Affine Multibanking for High-Level Synthesis

Ilham Lasfar, Christophe Alias, Matthieu Moy, Rémy Neveu, Alexis Carré

To cite this version:

Ilham Lasfar, Christophe Alias, Matthieu Moy, Rémy Neveu, Alexis Carré. Affine Multibanking for High-Level Synthesis. [Research Report] RR-9440, Inria - Research Centre Grenoble – Rhône-Alpes. 2021. hal-03481328

HAL Id: hal-03481328
https://inria.hal.science/hal-03481328v1

Submitted on 15 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Affine Multibanking for High-Level Synthesis

Ilham Lasfar, Christophe Alias, Matthieu Moy, Rémy Neveu, Alexis Carré

ISSN 0249-6399 ISRN INRIA/RR--9440--FR+ENG

RESEARCH REPORT
N° 9440
December 2021
Project-Team Cash

Affine Multibanking for High-Level Synthesis

Ilham Lasfar, Christophe Alias, Matthieu Moy, Rémy Neveu,
Alexis Carré*

Project-Team Cash

Research Report n° 9440 — December 2021 — 13 pages

Abstract: In the last decade, FPGAs appeared as a credible alternative for big data and high-performance computing applications. However, programming an FPGA is tedious: given a function to implement, the circuit configuration must be built from scratch by the developer. Hence the emergence of high-level circuit compilers (high-level synthesis, HLS), able to translate a C program to an FPGA circuit configuration. Unlike software parallelization, there is no operating system to place the computation and the memory at runtime. All the parallelization decisions must be done at compile time. In this report, we address the compilation of data placement under parallelism and resource constraints. We propose an HLS algorithm able to partition the data across memory banks, so parallel access will target distinct banks to avoid data transfer serialization. Our algorithm is able to minimize the number of banks and the maximal bank size.

Key-words: High-Level Synthesis, FPGA, data restructuring, polyhedral model, multibanking

* CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Multibanking affine pour la synthèse de haut-niveau

Résumé : Dans la dernière décennie, les FPGA sont apparus comme une alternative crédible pour le big data et le calcul haute performance. Malheureusement, la programmation des FPGA requiert la conception d'une configuration de circuit, ce qui est hors d'atteinte pour un programmeur. D'où l'émergence de la synthèse de circuit haut-niveau (High-Level Synthesis, HLS), capable de compiler un code C canonique en configuration de circuit FPGA. Contrairement à la parallélisation logicielle, toutes les décisions de placement des données et des calculs doivent être prises statiquement, à la compilation. Dans ce rapport, nous étudions la compilation d'un placement de données sous contrainte de parallélisme et de taille. Nous proposons un algorithme de HLS capable de partitionner les données sur des bancs, de sorte que deux données accédées en même temps soient sur deux bancs différents, et puissent ainsi être accessibles en parallèle. Notre algorithme est capable de trouver une partition qui minimise le nombre de bancs et la taille maximale d'un banc.

Mots-clés : Synthèse haut-niveau, FPGA, restructuration des données, modèle polyédrique, multibanking

1 Introduction

Since the end of Dennard scaling, the energy efficiency (flop/J) of computers has become a major challenge as soon as the energy budget is limited. This particularly applies to embedded systems and high-performance computers. The best solution is to rely on *specialized circuits*, which ultimately trade energy efficiency for programmability. In the last decade, FPGAs appeared as a credible alternative for big data and high-performance computing applications. However, programming an FPGA is tedious: given a function to implement, the circuit configuration must be built *from scratch* by the developer. Hence the emergence of high-level circuit compilers (*high-level synthesis*, HLS) [1, 6, 13, 14, 20], able to translate a C program to an FPGA circuit configuration. Unlike software parallelisation, there is no parallel runtime to place the computation and the data among processing elements: all the parallelization decisions must be taken at compile-time. High-level synthesis tools are now quite efficient for generating finite-state machines, for exploiting instruction-level parallelism, operator selection, resource sharing, and even for performing some form of software pipelining, for one given kernel. In other words, it is acceptable to rely on them for optimizing the heart of accelerators, i.e., the equivalent of back-end optimizations in standard (software) compilers. However, HLS tools lack high-level parallelization algorithms, like those of the polyhedral model [5], hence the need for *source-to-source* code optimizers, using the HLS tools as a back-end [4].

In this report, we propose a source-to-source transformation to address the *multibanking problem*: data are mapped to distinct memories (*banks*) so parallel access will target distinct banks to avoid data transfer serialization. Given a program and a schedule prescribing parallelism, we are able to infer a complete reorganization of the data into banks and to generate the transformed program accordingly. This problem has been investigated for simple kernels with perfect loop nests [7, 21, 22], usually on convolution-like kernels with differents patterns. We propose a general HLS algorithm which subsumes these approaches and makes the following contributions:

- We propose a novel and *general formalization of the multibanking problem*, which subsumes the previous approaches.
- We propose a *complete algorithm* to compute our multibanking transformation using the polyhedral model.
- Our approach *minimizes the number of banks and the maximal bank size*, without hindering parallel accesses.

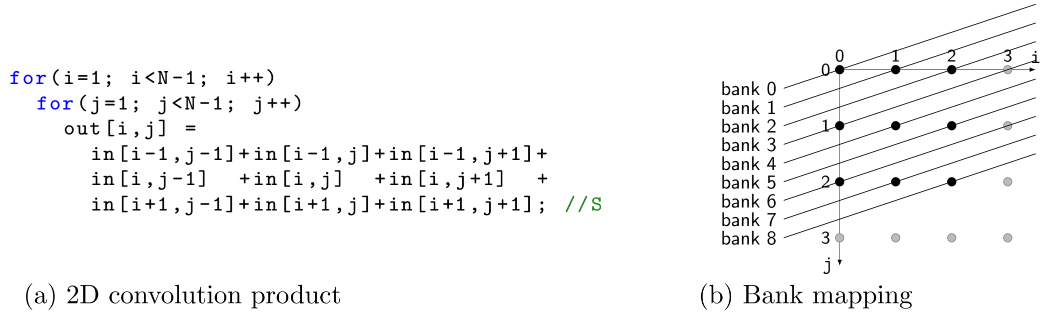
This report is structured as follows. Section 2 introduces the problem of multibanking and the polyhedral model. Section 3 presents the related work. Section 4 presents our algorithm for multibanking. Section 5 presents our experimental results. Finally, Section 6 concludes this report and outlines future work.

2 Preliminaries

This section introduces the multibanking problem and the polyhedral model, the compilation framework used by our multibanking approach.

2.1 Multibanking

The main advantage of circuit programming is to tune finely large scale of parallelism while optimizing energy consumption. Unlike software parallelization, there is no parallel runtime, nor



(a) 2D convolution product (b) Bank mapping

Figure 1: A simple example

hardware to rule computation/data placement and communications. Almost everything must be orchestrated at compile time. In particular, parallelism must come with efficient memory accesses, which enable parallel reads/writes as would do a cache coherence protocol. Hence the idea of *multibanking*: data are mapped to distinct memories (*banks*) so that parallel accesses will target distinct banks to avoid data transfer serialization. We illustrate the problem on the 2d convolution product, illustrated in Figure 1.(a). The loop is assumed to be executed in sequence, then array accesses are done in parallel.

Because of memory limitation, parallel references must be mapped to different banks. A solution depicted on Figure 1.(b) is to put each reference $in(i, j)$ into memory bank $\text{BANK}_{in}(i, j) = 3i + j \mod 9$ at offset $\text{OFFSET}_{in}(i, j) = i \mod N$. In general, we seek affine mappings $\text{BANK}_a : \vec{i} \mapsto \phi_a(\vec{i}) \mod \sigma(\vec{N})$ and $\text{OFFSET}_a : \vec{i} \mapsto \psi_a(\vec{i}) \mod \tau(\vec{N})$ for each array a where ϕ_a, ψ_a, σ and τ are affine functions. Note that those reallocation functions map *different arrays* to a *common array memory* organized with *outer bank dimensions* and *inner offset dimensions*. The size of that common memory is $\Pi_d \sigma^d(\vec{N}) \times \Pi_d \tau^d(\vec{N})$, where $\sigma^d(\vec{N})$ denotes the d-th dimension of vector $\sigma(\vec{N})$.

In this paper, we present a general algorithm to compute these functions such that parallel accesses are granted while minimizing the number of memory banks and their size. We use the formalism and the building blocks of the *polyhedral model* to express our algorithm.

2.2 Polyhedral model

The polyhedral model [8, 9, 11, 17, 18] is a general framework to design loop transformations, historically geared towards source-level automatic parallelization [11] and data locality improvement [5]. It abstracts loop iterations as a union of convex polyhedra – hence the name – and data accesses as affine functions. This way, precise – iteration-level – compiler algorithms may be designed (dependence analysis [8], scheduling [9] or loop tiling [5] to quote a few) . The polyhedral model manipulates program fragments consisting of nested `for` loops and conditionals manipulating arrays and scalar variables, such that loop bounds, conditions, and array access functions are *affine expressions* of surrounding loops counters and structure parameters (input sizes, e.g., N)). Thus, the control is static and may be analysed at compile-time. With polyhedral programs, each iteration of a loop nest is uniquely represented by the vector of enclosing loop counters $\vec{i}$. The execution of a program statement S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i} \rangle$ and is called an *operation* or an *execution instance*. The set $\mathcal{D}_S$ of iteration vectors is called the *iteration domain* of S. On Figure 1.(a) we have a single statement S with the iteration domain $\mathcal{D}_S = \{(i, j) \mid 1 \leq i, j < N - 1\}$.

Scheduling A *schedule* θ_S assigns each operation $\langle S, \vec{i} \rangle$ with a timestamp $\theta_S(\vec{i}) \in (\mathbb{Z}^d, \ll)$. Intuitively, $\theta_S(\vec{i})$ is the iteration of $\langle S, \vec{i} \rangle$ in the transformed program. A schedule is *correct* if $\langle S, \vec{i} \rangle \rightarrow \langle T, \vec{j} \rangle \Rightarrow \theta_S(\vec{i}) \ll \theta_T(\vec{j})$, where $\rightarrow$ denotes the *dependence relation* between operations. The lexicographic order ensures that the dependence is preserved. A schedule prescribes an execution order and a parallelism between operations. Two operations $\langle S, \vec{i} \rangle$ and $\langle T, \vec{j} \rangle$ are executed in parallel iff $\theta_S(\vec{i}) = \theta_T(\vec{j})$. All the accesses (read, write) for both operations are executed in parallel. If d_1 and d_2 are such accesses, we write $d_1 \parallel_\theta d_2$. Back to our example, assuming $\theta_S(i, j) = (i, j)$, the loop nest is simply executed sequentially, and the parallel accesses are those of a single operation $\langle S, i, j \rangle$. For instance $a[i-1, j-1] \parallel_\theta a[i-1, j]$.

Data reuse & array liveness A schedule prescribes an execution order, hence a repartition of liveness intervals for each data element. With polyhedral programs, we focus on arrays, which contain most of the data when dealing with data/compute intensive loop kernels. Two array cells $a(\vec{i})$ and $b(\vec{j})$ are in conflict iff there liveness intervals intersect. Formally, iff there exists operations writing $a(\vec{i})$ (W_a) and writing $b(\vec{j})$ (W_b) and operations reading $a(\vec{i})$ (R_a) and reading $b(\vec{j})$ (R_b) such that the intervals $[W_a, R_a]$ and $[W_b, R_b]$ intersect:

$$\theta(W_a) \ll \theta(R_b) \quad \text{and} \quad \theta(W_b) \ll \theta(R_a)$$

In that case, we write $a(\vec{i}) \bowtie_\theta b(\vec{j})$. $\bowtie_\theta$ is called the *conflict relation.*

3 Related work

This section presents different existing approaches to optimize memory bank allocation, with an emphasis on FPGA.

3.1 Memory partitioning

Cong et al. [7] propose a formulation of the problem of memory bandwidth limit applied to FPGAs. They propose a first solution for memory partitioning, where memory is divided into several blocks that are distributed into BRAMs in a cyclic manner. The approach is limited in several ways: it is restricted to perfect loop nests, and can only deal with a single array. Furthermore, the algorithm proposed does not always find a partitioning that allows an optimal scheduling. It can be necessary to modify the schedule to get an acceptable solution.

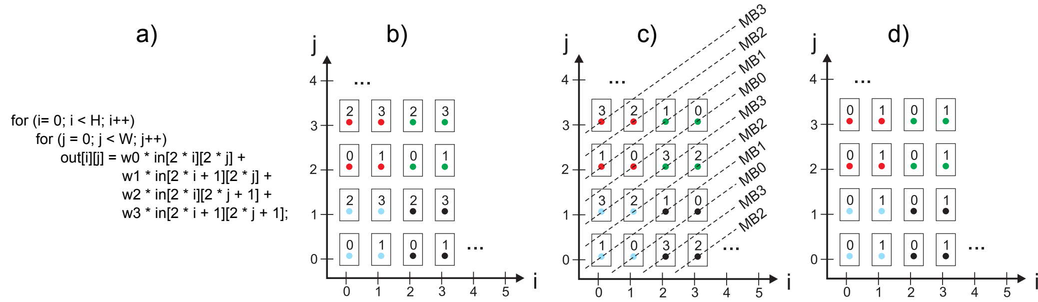
Wang et al. [22] reuse the results from [7] to propose a code transformation that allows an optimal schedule in the case of perfect loop nests. An important difference with the previous approach is the fact that programs using several arrays are properly handled. The number of memory banks is also optimized thanks to a memory merging algorithm, that reduces the number of memory banks without sacrificing performance.

Wang et al. [21] extend the work from [7]. The underlying framework is the same, but the solution provides a finer partitioning. The algorithm consists in finding the hyperplanes along which the array is to be partitioned. The obtained partitioning allows respecting the target schedule. Data in different banks are addressed using Ehrhart points counting, which leads to a complex address management logic. A heuristic is proposed to reduce this overhead using padding. While the heuristic is good for low-dimension array, the $d!$ complexity (d being the dimension of the array) makes it harder to apply for high-dimensionality arrays.

Gallo et al. extend the work from Wang and Cong in [12]. They show that solutions can be represented using affine integer latices. Cyclic partitioning and hyperplanes-based ones are included in this representation. An important point in the article is the discussion on the impact

of the area required by bank switching logic to implement banks. Hyperplane-based solutions can respect the optimal schedule, but it is possible to express other solutions respecting this schedule using less logic to deal with memory addressing.

Figure 2: Partitioning methods for the algorithm in a) : b) lattice-based solution, c) hyperplanes-based solution, d) cyclic solution [12]



In [23], Yin et al. extend the work from [21] in a direction different from Gallo et al. They improve the efficiency of the computation of hyperplanes and data addressing. This study extends the previous ones by allowing multiple array accesses in statements. An important point is bank merging: banks are merged to according to a criterion minimizing the logic.

3.2 Data reuse

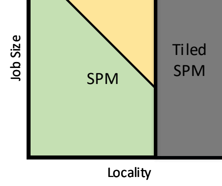
An approach complementary to partitioning is to reduce communication. A schedule is inefficient when it is too sequential, but it is also the case when the program makes numerous accesses to the slow memory. In this context, [16] proposes, within the polyhedral framework, an algorithm to find transformations that improve memory locality without reducing the parallelism of a program. The key idea is *loop tiling*. The work follows the ones from Alias, Darte and Plesco [4].

3.3 Cache partitioning

In [15], Jones et Hanna propose a cache model for FPGAs. They work on the dependency graph. In this graph, sets of memory accesses called *regions* are identified. Statements in different regions are independent, or indirectly dependent. Each region is then mapped to a cache. Regions can therefore be executed in parallel efficiently. Regions are combined in order to find an optimal combination with respect to circuit area and an estimation of the number of cache misses. The implementation of this approach with an HLS tool such as Vivado HLS is not possible. No currently available industrial HLS tool support the notion of cache.

The approach from Rogers et al. [19] compares the efficiency of cache compared to block-based memory architectures (scratchpad memories, SPM). On small sized computations or with bad memory locality, usage of SPM is more efficient. On larger computations with good memory locality, a cache memory is preferable. This is no longer true when the computation can be tiled: using an SPM is then better. The article presents a tool to define automatically the most efficient architecture.

Figure 3: Representation of optimal memory architecture for different problems [19]



4 Our multibanking algorithm

This section presents our multibanking algorithm. We first present the intuitions behind the derivation of the bank mapping 4.1 and the offset mapping 4.2. Then, we present our algorithm 4.3. Finally we illustrate our algorithm on an example 4.4.

4.1 Deriving the bank mapping

We first present the polyhedral formulation to obtain a correct bank mapping. Then we give a general formulation to minimise the number of banks.

Finding a correct bank mapping A bank mapping $\text{BANK}_a(\vec{i}) = \phi_a(\vec{i}) \mod \sigma(\vec{N})$ is correct w.r.t. the parallel execution prescribed by a schedule θ iff two different memory cells $a(\vec{i})$ and $a(\vec{j})$ accessed *at the same time* belong to *different banks*:

$$a(\vec{i}) \parallel_\theta b(\vec{j}) \wedge \vec{i} \neq \vec{j} \Rightarrow \text{BANK}_a(\vec{i}) \neq \text{BANK}_b(\vec{j})$$

We relax the formulation to postpone the computation of the modulo:

$$a(\vec{i}) \parallel_\theta b(\vec{j}) \wedge \vec{i} \ll \vec{j} \Rightarrow \phi_a(\vec{i}) \ll \phi_b(\vec{j})$$

$\phi_a(\vec{i}) \ll \phi_b(\vec{j})$ means that there exists a dimension d such that $\phi_a^d(\vec{i}) < \phi_b^d(\vec{j})$ and both vectors are identical above: $\phi_a^\ell(\vec{i}) = \phi_b^\ell(\vec{j})$ for all $\ell < d$. Hence, the dimensions of ϕ might be computed incrementally across dimensions d with the formulation:

$$a(\vec{i}) \parallel_\theta b(\vec{j}) \wedge \vec{i} \ll \vec{j} \Rightarrow \phi_a^d(\vec{i}) \leq \phi_b^d(\vec{j}) \quad (1)$$

Once a dimension ϕ^d is found, we focus on the unresolved parallel conflicts (still in the same bank):

$$\parallel_\theta := \parallel_\theta \cap \{(a(\vec{i}), b(\vec{j})) \mid \phi_a^d(\vec{i}) = \phi_b^d(\vec{j})\}$$

And we iterate on the next dimension until all conflicts are resolved: $\parallel_\theta \neq \emptyset$. This is the basic idea of Algorithm 1.

Minimizing the bank number The number of banks might be estimated as the maximum difference $\phi_b^d(\vec{j}) - \phi_a^d(\vec{i})$ for conflicting cells $a(\vec{i})$ and $b(\vec{j})$. It gives the modulo value σ^d, along dimension d:

$$a(\vec{i}) \parallel_\theta b(\vec{j}) \wedge \vec{i} \ll \vec{j} \Rightarrow \phi_b^d(\vec{j}) - \phi_a^d(\vec{i}) \leq \sigma^d(\vec{N}) \quad (2)$$

The coefficient of the affine form σ^d might be minimized lexicographically, under the constraints of correctness (Eq. 1) and efficiency (Eq. 2). Iterating the process for each dimension yield a general, correct and efficient bank mapping.

All these constraints might be turned to existentially guarded affine constraints thanks to the affine form of Farkas lemma, following the lines of [10], formalized as a domain-specific language in [2].

4.2 Deriving the offset mapping

Finding the offset in a bank might be achieved by the same algorithm, on different constraints. Again, we show the polyhedral formulation to compute a correct offset mapping. Then, we give a general formulation to minimize the bank size.

Finding a correct offset mapping An offset mapping $\textsc{offset}_a(\vec{i}) = \psi_a(\vec{i}) \mod \tau(\vec{N})$ is correct w.r.t. a schedule θ iff two array cells mapped to the same bank and whose *liveness conflict* are mapped to *different offsets*:

$$\textsc{bank}_a(\vec{i}) = \textsc{bank}_b(\vec{j}) \wedge a(\vec{i}) \bowtie_\theta b(\vec{j}) \wedge \vec{i} \neq \vec{j} \Rightarrow \textsc{offset}_a(\vec{i}) \neq \textsc{offset}_b(\vec{j})$$

Recall that $a(\vec{i}) \bowtie_\theta b(\vec{j})$ means that the liveness of $a(\vec{i})$ and $b(\vec{j})$ intersect, w.r.t. θ. This relation is computable, and might be expressed by an affine relation [3]. Similarly, we postpone the computation of the modulo, and we keep the following formulation so $\textsc{offset}$ might be computed incrementally across dimensions d:

$$\phi_a(\vec{i}) = \phi_b(\vec{j}) \wedge a(\vec{i}) \bowtie_\theta b(\vec{j}) \wedge \vec{i} \ll \vec{j} \Rightarrow \psi_a^d(\vec{i}) \leq \psi_b^d(\vec{j}) \quad (3)$$

When a dimension d is computed, we focus on unresolved constraints with:

$$\bowtie_\theta := \bowtie_\theta \cap \{(a(\vec{i}), b(\vec{j})) \mid \psi_a^d(\vec{i}) = \psi_b^d(\vec{j})\}$$

This is completely similar to the computation of the bank mapping.

Reducing the bank size We use exactly the same apparatus as for the bank mapping, this gives the modulo value $\tau^d(\vec{N})$, along dimension d:

$$\phi_a(\vec{i}) = \phi_b(\vec{j}) \wedge a(\vec{i}) \bowtie_\theta b(\vec{j}) \wedge \vec{i} \ll \vec{j} \Rightarrow \psi_b^d(\vec{j}) - \psi_a^d(\vec{i}) \leq \tau^d(\vec{N}) \quad (4)$$

Again, the coefficient of the affine form τ^d is be minimized lexicographically, under the constraints of correctness (Eq. 3) and efficiency (Eq. 4). Iterating the process for each dimension yield a general, correct and efficient offset mapping.

4.3 Our algorithm

We now present our algorithms to find the bank mapping and the offset mapping. Both algorithms apply the same computation pattern, and differ only with the constraints formulation. Hence, we adopt the following template formula for the correctness constraint:

$$\textsc{correct}(\mathcal{C}, \phi) : (a(\vec{i}), b(\vec{j})) \in \mathcal{C} \wedge \vec{i} \ll \vec{j} \Rightarrow \phi_a(\vec{i}) \leq \phi_b(\vec{j}) \quad (5)$$

and for the efficiency condition:

$$\textsc{efficient}(\mathcal{C}, \phi, \sigma) : (a(\vec{i}), b(\vec{j})) \in \mathcal{C} \wedge \vec{i} \ll \vec{j} \Rightarrow \phi_b(\vec{j}) - \phi_a(\vec{i}) \leq \sigma(\vec{N}) \quad (6)$$

These formula are assumed to be turned to a system of affine constraints using affine form of Farkas lemma, as mentioned above. The computation of the banking mapping is depicted on Algorithm 1. Then, the computation of the offset mapping is depicted on Algorithm 2.

Algorithm 1: FINDING THE BANK MAPPING

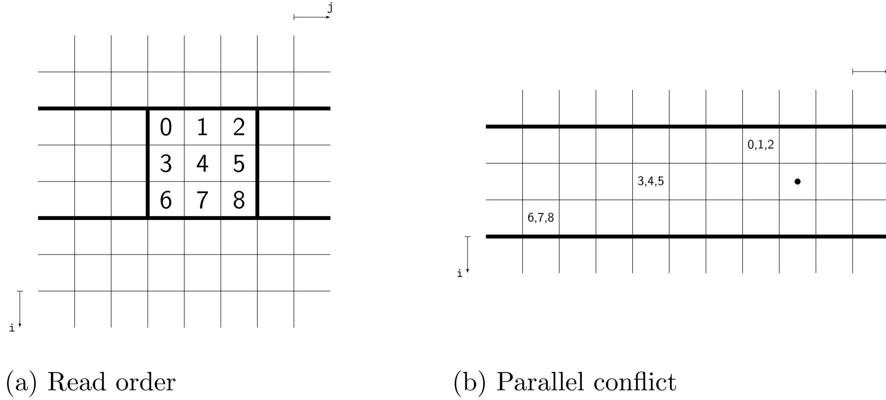
Data: Program (P, θ)
Result: Bank mapping $\textsc{bank}_a : (\vec{i}, \vec{N}) \mapsto \phi_a(\vec{i}) \mod \sigma(\vec{N})$, for each array a

1 **begin**
2 $\quad \mathcal{C} \leftarrow \{(a(\vec{i}), b(\vec{j})) \mid a(\vec{i}) \parallel_\theta b(\vec{j}) \wedge \vec{i} \ll \vec{j} \wedge \vec{i} \in \mathcal{D}_a \wedge \vec{j} \in \mathcal{D}_b\}$
3 $\quad d \leftarrow 0$
4 $\quad$ **while** $\mathcal{C} \neq \emptyset$ **do**
5 $\quad\quad \min_{\ll} \sigma^d$ coefficients s.t. $\textsc{correct}(\mathcal{C}, \phi^d) \wedge \textsc{efficient}(\mathcal{C}, \phi^d, \sigma^d) \wedge \phi^d \neq 0$
6 $\quad\quad \mathcal{C} \leftarrow \mathcal{C} \cap \{(a(\vec{i}), b(\vec{j})) \mid \phi_a^d(\vec{i}) = \phi_b^d(\vec{j})\}$
7 $\quad\quad d \leftarrow d + 1$
8 $\quad$ **end**
9 $\quad$ **return** BANK
10 **end**

Algorithm 2: FINDING THE OFFSET MAPPING

Data: Program (P, θ) and bank mapping $\textsc{bank}_a : (\vec{i}, \vec{N}) \mapsto \phi_a(\vec{i}) \mod \sigma(\vec{N})$ for each array a
Result: Offset mapping $\textsc{offset}_a : (\vec{i}, \vec{N}) \mapsto \psi_a(\vec{i}) \mod \tau(\vec{N})$, for each array a

1 **begin**
2 $\quad \mathcal{C} \leftarrow \{(a(\vec{i}), b(\vec{j})) \mid \phi_a(\vec{i}) = \phi_b(\vec{j}) \wedge a(\vec{i}) \bowtie_\theta b(\vec{j}) \wedge \vec{i} \ll \vec{j} \wedge \vec{i} \in \mathcal{D}_a \wedge \vec{j} \in \mathcal{D}_b\}$
3 $\quad d \leftarrow 0$
4 $\quad$ **while** $\mathcal{C} \neq \emptyset$ **do**
5 $\quad\quad \min_{\ll} \tau^d$ coefficients s.t. $\textsc{correct}(\mathcal{C}, \psi^d) \wedge \textsc{efficient}(\mathcal{C}, \psi^d, \tau^d) \wedge \psi^d \neq 0$
6 $\quad\quad \mathcal{C} \leftarrow \mathcal{C} \cap \{(a(\vec{i}), b(\vec{j})) \mid \psi_a^d(\vec{i}) = \psi_b^d(\vec{j})\}$
7 $\quad\quad d \leftarrow d + 1$
8 $\quad$ **end**
9 $\quad$ **return** OFFSET
10 **end**



(a) Read order (b) Parallel conflict

Figure 4: Pipelined convolution 2D

4.4 An example

We consider the 2D convolution kernel depicted on Figure 1. Unlike the introduction example, we assume the inner loop *for* j to be pipelined, and the array references to be read in the order depicted on Figure 4.(a): read R_0 (0) to read R_8 (8). The pipeline constraints might be expressed by the *read schedule*: $\theta_{R_0}(i,j) = (i,j)$, $\theta_{R_1}(i,j) = (i,j+1)$, ..., $\theta_{R_8}(i,j) = (i,j+8)$, the *computation schedule* $\theta_C(i,j) = (i,j+9)$, and the write schedule $\theta_C(i,j) = (i,j+10)$, from which the relations $\|_\theta$ and $\bowtie_\theta$ are derived. Figure 4.(b) depicts the parallel conflicts between the reads at the different pipeline stages executed at the same time: $\langle R_8, i, j\rangle, \langle R_7, i, j-1\rangle, \ldots, \langle R_0, i, j-8\rangle$. Applying the algorithms yield the mappings $\textsc{bank}_a(i,j) = i \mod 3$ and $\textsc{offset}_a(i,j) = j \mod N$. The HLS tool suite is assumed to synthesize properly the read reuse across references when applying loop pipelining, this is not the purpose of this work. Indeed only three reads per iteration are required.

5 Preliminary results

This section presents the preliminary results obtained with our multibanking approach.

We have applied our algorithm using the fkcc scripting tool [2] on the motivating kernel, assuming a sequential execution and the arrays references to be accessed in parallel for each iteration. The synthesis results were obtained using VivadoHLS 2019.1, targeting a Kintex 7 FPGA (XC6K70T-FBV676-1). We transformed each array reference $A[u(\vec{i})]$ as $\hat{A}[\textsc{bank}_A(u(\vec{i})), \textsc{offset}_A(u(\vec{i}))]$. Then, we used the VivadoHLS array partitioning pragma on the bank dimension.

Our results are depicted on the following table. The tool indicates the memory contention for the base kernel (base) are resolved on the transformed kernel (opt). The additional resources are due to the multibanking circuitry (steering logic). The overall latency is reduced from 79380 cycles to 31763 cycles. We suspect the speedup to be mitigated by the cost of the multibanking circuitry.

6 Conclusion

In this report, we have presented a unified, general HLS algorithm for multibanking, using the polyhedral model. Our approach minimizes the overall size of memory banks, without hindering

Kernel	Latency	BRAM	DSP	FF	LUT
conv2D-simple original	**79380**	0	0	471	981
conv2D-simple opt	**31763**	0	0	1013	1692

Table 1: Synthesis results

parallel memory accesses. Preliminary results encourage to pursue with this approach.

In the future, we plan to extend the field of experimental validation to more general linear algebra kernels under pipelining constraints. Also, we plan to explore how to minimize bank size separately, and how the schedule might adapted in the cases where data reuse mitigates the improvements.

References

[1] *Nios II C2H Compiler User Guide*, November 2009. Version 9.1. `http://www.altera.com`.

[2] Christophe Alias. fkcc: the Farkas Calculator. In *10th Workshop on Tools for Automatic Program Analysis*, Lecture Notes in Computer Science, Porto, Portugal, December 2019. Springer.

[3] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, 2007.

[4] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *ACM SIGDA Intl. Conference on Design, Automation and Test in Europe (DATE'13)*, Grenoble, France, 2013.

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

[6] Mentor CatapultC high-level synthesis. `http://www.mentor.com/products/esl/high_level_synthesis`.

[7] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems*, 16(2), 2011.

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

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

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

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

[12] Luca Gallo, Alessandro Cilardo, David Thomas, Samuel Bayliss, and George A. Constantinides. Area implications of memory partitioning for high-level synthesis on FPGAs. *Conference Digest - 24th International Conference on Field Programmable Logic and Applications, FPL 2014*, 2014.

[13] Gaut: High-level synthesis tool from C to RTL. `http://www-labsticc.univ-ubs.fr/www-gaut`.

[14] Impulse-C, accelerate software using FPGAs as coprocessors. `http://www.impulseaccelerated.com`.

[15] Bryant Jones and Darrin M. Hanna. Automatic cache partitioning method for high-level synthesis. *Microprocessors and Microsystems*, 67:71–81, 2019.

[16] Louis Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA*, pages 29–38, 2013.

[17] Patrice Quinton and Vincent van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 1(2):95–113, 1989.

[18] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 241 of *Lecture Notes in Computer Science*, pages 488–503. Springer Berlin Heidelberg, 1986.

[19] Samuel Rogers and Hamed Tabkhi. Locality aware memory assignment and tiling. *Proceedings - Design Automation Conference*, Part F1377, 2018.

[20] Ugh: User-guided high-level synthesis. `http://www-asim.lip6.fr/recherche/disydent/disydent_sect_12.html`.

[21] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. Memory partitioning for multidimensional arrays in high-level synthesis. *Proceedings - Design Automation Conference*, 2013.

[22] Yuxin Wang, Peng Zhang, Xu Cheng, and Jason Cong. An integrated and automated memory optimization flow for FPGA behavioral synthesis. *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pages 257–262, 2012.

[23] Shouyi Yin, Zhicong Xie, Chenyue Meng, Leibo Liu, and Shaojun Wei. Multibank memory optimization for parallel data access in multiple data arrays. *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 07-10-Nove, 2016.

Contents

1 Introduction **3**

2 Preliminaries **3**
2.1 Multibanking 3
2.2 Polyhedral model 4

3 Related work **5**
3.1 Memory partitioning 5
3.2 Data reuse 6
3.3 Cache partitioning 6

4 Our multibanking algorithm **7**
4.1 Deriving the bank mapping 7
4.2 Deriving the offset mapping 8
4.3 Our algorithm 8
4.4 An example 10

5 Preliminary results **10**

6 Conclusion **10**

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

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

ISSN 0249-6399