A Systematic Approach to Design and Optimise Streaming Applications on FPGA Using High-Level Synthesis

Mohammad Hosseinabady and Jose Luis Nunez-Yanez
Department of Electrical and Electronic Engineering University of Bristol, UK.
Email: {m.hosseinabady, j.l.nunez-yanez}@bristol.ac.uk

***Abstract*—This paper proposes a systematic approach to help designers to optimise a given streaming application for FPGAs using High-Level Synthesis (HLS). The proposed technique specifically addresses the two main issues in a streaming application that are determining the exact amount of loop unrolling in the HLS code to increase the throughput and finding the optimum buffers' size to prevent deadlocks. To evaluate the proposed techniques two applications from the machine learning optimisation area are studied in the paper. These applications are Hessian-vector product and Conjugate Gradient (CG). The experimental results show up to 38x speed-up in throughput compared to the original streaming implementations provided by knowledgeable engineers using the dataflow, loop pipelining and FIFO channel related pragmas provided by the HLS tool. In addition, these applications show up to 2.98 GB/sec usage of memory bandwidth which is 93.1% of the total memory bandwidth available on the system. The source codes of the designs are available at https://github.com/Hosseinabady/csdfg-hls.**

I. Introduction

Recently High-Level Synthesis (HLS) has emerged as a promising approach for designing computation and communication tasks. HLS tools translate an application described in a high-level language such as C/C++/SystemC/OpenCL into a high-performance Register Transfer Level (RTL) code which can later be synthesised into a netlist to be used for the ASIC fabrication or FPGA configuration. However, describing algorithms in C/C++ to be synthesised efficiently is a challenge that requires proposing systematic approaches.

This paper proposes a systematic approach to provide a high-performance HLS implementation of an intrinsic streaming application. The main technique of this paper in providing a high-performance design is to saturate the memory bandwidth by transferring data to the FPGA that directly take part in the computation. Fig. 1 shows the overview of the proposed methodology. This approach starts with a formal modelling of the application (i.e., Path (1) in Fig. 1) which can be used to optimise the throughput and memory-bandwidth of the corresponding HLS description. The proposed formal model is based on the Cyclo-Static Data Flow Graph (CSDFG) [1] which will be augmented with parameters (i.e., Path (3) in Fig. 1) provided by the HLS tool synthesising the original implementation of the design (i.e., Path (2) in Fig. 1). The optimization techniques direct the designers (i.e., Path (4) in Fig. 1) to add HLS pragmas to the application code and use a specific coding structure to improve performance and memory throughput. This paper addresses Paths 3 and 4 and assumes that designers are already familiar with Paths 1, 2, and 5. We propose to use the actor initiation interval (*II*), obtained by an HLS tool, instead of the common notion of time in the CSDFG. This helps us to propose a clock-based definition for throughput and buffers' length. Using the modified CSDFG two techniques are proposed to reduce the *II* and to detect artificial deadlocks. Whereas the former, called

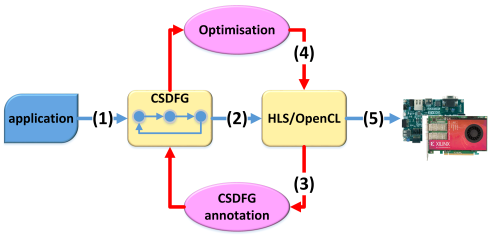


Fig. 1: Proposed systematic design flow

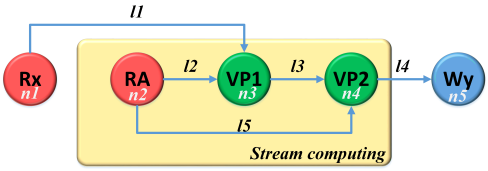
token grouping, increases the throughput the latter determines the minimum buffer length to prevent deadlocks.

For evaluation, we applied the proposed approach in designing two optimisation algorithms used in the area of machine leaning as case studies. The first algorithm is Hessian-vector product which is a compute-intensive operator used in logic regression. The second algorithm is Conjugate Gradient (CG) which is an iterative optimisation algorithm used in the deep learning area. The results of running these applications on the Xilinx Zynq SoC shows up to 38x speed-up in the throughput compared to the original streaming implementations provided by knowledgeable engineers using the dataflow, loop pipelining and FIFO channel related pragmas provided by the HLS tool. Furthermore, the applications consume up to 93.1% of the total bandwidth available on the four 64-bit High-Performance (HP) ports at 100MHz which is 3.2 GByte/sec.

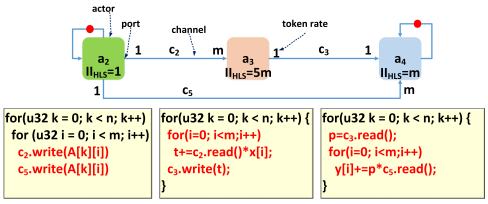
The rest of this paper is organised as follows. The next section clarifies the motivations behind this paper and explains our contributions. Section III briefly explains the previous work. The details of the proposed methodology are explained in Section IV. Section V shows how to use the proposed approach in designing two algorithm as case studies. Finally, Section VI concludes the paper.

II. Motivations and Contributions

Using a constructive example, this section explains the motivations and contributions of this paper. Let's consider $y = A^T Ax$ operator as a running example in which A is a matrix of size $n \times m$, and x and y are two vectors of size m. This operator and its modified versions appear in many applications such as equation and eigenvalue solvers, iterative Newton methods for logistic regression [2]. An implementation can perform this operator using two matrix vector product operations. The first product calculates the $z = Ax$ expression and the second one produces the output $y = A^T z$. The dataflow graph in Fig. 2a models this operator. This graph consists of five nodes in which Node $n1$ reads the x vector into a memory accessible by Node $n3$. The $n2$ node reads matrix A from the main memory and sends it to Node $n3$ and $n4$. Nodes $n3$ and $n4$ perform the two aforementioned matrix vector products. Finally, Node $n5$



(a) A dataflow graph for $y = A^T Ax$ operator



(b) An SDFG for Nodes $n2$, $n3$ and $n4$

Fig. 2: $y = A^T Ax$ graphs

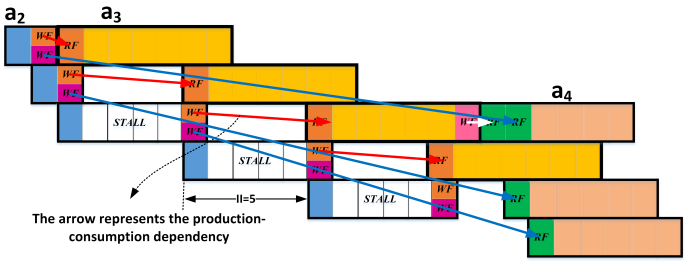


Fig. 3: Simplified timing diagrams of Fig. 2b

writes the results back to the main memory. The three $n2$, $n3$ and $n4$ nodes can perform their tasks in a streaming fashion which is shown by the corresponding Synchronous Dataflow Graph (SDFG) in Fig. 2b. This SDFG consists of three Actors a_2, a_3 and a_4 corresponding to Nodes $n2$, $n3$ and $n4$ in Fig. 2a, respectively. When the a_2 actor fires, it reads an element of A and pushes the corresponding token into channels c_2 and c_5. Then, Actor a_3, which has access to the x vector, reads these tokens and for every m input tokens generates an output token pushed into the c_3 channel towards the a_4 actor. Finally, the a_4 actor receives one token from c_3 and m tokens from c_5 in order to update the y vector. Below each actor in Fig. 2b, there is the corresponding C++ code consisting of nested loops that traverse over the input domain. Synthesising the pipelined version of these C++ codes by the Xilinx Vivado-HLS determines the minimum initiation interval (*II*) of actors that are $II_{a2} = 1$, $II_{a3} = 5$ and $II_{n4} = 1$. These minimum *II* may not be held when the actors are connected through FIFOs to collaborate in a stream computing fashion, as shown in Fig. 3. Since data generation and consumption rates of two actors a_2 and a_3 are not the same, thanks to their different *II*s, a_2 should wait for an empty space in the FIFO implementing the c_2 channel. This inserts some stalls into its pipeline stages which increases its real initiation interval to 5. This reduces the memory bandwidth usage by the a_2 actor, consequently, reduces the design throughput. As the c_5 channel connects the two Actors a_2 and a_4 which are apart in the pipelined timing diagram, its corresponding FIFO should have enough room to keep the tokens generated by the a_1 actor until they are consumed by a_4. A short length FIFO implementing the c_5 channel causes a deadlock in the design.

In summary, maximising the throughput and avoiding deadlocks are the main contributions of the proposed techniques in this paper.

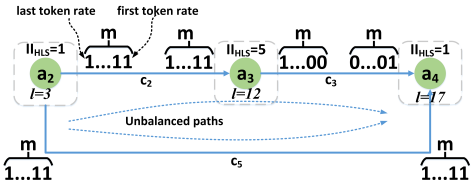


Fig. 4: A CSDFG for $y = A^T Ax$ operator

III. Previous Work

Synchronous Dataflow Graph (SDFG) and its generalisations such as CSDFG, Multidimensional SDFG have been studied extensively in modelling and designing streaming applications. Stuijk et al. [3] study the trade-off space between throughput and buffer size in cyclo-static and synchronous data flow graph. For this purpose they propose a systematic technique to formulate the problem and approximate the optimum buffer-size. Similarly, Benazouz et al. [4] minimizes the cyclo-static dataflow graph using new formulation. However, they have not considered the pipelined actors and the overlap between execution of two consecutive firings of an actor. Unlike this technique, our approach tries to explain the capability of CSDFG in modelling streaming applications to be synthesised by an HLS tool.

IV. Proposed Methodology

This section first defines the CSDFG modelling tool, augmented by the initiation interval and formulates the throughput and the memory bandwidth utilisation of a design. Then, using the parameters of throughput and bandwidth, it proposes the optimisation techniques.

A. CSDFG+II

A CSDFG is a directed graph in which nodes represent tasks, known as *actors*, and links denote communication media or channels between tasks. An actor in a cyclo-static dataflow graph (CSDFG), known as a *multi-phase actor*, has a periodic sequence of firings (or phases) with the size of N, each of which consumes a fixed number of tokens. Actors may execute different functions, defined by $f_{N-1}, ..., f_1, f_0$ during their phases. The consumption and production rates of tokens on each channel are represented by a sequence of numbers in contrast to the only one token rate in SDFG. The rates at each port p (where a channel is connected to an actor) are denoted by $t_{N-1}(p), ..., t_1(p), t_0(p)$. For the sake of simplicity, in this paper, we assume that each actor consumes and produces its tokens at the first and last lock cycle in each phase, respectively. Fig. 4 shows the CSDFG corresponding to the SDFG in Fig. 2b. In this paper, we restrict our discussion to acyclic CSDFG as the HLS tool considered in this paper only support this type of streaming dataflow synthesis. Note that for the sake of simplicity self-loops around each actor which represent their states (e.g., loops' indices) are omitted. In this case, each actor has $N = m$ phases.

Initiation Interval (*II*): The initiation interval of two consecutive phases of an actor (denoted by $II_i(a)$ for phases i and $i + 1$ of the a actor) is defined as the number of cycles between the starting point of the two phases. In this paper, we assume that the lower bound of the number of cycles between all consecutive phases of a given actor a are equal and denoted by $II_{HLS}(a)$ determined by an HLS tool. Therefore, $\forall i \in \{0, 1, ..., N-2\}, II_i(a) \geq II_{HLS}(a)$

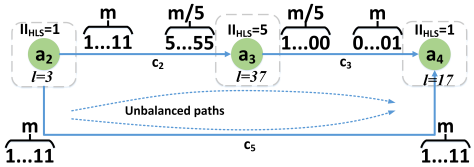


Fig. 5: Token grouping for $y = A^T Ax$ operator

B. Throughput Optimization

Equ. 1 represents the upper-bound of throughput in a CSDFG augmented with *II*.

$$Thr(a) \leq \frac{\sum_{i=0}^{i=N(a)} t_i(a)}{\sum_{i=0}^{i=N(a)} II_i(a)} \leq \frac{\sum_{i=0}^{i=N(a)} t_i(a)}{N(a) II_{HLS}(a)} \quad (1)$$

There are three main techniques to improve this throughput: utilising the wide-bus width, employing multiple memory ports and reducing the initiation interval. The experimental results show the impact of the first two techniques and the detail of the last one is described in the sequel.

The main factor determining the throughput of a streaming application mapped on FPGA is the achievable *II* at runtime. The initiation interval depends on the design and HLS tools and there are a few general techniques to improve it in a given HLS tool. In this paper, we propose a structural technique to increase the CSDFG throughput in which the actor with high *II* processes more than one token in its firing which is explained in the sequel.

Let's consider a computational actor a with $II(a) = k > 1$ and $N(a) = m$. Approximately, it takes km cycles to finish its task. However, if the actor can process k tokens in each phase without changing its II then $N(a) = m/k$ and consequently the number of cycles to finish the task reduces to $(m/k)k = m$. This technique is called *token grouping*. For example, the initiation interval of actor a_3 in Fig. 4 is 5 then by utilising this technique the throughput increases by a factor of 5. The modified CSDFG is shown in Fig. 5 in which the a_3 actor consumes all the tokens generated by actor a_2 with $II(a_2) = 1$ and still the FIFO implementing channel c_2 has the length of 1.

To increase the throughput of the CSDFG, the bottleneck actor which has high II should be found and unrolled with the factor of its II without changing its original initiation interval. This situation is possible for many compute-intensive applications. Section V shows this situation for two case study benchmarks. A systematic way to implement the token grouping is using the partial loop unrolling which is one of the basic optimization techniques in HLS tools.

C. Bounded Buffer

An insufficient communication buffer between two actors may lead to stall insertions into pipeline stages or even may induce an artificial deadlock mainly because of an unbalanced latency of two divergent paths that converge at an actor. This latency can have two sources: the *input data-size* and the *design structure*. For example in our running example of Fig. 4, the two paths $a_2 \rightarrow a_3 \rightarrow a_4$ and $a_2 \rightarrow a_4$ can cause a deadlock if the size of buffer implementing the c_5 channel is not sufficient to keep the tokens for actor a_4 when it is required. Actors a_2 and a_4 are the source and sink of the reconvergent paths, respectively. Actor a_4 can consume the tokens on the c_2 channel on each phase, as based on the graph, the source actor a_2 is able to provide them. However, actor a_4 cannot immediately consume a required token from the c_3 channel, as actor a_3 provides that with a delay on its last phase, after $II(a_3).m + l_{a_3}$ cycles, in which $II(a_3)$ is the initiation interval of a_3, m is the number of iterations that a_3 requires to generate a token on the c_3 channel and l_{a3} is the latency of this actor. Therefore, the first path i.e., $a_1 \rightarrow a_2 \rightarrow a_3$ has the latency of $mII + l_{a_3}$ cycles to deliver the first token to actor a_3 while during this time actor a_2 generates tokens that should be buffered in c_3. Since $II(a_2)$ determines the rate of a_2 generated tokens, then the lower bound of buffer size should be $(mII(a_3)+l_{a_3})/II(a_2)$. According to the discussion in previous subsection $II(a_2) = II(a_3) = 5$ and the latency of a_3 actor is 12 obtains after synthesis. Therefore, the minimum buffer size on the c_3 channel is $\lfloor m+12/5 \rfloor = m+3$. Note that, if the actor reads the token on channel c_3 with an offset of s clocks from the start of the iteration, it should be added to the length of the buffer.

V. Case Studies

This section explains two tasks used in machine learning considered as case studies. The source codes can be found at out Github site [5].

A. Hessian-vector product

The Hessian-vector product is defined as Equ. 2 in which d is a vector of size N and I, X and D are identity, normal and diagonal matrices of size $N \times N$, respectively. Note that the D can be represented by a vector of size N. Similar to the approach in [2], the right hand side of this equation can be calculated as four terms connecting together serially: Xd, $D(Xd)$, $X^T(DXd)$ and $d + X^T(DXd)$. In the rest of this paper, we assume the size of matrices and vectors in the Hessian-vector product are 1000×10000 and 1000, respectively. In the sequel, we will apply our design flow shown in Fig. 1 to this example.

$$\nabla^2 f(w) = (I + X^T DX)d \quad (2)$$

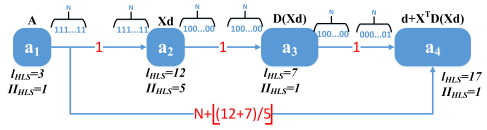
Path (1) in Fig. 1: Fig. 6a shows the corresponding CSDFG of a streaming implementation that a knowledgeable engineer who is familiar with stream computing will achieve using the pragmas available in the HLS tool. This CSDFG has a similar structure as the running example except for the extra actor and channel.

Path (2) in Fig. 1: The designer may use the RTL simulation provided by the HLS tool to determine the channels' buffer size through a trial and error effort or may be consider a high value for buffers' size. The second row of the table in Fig. 7b, denoted by *or* label, shows the resource utilisation of this implementation after synthesising by the Xilinx Vivado-HLS tool. The execution time of this implementation is about $50.2msec$ as shown in Fig. 7a.

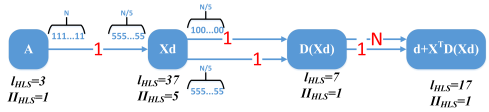
Path (3) in Fig. 1: The HLS synthesis also reveals the actors' *II* which are shown in the CSDFG of Fig. 6a. The two path $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4$ and $a_1 \rightarrow a_4$ in Fig. 6a converge with unbalanced latency. Therefore, the lower bound for the buffer size on path $a_1 \rightarrow a_4$ is $\lfloor N + (l_{a_2} + l_{a_3})/II_a \rfloor = N + 4$.

Path (4) in Fig. 1: Note that the token grouping can be applied to Actor a_2 as it is the bottleneck in achieving the high throughput thanks to its high *II*. Applying the token grouping on actor a_2 by processing 5 tokens (equal to its initiation interval) in each iteration improves its throughput by a factor of 5 (about 5 times faster as shown in Fig. 7a). In addition its corresponding resource utilisation is reported in the third row of table in Fig. 6a. The wide-bus and multiple memory port utilisations techniques focus on actor a_1 to improve the throughput. The fourth and fifth rows of table in Fig. 6a shows the corresponding resource utilisation of considering wide bus and multiple memory port, respectively.

Path (5) in Fig. 1: Fig. 7a depicts that applying all optimization techniques on the Hessian vector product speeds up the implementation by a factor of 38.0 compared to the original

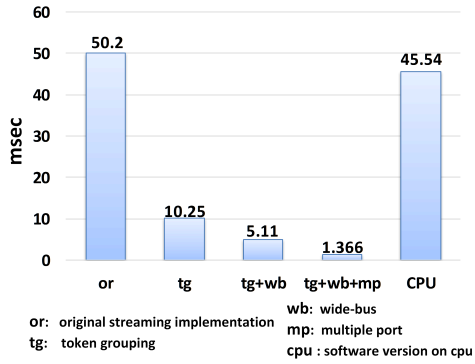


(a) Original CSDFG



(b) Modified CSDFG

Fig. 6: Hessian-vector product CSDFG



(a) Execution time

Implementation	LUTs(53200)	BRAM(140)	DSP(220)
or[1]	3389	8	15
tg[2]	3726	9	15
tg+wb[3]	5682	31	27
tg+wb+mp[4]	19896	125	108

or = original streaming implementation
tg = token grouping **wb** = wide-bus
mp = multiple memory port

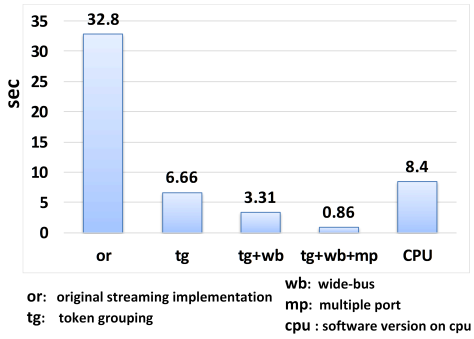
(b) Hardware resource usage

Fig. 7: Hessian-vector product

implementation. The CPU implementation on one core of the ARM Cortex-A9 available on the Zynq is also shown in this diagram for comparison. The total number of bytes transfer to the FPGA during this time is $1000 \times 1000 * 4 = 4MB$, therefore, the total memory bandwidth utilisation in the most optimised implementation is $4MB/1.366ms = 2.93GB/sec$. Note that, each 64-bit HP port on the Zynq SoC with 100MHz clock frequency can transfer 64-bit (i.e., 8 bytes) data in each clock cycle between the main memory and FPGA using the burst data transfer protocol. Therefore, the total memory bandwidth, provided by the four HP ports, is $8 * 100 * 4 = 3.2GB/sec$ of which $2.93GB/sec$ is used by our optimised design which shows 91.5% efficiency.

B. Conjugate Gradient (CG)

The CG method proposes an iterative algorithm to solve a linear equation denoted by $Ax = b$ in which A is a symmetric and positive-defined matrix of size $N \times N$ and x and b are vectors of size N [5]. Fig. 8a compares the execution time of the optimisation techniques (i.e., token grouping, wide bus utilisation and using five memory ports in parallel) applied to this solver with the matrix of size 4000×4000 and 40 iterations. The total number of bytes transfer to the FPGA during this time is $4000 \times 4000 * 4 * 40 = 2.56GB$, therefore, the total maximum memory



(a) Execution time

Implementation	LUTs(53200)	BRAM(140)	DSP(220)
or[1]	6554	41	32
tg[2]	7210	41	32
tg+wb[3]	8270	42	42
tg+wb+mp[4]	17132	64	102

or = original streaming implementation
tg = token grouping **wb** = wide-bus
mp = multiple memory port

(b) Hardware resource usage

Fig. 8: Conjugate gradient

bandwidth utilisation is $256MB/0.86sec = 2.98GB/sec$ which shows $2.98/3.2 = 93.1\%$ efficiency.

VI. Conclusions and Future Work

This paper has proposed a systematic design flow for streaming applications to be synthesised by a high-level synthesis tool for mapping on a target FPGA-based embedded system. The design flow utilises CSDFG model to describe the application. This model is annotated with information obtained from the HLS tool. The CSDFG is used to estimate the buffer size used as the communication channels between actors. Applying the proposed design flow to two applications (Hessian-vector product and conjugate gradient) shows its capability to detect bottlenecks and improve their execution time by a factor up to 38x.

References

[1] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static data flow," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, vol. 5, May 1995, pp. 3255–3258 vol.5.

[2] M. C. Lee, W. L. Chiang, and C. J. Lin, "Fast matrix-vector multiplications for large-scale logistic regression on shared-memory systems," in *Data Mining (ICDM), 2015 IEEE International Conference on*, Nov 2015, pp. 835–840.

[3] S. Stuijk, M. Geilen, and T. Basten, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1331–1345, Oct 2008.

[4] M. Benazouz, O. Marchetti, A. Munier-Kordon, and T. Michel, "A new method for minimizing buffer sizes for Cyclo-Static Dataflow graphs," in *2010 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, Oct 2010, pp. 11–20.

[5] M. Hosseinabady, "A systematic approach to design and optimise streaming applications on fpga using high-level synthesis: Source code," 2017. [Online]. Available: https://github.com/Hosseinabady/csdfg-hls