Affine Multibanking for High-Level Synthesis

Ilham Lasfar, Christophe Alias, Matthieu Moy, Rémy Neveu, Alexis Carré
Laboratoire de l'Informatique du Parallélisme
CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon, France
Firstname.Lastname@inria.fr

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 must be designed from scratch by the developer. In this short paper, 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 accesses will target distinct banks to avoid data transfer serialization. Our algorithm is able to reduce the number of banks and the maximal bank size. Preliminary evaluation shows promising results.

Keywords High-Level Synthesis, Multibanking

ACM Reference Format:
Ilham Lasfar, Christophe Alias, Matthieu Moy, Rémy Neveu, Alexis Carré. 2022. Affine Multibanking for High-Level Synthesis. In *Proceedings of 12th International Workshop on Polyhedral Compilation Techniques (IMPACT'22).* ACM, New York, NY, USA, 3 pages.

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 limitated. 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, 3, 6, 7, 9], 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.

In this short paper, we outline 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 [4, 10, 11], usually on convolution-like kernels with differents patterns. In a different context, approaches for false sharing removal [12] expose interesting ideas – though not directly transposable. We outline 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 *reduces the number of banks and the maximal bank size*, without hindering parallel accesses.

This paper is structured as follows. Section 2 illustrates the problem of multibanking on a motivating example. Section 3 outline the main ideas of our algorithm for multibanking. For a complete description, the reader is refered to [8]. Section 4 outlines our preliminary results. Finally, Section 5 concludes this paper and outlines future work.

IMPACT'22, June 20, 2022, Budapest, Hungary
2022.

2 Multibanking

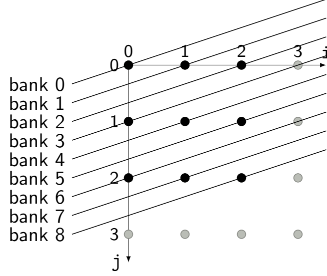
We illustrate the problem on the 2D convolution product, depicted in Figure 1.(a). The loop is assumed to be executed in sequence. For each iteration, array accesses are done in parallel. Because of memory limitation, parallel references must be mapped to different banks. A solution depicted on Figure 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. These functions define an *affine multibanking*. 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 $\sigma(\vec{N})$.

3 Overview of our multibanking algorithm

This section outlines our multibanking algorithm, we present the intuitions behind the derivation of the bank mapping and the offset mapping. For a deeper level of detail, reader is refered to [8].

```
for(i=1; i<N-1; i++)
  for(j=1; j<N-1; j++)
    out[i,j] =
      in[i-1,j-1]+in[i-1,j]+in[i-1,j+1]+
      in[i,j-1]  +in[i,j]  +in[i,j+1]   +
      in[i+1,j-1]+in[i+1,j]+in[i+1,j+1]; //S
```

(a) 2D convolution product



(b) Bank mapping

Figure 1. Motivating example

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

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, as an affine schedule would be. Once a dimension ϕ_d is found, we focus on the unresolved parallel conflicts (still in the same bank) with $\phi_a^d(\vec{i}) = \phi_b^d(\vec{j})$. And we iterate on the next dimension until all conflicts are resolved.

The number of banks might be estimated with 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:

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

The coefficients of the affine form σ^d might be minimized lexicographically, under the constraints of correctness exposed above. Iterating the process for each dimension yield a general, correct and efficient bank mapping. These constraints are turned to existentially guarded affine constraints

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

thanks to the affine form of Farkas lemma, following the lines of [5], formalized as a domain-specific language in [2].

Offset mapping Finding the offset in a bank might be achieved by the same algorithm, on different constraints. An offset mapping $\text{OFFSET}_a(\vec{i}) = \psi_a(\vec{i}) \mod \tau(\vec{N})$ is correct w.r.t. a schedule θ iff two array cells $a(\vec{i})$ and $a(\vec{j})$ mapped to the same bank and whose *liveness conflict* ($a(\vec{i}) \bowtie_\theta a(\vec{j})$) are mapped to *different offsets*:

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

The dimensions of ψ_a and $\tau(\vec{N})$ are computed incrementally in the same way as the bank mapping, with similar modulo minimization constraints. All the details are given in [8].

4 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 $\hat{A}[\text{BANK}_A(u(\vec{i})), \text{OFFSET}_A(u(\vec{i}))]$. Then, we used the VivadoHLS array partitioning pragma on the bank dimension.

Our results are depicted on Table 1. 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.

5 Conclusion

In this paper, we have outlined a unified, general HLS algorithm for multibanking, using the polyhedral model. Our approach reduces the overall size of memory banks, without hindering 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 as well as the trade-off surface/gain through a unified parametric formulation.

Acknowledgments

This work has been partially funded by the *fond recherche ENS-Lyon* through the HLSIMU project.

References

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

[2] Christophe Alias. 2019. fkcc: the Farkas Calculator. In *10th Workshop on Tools for Automatic Program Analysis (Lecture Notes in Computer Science)*. Springer, Porto, Portugal. https://hal.inria.fr/hal-02414224

[3] CatapultC [n. d.]. Mentor CatapultC High-Level Synthesis. http://www.mentor.com/products/esl/high_level_synthesis.

[4] 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 (2011). https://doi.org/10.1145/1929943.1929947

[5] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multi-Dimensional Time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420.

[6] Gaut [n. d.]. Gaut: High-Level Synthesis tool From C to RTL. http://www-labsticc.univ-ubs.fr/www-gaut.

[7] ImpulseC [n. d.]. Impulse-C, Accelerate Software using FPGAs as Coprocessors. http://www.impulseaccelerated.com.

[8] Ilham Lasfar, Christophe Alias, Matthieu Moy, Rémy Neveu, and Alexis Carré. 2021. *Affine Multibanking for High-Level Synthesis*. Research Report. Inria. https://hal.inria.fr/hal-03481328

[9] Ugh [n. d.]. Ugh: User-Guided High-Level Synthesis. http://www-asim.lip6.fr/recherche/disydent/disydent_sect_12.html.

[10] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. 2013. Memory partitioning for multidimensional arrays in high-level synthesis. *Proceedings - Design Automation Conference* (2013). https://doi.org/10.1145/2463209.2488748

[11] Yuxin Wang, Peng Zhang, Xu Cheng, and Jason Cong. 2012. An integrated and automated memory optimization flow for FPGA behavioral synthesis. *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC* (2012), 257–262. https://doi.org/10.1109/ASPDAC.2012.6164955

[12] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction*. 3–13.