Exploiting Outer Loops Vectorization in High Level Synthesis

Marco Lattuada and Fabrizio Ferrandi

Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioingegneria
marco.lattuada@polimi.it fabrizio.ferrandi@polimi.it

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 complex DoAll loops based on outer vectorization is proposed. Vectorization is not limited to the innermost loops: complex constructs such as nested loops, conditional constructs and function calls are supported. Experimental results on parallel benchmarks show up to 7.35x speed-up and up to 40% reduction of area-delay product.

1 Introduction

Heterogeneous multiprocessor systems are becoming very common in a large group of embedded system application fields because of their computational power and their power efficiency. This type of architecture requires that the different tasks in which an application is decomposed are assigned to the most suitable processing element. The parts of the application which are characterized by high degree of parallelism are good candidates to be mapped on programmable hardware devices 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 overcome or at least to mitigate this issue, High Level Synthesis [4] 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 hardware platforms. An important class of loops which are good candidates to be parallelized are DoAll loops [19]. 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 [13] (i.e., introduction of functional units processing vectors of data) to mitigate these problems. 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 complex loops (i.e., loops that contain nested loops, conditional constructs and function calls).
- It allows to selectively combine vectorization with local pipelined computation potentially exploiting benefits of both the approaches.

The rest of the paper is organized as follows. Section 2 presents related work while Section 3 presents a motivational example. 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 [19]: 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. [15] 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. [2] 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 does not have any applicability limitation, but 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. A similar approach (i.e., the automatic synthesis of OpenMP annotated applications) was proposed in [3] 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.

Parallelization of complex DoAll loops (i.e., outer loops) by means of vectorization was proposed for SIMD processors [13]: 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. A similar approach is proposed in this paper, but it is adopted during the synthesis of hardware accelerators. Finally, the effects of using vector functional units in High Level Synthesis have already been evaluated in [17]. 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.

3 Motivational Example

In this section a small motivational example is presented showing the potential advantages of the outer loop vectorization with respect to other loop optimization techniques when applied in High Level Synthesis. The example, presented in the left part of Fig. 1, consists of a brief fragment of code containing two nested loops. The outer loop is characterized by a fixed number of iterations (16) while the number (`k`) of iterations of inner loop cannot be computed at compile time. Moreover, the iterations of the outer loop can be parallelized while the iterations of the inner loop have to be executed in sequence. To allow the application of the most common types of loop parallelization techniques, the source code in the left part of Fig. 1 has been transformed by means of if-conversion removing

	Initial Code	Pre-Processed Code	Transformed Code
A	for(i=0; i<16; i++) {	for(i=0; i<16; i++) {	for($\overline{\texttt{i}}$={0,1}; i[0]<16; $\overline{\texttt{i}}$ = $\overline{\texttt{i}}$+{2,2}) {
B	sum = 0;	sum = 0;	$\overline{\texttt{sum}}$ = {0,0};
C	for(j=0; j<k; j++) {	for(j=0; j<k; j++) {	for(j=0; j<k; j++) {
D	if(sum < 10) {	c = sum < 10;	$\overline{\texttt{c}}$ = $\overline{\texttt{sum}}$ < 10

E		temp = in[i][j];	temp[0] = in[i[0]][j];
			temp[1] = in[i[1]][j];

F	sum = sum + in[i][j];	sumS = sum + temp;	$\overline{\texttt{sumS}}$ = $\overline{\texttt{sum}}$ + $\overline{\texttt{temp}}$;

G	}
	}
		sum = c ? sumS : sum;
		}
			sum[0] = c[0] ? sumS[0] : sum[0];
			sum[1] = c[1] ? sumS[1] : sum[1];
			}

H	res[i] = sum/k;
	}
		res[i] = sum/k;
		}
			res[i[0]] = sum[0]/k;
			res[i[1]] = sum[1]/k;
			}

Fig. 1. Example of application of the proposed methodology.

1	A(0,-)
	A(1,-)
		$\overline{\texttt{i}}$={0,1};

2	B(0,-)
	B(1,-)
	C(0,0)
	C(1,0)
		$\overline{\texttt{sum}}$={0,0};
		j=0;

3	D(0,0)
	D(1,0)
	E(0,0)
	E(1,0)
		$\overline{\texttt{c}}$=$\overline{\texttt{sum}}$<10;
		temp[0]=in[i[0]][j];
		temp[1]=in[i[1]][j];

4	F(0,0)
	F(1,0)
	C(0,1)
	C(1,1)
		$\overline{\texttt{sumS}}$=$\overline{\texttt{sum}}$+$\overline{\texttt{temp}}$;
		j++;

5	G(0,0)
	G(1,0)
		sum[0] = c[0] ? sumS[0] : sum[0];
		sum[1] = c[1] ? sumS[1] : sum[1];

6	D(0,1)
	D(1,1)
	E(0,1)
	E(1,1)
		$\overline{\texttt{c}}$=$\overline{\texttt{sum}}$<10;
		temp[0]=in[i[0]][j];
		temp[1]=in[i[1]][j];

7	F(0,1)
	F(1,1)
	C(0,2)
	C(1,2)
		$\overline{\texttt{sumS}}$=$\overline{\texttt{sum}}$+$\overline{\texttt{temp}}$;
		j++;

5	G(0,1)
	G(1,1)
		sum[0] = c[0] ? sumS[0] : sum[0];
		sum[1] = c[1] ? sumS[1] : sum[1];

9	H(0,-)
	H(1,-)
		res[i[0]]=sum[0]/k;
		res[i[1]]=sum[1]/k;

Fig. 2. Execution trace of the first two iterations of outer loop when outer loop vectorization is applied.

the conditional construct instruction D. Moreover, the complex instruction F has been decomposed into two simpler instructions (the reading from the matrix and the sum). The result of these transformations is shown in the central part of Fig. 1 where instructions E and G have been added. Nevertheless, the presence of a nested and non parallelizable loop with an unknown number of iterations prevents the application of some loop optimization techniques, but not of the proposed.

The only other loop parallelization technique which can be applied to the loops of Fig. 1 without any further change is the *Unrolling of inner loop* [6]. However, the instructions belonging to consecutive iterations of the loop cannot be executed in parallel because of data dependence between G and D, limiting the benefit of adopting this optimization. The *Unrolling of outer loop* [6], the *Pipelining of inner loop* [6], the *Pipelining of outer loop* [11] and the *Vectorization of inner loop* [12] cannot be applied to the example because of the variable number of iterations of inner loop and because of inter-iterations data dependence.

On the contrary, outer loop vectorization can be applied to the considered example: the right part of Fig. 1 shows the results of the optimization, while Fig. 2 reports which are the operations executed by an accelerator synthesized with the proposed methodology in each control step during first iteration of outer loop. For the sake of brevity and simplicity, it is assumed that the execution time of each synthesized instruction is one clock cycle, that chaining is not exploited and that `k=2`. A pair of indices has been associated to each instruction: the first index is the relative iteration number of the outer loop to which the instruction belongs while the second index is the relative iteration number of the inner loop. The effects of the outer vectorization are that the first iteration of nested loop executed during first iteration of DoAll loop is executed in parallel with first iteration of nested loop executed during second iteration of DoAll loop and so on. The details about the proposed solution and about how this can been obtained will be presented in the following section.

4 Proposed Methodology Flow

The proposed methodology is integrated in a High Level Synthesis flow and aims at synthesizing a parallel hardware accelerator by means of outer loop vectorization. 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. 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 significant part of the proposed methodology flow consists of the transformations applied to the loop to be synthesized. These transformations can be applied with similar results to the source code or to the high level intermediate representations adopted in the first phases of a High Level Synthesis design flow. The direct manipulation of high level representations allows to easily integrate the proposed methodology in existing High Level Synthesis flows, provided that they support vector functional units. The methodology assumes 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.

A loop can be synthesized with the proposed methodology if:

1. it is a DoAll loop, i.e., all iterations can be executed in parallel;
2. the number n of its iterations is multiple of the degree of parallelism P: $n \% P = 0$;
3. nested loops are not controlled by conditional constructs (i.e., nested loops are not contained in a *then* or in a *else* block; polyhedral transformations can help to remove violations to this constraint;
4. the number of iterations of nested loops does not depend on a value computed in the outer loop.

Note that the loop to be parallelized and the nested loops can contain conditional constructs which will be removed by if-conversion. Moreover, it is not required that nested loops are DoAll loops nor countable loops, but only that their iterations number does not depend on a value computed in the outer loop since they will be not internally parallelized nor unrolled. Indeed, the vectorization of the outer loop implicitly creates multiple copies of the inner loops. Each copy will be executed sequentially, but the different copies will be executed in parallel and in a completely synchronized way. The synchronization is implicit and it is guaranteed by the fourth precondition. Second constraint can potentially be removed by adding the possibility to execute in an ad-hoc way the last $n\%P$ iterations of the loop. In a similar way, also the constraint on the number of iterations of nested loops can be removed.

The proposed methodology flow is composed of several steps:

1. *Loops Analysis*: the specification is analyzed to identify DoAll loops.
2. *PreProcessing transformation*: conditional constructs in the loops are removed by transforming instructions controlled by them in speculated or predicated instructions; complex instructions are decomposed in simpler instructions.
3. *Instructions classification*: each instruction of the loops is analyzed to identify if it controls the execution of a loop and, if not, if it has to be transformed in a vector instruction or in a set of scalar instructions.
4. *Instructions transformation*: instructions which control 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.
5. *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 Fig. 1 will be shown.

Loops Analysis. The source code or its high level 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 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` [14].

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

Preprocessing transformation. In this step the original specification is modified to remove complex instructions and conditional constructs (i.e., `if`). First objective is achieved by replacing complex operations (i.e., operations which require more than one functional unit to be synthesized) with simpler operations. Second objective is obtained by applying if-conversion by means of speculation [7] and predication [10]. The recursive application of these transformations to the body of the DoAll loop and to its nested loops removes all the conditional constructs allowing to apply the following steps of the methodology. Note that, since this transformation is required to apply the loop vectorization, it always has to be performed, even when it is not profitable because of possible mispeculations.

The result of applying these transformations to the example is shown in the centre part of Fig. 1. Instruction F has been decomposed into two operations E and F (a read from a matrix and a sum), then instruction D has been transformed in a boolean assignment, while instruction E and instruction F have been speculated.

Instructions classification. During this phase of the methodology each instruction which is part of the analyzed loop or of a nested loop is classified into four different classes:

- *Vector instructions*: they will be transformed into vector instructions; in the presented example they are B, D, and F.
- *MultiScalar instructions*: they will be transformed into P scalar instructions; in the presented example they are E, G and H.
- *DoAll loop instructions*: they are the instructions controlling the execution of the DoAll loop; in the presented example A is the only one;
- *Nested loop instructions*: they are the instructions controlling the execution of nested loops; in the presented example C is the only one.

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 [8] [5] and smaller controller complexity [9].

If the second class of instructions was not introduced, all operations would be synthesized as ④, producing the best solution in terms of performance, but the increment of area with respect to the non parallelized 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., non-contiguous memory accesses) and all the instructions that require more than one clock cycle to be executed.

In the analyzed example, instructions E and G have been classified as *MultiScalar instructions* since vector functional units which implement these types of operations (non-contiguous load and conditional assignment) are not available. Finally, since the division requires more than one cycle, instruction H has been classified as *MultiScalar instruction.* In this way the 2 divisions can be synthesized as ① (1 divisor), ② (1 divisor which executes the two divisions in pipeline) or ③ (2 different divisors) according to the choices taken by the rest of the High Level Synthesis design flow.

Instructions Transformation. Different types of transformations are applied in this step. For the sake of brevity, it will be presented only how to transform simple `for` loops, but the proposed methodology can be applied even with different patterns (e.g., `while` loops). All the scalar variables defined inside the DoAll loop (with the exception of the induction variables of the nested loops) are transformed in vector variables, while the variables defined outside the DoAll loop are not modified. In the considered example `i`, `res`, `sum`, `sumS`, and `temp` are vectorized while `j`, and `k` are not.

DoAll loop instructions are transformed to support simultaneous execution of multiple iterations of the parallelized loop. The transformations to be applied are the following:

- primary induction variable is initialized with the values it would assume during first P iterations of the 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++` is transformed in `i = i + {2,2}` since $P = 2$.
- guard instruction is not transformed in a vector instruction; changes to operands can be necessary to extract the scalar variables from the vector variables and to fix the loop termination; in the presented example `i<16` is transformed in `i[0]<16`.

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

Nested loop instructions have to be transformed to support simultaneous execution of implicit multiple copies of the nested loops. The transformations to be applied in case of `for` instructions are limited to their operands. In the presented example, operands of `for(j=0; j<k; j++)` have not to be changed since `j` is defined in this instruction and `k` is not defined inside the DoAll loop.

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. The input variables of each instruction are opportunely modified to correctly manage scalar/vector data. The instructions to extract scalar values from vector variables (e.g., `var_0 = var[0]`) and to compose vector variables starting from scalar variables (e.g., `var = {var_0, var_1}`) may need to be added.

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.

5 Experimental Results

To evaluate the proposed methodology, this has been implemented in Bambu [16], 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` [14] 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 [2]. In OpenMP benchmarks each `#pragma omp for` has been replaced with `#pragma omp simd`, while pthread benchmarks have to be refactorized to replace pthread parallelism with `#pragma omp simd`. The proposed methodology cannot be applied to all the distributed benchmarks: some of them do not contain DoAll loops or contain DoAll loops which do not satisfy the constraints listed in Section 4.

Different degrees of parallelism have been considered: 1 (absence of parallelism), 2, 4 and 8. For each degree and for each benchmark a different hardware accelerator is produced by *Bambu*. The tool has been configured with maximum level optimization (-O3), to store input and output data on a dual port pipelined memory external to the hardware accelerator and to target 100MHz frequency. Two target platforms have been considered: the Xilinx Zynq-7000 xc7z020 and the Altera Cyclone II EP2C70F896C6. The solutions produced by High Level Synthesis have been finally synthesized with Xilinx Vivado [18] and Altera Quartus II [1]. The synthesis results obtained after place and route on different benchmarks with different degrees of parallelism are presented in Table 1. The area results refer only to the synthesized accelerator since the produced parallel hardware architectures, differently from the ones presented in [2], do not require any external processor nor external controller to be integrated in the system. Memory utilization has not been reported since all the benchmarks have been synthesized assuming that input and output data are stored in external memories. The results obtained on the different platforms are similar, so that the proposed methodology can actually be considered as appliable to different families of FPGAs. Moreover the results show how it is effectively able to save

Benchmark	Par. Degree	Xilinx Zynq-7000 xc7z020				
		Area(Ratio)		Cycles(Speedup)	FMax (Ratio)	Product Ratio
		LUT FF Pairs	DSPs			
Add	1	344 (1)	0	20018 (1)	165.21 (1)	1
	2	573 (1.67)	0	10014 (2.00)	171.73 (1.03)	0.80
	4	1245 (3.61)	0	7512 (2.66)	157.98 (0.95)	1.42
	8	2571 (7.47)	0	6261 (3.19)	135.34 (0.81)	2.85
Boxfilter	1	3880 (1)	0	492910 (1)	95.43 (1)	1
	2	7100 (1.83)	0	176024 (2.80)	102.57 (1.07)	0.60
	4	14803 (3.81)	0	104022 (4.73)	103.19 (1.08)	0.74
	8	27966 (7.20)	0	70021 (7.03)	100.35 (1.05)	0.97
Dotproduct	1	567 (1)	3(1)	30024 (1)	131.03 (1)	1
	2	917 (1.61)	6 (2)	18020 (1.66)	107.50 (0.82)	1.18
	4	1700 (3.00)	12(4)	12018 (2.49)	116.69 (0.89)	1.34
	8	3332 (5.87)	24 (8)	9017 (3.32)	110.75 (0.84)	2.08
Hash	1	722 (1)	0	192041 (1)	121.64 (1)	1
	2	1442 (1.99)	0	108029 (1.77)	118.19 (0.97)	1.08
	4	2318 (3.21)	0	78025 (2.46)	109.69 (0.90)	1.35
	8	4570 (6.32)	0	63023 (3.08)	101.38 (0.83)	2.33
Histogram	1	2655 (1)	0	202101 (1)	129.99 (1)	1
	2	3751 (1.41)	6 (-)	72100 (2.80)	102.79 (0.79)	0.63
	4	6620 (2.49)	12 (-)	45094 (4.48)	116.65 (0.89)	0.62
	8	12158 (4.57)	24 (-)	31591 (5.74)	107.34 (0.82)	0.86

Benchmark	Par. Degree	Altera Cyclone II EP2C70F896C6				
		Area(Ratio)		Cycles(Speedup)	FMax	Product Ratio
		Logic Elements	DSPs			
Add	1	360 (1)	0	40018 (1)	166.39 (1)	1
	2	649 (1.80)	0	20014 (2)	161.32 (0.96)	0.92
	4	1495 (4.15)	0	12512 (3.19)	147.23 (0.88)	1.46
	8	2599 (7.21)	0	8671 (4.61)	139.02 (0.83)	1.87
Boxfilter	1	4643 (1)		529905 (1)	117.03	1
	2	8291 (1.78)		184028 (2.87)	102.87 (0.87)	0.70
	4	17924 (3.86)		108024 (4.90)	89.93 (0.76)	1.02
	8	Not Available				
Dotproduct	1	668 (1)	6 (1)	36025(1)	141.58 (1)	1
	2	1238 (1.85)	12 (2)	21021 (1.71)	123.43 (0.87)	1.24
	4	2101 (3.14)	24 (4)	13519 (2.66)	124.08 (0.87)	1.34
	8	3604 (5.39)	48 (8)	9768 (3.68)	128.25 (0.90)	1.61
Hash	1	850 (1)	0	288050 (1)	121.15 (1)	1
	2	1413 (1.66)	0	144034 (1.99)	103.91 (0.85)	0.97
	4	3000 (3.52)	0	96028 (2.99)	91.16 (0.75)	1.56
	8	4885 (5.74)	0	72025 (3.99)	95.50 (0.78)	1.82
Histogram	1	3161 (1)	0	238011 (1)	132.14 (1)	1
	2	4965 (1.57)	8 (-)	90114 (2.64)	112.57 (0.85)	0.69
	4	8191 (2.59)	16 (-)	54108 (4.39)	105.45 (0.79)	0.73
	8	15421 (4.87)	32 (-)	36103 (6.59)	107.35 (0.81)	0.91

Table 1. Experimental Results of applying the proposed methodology.

resources with respect to the complete duplication of loop implementation: the area of the produced solutions indeed growths less that the parallel degree. The maximum resource saving has been obtained for *Histogram* benchmark when targeting both platforms with parallel degree of 8: more than 40%. The resource saving however is not effective on the usage of DSPs: their number growths linearly in *Dotproduct* benchmark while in case of *Histogram* benchmark they have been introduced only in vectorized implementation. On *Boxfilter* (when P=2 and P=4) and on *Histogram* (when P=2) the obtained speed-up is more than linear (i.e., it is larger than parallel degree). Further gain with respect to the linear speed-up is due to the if-conversion preprocessing phase which allows to improve the performances of the circuit implementation even when vectorization is not applied. However, for all the benchmarks the real speed-up grows less than parallel degree. The main cause of this reduction in speed-up growing is the considered memory architecture which has only two ports. The number of ports limits the exploitation of parallelism since limits to two the number of simultaneous memory accesses. Even if memory accesses are pipelined, solutions where degree of parallelism is larger than 2 are slowed and cannot achieve maximum performances. Memory partitioning, by increasing the number of possible concurrent accesses, can solve this problem, but it is not supported by *Bambu*.

The introduction of vector functional units does not decrease very much the maximum frequency of the circuits. In the worst case (*Hash* benchmark implemented on Altera board with $P = 4$) the maximum frequency is reduced of 25%. Note that the increasing of the parallel degree does not always imply a decreasing in the maximum frequency. There are specifications (e.g., *Boxfilter* when implemented on Zynq) for which the introduction of vectorization increases the maximum frequency. The gain in terms of area-delay product for most complex benchmarks (e.g., *Boxfilter* and *Histogram*) is quite significant (up to 40% obtained on *Boxfilter* on Zynq with P=2) since the performances grow faster than resource utilization thanks to the if-conversion and to the local pipelined computation. However, there is a general gain in terms of area-delay product also for most of the other benchmarks when the considered parallel degree is 2. On the contrary, because of the performances limitations due to memory accesses, the solutions with higher parallel degree present worse results.

Finally, it has to be highlighted that direct comparison of the results of the proposed methodology and the results presented in [2] is not possible, not only for the different analyzed benchmarks but also for the different types of built architectures. Differently from [2] indeed, the parallel accelerators built with the proposed methodology do not require to be coupled with a controller processor. For this reason, this has not been included in the resource utilization statistics in non-vectorized architecture nor in the vectorized, resulting in smaller area occupations and in smaller area-delay savings.

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. Altera: Quartus II. http://www.altera.com (2013)
2. Choi, J., Brown, S., Anderson, J.: From software threads to parallel hardware in high-level synthesis for fpgas. pp. 270–277. FPT '13 (Dec 2013)
3. Cilardo, A., Gallo, L., Mazzocca, N.: Design space exploration for high-level synthesis of multi-threaded applications. Journal of Systems Architecture 59(10, Part D), 1171 – 1183 (2013)
4. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-level synthesis for fpgas: From prototyping to deployment. IEEE TCAD 30(4), 473–491 (April 2011)
5. Cong, J., Jiang, W.: Pattern-based behavior synthesis for fpga resource reduction. pp. 107–116. FPGA '08, ACM, New York, NY, USA (2008)
6. Fingeroff, M.: High-Level Synthesis Blue Book. Xlibris Corporation (2010)
7. Gupta, S., Savoiu, N., Kim, S., Dutt, N., Gupta, R., Nicolau, A.: Speculation techniques for high level synthesis of control intensive designs. pp. 269–272. DAC '01, ACM, New York, NY, USA (2001)
8. Hadjis, S., Canis, A., Anderson, J.H., Choi, J., Nam, K., Brown, S., Czajkowski, T.: Impact of fpga architecture on resource sharing in high-level synthesis. pp. 111–114. FPGA '12, ACM, New York, NY, USA (2012)
9. Kurra, S., Singh, N.K., Panda, P.R.: The impact of loop unrolling on controller delay in high level synthesis. pp. 391–396. DATE '07 (2007)
10. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. SIGMICRO Newsl. 23(1-2), 45–54 (Dec 1992)
11. Morvan, A., Derrien, S., Quinton, P.: Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis. IEEE TCAD 32(3), 339–352 (March 2013)
12. Naishlos, D.: Autovectorization in GCC. In: GCC Developers Summit 2004. pp. 105–118
13. Nuzman, D., Zaks, A.: Outer-loop vectorization: Revisited for short simd architectures. pp. 2–11. PACT '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1454115.1454119
14. OpenMP: Application Program Interface, version 4.0 (July 2013)
15. Papakonstantinou, A., Gururaj, K., Stratton, J.A., Chen, D., Cong, J., Hwu, W.M.W.: Efficient compilation of cuda kernels for high-performance computing on fpgas. ACM TECS 13(2), 25:1–25:26 (Sep 2013)
16. Pilato, C., Ferrandi, F.: Bambu: A modular framework for the high level synthesis of memory-intensive applications. pp. 1–4. FPL '13 (Sept 2013)
17. Raghunathan, V., Raghunathan, A., Srivastava, M., Ercegovac, M.: High-level synthesis with simd units. pp. 407–413. ASP-DAC '02 (2002)
18. Xilinx: Vivado Design Suite. http://www.xilinx.com (2013)
19. Zuo, W., Liang, Y., Li, P., Rupnow, K., Chen, D., Cong, J.: Improving high level synthesis optimization opportunity through polyhedral transformations. pp. 9–18. FPGA '13, ACM, New York, NY, USA (2013)