Graph-Based Code Restructuring Targeting HLS for FPGAs

Afonso Canas Ferreira[1,2(✉)] and João M. P. Cardoso[1,2]

[1] Faculty of Engineering, University of Porto, Porto, Portugal
ascferreira@gmail.com, jmpc@acm.org
[2] The Institute for Systems and Computer Engineering, Technology and Science, INESC TEC, Porto, Portugal

Abstract. High-level synthesis (HLS) is of paramount importance to enable software developers to map critical computations to FPGA-based hardware accelerators. However, in order to generate efficient hardware accelerators one needs to apply significant code transformations and adequately use the directive-driven approach, part of most HLS tools. The code restructuring and directives needed are dependent not only of the characteristics of the input code but also of the HLS tools and target FPGAs. These aspects require a deep knowledge about the subjects involved and tend to exclude software developers. This paper presents our recent approach for automatic code restructuring targeting HLS tools. Our approach uses an unfolded graph representation, which can be generated from program execution traces, and graph-based optimizations, such as folding, to generate suitable HLS C code. In this paper, we describe the approach and the new optimizations proposed. We evaluate the approach with a number of representative kernels and the results show its capability to generating efficient hardware implementations only achievable using manual restructuring of the input software code and manual insertion of adequate HLS directives.

Keywords: Software code restructuring · HLS · Graph transformations · FPGA · Hardware accelerators

1 Introduction

Field-programmable gate arrays (FPGAs) can provide efficient hardware accelerators. Their use can contribute to the performance improvements and energy efficiency needed in many computing systems (see e.g., [1]), from embedded to high-performance computing systems. Custom hardware implementations provide concurrent execution of many independent operations, thereby improving the execution of algorithms with high operation-, data- and task-level parallelism. In order to design efficient hardware accelerators, one must have specific

This work was partially supported by the TEC4Growth project.

C. Hochberger et al. (Eds.): ARC 2019, LNCS 11444, pp. 230–244, 2019.
https://doi.org/10.1007/978-3-030-17227-5_17

skills and understand very distinct programming languages and tools than a typical software developer. Additionally, hardware design is still a very error-prone and time-consuming task. As these aspects impose substantial barriers to use FPGAs as accelerators, many efforts in high-level synthesis (HLS) focus on improvements in terms of the use of FPGAs by developers (including software programmers) by providing higher abstraction levels and the use of typical software programming languages.

The high-level of abstraction provided by HLS tools thus intends to allow developers to program FPGAs more easily and be able to handle more complex applications, without the long time efforts needed by typical hardware design. Even thought most HLS tools start from software programming languages, they still require hardware expertise to generate efficient hardware. For example, the C programming language is a common input for many HLS tools [1]. However, the C programming model is tailored to CPUs and does not consider the concurrent nature of hardware and the possible customization. HLS tools circumvent these limitations by allowing programmers to guide the synthesis through directives. Nonetheless, the structure of the code has a large impact on the performance of the generated hardware via HLS [2]. Complex code restructuring is usually required and HLS tools and compilers may neither provide such optimizations nor ensure their automatic application. As it is well known that current HLS tools still have a barrier of entry for software programmers, by lowering this barrier more developers will be able to use the computing power of FPGAs, e.g., to accelerate applications. In order to make C-based HLS more accessible, we need a way to easily restructure the input software code.

This paper presents an approach to automatically restructure C code targeting HLS for FPGAs. Our approach is based on a dataflow graph (DFG), currently generated from execution traces of the input critical function, and on graph transformations, such as folding and unfolding, before generating C code added with HLS directives. Although a DFG could be generated by compilers, the current trace-based approach is taken, so that in future work we can also specialize the hardware generation with the use of runtime information. The global approach was firstly introduced in [3] and here we describe in more detail important aspects of the approach and provide useful extensions with significant impact in the results achieved. The main contributions of this paper are:

- an automatic code restructuring approach based on dataflow graph transformations and on a framework, partially implementing the approach, tuned to code restructuring and insertion of HLS directives for FPGA-based accelerators;
- graph-based optimizations allowing the generation of C code and considering different aspects such as folding/unfolding, loop pipelining, arithmetic optimizations and array partitioning;
- an evaluation of the approach using a number of kernels and results that show some advantages of the approach. This includes a comparison to the optimized code for an SVM implementation provided in [4] and evidence

of the capability of the approach and current framework to automatically achieve comparable code restructuring.

This paper is organized as follows. Section 2 presents a motivating example regarding code restructuring and HLS directives. Section 3 presents our approach and describes the framework developed to implement and evaluate the approach and the main optimizations already implemented. In Sect. 4, we show the results obtained by applying our framework to a number of benchmarks. We present in Sect. 5 some of the most relevant related work, and we finalize in Sect. 6 with some concluding remarks and planned future work.

2 Motivating Example

In order to show the possible code restructuring and HLS directives needed to achieve an efficient hardware accelerator, we show here the C code of the *filter subband* function (see Fig. 1), a function present in an MPEG audio encoder [5]. This function consists of a nested loop that calculates y values that are then used in a second nested loop to calculate the output array s.

```
void filter_subband(double z[Nz],
  double s[Ns], double m[Nm]){
  double y[Ny];
  int i,j;
  for (i=0;i<Ny;i++){
    y[i] = 0.0;
    for (j=0; j<(int)Nz/Ny;j++)
      y[i] += z[i+Ny*j];
  }
  for (i=0;i<Ns;i++){
    s[i]=0.0;
    for (j=0; j<Ny;j++)
      s[i] += m[Ns*i+j] * y[j];
 }
}
```

(a) Original *Filter subband* source code

```
void filter_subband_pipe(double z[512],
  double s[32], double m[1024]){
  #pragma HLS array_partition
   variable=s cyclic factor=16 dim= 1
  #pragma HLS array_partition
   variable=z cyclic factor=16 dim= 1
  #pragma HLS array_partition
   variable=m cyclic factor=64 dim= 1
    s[0]=0;
          ...
    s[31]=0;
  for( int i =0; i < 64; i=i+4){
  #pragma HLS pipeline
     part11=z[i+320] + z[i+256];
     part12=z[i+321] + z[i+257];
     part13=z[i+322] + z[i+258];
     part14=z[i+323] + z[i+259];
     ...
     y0=final_part1;
     y0_a10=final_part2;
     y0_a20=final_part3;
     y0_a30=final_part4;
     for( int j =0; j < 32; j=j+1){
       temp1=m[(32)*j+i] * y0;
       temp2=m[(32)*j+i] * y0_a10;
       temp3=m[(32)*j+i] * y0_a20;
       temp4=m[(32)*j+i] * y0_a30;
       partial_in1=temp1+temp2;
            ...
       final_partin=part_in3 + part_in4;
       s[j]=s[j] + final_partin;
     }
   }
}
```

(b) *Filter subband* restructured source code, added with Vivado HLS directives

Fig. 1. *Filter subband* source code considering Nz, Ns, Nm and Ny equal to 512, 32, 1024 and 64, respectively

Figure 1b shows the C code after code restructuring and insertion of Vivado HLS directives. This new C code of the *filter subband* provides an efficient FPGA implementation. Although this code implements the same algorithm, it has been restructured substantially and to the best of our knowledge none HLS tool is able to automatically apply the code restructuring stages needed to achieve the code presented. The code in Fig. 1b consists of a single nested loop instead of two nested loop structures. In each iteration of the new outermost loop, four y values are calculated and then used to calculate values for the output array s. The array y is promoted to scalar variables. The outermost loop is then pipelined in hardware due the use of the Vivado HLS pipeline directive. This representation of the algorithm leads to more efficient implementation than the original (Fig. 1a). When using the original code, the hardware resultant implementation calculates all the y values in the first nested loop, stores them in BRAMs, and use them in the next nested loop.

In the restructured code version, presented in Fig. 1b, the calculated y values are being used for the calculation of the output array. They can also be concurrently calculated in the pipeline, which would not be possible with the previous representation. Additionally, the calculated y values are used immediately to calculate the outputs, so they do not need to be stored in memory. Furthermore, the accumulations are implemented using partial sums that allow for more concurrent summations. Also, array partitioning directives are used to increase the memory throughput, so that the resultant loop pipelining has a lower initiation interval (II).

Although both codes implement the same algorithm, the restructured version generates a more efficient FPGA hardware. In the following section we show how our framework can automatically generate the C code in Fig. 1b from the C code in Fig. 1a.

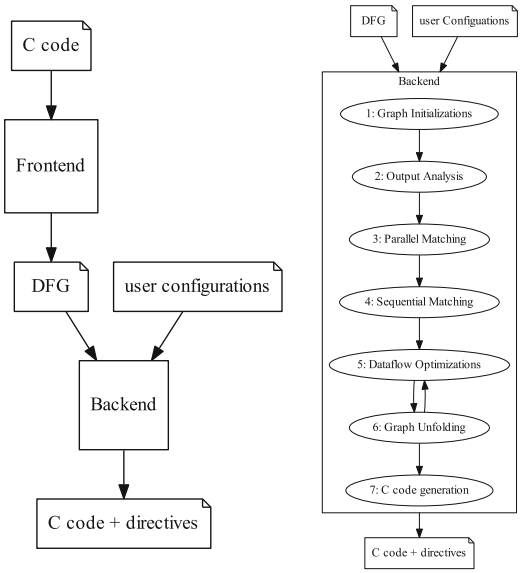
3 Our Approach

Our approach to automatically restructure C code targeting HLS tools is based on graph transformations. The current implementation of our approach (see Fig. 2a) consists of two main components: a frontend and a backend. The frontend transforms a given execution trace in a dataflow graph (DFG). Each DFG is then processed and optimized in a backend that, as final step, generates C code for a HLS tool.

We chose DFGs for our graph-based approach, as DFGs are tailored to represent the flow of data and naturally express parallelism, both essential for hardware implementations. Additionally, we focused on a flexible frontend to make possible the generation of DFGs from multiple input languages as this may allow programmers of different languages to use C-based HLS tools.

3.1 Frontend

As already mentioned, the frontend of our framework generates a DFG from an execution trace of the input code. Figure 3 shows a simplified DFG for the *Filter*



(a) framework main stages (b) backend main stages

Fig. 2. Compilation flow of our framework

subband function in Fig. 1. The DFG represents in nodes the operations from the original execution and in edges the data dependencies between operations. We kept the frontend as simple and generic as possible in order to address different input languages. Although our initial frontend was implemented for C code input, it can be easily ported to other software programming languages.

Our current approach to generate the DFG representing the execution of a kernel is to write the dot (GraphViz) description to a file at runtime. By injecting instrumentation code into the original C code (before each statement), compiling and executing, the input DFG is generated.

3.2 Backend

Currently, the backend consists of seven stages (see Fig. 2b) focused on analysis and optimizations of the DFG. It implements all the code restructuring, optimizations and insertion of directives for the target HLS tool. The exact optimizations applied depend on the input DFG and on the configurations provided by users. In a configuration file, users can define the number of simultaneous load/stores supported by the hardware - important for the tool to explicitly generate code with a number of load/store statements -, inputs and outputs of the kernel and optimization options.

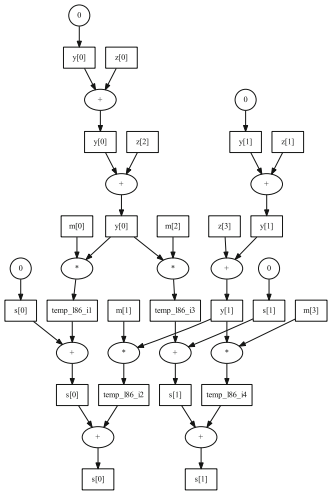


Fig. 3. DFG for the *filter subband* considering an execution with Nz, Ns, Nm and Ny equal to 4, 2, 1024, and 2, respectively

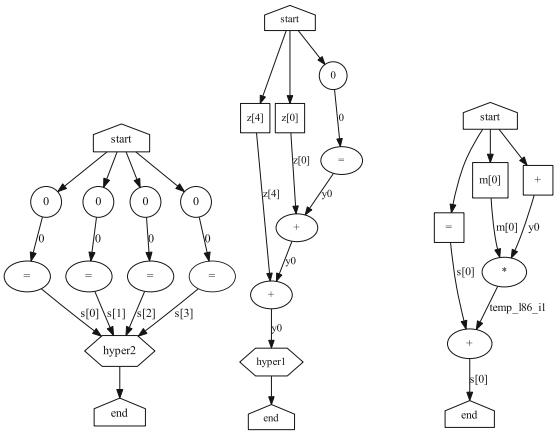
The first stage of the backend compacts the graph by pruning unnecessary nodes and prepares it for the following stages. Afterwards, the tool tries to obtain improved dataflow representations. Since currently the input DFG is fully unfolded, it is important to identify repeating patterns that can be folded. As these patterns may occur multiple times, the tool can optimize a large part of some applications by improving these patterns. Stage 2 separates the dataflows that generate each individual output and Stage 3 tries to find matches among these dataflows. If a repeating pattern is identified, this stage folds them into a loop that is represented by a dataflow of a single iteration. Stage 4 attempts to optimize the dataflow by identifying loop pipelining opportunities. Stage 5 applies various optimizations to the DFG and Stage 6 unrolls some of the generated loops based on the users' configurations. Finally, Stage 7 generates the C output code plus the appropriate HLS directives.

3.3 Backend Optimizations

In this subsection we describe in more detail the main backend stages (see Fig. 2b) that optimize the DFG. We describe Stage 6 before Stage 5 as graph unfolding has an impact on the graph optimizations.

Sequential Matching (Stage 4). Although the first three stages compact the DFG, it can still be very large and contain properties to be further explored. In Stage 4, the tool identifies a potential variable and pipelines the graph along this variable. This variable is selected by traversing the DFG and identifying which variable is written more often (an heuristic that attempts to build the longest pipeline).

The backend then proceeds to match all the dataflows that generate all the separate writes of the selected variable. After the tool has obtained the pipelining structure, it handles dataflows without matches. The tool moves the subgraphs that represent the dataflow of the pipelining into a "hyper" node (represents a loop) and all the nodes that do not fit in the pipelining are maintained outside of this "hyper" node.



(a) DFG of the highest hierarchy level (b) outer loop (c) inner loop

Fig. 4. DFG for *filter subband* after Stage 4 of the tool and considering an execution with Nz, Ns, Nm and Ny equal to 8, 4, 1024, and 4, respectively

By applying the Stage 4 pipelining to *filter subband*, the tool obtains the DFG shown in Fig. 4. The graph is pipelined along the array *s*. The subgraphs in Figs. 4b and c represent single iterations of the outer and inner loops of the pipelining. In each iteration, the outer loop calculates a y value, which is then used in the inner loop. In the inner loop, each y is used to calculate all the outputs of the *s* array. The subgraph in Fig. 4a shows the dataflows that do not match the pipelining. In this case, they represent the initialization of the *s* array.

It is through Stage 4 that the tool obtains the improved code structure depicted in the example in Fig. 1b. By transforming the DFG according to pipelining, the tool identifies a better structure for the algorithm. By comparing this DFG to the input DFG seen in Fig. 3, and although the input sizes are different, we can still recognize the patterns that are compacted into the pipelining in the smaller version of the input DFG.

Graph Unfolding (Stage 6). Stage 6 is dedicated to unfolding loops that were generated in stages 3 and 4. Unfolding loops opens new avenues for optimizations in Stage 5. As mentioned before, compacting the DFG is very efficient for optimizations, but to take further advantage of Instruction Level Parallelism the tool needs to unfold some of the loops. Due to the DFG-based approach, the tool can unfold a loop simply by copying the dataflow multiple times and updating the indexes of array accesses and appending a label to the new variables.

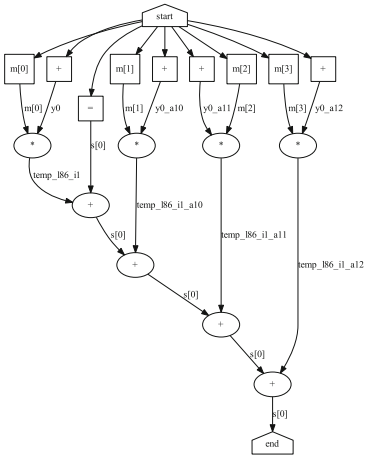


Fig. 5. Unrolled dataflow of the inner loop of the pipelined *filter subband* of Fig. 4c by a factor of four

The unfolding process starts at the innermost loops. After a loop is unrolled the resulting dataflow is checked for Stage 5 optimizations, and after these the resulting DFG returns to Stage 6. The unfolding process needs to be ordered from innermost to outermost loops, because unfolding an inner loop does not affect the outer loop, but unfolding an outer loop affects its nested loops. In case the outer loop is pipelined, the inner loop is not unrolled as Vivado HLS automatically unrolls it.

When dealing with inner loops, it is essential to distinguish which loop to unfold. Therefore, the tool starts the unrolling transformation with the name, unfolding factor and loop type of the initial loop. If that loop has a nested loop, the tool unrolls it and propagates the name of the outer loop, the unfolding factor and type. The inner loop is unrolled based on that inherited information. Thus, if a memory access depends on the iteration of the inner and outer loop, the tool can correctly identify how to calculate the index of the new access.

Figure 5 shows the result of unfolding the inner loop of *filter subband* pipeline loop shown in Fig. 4. The backend replicates the dataflow, in this case a sum and a multiplication, and then connects them to maintain the correct dependencies between iterations, resulting in the accumulation chain. It is through this unfolding the tool obtains the unfolded iterations seen in the code in Fig. 1b.

Dataflow Optimizations (Stage 5). Stage 5 is dedicated to various dataflow optimizations. Currently, it involves two types of optimizations. One focuses on arithmetic optimization, such as the accumulation optimization, which restructures an accumulation as partial sums. The backend substitutes accumulation chains with the same calculations through balanced trees. The result of the balanced tree is then summed with the starting value of the accumulation. In case the optimized chain consists of floats or doubles the user would need to verify if the result is within acceptable accuracy.

As illustrated in Fig. 5, the first addition depends on the result of the last sum. Therefore, the next stage of the pipelining can only initiate after the clock cycles necessary to execute that chain. However, by balancing the chain, the value calculated in the previous iteration is used only once at the final sum. Therefore, the pipeline only needs to be delayed for the duration of a single sum instead of an entire accumulation chain. It is through balancing that we obtain the partial sums in the motivating example (see Fig. 1b). Vivado HLS is able to automatically balance operations, but it does not balance floats or doubles without changing the settings, which would require the knowledge of an experienced user.

Another arithmetic optimization is applied to divisions. In this case, if at least one operand is unique to multiple divisions, the tool extends the DFG with the calculation of its inverse, and substitutes the divisions by multiplications with the inverse.

The user can also choose to optimize memory accesses. One of the optimizations is data reuse. The tool analyzes the current loops to identify if there are redundant memory accesses between two consecutive iterations of a loop. If the same memory location is also accessed in the next iteration of a loop, the tool uses buffers to store values between iterations, reducing the number of memory reads. This can greatly minimize the memory bottlenecks of certain applications.

Another optimization the user can choose to diminish memory bottlenecks is the full partitioning of arrays. This optimization can be applied through array partitioning directives provided by HLS tools. If the user chooses to fully partitioning the arrays, the tool makes a final pass through the whole DFG. Based on the number of separate concurrent accesses, the tool sets the appropriate array partitioning factor so that the maximum number of concurrent accesses detected can be scheduled in a single cycle. This optimization can significantly increase the resource usage. First by using more BRAMs. Second by lowering the memory bottleneck more operations can be executed in parallel. This optimization does not change the structure of the graph, it only leads to different directives. When applied to the *filter subband* function, this optimization injects the array partitioning directives included in the motivating example (see Fig. 1b).

4 Experimental Results

This section presents some experimental results achieved by our framework. The benchmarks used represent DSP algorithms and are either from the DSPLIB from Texas Instruments [5], the UTDSP Benchmark Suite [6] or from an MPEG audio encoder [7]. *dotproduct* and *Autocorrelation* are from DSPLIB. *1D fir* is a typical code implementing a FIR filter with N taps. *filter subband* is from an MPEG audio encoder. *2D Convolution* is the largest benchmark and is a kernel that performs a 2D convolution, which is part of the Sobel edge detection benchmark provided in UTDSP. The source code used for the *SVM* kernel is from [4].

Table 1. Framework optimization levels

Optimization level	Brief description
01	None of the optimizations
02	DFG is folded as much as possible, and unfolded according to user configurations
03	Adds array partitioning to level 02 to complement the unfolded DFG
04	Adds data reuse to level 03
05	Adds arithmetic optimizations to level 03
06	Adds arithmetic optimizations to level 04
07	Adds full array partitioning optimization from Stage 5 to level 05
08	Adds full array partitioning optimization from Stage 5 to level 06

Table 2. Versions of input code used for comparisons

Comparison code	Brief description
C	Original code without any modifications
C-inter	Input code optimized with basic directives such as the pipeline directive
C-high	Improve the C-inter implementation with unroll and memory partitioning directives

We analyze the effectiveness of our tool for multiple optimization levels as depicted in Table 1. The C code baselines are briefly summarized in Table 2. It is a fair assumption that a typical software programmer could use a number of very basic directives, but is not proficient with all types of directives. This approach to the evaluation allows us to study the effectiveness of our tool when comparing to different levels of hardware design knowledge.

Table 3. Resource usage for fastest optimization levels up to level 04 and Level 08

Benchmark	LUT	FF	DSP	BRAM	LUT	FF	DSP	BRAM
filter subband	12605	18849	59	0	47537	42589	118	0
Autocorrelation	9083	7277	160	0	8025	7114	160	0
dotproduct	294	581	8	0	294	581	8	0
1D fir	4587	6579	192	0	4297	5641	192	0
2D Convulution	5354	6575	54	0	6376	3408	57	0
SVM	9228	9068	56	68	14203	12506	91	76

Table 4. Speedups for fastest optimization levels up to level 04 and Level 08

Benchmark	Latency	Period (ns)	Speedup C	Speedup C-inter	Speedup C-high	Latency	Period (ns)	Speedup C	Speedup C-inter	Speedup C-high
filter subband	563	18.34	39.60	2.81	2.81	293	17.09	81.66	5.79	5.79
Autocorrelation	96	8.6	49.6	16.4	7.91	16	8.6	297.7	98.6	47.5
dotproduct	255	8.93	16.81	5.61	1.00	255	8.93	16.81	5.61	1.00
1D fir	135	8.74	211	26.7	14.4	120	8.74	237.3	30	16.2
2D Convulution	8563	8.74	34.5	2.25	1.36	3886	8.74	76.1	5	3
SVM	11365	9.38	31	0.9	0.9	3208	8.4	123.4	3.5	3.5

Speedups and FPGA resource values are obtained through synthesizing the C code with Vivado HLS 2017.4 [8], in a PC with an Intel Core i7-7700 with 32 GB RAM, and targeting a Xilinx Artix™-7 FPGA (xc7z020clg484-1). All of the benchmarks had a time constraint of 10 ns except *filter subband*, which has a constraint of 20 ns. The total time of each hardware implementation is calculated by multiplying the minimum clock period reported and the latency. The speedups are the result of dividing the total time of the implementations from Table 2 by the total time of the implementations from code generated with different framework optimizations levels.

Tables 3 and 4 show the results presented in [3], which considered 04 as the highest level. Level 03 achieves the fastest implementations for the *filter subband* and *dotproduct*. The remaining benchmarks achieve the fastest implementations at Level 04. The results showed that it was essential to reduce the memory bottleneck to increase the throughput of the implementations. Those results also show that just through folding and unfolding the input DFG, the resulting implementation was already faster for *Autocorrelation* and *filter subband*. Overall, the results show the benefits of our approach in terms of speedups and the enhancements when adding the optimizations (arithmetic optimizations and array partitioning) proposed in this paper.

Tables 3 and 4 also present the results from the synthesis reports of Vivado HLS for the benchmarks, considering the manually improved C versions and the C code automatically generated using our tool, with optimization levels between 05 and 08. With more optimizations it is possible to achieve higher speedups for every benchmark with the exception of *dotprod*. For *filter subband*, the highest speedup was achieved at Level 07 with 5.8× and 81.7× compared with C-high and C, respectively. This is due to improving the Level 03 pipeline with arithmetic optimizations and array partitioning, thereby improving the Latency and II of the pipelining. Level 08 would possibly achieve an even higher speedup, but would require a larger FPGA.

The *Autocorrelation* achieves the best speedup at Level 08, which is considerably higher than the previous best result. This is due to the fact that through data reuse the main loop of the kernel is highly optimized to the point that considerable clock cycles are dedicated to filling the buffers, thus partitioning the memory has a large impact. The same applies to the *1D fir*, but the increase is not as large due to less buffers being used. There is also a large improvement

for the *2D Convolution* benchmark whose best speedup went from 1.4× to 3× compared to C-high at Level 07. Compared to C and C-inter, the best speedups for this benchmark are 76× and 5×, respectively. As with the previous cases, the arithmetic and memory optimizations lead to a pipelined loop with both lower latency and an initiation interval value. Initially, Level 08 was expected to achieve the fastest implementation and it in fact achieves a lower latency than Level 07. However, the way Vivado HLS schedules the code, Level 08 results in a higher minimum clock period leading to a slower implementation when considering executions operating at the maximum clock frequencies.

In addition to the above benchmarks, we also applied our framework to the machine-learning *SVM* (Support Vector Machine) kernel presented in [4]. The results achieved by the framework are presented in Tables 3 and 4. Compared with previous benchmarks, Levels 02 to 06 have lower performances than C-high. The loop generated by the backend contains 36 accesses to the matrix that contains the support vectors. There is no redundancy between memory accesses and thus Level 04 has little impact. Level 05 shows that the arithmetic optimizations alone have little impact, merely an insignificant reduction of the latency compared to Level 03. In order to lower the bottleneck caused by the memory accesses, it is necessary to use array partitioning. By partitioning the support vectors, the speedup compared to C-high is only 1.22×. However, when array partitioning is combined with arithmetic optimizations as in levels 07 and 08 the speedups are 123× and 3.47× relative to C and C-high, respectively.

In [4] the authors optimize an FPGA implementation of the *SVM* kernel by manually restructuring the code, and then use design space exploration (DSE) for selecting parameter values and HLS directives. When comparing the code proposed by them with the one generated using our tool, there are many similarities. The main difference is that our tool does not partition the *SVM* kernel itself to increase concurrency. Our tool attempts to obtain a similar result through unfolding the outer loop and applying array partitioning directives. The rest of the optimizations proposed in [4] are very similar, such as balancing the accumulations in a tree, unrolling loops and applying pipeline and array partitioning directives. Thus, our tool automatically obtains a similar code compared to the optimized one shown in [4], depending on the users given configurations. These results show once again the capability of our framework to achieve efficient code restructuring plus HLS directives.

4.1 Limitations

The current version of the framework imposes restrictions on the input code to handle. Some limitations are due to the framework being at an initial stage, others are due to inherent characteristics of the approach. One limitation is related to the information loss through the execution tracing. As described, our approach simply represents the dataflow and executed operations and does explicitly represent constructs such as *for*, *while* or conditional statements at the frontend. The DFG at the frontend only represents the execution for the given inputs. Conditional statements or loops branching into different dataflow paths

depending on the inputs will conduct to different DFGs for the same input code. Currently, the existence of control-flow that may make a certain DFG invalid would require exit points or a decision about the execution on the accelerator. Our future plans consider the merging of DFGs representing different execution traces and the representation in the DFG of ternary conditional operators.

One of the major bottlenecks of the current implementation of our approach is related to scalability. The DFGs generated by the frontend are fully unfolded and represent each operation executed with a distinct DFG node. This results in large DFGs, even when input datasets and/or loop iterations are not so big. One possibility is to generate condensed DFGs by using expressions and parameters that represent the repetition of certain patterns. We recognize the importance to solve this problem and our future work plans include R&D of techniques to improve the scalability of our approach.

5 Related Work

Source to source transformations have been the subject of study in the field of HLS. For example, Cong et al. [9] presents a framework to facilitate code restructuring for software developers. Cardoso et al. [10] present an approach to allow users to program strategies to apply code transformations and insertion of directives. The LegUP HLS tool [11] also accepts C as an input and implements code restructuring through a modified LLVM compiler [12] to implement HLS optimizations.

Although the previous work efforts on code restructuring for HLS, it is well known that the problem is complex and difficult to make automatic as in many cases to achieve the required code a sequence of specific optimizations is needed [10]. Furthermore, in this sequence of optimizations there might be needed compiler optimizations that *per se* do not justify their inclusion in a typical compiler, and the selection of the optimizations (and associated parameter values) and the way to devise their sequence of application require exploration of a large design space.

Also relevant are the approaches dealing specifically with data streaming based computations. For example, Mencer et al. [13] present an approach that uses a C-based language called ASC to implement data-streaming based computations in hardware. With some similarities, the Max-Compiler [14] is a HLS tool to implement streaming computations described as dataflow graphs in a programming language based on Java and named as MaxJ. In [15] the authors discuss DFG optimizations for generating better FPGA implementations in the context of the MaxJ compiler. Most of these optimizations are also suitable for our approach.

A pertinent approach to source to source code optimizations is the inclusion of loop transformations based on polyhedral models [16] as presented, e.g., by Cong et al. [9]. For example, the polyhedral models focused on nested loops transformations can be used to optimize code, so that HLS tools can implement improved pipelines in hardware [17,18]. Although polyhedral models can only be

successfully applied to nested loops with specific structure, memory accesses and predetermined upper and lower bounded loops, a future analysis and comparison with the approach presented in this paper is required.

Our approach addresses code restructuring as a graph transformation problem and can automatically achieve more aggressive code restructuring. Although in the current work we consider C code as input, our approach has the potential to address different input programming languages via the inclusion of adequate instrumentation code. We also believe that our approach can target programming models such as the one used by MaxJ and in this case our approach could possibly act as an optimizer for the MaxCompiler.

6 Conclusion

This paper presented an automatic code restructuring approach to output software code more suitable to high-level synthesis (HLS) tools. Our approach starts with a dataflow graph (DFG) representation of the computations, currently obtained by executing the critical functions of the application previously added with instrumentation code, followed by graph optimizations and folding/unfolding graph operations. The proposed approach has been implemented in a framework able to automatically optimize DFGs to fully generate HLS-friendly C code added with HLS directives. The experimental results show that the C code automatically generated by our tool outperforms the original code (including the insertion of HLS directives) by achieving significant speedups. The restructured C code is even comparable to, and in most cases better than, manually optimized C code added with directives. Although the C code plus directives generated by the tool can be always replicated by manual code transformations applied by experts, our approach can enable software developers to target efficient hardware accelerators using HLS tools as backend and without requiring support of HLS experts.

We note however that our framework is at the moment a proof of concept for our approach and further work needs to be done to improve it. Ongoing work is focused on the generation of DFGs and on additional DFG optimizations. Future work will focus on more complex memory optimizations through analyses of the DFG, and on parameterized schemes to make possible to represent large execution traces in a more compact DFG.

Acknowledgments. This work was partially funded by the project "NORTE-01-0145-FEDER-000020", financed by the North Portugal Regional Operational Programme under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF) through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme, and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016883.

References

1. Nane, R., et al.: A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **35**(10), 1591–1604 (2016)
2. Cardoso, J.M.P., Weinhardt, M.: High-level synthesis. In: Koch, D., Hannig, F., Ziener, D. (eds.) FPGAs for Software Programmers, pp. 23–47. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26408-0_2
3. Ferreira, A.C., Cardoso, J.M.P.: Unfolding and folding: a new approach for code restructuring targeting HLS for FPGAs. In: FSP Workshop 2018: Fifth International Workshop on FPGAs for Software Programmers, Dublin, Ireland, pp. 1–10 (2018)
4. Tsoutsouras, V., et al.: An exploration framework for efficient high-level synthesis of support vector machines: case study on ECG arrhythmia detection for Xilinx Zynq SoC. J. Sig. Process. Syst. **88**(2), 127–147 (2017)
5. Texas Instrument, TMS320C6000 DSP Library (DSPLIB). Accessed 16 June 2018. http://www.ti.com/tool/sprc265
6. Lee, C.G.: 15 August 2002. http://www.eecg.toronto.edu/~corinna/. Accessed 16 June 2018
7. Cardoso, J.M.P., et al.: REFLECT: rendering FPGAs to multi-core embedded computing. In: Cardoso, J., Hübner, M. (eds.) Reconfigurable Computing, pp. 261–289. Springer, New York (2011). https://doi.org/10.1007/978-1-4614-0061-5_11
8. Xilinx. Vivado design suite user guide: high level synthesis, 20 December 2017
9. Cong, J., Huang, M., Pan, P., Wang, Y., Zhang, P.: Source-to-source optimization for HLS. In: Koch, D., Hannig, F., Ziener, D. (eds.) FPGAs for Software Programmers, pp. 137–163. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26408-0_8
10. Cardoso, J.M.P., et al.: Specifying compiler strategies for FPGA-based systems. In: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, pp. 192–199, April 2012
11. Canis, A., et al.: LegUP: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems. ACM Trans. Embed. Comput. Syst. **13**(2), 24:1–24:27 (2013)
12. LLVM. The LLVM compiler infrastructure project (2018). https://llvm.org
13. Mencer, O.: ASC: a stream compiler for computing with FPGAs. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **25**(9), 1603–1617 (2006)
14. Maxeler Technologies. Maxcompiler white paper (2017)
15. Voss, N., et al.: Automated dataflow graph merging. In: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS 2016), pp. 219–226, July 2016
16. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008), pp. 101–113. ACM, New York (2008)
17. Zuo, W., Liang, Y., Li, P., Rupnow, K., Chen, D., Cong, J.: Improving high level synthesis optimization opportunity through polyhedral transformations. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 2013). ACM, New York, pp. 9–18 (2013)
18. Morvan, A., Derrien, S., Quinton, P.: Polyhedral bubble insertion: a method to improve nested loop pipelining for high-level synthesis. Trans. Comput.-Aided Des. Integr. Circ. Syst. **32**(3), 339–352 (2013)