Automatic Code Generation for Iterative Multi-dimensional Stencil Computations

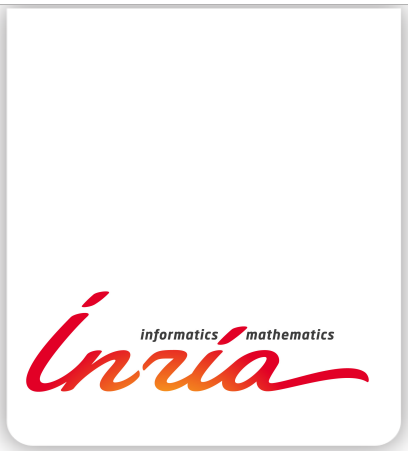
Mariem Saied, Jens Gustedt, Gilles Muller

Automatic Code Generation for Iterative Multi-dimensional Stencil Computations

Mariem Saied **Jens Gustedt** **Gilles Muller**

RESEARCH REPORT
N° 8928
June 2016
Project-Team Camus

ISSN 0249-6399 ISRN INRIA/RR--8928--FR+ENG



Automatic Code Generation for Iterative Multi-dimensional Stencil Computations

Mariem Saied*† Jens Gustedt*† Gilles Muller‡§

Project-Team Camus

Research Report n° 8928 — June 2016 — 23 pages

Abstract: We present a source-to-source auto-generating framework that enables a large programmer community to easily and safely implement parallel stencil codes within the framework of Ordered Read-Write Locks (ORWL). It meets the specific needs of the application at a high level of abstraction. ORWL is an inter-task synchronization model for iterative data-oriented parallel and distributed algorithms that uses strict FIFO ordering for the access to all resources. It guarantees equity, liveness and efficiency for a wide range of applications. The main hurdle for using ORWL lies in its initialization phase, where the programmer has to specify the access scheme between tasks and resources and the initial positions of the tasks in the FIFOs. We provide a user-friendly interface based on a Domain-Specific Language (DSL) that captures domain semantics and automatically generates ORWL parallel high-performance stencil code. We conducted experiments that proved the validity of our approach, as well as the efficiency and scalability of the generated code.

Key-words: stencil computations, ordered read-write locks, domain-specific language, experiments

* INRIA, Nancy – Grand Est, France
† ICube – CNRS, Université de Strasbourg, France
‡ INRIA – Paris, France
§ LIP6 – Sorbonne Universités, CNRS, UPMC, France

RESEARCH CENTRE NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Génération automatique de programmes pour des calcul multi-dimensionnels de type stencil

Résumé : Nous présentons un outil de génération automatique source-à source de code qui permet à une grande communauté de programmeurs d'implémenter des codes stencils dans le cadre des verous ordonnés de lecture-écriture (ORWL), et ceci de façon facile et sûre. L'outil proposé remplit les besoins spécifiques pour permettre un niveau d'abstraction élevé. ORWL est un modèle de synchronisation entre tâches pour des calculs itératifs qui sont centrés sur les données. Il utilise un ordonnocement strict imposé par des files d'attentes qui sont associées à toutes les resources. Ainsi, il garantit l'égalité, la vivacité et l'efficacité pour une large catégorie d'applications. L'obstacle majeur pour l'utilisation d'ORWL est sa phase d'initialisation, où le programmeur doit spécifier le schéma d'accès entre tâches et resources, ainsi que les positions initiales des tâches dans les files d'attente. Nous fournissons une interface conviviale, basée sur un langage dédié (DSL) qui capte la sémantique spécifique au domaine et génère du code stencil parallèle de haute performance. Nous avons effectué des expériences qui prouvent la validité de notre approche, ainsi que l'efficacité et l'extensibilité du code généré.

Mots-clés : calcul type stencil, verous ordonnés lecture-écriture, languages dédiés, expérimentations

1 Introduction

Stencil kernels appear in a wide range of scientific and engineering applications ranging from numerical solvers and PDE solvers (Partial Differential Equations) to computational physics [1,2,3], as well as image processing [4,5]. The computational pattern for stencils consists of iterative sweeps over a data grid, where each grid element is updated as a function of its neighbouring elements. Stencil computations are often parallelized through a decomposition of the data grid into several *blocks* or *tiles*, such that the computation of a specific block requires updated information from neighbouring blocks. Conceptually, this enforces the use of so-called shadow regions that surround each block with its updated neighborhood information. These are very difficult to handle since they often need complex case analysis for communication statements and index calculations. Usually, the burden of the creation and management of the shadow regions lays on the application programmer. Apart from that, the programmer is responsible for handling overlapped computations and communications over block resources. Therefore, he needs the support of a powerful inter-task synchronization mechanism such as ORWL.

ORWL, see [6], is an inter-task synchronization paradigm for iterative data-oriented parallel and distributed algorithms. It favors algorithmic control and data consistency by introducing a FIFO-based lock mechanism that handles data-dependencies between tasks and data resources. One of the originalities of ORWL is the proactive announcement of the resources that a task requires for a future computation. ORWL can be used in shared memory, distributed (network) or mixed contexts.

As stencil kernels are well structured and computation-intensive, they present very interesting applications of ORWL. Altough the modeling power of ORWL largely exceeds the stencils, in this work we concentrate on these.

However, currently the main hurdle for using ORWL lies in its demanding initialization phase where the programmer has to specify the access scheme between tasks and resources and the initial positions of the tasks in the FIFOs. The latter are decisive for the liveness of the application, and thus should be attributed with a lot of care. Once this tedious but necessary step is done correctly, all subsequent iterations are guaranteed to be deadlock-free and fair.

In order to enhance development productivity when writing ORWL applications, we aim to alleviate this burden and automate the initialization phase, including the attribution of initial handle positions in the FIFOs. Our approach is to add a layer, based on an implicitly parallel domain-specific language (DSL), [7] on top of ORWL.

On one hand, DSLs offer appropriate domain-specific notations, constructs and abstractions, improve the expressiveness and thus the productivity in the particular domain they are designed for, compared with general-purpose programming languages (GPL). On the other hand, DSLs achieve code efficiency thanks to the incorporated domain-specific knowledge. DSLs usually provide acceptable performance and can sometimes reach the performance levels that are attained by hand-coded implementations. Thereby, they allow to balance

programmability and performance.

Not surpisingly, in recent years, DSLs have been widely used in parallel programming to spare the user the details and the complexity related to parallel programming. In particular, numerous research efforts have adopted DSL solutions to optimize stencil computations. Some of them, suggest autotuning frameworks for multicore architectures [8] and GPU accelerators within DSLs [9,10]. Others suggest DSL-based stencil compiler transformations to generate efficient code for GPUs and multicore processors [11]. The Pochoir stencil compiler [12], for example, uses cache-oblivious parallelograms for parallel execution on shared-memory systems to produce high-perfomance code for stencils. With this work, we propose a DSL-based solution to make stencil implementations within the ORWL framework simpler and more user-friendly. By that, we bridge the gap between productivity and high performance. We aim to allow scientific domain experts or average programmers to develop ORWL stencil codes that meet their specific application needs without becoming experts in parallel programming in general nor in ORWL in particular. In this paper, we present Dido, a DSL for developing parallel stencil computations within ORWL. Dido is implicitely parallel, expressive, productive and highly-efficient. It meets the specific needs of the application at a high level of abstraction. It combines performance and correctess, without requiring much effort from the programmer. The goal of Dido is to bridge the gap between productivity and high performance, often considered as mutually exclusive.

Our main contributions are:

- We provide a user-friendly interface based on a Domain-Specific Language (DSL) that captures high level stencil abstractions and automatically generates ORWL parallel high-performance stencil code.
- We present a pattern for ORWL that we call ComUP, that is relevant for all ORWL implementations not only for stencil computations. It ensures expressiveness, deadlock-freeness and better performance for ORWL programs.
- We show that the DSL achieves a huge progress in terms of programmer productivity without sacrificing the performance. We expose the complexity of the generated code and the amount of complex details the DSL spares the user.
- We present experiments that prove the efficiency and scalability of the generated code that outperforms hand-crafted code.

The remainder of this paper is organized as follows. In Section 2, we present the Ordered Read-Write Locks model. An overview of the stencil computations we consider in this work is presented in Section 3. Section 5 provides a full specification of our DSL Section 5. In Section 6 and Section 7, we highlight respectively the programmer productivity and performance benefits achieved by our framework. We summarize our work and future directions in Section 8.

Listing 1 – A simple critical section operating on one resource through an `orwl_handle`.

```
/* announce the access                          */
orwl_write_request(&loc, &handle);
/* some operation without the resource          */
/* then, block until acces granted              */
orwl_acquire(&handle);
/* some critical operation with locked resource */
/* then, free the resource                      */
orwl_release(&handle);
```

2 ORWL Overview

ORWL (Ordered Read-Write Locks) is an inter-task synchronization paradigm for resource-oriented parallel and distributed algorithms. It provides synchronization methods that allow an implicit deadlock-free and equitable control of protected resources. Here, a resource can be an abstraction of data, hardware or software on which tasks interact. ORWL provides a way to control the execution order of tasks based on their data dependencies. It is based on the following features:

1. An access queue (FIFO) for each such lock.
2. An explicit association of a lock with each resource.
3. A distinction between locking for writing (exclusive) and locking for reading (inclusive).
4. A three-step lock operation by a sequence of *request* (queue insertion), *acquire* (blocking until first in queue) and then *release*.
5. A distinction between locks (opaque objects) and lock handles (user interfaces acting on the locks).

The logic behind ORWL is that a task anticipates and proactively announces the resources it is going to require for future computation. Through a lock handle, it *requests* a slot in the queue of the resource (see Listing 1). Only when the resource becomes necessary for the process to continue, the process attempts to *acquire* and is blocked until access to the resource is granted. Such a mechanism enables the run time system to anticipate the access, *e.g.* by doing a data prefetch, at a reduced cost. In particular, it allows to hide access latency that could be caused be slow communication links. A task is said to be active and can be executed only when all the locks it has requested have been acquired. An implementation of the ordered read-write lock paradigm using standard languages and interfaces (C and POSIX), has been presented in [13], together with its basic use patterns. It can be used in shared, distributed or mixed contexts.

Listing 2 – A critical section operating on one resource, accessed iteratively, through an `orwl_handle2`.

```
/* bind the pair of handles to the resource     */
orwl_write_request2 (&loc, &handle2);
/* some operation without the resource          */
/* then, block until acces granted              */
orwl_acquire2 (&handle2);
/* some critical operation with locked resource */
/* free resource + new request for next iteration */
orwl_release2 (&handle2);
```

Experiments have shown that ORWL is a powerful synchronization tool that guarantess liveness, equity and efficiency for a wide range of applications. It also presents a valid choice for out-of-core computation [14].

2.1 Iterative tasks

The pro-active locking enables a thread or a process to define several handles on the same lock, and thereby to newly request a lock by means of one handle while still actively holding a lock via another handle. Thereby, iterative tasks may insert their request for the next iteration in the FIFO while still holding a lock for the current one. This is a big advantage for iterative computations that access data in a cyclic pattern, and the library provides specific tools that implement this pro-active iteration scheme.

The type `orwl_handle2` presents a pair of `orwl_handle` that are bound to the same resource and used in alternation. At the start of each iteration, one of the two handles is bound to the resource to grant the access for the current iteration, while the other is inactive. At the end of the iteration, before releasing the lock, a new request for the next iteration is automatically posted through the inactive handle. This guarantees the reservation of the resource for the next iteration at the same initial FIFO ordering, before releasing the lock to grant access to other tasks operating on the same resource.

2.2 Initialization Phase

In this iterative setting, ORWL can guarantee the crucial properties of liveness and equity for all tasks, although this comes not without effort. It has been shown, see [6], that the initial FIFO positions of the locking requests are decisive. They must be attributed with a lot of care in order to avoid cyclic dependencies that may lead to a deadlock. In consequence, the initialization phase where the programmer specifies the initial access order of the tasks to the resources is tedious to implement and error prone. On the other hand, once this key step is done correctly, all subsequent iterations are guaranteed to be deadlock-free and fair. One of our main motivations is to alleviate this programming task and

to automate the initialization phase, including the attribution of initial FIFO positions.

2.3 A programming paradigm with a local view

An ORWL program is specified as a local description of a set of equivalent tasks that compose the application and that are to be executed concurrently. Each task has a set of resources it "owns" and it executes its specific program code. A task can refer to one of its own resources (local) or those of other tasks (remote). The tasks interact through communication and synchronization operations via the locks they subsequently hold for these resources. This includes all operations executed on the locally owned resources. The tasks are realized by threads (local execution) and processes (remote execution) and the operations on the resources are implemented in shared memory, when possible, or through network communications on the socket level.

3 Stencil Computations

Stencil computations constitute a widely used computational pattern that performs iterative global sweeps over a multidimensional regular grid, realizing nearest neighbor computations. At each iteration, each element is updated following a function of a subset of its neighboring elements from previous iterations. In this work, we consider stencil computations performing iterative point-wise updates over an n-dimensional grid, according to a function involving the following computation:

$$
\begin{aligned}
&R^{\varphi}[i_1][i_2]...[i_n] = \\
&\sum_{\nu=1}^{\mu} \left(\sum_{k} C_k[i_1][i_2]...[i_n] \times A^{\varphi-\nu}[i_1 \pm h^{\nu}_{k,1}][i_2 \pm h^{\nu}_{k,2}]...[i_n \pm h^{\nu}_{k,n}] \right. \\
&\left. + \sum_{m} \alpha_m \times A^{\varphi-\nu}[i_1 \pm \ell^{\nu}_{m,1}][i_2 \pm \ell^{\nu}_{m,2}]...[i_n \pm \ell^{\nu}_{m,n}] \right)
\end{aligned}
$$

Here, φ is the current iteration. The center element and its neighbouring elements from previous iterations $A^{\varphi-\nu}$, $\nu = 1, \ldots, \mu$, are weighted by coefficient grids C_k and scalar constants α_m. The new element A^{φ} is then deduced from intermediate value R^{φ} by any set of statements.

We refer to A as the *main data*, that is the grid that undergoes the computation. We presume that there is only one such main data, but that there can be multiple coefficient grids that we call *auxiliary data*. The auxiliary data are constrained to have the same topology and size as the main data. They must also be accessed at the same position $[i_1][i_2]...[i_n]$ as the center element. Unlike the coefficient grids, the main grid A can be accessed at any position. $h^{\nu}_{k,d}$ and $h^{\nu}_{l,d}$ present offsets separating the accessed neighboring elements from the central element. We call the maximum of these offsets the *halo* of the computation

$$halo = \max_{\nu,k,d,m} \{h^{\nu}_{k,d}, \ell^{\nu}_{m,d}\}.$$

It has to be noted however, that the formula above is valid only for inner computations of elements that are not on the boundaries of the main data. In Section 5, we evoke the different measures we have taken to handle boundary conditions.

3.1 Stencils within ORWL

An intuitive method to parallelize stencil computations is to decompose the main data into a set of blocks. While the computation in each block that is sufficiently far from the edges is independent from other blocks, the computation near the edges and corners depends on a subset of neighboring blocks, that are computed by different tasks. Thus, communication is needed to exchange updated values between blocks after each iteration. Thus, a specific mechanism must ensure the synchronization of overlapping computations and communications that access a protected resource. Here, our inter-task synchronization model ORWL could be very helpful. Stencil modeling, within ORWL, implies the definition of a number of data locations, where an ORWL is associated with each data location. One possible solution is to define several *shadow locations* in addition to the *main location* that consists of the block itself. They are buffers where, at the end of each iteration, the newly computed data of block edges is saved, representing a push scheme. The *shadow locations* are then available to neighbouring tasks for reading. The number of ORWl data locations depends on the neighbourhood type of the stencil computation and whether the neighborhood is composed of all the elements surrounding the central element, including diagonal elements, or composed of only the elements in direction of the axis. Figure 1 shows an example of a simple decomposition of a 2D matrix into four blocks: Each task consists of one compute and 8 update operations acting on 9 locations that are the *main location* and 8 *shadow locations*. The compute operation reads the data it requires through handles from the *shadow locations* of the remote neighboring blocks. Then, it executes the computation kernel and writes the results to the main location. The update operations, one for each *shadow location*, are used to export the data of the edges and corners to the neighboring tasks.

Each task has only one exclusive write access on one single location and each data location is only required by one exclusive write access and therefore one task only. This is called the canonical form that we use in order to guarantee the use of all ORWL proofs.

3.2 Motivations

As mentioned in Section 2, to take full advantage of ORWL properties, the programmer has to go through a tedious but necessary initialization phase. They have to carefully specify the access scheme between tasks and resources and to assign the initial positions of lock handles in the FIFOs of the resources. Mistakes at this level may lead to deadlocks. Our principal motivation for developing our DSL is to alleviate this burden by automating the generation of this initialization phase in a fail-safe manner. Furthermore, without the aid

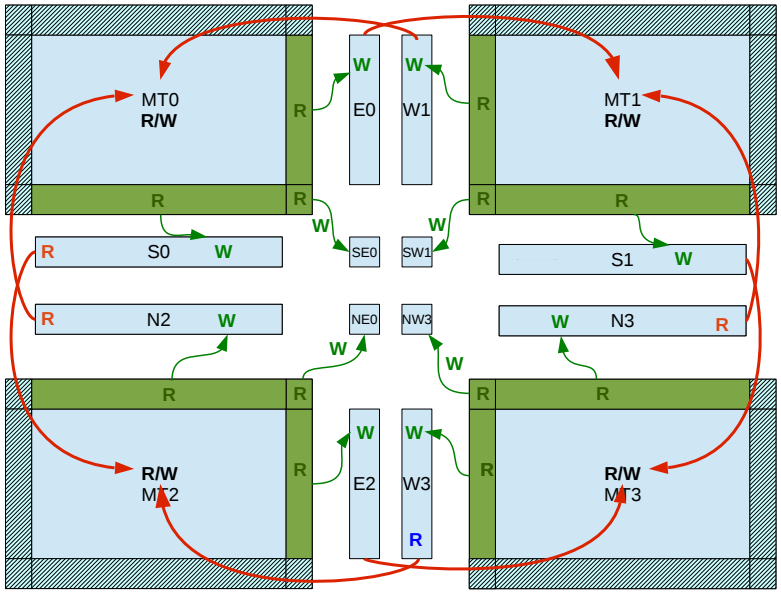


Figure 1 – An example of a 2D stencil ORWL modeling with a decomposition into four blocks

of the DSL, the programmer is responsible for handling the updates of the halo regions. They have to define the ORWL data locations explicitly, allocate their associated buffers and write the update operations. These programming tasks are by far longer and more involved than the computational kernel they serve. Additionally, it is very likely that the programmer makes mistakes when writing the indices in the complex update operations that have to be specified by using the halo margin offsets in each direction of the problem grid. The more dimensions the main data has, the more complex the indices become and the higher is the risk of making mistakes. With our DSL, we spare the programmer from writing complex indices and error-prone parts, by fully generating them. Apart from that, it is difficult for non-experienced developers to achieve good parallel performance results, be it by using ORWL or any other programming framework. Our DSL also allows us to incorporate our ORWL domain-specific knowledge in a code generation framework that helps the user achieve high performance as well as productivity benefits.

3.3 Benchmarks

In this paper, we choose the following stencil computations as bench- marks.

- a 2-dimensional 5-point *Jacobi*.
- a 3-dimensional 7-point *Jacobi*.

Listing 3 – Livermore Kernel 23.

```
for (i = 1; i < N-1; ++i) {
  for (j = 1; j < M-1; ++j ) {
    q = data[i-1][j] * zb[i][j]
      + data[i][j-1] * zv[i][j]
      + data[i][j+1] * zu[i][j]
      + data[i+1][j] * zr[i][j]
      + zz[i][j] - data [i][j];
    data [i][j] += 0.175 * q ;
  }
}
```

- *Game of life:* a 2D 9-point stencil.
- Livermore Kernel 23.

The *Livermore Kernel 23*, a classic benchmark taken from LINPACK, has always been considered as a refernce benchmark for ORWL. As shown in Listing 3, it is a 5-point Stencil. Each element is updated by the element at the same position from the previous iteration and 4 neighbors offset by 1 on each direction. This stencil algorithm is cache-unfriendly. It has the partcicularity that data grows quickly in memory, as it requires a set of five coefficient grids (zb, zv, zu, zr and zz) in addition to the main data grid, which makes the overall memory footprint bigger.

4 An ORWL pattern for stencil applications

The aim of this work is to automate the generation of multidimensional stencil code within the ORWL platform. To this end, deep knowledge of the domain semantics was needed. We had first to analyze ORWL stencil implementations and extract the contraints they must meet in order to define a generalized use pattern that fulfills the deadlock-freeness and liveness proprieties for ORWL stencils. The pattern was extracted from 2D and 3D implementations and generalized for multidimensional stencils.

4.1 The CompUp form

To enhance the expressiveness of our tool and make code generation easier, we agreed to express ORWL programs in a form that we named CompUp. Here, we cast an ORWL program in the form of three types of operations: *local update*, *global update* and *compute*.

1. The *compute* operation performs the computation. It reads the data that is imported by global update operations and saved on local buffers. Then, it executes the computation kernel and writes the results on the main location.

2. The *local update* operation ensures the data transfer between different resources of the same task. It reads the updated data from the main location and stores it in buffers to make it available for neighboring tasks.

3. The *global update* operation ensures the external communication between the current task and other tasks. It reads the data in the neighboring tasks' buffers, updated by their own local update operations, and write it on local buffers, making it available for the compute operation.

In order to meet the canonical form constraints, careful efforts have been made to associate one single operation to each location, and thus one exclusive write access to each location.

We add additional constraints on the initial positions that the previously enumerated operations take in the data location FIFOs. First, each operation should have the same initial position over all the resources it needs. Second, we impose that these initial positions of priorities follow the order above. Namely, the compute operation has priority over the others. Then, the local update comes second to save the computed results on local buffers. The last priority is assigned to the global update operation. It has been proven that by adding these constraints to the CompUp form, the subsequent computation is guranteed to be deadlock-free.

In addition to the liveness guarantees that it provides, the CompUp form enables the automatic attribution of the FIFO initial positions. It has to be mentioned that the CompUp form is valid for all ORWL implementations not only for stencil computations i.e graph processing. But, these are not the object of this paper.

4.2 Overlapped data partitioning

Instead of standard block partitioning, (cf. section 3), we extend the blocks to include the halo region elements. Each block is then enlarged by twice the halo offset on each dimension. The overlapped data partitioning greatly simplifies the compute operation code and more precisely the computation of the frontier elements of each block. On one hand, it spares the analysis and specific treatment details that would have been, otherwise, necessary for their computation. On the other hand, it helps avoid the complex indexing relative to the halo region updates. The computation part in the compute operation is then reduced to one treatment case to be applied on all the block elements, including the edges. This considerably simplifies the code, making it clearer and easier to understand, to write and thus to generate. Preliminary experiments has shown that the overlapped data partitioning has no negative impact on performance.

4.3 Example

In this part, we combine the CompUp form with the overlapped data partitioning in a 2D stencil example. Each task consists of one compute operation, 8 local update operations and 8 global update operations acting on 17 locations:

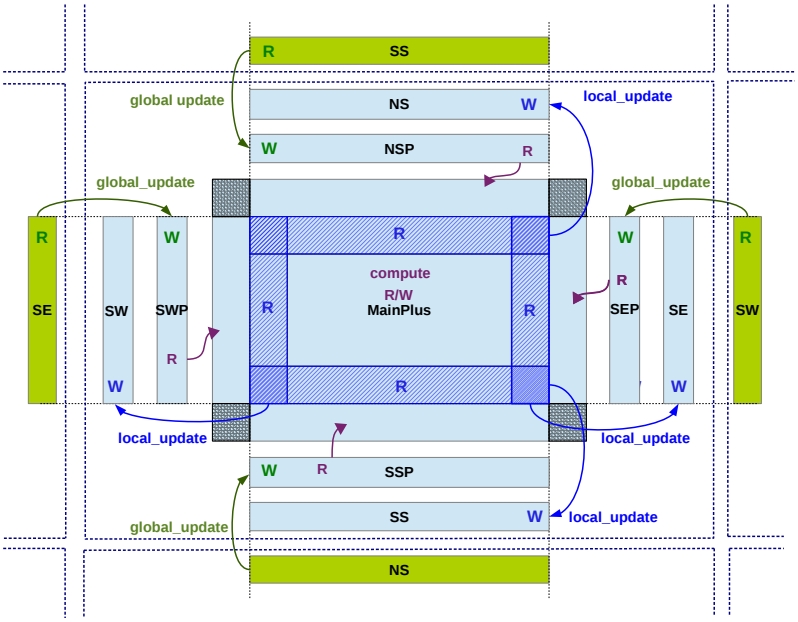


Figure 2 – Local Update - Global Update – Compute modeling in a 2D stencil example

- The *main location* as shown in Fig. 2 is extended and includes the halo region elements.
- 8 of the 16 *auxiliary locations (NS, SS, SE, SW)* present locations where the data of the current block edges is exported, saved and made available to neighbouring tasks for reading.
- The remaining 8 *auxiliary locations (NSP, SSP, SEP, SWP)* are needed to store updated data that is imported from neighbours.

As shown in Fig. 2, there are three types of operations:

- Compute: It has read/write access to the main location and read access to the *auxiliary* locations (NSP, SSP, SEP, SWP). At each iteration, it updates first the halo margin elements on the *main location*. Then, it executes the computation kernel.
- Local Update: copies updated data from *main location* and store it in local buffers. The updated data is then made available for neighboring tasks and ready to be read without interrupting the main block computation. It has read access to the main location and write access to the corresponding *auxiliary location*.
- Global Update: reads the data from the *auxiliary locations* of the neighbouring tasks and writes it on local buffers.

5 Dido: a DSL for ORWL stencils

Our DSL is intended to generate ORWL code following the pattern we have described above (cf. Section 4). Therefore, additional parameters have to be provided by the user, specifying the topology and size of their specific problem instance. Our framework enables the description of those parameters in a concise and intuitive syntax. The user provides specifications written in the introduced language. These are parsed in order to extract stencil features that are used, in a second step, to generate the corresponding ORWL code. The DSL enables a natural description of the parallel computation, where the compute kernel itself is specified in the form of a conventional (sequential) C function. The name of a header file that contains this function is specified within the DSL. This choice is made on purpose. It allows to handle all types of stencil computations, makes it possible to combine different types of statements that may be needed by the user, and allows to reuse legacy code, easily. Listing 4 shows a specification of a 2D stencil computation within the DSL. The main data (see Line 1) is specified by its name (here `za`), dimension (`2D`), and names for the problem axes (`x` and `y`). The *auxiliary data*, for their part, are specified by their names. According to a naming convention, these names will be used for the generation of the ORWL location and task identifiers. The convention has been chosen carefully in order to guarantee the readability and expressiveness of the generated code. The user has to provide a header file (here `"type.h"`) that contains all types, data structures and scalar contstants that are necessary for the application on the level of the C code (see Line 9). This header file will then be included in the generated C code of the ORWL program. They also have to specify the data type of the main data elements (see Line 14). It can be either a basic type (`float`, `double`, `char`, etc.) or an aggregate data type that has to be defined in the specified type header file. Additionally, to be able to handle high order stencils, the *halo* of the computation has to be specified (see Line 10), as well as the number of previous iterations involved in the computation (see Line 11). The latter is needed, because in the computate kernel the user may reuse data elements from previous iterations. There is also one crucial element to be specified, which is whether the computation involves other elements or not. Furthermore, to match the needs of real applications, we provide different ways to specify the boundary conditions. They can either be a constant value for borders (as in the example, Line 12) or define a specific function to compute element values on the edges.

In order to save compile time and efforts, the DSL is divided into two parts: The first part encompasses structural parameters of the application. Those parameters define the problem and enable the generation of the most complex parts of the ORWL code. This includes the data locations, the operations, the handle initializations and the initial positions in the FIFOs and the critical sections. The compilation time of the ORWL code generated by this part of the DSL could be moderately long. The second part consists of the execution parameters for a particular problem instance. Here, the user should specify a header file (here `"init.h"`) that contains initialization functions for the main and

Listing 4 – ORWL 2D Stencil specification within the suggested DSL.

```
1   Main_Data = {
2     za (2D) in (x,y);
3   }
4   Aux_Data = {
5     zr; zb; zu; zv; zz;
6   }
7   Application = {
8     kernel = "kernel.h";
9     types = "types.h";
10    halo = 1;
11    iteration_halo = 1;
12    border_value = 2;
13    neighbourhood_type = no corners;
14    element_data_type = float;
15  }
16  Execution_Parameters = {
17    iterations_number = 100;
18    data [1000][1000] into [100][100];
19    number_nodes = 25;
20    number_tasks_per_node = 4;
21    init_file = "init.h";
22   }
```

auxiliary data. A stopping criterion should also be provided. It can be either an iteration number (cf. Line 17) or a convergence criterion. They also have to specify the global size of the main data as well as the block sizes (see Line 18) into which it is partitioned. Finally, they have to specify the number of nodes on which the computation is deployed as well as the number of tasks per node (see Line 19-20). The code resulting from the *structural* part of the DSL can be used for different problem instances. The user has the option to generate all the code from scratch or just the main function depending on the execution parameters specified in the *instantial* part. Prior to code generation, the framework verifies that all the specified parameters within the DSL are coherent. If not, an error message is displayed. *E.g* in the example we see that the problem is divided into $10 \times 10 = 100$ blocks, and that this number corresponds to the total number of tasks, $25 \times 4 = 100$.

Structure of Generated Code

Given a stencil specification that does not exceed a few lines of trivial code, the DSL generates handreds of lines of ORWL parallel code. It consists of C code with includes of both P99[1].and ORWL libraries. Apart from being correct and error-free, the generated code is well indented and readable thanks to the well-chosen naming conventions that are used to name the different tasks and locations.

[1] `https://www.p99.gforge.inria.fr/`

Listing 5 – Main function of ORWL genrated parallel code.

```
int main(int argc, char **argv) {
  orwl_init();
  for (size_t i = 0; i < orwl_ll; i++) {
    task_thread* task = P99_NEW(task_thread);
    *task = (task_thread) {
      .n = n,
      .iterations = iterations,
      .global_y = global_y,
    };
    task_thread_operation(task, orwl_locids[i]);
  }
  return EXIT_SUCCESS;
}
```

Locations and tasks

The number of data locations and neighbouring tasks depends on the dimension of the main data and whether it has corners. If the computation includes elements on corners, then the number of tasks is $\sum_{k=1}^{n} 2^k * C_n^{n-k}$. Otherwise, the number of tasks is reduced to $2 * n$. Following the CompUp form, the number of the *auxiliary locations* is twice the number of tasks, in addition to the *main location*.

An ORWL program: a loop over locations

As an immediate result of the canonical form, the main function of an ORWL program is nothing but a loop over all ORWL locations as shown in Listing 5. For each location, an operation is automatically generated. That operation is then instantiated at run time by a separate thread. The type and structure of the operation, whether it is a global update, local update or compute, depends on the identifier of the associated location as shown in Listing 6.

Operations

But, all operations have common features. As shown in Listing 7, each operation consists of several steps:

- Lock objects and handles are initialized.
- Buffer sizes corresponding to locations are initialized.
- Initial requests to the specified locations are inserted for each handle.
- An introductory iteration, that distributes all control and data to the appropriate operations is executed.
- The proper computations (or data copies) are run in an iteration loop.

Listing 6 – Operation thread instantiation based on associated location ID.

```
DEFINE_THREAD(task_thread) {
  size_t myloc = orwl_myloc;
  orwl_server *const srv = orwl_server_get();
  orwl_locations task1 = ORWL_LOCAL(myloc);
  if (task1 == MAIN) {
      compute_task(Arg, srv, myloc);
  } else {
    if (task1 <= LOCATION_x_n) {
      update_local_task (Arg, srv, myloc, task1);
    } else  {
      update_global_task (Arg, srv, myloc, task1);
    }
  }
}
```

Table 1 – Number of hand-written lines of code vs. number of generated lines of code

metric	Livermore	Game of Life	3D Jacobi
hand-written lines	68	86	34
generated lines	711	988	942

These steps are synchronized by an ORWL barrier and a scheduling step. The latter ensures that the initial FIFO positions of all handles of all operations are inserted consistently at all locations. Thereafter, the execution order of tasks based on their data dependencies is fixed globally, and the operations can run concurrently without the need of a further global synchronization.

6 Programmer Productivity

The DSL was designed to improve the programmer productivity by sparing them from writing the complex parts of ORWL parallel code. Given a stencil specification that does not exceed a few lines of straightforward code, bare of any details needed for parallelization, hundreds of lines of code are generated. The generated code is guaranteed to be correct and error-free. Table 1 depicts the number of hand-written lines compared to the number of generated code lines for the considered benchmarks. The number of lines to be written by the user, in the case of 3D Jacobi for example, is reduced by 96%. We estimate this to be a considerable improvement in terms of programmer productivity. This measurement of a gain in productivity uses a comparison to explicit hand coding with ORWL. We think that the result would not be much different if we would compare to coding with other paradigms for parallel computation, as long as these involve explicit creation of buffer space, data communications or access locks.

Listing 7 – Structure of an ORWL Local Update operation

```
/*           data size buffers                */
/*         Lock Initialization Step           */
orwl_handle2 here = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 there = ORWL_HANDLE2_INITIALIZER;
orwl_global_barrier_wait(myloc, 1, srv);
size_t gid = ORWL_LOCATION(ORWL_TASK(myloc), MAIN);
orwl_write_insert(&here, myloc, 1, seed);
orwl_read_insert(&there, gid, 1, seed);
orwl_schedule(myloc, 1, srv);
/*         Initialization iteration           */
ORWL_SECTION(&here, 1, seed) {
  orwl_truncate(&here, sizeof(fline));
  ORWL_SECTION(&there, 1, seed) {
    line *const *matrixP = orwl_read_map(&there);
    fline *frontier = orwl_write_map(&here);
    ...
  }
}
/*         Computation iterations              */
for (size_t iter = 0 ; iter <iterations; iter++){
  ORWL_SECTION(&here, 1, seed){
    ORWL_SECTION(&there, 1, seed){
      line *const* matrixP = orwl_read_map(&there);
      fline *frontier = orwl_write_map(&here);
      ...
    }
  }
}
orwl_disconnect(&there, 1, seed);
orwl_disconnect(&here, 1, seed);
```

Not only does the DSL reduces the number of hand-coded lines, but it also avoids to manually write complex and error-prone parts such as the halo region update operations. A lot of time and effort is saved by automatically generating the indexing expressions and communication statements. Table 2 list the number of complex programming details such as communication statements and indexing operations that are automatically generated by the DSL. For example, if the user had to write the ORWL parallel code of a 3D Jacobi without the aid of the DSL, they would have to handle 13 data locations. To do so, they would have to initialize 11 handles and write 43 communication statements and 166 indexing operations.

Seeing all these properties, the total development time of a parallel stencil computation within our framework can be just a few minutes. We estimate the generated code to be clear and expressive. This is due, on one hand, to the carefully chosen naming conventions for tasks and locations. On the other hand, the overlapping data partitioning greatly simplifies the compute operation and reduces the number of the complex indexing expressions relative to the halo region updates. As a result, the generated code is comprehensive and could

Table 2 – Parallel programming details automatically generated and spared for the user

metric	Livermore	Game of Life	3D Jacobi
neighbouring tasks	4	8	6
locations	9	17	13
handle initializations	9	13	11
communication statements	29	48	43
indexing expressions	124	167	166

easily be modified, if ever some adjustments or optimizations are needed. For all those reasons, it seems clear that the DSL achieves a considerable improvement in terms of programmer productivity.

7 Perfomance Evaluation

We have measured the efficiency of our approach experimentally. In order to validate our modeling, we investigate the overhead, if any, that it might add to the computation. The experiments have been conducted on the graphene cluster of the Grid'5000 experimental testbed[2]. Each node is composed of 4 cores at 2.53 GHz, 16 GiB of memory, and a Gigabit Ethernet interconnection network. All the following results are obtained after running 100 iterations of Livermore Kernel 23. For greater accuracy, we take an average over several runs.

7.1 Average computation time per matrix element

The aim of this experiment is to study the variation of the average execution time per data element depending on the blocks' number and size. For the case of Livermore Kernel 23, we consider problems of 4, 16, 36, 64 and 100 blocks, where one compute node is reserved for each block. For each configuration, we increase the global problem size, until we reach the maximum size per block that fits into the RAM of the target machines. As shown in Figure 3, the computation time per data element decreases when the problem size increases. It tends to a lower limit. The times for different block partitions are almost indistinguishable, especially for large block sizes. This proves that the overhead for subdividing into more blocks is negligible.

7.2 Computation efficiency

The compute operation may spend some time waiting for frontier data. In this experiment, we relate the time spent with computation to the time spent with waiting for some frontier data. Otherwise, the setup is the same as in the previous experiment. Figure 4 shows that for small problems, almost all the time

[2] `https://www.grid5000.fr/`

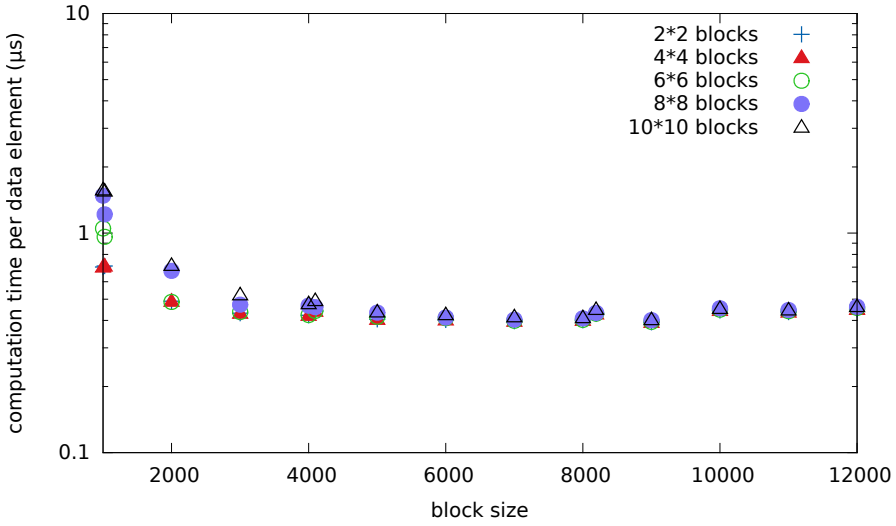


Figure 3 – Average computation time for a data element

is spent waiting. However, for larger problems, the time spent on computation increases and finally reaches about 99% of the total execution time.

7.3 Comparison with handcrafted code

In [13], an implementation of the Livermore Kernel 23, following the modeling presented in Subsection 3.1, was used in order to evaluate the performance of the model. The results have shown that the time spent on computation reaches only 55% of the total execution time. This is because, with that modeling, the computation on one block excludes computation on all of its neighbors. As a consequence, the computation converges to a steady state where it alternates between odd and even numbered anti-diagonals of the block-matrix. Therefore, overcomittment by several tasks per core was used in order to engage all cores in the computation and to improve execution times. In our modeling, there is no specific need for overcommitment, since the computation time now reaches 99% of the total execution time. This is due to the pattern, we have defined in Section 4. It ensures for tasks a certain independence from their neighbours. In fact, as soon as the shadow regions are updated in the main data location, the buffers are released and ready to receive next data updates from neighbours without interfering with the computation. As soon as the compute operation needs updated values, those are already available and ready to be read. As a result, compute operations of neighboring tasks can be executed simultaneously, which considerabely improves the execution times. The generated version of Livermore Kernel 23 outperfoms the handcoded code used in [13] with an average speedup of 11%.

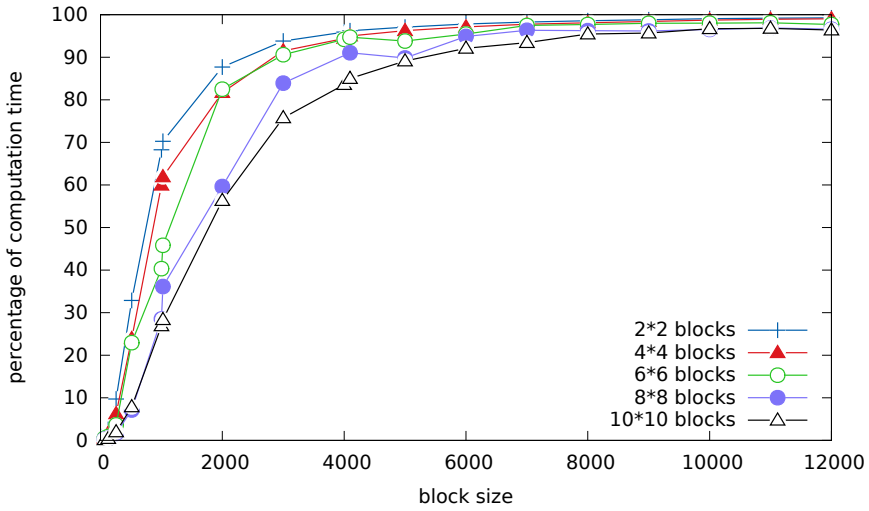


Figure 4 – Percentage of computation time

7.4 Size up

In the following experiments, we study the scalability of the generated code, by increasing the global problem size, for different stencil structures and neighbourhood dependencies,

7.4.1 2D Jacobi and Game of life

We conducted the same experiment for both 2D Jacobi and Game of life benchmarks. We place 2 tasks per node on 8, 16, 32, 48 and 64 quad-core nodes. Each task computes a 10000x10000 element block. Results are shown in Figure 5.

7.4.2 3D Jacobi

This last experiment has been conducted on the grisou cluster of the Grid'5000 experimental testbed. Each node is composed of two CPUs, each having 8 cores at 2.4 GHz, 126 GiB of memory and a Gigabit Ethernet interconnection network. We place 2 tasks per node on 8, 16, 24, 32, 40 and 48 nodes. Each task computes a 1000^3 block. Results are shown in Figure 6.

We can notice from previous experiments that increasing the over-all problem size does not increase the average computation time per iteration, if we provide a proportional amount of compute nodes. Thus, the ORWL stencil modeling we have presented in Section 4 allows us to construct sizable applications. The DSL does not have any negative impact on the sizability of the generated ORWL stencil code.

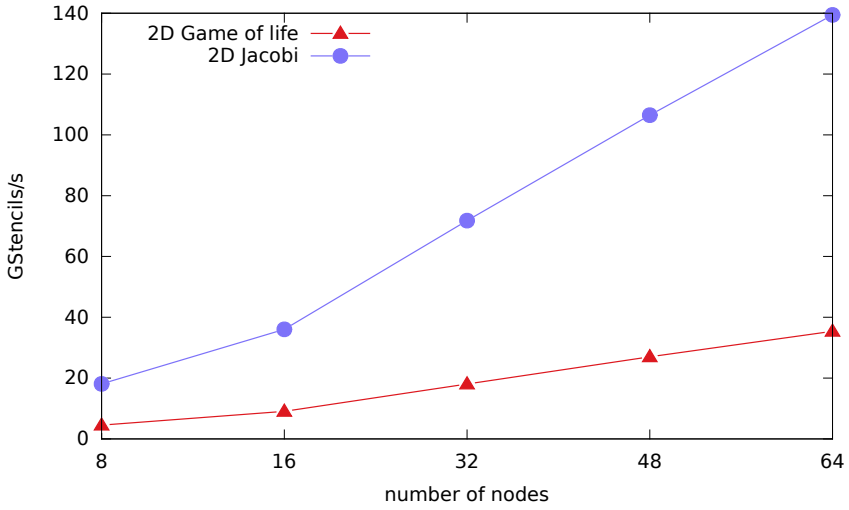


Figure 5 – Overcommitment and problem size up

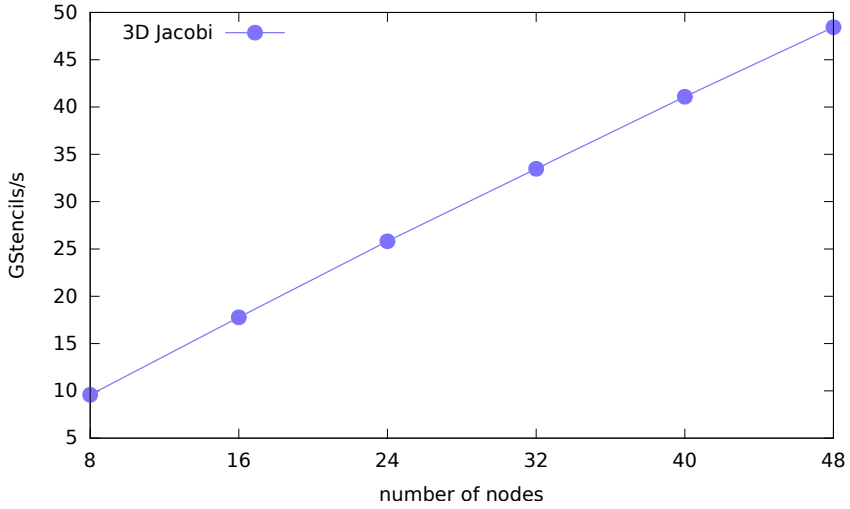


Figure 6 – Problem size up: 3D Jacobi

8 Conclusion & Future Work

We have presented Dido, an implicitly parallel DSL for ORWL stencil code generation that achieves both productivity and performance benefits. It allows a large programmer community to implement parallel multidimensional stencil computations and take advantage, at a low cost, of properties of the ORWL paradigm. It meets the needs of real applications by suppporting multiple data types and boundary conditions. The DSL captures high-level stencil abstractions written by the user in a simple syntax and generates all the complex parts such as the shadow region updates and the lock handle positions in the FIFOs. This presents a considerable improvement in terms of programmer productivity. Additionally, experiments have shown that productivity and performance, often considered as antagonistic, can be reconciled when using Dido. The generated code achieves competitive performance that is comparable with hand-crafted code. This is due to the domain-specific knowledge about ORWL that went into the defined patterns. It has to be noted that the CompUp form that we have presented can be used for different ORWL applications not only for stencil computations. It ensures expressiveness, deadlock-freeness and better performance of ORWL programs. In the future, we plan to take the execution architecture into consideration such that architecture-adapted code can be generated. ORWL runtime can then ensure a better placement of the tasks, close to the resources. Later, we aim to enhance our first-cut design in order to cover a wider range of applications. One promising direction are sparse matrices and applications. This type of algorithms is in widespread use by scientific communities and lacks user-friendly specific tools.

References

[1] A. Taflove and K. R. Umashankar, "The finite-difference time-domain method for numerical modeling of electromagnetic wave interactions," *Electromagnetics*, vol. 10, no. 1-2, pp. 105–126, 1990.

[2] R. Bleck, C. Rooth, D. Hu, and L. T. Smith, "Salinity-driven thermocline transients in a wind-and thermohaline-forced isopycnic coordinate model of the north atlantic," *Journal of Physical Oceanography*, vol. 22, no. 12, pp. 1486–1505, 1992.

[3] A. Nakano, R. K. Kalia, and P. Vashishta, "Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers," *Computer Physics Communications*, vol. 83, no. 2-3, pp. 197–214, 1994.

[4] J. Cong, M. Huang, and Y. Zou, "Accelerating fluid registration algorithm on multi-FPGA platforms," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on.* IEEE, 2011, pp. 50–57.

[5] J. Cong and Y. Zou, "Lithographic aerial image simulation with FPGA-based hardware acceleration," in *Proceedings of the 16th international*

ACM/SIGDA symposium on Field programmable gate arrays. ACM, 2008, pp. 67–76.

[6] P.-N. Clauss and J. Gustedt, "Iterative computations with ordered read-write locks," *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 496–504, 2010. [Online]. Available: http://hal.inria.fr/inria-00330024/en

[7] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao, "Manticore: A heterogeneous parallel language," in *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. ACM, 2007, pp. 37–44.

[8] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 4.

[9] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3d stencil codes on gpu clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 155–164.

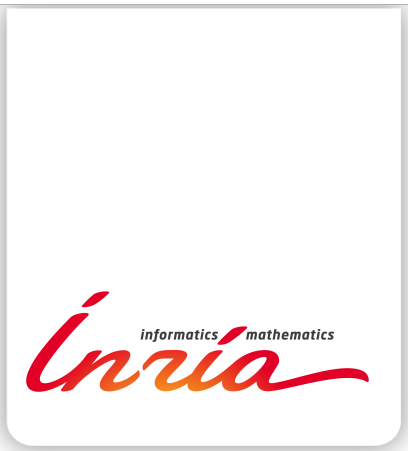
[10] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 311–320.

[11] T. Henretty, J. Holewinski, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "A domain-specific language and compiler for stencil computations on short-vector simd and gpu architectures."

[12] Y. Tang, R. Chowdhury, C.-K. Luk, and C. E. Leiserson, "Coding stencil computations using the pochoir stencil-specification language," in *Poster session presented at the 3rd USENIX Workshop on Hot Topics in Parallelism*, 2011.

[13] J. Gustedt and E. Jeanvoine, "Relaxed synchronization with ordered read-write locks," in *Euro-Par 2011: Parallel Processing Workshops*, ser. LNCS, M. Alexander *et al.*, Eds., vol. 7155. Bordeaux, France: Springer, Aug. 2011, pp. 387–397. [Online]. Available: https://hal.inria.fr/hal-00639289

[14] P.-N. Clauss and J. Gustedt, "Experimenting iterative computations with ordered read-write locks," in *18th Euromicro International Conference on Parallel, Distributed and network-based Processing*, M. Danelutto, T. Gross, and J. Bourgeois, Eds. Pisa, Italy: IEEE, 2010, pp. 155–162. [Online]. Available: http://hal.inria.fr/inria-00436417/en



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

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

ISSN 0249-6399