INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

An Experience with the Altera C2H HLS Tool

Christophe Alias — Alain Darte — Alexandru Plesco

N° 7281

May 2010

ISSN 0249-6399 ISRN INRIA/RR--7281--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE – RHÔNE-ALPES

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

An Experience with the Altera C2H HLS Tool

Christophe Alias [*], Alain Darte [†], Alexandru Plesco[‡]

Thème : Architecture et compilation
Équipe-Projet Compsys

Rapport de recherche n° 7281 — May 2010 — 19 pages

Abstract: High-level synthesis tools are now getting more mature for generating hardware accelerators with an optimized internal structure, thanks to efficient scheduling techniques, resource sharing, and finite-state machines generation. However, interfacing them with the outside world, i.e., integrating the automatically-generated hardware accelerators within the complete design, with optimized communications, so that they achieve the best throughput, remains a very hard task, reserved to expert designers. In general, the designers are still responsible for programming all the necessary glue (most of the time in VHDL/Verilog) to get an efficient design. Taking the example of C2H, the HLS tool from Altera, and of hardware accelerators communicating to an external DDR-SDRAM memory, we show that 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 shows that HLS tools can be used as back-end compilers 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.

Key-words: High-level synthesis tools, hardware accelerators, DDR SDRAM, optimized communications, program transformations, FPGA.

[*] INRIA Researcher
[†] CNRS Researcher
[‡] PhD student at ENS Lyon

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Optimisation des communications DDR-SDRAM au niveau source pour la génération automatique d'accélérateurs matériels

Une expérience avec l'outil de synthèse haut-niveau C2H d'Altera

Résumé : Les outils de synthèse de circuit haut-niveau sont désormais assez matures pour générer des accélérateurs matériels avec une structure interne optimisée, grâce à des techniques efficaces d'ordonnancement, de partage des ressources et de génération d'automates de contrôle. Cependant, l'interface avec le monde extérieur, c'est-à-dire l'intégration des accélérateurs générés automatiquement, avec des communications optimisées pour obtenir une bonne bande passante, reste une tâche très difficile, réservée aux experts. En général, les concepteurs doivent encore se charger de la programmation de la glue logique (le plus souvent en VHDL/Verilog) pour obtenir une réalisation efficace. Prenant l'exemple de C2H, l'outil de synthèse haut-niveau d'Altera et d'accélérateurs matériels communiquant avec une mémoire DDR-SDRAM externe, nous montrons qu'il est possible de restructurer le code de l'application pour générer des modules de communication efficace, en C, et de les compiler avec C2H, de telle manière que l'application résultante soit optimisée en bande passante mémoire. En d'autres termes, notre étude montre que les outils de synthèse haut-niveau peuvent être utilisés comme des compilateurs back-end après des optimisations front-end, comme c'est déjà le cas en compilation standard avec des transformations haut-niveau développées en amont des optimiseurs de code assembleur. Nous pensons que c'est la voie à suivre pour que les outils de synthèse haut-niveau soient viables.

Mots-clés : Outils de synthèse haut-niveau, accélerateurs matériels, DDR-SDRAM, communications optimisées, transformations de programme, FPGA.

1 Introduction and motivation

High-level synthesis (HLS) has become 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, like telecommunications and game platforms, where fast turn-around and time-to-market minimization are paramount.

Today, synthesis tools for FPGAs or ASICs come in many shapes. At the lowest level, there are proprietary Boolean, layout, and place and route tools, whose input is a VHDL or Verilog specification at the structural or register-transfer level (RTL). Direct use of these tools is still difficult or, at least, restricted to expert designers and VHDL programmers. Indeed, a structural description is completely different from an usual algorithmic language description, as it is written in term of interconnected basic operators. Also, synthesis tools have trouble handling loops. This is particularly true for logic synthesis systems, where loops are systematically unrolled (or considered as sequential) before synthesis. More generally, a VHDL design is at a too low level to allow the designer to perform, easily, higher-level code optimizations, especially on multi-dimensional loops and arrays, which are of paramount importance to exploit parallelism, pipelining, and perform memory optimizations.

In the last decade, important efforts have been made to make possible the generation of VHDL from higher-level specifications, in particular from C-like descriptions, with the introduction of languages extensions of (subsets of) C or languages such as Handel-C [2] and the developments of HLS tools, both in academia, such as Spark [24], Gaut [14], Ugh [26], Nisc [17], and in industry such as C2H [6], CatapultC [7], Impulse-C [15], Pico Express [20], to quote but a few. 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, for one given kernel. In other words, it can now be acceptable to rely on them for optimizing the heart of accelerators, i.e., the equivalent of back-end optimizations in standard (software) compilers.

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, in other words, a solution where bandwidth is the limiting factor. The HLS tool should then instantiate the necessary hardware and schedule computations inside the hardware accelerator so that data are consumed and produced at the highest possible rate. But, for most tools, the designer has still the responsibility to decompose his/her 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 today's 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 all the necessary glue (explicit communications, scheduling of communications, synchronizations) in VHDL or with ad-hoc libraries and [13] [9, Chap. 9]. Some other tools, e.g., [26, 7], can rely on FIFO-based communication. 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, also

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.

The goal of this paper is to demonstrate that such transformations, in front of HLS tools, are needed and can be automated. We believe that this is the *sine qua non* condition for making HLS tools a usable thus viable solution to hardware design, in the same way a traditional front-end compiler can perform high-level optimizations on top of any assembly-code optimizer. The challenge is thus to be able to perform code optimizations at C level that are directly beneficial when used in front of an HLS tool, with no modification of the tool itself. For that however, the HLS tool should behave in a predictable way, in particular with predictable performances. This may require to be able to pass higher-level information to the (back-end) HLS tool, such as non-aliasing information (as the pragma `restrict`), or to use the tool in a very specific and controllable way. Our study is done with the C2H Altera tool, for accelerators with external accesses to a DDR memory, 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.

2 C2H important features

Some HLS tools rely on a somehow straightforward mechanism to map the C syntax elements to their corresponding hardware parts, for example a loop is encoded into a simple finite-state machine and not unrolled, a scalar variable is mapped to a register and an array to a local memory with no memory reorganization, etc. This can be a disadvantage from the point of view of automatic code synthesis. Indeed, to get good performance results, the user should know very well the underlying synthesis concepts and methods together with the hardware structures that the tool generates from the C-level algorithmic description. This is a major issue of a majority of the HLS tools.

However, a direct correspondence between software and hardware can also be an advantage as it gives a mean to control what the HLS tool will produce. This is the approach of Ugh (user-guided HLS) [26], where each scalar variable is register-allocated at C level by the user to guide the hardware generation. This is also an advantage if the HLS tool is used with a source-to-source preprocessing tool, as we do. The more information the preprocessing could give to the HLS tool, the more precise version of the required hardware will be obtained. This is one of the reasons why we chose C2H, the HLS tool designed by Altera Corporation, as a target for our source-level optimizations.

We now recall the main features of C2H that will impact our technique. Further details are provided in the Nios II C-to-Hardware Acceleration Compiler (C2H) User Guide [6].

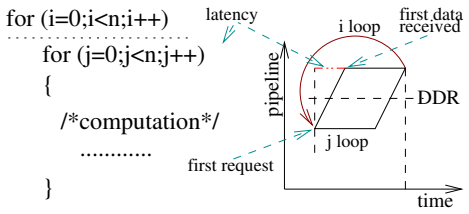


Figure 1: Latency penalty for an outer loop

2.1 Integration within the complete system

C2H is aimed to create a custom hardware accelerator that offloads the `Nios II` processor. The description of the hardware accelerator is passed as a function written in the `ANSI C` language. C2H supports most of the C constructs such as pointers, structures, loops, and subfunction calls. It is integrated in the development flow of the Altera FPGAs and works with `Quartus II`, `SOPC Builder`, and `Nios II IDE`. This makes the integration in the complete software/hardware design much easier. Being connected with other modules of the system thanks to the `Avalon` system interconnect fabric, the hardware accelerator can communicate, not only by means of FIFOs, but also using the mapped address space connection. In fact, this is the default communication method. [1] It eliminates the need for the hardware interface designer to accommodate and integrate the accelerator to the complete system as required in many popular HLS tools described in [9]. This communication interface also supports pipelined memory accesses, which is mandatory to achieve good performance of the final system. Indeed, most of the time, the performances are limited by the external data transfer bandwidth and rate, but not by a lack of parallelism within the function to be accelerated.

2.2 Inner Control with State Machines

The control unit of the accelerator is composed of finite state machines (FSM) that work synchronously. C2H generates a distinct FSM for each function, loop, and subfunction. Each loop is software-pipelined so as to optimize its CPLI (cycles [2] per loop iteration). If a loop contains another loop, the inner loop is considered as an atomic instruction for defining the software pipelining of the outer one, and the cycle of the outer FSM that contains the inner loop will stall and wait for its end before proceeding to the next cycle. The same happens for an inner subfunction. Memory transactions are also pipelined assuming an optimistic latency, and implicit FIFOs are created to store transferred data. Again, the FSM stalls if the data arrives later than expected.

This hierarchical FSM semantics is an important feature to consider for optimizing external DDR accesses, as illustrated in Fig. 1. Suppose the `j` loop reads data from an external DDR. Even if its CPLI is 1, which means that one data item is consumed at each cycle, its total latency can be quite long, due to a communication latency. This latency is paid as a penalty for each iteration of the `i` loop.

[1] To communicate using a FIFO, there should be a FIFO instantiated in the system by means of the `SOPC Builder` and connected using a memory mapped interface to the specified port of the accelerator.

[2] It is a "virtual" cycle, not a hardware clock cycle.

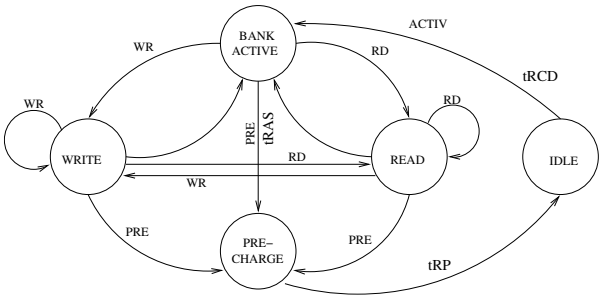


Figure 2: Simplified DDR state diagram

2.3 Pragmas for aliasing and connections

As many tools, the dependence analysis of C2H is limited to an analysis of "names" instead of memory locations, in particular, it has no dependence analysis of array elements, unlike Pico Express, which relies on the Omega Library (see [9, Chap. 4]). For example, C2H is enable to pipeline the following loop due to a false dependence:

```
for (i = 0; i < n; i++) a[i] = a[i] + 1;
```

However, some potential memory aliasing can be removed by the user, thanks to the pragma `restrict`, defined in the `ANSI C` standard as a pointer type qualifier.

Specific pragmas can also be used to specify the connections of the generated communication ports to other modules in the system. If no connection ports are specified, C2H connects each port to every other port in the system, since it does not know what memory address space the port associated with a pointer variable will address. This connection pragma reduces considerably the arbitration logic and thus maximizes the operating frequency of the bus.

Finally, an `arbitration share` pragma, defined to each master/slave port pair, can be used to specify how many transfers the master can perform with the slave, without requiring to rearbitrate. This pragma is useful to force consecutive accesses to a resource and obtain an improved throughput and latency. However, of course, it cannot re-order computations that belong to different cycles of a FSM.

3 Motivating examples with DDR accesses

Our goal is to optimize a particular class of hardware accelerators: those working on a large data set that cannot be completely stored in local memory, but need to be transferred from a DDR-SDRAM memory at the highest possible rate, and possibly stored temporarily locally. This section motivates, with three simple examples, why a naive use of C2H does not achieve the adequate performances.

3.1 DDR-SDRAM principles

We first recall the general principles of a DDR-SDRAM memory (DDR for short). For more details, refer to the JEDEC specification of the DDR standard [1]. Each DRAM unit is organized in multiple banks, from 4 (DDR1 standard) to 8 (latest DDR3 memories) and

the address is divided in 2 parts (row and column) to save I/O pins on the chip. Each bank has its own state machine as depicted in Fig. 2. Selecting a bank automatically activates a row.

A transition of the controller state machine takes some time, defined in the technical memory specification. For example, going from the PRECHARGE state to the IDLE state takes a minimum amount of time tRP. Also, when, after performing a read, the controller is in the BANK ACTIVE state and a read request to another row arrives, the controller has to meet some timing constraints before it can pass to the READ state on the other row. Considering for simplicity the most significant timing constrains, going from BANK ACTIVE, through PRECHARGE, IDLE, BANK ACTIVE, to READ takes at least a time equal to tPASSr = tRAS + tRP + 2*tCK (data transfer time). [3] In our study, we use a DDR memory with the following specification: DDR-400 128Mbx8, size of 16MB, and CAS of 3 at 200MHz memory clock. Using the memory specification parameters, tPASSr = 40ns + 15ns + 10ns = 65ns. A successive read to the same row takes tNORM = 2*tCK = 10ns time. In other words, a read from the same row is tPASS / tNORM = 6.5 times faster. In practice, for a non-optimal implementation of the DDR controller JEDEC specification, this ratio can be larger.

As can be deduced from above, the memory approaches its maximum throughput when the state changes in the DDR controller are reduced. If the hardware accelerator is optimized, as it is often the case, to have a CPLI equal to 1, its throughput directly depends on the throughput of the data accesses. If successive data items are accessed within the same row, the accelerator will work at full speed, otherwise it will stall, waiting for data to arrive. Thus, we seek transfers with as many successive reads or writes to the same row as possible, without changing the direction (read/write).

3.2 Examples

DMA transfer The first example is the synthesis of a DMA transfer from and to the external DDR memory:

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

Such a transfer occurs in software when invoking the function call memcpy. The presence of cache memory in modern processors imposes a data burst transfer of the cache line size from the main memory, even though data are accessed one by one. However, hardware accelerators usually do not have cache memory, because of the high price of the cache memory compared to SRAM memory and also the regularity of the algorithms that are accelerated. In this example, we are interested in the case where the size of the arrays a and b are much larger than the size of a row in the memory. In this case, for the majority of accesses, the elements a[i] and b[i] belong to different rows. To pass from the read of a[i] to the write of b[i], the DDR state machine has to pass through the states of PRECHARGE, BANK ACTIVE, and WRITE, with an important performance penalty. The resulting memory accesses (see Fig. 3) are very inefficient from the point of view of the DDR memory architecture.

[3] We do not include tRCD since the memory can accept a read before the previous one is fully executed (for writes it is more complicated).

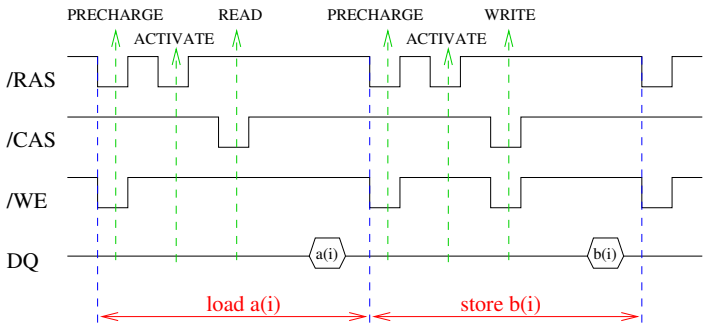


Figure 3: DMA transfer: DDR time diagram

Sum of two vectors This example computes the sum of two vectors a and b and writes it into the vector c:

```
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;
}
```

As previously, for each read or write, the state machine has to pass through all the states mentioned before, accessing alternatively an element of a, of b (both as a read), and of c (as a write), resulting in a very inefficient use of the DDR.

Matrix-matrix product The matrix-matrix product code given below has two linearized matrices a and b. The access to the matrix c was replaced by a local accumulator tmp, otherwise, as explained earlier, C2H detects false dependences and the resulting design is not pipelined.

```
int matrix_multiply (int* __restrict__ a,
  int* __restrict__ b, int* __restrict__ c,
  int n0, int m0, int m1) {
    int i, j, k, tmp;
    for (i=0; i<n0; i++)
      for (j=0; j<m1; j++) {
        tmp = 0;
        for (k=0; k<m0; k++)
          tmp += *(a+(i*m0+k)) * *(b+(k*m1+j));
        *(c+(i*m1+j)) = tmp;
      }
  return 0;
}
```

The inner loop has two interleaved read requests to the DDR, with the same access penalty seen for a and b in previous examples. Also, as explained earlier, each time the accelerator enters a loop, a latency penalty is incurred because the loop pipeline needs to be restarted. In this example, there are three nested loops. The i loop has only loop control and initialization states, and C2H schedules it with a small start-up latency of 5, i.e., it has 5 "virtual" cycles and the duration of each cycle depends on the inner FSM it contains. The j loop has the same states as the i loop, with a write to the matrix c.

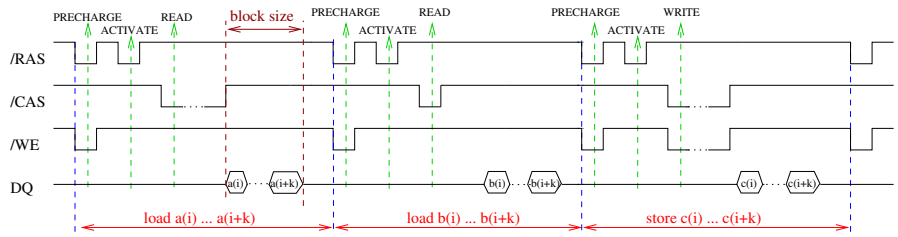


Figure 4: Accesses for optimized vector sum

Since a write can be done immediately after the data is ready, the "apparent" latency of the `j` loop is 7. However, the latency of the `k` loop is 43, because of the long pipeline involved to access the data needed for the computation, which leads to idle cycles for the `j` loop.

3.3 Optimizing `DDR` accesses

The previous examples illustrate two important reasons for performance loss, the first one due to the non-consecutive `DDR` accesses (*row change penalty*), the second due to nested loops containing `DDR` accesses (*data fetch penalty*). To summarize, for our platform and designs:

- at best, the accelerator can receive 32 bits every 10 ns.
- if successive data are not in the same row, the accelerator is able to receive 32 bits only every 65 ns, roughly.
- if, before sending a new request, the accelerator needs to wait the complete communication delay from the request to the arrival of a data, it will have to wait an order of magnitude longer. For simple C2H designs, this delay is roughly 400 ns (40 cycles at 100 MHz).

The row change penalty can occur in inner loops and thus directly impacts the throughput of the accelerator. The data fetch penalty occurs less often (not for inner loops, unless they are not pipelined), but with a higher penalty. To get better performances, the code must be restructured so that:

- arrays are accessed by blocks of elements belonging to the same row of the `DDR`;
- the re-organization of accesses should not increase the CPLI of the computations;
- nested loops containing data accesses should be avoided to not pay long data fetch latencies.
- all necessary glue and house-keeping should be written at C-level and compile into hardware by C2H.

For that, we will have to use the local memory to store some data that cannot be consumed immediately. The next sections explain the generic solution we designed to achieve these goals. For the vector sum, we will obtain the optimized time diagram of Fig. 4, with an optimal `DDR` usage.

4 A solution with communicating accelerators

We first show, with the vector sum example, why direct approaches do not work, due to C2H features/limitations. We then detail our solution, based on multiple accelera-

tors that orchestrate communications, and how the required transformations and code generations can be automated.

4.1 First attempts toward a solution

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];
}
```

The two prefetching loops occur in parallel since there is no data dependence between them, thus requests of `a` and `b` are still interleaved. The use of the `arbitration share` pragma could avoid the interleaving but only for a limited block size. It does not avoid the data fetch penalties paid for each iteration of the `i` loop due to the inner loops.

To avoid the data fetch penalties, one can unroll the inner loops and store each data read in a different scalar variable. If `BLOCK` is 8, the `i` loop has a CPLI of 16 and performs 16 read accesses, optimally. With some luck (or is it a feature of the scheduler of C2H?), the data are fetched in the textual order of the requests. The downside of this approach 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 [10], is to linearize the 3 inner loops into one loop `k`, emulating the desired behavior thanks to an automaton that retrieves the original indices. The code then looks like:

```
int ptr_local, ptr_ddr;
bi = 0; j = 0;
for (k=0; k<3*MAX; k++) {
 if (j==2) { *(c_out+i+bi) = *(a_tmp+i) + *(b_tmp+i); }
 else {
   if (j==0) { ptr_ddr = a+(i+bi); ptr_local = a_tmp+i;}
   else { ptr_ddr = b+(i+bi); ptr_local = b_tmp+i;}
   *ptr_local = *ptr_ddr; /* data transfer */}
 if (i++==BLOCK) { i=0; if (j++==3) {j=0; bi += BLOCK}}
}
```

Unfortunately, due to false dependences, C2H is now unable to pipeline the accesses. If the `restrict` pragma is added for `ptr_local` and `ptr_ddr`, the loop is pipelined, with CPLI equal to 1, but in some cases, depending on the scheduler and of runtime latencies, the code is incorrect: the computation should start after the last data of array `b` has arrived, not just after the request itself. The `restrict` pragma is too global, it cannot express the restriction between only two pointers. Also, with C2H, data requests in `if` instructions are still initiated, speculatively, so as to enable their pipelining, which, here, leads to interleaved reads and writes again. We tried many other variants of this code, with different pointers, different writing, trying to enforce dependences when needed and remove false dependences. 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.

4.2 Overlapping communications and computations with multiple accelerators

The previous discussion shows that it is inefficient, if not impossible, to write the code managing the communication in the code managing the computation. If it is placed in a previous loop, the accelerator has to wait for the data to arrive before starting the computation (data fetch penalty). If it is interleaved within the computation code, controlled by an automaton as for the juggling technique, it is very difficult to ensure that this extra housekeeping code does not alter the optimal data rate. This is even more true when trying to mix a communication code, whose CPLI should be 1, with computations at CPLI > 1 but, possibly, at higher frequency. A natural solution would implement the data transfer in a single-loop accelerator, synchronized with the computation accelerator. However, since all instructions from a loop are guaranteed to be executed only when the loop state machine finishes its execution, it is again not possible to enforce data coherency with a correct synchronization between the two accelerators. All these considerations pushed us toward a more involved solution that we now expose.

The template architecture is presented in Fig. 5. 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`. A dual buffering approach allows the use of single-port local memories, which consume fewer resources and are preferred over dual port ones, even if they are now usually available on FPGA platforms. More importantly, with two accelerators, the data transfer of one will be used to hide the data fetch penalty of the other one. The generated accelerators are represented as bold rounded rectangles, local memories as normal bold 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. 6 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 array `b`. The inner loop uses the same mechanism as the juggling code of Section 4.1 to emulate

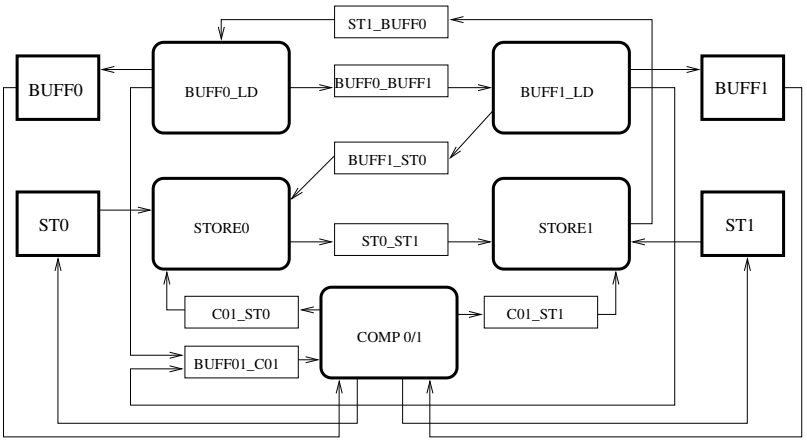


Figure 5: Accelerators module architecture

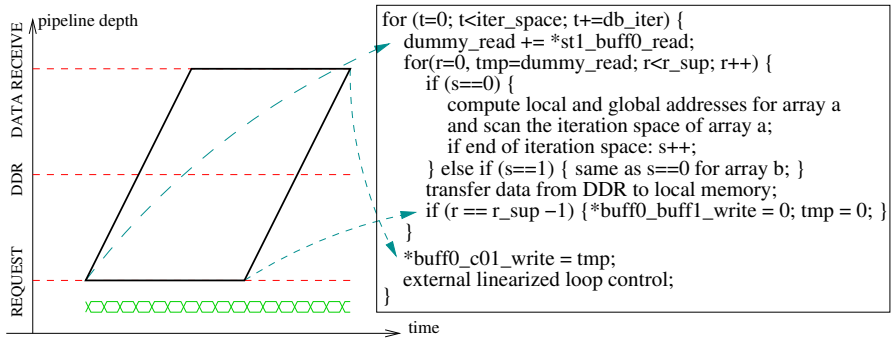


Figure 6: Simplified template C code

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. Here, as there are only reads, the code can be fully pipelined with CPLI equal to 1. The same is true for the writing accelerators.

The key point is how this accelerator is synchronized, at C level, with the other ones. Before each invocation of the inner loop, the accelerator performs a blocking read from a synchronization FIFO. Since C2H may schedule independent instructions in parallel, we guarantee that the inner loop starts after this synchronization by introducing a dependence with the variable `dummy_read`. At the last iteration of the innermost loop, the accelerator has finished sending requests to the memory, and a synchronization token is sent so that another accelerator can start requesting data from the memory controller. When the innermost loop state machine receives all the requested data from the memory, a synchronization token is sent to the computation accelerator. Similar to `dummy_read`, the variable `tmp` guarantees that this synchronization takes place 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. 7 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` could write to the location that `BUFF0_LD` reads. Without this assumption, the synchronization could be moved to the last request of `BUFF0_LD`. However, if the computation finishes later, there is still a gap due to the synchronization between `COMP0` and `STORE0`. It can be eliminated by reducing the computation time with parallelization techniques. Or one can shift all stores to the right (i.e., to delay them by one iteration) as depicted with dotted lines in Fig. 7. But this requires duplicating the local memory of the computed data, as `COMP0(t)`

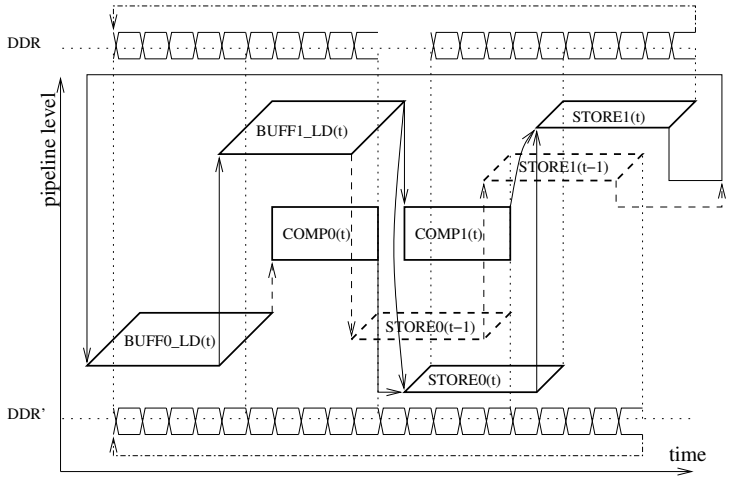


Figure 7: Synchronization diagram

now overlaps with `STORE0(t-1)`. Another solution, with no duplication, is given in Section 4.3, once the problem is formulated as a software pipelining problem.

4.3 Coarse-grain software pipelining

In Section 4.2, the optimization scheme we propose, with communicating accelerators, was illustrated with one particular synchronization pattern. We now show how such a synchronization structure can be found automatically.

Each (reading, computing, or writing) accelerator works at the block granularity, and iterates thanks to an outer loop over blocks. As depicted in Fig. 5, the different accelerators synchronize each other, at the block boundary, using blocking FIFOs of size 1, each acting as a token. Depending on the desired synchronization semantics, the token is sent either at the last iteration of the inner loop that scans a block, or just after this loop (see Fig. 6). In the first case, the token is used to enforce a *resource constraint*, i.e., to sequentialize the accesses to a given resource (here the communication medium on which `DDR` requests are sent). In the second case, the token is used to enforce a *dependence constraint*, e.g., to make sure that a data read in the local memory has indeed arrived. These synchronizations, all together, finally enforce a particular pipelined execution of the blocks computed by the accelerators, as depicted in Fig. 7.

What we did here is nothing but a coarse-grain *software pipelining* at block level, considering each individual accelerator as a macro-instruction that reads or writes (local and external) memories, in a common outer block loop to be pipelined. The standard approach is to define a directed multi-graph, where each vertex is a macro-instruction and each edge describes a dependence, as depicted in Fig. 8 for a simple read/write case with double buffering as in Section 4.2. Each vertex requires a particular resource, either a computing accelerator or the communication medium, if possible with corresponding durations. Edges are labelled with dependence distances, usually 0 (loop independent) or 1 (loop carried) and, if possible, with corresponding latencies. Additional dependences can be added, to constrain the problem, as for example to ensure that the block orders are preserved (dashed lines in Fig. 8). Then, standard

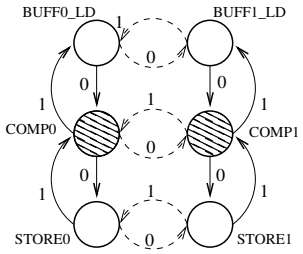


Figure 8: Dependence graph

software pipelining techniques [23] can be applied to define a periodic schedule for the particular instance to solve. This periodic schedule defines a sequential order for each resource, in particular for the communication medium. When this order is not already structurally ensured, a synchronization is added. For example, if the selected schedule placed `BUFF0_LD(i)` just before `BUFF1_LD(i)`, a synchronization is added, thanks to a FIFO `BUFF0_BUFF1`, as in Fig. 7.

This model makes the search for a good schedule more systematic. For example, to avoid the gap mentioned in Section 4.2 when `COMP0(i)` delays `STORE0(i)`, it is possible to find a solution with no overlap between `STORE0(i-1)` and `COMP0(i)`, thus no extra memory duplication, as the (non intuitive) software pipeline of Fig. 9 shows.

4.4 Automation of the process

Sections 4.2 and 4.3 defined the synchronization mechanisms that enable to pipeline communications by blocks, avoiding both the row change penalty, due to the `DDR` specification, and the data fetch penalty, due to the way C2H schedules nested loops. The latter can be considered as a limitation of C2H but, actually, this is also what makes the synchronization at C level possible. These synchronized accelerators can now be considered as 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. They identify blocks of computations, prefetch the required data from the `DDR`, store

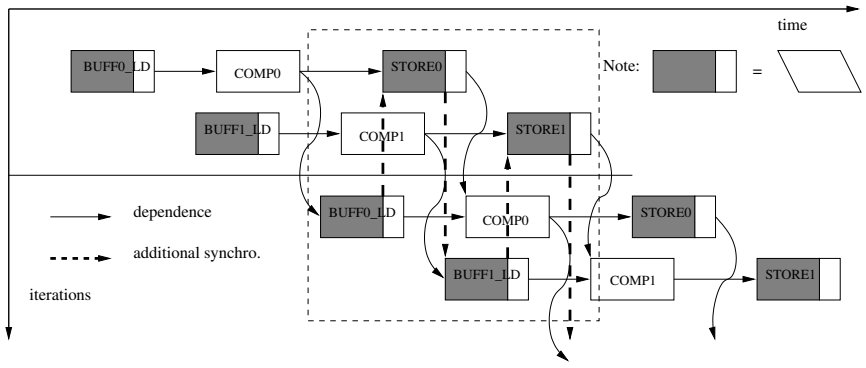


Figure 9: Software pipeline

them locally in the accelerator, and transfer results to the DDR, in other words, i.e., they fill the communication and computation templates defined in Section 4.2 with the adequate codes. These tasks are common in high-performance computing (HPC), in particular in compilers for languages such as High Performance Fortran (HPF). They require both program analysis and code transformations, in particular optimizations based on polyhedral techniques. We list here the main steps that are necessary, as well as related references.

Loop tiling and partial unrolling First, loop tiling [27] (or, for a single loop, strip-mining) divides the kernel into elementary blocks of computations to be executed with a double buffering scheme. For the matrix-matrix product, this corresponds to a block-based version. The double buffering scheme is then performed along the last loop describing tiles (loop unrolling by 2). A good tiling should reduce the volume of necessary data transfers, as well as the sizes of the local memories needed (memory footprint optimization), while trying to leave some data in local memory from one tile to another (temporal reuse). Many approaches exists in the HPC community to compute such a tiling (see references in [27]), even for non perfectly-nested loops [4].

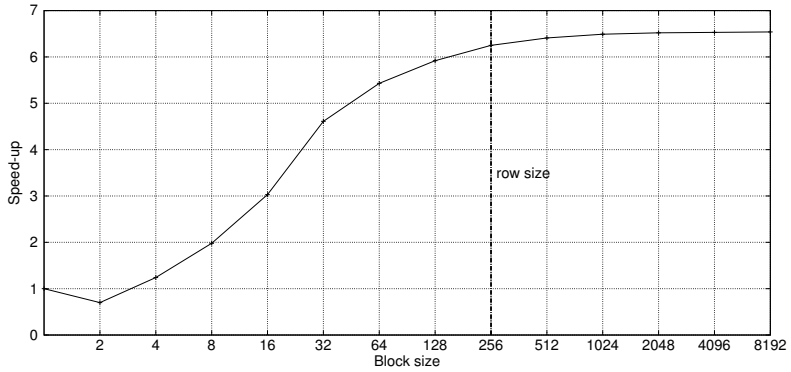
Communication coalescing The second step is to identify, for a given tile, the data to read from and to write to the DDR, excluding those already stored locally or that will be overwritten before their final transfer to the DDR (as the array c in the matrix-matrix product of Section 3.2). This is a particular form of communication coalescing as described in [8], for HPF, to host communications outside loops (here the loops describing one tile). Then, the set of transferred data is scanned [22, 5], preferably row by row. Even if an array is accessed by column in a tile, as for the matrix-matrix product, the corresponding data can be transferred by row.

Contraction of local arrays Then, a mapping function must be defined that convert indices of the global array (in the DDR) to local indices of a smaller array in which the transferred data are stored. Standard lifetime analysis and array contraction techniques [11, 16, 3] can be used for that.

Code specialization The final transformation is to replace nested loops that scan data sets or that define the computations in a tile by a linearization, as in the juggling code of Section 4.1. This, again, is to avoid any data fetch penalty. Also, once the different accelerators are defined, a coarse-grain software pipeline is determined, as explained in Section 4.3, and synchronizations (`fifo_read` or `fifo_write`) are placed as explained in Section 4.2.

5 Conclusion

We focused on the optimization of 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. 10 – here, the vector sum synthesized on Altera Stratix II EP2S180F1508C3 device, for vectors of size 16K – is typical of the results



	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 10: Speedups and resource usage

that we obtain: a small initial degradation due to the extra management with FIFOs, an increasing speed-up as the row change penalty becomes less frequent, and a plateau once the row size of the DDR is reached.

Considering high-level loop transformations to improve the performance and memory usage of hardware accelerators [18, 21, 12] is not new. 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 [19, 25]. We believe such a study is the necessary step to be able to consider HLS tools as back-end optimizers for source-to-source optimizers.

References

[1] Double data rate (DDR) SDRAM specification JESD79F. `http://www.jedec.org/download/search/JESD79F.pdf`.

[2] Handel-C language reference manual. `http://www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf`.

[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 SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, volume 42, pages 73–82, San Diego, USA, June 2007.

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM International Conference on Programming Languages Design and Implementation (PLDI'08)*, pages 101–113, Tucson, Arizona, June 2008.

[5] Pierre Boulet and Paul Feautrier. Scanning polyhedra without do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, Washington, DC, 1998.

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

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

[8] D. Chavarria-Miranda and J. Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 14–25, Chicago, IL, USA, 2005.

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

[10] Alain Darte, Rob Schreiber, Bob Ramakrishna Rau, and Frï¿½dï¿½ric Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems*, 7(1):159–172, 2002.

[11] Eddy De Greef, Francky Catthoor, and Hugo De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23:1811–1837, 1997.

[12] Harald Devos, Jan Van Campenhout, and Dirk Stroobandt. Building an application-specific memory hierarchy on FPGA. In *Proceedings of the 2nd HiPEAC Workshop on Reconfigurable Computing*, pages 53–62, Göteborg, 1 2008.

[13] Antoine Fraboulet and Tanguy Risset. Master interface for on-chip hardware accelerator burst communications. *Journal of VLSI Signal Processing*, 2(1):73–85, 2007.

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

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

[16] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.

[17] Nisc: No instruction-set computer (C-to-RTL). `http://www.ics.uci.edu/~nisc`.

[18] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Exploiting off-chip memory access modes in high-level synthesis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'97)*, pages 333–340, San Jose, California, United States, 1997.

[19] Joonseok Park and Pedro C. Diniz. Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines. In *ACM International Symposium on Systems Synthesis (ISSS'01)*, pages 221–226, 2001.

[20] Synfora Pico Express algorithmic synthesis in SoC. `http://www.synfora.com/products/picoexpress.html`.

[21] Alexandru Plesco and Tanguy Risset. Coupling loop transformations and high-level synthesis. In *SYMPosium en Architectures nouvelles de machines (SYMPA'08)*, Fribourg, Switzerland, February 2008.

[22] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[23] B. R. Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1):3–64, 1996.

[24] Spark: A parallelizing approach to the high-level synthesis of digital circuits. `http://mesl.ucsd.edu/spark`.

[25] Greg Stitt, Gaurav Chaudhari, and James Coole. Traversal caches: A first step towards FPGA acceleration of pointer-based data structures. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS'08)*, pages 61–66, Atlanta, GA, USA, 2008.

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

[27] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

Contents

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399