_{green} (step 4). Here, we obtain the mapping $\beta_{\text{green}}(\ell_{\text{red}}, S, i) = 12N - 6i - 3\ell_{\text{red}} - 9$ and $\beta_{\text{green}}(\ell_{\text{red}}, T, i, j) = 12N - 6(i + j) - 3\ell_{\text{red}} - 15$. Finally, step 6 computes the parallelism volume similarly to step 5. Again, the obtained mapping, π_{green}, depends of the red frontier ℓ_{red}.

3.2 Code generation: finding red and green frontiers

The final code is generated with the Algorithm 2. The red frontier is computed at runtime with the function red_frontier (lines 2 to 10). The domain range β_{red} is a general Presburger set. We propose to iterate on an integral approximation using [8], then to guarding the test with the actual expression of range β_{red} (line 5, which could be long). In particular, we generate the code to evaluate a general piecewise Ehrhart polynomial. Once the red frontier is known (line 34), the green frontier can be computed similarly (line 35). The predicate to check whether a frontier has been reached by an operation $(S, \vec{i})$ is simply $\beta_{\text{red}}(S, \vec{i}) \lessdot \ell_{\text{red}}$ for the red frontier (lines 22 to 24) and $\beta_{\text{green}}(\ell_{\text{red}})(S, \vec{i}) \lessdot \ell_{\text{green}}$ for the green frontier (lines 27 to 29) Here, we obtain $\ell_{\text{red}} = 8$ and $\ell_{\text{green}} = 7$, the red and green frontiers are depicted on figure 1.

Algorithm 1: Compute the bottom-level (green and red) and the related parallelism

Input : A program, represented as a polyhedral dependence graph $\rightarrow$ and the dependence delays δ_{red} (GPU) and δ_{green} (average CPU/GPU).
Output: The bottom-levels $\beta_{\text{red}}, \beta_{\text{green}}$ and the related parallelism level $\pi_{\text{red}}, \pi_{\text{green}}$

/* $\beta_{\texttt{red}}$ */
1 $\rightarrow_{\text{red}} := \{(y, x) \mid x \rightarrow y\}$;
2 $\beta_{\text{red}} :=$ GREEDY$(\rightarrow_{\text{red}}, \delta_{\text{red}})$;
/* $\beta_{\texttt{green}}$ */
3 $\rightarrow_{\text{green}} (\ell_{\text{red}}) := \rightarrow_{\text{red}} \cap \{(x, y) \mid \beta_{\text{red}}(x) \gg \ell_{\text{red}} \wedge \beta_{\text{red}}(y) \gg \ell_{\text{red}}\}$;
4 $\beta_{\text{green}}(\ell_{\text{red}}) :=$ GREEDY$(\rightarrow_{\text{green}} (\ell_{\text{red}}), \delta_{\text{green}})$;
/* $\pi_{\texttt{red}}, \pi_{\texttt{green}}$ */
5 $\pi_{\text{red}}(\ell) := \text{card}\{x \mid \beta_{\text{red}}(x) = \ell\}$;
6 $\pi_{\text{green}}(\ell_{\text{red}}, \ell) := \text{card}\{x \mid \beta_{\text{green}}(\ell_{\text{red}})(x) = \ell\}$;

4 Conclusion

In this report, we have described a compilation-centric scheduling strategy. We propose a compilation algorithm to split the tasks in three parts, whose properties are intended to guide the runtime scheduling decisions. The compiler is available and may be tried online at `http://foobar.ens-lyon.fr/kut`. In particular, we show how the polyhedral model may provide a precious help to compute tricky scheduling (bottom-level) and parallelism (parallel volume) informations. The compiler has been applied successfully to 5 HPC kernels (Forward substituion, cholesky factorization, LU and LQ decomposition). The results may be obtained with the online demonstrator. The next step is to connect this compiler to the scheduler. In the future, we plan to refine the compilation scheme to preallocate the processing units and to limit the memory footprint.

Algorithm 2: Code generation

Input : The bottom-levels $\beta_{\text{red}}, \beta_{\text{green}}$ and the related parallelism level $\pi_{\text{red}}, \pi_{\text{green}}$
Output: The target C program with bottom-level predicates

```
1  Emit
2      function red_frontier()
3      begin
4          foreach ℓ ∈ range β_red, increasing do
5              if π_red(ℓ) ≥ #gpu then
6                  return ℓ;
7              end
8          end
9          Error("red level not found");
10     end
11
12     function green_frontier(ℓ_red)
13     begin
14         foreach ℓ ∈ range β_green(ℓ_red), increasing do
15             if π_green(ℓ_red, ℓ) ≥ #cpu + #gpu then
16                 return ℓ;
17             end
18         end
19         Error("green level not found");
20     end
21
22     function below_red_frontier(S, i⃗, ℓ_red)
23     begin
24         return β_red(S, i⃗) ≦ ℓ_red;
25     end
26
27     function below_green_frontier(S, i⃗, ℓ_red, ℓ_green)
28     begin
29         return β_green(ℓ_red)(S, i⃗) ≦ ℓ_green;
30     end
31
32     function main()
33     begin
34         ℓ_red := red_frontier();
35         ℓ_green := green_frontier(ℓ_red);
36         ...
37     end
38 end
```

Annex – `kut` input syntax

Our compiler, `kut`, is available and may be tried online at `http://foobar.ens-lyon.fr/kut`. `kut` inputs a C program with a single function annotated with pragmas:

```
int N;

void kernel_trisolv()
{
  int i, j;

  //INPUTS
  int** L;
  int* x;
  int* b;

#pragma begin_scop
#pragma resource[NCPUS][NGPUS]

  for (i = 0; i < N; i++)
    {
#pragma latency[COPY_CPU][COPY_GPU]
      x[i] = b[i];
      for (j = 0; j <i; j++)
#pragma latency[GEMM_CPU][GEMM_GPU]
        x[i] = x[i] - L[i][j] * x[j];
#pragma latency[TRSM_CPU][TRSM_GPU]
      x[i] = x[i] / L[i][i];
    }
#pragma end_scop
}
```

`kut` uses the following pragmas:

- pragmas `begin_scop` and `end_scop` delimits the portion to be processed.
- pragma `resource` specifies the number of CPU/GPU in the target platform
- For each statement, pragma `latency` specifies (a model of) the latency on CPU/GPU. A latency is always an integer, it does not depend on the input. At first glance, this may seem rough. Typically a statement is a BLAS kernel, whose latency only depends on the input size (matrix/vector dimension), not on input values. Since the size is always the same, this model makes sense.
- Additional pragmas allow to specify multiple outputs (which happens often with BLAS kernels). They are not detailed in this report.

References

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 101–113, 2008.

[3] Philippe Clauss. 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 1996, Philadelphia, PA, USA, May 25-28, 1996*, pages 278–285, 1996.

[4] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(02):73–81, 1991.

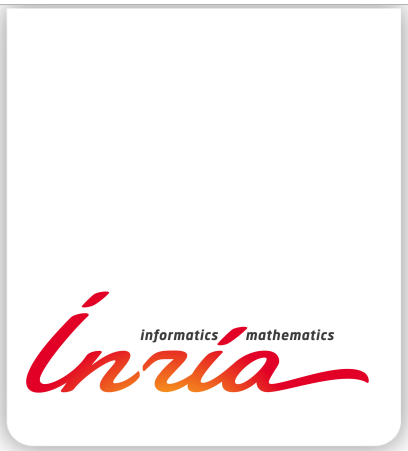
[5] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[6] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II: Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[7] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. 2011.

[8] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International journal of parallel programming*, 28(5):469–498, 2000.

Contents



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399