Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning

ALESSANDRO CILARDO and LUCA GALLO, Department of Electrical Engineering and Information Technologies, University of Naples Federico II

Emerging architectures, such as reconfigurable hardware platforms, provide the unprecedented opportunity of customizing the memory infrastructure based on application access patterns. This work addresses the problem of automated memory partitioning for such architectures, taking into account potentially parallel data accesses to physically independent banks. Targeted at affine static control parts (SCoPs), the technique relies on the Z-polyhedral model for program analysis and adopts a partitioning scheme based on integer lattices. The approach enables the definition of a solution space including previous works as particular cases. The problem of minimizing the total amount of memory required across the partitioned banks, referred to as storage minimization throughout the article, is tackled by an optimal approach yielding asymptotically zero memory waste or, as an alternative, an efficient approach ensuring arbitrarily small waste. The article also presents a prototype toolchain and a detailed step-by-step case study demonstrating the impact of the proposed technique along with extensive comparisons with alternative approaches in the literature.

Categories and Subject Descriptors: B.5.2 [Register-Transfer-Level Implementation]: Design Aids—*Automatic Synthesis*; D.1.3 [Programming Techniques]: Concurrent Programming—*Parallel Programming*; D.3.4 [Programming Languages]: Processors—*Code Generation*; D.3.4 [Programming Languages]: Processors—*Compilers*

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Memory partitioning, polyhedral model, fine-grained distributed shared memory, field-programmable gate arrays

ACM Reference Format:

Alessandro Cilardo and Luca Gallo. 2015. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Trans. Architect. Code Optim.* 11, 4, Article 45 (January 2015), 25 pages. DOI: <http://dx.doi.org/10.1145/2675359>

1. INTRODUCTION

During the past few years, a particularly important trend has clearly emerged in parallel computing architectures, indicating that significant improvements in performance can only be achieved by customizing the computing platform to some extent. In particular, the term *heterogeneous computing* refers to systems made of a variety of different general- and special-purpose computational units, such as graphics processing units (GPUs) [Brodtkorb et al. 2013], digital signal co-processors [Li et al. 2012a], and custom accelerators typically implemented on field-programmable gate arrays (FPGAs) [Cilardo et al. 2013]. Many of such advanced computing platforms are provided with

Authors' address: A. Cilardo and L. Gallo, Department of Electrical Engineering and Information Technologies, University of Naples Federico II, via Claudio 21, 80125, Napoli, Italy; emails: {acilardo, luca.gallo}@unina.it.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1544-3566/2015/01-ART45 \$15.00

DOI: <http://dx.doi.org/10.1145/2675359>

several independent memory banks that can be accessed simultaneously by parallel computing elements through complex interconnects. This potentially provides an opportunity for improving the memory bandwidth available to the application. However, to take full advantage of the memory architecture, adopting suitable memory partitioning strategies based on the actual application access patterns is of paramount importance.

This work proposes a methodology for automated memory partitioning in architectures provided with multiple independent memory banks accessed by parallel computing units. The approach encompasses both the problem of bank mapping and the minimization of the total amount of memory required across the partitioned banks, referred to as storage minimization here. Most of the results presented apply to affine static control parts (SCoPs)—that is, code segments in performance-critical loops where loop bounds, conditionals, and subscripts are affine functions of the surrounding loop iterators and of constant parameters possibly unknown at compile time. The methodology is based on the Z-polyhedral mathematical framework [Gupta and Rajopadhye 2007], allowing a compact and comprehensive representation of the code and the related transformations. Furthermore, it adopts a partitioning scheme based on integer lattices, identifying memory banks with different full-rank lattices that constitute a partition of the whole memory. We rely on a method for enumerating the solution space exhaustively and evaluating each solution based on the time overhead caused by conflicting accesses. We also propose a technique for representing the transformed program, suitable for existing tools that generate code from Z-polyhedra. For the storage minimization problem, we rely on an optimal approach yielding asymptotically zero memory waste or, as an alternative, an efficient approach ensuring arbitrarily small waste. The preceding techniques are demonstrated through a prototype toolchain relying on a range of software libraries for polyhedral analysis, whereas the experimental data are collected through high-level synthesis (HLS) of FPGA-based hardware accelerators. Adopting a lattice-based approach to memory partitioning in the context of HLS, the proposed technique improves on a few very recent results in the technical literature that formalize the same problem by means of less powerful mathematical tools, resulting in narrower solution spaces and thus missing potential solutions. In the last part of the article, we present extensive comparisons with state-of-the-art proposals concerned with memory partitioning in the context of HLS, showing that the lattice-based approach can effectively identify a superset of the solutions spanned by existing works.

The text is structured as follows. Section 2 presents a motivational example to introduce the problem. Section 3 recapitulates the relevant results on integer lattices and the Z-polyhedral model. Section 4 describes the proposed lattice-based partitioning technique. Section 5 reviews the overall approach by discussing a step-by-step case study. Section 6 presents some comparisons with existing literature works. Section 7 concludes the article with some final remarks on possible developments of this work.

2. MOTIVATIONAL EXAMPLE

Our motivational example is a downsizing algorithm used for image resizing relying on bilinear interpolation [Danahy et al. 2007], where the image is shrunk by a factor of 2 in both dimensions. For the sake of simplicity, we neglect boundary conditions and the source image is supposed to be grayscale. Each pixel in the target image is determined by averaging a 2×2 block of the source image. The kernel consists of a perfect loop nest and comprises only one statement. There are neither loop-carried dependencies nor loop-independent dependencies [Kennedy and Allen 2002]. The code is thus fully parallel. According to the bilinear interpolation method, each instance of the statement accesses simultaneously four locations of the source image. Figure 1(a) shows the

Fig. 1. Motivation example. (a) Kernel code. (b) Lattice-based partitioning. Each dotted box embraces the locations accessed by a specific iteration (i.e., a specific value of i and j), whereas the numbers associated to each location indicate the memory bank where the location is mapped. (c) Hardware datapath inferred from the memory partitioning solution of part (b).

kernel code, whereas Figure 1(b) highlights the access patterns of a few iterations, where each dotted box corresponds to the locations accessed by the same iteration (i.e., the same value of i and j). Clearly, if the number of memory ports equalled the number of pixels in the source image, all read operations could virtually be performed in parallel.¹ In contrast, if we have limited memory banks and the array elements are not properly mapped to the available banks, memory conflicts may arise and the inherent parallelism is not fully exploited. A few examples of partitioning approaches include cyclic and a block partitioning strategies [Cong et al. 2011]. Both first linearize the array. Then, cyclic partitioning assigns a memory location m to memory bank $m \bmod NB$, where NB is the number of banks, whereas block partitioning maps m to memory bank $\lfloor m/NB \rfloor$. Assume, for example, that we have only four memory banks available. In fact, with neither of the two approaches, four ports are sufficient to completely avoid conflicts. For instance, if we have a 480×640 image and adopt a cyclic partitioning strategy, iteration $i = 0, j = 0$ would simultaneously access both $A[0 \cdot 640 + 1]$ and $A[1 \cdot 640 + 1]$ in the flattened array and a conflict would arise since $1 \bmod 4 = 641 \bmod 4$. In essence, the technique proposed in this work seeks different approaches for partitioning the memory space such that improved parallelism in memory accesses can be achieved. Figure 1(b) shows the mapping determined by the lattice-based partitioning technique, which is introduced later in the article. As highlighted by the figure, the memory accesses in the statement can be completely executed in parallel. Figure 1(c) contains the hardware datapath implementing the example kernel based on our partitioning strategy. Access parallelization can be achieved with the lowest overhead in terms of steering and control circuitry. In fact, the mapping scheme follows a regular pattern, and consequently, a direct connection between the memory banks and the functional units is sufficient to access the required data across all iterations.

3. MATHEMATICAL BACKGROUND AND PROBLEM FORMULATION

This section briefly reviews a few mathematical concepts and results that are essential for the formulation of our lattice-based partitioning technique.

Given n linearly independent vectors $\vec{b}_0 = (b_{0,0}, \dots, b_{0,m-1}), \dots, \vec{b}_{n-1} = (b_{n-1,0}, \dots, b_{n-1,m-1}) \in \mathbb{R}^m$, a *lattice* $\mathcal{L}$ is a subset of $\mathbb{R}^m$ defined by $\{\sum_{j=0}^{n-1} z_j \cdot \vec{b}_j | z_j \in \mathbb{Z}\}$. We refer to $\vec{b}_0, \dots, \vec{b}_{n-1}$ as a *basis* of the lattice [Kelner 2009]. The rank of the lattice is n , whereas its dimensionality is m . If $n = m$, the lattice is a full-rank lattice. In

¹The same would apply to write operations, although we ignore them in this example for the sake of simplicity.

particular, if the basis is made of integer points, the lattice is an *integer lattice*. This article will only deal with integer lattices.

A basis can be also regarded as an $m \times n$ matrix $\mathbf{B} = \{b_{ij}\}$ having the vectors $\vec{b}_j$ as columns. We will say that $\mathbf{B}$ spans a lattice by using the notation $\mathcal{L}(\mathbf{B})$. Two different bases $\mathbf{B} = \{b_{ij}\}$, $\mathbf{C} = \{c_{ij}\}$ span the same lattice $\mathcal{L}$ if and only if there exists a unimodular matrix $\mathbf{U}$ (i.e., a square matrix with integer entries and determinant equal to ± 1) such that $\mathbf{B} = \mathbf{C} \cdot \mathbf{U}$.

Given a basis $\mathbf{B}$, a *fundamental parallelepiped* associated with $\mathbf{B}$ is a set of points in $\mathbb{R}^m$ defined by $FP(\mathbf{B}) = \{\sum_{j=0}^{n-1} r_j \vec{b}_j : 0 \leq r_j < 1\}$. The set $FP(\mathbf{B})$ is clearly half-open and the points in its translates $FP(\mathbf{B}) + \vec{t}$, with $\vec{t} \in \mathcal{L}(\mathbf{B})$, form a partition of $\mathbb{R}^n$. The *determinant* of a lattice, denoted $\det(\mathcal{L}(\mathbf{B}))$, is the n -dimensional volume of its fundamental parallelepiped and may be regarded as the number of integer points contained in it. For a full-rank lattice, $\det(\mathcal{L}(\mathbf{B})) = |\det(\mathbf{B})|$.

An integer *affine* lattice $\mathcal{L}_t$ is obtained by translating an integer lattice $\mathcal{L}(\mathbf{B})$ by a constant offset $\vec{t} = (t_0, \dots, t_{m-1}) \in \mathbb{Z}^m$ —that is, $\mathcal{L}_t = \{\mathbf{B} \cdot \vec{z} + \vec{t} \mid \vec{z} \in \mathbb{Z}^n\}$.

An *integer polyhedron* P is a subset of vectors in $\mathbb{Z}^m$ satisfying a finite number of affine (in)equalities with integer coefficients: $P = \{\vec{z} \in \mathbb{Z}^m \mid \mathbf{Q} \cdot \vec{z} + \vec{q} \geq \vec{0}\}$, where matrix $\mathbf{Q}$ and vector $\vec{q}$ specify the (in)equalities. An integer *polytope* is a bounded integer polyhedron.

By combining multiple affine constraints on integer variables, either equalities or inequalities, by means of logical operators ($\neg$, $\wedge$, and $\vee$) and quantifiers ($\forall$ and $\exists$), we can specify special sets, called *Presburger sets*, which are particularly relevant to our work. Consider an integer polyhedron, $P = \{\vec{z} \in \mathbb{Z}^m \mid \mathbf{Q} \cdot \vec{z} + \vec{q} \geq \vec{0}\}$, and an affine function, $F(\vec{z}) = \mathbf{F} \cdot \vec{z} + \vec{f}$, where $\mathbf{F}$ is an $n \times m$ matrix. The image of P under F is in the form $F(P) = \{\mathbf{F} \cdot \vec{z} + \vec{f} \mid \mathbf{Q} \cdot \vec{z} + \vec{q} \geq \vec{0}, \vec{z} \in \mathbb{Z}^m\}$. These structures are linearly bound lattices (LBLs) [Teich and Thiele 1993] and can in fact be expressed as Presburger sets.

A *Z-polyhedron* Z is the intersection of a polyhedron $P^c = \{\vec{z} \in \mathbb{Z}^m \mid \mathbf{Q} \cdot \vec{z} + \vec{q} \geq \vec{0}\}$ and an integer full-dimensional lattice $\mathcal{L} = \{\mathbf{B} \cdot \vec{z}, \vec{z} \in \mathbb{Z}^m\}$: $Z = \mathcal{L} \cap P^c$. A Z-polyhedron can in fact be regarded as an affine image of an integer polyhedron. The hypotheses on the affine function ensure the compliance with the conditions of Le Verge [1995] for an LBL to be a Z-polyhedron. This representation has been showed to be complete by Quinton et al. [1996]. We will rely on a number of previous results concerning Z-polyhedra [Gupta and Rajopadhye 2007]:

- The intersection of two Z-polyhedra is still a Z-polyhedron.
- The union of Z-polyhedra, called a *Z-domain*, is not a Z-polyhedron in general.
- The difference between two Z-polyhedra is a Z-domain.
- The preimage of a Z-polyhedron by an affine invertible function is a Z-polyhedron.
- The image of a Z-polyhedron by an arbitrary affine function is an LBL.
- An LBL, and hence a Presburger set, can be expressed as a union of Z-polyhedra [Gupta and Rajopadhye 2007]. A solution for deriving such union from a generic LBL is given in Seghir [2012].

Z-polyhedra can be used to represent execution information of program loop nests, particularly the so-called affine SCoPs, having compile-time predictable control flow as well as loop bounds, subscripts, and conditionals expressed as affine functions of the loop iterators, which is common in a wide range of HPC and scientific program kernels, such as image processing and linear algebra operations. Representing SCoP code by means of parametric polyhedra enables a precise and instance-wise representation of the execution information, unlike abstract syntax trees (ASTs) normally used by compilers, as well as the composition of well-known loop transformations in a single step

Fig. 2. (a) A simple example of a parallel loop. (b) The schedule function of the statement in the loop body. (c) A representation of the memory space. The (i, j) pairs corresponding to the iteration domain of the loop are emphasized by the gray background. The blue rectangles represent parallel sets of iterations. (d) The red parallelograms represent the memory locations (datasets) accessed by the parallel iterations in part (c).

[Cohen et al. 2004]. In fact, some compilers currently embody tools for code manipulation based on the polyhedral abstraction, such as GCC Graphite, LLVM Polly, and RStream [Meister et al. 2011]. Following are a few mathematical concepts linking the Z-polyhedral model with the representation of SCoP code.

The *iteration vector* $\vec{v}$ of a given loop nest is a vector having as elements the indices of the surrounding loops along with the parameters and the constant term. As an example, the iteration vector for the loop nest in Figure 2(a) is $(i, j, N, 1)$.² Whereas the iteration vector represents an execution instance of the loop nest, the *iteration domain* D_S of a statement S in a loop nest is the set of values $\vec{v}$ satisfying the loop bound constraints. Because an iteration domain is delimited by affine loop-bound constraints, it can be expressed as $D_S = \{\vec{v} : \mathbf{D}_S \cdot \vec{v} \geq \vec{0}\}$ for a suitable matrix $\mathbf{D}_S$. Hence, it turns out to be a (parametric) integer polytope, or a Z-polytope in case of nonunit strides.

Each reference to an array A in the loop nest is characterized by a *memory access function* $F(\vec{v}) : \mathbb{Z}^n \rightarrow \mathbb{Z}^d$, where d is the dimensionality of array A in memory. The access function F associates each value of the iteration vector $\vec{v}$ with a unique cell of array A . Since the subscripts in SCoP code are affine functions, F can always be expressed as $F = \mathbf{F} \cdot \vec{v}$, where $\mathbf{F}$ is a $d \times |\vec{v}|$ matrix. As an example, the first reference to array A in Figure 2(a) has the access function $F(\vec{v}) = \mathbf{F} \cdot \vec{v} = (i - 1, j - 1)$. The set of integer points (memory locations) accessed by a certain reference in a statement S can be regarded as the image of the Z-polytope D_S (i.e., the iteration domain) by the affine access function F of that memory reference. The result is generally an LBL, hence a union of Z-polyhedra, since we cannot guarantee the invertibility of F .

Although not relevant to this work, polyhedra are also essential for representing data dependencies between statements. In particular, given two statements S_i and S_j and their iteration vectors $\vec{v}$ and $\vec{w}$, respectively, we can build a *dependence polyhedron*, having a dimensionality equal to $|\vec{v}| + |\vec{w}|$, that includes all pairs of instances $(\vec{v}, \vec{w})$ such that $\vec{v} \in D_{S_i}$, $\vec{w} \in D_{S_j}$ and $\vec{w}$ has a dependency on $\vec{v}$.

Each instance $\vec{v}$ of a statement S in a loop nest can be associated to a point in time based on a *schedule function* $\Theta_S(\vec{v})$, which in fact establishes an ordering of the instances of S , possibly based on a parallelizing transformation. The schedule function has the form $\Theta_S(\vec{v}) = \Theta \cdot \vec{v}$, where Θ is an $n \times |\vec{v}|$ matrix. The parallelism in the code can be exposed by transforming the iteration domain and making the schedule matrix Θ have a zero column corresponding to each parallel for loop. In Figure 2(b), the schedule has four columns: two for the loop iterators i and j , one for parameter N , and one for the constant term. In this particular example, the outer loop was kept serial, whereas the inner loop was parallelized. Consequently, the schedule is one dimensional—that is, the matrix has one row. The parallelism can be easily recognized by the fact that

²Throughout the article, the notation $(i, j, \dots)$ will always denote a column vector.

the schedule has a zero term in the position corresponding to the parallel loop iterator: all instances with the same value of i can be executed in parallel independent of the value taken by j .

To model data-level parallelism, we rely on the concept of parametric *polyhedral slice*, defined for each statement S as follows:

$$P_S(\vec{k}) = \left\{ \vec{v} : \begin{pmatrix} \mathbf{D}_S & \mathbf{0} \\ \Theta_S & -\mathbf{I} \end{pmatrix} \begin{pmatrix} \vec{v} \\ \vec{k} \end{pmatrix} \begin{array}{c} \geq \\ = \end{array} \begin{pmatrix} \vec{0} \end{pmatrix} \right\}.$$

The horizontal line in the preceding notation separates the inequality constraints, here $\mathbf{D}_S \cdot \vec{v} \geq \vec{0}$, from the equality constraints, here $\Theta_S \cdot \vec{v} = \vec{k}$. The slice $P_S(\vec{k})$ identifies all iteration instances in D_S that correspond to the same schedule value $\vec{k}$. Notice that, being an intersection of Z-polyhedra, $P_S(\vec{k})$ is still a Z-polyhedron. Each component k_h of the parameter vector $\vec{k}$ varies within the projection of the domain D_S on the corresponding serial dimension. In Figure 2, the set of parallel instances are the points along the planes having i as a constant value. Two of these sets are represented by the blue rectangles in Figure 2(c). Vector $\vec{k}$ has only one component because we have only one outer sequential dimension corresponding to the i iterator.

Given an array A and a statement S containing a number of memory references, each with affine access function F_h , call $M(\vec{v})$ the set of memory cells of array A accessed by the statement instance $\vec{v}$. Clearly, $M(\vec{v}) = \bigcup_h F_h(\vec{v})$. The definition can be obviously extended to sets of instances. In particular, $M(P_S(\vec{k})) = \bigcup_h \text{Image}_{F_h}(P_S(\vec{k}))$ is the set of the memory cells referenced by the parallel iterations in the parametric slice $P_S(\vec{k})$. We call $M(P_S(\vec{k}))$ a *dataset*. For a certain statement S , the dataset is a function of the parameter $\vec{k}$ and includes all memory locations that can be potentially accessed simultaneously according to the given schedule. Notice that although $P_S(\vec{k})$ is a Z-polyhedron, $M(P_S(\vec{k}))$ is not necessarily a Z-polyhedron for two different reasons. First, the union of a finite number of Z-polyhedra is not generally a Z-polyhedron. Second, the image of a Z-polyhedron is not necessarily a Z-polyhedron, since the access functions may be noninvertible. In general, $M(P_S(\vec{k}))$ is an LBL, which is formally equivalent to a union of Z-polyhedra, as remarked earlier. Figure 2(d) depicts two datasets, represented by two red parallelograms, corresponding to the parametric slices $i = 1$ and $i = 2$, shown in Figure 2(c).

The presence of multiple statements affects the calculation of the datasets. In particular, some of the statements may have the same schedule function and, hence, be run in parallel. The formulation shown earlier could be extended by defining multiple datasets, each being the union of the datasets of single statements, having the same schedule. For the sake of simplicity, however, in the following we will refer to the case of a single statement.

The parallel datasets defined previously can be expressed as finite unions of Z-polyhedra, allowing a closed formulation of the memory partitioning problem. Furthermore, they can be easily manipulated by existing polyhedral tools, such as those of Loechner [1999], Seghir [2012], and Verdoolaege and Woods [2008].

3.1. Problem Formulation

Although all statement instances in a parametric slice $P_S(\vec{k})$ are scheduled in parallel, they normally access the same data structure in memory. Accesses that conflict on the same memory port may cause parallel instances to be serialized, introducing a considerable performance bottleneck. Ideally, if the memory locations accessed by the

iterations within the same parametric slice (described by the $M(P_S(\vec{k}))$ set) are mapped to independent physical banks, then full parallelization can be achieved.

Consider again Figure 2. Each point (i.e., each memory location of array A) is labeled with the identifier of the bank where the location is mapped. For instance, $A[3][1]$ is mapped to memory bank 4. As shown in the figure, six different banks are used, and the mapping is such that there never are two equal labels in each red parallelogram of Figure 2(d) (i.e., full access parallelization is achieved).

To generalize the earlier reasoning, we introduce the concept of *conflict count*, denoted $MC(\vec{k})$, identifying the maximum number of distinct memory locations in $M(P_S(\vec{k}))$ mapped to the same bank, as a function of $\vec{k}$. In essence, $MC(\vec{k})$ represents the number of *serialized* distinct memory accesses in slice $P_S(\vec{k})$, which is indicative of the time spent for handling memory references. In addition, notice that the definition of $MC(\vec{k})$ only refers to distinct memory accesses. In fact, concurrently scheduled accesses might also include multiple references to the same location, possibly due to more than one parallel statement being executed. Notice that since we assume that the code is correctly parallelized, concurrent conflicting accesses cannot include more than one write operation, whereas the semantic of concurrent write and read operations assumes that the read access gets the old value held by the accessed location, which is consistent with the physical meaning of the concurrent accesses (e.g., in an FPGA memory bank). On the other hand, multiple read operations are simply handled by broadcasting the value to the different operators requesting it (e.g., FPGA-implemented processing blocks).

Based on the definition of conflict count, we can introduce our cost function, simply defined as the summation of values $MC(\vec{k})$ across all the values of $\vec{k}$:

$$C_{\text{time}} = \sum_{\vec{k}} MC(\vec{k}). \quad (1)$$

The preceding function is representative of the overall time cost incurred for memory operations across the execution of the entire kernel. As we will show in the following section, it can be expressed in a closed mathematical form with lattice-based memory partitioning. In particular, the Z-polyhedral theory provides useful results for counting the points in unions of parametric Z-polyhedra. Based on such results, the conflict count turns out to be a piecewise quasipolynomial function of the parameter $\vec{k}$ [Seghir 2012]. In case of multiple statements, as explained earlier, the dataset is indeed a list of sets, each parameterized in $\vec{k}$. In this case, the cost functions C_{time} can still be calculated for every dataset separately and then summed over all sets of the list to evaluate the overall cost, since any two accesses of two different datasets are executed sequentially.

For a given number of physical memory banks NB , each of size $SIZE_i$, the memory partitioning problem can be formulated as follows. A *memory partition* of an array A is a pair of scalar integer functions $g(\vec{m})$, $f(\vec{m})$. Vector $\vec{m}$ has as many components as the dimensionality of A . It represents a memory location and varies in the integer parallelepiped defined by A . Function $g(\vec{m})$ identifies the bank to which $\vec{m}$ is mapped, whereas $f(\vec{m})$ is the (linear) address in that bank. Clearly, we must have $0 \leq f(\vec{m}) < SIZE_{g(\vec{m})}$, whereas the total amount of memory taken by the arrays across the physical banks is

$$C_{\text{size}} = \sum_{0 \leq i < NB} \max_{g(\vec{m})=i} \{f(\vec{m})\}. \quad (2)$$

The memory partitioning problem can be decomposed into

- a bank mapping problem, which consists of finding a suitable function $g(\vec{m})$ that assigns all used locations to existing banks (i.e., $0 \leq g(\vec{m}) < NB$) and minimizes C_{time} ; and
- a storage minimization problem, which consists of finding a suitable function $f(\vec{m})$ that avoids colliding assignments to the same bank (i.e., $\forall \vec{m} \neq \vec{n}, g(\vec{m}) = g(\vec{n}) \implies f(\vec{m}) \neq f(\vec{n})$) and minimizes the total amount of memory C_{size} .

In the ideal case, C_{size} coincides with the number of locations in array A , essentially meaning that the partitioned arrays are perfectly exploited with no holes in memory allocation.

The two preceding problems are tackled separately by two sequential steps. Since we aim to maximize parallelism, we determine the bank mapping as a first step by using the proposed lattice-based partitioning technique. Then, we solve the other problem by an optimal storage minimization approach described later.

4. LATTICE-BASED MEMORY PARTITIONING

The approach presented in this work aims to automatically define efficient partitioning choices minimizing structural conflicts on the memory ports. The proposed methodology assumes that the code is already parallelized by properly rearranging the loops [Griebel and Lengauer 1996; Bondhugula et al. 2007; Feautrier 1992], which corresponds to having zero columns in the schedule matrix, through a preliminary code transformation step based on existing polyhedral tools [Loechner 1999; Verdoolaege and Woods 2008; Seghir 2012]

In essence, the lattice-based memory partitioning technique proposed in this work

- regards a d -dimensional memory array as a polyhedron, precisely a hyperrectangle;
- partitions the hyperrectangle into separate sets—the sets are $\mathbb{Z}$ -polyhedra delimited by the same polyhedron (i.e., the memory array, but having different underlying affine lattices) obtained as translates [Barvinok 2002] of a particular lattice chosen by the methodology so as to minimize the conflict count; and
- generates the new polyhedral representation of the code featuring optimized memory accesses.

As an example, in Figure 2, each set of locations assigned to the same bank forms an integer lattice. Each lattice can be thought of as a translate of the lattice marked with 0, which we denote $\mathcal{L}_0$. Mathematically, $\mathcal{L}_0$ is a full-rank bidimensional lattice spanned by the basis $\mathbf{B}_0 = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$. The remaining lattices are affine lattices. For instance, the set of integer points $\mathcal{L}_1$ can be obtained by translating $\mathcal{L}_0$ by one position along the j dimension: $\mathcal{L}_1 = \mathcal{L}_0 + (1, 0)$. Similarly, $\mathcal{L}_3 = \mathcal{L}_0 + (0, 1)$.

In general, we define the lattice containing the origin as the *fundamental lattice*. As implied by the results summarized in Section 3, it forms a partition of $\mathbb{Z}^d$ along with its translates. The number of distinct translates, including the fundamental lattice itself, is equal to the determinant of the lattice [Barvinok 2002]. Since lattice-based partitioning maps each memory bank to a different translate, the determinant of the fundamental lattice must be equal to the number of actually used banks to fully cover the d -dimensional memory. This establishes a fundamental link between a physical characteristic, the number of memory banks, and the main mathematical object handled by our technique—the lattices. The problem of bank mapping (i.e., determining the $g(\vec{m})$ function defined earlier) directly corresponds to finding the best fundamental lattice.

4.1. Generation of the Solution Space

As implied by the previous remark, the total number of available memory banks NB is an upper bound to the determinant of the fundamental lattice to choose. We carry out an exhaustive search of all candidate lattices based on the ideas presented in Darte et al. [2005], considering all lattices having a determinant less than or equal to NB . Among the solutions reaching the minimum conflict count, we then choose the one requiring the least number of banks—that is, the least determinant. Two solutions ensuring the minimum conflict count and the same number of banks are deemed equivalent, although they may have different second-order implications on the implementation costs. As part of our future work, we will also take into account such implications.

Since d -dimensional lattices are spanned by full-rank $d \times d$ matrices, we can equivalently generate all matrices corresponding to distinct lattices. In that respect, given a matrix $\mathbf{B}$ of rank d in $\mathbb{Z}^{d \times d}$, there exists a unique lower triangular matrix $\mathbf{H} = \{h_{ij}\}$ and a unimodular matrix $\mathbf{U}$ such that $\mathbf{H} = \mathbf{B} \cdot \mathbf{U}$ and $0 \leq h_{ij} < h_{ii}$ for all $j < i$. The matrix $\mathbf{H}$ is called the Hermite normal form (HNF) of $\mathbf{B}$ [Schrijver 1986]. Consequently, we can generate all integer lattices of a given determinant δ by enumerating all distinct lower triangular matrices $\mathbf{H} = \{h_{ij}\}$ such that $\det(\mathbf{H}) = \delta$ and $0 \leq h_{ij} < h_{ii}$. For a given rank d and determinant δ , the number of such matrices, denoted $H_d(\delta)$, is equal to $\prod_{j=1}^{d-1} (p^{k+j} - 1)/(p^j - 1)$, when $\delta = p^k$ for a prime p , whereas $H_d(\delta) = H_d(p)H_d(q)$, if $\delta = p \cdot q$, where p and q are relatively prime numbers [Newman 1972]. For a maximum number of banks NB , the search space thus contains $\sum_{i=2}^{NB} H_d(i)$ solutions. As an example, the fundamental lattice of Figure 2 is one of 32 potential solutions, since we have a two-dimensional array and 6 banks. In case we had 8, 12, 16, 24, or 32 banks, the number of candidates would become 55, 116, 209, 480, 846, respectively.

4.2. Evaluation of the Solutions

Lattice-based partitioning enables the conflict count $MC(\vec{k})$ to be expressed in a closed mathematical form. Call $\mathcal{L}_t$, the t th translate of a given fundamental lattice $\mathcal{L}$, with $0 \leq t < \det(\mathcal{L})$. Then, we simply have

$$MC(\vec{k}) = \max_t |\mathcal{L}_t \cap M(P_S(\vec{k}))|.$$

The intersection picks the points of the dataset M that belong to translate $\mathcal{L}_t$. For each value of $\vec{k}$, the lattice incurring the maximum conflict count determines the worst-case serialization in the memory accesses. The set $\mathcal{L}_t \cap M(P_S(\vec{k}))$ is a union of parametric Z-polyhedra. We first use the technique proposed in Seghir [2012] to count the integer points in the set as a function of $\vec{k}$, obtaining different expressions for determined intervals of $\vec{k}$. Then, by summing such expressions together, we obtain the overall value of the cost function $C_{\text{time}} = \sum_{\vec{k}} MC(\vec{k})$. As an example, in Figure 2, we have $|\mathcal{L}_t \cap M(P_S(\vec{k}))| = 1$ for each of the six translates t , and hence $C_{\text{time}} = N - 1$.

4.3. Enumeration of the Translates

The evaluation of the solutions requires the exhaustive enumeration of the translates of a given fundamental lattice. To this aim, we rely on the following property, which can be proven easily.

PROPERTY 1. *Let $\mathbf{H} = \{h_{ij}\}$ be the HNF basis of a lattice $\mathcal{L}$ and $\vec{m}$ be one of its points. Then, there does not exist any point $\vec{n} \neq \vec{m}$ such that $\forall i \ 0 \leq n_i - m_i < h_{ii} \wedge \vec{n} \in \mathcal{L}$.*

The property implies that if we take a point in $\mathcal{L}$, we cannot find any different point of $\mathcal{L}$ in the parallelepiped having $\vec{m}$ as the lower corner and the diagonal elements h_{ii} as

heights. In fact, notice that the parallelepiped has a volume equal to the determinant of $\mathbf{L}$ because $\mathbf{H}$ is in HNF. As a direct consequence, considering $\vec{m} = \vec{0}$, if we take all elements in the parallelepiped having the diagonal elements h_{ii} as heights, we pick exactly $|\det(\mathbf{H})|$ points belonging to different translates. Since there are exactly $|\det(\mathbf{H})|$ translates, those points cover all of the possible translation vectors. As an example, the HNF of the fundamental lattice $\mathcal{L}_0$ in Figure 2 is $\begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$, so the remaining five translates are given by $\mathcal{L}_0 + \vec{t}$, with $\vec{t} = (1, 0), (2, 0), (0, 1), (1, 1), (2, 1)$.

4.4. Generation of the New Polyhedral Representation

After picking the best partitioning solution, a new version of the code must be generated so as to capture the allocation to memory banks. A possibility is to use an additional dimension in the partitioned array and then apply block or cyclic mapping. Notice, however, that we are mainly interested in the synthesis of hardware accelerators from high-level C code. Although existing HLS tools [Xilinx Inc. 2012] allow block or cyclic partitioning along a given dimension, they look at statements, not *instances* of statements. In other words, they parallelize two memory accesses, say M_1 and M_2 , only if it can be proved that all instances of M_1 and M_2 in the iteration domain of the loop nest access different banks. To address this problem, we decided to explicitly partition the array in the code by declaring multiple distinct arrays. Consider the example in Figure 2. The statement in the loop body contains two memory references, or accesses (i.e., $A[i][j]$ and $A[i-1][j-1]$). Different values of the iterators (j, i) lead to a different bank accessed for each of the two references. For example, instance $(1, 1)$ accesses banks 4 and 0, whereas instance $(2, 1)$ accesses 5 and 1. Indeed, since the dataset has a constant shape and the banks in the example are periodic with period 3 along the j dimension and 2 along the i dimension, there are exactly six different cases (i.e., the number of banks NB), each corresponding to a different statement body:

```

Y[i][j] = A0[i-1][j-1] + A4[i][j]; when (j mod 3, i mod 2) = (1, 1)
Y[i][j] = A1[i-1][j-1] + A5[i][j]; when (j mod 3, i mod 2) = (2, 1)
Y[i][j] = A2[i-1][j-1] + A3[i][j]; when (j mod 3, i mod 2) = (0, 1)
Y[i][j] = A3[i-1][j-1] + A1[i][j]; when (j mod 3, i mod 2) = (1, 0)
Y[i][j] = A4[i-1][j-1] + A2[i][j]; when (j mod 3, i mod 2) = (2, 0)
Y[i][j] = A5[i-1][j-1] + A0[i][j]; when (j mod 3, i mod 2) = (0, 0).

```

In general, depending on the structure of the dataset, there may be up to NB^h different combinations of bank accesses, where h is the number of memory references in the statement. The essential idea in our approach is to generate a different statement for each combination of bank accesses (e.g. six statements in the preceding example). To describe the new statements in terms of polyhedral representation, we start from the original iteration domain D_S and generate NB smaller integer sets, each corresponding to a different statement body, covering a specific combination of bank accesses. To express this mathematically, further constraints must be added to the algebraic form describing the original polyhedron. Call r the number of distinct memory references in the statement, and let d be the dimensionality of the array (e.g., $d = 2$ for bidimensional arrays). For each combination p of bank accesses, the iteration domain of the corresponding statement can be expressed as follows:

$$D_{Sp} = \left\{ \vec{v} : \exists (\vec{y}_0 \dots \vec{y}_{r-1}) \in \mathbb{Z}^{r \cdot d} : \begin{pmatrix} \mathbf{D}_S & \mathbf{0} & \mathbf{0} \dots \mathbf{0} \\ \mathbf{F}_0 & -\mathbf{B} & \mathbf{0} \dots \mathbf{0} \\ \mathbf{F}_1 & \mathbf{0} & -\mathbf{B} \dots \mathbf{0} \\ \vdots & \vdots & \vdots \\ \mathbf{F}_{r-1} & \mathbf{0} & \mathbf{0} \dots -\mathbf{B} \end{pmatrix} \begin{pmatrix} \vec{v} \\ \vec{y}_0 \\ \vdots \\ \vec{y}_{r-1} \end{pmatrix} \geq \begin{pmatrix} \vec{0} \\ \vec{T}_0^p \\ \vdots \\ \vec{T}_{r-1}^p \end{pmatrix} \right\}.$$

The matrix has as many columns as the original polyhedron matrix $\mathbf{D}_S$ plus the number of accesses r times the lattice rank d . $\mathbf{B}$ is the $d \times d$ basis of the chosen fundamental lattice. $\mathbf{F}_i$ are the $d \times |\vec{v}|$ access matrices in the original code. The r vectors $\vec{y}_i$ contains each d integer parameters. $\vec{T}_i^p$ is a d -element vector denoting the translate (i.e., the bank), where the i th access is mapped in combination p . The set of vectors $\vec{T}_i^p$ identifies one of the possible combinations of bank accesses. Each line in the bottom part of the preceding matrix multiplication results in a constraint looking like $\mathbf{F}_i \cdot \vec{v} - \mathbf{B} \cdot \vec{y}_i = \vec{T}_i^p$. For instance, concerning the example in Figure 2, the constraint $\exists (y_1, y_2) \in \mathbb{Z}^2 : (j, i) - \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} (y_1, y_2) = (1, 0)$ for access $A[i][j]$ picks the iterations (j, i) that access locations belonging to translate $(1, 0)$ of the fundamental lattice $\begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$. Notice that $\vec{T}_i^p$ cannot be made parameters because we need to pass separate algebraic structures to the code generator, one for each version of the statement. The preceding structures can be handled by existing tools (e.g., ISL, CLoog) for code generation from the polyhedral model, which can identify and efficiently prune out empty polyhedra. In addition, notice that due to the potentially noninvertible nature of the affine access functions, the sets D_{Sp} can be unions of Z-polyhedra rather than single Z-polyhedra. However, in case the access functions are invertible, D_{Sp} can be proved to be a single Z-polyhedron, delimited by the same polyhedron as D_S , yielding a representation that can be manipulated easier by existing polyhedral tools.

4.5. Storage Minimization

The problem of storage minimization—that is, the definition of the $f(\vec{m})$ function—consists of assigning a new address within the new bank to each original memory location $\vec{m}$. This corresponds to determining new access functions for each memory reference such that the proper location is accessed in each iteration. Call $F(\vec{v})$ the access function corresponding to the iteration vector $\vec{v}$ for a certain reference, and let $\vec{m} = (m_0, \dots, m_{d-1})$ be the accessed location (i.e., $\vec{m} = F(\vec{v})$). Let $\vec{m}' = (m'_0, \dots, m'_{d-1}) = F'(\vec{v})$ be the new access function. Lattice-based partitioning allows a straightforward solution to the storage minimization problem. In fact, because of the cyclic repetitions along the axes of the memory space, which occurs for any lattice chosen, the address in each bank can simply be obtained by scaling the original addresses. Precisely, given $\mathbf{H} = \{h_{ij}\}$, the HNF basis of the fundamental lattice, the new address $\vec{m}'$ can simply be obtained as $\vec{m}' = (\lfloor \frac{m_0}{h_{00}} \rfloor, \dots, \lfloor \frac{m_{d-1}}{h_{d-1,d-1}} \rfloor)$. In other words, the new access can be written as $\vec{m}' = \lfloor \mathbf{S} \cdot \mathbf{F} \cdot \vec{v} \rfloor$, where

$$\mathbf{S} = \begin{pmatrix} \frac{1}{h_{00}} & 0 & 0 & \dots & 0 \\ 0 & \frac{1}{h_{11}} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \frac{1}{h_{d-1,d-1}} \end{pmatrix}.$$

The scalar function $f(\vec{m})$ is then given by the linearized address of $\vec{m}'$ in bank $g(\vec{m})$. It can be easily shown that the preceding solution is consistent and asymptotically optimum. In fact, by Property 1, if $\vec{m}$ and $\vec{n}$ are two distinct locations belonging to the same lattice, then there must be at least one i such that $|m_i - n_i| \geq h_{ii}$. Consequently, they are certainly mapped to different locations in the same bank, since $|\lfloor \frac{m_i}{h_{ii}} \rfloor - \lfloor \frac{n_i}{h_{ii}} \rfloor| \geq 1$. Furthermore, calling D_i the size of the original array along dimension i , the size of each scaled array along the same dimension is no larger than $\lceil \frac{D_i}{h_{ii}} \rceil$. In other words, each array will require no more than $\prod_i \lceil \frac{D_i}{h_{ii}} \rceil$ locations. Since the number of banks is $NB = \prod_i h_{ii}$, then the total number of locations across all banks (i.e., the C_{size} cost function defined

Fig. 3. Prototype toolchain implementing lattice-based partitioning.

earlier) is upper bounded by $\prod_i \lceil \frac{D_i}{h_{ii}} \rceil \cdot h_{ii} = \prod_i (\frac{D_i}{h_{ii}} + \delta_i) \cdot h_{ii} = \prod_i (D_i + \delta_i h_{ii})$, with $\delta_i < 1$. Under the assumption that $h_{ii} = o(D_j) \forall i, j$, the preceding upper bound to C_{size} can be written as $\prod_i D_i + o(\prod_i D_i) = D + o(D)$, where $D = \prod_i D_i$ is the total amount of memory taken by the original array. In other words, C_{size} is asymptotically equal to the original amount of memory—that is, the banks are densely populated with a number of holes becoming negligible as the size of the array is increased.

The preceding optimal solution involves a zero amount of asymptotic memory waste. Unfortunately, however, it requires an integer division by h_{ii} for each component in $\vec{m}$ before starting a memory access. In a case where h_{ii} is a power of 2, which indeed is likely to happen in practice, division simply coincides with a right shift. For cases where h_{ii} is not a power of 2, on the other hand, we propose an efficient solution that involves an arbitrarily small amount of asymptotic waste along each dimension of the memory. In essence, the solution is based on the idea of replacing the integer division by h_{ii} with a more efficient multiplication by an integer constant a , followed by a right shift by b bits. In the following, we will refer to a single component m_i for the sake of simplicity. In essence, we replace $m'_i = \lfloor \frac{m_i}{h_{ii}} \rfloor$ with $m'_i = \lfloor \frac{m_i \cdot a}{2^b} \rfloor$, with $\frac{2^b}{a} \leq h_{ii}$. The maximum value taken by the m'_i will thus increase from $\lfloor \frac{D_i-1}{h_{ii}} \rfloor$ to $\lfloor \frac{(D_i-1) \cdot a}{2^b} \rfloor$, stretching the memory used by around $\frac{h_{ii} \cdot a}{2^b}$ along dimension i . We show that, in principle, it is possible to choose a and b such that any arbitrarily small amount of waste can be obtained. Call ω the fraction of memory waste to obtain (e.g., $\omega = 0.1$ indicates a memory waste of 10% along dimension i). Take $b > \log_2 \frac{h_{ii}}{\omega}$ and $a = \lceil \frac{2^b}{h_{ii}} \rceil$. First of all, based on this choice, we have that $\frac{a}{2^b} \geq \frac{1}{h_{ii}}$, still ensuring that $|\lfloor \frac{m_i \cdot a}{2^b} \rfloor - \lfloor \frac{m_i}{h_{ii}} \rfloor| \geq 1$ if $|m_i - n_i| \geq h_{ii}$. Furthermore, the stretching factor is $\frac{h_{ii} \cdot a}{2^b} = \frac{h_{ii} \cdot \lceil \frac{2^b}{h_{ii}} \rceil}{2^b} = \frac{h_{ii} \cdot (\frac{2^b}{h_{ii}} + \delta)}{2^b} = 1 + \frac{h_{ii} \cdot \delta}{2^b}$, with $\delta < 1$. The percentage waste, represented by the term $\frac{h_{ii} \cdot \delta}{2^b}$, is by construction less than ω , as required, because $b > \log_2 \frac{h_{ii}}{\omega}$.

Depending on the values of h_{ii} and the actual cost of an integer division, one of the two solutions shown earlier can be adopted to effectively address the problem of storage minimization. As an example, assume that we have $h_{ii} = 3$ for dimension i , as in the example of Figure 2, and that we require a percentage waste upperbounded by 10%. We can choose $b > \log_2 \frac{h_{ii}}{\omega} = \log_2 30$ (e.g., $b = 5$) and $a = \lceil \frac{2^b}{h_{ii}} \rceil = \lceil \frac{32}{3} \rceil = 11$. Then, in the transformed code, each array subscript corresponding to dimension i , expressed as $\mathbf{F}_i \cdot \vec{v}$ (where $\mathbf{F}_i$ is the i th row of the access matrix $\mathbf{F}$) will be replaced with $(11 \cdot \mathbf{F}_i \cdot \vec{v}) \gg 5$.

5. A DETAILED CASE STUDY

To demonstrate the lattice-based partitioning technique, we built a prototype toolchain, depicted in Figure 3. The implementation is based on a variety of open-source tools for

```

for(t=0; t<T; t++)
  for(i=1; i<N-1; i++)
    for(j=1; j<N-1; j++)
      S(i,j): Y[i][j] =
        W1*A[i-1][j-1] + W2*A[i-1][j] +
        W3*A[i-1][j+1] + W4*A[i][j-1] +
        W5*A[i][j] + W6*A[i][j+1] +
        W7*A[i+1][j-1] + W8*A[i+1][j] +
        W9*A[i+1][j+1];
(a)

```

```

for(t=0; t<=T-1; t++)
  for(i=0; i<=N-3; i++)
    for(j=0; j<=(N-4)/2; j++)
      parfor(p=0; p<=1; p++)
        S(i,j+p): Y[i+1][2*j+1+p] =
          W1*A[i][2*j+p] + W2*A[i][2*j+1+p] +
          W3*A[i][2*j+2+p] + W4*A[i+1][2*j+p] +
          W5*A[i+1][2*j+1+p] + W6*A[i+1][2*j+2+p] +
          W7*A[i+2][2*j+p] + W8*A[i+2][2*j+1+p] +
          W9*A[i+2][2*j+2+p];
(b)

```

Fig. 4. 2D Windowing kernel. (a) Original version. (b) $2 \times$ Stripmined and parallelized version.

polyhedral analysis. First, the input is analyzed by Clan,³ used to extract a polyhedral representation. Then, it is scheduled to extract as much internal parallelism as possible. This step is optional since our mathematical framework does not deal with parallelism extraction but assumes an already scheduled loop nest. Subsequently, the representation is manipulated by ad hoc tools applying the proposed methodology. The process relies on the OpenScop⁴ format to represent the information on the code being transformed, PolyLib [Loechner 1999] to manipulate polyhedra, and ZPolyTrans [Seghir 2012] to handle images of Z-polyhedra by noninvertible functions and parametric counting of points in unions of Z-polyhedra. There are other tools that could serve the purpose, particularly those of Verdoolaege and Woods [2008], whereas ISL supports relations and hence Z-polyhedra, but it does not provide specialized algorithms [Iooss and Rajopadhye 2012]. Last, after manipulation, C code is regenerated using CLoog [Bastoul 2004].

Based on the preceding toolchain, this section provides a step-by-step illustration of the lattice-based partitioning technique to a real-world application, a bidimensional window filtering kernel, where a waveform/sequence is multiplied by a window function [Weisstein 2003]. Applications of window functions include spectral analysis, filter design, and beamforming. More specifically, in this section, we target bidimensional signals expressed as arrays (e.g., images) and 3×3 window functions expressed as nine separate coefficients. Figure 4(a) shows the bidimensional filter kernel consisting of a perfect nest made of three loops. The inner loops iterate over space, whereas the outer loop iterates over time since the filter may be applied several times to the same signal to amplify the effect or use different coefficients.

Some parallelism has been extracted by stripmining the innermost loop by a factor SF . Figure 4(b) shows the stripmined and parallelized code. Although we can set SF to any value less than $N - 1$, for the sake of simplicity, here we use $SF = 2$. The constant N is assumed to be even. All loops in the code are normalized—that is, their iterators start from 0 and have unit stride. The innermost for construct is marked as parallel. In fact, it is evident that it can be parallelized since there are no true dependencies carried by the three innermost loops.

In our example, the memory architecture is assumed to provide up to six independent memory banks. We apply the lattice-based partitioning technique to the input array A ; a similar procedure may be applied to the output array Y .

Polyhedral model of the kernel. The kernel contains four loops, and hence the iteration vector $\vec{v}$ is $\vec{v} = (t, i, j, p, T, N, 1)$, with N and T being two constant parameters. The

³<http://icps.u-strasbg.fr/~bastoul/development/clan/index.html>.

⁴<http://icps.u-strasbg.fr/~bastoul/development/openscop/>.

normalized iteration domain of statement S is the following integer polyhedron:

$$D_S = \{\vec{v} : \mathbf{D}_S \cdot \vec{v} \geq \vec{0}\}, \quad \mathbf{D}_S = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & -3 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 1 & -4 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix}.$$

A three-dimensional schedule can be derived for statement S directly from the code in Figure 4(b):

$$\Theta_s(\vec{v}) = \Theta_S \cdot \vec{v} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \vec{v}$$

The fourth column, corresponding to p , contains all zero elements, meaning that the loop iterations are scheduled concurrently.

Modeling data-level parallelism. The first step of the methodology consists of modelling the parallelism of the memory accesses. First, we need to build the parallel parametric slice $P_S(\vec{k})$. In this example, $\vec{k}$ is $\vec{k} = (k_1, k_2, k_3)$ because we have three serial loops. Parameters k_h directly correspond to the iterators of the three outermost loops (i.e., t , i , and j) and vary within the same intervals. The parametric slice $P_S(\vec{k})$ is the polyhedron of iterations $\vec{v}$ satisfying

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & -3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 1 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ - & - & - & - & - & - & - & - & - & - \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} t \\ i \\ j \\ p \\ T \\ N \\ 1 \\ k_1 \\ k_2 \\ k_3 \end{pmatrix} \begin{matrix} \geq \\ - \\ = \end{matrix} \vec{0}.$$

For each value of the parameter $\vec{k} = (k_1, k_2, k_3)$, the slice turns out to contain two iterations, those corresponding to $p = 0$ and $p = 1$.

The statement in the kernel of Figure 4(b) contains nine bidimensional affine accesses to array A , each having its own access function. As an example, $A[i + 1][2j + p]$ has $F(\vec{v}) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 1 & 0 & 0 \end{pmatrix} \cdot \vec{v}$ as its access function. The next step thus concerns the identification of the datasets determined by these accesses—that is, the set $M(P_S(\vec{k}))$ containing the memory locations accessed in parallel by concurrent iterations. As explained earlier, the datasets are the union of the images of $P_S(\vec{k})$ by the affine access functions. Each image represents a set of locations accessed in parallel by a specific memory reference in the statement. In this case, although the memory access functions are noninvertible, the images are still Z-polyhedra (here, indeed, they are simply polyhedra). Furthermore, their unions also turn out to be polyhedra. Figure 5 depicts a few examples of images and datasets for a couple of values of the parameter $\vec{k} = (k_1, k_2, k_3)$. Although the datasets are a function of $\vec{k}$ and their cardinality may generally vary, here all datasets always contain 12 memory points. In this example, they can be formally

Fig. 5. Example of datasets for our case study. For dataset $M(P_S(t, 0, 0))$, the figure also shows the images of the slice P_S by two single access functions. Overall, each dataset is formed by nine such images.

expressed as follows:

$$M(P_S(\vec{k})) = \left\{ (j, i) \in \mathbb{Z}^2 : \begin{pmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 0 & -2 & 0 \\ 0 & -1 & 0 & 0 & 2 & 3 \end{pmatrix} \begin{pmatrix} i \\ j \\ k_1 \\ k_2 \\ k_3 \\ 1 \end{pmatrix} \geq \vec{0} \right\}.$$

Generation of the solution space. Since there are six memory banks available, the methodology looks for the best bidimensional lattice-based partitioning solution having a determinant less than or equal to 6. Indeed, since each dataset contains 12 distinct accesses, the best we can do is to have 2 accesses per memory bank in each dataset, which is only possible with six banks. As a consequence, in this example, we will only discuss the case $\det(\mathcal{L}) = 6$. As explained in Section 4.1, enumerating all distinct bidimensional lattices is equivalent to generating all possible 2×2 matrices in HNF with a determinant equal to 6. Relying on the quantitative formula given in Section 4.1, we know that there are $H_2(6) = H_2(3)H_2(2) = 3 \times 4 = 12$ such matrices. Next, we list all of them:

$$\mathbf{B}_1 = \begin{pmatrix} 6 & 0 \\ 0 & 1 \end{pmatrix}; \mathbf{B}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 6 \end{pmatrix}; \mathbf{B}_3 = \begin{pmatrix} 1 & 0 \\ 1 & 6 \end{pmatrix}; \mathbf{B}_4 = \begin{pmatrix} 1 & 0 \\ 2 & 6 \end{pmatrix}; \mathbf{B}_5 = \begin{pmatrix} 1 & 0 \\ 3 & 6 \end{pmatrix}; \mathbf{B}_6 = \begin{pmatrix} 1 & 0 \\ 4 & 6 \end{pmatrix};$$

$$\mathbf{B}_7 = \begin{pmatrix} 1 & 0 \\ 5 & 6 \end{pmatrix}; \mathbf{B}_8 = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}; \mathbf{B}_9 = \begin{pmatrix} 2 & 0 \\ 1 & 3 \end{pmatrix}; \mathbf{B}_{10} = \begin{pmatrix} 2 & 0 \\ 2 & 3 \end{pmatrix}; \mathbf{B}_{11} = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}; \mathbf{B}_{12} = \begin{pmatrix} 3 & 0 \\ 1 & 2 \end{pmatrix}.$$

Each enumerated solution is representative of an infinite number of bases all equal up to a unimodular transformation. Figure 6 shows the lattices corresponding to $\mathbf{B}_8$ and $\mathbf{B}_3$. Once the solution space is generated, we need to evaluate each potential solution to identify the best fundamental lattice.

Enumeration of the translates. To proceed, we first need to enumerate the translates of each lattice identified previously. Take $\mathbf{B}_8 = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$ as an example. As pointed out in Section 4.3, we need to enumerate all points in the parallelepiped identified by the diagonal of $\mathbf{B}_8$, here a 2×3 rectangle, with the lower corner in the origin. The points are as follows:

$$t_0 = (0, 0), \quad t_1 = (1, 0), \quad t_2 = (0, 1), \quad t_3 = (1, 1), \quad t_4 = (0, 2), \quad t_5 = (1, 2).$$

These vectors correspond to the six translates of the fundamental lattice needed to cover $\mathbb{Z}^2$. By construction, their number is equal to the number of memory banks.

Fig. 6. Two different mapping solutions corresponding to the fundamental lattices $\mathcal{L}(\mathbf{B}_8)$ and $\mathcal{L}(\mathbf{B}_3)$. For each of the two solutions, the figure highlights one of the translates causing the highest conflict count.

Evaluation of the solution space. To evaluate a solution, we compute the maximum number of conflicting accesses (i.e., accesses to the same memory bank) within all parallel datasets (see Section 4.2). For each parallel dataset, the conflicts for a given translate $\mathcal{L}_t$ are given by the intersection of the affine lattice $\mathcal{L}_t$ itself and the polyhedron describing the dataset. The number of integer points, what we called $MC(\vec{k})$, contained in such Z-polyhedron represents, parametrically, the number of conflicts corresponding to a particular translate $\mathcal{L}_t$ for a certain solution $\mathcal{L}$. Figure 6 also shows graphically $MC(\vec{k})$ for $\vec{k} = (t, 0, 0)$ and the translates corresponding to the vectors $(0, 0)$ and $(1, 0)$ for $\mathcal{L}(\mathbf{B}_8)$ and $\mathcal{L}(\mathbf{B}_3)$, respectively. The number of points in the first Z-polyhedron is 2, whereas we count 3 points in the second Z-polyhedron, resulting in worse performance. In both cases, they are independent of $\vec{k}$. Following are the overall memory access times, computed by taking into account the worst-case conflicts within the datasets (which are necessarily serialized in time):

$$\begin{aligned}
 C_{\text{time}}(\mathbf{B}_1) &= (N-2) \cdot (N-2) \cdot T \cdot 3 & C_{\text{time}}(\mathbf{B}_2) &= (N-2) \cdot (N-2) \cdot T \cdot 4 \\
 C_{\text{time}}(\mathbf{B}_3) &= (N-2) \cdot (N-2) \cdot T \cdot 3 & C_{\text{time}}(\mathbf{B}_4) &= (N-2) \cdot (N-2) \cdot T \cdot 3 \\
 C_{\text{time}}(\mathbf{B}_5) &= (N-2) \cdot (N-2) \cdot T \cdot 2 & C_{\text{time}}(\mathbf{B}_6) &= (N-2) \cdot (N-2) \cdot T \cdot 3 \\
 C_{\text{time}}(\mathbf{B}_7) &= (N-2) \cdot (N-2) \cdot T \cdot 3 & C_{\text{time}}(\mathbf{B}_8) &= (N-2) \cdot (N-2) \cdot T \cdot 2 \\
 C_{\text{time}}(\mathbf{B}_9) &= (N-2) \cdot (N-2) \cdot T \cdot 2 & C_{\text{time}}(\mathbf{B}_{10}) &= (N-2) \cdot (N-2) \cdot T \cdot 2 \\
 C_{\text{time}}(\mathbf{B}_{11}) &= (N-2) \cdot (N-2) \cdot T \cdot 4 & C_{\text{time}}(\mathbf{B}_{12}) &= (N-2) \cdot (N-2) \cdot T \cdot 3.
 \end{aligned}$$

In Section 4.2, the expression of C_{time} contained a sum on $\vec{k}$. Here, because there is no dependence on $\vec{k}$, the sum is reduced to a multiplication by the number of values spanned by $\vec{k} = (k_1, k_2, k_3)$, i.e., $(N-2) \cdot (N-2) \cdot T$. Notice that $\mathbf{B}_5$, $\mathbf{B}_8$, $\mathbf{B}_9$, and $\mathbf{B}_{10}$ all achieve the minimum number of conflicts. Here, we choose $L = \mathbf{B}_8$.

Storage minimization. The chosen fundamental lattice is $\mathcal{L}(\mathbf{B}_8)$, with $\mathbf{B}_8 = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$. Based on the results in Section 4.5, we can simply replace each original access in the

new arrays by scaling the column subscript by 2 and the row subscript by 3, obtaining an asymptotically zero memory waste. As an alternative, to avoid the integer division by 3, we can adopt the approximate approach introduced in Section 4.5, which consists of replacing the division with a multiplication by an integer a followed by a b -bit right shift. As already exemplified in Section 4.5 for the case $h_{ii} = 3$ and a maximum waste of 10%, we can use $b > \log_2 \frac{3}{0.1}$ (e.g., $b = 5$) and consequently $a = \lceil \frac{2^5}{3} \rceil = 11$. Formally, for each access function in the statement of Figure 4, the new access function can be obtained as $F'(\vec{v}) = \lfloor (\frac{1/2}{0 \ 11/32}) \cdot F(\vec{v}) \rfloor$.

As a practical example of memory waste, consider the case $N = 100$, requiring $N^2 = 10,000$ memory locations. The highest modified subscripts are $[11 \cdot (i + 2) >> 5][2 \cdot j + 2 + p >> 1]$, and they take as their maximum value $[34][48]$, corresponding to the bounds $i = N - 3 = 97$, $j = (N - 4)/2 = 48$, $p = 1$. In other words, we need $35 \cdot 49 = 1,715$ locations for each of the six arrays—that is, 10,290 locations overall, with an actual waste below 3%.

Code generation. Based on the technique presented in Section 4.4, we can generate the new polyhedral representation maximizing parallel memory bank accesses. The representation leads to the following optimized code:

```
for( t=0; t<=T-1; t++ ) {
  for( i=0; i<=N-3; i++ ) {
    if( i%3==0 ) {
      for( j=0; j<=(N-4)/2; j++ ) {
        parfor( p=0; p<=1; p++ ) {
          if( (j+p)%2==0 )
            S1: Y[i+1][2*j+1+p] = W1*A0[(11*i)>>5][(2*j+p)>>1] +
              W2*A1[(11*i)>>5][(2*j+1+p)>>1] + W3*A0[(11*i)>>5][(2*j+2+p)>>1] +
              W4*A2[(11*i+11)>>5][(2*j+p)>>1] + W5*A3[(11*i+11)>>5][(2*j+1+p)>>1] +
              W6*A2[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A4[(11*i+22)>>5][(2*j+p)>>1] +
              W8*A5[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A4[(11*i+22)>>5][(2*j+2+p)>>1];
          else
            S2: Y[i+1][2*j+1+p] = W1*A1[(11*i)>>5][(2*j+p)>>1] +
              W2*A0[(11*i)>>5][(2*j+1+p)>>1] + W3*A1[(11*i)>>5][(2*j+2+p)>>1] +
              W4*A3[(11*i+11)>>5][(2*j+p)>>1] + W5*A2[(11*i+11)>>5][(2*j+1+p)>>1] +
              W6*A3[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A5[(11*i+22)>>5][(2*j+p)>>1] +
              W8*A4[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A5[(11*i+22)>>5][(2*j+2+p)>>1];
        }
      }
    } else if( i%3==1 ) {
      for( j=1; j<=N - 2; j+=2 ) {
        parfor( p=0; p<=1; p++ ) {
          if( (j+p)%2==0 )
            S3: Y[i+1][2*j+1+p] = W1*A2[(11*i)>>5][(2*j+p)>>1] +
              W2*A3[(11*i)>>5][(2*j+1+p)>>1] + W3*A2[(11*i)>>5][(2*j+2+p)>>1] +
              W4*A4[(11*i+11)>>5][(2*j+p)>>1] + W5*A5[(11*i+11)>>5][(2*j+1+p)>>1] +
              W6*A4[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A0[(11*i+22)>>5][(2*j+p)>>1] +
              W8*A1[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A0[(11*i+22)>>5][(2*j+2+p)>>1];
          else
            S4: Y[i+1][2*j+1+p] = W1*A3[(11*i)>>5][(2*j+p)>>1] +
              W2*A2[(11*i)>>5][(2*j+1+p)>>1] + W3*A3[(11*i)>>5][(2*j+2+p)>>1] +
              W4*A5[(11*i+11)>>5][(2*j+p)>>1] + W5*A4[(11*i+11)>>5][(2*j+1+p)>>1] +
              W6*A5[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A1[(11*i+22)>>5][(2*j+p)>>1] +
              W8*A0[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A1[(11*i+22)>>5][(2*j+2+p)>>1];
        }
      }
    }
  }
}
```

```

}else if( i%3==2 ){
    for( j=1; j<=N - 2; j+=2 ) {
        parfor( p=0; p<=1; p++ ) {
            if( (j+p)%2==0 )
                S5: Y[i+1][2*j+1+p] = W1*A4[(11*i)>>5][(2*j+p)>>1] +
                    W2*A5[(11*i)>>5][(2*j+1+p)>>1] + W3*A4[(11*i)>>5][(2*j+2+p)>>1] +
                    W4*A0[(11*i+11)>>5][(2*j+p)>>1] + W5*A1[(11*i+11)>>5][(2*j+1+p)>>1] +
                    W6*A0[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A2[(11*i+22)>>5][(2*j+p)>>1] +
                    W8*A3[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A2[(11*i+22)>>5][(2*j+2+p)>>1];
            else
                S6: Y[i+1][2*j+1+p] = W1*A5[(11*i)>>5][(2*j+p)>>1] +
                    W2*A4[(11*i)>>5][(2*j+1+p)>>1] + W3*A5[(11*i)>>5][(2*j+2+p)>>1] +
                    W4*A1[(11*i+11)>>5][(2*j+p)>>1] + W5*A0[(11*i+11)>>5][(2*j+1+p)>>1] +
                    W6*A1[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A3[(11*i+22)>>5][(2*j+p)>>1] +
                    W8*A2[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A3[(11*i+22)>>5][(2*j+2+p)>>1];
        }
    }
}
}
}
}

```

Complying with the mapping choice previously identified, no more than two conflicts per parallel set of iterations are incurred. Consider the first two parallel statements S_1 and S_2 as an example. $A0$ appears three times, and thus the iteration seemingly contains three simultaneous accesses. However, the three accesses can only involve two different values because of the mapping that we enforced. In fact, $A0[(11 \cdot i) >> 5][(2 \cdot j + 1 + p) >> 1]$ in the else branch coincides with $A0[(11 \cdot i) >> 5][(2 \cdot j + 2 + p) >> 1]$ in the then branch if j is even (because if j is even, p must be equal to 1 in the else branch and 0 in the then branch) or with $A0[(11 \cdot i) >> 5][(2 \cdot j + p) >> 1]$ in the then branch if j is odd (because if j is odd p must be equal to 0 in the else branch and 1 in the then branch). The same considerations apply to the other accesses.

Validation of the cost function. To conclude our case study, we show a few results collected from a practical implementation of the kernel targeted at an FPGA technology by means of an HLS process. In particular, the preceding C code was synthesized to VHDL by the Vivado HLS tool [Xilinx Inc. 2012] and analyzed by cycle-accurate simulation. The actual execution times, denoted ET and expressed in terms of clock count, refer to the case $N = 20$, $T = 8$. They were measured for each of the 12 lattice-based partitioning solutions explored earlier:

$$ET(\mathbf{B}_1) = 3,889; \quad ET(\mathbf{B}_2) = 5,187; \quad ET(\mathbf{B}_3) = 3,889; \quad ET(\mathbf{B}_4) = 3,889;$$

$$ET(\mathbf{B}_5) = 2,595; \quad ET(\mathbf{B}_6) = 3,889; \quad ET(\mathbf{B}_7) = 3,889; \quad ET(\mathbf{B}_8) = 2,595;$$

$$ET(\mathbf{B}_9) = 2,595; \quad ET(\mathbf{B}_{10}) = 2,595; \quad ET(\mathbf{B}_{11}) = 5,187; \quad ET(\mathbf{B}_{12}) = 3,889.$$

The results provide a clear confirmation of the significance of the cost function C_{time} that we used to drive the optimization process and confirm the impact that memory conflicts have on the actual execution time of the synthesized kernel.

6. RELATED WORK

The application of integer lattices to memory allocation problems is addressed by Darte et al. [2005], who propose a methodology for reducing the memory needed for the execution of an algorithm by analyzing variable liveness. The authors establish a correspondence between an integer lattice and a modular mapping of the array indices. As an example, the lattice spanned by the basis $\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$ may equivalently be expressed by

the two-dimensional mapping $(b_0, b_1) = (j \bmod 2, i \bmod 3)$, where the pair (b_0, b_1) , with $0 \leq b_0 < 2$ and $0 \leq b_1 < 3$, is used as a bidimensional bank index. Although we base our work on a similar mathematical framework and use some of their results, such as the enumeration of the solutions, they solve a different problem—memory reuse. In particular, in our work, the determinant of the lattice corresponds to the number of memory banks, and the objective function is chosen to minimize the number of conflicts on memory ports. On the other hand, in their work, the number of memory locations required by an optimum modular mapping is equal to the determinant of the underlying lattice. Furthermore, they consider indices as conflicting when they are simultaneously live under a given schedule, whereas we have a conflict when there is an access to the same memory bank by simultaneous executions of loop iterations.

The development of mathematical methodologies for the problem of partitioning was also studied extensively for optimizing locally sequential globally parallel (LSGP) computations. Darte [1991] aims to find valid boxes (i.e., parallelepipeds of points) for systolic arrays. Their solution space consists of all left Hermite forms of the so-called activity basis that identifies the computations simultaneously active. In Darte [1991], integer points represent computations that may be mapped or not onto the same processor and not memory locations to map onto specific banks. A similar problem is analyzed in Darte et al. [2002], providing a closed form for enumerating all schedules on a cluster of VLIW processors and constructing a linear function that schedules precisely one iteration per cycle of a loop nest. Memory partitioning for LSGP systems has been also studied by Gupta [1992], Chatterjee et al. [1995], and Verdoolaege et al. [2007] to reduce the communication among processors.

Recently, mathematical optimization techniques have targeted reconfigurable devices provided with multiple independent fine-grained memory blocks. In this scenario, minimizing the number of conflicts on the distributed memory banks is of paramount importance for performance. A few recent contributions [Cong et al. 2011; Li et al. 2012b; Wang et al. 2013, 2014] are concerned with partitioning of on-chip memory. In Cong et al. [2011], an automated memory partitioning algorithm is proposed to support multiple simultaneous affine memory references to the same array. In Li et al. [2012b], memory accesses in different loop iterations are partitioned across different memory banks and scheduled in the same cycle, minimizing the number of required banks. In the preceding works, a multidimensional array is first flattened into a single-dimensional array before partitioning. Since memory addresses after flattening depend on the array size, different partitioning schemes are generated for different array sizes, many of which are suboptimal. An improved approach is proposed in Wang et al. [2013]. They model memory ports as n -dimensional hyperplanes. However, by limiting their solutions to a single family of hyperplanes, they might ignore some potential solutions. Unlike the work of Wang et al. [2013], which uses a single family of hyperplanes to express bank mapping, we essentially exploit a number of hyperplanes equal to the dimensionality of the array to be partitioned. Furthermore, the work of Wang et al. [2013] does not express parallel datasets formally. Hence, the methodology is valid only for the specific problem of avoiding memory conflicts when loop pipelining techniques are applied. Our technique, on the other hand, explicitly models parallel iterations by means of Z-polyhedral techniques and generalizes the solution space by adopting lattices instead of hyperplanes—that is, a strict superset of the solution space in Wang et al. [2013]. Wang et al. [2014] extend their previous work by introducing block-cyclic partitioning but still relying only on a single family of hyperplanes. The work in Liu et al. [2009] introduces a geometric programming framework combining data reuse with data-level parallelism. Although the problem is still related to data-level parallelization, they assume that each processing element is connected to a single memory and replicate data when needed. This assumption largely simplifies the problem but

Fig. 7. Hyperplane-based memory partitioning. (a) Memory bank mapping. (b) Hardware datapath inferred from memory bank mapping. The symbol MB_i in the figures denotes the i th bank.

neglects the possibility of avoiding the replication of data, possibly incurring memory waste and a potential loss of available memory ports. Similar to our work, Chen and Postula [2000] use integer lattices in the context of memory partitioning to improve memory bandwidth by using different physical storage blocks. The work explores different periodic partitioning strategies by varying both the number of banks and the system clock frequency. Their main contribution is related to address generation given a specific partitioning strategy. However, their solution space exploration does not resort to any mathematical technique to model the code and its memory access patterns. As a consequence, the work is based on evaluating the schedule length and the system cost for each possible solution by adopting scheduling algorithms, such as list scheduling.

There are other approaches concerned with optimizing on-chip memory usage, although focused on orthogonal problems. The work in Lu et al. [2009] develops a compile-time framework for data locality optimization via data layout transformation targeting NUCA chip multiprocessors. Liu et al. [2007] use polyhedral abstractions to reduce the occupied on-chip scratchpad memory. The approach is useful when dealing with algorithms working with a great deal of data. Unlike our approach, it aims to improve data reuse instead of maximizing the available on-chip memory bandwidth. Differently, Pouchet et al. [2013] introduce a framework in the context of HLS for transforming loop nests to maximize on-chip memory reuse and minimize accesses to off-chip memory blocks. A different problem is tackled in the work of Bayliss and Constantinides [2012], which proposes a technique to embed some physical features related to external SDRAMs in the iteration domains of code statements to minimize row activations and maximize reuse. It addresses external memory modules, and in that respect, it is complementary to our work, which is focused on on-chip memory. The work of Alias et al. [2013] optimizes remote accesses for offloaded kernels on reconfigurable platforms.

6.1. Comparisons with the Hyperplane-Based Approach

As summarized previously, the work of Wang et al. [2013, 2014] provides current state-of-the-art solutions for memory partitioning in the context of HLS, limited to solutions based on a single family of hyperplanes. In that respect, lattices represent an extension of previous approaches. In fact, partitioning solutions based on one single family of hyperplanes (e.g., $(j+i) \bmod 2$) can always be represented as lattices (e.g., $\begin{pmatrix} 2 & 0 \\ 1 & 1 \end{pmatrix}$). In some cases, hyperplane-based partitioning may achieve the same level of access parallelism as lattice-based partitioning, but yielding a more complex steering circuitry. For instance, the memory in the example of Figure 1(a) can be partitioned as shown in Figure 7(a) by using hyperplanes, fully exploiting the available four memory banks. However, since the access patterns to the different banks are not always the same in each dataset, each input of each adder needs to receive the operands from two different memory banks depending on the specific value of the iterators, complicating the

Table I. Synthesis Results

Benchmark	NB/ BRAMs	Lattices		Hyperplanes	
		LUTs	Statements (#)	LUTs	Statements (#)
Image Resize kernel (OSF = 2, ISF = 2)	2	536	4	536	4
	4	536	4	536	4
	8	592	4	1,024	8
	16	526	4	1,136	8
Image Resize kernel (OSF = 4, ISF = 4)	2	2,130	16	2,130	16
	4	2,130	16	2,130	16
	8	2,212	16	2,212	16
	16	2,212	16	2,580	32
2D-Seidel kernel (OSF = 2, ISF = 2)	2	2,745	4	2,745	4
	4	2,526	4	2,526	8
	8	5,584	8	7,387	16
	16	8,178	16	11,254	32
2D-Seidel kernel (OSF = 4, ISF = 4)	2	10,384	16	10,348	16
	4	10,339	16	10,339	16
	8	10,381	16	17,516	32
	16	10,202	16	11,290	32
2D-Jacobi kernel (OSF = 2, ISF = 2)	2	1,862	4	1,862	4
	4	2,118	4	3,409	8
	8	4,007	8	5,753	16
	16	6,419	16	9,129	16
2D-Jacobi kernel (OSF = 4, ISF = 4)	2	4,893	16	4,893	16
	4	5,387	16	5,387	16
	8	5,049	16	11,369	32
	16	4,878	16	16,978	32

OSF, outer stripmining factor; ISF, inner stripmining factor.

steering logic of the datapath, as well as causing potential increases in area, clock period, and power consumption. Notice that because of the commutativity of addition, this logic overhead may be optimized out in this specific case. However, this is not true in general. On the other hand, covering a strict superset of hyperplane-based partitioning, lattice-based partitioning allows us to identify the more efficient solution shown in Figure 1(c), where in addition to the fully parallelizable read operations, the access patterns to the banks are the same for each dataset, inherently simplifying the steering logic. To confirm the preceding considerations, Table I shows the area occupation measured in look-up tables (LUTs) on a Xilinx Virtex-7 FPGA for three different benchmarks. The considered benchmarks are typical HLS applications with loop nests having a high degree of parallelism. In particular, the first benchmark is an ordinary image resize kernel using bilinear interpolation. The remaining two benchmarks perform different stencil computations—a 2D Jacobi kernel and a 2D Gauss-Seidel kernel e.g., see Leopold [2002] for the details of these algorithms). The kernels were coded in plain C and synthesized by means of the Vivado HLS tool [Xilinx Inc. 2012]. For each benchmark, we varied both the stripmining factors of the two innermost loops and the number of memory banks, here coinciding with BlockRAM components (BRAMs), normally available on Xilinx FPGAs. The table reports the number of LUTs used by the two techniques for the respective optimal solutions. Because of the spatial regularity of lattices, our technique achieves more compact datapaths having a simplified steering logic. These improvements are also reflected in the code complexity. In fact, Table I also contains the overall number of statements present in the final code. As shown by the

Fig. 8. (a) Code snippet and related datasets on a lattice-based partitioning solution. (b) Possible hyperplane-based solutions.

table, for the chosen benchmarks, our method also helps to keep the code complexity reasonably low.

In addition to a more complex steering logic, there may be situations where the work of Wang et al. [2013] requires more memory banks than strictly necessary to achieve full parallelization, particularly when the kernel has a complex control flow. As an example, consider the code in Figure 8. The kernel contains a normalized perfect loop nest with three statements. There is a statically predictable branch, so the nest is still an affine SCoP. Loop iterations access a variable shape 2×2 window. In particular, Figure 8(a) shows the parallel datasets for $(i = 1, j = 1)$, $(i = 1, j = 2)$, and $(i = 1, j = 3)$. Each of them contains four points. If four different memory banks are considered, the only possible solution to not have conflicts is showed in part (a) of the figure, corresponding to the fundamental lattice $\mathcal{L}(\begin{smallmatrix} 2 & 0 \\ 0 & 2 \end{smallmatrix})$. This solution cannot be expressed as a set of translate hyperplanes. Figure 8(b) shows all possible hyperplane-based solutions. The green line represents the hyperplane causing the conflict. It can be easily recognized that four memory banks are not sufficient here to avoid conflicts completely, and the optimal solution identified by the lattice-based partitioning technique is missed.

7. CONCLUSIONS AND FUTURE DEVELOPMENTS

This work addresses the problem of automated memory partitioning for emerging architectures, such as reconfigurable hardware platforms, providing the opportunity of customizing the memory architecture based on the application access patterns. Targeted at affine SCoPs, the technique exploits the Z-polyhedral model for program analysis, yielding a powerful and elegant formalism capturing both the problem of bank mapping and storage minimization. In particular, the approach was based on integer lattices, enabling us to generate a solution space for the bank mapping problem, which includes previous results as particular cases. The problem of storage minimization, on the other hand, was tackled by an optimal approach ensuring asymptotically zero memory waste, or as an alternative, an efficient approach ensuring arbitrarily small waste. The theoretical results were also demonstrated through a prototype toolchain and a detailed step-by-step case study described in the article, along with some comparisons with different approaches found in the technical literature.

The work also opens up a range of further investigation paths. First of all, as pointed out by our case study, there may be different lattices all achieving the minimum number

of conflicts for a given number of banks. However, they might not be equivalent in terms of the generated code, which may cause different delays and possibly area results in those computing platforms where the code is directly translated to hardware, such as FPGAs. Analyzing these effects systematically is part of our future work. Furthermore, although the search space is likely to be limited for practical problems, we will also explore the adoption of ad hoc heuristics to explore it more efficiently. In particular, from a purely mathematical point of view, there are situations where given a certain determinant, the solution space of the lattice-based partitioning technique collapses to one single family of hyperplanes. Although this happens for low dimensionalities of the array and very low numbers of memory banks, a precise formalization would help to reduce the solution space in such cases. A further possibility of improvement concerns the storage minimization scheme. Although it is asymptotically optimal in terms of memory waste, it does not include liveness analysis of memory locations, unlike that in the work of Darté et al. [2005]. We thus plan to extend our methodology with per-bank liveness analysis. In addition, like most similar works, the approach in this article is focused on partitioning a single array in memory. When different arrays are concurrently accessed in the same kernel, they may be processed separately, or alternatively, they may be seen as parts of a single memory space. These choices might variously impact performance or result in additional opportunities for optimization, leaving room for further developments in this direction. Last, instead of making the partitioning explicit in the code, such as by using different array names, which leads to a static assignment of banks, a different possibility would be to insert ad hoc hardware components that route the memory requests to the corresponding banks by computing the mapping dynamically. This possibility was not explored here, essentially because a static code-level solution can be easily automated and does not interfere with the HLS process in itself. However, its implementation would transfer the complexity of the approach from the software to the underlying hardware architecture. We thus consider the automated generation of such hardware memory access managers as a potential future development of this work.

REFERENCES

- Christophe Alias, Alain Darté, and Alexandru Plesco. 2013. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *Proceedings of the Conference on Design, Automation, and Test in Europe*. EDA Consortium, San Jose, CA, 575–580. <http://dl.acm.org/citation.cfm?id=2485288.2485430>.
- Alexander Barvinok. 2002. *A Course in Convexity*. American Mathematical Society.
- Cédric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, Los Alamitos, CA, 7–16.
- Samuel Bayliss and George A. Constantinides. 2012. Optimizing SDRAM bandwidth for custom FPGA loop accelerators. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, New York, NY, 195–204.
- Uday Bondhugula, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2007. *PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer*. Technical Report OSU-CISRC-10/07-TR70. Ohio State University, Columbus, OH.
- Andr R. Brodtkorb, Trond R. Hagen, and Martin L. Stra. 2013. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing* 73, 1, 4–13.
- Siddhartha Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, and Shang-Hua Teng. 1995. Generating local addresses and communication sets for data-parallel programs. *ACM SIGPLAN Notices* 28, 7, 149–158.
- Song Chen and Adam Postula. 2000. Synthesis of custom interleaved memory systems. *IEEE Transactions on VLSI Systems* 8, 1, 74–83.
- Alessandro Ciarlo, Luca Gallo, and Nicola Mazzocca. 2013. Design space exploration for high-level synthesis of multi-threaded applications. *Journal of Systems Architecture* 59, 10, 1171–1183.

- Albert Cohen, Sylvain Girbal, and Olivier Temam. 2004. A polyhedral approach to ease the composition of program transformations. In *Euro-Par. Lecture Notes in Computer Science*, Vol. 3149. Springer, 292–303.
- Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. 2011. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems* 16, 2, Article No. 15.
- Ethan E. Danahy, Sos S. Agaian, and Karen A. Panetta. 2007. Algorithms for the resizing of binary and grayscale images using a logical transform. In *Image Processing: Algorithms and Systems V* SPIE Proceedings, Vol. 6497. SPIE, 64970.
- Alain Darte. 1991. Regular partitioning for synthesizing fixed-size systolic arrays. *Integration* 12, 3, 293–304.
- Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frederic Vivien. 2002. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems* 7, 1, 159–172.
- Alain Darte, Robert Schreiber, and Gilles Villard. 2005. Lattice-based memory allocation. *IEEE Transactions on Computing* 54, 10, 1242–1257.
- Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming* 21, 6, 389–420.
- Martin Griehl and Christian Lengauer. 1996. The loop parallelizer LooPo—announcement. In *Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science*, Vol. 1239. Springer, 603–604.
- Gautam Gupta and Sanjay Rajopadhye. 2007. The Z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, 237–248.
- Manish Gupta. 1992. *Automatic Data Partitioning on Distributed Memory Multicomputers*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL.
- Guillaume Iooss and Sanjay Rajopadhye. 2012. A library to manipulate Z-polyhedra in image representation. In *Proceedings of IMPACT 2012*.
- Jonathan Kelner. 2009. Lecture 18, An Algorithmists Toolkit.
- Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, San Francisco, CA.
- H. Le Verge. 1995. Recurrences on lattice polyhedra and their applications. Based on a manuscript written by H. Le Verge.
- Claudia Leopold. 2002. On optimal temporal locality of stencil codes. In *Proceedings of the 2002 ACM Symposium on Applied Computing*. ACM, New York, NY, 948–952. DOI : <http://dx.doi.org/10.1145/508791.508975>
- Jia-Jhe Li, Chi-Bang Kuan, Tung-Yu Wu, and Jenq Kuen Lee. 2012a. Enabling an OpenCL compiler for embedded multicore DSP systems. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*. IEEE, Los Alamitos, CA, 545–552.
- Peng Li, Yuxin Wang, Peng Zhang, Guojie Luo, Tao Wang, and Jason Cong. 2012b. Memory partitioning and scheduling co-optimization in behavioral synthesis. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, New York, NY, 488–495.
- Qiang Liu, George A. Constantinides, Konstantinos Masselos, and Peter Y. Cheung. 2007. Automatic on-chip memory minimization for data reuse. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*. 251–260.
- Qiang Liu, George A. Constantinides, Konstantinos Masselos, and Peter Y. K. Cheung. 2009. Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: A geometric programming framework. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 3, 305–315.
- Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra.
- Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, Ponnuswamy Sadayappan, Yongjian Chen, Haibo Lin, and Tin-Fook Ngai. 2009. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 348–357.
- Benot Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. R-Stream compiler. In *Encyclopedia of Parallel Computing*. Springer, 1756–1765.
- Morris Newman. 1972. *Integral Matrices*. Pure and Applied Mathematics, Vol. 45. Academic Press.
- Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, New York, NY, 29–38.

- Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. 1996. *On Manipulating Z-Polyhedra*. Technical Report.
- Alexander Schrijver. 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, NY.
- Rachid Seghir. 2012. ZPolyTrans: A library for computing and enumerating integer transformations of Z-polyhedra. In *Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. 6.
- Jürgen Teich and Lothar Thiele. 1993. Partitioning of processor arrays: A piecewise regular approach. *Integration: The VLSI Journal* 14, 3, 297–332.
- Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. 2007. pn: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems* 2007, 1, 19.
- Sven Verdoolaege and Kevin M. Woods. 2008. Counting with rational generating functions. *Journal of Symbolic Computation* 43, 2, 75–91.
- Yuxin Wang, Peng Li, and Jason Cong. 2014. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, 199–208.
- Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. 2013. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, New York, NY, Article No. 12.
- Eric W. Weisstein. 2003. *CRC Concise Encyclopedia of Mathematics*. CRC Press, Boca Raton, FL.
- Xilinx Inc. 2012. *Vivado Design Suite User Guide: High-Level Synthesis*. Available at <http://www.xilinx.com>.

Received March 2014; revised August 2014; accepted October 2014