Optimizing DDR-SDRAM Communications at C-level for Automatically-Generated Hardware Accelerators

An Experience With the Altera C2H HLS Tool

Christophe Alias, Alain Darte, and Alexandru Plesco
LIP, UMR 5668 CNRS—ENS Lyon—UCB Lyon—Inria, France
Firstname.Lastname@ens-lyon.fr

***Abstract*—Thanks to efficient scheduling, resource sharing, and finite-state machines generation, high-level synthesis (HLS) tools are now more mature for generating hardware accelerators with an optimized internal structure. But interfacing them within the complete design, with optimized communications, to achieve the best throughput remains hard. Expert designers still need to program all the necessary glue (in VHDL/Verilog) to get an efficient design. Taking the example of C2H, the Altera HLS tool, and of accelerators communicating to an external DDR memory, we show it is possible to restructure the application code, to generate adequate communication processes, in C, and to compile them all with C2H, so that the resulting application is highly-optimized, with full usage of the memory bandwidth. In other words, our study demonstrates that HLS tools can be used as back-end optimizers for front-end optimizations, as it is the case for standard compilation with high-level transformations developed on top of assembly-code optimizers. We believe this is the way to go for making HLS tools viable.**

***Keywords*-High-level synthesis tools, hardware accelerators, DDR SDRAM, optimized communications, program transformations, reconfigurable architectures, FPGA.**

I. I

High-level synthesis is a necessity mainly because the exponential increase in the number of gates per chip far outstrips the productivity of human designers. Besides, applications that need hardware accelerators usually belong to domains, where fast turn-around and time-to-market minimization are paramount. VHDL (or Verilog) experts can make a direct use of low-level synthesis tools, by programming at structural or register-transfer level (RTL). But, at this level, it is hard to perform high-level code optimizations, especially on multi-dimensional loops and arrays, which are important to exploit parallelism, pipelining, and to optimize memory transfers and organization. In the last decade, the generation of VHDL from higher-level specifications has been considered, in particular from C-like descriptions, with the introduction of extensions of (subsets of) C or languages such as Handel-C, and the developments of HLS tools, both in academia (e.g., Spark, Gaut, Ugh, Nisc, MMAlpha) and in industry (e.g., C2H, CatapultC, Impulse-C, PicoExpress).

These 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. However, this is only part of the complete design. In general, the designer seeks a pipelined solution with optimal throughput, where the mediums for data accesses (either to local memory or for outside communications) are saturated, i.e., a solution where bandwidth is the limiting factor. An HLS tool can be used to optimize the heart (the "compute" part) of the accelerator so that data are consumed and produced at the highest possible rate. But, in general, the designer has still to decompose the application into smaller communicating processes, to define the adequate memory organization or communicating buffers, and to integrate all processes in one complete design with suitable synchronization mechanisms. This task is extremely difficult, time-consuming, and error-prone. Some designers even believe that relying on HLS tools to get the adequate design is just impossible and they prefer to program directly in VHDL. Indeed, some HLS tools do not consider the interface with the outside world at all: data are assumed to be given on input ports, available for each clock cycle, possibly with a timing diagram to be respected. Then, the designer has to program the necessary glue (explicit communications, scheduling of communications, synchronizations) in VHDL or with ad-hoc libraries [1] [2, Chap. 9]. Some tools (e.g., Ugh, CatapultC) can rely on FIFO-based communication but the designer still needs to define the FIFO sizes, the number of data packed together in a FIFO slot (to provide more parallelism), and to prefetch data to hide memory latencies. Finally, some tools, such as C2H, allow direct accesses to an external memory and is (sometimes) able to pipeline them. But, again, the designer has to perform preliminary code transformations to change the computations order and the memory organization to hide the latency and exploit the maximal bandwidth.

This paper aims to demonstrate that such transformations, in front of HLS tools, are needed and can be automated. We believe it is a *sine qua non* condition for HLS tools to be a usable thus viable solution to hardware design, in the same way a traditional front-end compiler performs high-level optimizations on top of an assembly-code optimizer. The challenge is to be able to perform optimizations at C level that are directly beneficial when used in front of an HLS tool, *with no modification of the tool itself*. Our study is done with the C2H Altera tool, for accelerators with external accesses to a DDR-SDRAM memory (DDR for short), always keeping in mind that any code transformation we perform can be automated. Our contributions are the following:

1) We analyze C2H and we identify the features that make DDR optimizations feasible or hard to perform.
2) We propose a technique based on tiling, the generation

of communicating processes, and of software pipelining that can lead to fully-optimized DDR accesses.

3) We show how our scheme can be automated, with standard techniques from high-performance compilation.

II. C2H

One of the reasons why we chose C2H is that it relies on a quite direct mechanism to map the C syntax elements to the corresponding hardware, e.g., encoding a loop with a simple finite-state machine (FSM) instead of unrolling it, mapping each scalar variable to a register and each array to a distinct local memory, etc. This may seem a limitation but, at the same time, it gives a mean to control what the HLS tool produces, which is particularly important when used with source-to-source preprocessing, as we do. This requirement for predictability and control is also the basis of Ugh [2, Chap. 10], where each scalar variable is register-allocated at C level, by the user, to guide the hardware generation.

C2H [3] creates a custom hardware accelerator, described by a C function, offloading the Nios II processor. Most C constructs (pointers, structures, loops, subfunction calls) are supported. Integrated in the development flow of Altera FPGAs (with Quartus II, SOPC Builder, Nios II IDE), the accelerator can communicate, not only through FIFOs, but also using memory mapped address space connection. This eliminates the need for the hardware interface designer to integrate the accelerator in the whole system as required for many HLS tools [2]. The communication interface also supports pipelined memory accesses, a mandatory optimization to achieve good performances, which are, in general, limited by external data transfer bandwidth and rate, not by a lack of parallelism within the function to be accelerated.

The accelerator is controlled by several synchronized FSMs, one for each function or loop. Each loop is software-pipelined to optimize its CPLI (cycles per loop iteration). Memory transactions are pipelined with an optimistic latency (the FSM stalls if the data arrives later) and implicit FIFOs are created to store transferred data. If a loop contains another loop, the FSM of the outer loop stalls at the cycle containing the inner loop and waits for its end. With this hierarchical principle, each time the accelerator enters a loop involving communications, a latency penalty is incurred as the loop pipeline needs to be restarted (see Fig. 1 for 2 nested loops). For example, with C2H, the inner loop of a simple matrix-matrix product can be optimized to have CPLI 1 but, then, a latency of 43 cycles is paid for each iteration of the outer loop, due to the long pipeline involved to access data.

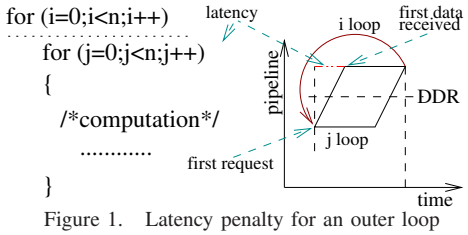


Figure 1. Latency penalty for an outer loop

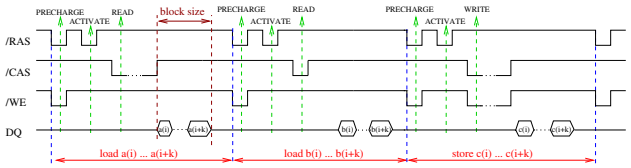


Figure 2. Accesses for optimized vector sum

Unlike PicoExpress, which relies on the Omega Library (see [2, Chap. 4]), dependence analysis in C2H is limited to an analysis of "names", with no analysis of array elements. Some potential aliasing can be removed, thanks to the pragma restrict but, still, this weakness is a real difficulty for source-to-source transformations. Other pragmas can be used to specify the connections of the generated communication ports and how many successive transfers can be performed without requiring to rearbitrate (pragma arbitration share). See [3] and the longer version of this paper [4] for details on C2H features and limitations.

III. O DDR

We target a particular class of accelerators: those working on a large data set that cannot be entirely stored in local memory, but need to be transferred from a DDR at the highest possible rate, and possibly stored temporarily locally. The maximum throughput for accesses to a DDR (see the JEDEC specification) is when the state changes in the FSM of its controller are reduced. In particular, in our study (DDR-400 128Mbx8, size 16MB, CAS 3 at 200MHz), a read is at least 6.5 times faster if it does not imply a change of row.

Consider the example of the sum of two (long) vectors:

```
int vector_sum (int* __restrict__ a, int* __restrict__ b,
                int* __restrict__ c, int n) {
    int i;
    for (i=0; i<n; i++) c[i] = a[i] + b[i];
    return 0;
}
```

In software, a cache imposes a data burst transfer of the cache line size, even though data are accessed one by one. Hardware accelerators usually do not have a cache, because of its high price compared to SRAM memory and because of the regularity of the accelerated algorithms. In this example, if a, b, and c are much larger than the DDR row size then most accesses belong to different rows, resulting in an important performance penalty. Thus, we seek transfers with as many successive reads to the same row as possible (same for writes), in particular communications by blocks, to obtain the time diagram of Fig. 2, with an optimized DDR usage.

IV. A

The previous sections identified two important reasons for performance loss. The *row change penalty*, due to consecutive accesses in different DDR rows, can occur in inner loops with a direct impact on the accelerator throughput. The *data fetch penalty*, due to nested loops with DDR accesses, occurs less often (not for inner loops, unless not pipelined), but with a higher penalty. To get better performances, the code must be restructured so that: a) arrays are accessed by blocks of the same DDR row; b) the accesses re-organization should

not increase the CPLI of computations; c) nested loops with remote accesses should be avoided to not pay data fetch latencies; d) all necessary house-keeping should be written in C and synthesized with C2H. For that, a local memory may be used to store data that cannot be consumed immediately.

To get accesses per block, a natural solution is to use strip-mining and loop distribution as follows:

```
for (i=0; i<MAX; i=i+BLOCK) {
  for(j=0; j<BLOCK; j++) a_tmp[j] = a_in[i+j];//prefetch
  for(j=0; j<BLOCK; j++) b_tmp[j] = b_in[i+j];//prefetch
  for(j=0; j<BLOCK; j++) c_out[i+j] = a_tmp[j] + b_tmp[j];
}
```

C2H schedules the 2 independent prefetch loops in parallel, thus requests of `a` and `b` are still interleaved (which is bad). With `arbitration share`, this interleaving can be avoided but only for a limited block size. However, data fetch penalties are paid for each iteration of the `i` loop. A possibility is to unroll the inner loops and to store each data read in a different scalar variable. With some luck, the data may be fetched by the scheduler in the textual order of the requests. The downside is the code explosion and hence the resource need explosion. Also, it requires a non parametric unrolling factor. A more involved solution, similar to the juggling technique [5], is to linearize the 3 inner loops into one loop, emulating the desired behavior thanks to an automaton that retrieves the original indices. We tried many variants to implement this technique at source level (see [4]), with different pointers, different writing, trying to enforce dependences when needed and to remove false dependences with `restrict`. We did not find any satisfactory solution. Either the code is potentially incorrect, depending on the schedule, or its CPLI increases, or it is not pipelined at all.

These considerations pushed us towards a more involved solution with several communicating accelerators (see the template architecture in Fig. 3). The data required for a given block of computations are transferred in the order desired for optimizing `DDR` accesses using a double buffering approach implemented by two accelerators `BUFF0_LD` and `BUFF1_LD`. This allows the use of single-port local memories, usually preferred over dual port ones. Also, with two accelerators, the data transfer of one can hide the data fetch penalty of the other one. The generated accelerators are represented as bold rounded rectangles, local memories as normal bold

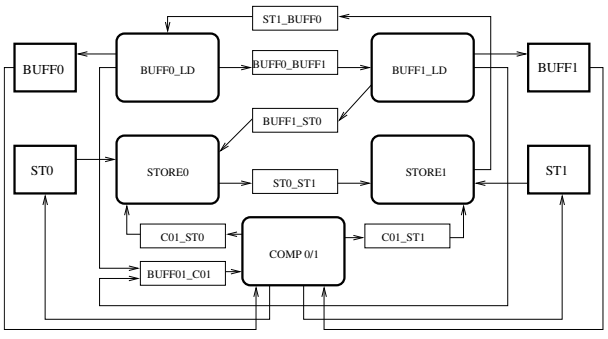


Figure 3. Accelerators module architecture

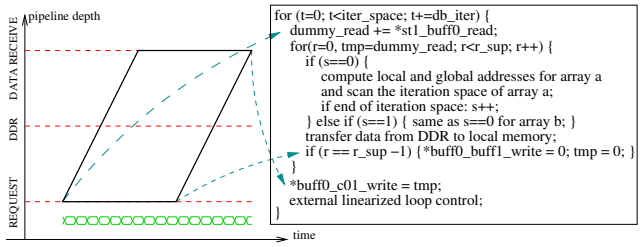


Figure 4. Simplified generic template C code

rectangles, and the rest are FIFOs. In this design, FIFOs are used only for synchronization. The arrays on which the computations are performed are located in local memories.

Fig. 4 shows the template code of the `BUFF0_LD` accelerator for a situation similar to the vector sum example. The code has two nested loops. The outer loop iterates over the blocks (tiles), here each tile is a block of array `a`, followed by a block of `b`. The inner loop uses a juggling-like strategy to emulate the traversal of the read requests, in the right order. After the desired local and external addresses are computed, the data is transferred from external to local memory. As the code has only reads, it can be fully pipelined with CPLI 1. The same is true for the writing accelerators. The key point is how this accelerator is synchronized, at C level, with the others. Before each inner loop invocation, the accelerator performs a blocking read from a synchronization FIFO. We guarantee that the inner loop starts after this synchronization with the dependence on the variable `dummy_read`. At the last inner loop iteration, the accelerator has finished sending requests to memory: a synchronization token is sent so that another accelerator can start requesting data from the memory controller. When the inner loop FSM receives all requested data, a synchronization token is sent to the computation accelerator. As for `dummy_read`, the variable `tmp` guarantees this synchronization occurs after the loop.

With this generic technique, it is possible to fetch, in an optimized blocked manner, many blocks of different sizes, each with its individual access addresses, without increasing the hardware resources too much (the only increase is the state machine size of the inner loop). Another advantage is that we can dispatch one or multiple arrays to multiple memories. This should be used jointly with optimizations of the computation accelerator so that parallel computations can be performed on data from different local memories.

Fig. 5 shows a possible synchronization of the whole system, with two kinds of synchronizations, due to data dependencies (e.g., from `BUFF0_LD` to `COMP0`) and due to resource utilization (e.g., from `BUFF0_LD` to `BUFF1_LD`). Here, the `DDR` transfers are still not optimal: there is a small gap between the load and the store, due to the conservative synchronization between `BUFF1_LD` and `STORE0`. We indeed assumed here, for the sake of illustration, that `STORE0` may write to the location that `BUFF0_LD` reads. A generic solution is given in [4], formulating the problem as a coarse-grain software pipelining at block level, considering each

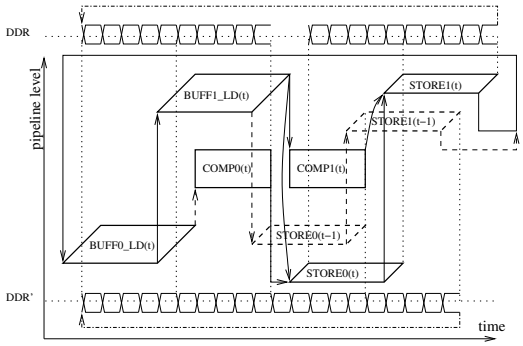


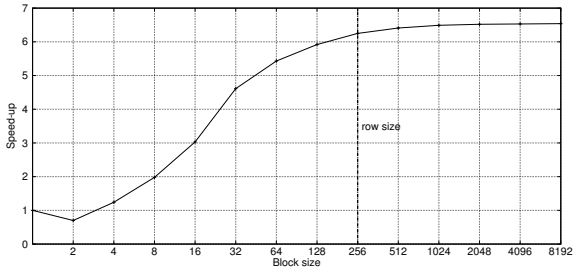
Figure 5. Synchronization diagram

accelerator as a (cyclic) macro-task that reads or writes (local and external) memories, in a common outer block loop to be pipelined. As in Fig. 3, the different tasks synchronize each other, at the block boundary, using blocking FIFOs of size 1, each acting as a token. These synchronizations, all together, enforce a particular pipelined execution of the blocks computed by the accelerators, similar to Fig. 5.

V. F :

We focused on optimizing DDR transfers for a hardware accelerator, automatically-generated from a C description. Our study demonstrates that such an optimization is possible with high-level transformations, fully developed in C, on top of a HLS tool (in our case, Altera C2H), without modifying it. We proposed a generic solution, based on communicating accelerators, themselves compiled from C with the same HLS tool, in a form of meta-compilation. Fig. 6 is typical of the results we obtain for different block sizes (here, for the vector sum synthesized on Altera Stratix II EP2S180F1508C3, for vectors of size 16K): a small initial degradation due to the extra FIFO management, an increasing speed-up as the row change penalty becomes less frequent, and a plateau once the block size gets close to the DDR row size.

The generic solution of Section IV was designed to be automated. The synchronized communicating accelerators define a kind of run-time system, described at C level and compiled by C2H itself, on top of which high-level transformations are made to re-organize the code and fill the communication and computation templates with the adequate codes. This automation requires both program analysis and code transformations, in particular optimizations based on polyhedral techniques, that were developed in the context of high-performance computing. We list here the main necessary steps. See more details and references in [4].



	LC ALUTs	Registers	Memory Bits	Freq. Max
Original (whole system)	4912	4517	68956	160 MHz
Optimized (whole system)	14110	14783	269148	139,04 MHz
Original (kernel alone)	564	1228	2048	307 MHz
Optimized (kernel alone)	1143	1603	1024	285 MHz

Figure 6. Speedups and resource usage

Loop tiling divides the kernel into blocks of computations to be executed with a double buffering scheme (additional *loop unrolling* by 2). *Communication coalescing* identifies, for a given tile, the set of data to be exchanged with the DDR, which is *scanned*, preferably row by row. *Array contration* defines a mapping function to convert indices of the global array (in the DDR) to local indices of a smaller array in which the transferred data are stored. Finally, nested loops *linearization* is used to avoid any data fetch penalty.

Considering high-level loop transformations to improve the performance and memory usage of hardware accelerators is not new [6]–[8]. However, so far, such transformations were either plugged in a HLS tool or required the development of special hardware structures, designed by hand, for optimizing transfers [9], [10]. We believe such our study is a necessary step to be able to consider HLS tools as back-end optimizers for source-to-source optimizers.

R

[1] A. Fraboulet and T. Risset, "Master interface for on-chip hardware accelerator burst communications," *Journal of VLSI Signal Processing*, vol. 2, no. 1, pp. 73–85, 2007.

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

[3] "Altera C2H: Nios II C-to-hardware acceleration compiler," http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html.

[4] C. Alias, A. Darte, and A. Plesco, "Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators," INRIA, Research Report 7281, May 2010.

[5] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien, "Constructing and exploiting linear schedules with prescribed parallelism," *ACM TODAES*, vol. 7, no. 1, pp. 159–172, 2002.

[6] P. R. Panda, N. D. Dutt, and A. Nicolau, "Exploiting off-chip memory access modes in high-level synthesis," in *IEEE/ACM Computer-Aided Design (ICCAD'97)*, 1997, pp. 333–340.

[7] A. Plesco and T. Risset, "Coupling loop transformations and high-level synthesis," in *SYMPosium en Architectures nouvelles de machines (SYMPA'08)*, Fribourg, Switz., 2008.

[8] H. Devos, J. Van Campenhout, and D. Stroobandt, "Building an application-specific memory hierarchy on FPGA," in *HiPEAC Workshop on Reconfig. Computing*, 2008, pp. 53–62.

[9] J. Park and P. Diniz, "Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines," in *ACM ISSS'01*, 2001, pp. 221–226.

[10] G. Stitt, G. Chaudhari, and J. Coole, "Traversal caches: A first step towards FPGA acceleration of pointer-based data structures," in *ACM/IEEE CODES-ISSS'08*, 2008, pp. 61–66.