POLITECNICO
MILANO 1863

Exploiting Vectorization in High Level Synthesis of Nested Irregular Loops

Marco Lattuada, Fabrizio Ferrandi

Marco Lattuada and Fabrizio Ferrandi. Exploiting vectorization in high level synthesis of nested irregular loops. *Journal of Systems Architecture*, 75:1 – 14, 2017

The final publication is available via http://dx.doi.org/http://dx.doi.org/10.1016/j.sysarc.2017.03.001

Exploiting Vectorization in High Level Synthesis of Nested Irregular Loops

Marco Lattuada, Fabrizio Ferrandi

Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioingegneria

Piazza Leonardo da Vinci 32, 20133 - Milano - Italy

Abstract

Synthesis of DoAll loops is a key aspect of High Level Synthesis since they allow to easily exploit the potential parallelism provided by programmable devices. This type of parallelism can be implemented in several ways: by duplicating the implementation of body loop, by exploiting loop pipelining or by applying vectorization.

In this paper a methodology for the synthesis of nested irregular DoAll loops based on outer vectorization is proposed. The methodology transforms the intermediate representation of the DoAll loop to introduce vectorization and it can be easily integrated in existing state of the art High Level Synthesis flows since does not require any modification in the rest of the flow. Vectorization is not limited to perfectly nested countable loops: conditional constructs and loops with variable number of iterations are supported. Experimental results on parallel benchmarks show that the generated parallel accelerators have significant speed-up with limited penalties in terms of resource usage and frequency decrement.

Keywords: High Level Synthesis, Vectorization, Code Transformations

1. Introduction

Heterogeneous multiprocessor systems are becoming very common in a large group of embedded system application fields because their heterogeneity allows to efficiently execute tasks with different characteristics. The parts of the application which are characterized by high degree of parallelism are good candidates to be mapped on programmable hardware devices like Field Programmable Gate Arrays (FPGA) since their hardware implementation can potentially have very significant speed-up with respect to software implementation. Design by hand efficient hardware implementations can be a hard task since requires the knowledge of hardware description languages which is typically a rare expertise. To

Email address: {marco.lattuada},{fabrizio.ferrandi}@polimi.it (Marco Lattuada, Fabrizio Ferrandi)

overcome or at least to mitigate this issue, High Level Synthesis [1] has been introduced: it consists of a (semi)-automatic design flow, potentially composed of several methodologies, that starting from a high level representation of a specification (e.g., a C/C++ source code implementation) produces its hardware implementation.

Loop parallelization is one of the most used techniques exploited by High Level Synthesis to take advantage of the parallelism provided by the hardware platforms. An important class of loops which are good candidates to be parallelized are *DoAll loops* [2]. These loops are characterized by the absence of inter-iteration dependences which allows completely independent execution of different iterations. A parallel hardware implementation of this type of loop can be obtained by replicating multiple times the module implementing its body. This type of approach potentially provides good results in terms of performance, but it can significantly increase the resources usage. Moreover, the obtained speed-up can be partially reduced by the concurrent accesses to shared resources (e.g., shared memory) performed by the different module replicas. The contention resolutions can indeed introduce overhead both in terms of delay in critical path (e.g., for the presence of the arbiter) and of cycles (e.g., because of the stalls introduced during resources acquisition).

This paper proposes a methodology for High Level Synthesis of DoAll loops based on vectorization [3] (i.e., introduction of functional units processing vectors of data) to mitigate these problems. This type of approach is the fundaments of the architecture of Graphic Processing Units [4] but it can be adopted also for General Purpose Processors [5]). Applying this type of parallelization in High Level Synthesis has been proposed in [6], but with significant limitations to its applicability. In this paper an extension of the methodology is proposed which removes all the constraints allowing to apply vectorization to *DoAll loops* with arbitrary structure. The methodology does not introduce any significant change to the structure of the Finite State Machine nor to the hardware accelerator interface, so it can be easily integrated in existing High Level Synthesis design flows provided that they already support synthesis of vector operations. Its main contributions are the following:

- It extends the applicability of vectorization in High Level Synthesis by allowing vectorization of arbitrary *DoAll loops* containing nested loops with arbitrary number of iterations and presence of conditional constructs which provoke control divergences.
- It does not require any change to the rest of the High Level Synthesis flow and it can be applied even if the rest of the High Level Synthesis flow does not support synthesis of vector instructions at all.

The rest of the paper is organized as follows. Section 2 presents related work while Section 3 presents the example on which the proposed methodology will be applied and introduces some preliminary definitions. Section 4 describes the proposed methodology whose experimental results are presented in Section 5. Finally Section 6 presents the conclusions of the paper.

2. Related Work

Synthesis of *DoAll loops* is a very well studied topic of High Level Synthesis so that many approaches have been proposed to address this problem. Identification of this type of loops can be performed by means of Polyhedral methodologies, which allow to analyze and transform source code specifications exposing the different possibilities of parallelizing a loop. An example of framework aiming at performing such type of transformations is presented in [2]: this framework is able to systematically identify effective access patterns and to apply both inter- and intra- block optimizations, exposing several types of possible parallelization. The framework then evaluates each of them, and when estimated it as profitable, applies it to the specification source code.

Despite completeness of existing frameworks and methodologies for polyhedral analysis, this type of techniques is still limited to loops with limited irregularity in their structure. For this reason, most of the recent synthesis techniques for *DoAll loops* start from applications where parallelism has already been identified. Papakonstantinou et al. [7] proposed the automatic synthesis of applications written with CUDA programming model. The proposed approach adopts *FCUDA*, a design flow which translates the CUDA code into task-level parallel C code. This code is then provided as input to AutoPilot which performs the actual synthesis producing a multi accelerators system. In a similar way, Choi et al. [8] proposed the automatic synthesis of applications already parallelized, but they start from applications exploiting pthreads and OpenMP API. In this case, the methodology directly produces parallel hardware implementations of the loops which have been annotated with `#pragma omp for` (they are *DoAll loops* with compile time known number of iterations). The parallel architecture is obtained by replicating multiple times the hardware accelerator which implements the body loop. This approach implies to replicate multiple times the whole implementation of the loop and requires a processor to synchronize the execution of the accelerators, with a significant increase of resources usage. Castellana et al. [9] proposed an High Level Synthesis design flow which address the parallelization of parallel loop implementing RDF queries by exploiting a distributed controller which manages the execution of the different iterations. In this case, the memory accesses are managed by a memory interface controller which provides dynamic routing of memory accesses and conflicts management. Tan et al. [10] instead presented an architecture for parallelizing the execution of pipelined loops which are a superclass of the *DoAll loops*. In this type of applications, nested irregular loops (i.e., nested loops with variable number of iterations) can introduce delays because of control divergence. To mitigate this issue, some stages of the pipeline architecture are selectively replicated, reducing the stalls to be introduced to maintain the pipeline synchronized. Another approach to solve this type of problems (i.e., the automatic synthesis of OpenMP annotated applications) was proposed in [11] but targeting heterogeneous systems implemented onto FPGAs. All these approaches, since the different accelerator replicas potentially access at the same time to external data, require to add logic to control resources contention, potentially delaying

requests performed by the single accelerators and limiting the scalability of the approach.

Parallelization of complex *DoAll loops* (i.e., outer loops) by means of vectorization was proposed for SIMD processors [3]: loops are vectorized during compilation for SIMD architectures, even if they contain other loops or conditional constructs, provided that some conditions are met. In particular, the outer and the inner loops must be countable and all the conditional constructs must be removable by means of if-conversion. Moreover, ad-hoc analyses and transformations are applied trying to maximize the number of aligned accesses. Karrenberg and Hack instead proposed a methodology [5] for the vectorization of arbitrary code. The methodology is based on the transformations of the Control Flow Graph as the methodology proposed in this paper and the control divergence is mainly removed by removing the conditional constructs and by speculating instructions, potentially increasing the latency of the iterations of the vectorized loops. On the contrary, in the methodology proposed in this paper, the control divergence is addressed by exploiting conditional assignments but trying to preserve the nesting relationships among conditional constructs without increasing the iteration loop latency.

Vectorization is heavily exploited in Graphical Processing Units [4] whose architectures are based on this paradigm. However, even if they contain dedicated components to efficiently support control divergence, this can still be a significant issue for some applications so that compiling optimization techniques (e.g., [12]) have been proposed to mitigate the problem. The architecture generated with the proposed technique in this paper does not exploit any special components to manage the control divergence, so that they do not require further modifications of the rest of the High Level Synthesis flow. Moreover, the code transformations presented in [5] and in [12] and aimed at reducing the control divergences, for example by changing how parallel iterations are grouped in vectors, can be easily integrated in the proposed design flow. Both the vectorization implemented on SIMD processors and on Graphical Processing Units have the same limitation: the degree of parallelism is fixed. Implementing solutions with a smaller degree of parallelism can only provide potential advantages in terms of power consumption. On the contrary, because of their programmabilty, FPGAs allow to implement vector architectures with an arbitrary degree of parallelism.

Finally, the effects of using vector functional units in High Level Synthesis have already been evaluated in [13]. The authors proposed the adoption of configurable vector functional units which can implement at the same time both scalar operations and vector operations. This approach produces better solutions both in terms of performances and power consumption, showing the effectiveness of using parallel functional units, but it is limited to the parallelization of some operations of the specification.

```
#pragma omp parallel for
for(i = 0; i < RowsNumb; i++) {



  local = 0;
  rl = RowLength[i];
  if(rl){

    j=0;
    while(j < rl); {




      local += Matrix[i][j];

      j++;
    }

  }
  else {
    if(old_row[i]>0) {


       local = old_sum[i];

    }

  }
  avg[i] = local/rl;



}
```

```
BB1    1   OuterLoop:
       2     i_2 = PHI<0, i_28>;
       3     c_3 = i_2 < RowsNumb;
       4     if(c_3){
BB2    5       local_5 = 0;
       6       rl_6 = RowLength[i_2];
       7       c_7 = rl_6 != 0;
       8       if(c_7) {
BB3    9         j_9 = 0;
BB4   10         InnerLoop:
      11           j_11 = PHI<j_9,j_18>
      12           local_12 = PHI<local_5, local_16>
      13           c_13 = j_11 < rl_6;
      14           if(c_13) {
BB5   15             t_15 = Matrix[i_2][j_11];
      16             local_16 = local_12 + t_15;
      17             j_17 = j_11 + 1;
      18             goto InnerLoop;
                   }
                 }
                 else {
BB6   19           t_19 = old_row[i_2];
      20           c_20 = t_19 > 0;
      21           if(c_20){
BB7   22              t_22 = old_sum[i_2];
      23              local_23 = t_22;
                   }
BB8   24           local_24 = PHI<local_5, local_23>;
                 }
BB9   25         local_25 = PHI<local_12, local_24>;
      26         t_26 = local_25/rl_6;
      27         avg[i_2] = t_26;
      28         i_28 = i_i2 + 1;
      29         goto OuterLoop;
               }
BB10           /* After the loop*/
```

Figure 1: Example of parallel loop and corresponding compiler intermediate representation.

3. Preliminaries

This section presents the code of the parallel loop which will be used in the rest of the paper to show an example of application of the proposed methodology and some preliminary definitions which are used in the description of the proposed methodology. The code of the parallel loop is shown in Figure 1: its number of iterations is not fixed (`NumRows` is a variable) and it contains a nested loop whose number of iterations is also variable (`RowLength(i)`) and depends on the particular iteration `i` of the outer loop.

The methodology does not work directly on the source code of the applications to be synthesized but on an intermediate representation which must satisfies the following properties:

- it must adhere to the Static Single Assignment (SSA) Form [14]: each scalar variable must be assigned only once in the code.
- complex instructions must be decomposed in a sequence of simpler instructions.

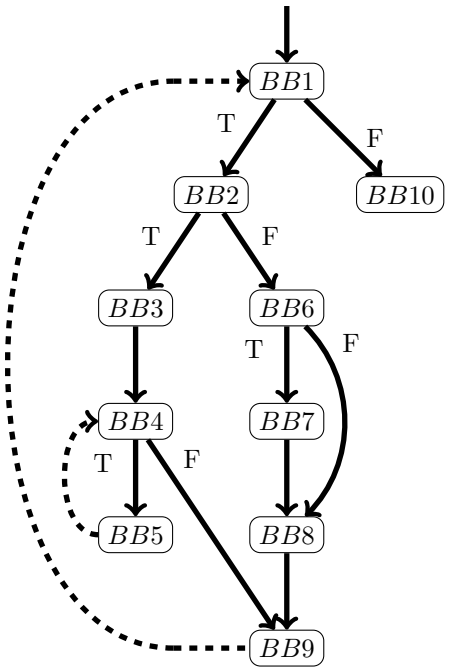


Figure 2: The Control Flow Graph of loop of example of Figure 1. The feedback edges are dashed.

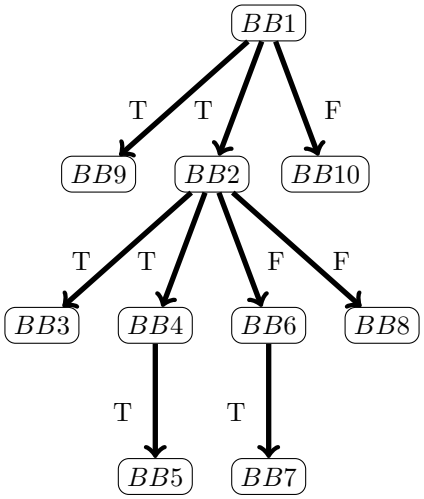


Figure 3: The Control Dependence Graph of loop of example of Figure 1.

The right part of Figure 1 reports the intermediate representation of the considered example. The intermediate representation is in SSA form: for example, the variable `local` has been replaced with `local_5`, `local_12`, `local_16`, `local_23`, `local_24`, and `local_25` each of which is assigned by only one instruction. The `for` and the `while` instructions have been expanded in combination of `if`, `goto`, and `label`. Finally complex instructions which cannot be directly synthesized on a single functional unit have been split into multiple instructions. For example instruction `avg[i]` has been expanded in a division (`t_26 = local_25/rl_6`) and in a store (`avg[i_2] = t_26`).

The code just presented will be used in the following sections to show the application of the proposed methodology, while in this section is exploited to show which kind of information can be extracted from the intermediate representation of a code. In particular, the information is:

- *Control Flow Graph* (CFG) [15]: a directed graph $CFG = (BBs, E_{CFG})$, which is an abstract representation of the paths (i.e., the sequences of branches) that might be traversed during the execution of the code. A Control Flow Graph is said to be *Structured* [16] if the corresponding program can be written using only simple condition branches (`if`) and single exit loop (`while` without `break`) without any jump to arbitrary position (`goto`). The Control Flow Graph of Figure 1, which is shown in Figure 2, is structured.

- *Feedback Edges* [17]: the edges which close cycles in Control Flow Graph. In the example $BB5 - BB4$ and $BB9 - BB1$ are feedback edges. For the sake of simplicity in the rest of the paper only reducible loops are considered (i.e., loops with a single entry point).

- *Headers*: the entry points of reducible loops. In the example, the headers are $BB1$ and $BB4$.

- *Landing Pads*: the destination basic blocks of branches which exit from loops. In the example they are $BB9$ and $BB10$.

- *OutDegree(BB_v)*: the number of outgoing edges of vertex BB_v. In the example the OutDegree of $BB1$, $BB2$, $BB4$, and $BB6$ is 2 while the OutDegree of all the remaining basic blocks is 1.

- *Control Dependence Graph* (CDG) [18], a directed graph $CDG = (BBs, E_{CDG})$, which represents the control dependences of the basic blocks. For example $BB6$ controls $BB7$.

- *Controller(BB_v)*: the BB_u which controls the execution of BB_v, i.e., (BB_u, BB_v) $\in E_{CDG}$. In a structured CFG the controller of each basic block is unique. For example the Controller of $BB7$ is $BB6$.

- *Condition(BB_v)*: if BB_b ends with a conditional construct, the condition of it. For example *Condition($BB2$)* is `c_7`.

- *EndIf(BB_v)*: given a BB_u ending with a conditional construct and which is not a loop Header, the immediate post dominator [15] of BB_u, i.e., the first basic block in which all the paths crossing BB_v reconverge. In the example *EndIf($BB2$)* is $BB9$ and *EndIF($BB6$)* is $BB8$.

- *Then(BB_v)*: given a basic block BB_v which ends with a conditional construct `if(cond)`, the successor of BB_v when `cond` is evaluated to true. In the example *Then($BB1$)* is $BB2$, *Then($BB2$)* is $BB3$, *Then($BB4$)* is $BB5$, and *Then($BB6$)* is $BB7$.

- *Else(BB_v)*: given a basic block BB_v which ends with a conditional construct `if(cond)`, the successor of BB_v when `cond` is evaluated to false. In the example *Else($BB1$)* is $BB10$, *Else($BB2$)* is $BB6$, *Else($BB4$)* is $BB9$, and *Else($BB6$)* is $BB8$.

- *ThenBBs(BB_v)*: given the basic blocks BB_v, the set of BB_i such that there is a path from *ThenBB(BB_v)* to BB_i in CDG (i.e., the set of basic blocks controlled by *Then(BB_v)*. *Then(BB_b)* is considered part of this set. For example *ThenBBs($BB2$)* is $BB3$, $BB4$, and $BB5$.

4. Proposed Methodology Flow

The proposed methodology aims at synthesizing a parallel hardware accelerator by means of outer loop vectorization integrated in High Level Synthesis. A fixed number P of iterations of the loop is coupled and merged so that the execution of an iteration of the transformed loop corresponds to the execution of P iterations of the original loop. Each copy will be executed sequentially, but the different copies will be executed in parallel and in a completely synchronized way. P identifies the degree of introduced parallelism: different loops can be parallelized with different degrees of parallelism and different implementations of the same loop can be obtained by varying its degree of parallelism.

The vectorization is obtained by modifying the intermediate representation of the specification during the High Level Synthesis flow. The methodology only exploits the possibility of a High Level Synthesis flow of synthesizing vector variables and vector functional units: differently from GPU implementation [19] special hardware support to control the vector execution is not required. The scheduling of the memory accesses is fully deterministic and it is completely computed at this time. For this reason, the proposed methodology does not require complex memory controller to arbitrate the parallel memory accesses as in [8], [10], [9]. Moreover, in principle it is possible to exploit the proposed methodology even if the rest of the flow does not support vectorization at all.

Differently from the methodology presented in[6], the only requirement is that the loop to be optimized must be a DoAll loop (i.e., all iterations can be executed in parallel). No further preconditions about the loop are required: the number of its iterations can be not multiple of the degree of parallelism and it can contain conditional constructs and irregular loops (i.e., loops with variable number of iterations). Allowing the presence of conditional constructs in the parallel

loop implies to allow control divergences: the different elements of the vector execution can require the execution of different basic blocks which are in mutual exclusion. For example, in the code of Figure 1, if P is 2, `RowLength[0] == 0`, and `RowLength[1] != 0`, the execution of the first two iterations, which are merged by vectorization, diverges since it requires to execute `BB3` and `BB6` which are in mutual exclusions. In the following, it will be shown how the intermediate representation is modified to address this issue.

The proposed methodology flow is composed of these steps:

1. *Generation of Intermediate Representation*: the intermediate representation which will be manipulated by the proposed methodology is generated.
2. *DoAll loop analysis*: the intermediate representation is analyzed to identify *DoAll* loops.
3. *Control Flow Graph Linearization*: the *Control Flow Graph* [15] of the loops to be parallelized is transformed to remove mutual exclusions in execution of basic blocks.
4. *Guard Conditions Computation*: for each basic block of *DoAll loops*, the *Guard Condition* associated with it is computed.
5. *Instructions Predication*: each instruction which cannot be speculated is transformed in a predicated instruction.
6. *Conditional Assignments Insertion*: the code to correctly manage data in case of control divergence is added.
7. *Instructions classification*: each instruction of the loops is analyzed to identify if it is a control construct and, if not, if it has to be transformed in a vector instruction or in a set of scalar instructions.
8. *Instructions transformation*: instructions which control the execution of loops are transformed to support parallel execution of iterations; other instructions are transformed in vector instructions or in sets of scalar instructions according to how they have been classified.
9. *Synthesis*: the transformed loops are synthesized by means of High Level Synthesis flow.

In the following each of these steps will be detailed and its application to the example of Figure 1 will be shown.

4.1. Generation of Intermediate Representation

The intermediate representation which will be exploited in the design flow is generated. The intermediate representation must have the properties which have been described in Section 3: SSA form and absence of complex instructions. Note that most of the state of the art High Level Synthesis tools (e.g., [8], [9], [20]) already adopt an internal intermediate representation which satisfies these requirements.

4.2. DoAll loop analysis

The intermediate representation is analyzed to identify *DoAll loops*. How this analysis is performed is out of the scope of this paper: all state of the art

techniques such as polyhedral analyses [21] can be exploited. However, since not all the *DoAll loops* can be actually identified by static analyses, loops which have to be parallelized by means of vectorization can be directly annotated by the designer with annotations like OpenMP pragma `simd` [22].

The outmost loop of Figure 1 has been annotated with OpenMP pragma `simd` to be synthesized with the proposed methodology.

4.3. Control Flow Graph Linearization

The presence of basic blocks whose execution is in mutual exclusion potentially prevents the vectorization of a loop since the destination of a branch for different parallel iterations grouped in the same vector can be different. To overcome this issue, the Control Flow Graph is linearized ([5], [12]): the graph is transformed in such a way that basic blocks which were in mutual exclusions are now connected by a directed path. For example, the *then* and the *else* blocks of a `if` instruction are executed in sequence.

Before applying the linearization, the Control Flow Graph is transformed to make it structured. Note that the Control Flow Graph of an intermediate representation can be unstructured even if the starting code is structured. Some of the possible causes are the presence of short-circuit conditions and the compiler optimizations. If the intermediate representation contains unstructured Control Flow subgraphs, they are transformed by applying the technique described in [12]: the basic blocks of the unstructured subgraph are topological sorted and then transformed in a sequence of basic blocks. To guarantee that only the correct subset of these basic blocks is executed, a new basic block is associated with each of them to control their execution. Structured subgraphs contained in unstructured subgraphs are not modified in this phase, but they are considered as they were atomic nodes.

After that the starting Control Flow Graph has been transformed in a structured graph, linearization can be applied. Differently from techniques presented in [5] and [12], the linearization applied in this methodology flow does not flatten all the basic blocks of a loop in a single sequence. On the contrary, the hierarchy of nested conditional constructs is preserved: linearization is only locally applied to group of subgraphs controlled by the same basic block.

The details of how linearization is performed are described by Algorithm 1. For the sake of simplicity it is assumed that the `then` block of a conditional construct is not empty. If a `then` basic block does not satisfy this condition, `then` and `else` are switched and the condition of the conditional construct is negated.

All the basic blocks of the Control Flow Graph (line 1) are analyzed looking for basic blocks ending with conditional constructs (line 2). Conditional constructs which determine the exit from the loops (line 5) have to be ignored as well as conditional constructs without `else` block (line 11). On the contrary subgraphs corresponding to `if(cond) ThenBB else ElseBB` has to be restructured as `if(cond) ThenBB if(not cond) ElseBB`. This is obtained by removing the edges `BB-ElseBB` and the edges from the *ThenBBs(BB)* subgraph to `EndIfBB`

Algorithm 1: Linearization

```
   input: CFG
 1 foreach BB ∈ BBs do
 2 |   if OutDegree(BB) < 2 then
 3 |   |   Continue
 4 |   end
 5 |   if IsExitLoop(BB) then
 6 |   |   Continue
 7 |   end
 8 |   ThenBB = Then(BB)
 9 |   ElseBB = Else(BB)
10 |   EndIfBB = EndIf(BB)
11 |   if ElseBB == EndIfBB then
12 |   |   Continue
13 |   end
14 |   newBB = CFG.AddNegBlock(BB)
15 |   CFG.AddEdge(BB, newBB)
16 |   CFG.RemoveEdge(BB, ElseBB)
17 |   CFG.AddEdge(newBB, EndIfBB)
18 |   CFG.AddEdge(newBB, ElseBB)
19 |   foreach SourceBB ∈ InEdges(EndIfBB) do
20 |   |   if SourceBB ∈ ThenBB(BB) then
21 |   |   |   CFG.RemoveEdge(SourceBB, EndIfBB)
22 |   |   |   CFG.AddEdge(SourceBB, NewBB)
23 |   |   end
24 |   end
25 |   FixPhis(newBB)
26 |   FixPhis(EndIfBB)
27 end
```

(lines 16 and 21). Then the newly added basic block (i.e., `newBB` - the basic block containing `if(not cond)`) has to be inserted (line 14) and connected with BB (line 15) and the last basic blocks of the *ThenBBs(BB)*. Finally the PHIs of `newBB` and of `EndIfBB` has to be fixed to update the reaching definitions from the different basic blocks. At end a structured Control Flow Graph without mutual exclusions is obtained.

A further transformation is performed on the Control Flow Graph to simplify the *Conditional Assignments Insertion* step. If a landing pad of a loop has more than one incoming edge, an empty basic block is inserted between the exit of the loop and the landing pad. This basic block becomes the new landing pad of the loop.

Figure 4 shows the linearized Control Flow Graph of the example of Figure 1 while Figure 5 shows the corresponding intermediate representation. $BB11$

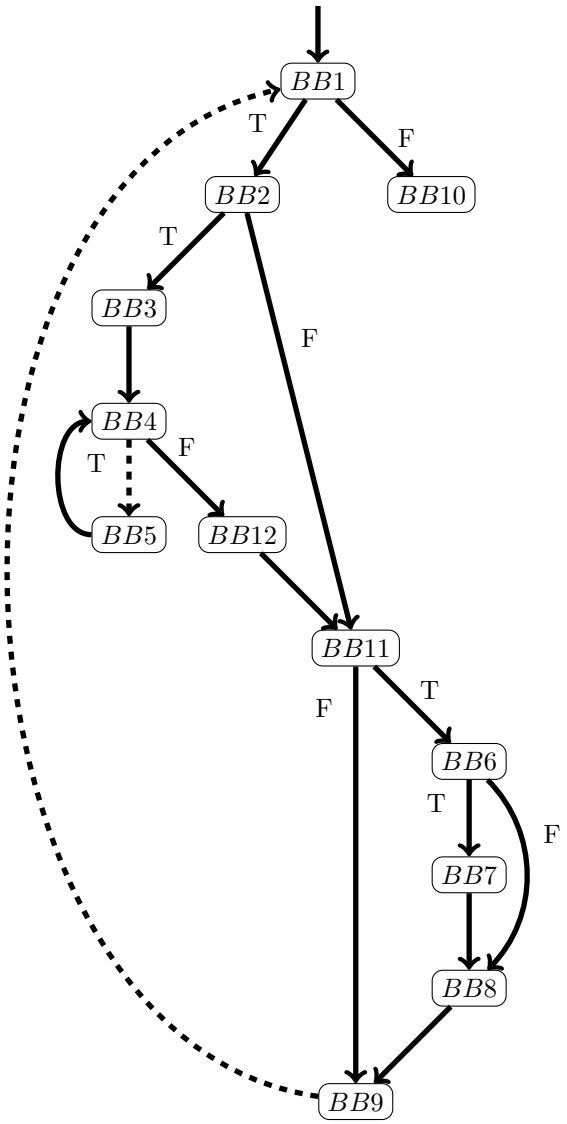


Figure 4: The linearized Control Flow Graph of the example of Figure 1

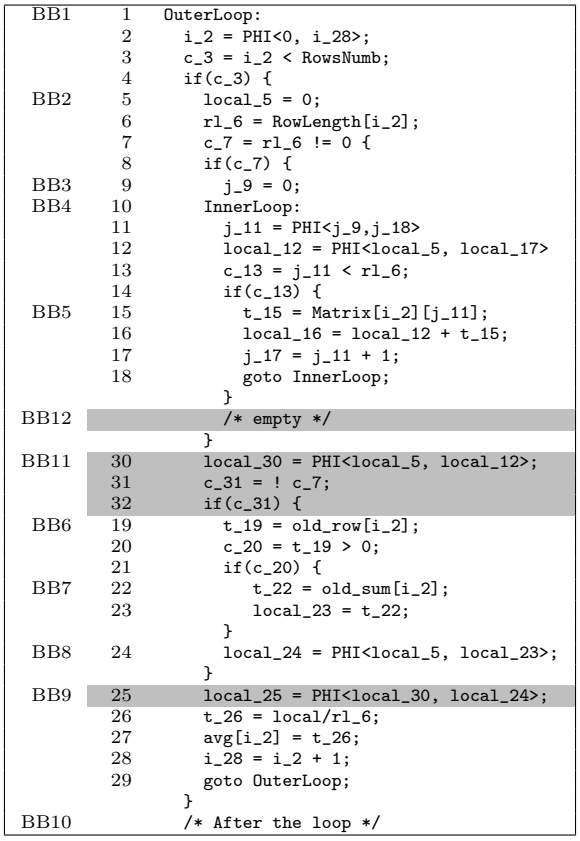


Figure 5: The intermediate representation of the example of Figure 1 after linearization. The added and modified instructions are highlighted.

$g(BB1)$ =PHI<true,$g(BB9)$>	$g(BB7) = g(BB6) \wedge$ c_20
$g(BB2) = g(BB1) \wedge$ c_3	$g(BB8) = g(BB11) \wedge$ c_31
$g(BB3) = g(BB2) \wedge$ c_6	$g(BB9) = g(BB1) \wedge$ c_3
$g(BB4)$ =PHI< ($g(BB_3) \wedge$c_7,$g(BB5)$>	$g(BB11) = g(BB1) \wedge$ c_3
$g(BB5) = g(BB4) \wedge$ c_13	$g(BB12) = g(BB2) \wedge$ c_7
$g(BB6) = g(BB11) \wedge$ c_31	

Table 1: Guards of the basic block of example of Figure 5.

has been added and instruction 32, which is a conditional construct with the opposite condition of instruction 8, has been inserted into it. Moreover PHI 30 has been added and PHI 25 has been fixed to correctly manage the propagation of different definitions of `local`. Finally, since the new landing pad $BB11$ has two incoming edges, a new empty basic block $BB12$ is inserted between $BB4$ and $BB12$ becoming the new landing pad.

4.4. Guard Conditions Computation

In this step, a *Guard Condition* [23] for each basic block is computed. A *Guard Condition* is a boolean expression which specifies if the instructions of the corresponding basic block have to be executed or not. Differently from [23] and [5] the predicates are not exploited to completely remove control constructs from the parallel loop. These expression will indeed be used in the following steps as operand of predicated instructions and to correctly manage reaching definitions.

To compute the guards, the basic blocks of the *DoAll loop* and of the nested loops are analyzed in topological order. The guard $g(BB_i)$ of a basic block BB_i is:

- if BB_i is the header of the parallel loop, $g(BB_i)$=PHI<true, $g(BB_j)$> where BB_j is (are) the source(s) of the feedback edges of the loop.
- if BB_i is the header of a loop nested in the parallel loop, $g(BB_i$=PHI<cond, $g(BB_j)$> where BB_j is (are) the source of the feedback edges of the loop and cond $= g(Controller(BB_i)) \wedge Condition(Controller(BB_i))$.
- if BB_i is not an header, $g(BB_i) = g(Controller(BB_i) \wedge Condition(Controller(BB_i))$.

Note that the guards can be computed as described because the Control Flow Graph is structured and linearized. Computation of guards in case of arbitrary Control Flow Graphs is more complex.

The guards for the basic blocks of the code of Figure 5 are listed in Table 1. The intermediate representation of the example of Figure 1 enriched with guard computation is reported in Figure 6. The instructions for computing the guards have been reported without any optimizations for the sake of readability.

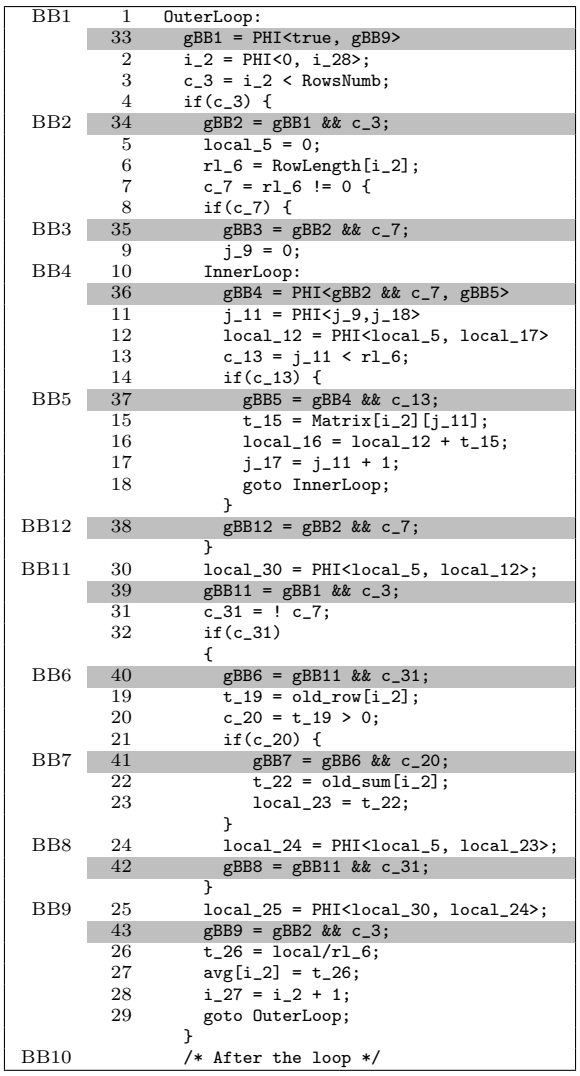


Figure 6: Intermediate representation of example of Figure 1 enriched with guards computation (highlighted instructions).

4.5. Instructions Predication

The vectorization of the basic blocks which have to be conditionally executed implies the execution of instances of instructions which would not be performed in the non vectorized code. While the effects of arithmetic instructions can be nullified by introducing conditional assignments as it will be shown in the next section, the effects of other types of instructions cannot be easily reverted. For this reason, the instructions with side effects (e.g., function calls and stores) are predicated: a boolean expression is associated with the instruction specifying if that instruction has to be actually executed or not. In particular, the predicate to be associated with an instruction is the guard condition computed in the previous section. This flag will be then transformed into a mask during the vectorization.

The instructions with the side effects are not the only to be predicated. Also load instructions for example can require to be predicated: the vectorization can introduce an access to a not allocated memory addresses potentially provoking stalling of the hardware implementation. Ad-hoc analysis techniques such as out of bound analyses can help in reducing the number of loads that have to be predicated, but this analysis is out of the scope of this paper. Note that in the proposed methodology the predication does not imply the removal of any conditional construct nor the speculation of the predicated instructions. The predication indeed is introduced to selectively prevent the execution of some elements of vector instructions because of the control divergences. In Section 4.9 it will be shown how the proposed methodology can be exploited even if the rest of the High Level Synthesis flow does not support synthesis of predicated instructions.

The intermediate representation of the example after this step of the methodology is reported in Figure 7: the predicated instructions are the store (i.e., instruction 27) and all the loads (i.e., instructions 6, 15, 19, 22). For example if P (degree of parallelism) is 2 and `RowsNumb` and the size of `RowLength` are even, during the last iteration of the vectorized loop instruction 6 will try to access to `RowLength[Rowsnumb-1]` and `RowLength[Rowsnumb]` but the second element is not allocated in memory. Since this can result in an infinite run of the generated circuit, since there is not any device memory replying to a memory request for that address, predication has been added for this type of instructions. On the contrary, the division operation (26) has not been transformed into a predicate instruction because usually modules implementing division produced by High Level Synthesis flows can execute division by 0 without blocking the computation, even if the produced values will be not significant.

4.6. Conditional Assignments Insertion

The presence of conditional constructs in the loop to be vectorized can cause control divergence between the different parallelized iterations which have been grouped together in the same vector (i.e., some iterations execute a given basic block while other iterations do not). This divergence is not an issue for predicated instructions, since only the actually required instances are executed, but

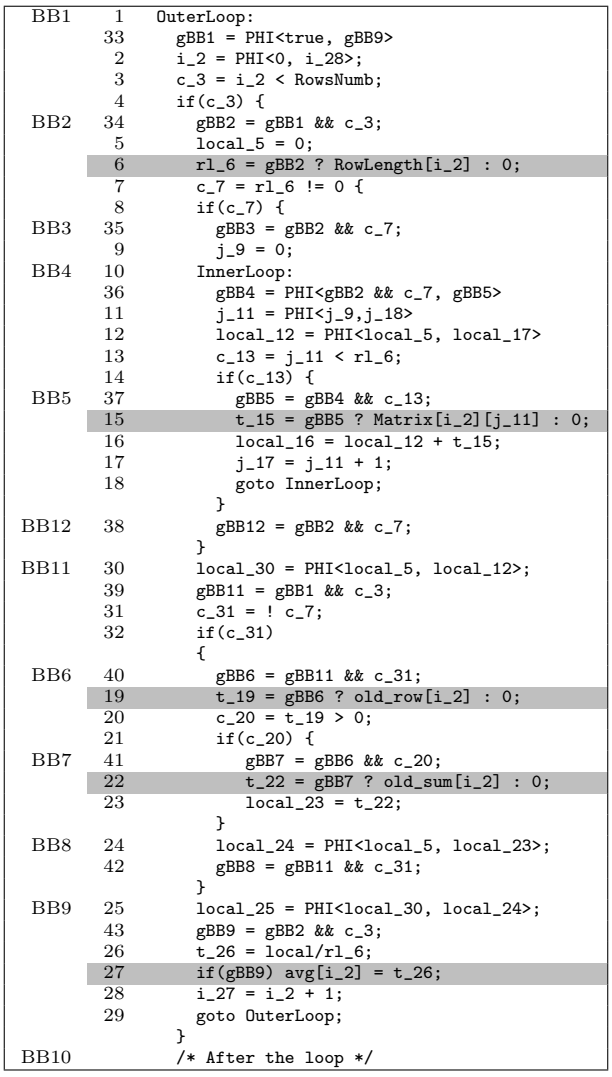


Figure 7: Intermediate representation of example of Figure 1 with highlighted predicated instructions.

requires to correctly manage the merging in reconvergence points of the data produced by non predicated instructions. In particular, given a basic block with more than one incoming edge, if this is a reconvergence point for a set of parallelized iterations (i.e., if the previously executed basic block is not the same for all the parallelized iterations), the vector PHIs contained would have to select the single elements from the different input vectors. For example, if $P = 2$ and $gBB6$ is 0 for the first iteration and 1 for the second iteration, the PHI 25 should take the first element from local_30 and the second element from local_24. Synthesizing such type of vector PHI would be possible, but it would require to modify part of the High Level Synthesis flow. A possible different solution is to replace the PHI with a conditional assignment, but in this case the intermediate representation would not be anymore in strict SSA form [24] (i.e., there would be a definition of a SSA which does not dominate its use). For example local_25 = PHI<local_29, local_24> could be replaced with local_25 = c_31 ? local_30 : local_24, but in this case the definition of local_30 (instruction 30) would not dominate its use in instruction 25. For this reason a different solution has been adopted. Given a piece of code with this structure:

```
/*BB_i*/
if(cond)
{
   /*BB_j*/
   gBB_j = gBB_i && cond;
   ssa_2 = ...
}
/*BB_k*/
ssa_0 = PHI<ssa_1, ssa_2>
```

where ssa_1 is the definition coming from BB_i and ssa_2 is the definition coming from BB_j, for each PHI of BB_k two transformations are performed. The first is the insertion of a conditional assignment at the end of BB_j: ssa_3 = gBB_j ? ssa_2 : ssa_1. Then the PHI is updated to use the new definition: ssa_0 = PHI<ssa_1, ssa_3>. The resulting code is finally:

```
/*BB_i*/
if(cond)
{
   /*BB_j*/
   gBB_j = gBB_i && cond;
   ssa_2 = ...
   ssa_3 = gBB_j ? ssa_2 : ssa_1;
}
/*BB_k*/
ssa_0 = PHI<ssa_1, ssa_3>
```

In this way the convergence has been implicitly anticipated in the conditional assignment added in BB_j, but this solution does not require special vector PHI

nor break the strict SSA form. The `PHI`s in loop headers have to be modified in a similar way with the exception of the `PHI` computing the guard conditionis which do not require changes. It is worth noting that the proposed methodology flow until this step does not yet introduce any vector instruction in the intermediate representation. All the steps performed up to now are preparatory for the actual vectorization which will be described in 4.8.

The intermediate representation of the example of Figure 1 produced after this step is shown in Figure 8. The conditional assignment which have been introduced are the instructions 45, 46, 47, 48 while the modified `PHI`s are the instructions 11, 30, 24, and 25.

4.7. Instruction Classification.

During this step of the methodology each instruction which is part of the *DoAll loop* or of a nested loop is classified into five different classes:

- *Vector instructions*: they will be transformed into vector instructions.
- *MultiScalar instructions*: they will be transformed into P scalar instructions.
- *DoAll loop instructions*: they are the instructions that have to be managed ad hoc in order to control the execution of the vectorized *DoAll loop*.
- *Control instructions*: they are the conditional constructs instructions included in the *DoAll loop*.
- *Scalar instructions*: they will not be transformed.

The reason for which the second class has been introduced depends on how a vector instruction can be implemented:

① *Single scalar unit*, i.e., a single scalar functional unit which executes P scalar operations in sequence; this is the worst solution in terms of clock cycles, but the best in terms of area.
② *Single pipeline unit*, i.e., a single pipeline functional unit which executes P scalar operations in pipelined way; for complex operations (i.e., operations which require more than one cycle) it provides good performances (better than ①) with a slight area increment.
③ *Multiple scalar units*, i.e., P scalar functional units which execute P scalar operations in parallel; this is the best solution in terms of clock cycles, but the worst in terms of area.
④ *Vector parallel unit*, i.e., a single vector functional unit; it provides the same performances of ③ but better area savings because of better resource sharing [25] [26] and smaller controller complexity [27].

If the second class of instructions was not introduced, all the data computation instructions would be synthesized as ④, producing the best solution in terms of performance, but the increment of area with respect to the non parallelized

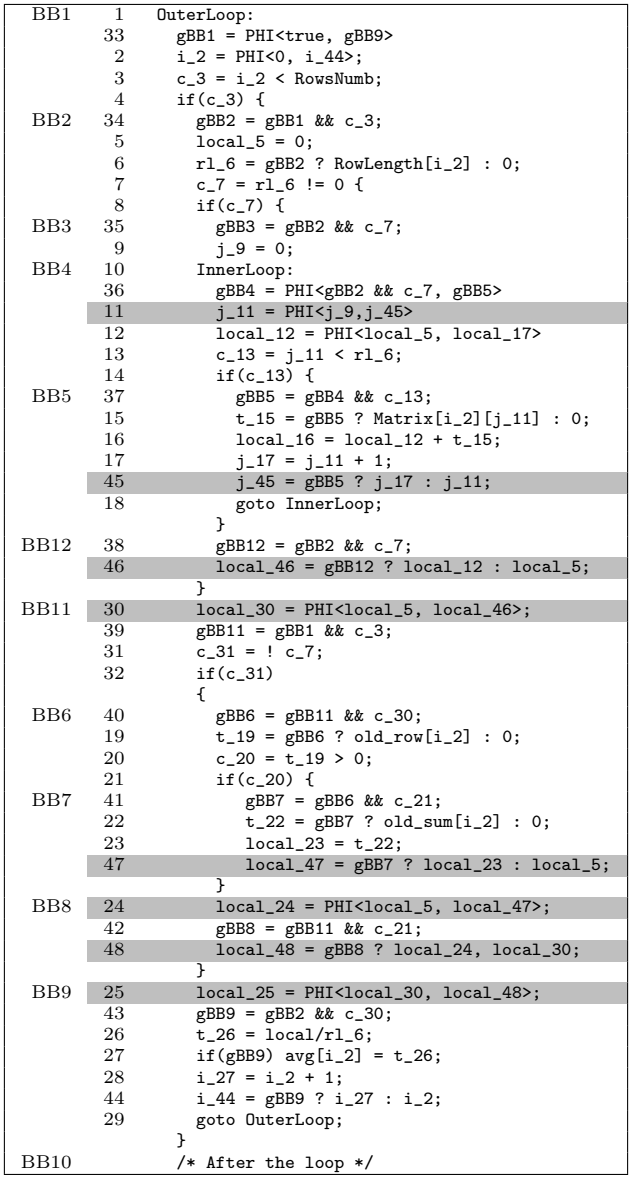
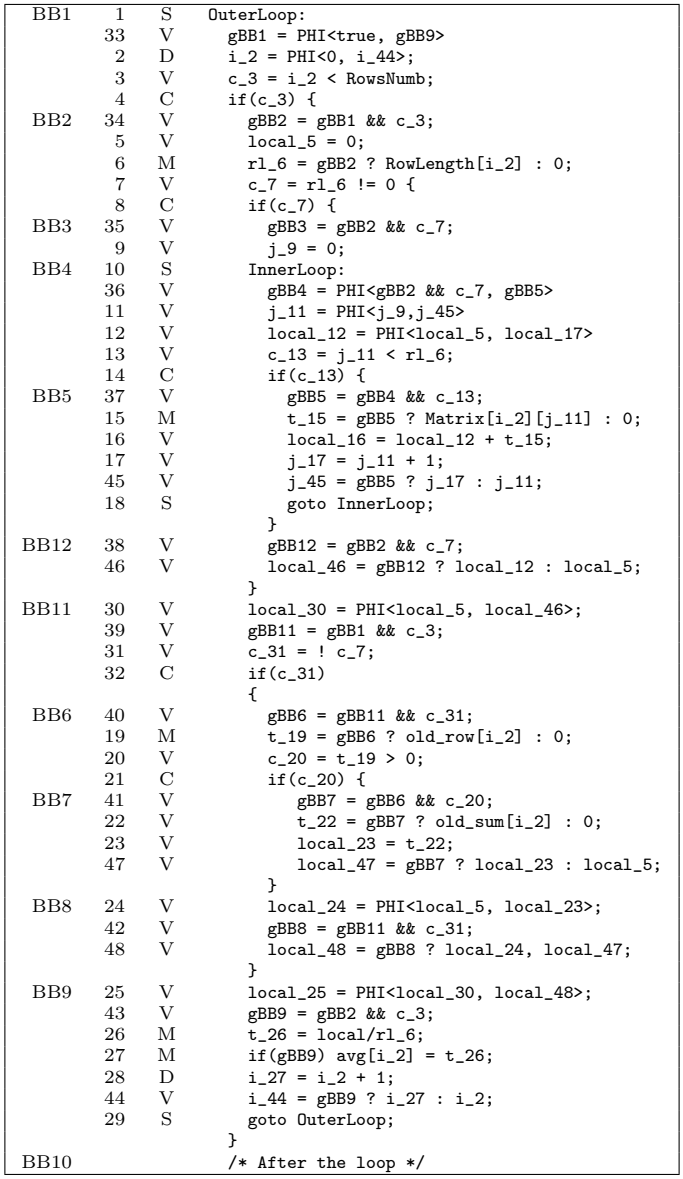


Figure 8: Intermediate representation of example of Figure 1 after conditional assignments insertion. Highlighted instructions are the changes with respect to the outcome of the previous step of the methodology.



BB1	1	S	OuterLoop:
	33	V	gBB1 = PHI<true, gBB9>
	2	D	i_2 = PHI<0, i_44>;
	3	V	c_3 = i_2 < RowsNumb;
	4	C	if(c_3) {
BB2	34	V	gBB2 = gBB1 && c_3;
	5	V	local_5 = 0;
	6	M	rl_6 = gBB2 ? RowLength[i_2] : 0;
	7	V	c_7 = rl_6 != 0 {
	8	C	if(c_7) {
BB3	35	V	gBB3 = gBB2 && c_7;
	9	V	j_9 = 0;
BB4	10	S	InnerLoop:
	36	V	gBB4 = PHI<gBB2 && c_7, gBB5>
	11	V	j_11 = PHI<j_9,j_45>
	12	V	local_12 = PHI<local_5, local_17>
	13	V	c_13 = j_11 < rl_6;
	14	C	if(c_13) {
BB5	37	V	gBB5 = gBB4 && c_13;
	15	M	t_15 = gBB5 ? Matrix[i_2][j_11] : 0;
	16	V	local_16 = local_12 + t_15;
	17	V	j_17 = j_11 + 1;
	45	V	j_45 = gBB5 ? j_17 : j_11;
	18	S	goto InnerLoop;
			}
BB12	38	V	gBB12 = gBB2 && c_7;
	46	V	local_46 = gBB12 ? local_12 : local_5;
			}
BB11	30	V	local_30 = PHI<local_5, local_46>;
	39	V	gBB11 = gBB1 && c_3;
	31	V	c_31 = ! c_7;
	32	C	if(c_31)
			{
BB6	40	V	gBB6 = gBB11 && c_31;
	19	M	t_19 = gBB6 ? old_row[i_2] : 0;
	20	V	c_20 = t_19 > 0;
	21	C	if(c_20) {
BB7	41	V	gBB7 = gBB6 && c_20;
	22	V	t_22 = gBB7 ? old_sum[i_2] : 0;
	23	V	local_23 = t_22;
	47	V	local_47 = gBB7 ? local_23 : local_5;
			}
BB8	24	V	local_24 = PHI<local_5, local_23>;
	42	V	gBB8 = gBB11 && c_31;
	48	V	local_48 = gBB8 ? local_24, local_47;
			}
BB9	25	V	local_25 = PHI<local_30, local_48>;
	43	V	gBB9 = gBB2 && c_3;
	26	M	t_26 = local/rl_6;
	27	M	if(gBB9) avg[i_2] = t_26;
	28	D	i_27 = i_2 + 1;
	44	V	i_44 = gBB9 ? i_27 : i_2;
	29	S	goto OuterLoop;
			}
BB10			/* After the loop */

Figure 9: Classification of instructions of example of Figure 8. V are the *Vector instructions*, M are the *MultiScalar instructions*, D are the *DoAll loop instructions*, C are the *Control instructions*, S are the *Scalar instructions*.

solution would be too large. Moreover some operations cannot be implemented in this way (e.g., non aligned memory accesses).

On the contrary, the introduction of the second class of instructions provides more flexibility to the High Level Synthesis design flow because allows to perform outer loop vectorization of loops containing instructions which cannot be vectorized. Moreover the choice between ①, ② and ③ allows to explore different possible trade-offs between area and performance in the produced solutions.

Note that classifying an instruction as *Vector* or *MultiScalar* determines only if an instruction will be synthesized as ④ or not. Since the choice between ①, ② and ③ does not concern vector functional units, this can be demanded to the rest of the High Level Synthesis design flow. The proposed methodology classifies as *MultiScalar* all the instructions which cannot be implemented by vector functional units (e.g., predicated load and stores) and all the instructions that require more than one clock cycle to be executed.

The classification of the instruction of example of Figure 1 is shown in Figure 9. In particular, all the predicated instructions have been classified as *MultiScalar instructions* since vector functional units which implement these types of operations (predicated load and predicated store) are not available. Instruction 27 which is a division has been classified as *Multiscalar instruction* since it takes more than one clock cycle to be executed.

4.8. Instructions Transformation

In this step the actual vectorization of the *DoAll loop* is performed. Different types of transformations are applied to the different classes of instructions identified in the previous phase. Before applying the instruction transformations, all the scalar SSAs defined inside the *DoAll loop*, included all the induction variables, have to be transformed in vector variables. Aggregate variables (i.e., structures and arrays) whose scope is in the *DoAll loop* have to be vectorized as well by transforming them in vector of the original type. Finally, all the remaining variables have not to be modified. In the following the vectorized variables will be identified by overlining them.

In the presented example, all the variables but `RowsNumb`, `RowLength`, `Matrix`, `old_row`, `old_sum`, and `avg` have to be vectorized.

How to transform the *DoAll instructions* depends on the characteristics of the parallel loop. For the sake of brevity, it will be presented only how to transform *DoAll instructions* in case of `for` loops, but the proposed methodology can be applied even with different patterns (e.g., `while` loops). Note that the presented transformations are applied also to uncountable loops, i.e., when the number of iterations is not known at design time. The transformations to be applied are the following:

- the primary induction variable is initialized with the values that it takes during first the P iterations of the loop by modifying the values in the `PHI` in the header of the *DoAll* loop; in the presented example it is initialized to `{0,1}`.

- increment instruction is transformed in a vector instruction; the added constant is the increment of the sequential loop multiplied by P; in the presented example `i_27 = i_2 + 1` is transformed in `i_27 = i_2 + {2,2}` since $P = 2$.

If secondary induction variables are present, further changes can be necessary.

Control instructions have to be transformed to support the execution of conditioned basic block when at least one of the element of the vector execution requires it. The conditional constructs have to be transformed in such way that their condition is true if at least one of the element of the vector version of the original condition is true. For example `if(c_31)` has to be modified in such way that the new condition is true if at least one of the element of the vector version of $\overline{\text{c\_31}}$. This is obtained by creating a new condition $\overline{\text{c\_31}}$`[0] || `$\overline{\text{c\_31}}$`[1]`.

Each *Vector instruction* is transformed in a single vector instruction which directly writes a whole vector variable. Finally, each *MultiScalar instruction* is transformed in P scalar instructions, each of which writes a different element of a vector variable or performs a scalar store.

Figure 10 shows the final intermediate representation after that vectorization has been applied.

4.9. Synthesis

After that the previous steps of the proposed methodology flow have been applied, state-of-the-art High Level Synthesis flows can be applied. Since the transformed intermediate representation contains vector instructions, the design flows have to support synthesis of vector functional units. If the intermediate representation contains a predicated instruction not supported by the High Level Synthesis flow, this can still be synthesized by reintroducing an ad-hoc basic block containing it. For example given an intermediate representation containing some conditional stores:

```
/* Code before predicated stores */
predicate[0] ? array[0] = value[0] : (void) 0;
predicate[1] ? array[1] = value[1] : (void) 0;
/* Code after predicated stores
```

if the High Level Synthesis does not support the synthesis of such type of instructions, the intermediate representation can be modified by inserting two new basic blocks each one containing one of the two stores:

```
/* Code before predicated stores */
if(predicate[0])
{
   /* New basic block */
   array[0] = value[0];
}
if(predicate[1])
{
```

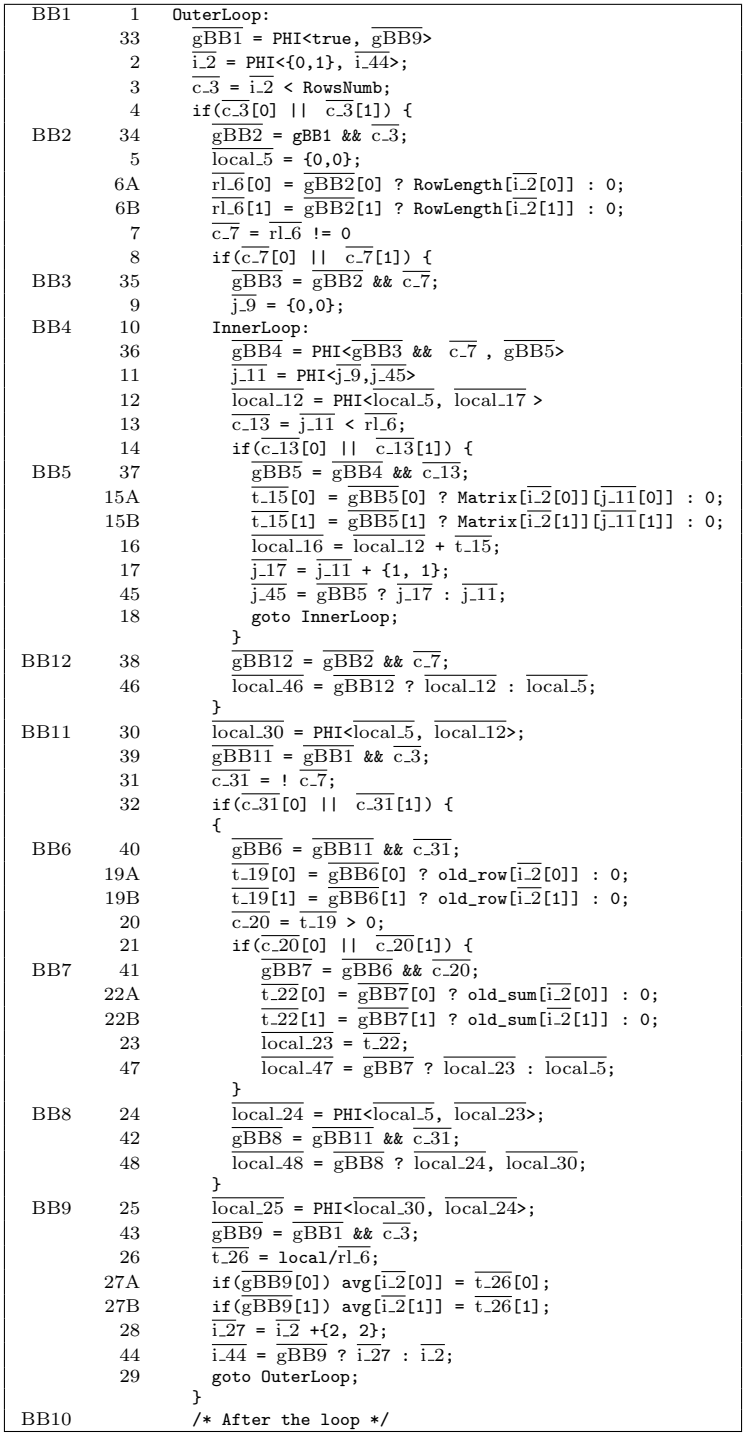


Figure 10: Intermediate representation of example of Figure 1 after application of vectorization.

```
   /* New basic block */
   array[1] = value[1];
}
/* Code after predicated stores */
```

The same type of transformation can be applied to whatever type of predicated instructions.

The methodology assumes also that vector variables are synthesized as registers: if vector variables were mapped on BRAM, the methodology is still applicable, but the memory accesses overhead would completely nullify the benefits of the vectorization. It is worth noting that the rest of the High Level Synthesis flow can still modify the structure of the Control Flow Graph and the contents of the different basic blocks. After that the proposed methodology has been applied, indeed is not required anymore to maintain the Control Flow Graph structured, so Control Flow Graph optimizations and instructions speculation can be applied.

5. Experimental Results

The proposed methodology has been implemented in Bambu [28], a modular framework for High Level Synthesis developed at Politecnico di Milano. Since the identification of the DoAll loops is out of the scope of this paper, this type of analysis has not been implemented: benchmarks have to be annotated by hand with a `#pragma omp simd` [22] to be vectorized. The degree of parallelism of each loop can be specified by the designer by means of the `safelen` clause associated with each `#pragma omp simd`.

The proposed methodology has been verified on a set of parallel benchmarks distributed with Legup [8]. In OpenMP benchmarks each `#pragma omp for` has been replaced with `#pragma omp simd`, while pthread benchmarks have to be re-factorized to replace pthread parallelism with `#pragma omp simd`. Different degrees of parallelism have been considered: 1 (absence of parallelism), 2, 3, 4 and 8. The degree of parallelism of 3 in particular has been chosen to show the application of the proposed methodology when the degree of parallelism is not a multiple of the number of iterations of the *DoAll loop*. For each degree and for each benchmark a different hardware accelerator is produced by *Bambu*. The tool has been configured with default options: the level of optimization is O2, input and output data are stored on dual port block RAM memories and the target frequency is 100MHz. Two target platforms have been considered: the Xilinx Virtex 7 xc7vx690t and the Altera Stratix-V 5SGXEA7N2F45C1.

The solutions produced by High Level Synthesis have been finally synthesized with Xilinx Vivado [20] and Altera Quartus II [29]. The synthesis results obtained after place and route on different benchmarks with different degrees of parallelism are presented in Table 2 and Table 3. The hardware accelerators for *Mandelbrot* with parallel degree of 8 have not been generated since the number of iterations of the *DoAll loop* contained in it is 4. The synthesis of *Blackschoels* on the Stratix V with P = 3,4,8 was not been possible because

Benchmark	P	Area(Ratio)		Cycles(Speedup)	FMax (Ratio)	Product Ratio
		LUT FF Pairs	DSPs			
Add	1	133 (1)	0	20018 (1)	429.37 (1)	1
	2	118 (0.88)	0	10010 (2.00)	364.56 (0.85)	0.51
	3	352 (2.64)	0	7511 (2.66)	294.55 (0.69)	1.45
	4	118 (0.88)	0	5008 (4.02)	421.94 (0.98)	0.21
	8	127 (0.95)	0	2507 (8.02)	430.66 (1.00)	0.12
Blackscholes	1	10755 (1)	71 (1)	798351 (1)	102.18 (1)	1
	2	17022 (1.58)	154 (2.16)	431548 (1.84)	107.33 (1.05)	0.81
	3	23993 (2.23)	232 (3.26)	530518 (1.50)	102.22 (1.00)	1.47
	4	29129 (2.70)	343 (4.83)	237724 (3.35)	100.98 (0.99)	0.81
	8	52619 (4.89)	846(11.92)	261382 (3.05)	99.12 (0.97)	1.66
Boxfilter	1	3809 (1)	0	266947 (1)	106.41 (1)	1
	2	7221 (1.90)	0	139979 (1.91)	114.32 (1.07)	0.92
	3	10449 (2.74)	0	131974 (2.02)	114.32 (1.07)	1.25
	4	13967 (3.67)	0	87989 (3.03)	107.77 (1.01)	1.20
	8	28649 (7.52)	0	61994 (4.30)	106.35 (1.00)	1.73
Dotproduct	1	839 (1)	3(1)	12027 (1)	115.83 (1)	1
	2	1067 (1.27)	6 (2)	9019 (1.33)	110.29 (0.95)	1.00
	3	1411 (1.68)	9 (3)	9020 (1.33)	106.11 (0.92)	1.37
	4	1346 (1.60)	12(4)	7514 (1.60)	107.33 (0.93)	1.06
	8	1817 (2.17)	24 (8)	6774 (1.77)	107.43 (0.93)	1.31
Hash	1	994 (1)	0	192027 (1)	144.72(1)	1
	2	1715 (1.73)	0	96019 (2.00)	135.32 (0.94)	0.92
	3	2230 (2.24)	0	99020 (1.93)	133.72 (0.92)	1.26
	4	2274 (2.29)	0	66017 (2.91)	127.99 (0.88)	0.88
	8	3539 (3.56)	0	57024 (3.37)	121.07 (0.87)	1.22
Histogram	1	2585 (1)	0	158752 (1)	140.08 (1)	1
	2	3729 (1.44)	0	88029 (1.80)	120.61 (0.86)	0.92
	3	5148 (1.99)	0	80935 (1.96)	118.62 (0.84)	1.20
	4	6004 (2.32)	0	54102 (2.93)	117.44 (0.84)	0.94
	8	11156 (4.31)	0	36140 (4.39)	108.50 (0.77)	1.29
Mandelbrot	1	1053 (1)	12(1)	852239 (1)	114.32 (1)	1
	2	1761 (1.62)	24(1)	426123 (2.00)	112.57 (0.98)	0.82
	3	2669 (2.53)	36(1)	434317 (1.96)	110.40 (0.97)	1.30
	4	3033 (2.88)	48(1)	217162 (3.92)	106.59 (0.93)	0.77
	8	-	-	-	-	-

Table 2: Experimental Results of applying the proposed methodology targeting Xilinx Virtex 7 xc7vx690t.

the device does not contain enough DSPs to implement the vectorized version of the benchmarks. The area results refer only to the synthesized accelerators since the produced parallel hardware architectures, differently from the ones presented in [8], do not require any external processor nor external controller to be integrated in a system. Memory utilization has not been reported since it is independent from the degree of parallelism.

The results obtained on the different platforms are similar, so that the proposed methodology can actually be considered as applicable to different families of FPGAs. Moreover the results show how it is effectively able to save resources with respect to the complete duplication of loop implementation: the area of the produced solutions indeed growths less that the parallel degree. In a single case (with $P = 3$ and Stratix-V as target), the growth in terms of resources of the parallelized accelerator is more than linear. On the opposite side, the maximum area saving has been obtained for *Add* on the Xilinx board where the number of used LUT FF pairs is almost the same for most of the parallel degrees. When the parallel degree grows, the number of LUTs does not grow

Benchmark	P	Area(Ratio) ALMs	Area(Ratio) DSPs	Cycles(Speedup)	FMax	Product Ratio
Add	1	59 (1)	0	20018 (1)	594.88 (1)	1
	2	58 (0.98)	0	10010 (2.00)	578.03 (0.97)	0.51
	3	123 (2.08)	0	7511 (2.66)	465.98 (0.78)	1.00
	4	75 (1.27)	0	5008 (4.00)	619.20 (1.04)	0.31
	8	76 (1.28)	0	2507 (7.98)	599.16 (1.01)	0.16
BlackSchoels	1	6301 (1)	78 (1)	1460847 (1)	93.60 (1)	1
	2	9729 (1.54)	173 (2.20)	780908 (1.87)	108.94 (1.16)	0.71
	3	Not Available				
	4					
	8					
Boxfilter	1	2186 (1)	0	322947 (1)	127.93 (1)	1
	2	4102 (1.88)	0	167979 (1.92)	117.36 (0.92)	1.06
	3	6752 (3.08)	0	152974 (2.11)	119.66 (0.94)	1.55
	4	8554 (3.91)	0	101989 (3.16)	113.22 (0.89)	1.39
	8	16492 (7.54)	0	68994 (4.68)	113.55 (0.89)	1.81
Dotproduct	1	375 (1)	2 (1)	12027(1)	127.06 (1)	1
	2	473 (1.26)	4 (2.00)	9019 (1.33)	131.56 (1.04)	0.91
	3	640 (1.71)	6 (3.00)	9020 (1.33)	128.83 (1.01)	1.27
	4	589 (1.57)	8 (4.00)	7517 (1.60)	137.01 (1.08)	0.90
	8	854 (2.27)	16 (8.00)	6774 (1.78)	124.61 (0.98)	1.30
Hash	1	880 (1)	0	192027 (1)	135.70 (1)	1
	2	1405 (1.60)	0	96019 (2.00)	132.36 (0.98)	0.81
	3	1678 (1.91)	0	99020 (1.93)	137.70 (1.01)	0.98
	4	1637 (1.86)	0	66017 (2.91)	137.38 (1.01)	0.63
	8	2137 (2.43)	0	57024 (3.37)	124.64 (0.91)	0.79
Histogram	1	1343 (1)	2 (1)	158752 (1)	140.92(1)	1
	2	2094 (1.56)	4 (2.00)	88029 (1.80)	129.13 (0.92)	0.94
	3	2893 (2.15)	6 (3.00)	80935 (1.96)	131.13 (0.93)	1.17
	4	3439 (2.56)	8 (4.00)	54102 (2.93)	134.10 (0.95)	0.92
	8	7189 (5.35)	16 (8.00)	36140 (4.39)	114.31 (0.81)	1.50
Mandelbrot	1	389 (1)	6 (1)	852239 (1)	140.34 (1)	1
	2	549 (1.41)	12 (2.00)	426123 (2.00)	136.35 (0.97)	0.72
	3	873 (2.24)	18 (3.00)	434317 (1.96)	136.57 (0.97)	1.17
	4	884 (2.27)	24 (4.00)	217162 (3.92)	134.41 (0.96)	0.60
	8	-	-	-	-	-

Table 3: Experimental Results of applying the proposed methodology targeting Altera Stratix-V 5SGXEA7N2F45C1.

because their are better exploited (i.e., more of their inputs are used): in particular the number of 6 inputs LUTs is increased while the number of 2 inputs LUTs is decreased. The resource saving however is not effective on the usage of DSPs: their number growths linearly in *Dotproduct* and *Mandelbrot* benchmark while in case of *Blackscholes* benchmark with target the Virtex 7 device, their number grows even more than linearly since *Bambu* has more difficulties in sharing DSPs among operations in produced accelerators.

Differently from [6], the speed-up obtained on *Boxfilter* and on *Histogram* is not more than linear (i.e., larger than parallel degree) since there is not the gain due to the if-conversions which are not required by this methodology. On the contrary, for most of the benchmarks the real speed-up grows less than parallel degree. There are two main causes of this reduction: the considered memory architecture and the control divergence. The first cause consists of adopting a memory architecture which has only two ports which limits the exploitation of parallelism since limits to two the number of simultaneous memory accesses. Memory partitioning, like the approach considered in [9], can mitigate in a sen-

sible way this issue but requires an ad-hoc memory architecture. The effect of the limit on the number of ports is more sensible when comparing the results of $P = 2$ and $P = 3$. In this case, the benefit of the increment of the parallelization can be nullified by the requirement of serializing the third load or store when memory operations cannot be executed in a vectorized way. Moreover, these added load and store operations introduce a penalty in terms of execution latency even if their predicate is false (i.e., they have not to be really executed).

For most of the benchmarks there is not any significant difference in terms of frequency when vectorization is applied, even by considering $P = 8$. For some combinations of benchmark, target and P the frequency of the vectorized accelerator is even better than the frequency of the scalar version (e.g., *Boxfilter* on Virtex 7 with $P = 2$ and $P = 3$). In a single case (*Add* benchmark implemented on Xilinx board with $P = 3$) the vectorization reduces the frequency by 31%, but the obtained frequency is still much larger than target, so it can be expected that the synthesis tool has not fully optimized the designed accelerator. It is worth noting that the generated accelerators meet almost always the target frequency (100MHz). The timing constraint is violated by a single accelerator (*Blackscholes* on Virtex 7 with $P = 8$), but the obtained frequency (99.12MHz) is very close to the target.

There is a gain in terms of area-delay product for most of the benchmarks with $P = 2$ and $P = 4$ up to 40% (*Mandelbrot* with $P = 4$ on the Stratix V) since the performances grow faster than resource utilization because of sharing and logic synthesis optimizations applied to the generated accelerators. On the contrary, because of the performances limitations due to memory accesses, the solutions with $P = 3$ presents worse results. The accelerators characterized by $P = 8$ are penalized in terms of area-delay product by the limited speed-up due to the effects of control divergence, which are more evident on them than on the other accelerators. The area-delay gain on the *Add* benchmark is very large because of its characteristics: the area of vectorized versions does not grow (or grows in a very limited way) because the logic synthesis optimizations allow to use the same number of LUTs exploiting more their inputs. On the contrary, the speed-up grows almost in a linear way, so the combination of these two effects is a linear decrement of the area-delay product.

Finally, it has to be highlighted that direct comparison of the results of the proposed methodology and the results presented in [8] is not possible, not only for the different analyzed benchmarks but also for the different types of experimental setup (tool and devices) and types of built architectures. Differently from [8] indeed, the parallel accelerators built with the proposed methodology do not require to be coupled with a controller processor.

6. Conclusions

In this paper a methodology for the synthesis of parallel accelerators based on vectorization has been presented. This methodology is able to synthesize by means of outer loop vectorization also irregular loops: nested loops, conditional constructs and operations which cannot be vectorized are supported. Since it

transforms high level specifications, it can be easily integrated in existing design flows if they support synthesis of vector functional units. Experimental results show the effectiveness of the proposed methodology: the parallel produced solutions present a significant speed-up with a limited resource usage growth with respect to non vectorized solutions.

References

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, Z. Zhang, High-level synthesis for fpgas: From prototyping to deployment, IEEE TCAD 30 (4) (2011) 473–491.

[2] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, J. Cong, Improving high level synthesis optimization opportunity through polyhedral transformations, FPGA '13, ACM, New York, NY, USA, 2013, pp. 9–18.

[3] D. Nuzman, A. Zaks, Outer-loop vectorization: Revisited for short simd architectures, PACT '08, ACM, New York, NY, USA, 2008, pp. 2–11. doi:10.1145/1454115.1454119.

[4] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, Nvidia tesla: A unified graphics and computing architecture, Micro, IEEE 28 (2) (2008) 39–55. doi:10.1109/MM.2008.31.

[5] R. Karrenberg, S. Hack, Whole-function vectorization, in: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 141–150.

[6] M. Lattuada, F. Ferrandi, Exploiting outer loops vectorization in high level synthesis, in: Architecture of Computing Systems - ARCS 2015 - 28th International Conference, Porto, Portugal, March 24-27, 2015, Proceedings, 2015, pp. 31–42. doi:10.1007/978-3-319-16086-3_3.

[7] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, W.-M. W. Hwu, Efficient compilation of cuda kernels for high-performance computing on fpgas, ACM TECS 13 (2) (2013) 25:1–25:26.

[8] J. Choi, S. Brown, J. Anderson, From software threads to parallel hardware in high-level synthesis for fpgas, FPT '13, 2013, pp. 270–277.

[9] V. G. Castellana, M. Minutoli, A. Morari, A. Tumeo, M. Lattuada, F. Ferrandi, High level synthesis of RDF queries for graph analytics, in: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2-6, 2015, 2015, pp. 323–330. doi:10.1109/ICCAD.2015.7372587.

[10] M. Tan, G. Liu, R. Zhao, S. Dai, Z. Zhang, Elasticflow: A complexity-effective approach for pipelining irregular loop nests, in: Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on, 2015, pp. 78–85. doi:10.1109/ICCAD.2015.7372553.

[11] A. Cilardo, L. Gallo, N. Mazzocca, Design space exploration for high-level synthesis of multi-threaded applications, Journal of Systems Architecture 59 (10, Part D) (2013) 1171 – 1183.

[12] J. Anantpur, R. Govindarajan, Taming control divergence in gpus through control flow linearization, in: Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, 2014, pp. 133–153. doi:10.1007/978-3-642-54807-9_8.

[13] V. Raghunathan, A. Raghunathan, M. Srivastava, M. Ercegovac, High-level synthesis with simd units, ASP-DAC '02, 2002, pp. 407–413.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems 13 (4) (1991) 451–490.

[15] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: principles, techniques, and tools, Addison-Wesley Longman Publishing Co., Inc., 1986.

[16] C. Böhm, G. Jacopini, Flow diagrams, turing machines and languages with only two formation rules, Commun. ACM 9 (5) (1966) 366–371. doi:10.1145/355592.365646.

[17] V. C. Sreedhar, G. R. Gao, Y. Lee, Identifying loops using DJ graphs, ACM Transactions on Programming Languages and Systems 18 (6) (1996) 649–658.

[18] R. Cytron, J. Ferrante, V. Sarkar, Compact representations for control dependence, in: PLDI, 1990, pp. 337–351.

[19] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, K. Asanović, Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures, in: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47, IEEE Computer Society, Washington, DC, USA, 2014, pp. 101–113. doi:10.1109/MICRO.2014.48.

[20] Xilinx, Vivado Design Suite, http://www.xilinx.com (2013).

[21] P. Feautrier, Automatic parallelization in the polytope model, in: Laboratoire PRiSM, Universit des Versailles St-Quentin en Yvelines, 45, avenue des tats-Unis, F-78035 Versailles Cedex, Springer-Verlag, 1996, pp. 79–103.

[22] OpenMP, Application Program Interface, version 4.0 (July 2013).

[23] J. R. Allen, K. Kennedy, C. Porterfield, J. Warren, Conversion of control dependence to data dependence, in: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '83, ACM, New York, NY, USA, 1983, pp. 177–189. doi:10.1145/567067.567085.

[24] B. Boissinot, F. Brandner, A. Darte, B. D. de Dinechin, F. Rastello, A Non-iterative Data-Flow Algorithm for Computing Liveness Sets in Strict SSA Programs, in: Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings, 2011, pp. 137–154. doi:10.1007/978-3-642-25318-8_13.

[25] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, T. Czajkowski, Impact of fpga architecture on resource sharing in high-level synthesis, FPGA '12, ACM, New York, NY, USA, 2012, pp. 111–114.

[26] J. Cong, W. Jiang, Pattern-based behavior synthesis for fpga resource reduction, FPGA '08, ACM, New York, NY, USA, 2008, pp. 107–116.

[27] S. Kurra, N. K. Singh, P. R. Panda, The impact of loop unrolling on controller delay in high level synthesis, DATE '07, 2007, pp. 391–396.

[28] C. Pilato, F. Ferrandi, Bambu: A modular framework for the high level synthesis of memory-intensive applications, FPL '13, 2013, pp. 1–4.

[29] Altera, Quartus II, http://www.altera.com (2013).