Optimizing Remote Accesses for Offloaded Kernels: Application to High-Level Synthesis for FPGA

Christophe Alias Alain Darte Alexandru Plesco
Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon
firstname.lastname@ens-lyon.fr

Abstract

In the context of the high-level synthesis (HLS) of regular kernels offloaded to FPGA and communicating with an external DDR memory, we show how to automatically generate adequate communicating processes for optimizing the transfer of remote data. This requires a generalized form of communication coalescing where data can be transferred from the external memory even when this memory is not fully up-to-date. Experiments with Altera HLS tools demonstrate that this automatization, based on advanced polyhedral code analysis and code generation techniques, can be used to efficiently map C kernels to FPGA, by generating, entirely at C level, all the necessary glue (the communication processes), which is compiled with the same HLS tool as for the computation kernel.

Categories and Subject Descriptors B.5.2 [*Register-Transfer-Level Implementation*]: Design Aids—Automatic Synthesis; D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Design, Experimentation, Performance, Theory

Keywords Polyhedral optimizations, communication coalescing, pipelined processes, DDR memory, FPGA, HLS

1. Introduction

Most HLS tools for C-like languages [9], e.g., Catapult-C, C2H, Gaut, Impulse-C, Pico-Express, Spark, Ugh, use state-of-the-art back-end compilation techniques and are thus able to derive an optimized internal structure. However, integrating the automatically-generated hardware accelerators within the complete design, with optimized communications, synchronizations, and local buffers, remains a hard task, reserved to expert designers. In addition to the VHDL glue that has to be added, the input program must often be rewritten. For HLS tools to be viable, this tricky and error-prone step should be automated too. This paper shows how the handmade restructuring of [2], developed on top of Altera C2H HLS tool, can be *fully automated*, entirely at source level (i.e., in C).

We focus on the optimization of hardware accelerators that work on a large set of data to be transferred from a DDR memory at the highest possible rate, and possibly temporarily stored locally. For such a memory, making sure that successive requests access the same row (such accesses are pipelined an order of magnitude faster) is a direct way of improving the performances: if not, the hardware accelerator, even if its computational part is highly-optimized, keeps stalling and runs at the rate of the (unoptimized) DDR accesses. Our technique relies on loop tiling to increase the granularity of computations and communications. In each strip of tiles, transfers from/to the DDR are performed in a pipelined double-buffering fashion thanks to the introduction of communication processes in addition to the initial computation process. The accesses within each process are pipelined thanks to fine-grain software pipelining while the execution of the different processes is orchestrated thanks to coarse-grain software pipelining. Data reuse within a strip is fully exploited to avoid remote accesses when data are already available locally in the accelerator. To reduce the size of local storage, loads from the DDR (resp. stores to the DDR) are done as late (resp. soon) as possible. This requires a generalized form of communication coalescing, where loads are performed even when the external memory may not be up-to-date. Local memories are automatically generated, using allocations with modulo [1, 10, 12], to store the communicated data and exploit data & memory reuse.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

2. Generalized Communication Coalescing

Our method can be applied to offload a kernel on which loop tiling [16] and polyhedral code transformations can be applied, i.e., a set of `for` nested loops, manipulating arrays and scalar variables, where loop bounds, `if` conditions, and access functions, are affine expressions of surrounding loop counters and structure parameters. This model can be extended through approximations when access functions or `if` conditions are not fully analyzable.

Example The code of Fig. 1 computes, in array c, the product of two polynomials of degree N, stored in arrays p and q. The offloaded kernel is the second set of loops. If commutativity and associativity are not exploited, loops are not permutable. A possible tiling is specified from the transformation $\theta : (i, j) \mapsto (N - j, i)$ (i.e., loop interchange + reversal of the outer loop), see Fig. 1. In each horizontal tile strip, tiles are pipelined so that the transfers of a tile overlap with the computations of the previous tile. Communications are optimized resulting in maximal inter-tile (resp. intra-tile) reuse for q (resp. p) and some intra- and inter-tile reuse for c (for adjacent tiles). For example, the elements of c that are loaded (resp. stored) before (resp. after) each tile are shown in grey (resp. black).

```
for (i=0; i<=2*N; i++) {
  c[i] = 0;
}

for (i=0; i<=N; i++) {
  for (j=0; j<=N; j++) {
    c[i+j] += p[i]*q[j];
  }
}
```

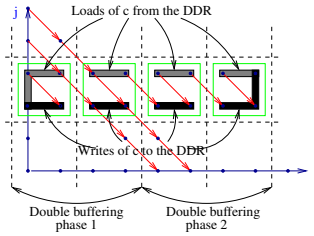

Figure 1. Product of polynomials example

In the tiled code, iterations are identified by a 4d vector (I, J, ii, jj) where $(ii, jj) = \theta(i, j)$, $I = \lfloor ii/b \rfloor$, $J = \lfloor jj/b \rfloor$, and b is the tile size. The counters I and J iterate over the tiles, ii and jj within a tile. For a tile strip indexed by I, a fixed b, our technique derives, thanks to parametric linear programming, the set Load(J) of array elements $c(m)$ loaded before the tile (I, J). With $b = 10$, we get Load$(J) = \{m \mid \max(0, N-10I-9) \leq m \leq N-10I\}$ if $J = 0$ and $\{m \mid \max(1, 10J) \leq m + 10I - N \leq \min(N, 10J + 9)\}$ otherwise, which corresponds to the sets of loads depicted in Fig. 1.

We now give the main principles of our method to select the array regions to be loaded from and stored to the external DDR memory. This step impacts the amount of communications, the lifetimes of array elements in the local memory, and the size of this memory. Details are provided in the companion reports [3, 4].

To perform data transfers, the naive solution is to access the DDR for each remote data access. No local memory is needed but the latency to the DDR is paid for each access, roughly 400 ns on our platform. However, if data accesses are reorganized by blocks on the same row, thanks to loop tiling, and fully pipelined, the accelerator can work at full rate, receiving 32 bits every 10 ns. This can be done thanks to *communication coalescing* – a standard technique used in compilers of parallel languages and scratch-pad memory optimizations [5–8, 13–15] – which amounts to hoist transfers out of a tile and regroup the same accesses to eliminate redundancy. The form of communication coalescing we develop is more general as it exploits, *at the granularity of individual array elements*, not only intra-tile reuse but also inter-tile reuse, even if data dependences exist between tiles, while minimizing the lifetimes of array elements in the local memory. Usually, the approach is to load, just before executing a tile, all the data read in the tile, then to store to the DDR all data written in the tile, without exploiting inter-tile data reuse. The other solution is to first load all data needed in a tile strip, to execute all tiles in the strip, and finally to store to the DDR all data produced by the strip, in other words, to hoist communications outside the innermost tile loop. This exploits data reuse but requires a large local memory to store all needed data. Also, computations cannot start before all data have arrived. Another important difference is that our technique performs loads from the external memory *during the execution of the tile strip*, and before actual stores, thus even when the external memory is not fully up-to-date. This may cause memory consistency problems that need to be addressed. We solve this issue by generating *exact* communication sets when possible, in a way similar to exact data-flow analysis [11], and by defining valid approximations otherwise.

Our strategy consists in scheduling a load request just before a first read (unless previously written) and a store request just after a last write. Data are loaded/stored in a strip *only once* and, between the first and last accesses, they are kept and used (read and written) in local memory, exploiting data reuse. As a bonus, this method handles naturally the case where dependences exist between tiles: as data involved in inter-tile dependences are kept in local memory, the sequential execution of tiles guarantees the program correctness. Another consequence is that, unlike previous approaches where the resulting lifetimes of array elements are identical (either between the first and last tile, or just within a tile), memory allocation based on bounding box as in [5, 14, 15] is not enough: to exploit different lifetimes, modular mappings [1, 10, 12] are more suitable. For the previous example, a memory of only $3b$ elements is used to store the elements of array c, and even only $2b$ if the first tile of a strip does not overlap with the second tile.

3. Implementation and Experimental Results

We implemented all necessary program analysis, generation of communicating sets, and code generation for the different communicating processes, making an extensive use of the parametric linear programming tool PIP (`www.piplib.org`). Our prototype generates, from the C code of a small kernel to be optimized, a C code that implements a double-bufferized version of it. This code can be simulated using linux processes, FIFOs, and shared memories. Its different processes are then synthesized and integrated automatically using C2H and Altera SOPC builder. Before, we currently still need to do a few modifications by hand, such as inserting some adequate pragmas for C2H, linearizing array addresses with the right base addresses, instantiating memories in the SOPC builder, changing some arrays into non-aliasing pointers. All these changes are systematic, but not integrated yet in our code generator.

For the 3 kernels analyzed in [2], we retrieve the performances of the versions optimized by hand. They can run 6x or more faster than the direct implementations (the maximal speed-up is 8, if, in the initial code, successive DDR accesses are in different rows). Note that these speed-ups are obtained not because computations are parallelized (tiles are run sequentially) but because DDR requests are reorganized, fully pipelined, overlapped with computations, and because data reuse is exploited. These first experimental results show that our method is effective and promising compared to handmade design. To our knowledge, this is the first time, in the context of HLS, that such accelerators are automatically generated.

References

[1] C. Alias, F. Baray, and A. Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM Conference LCTES'07*, San Diego, USA, June 2007.

[2] C. Alias, A. Darte, and A. Plesco. Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators. In *IEEE Int. Conference ASAP'10*, pages 329–332, July 2010.

[3] C. Alias, A. Darte, and A. Plesco. Program analysis and source-level communication optimizations for HLS. TR 7648, Inria, June 2011.

[4] C. Alias, A. Darte, and A. Plesco. Kernel offloading with optimized remote accesses. TR 7697, Inria, July 2011.

[5] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *ACM Symp. PPoPP'08*, pages 1–10, 2008.

[6] D. Chavarría-Miranda and J. Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *ACM Symposium PPoPP'05*, pages 14–25, Chicago, IL, USA, 2005.

[7] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. In *IEEE Int. Conf. on Parallel Arch. and Compilation Techniques (PACT'05)*, pages 267–278, 2005.

[8] J. Cong, H. Huang, C. Liu, and Y. Zou. A reuse-aware prefetching scheme for scratchpad memory. In DAC'11, pages 960–965, 2011.

[9] P. Coussy and A. Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[10] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, Oct. 2005.

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

[12] E. D. Greef, F. Catthoor, and H. D. Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23:1811–1837, 1997.

[13] I. Issenin, E. Borckmeyer, M. Miranda, and N. Dutt. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM TODAES*, 12(2), Apr. 2007. Article 15.

[14] A. Leung, N. Vasilache, B. Meister, M. M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *ACM Workshop GPGPU'10*, pages 51–61, Mar. 2010.

[15] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC'02*, pp. 628–633, 2002.

[16] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic, 2000.