Constructive Synthesis of Memory-Intensive Accelerators for FPGA from Nested Loop Kernels

Matthew Milford, *Member, IEEE,* and John McAllister, *Senior Member, IEEE,*

Abstract

Field programmable gate array are ideal hosts to custom accelerators for signal, image and data processing but demand manual register transfer level design if high performance and low cost are desired. High level synthesis reduces this design burden but requires manual design of complex on-chip and off-chip memory architectures, a major limitation in applications such as video processing. This paper presents an approach to resolve this shortcoming. A constructive process is described which can derive such accelerators, including on and off-chip memory storage from a C description such that a user-defined throughput constraint it met. By employing a novel statement-oriented approach, dataflow intermediate models are derived and used to support simple approaches for on/off-chip buffer partitioning, derivation of custom on-chip memory hierarchies and architecture transformation to ensure user-defined throughput constraints are met with minimum cost. When applied to accelerators for full search motion estimation, matrix multiplication, sobel edge detection and fast fourier transform it is shown how real-time performance up to an order of magnitude in advance of existing commercial HLS tools is enabled whilst including all requisite memory infrastructure. Further, optimisations are presented which reduce the on-chip buffer capacity and physical resource cost by up to 96% and 75% respectively, whilst maintaining real-time performance.

Index Terms

Field Programmable Gate Array (FPGA), Processor, Streaming, Motion Estimation, Matrix Multiplication, Fast Fourier Transform (FFT)

I. Introduction

Modern Field Programmable Gate Array (FPGA) are highly diverse technologies, ranging between variants composed entirely of programmable logic to ’MPSoC’-FPGA, which add on-chip processors, coprocessors and Graphics Processing Units (GPUs). An important problem facing FPGA-based system designers is the creation of ’accelerators’ - custom circuits which perform a given function with high performance and low cost.

The resources available on modern FPGA for construction of these accelerators are unprecedented, with per-second access to billions of multiply-add operations and billions of bits of memory enabled via on-chip DSP units

Authors undertook this work at the Institute of Electronics, Communications and Information Technology (ECIT), Queen’s University Belfast, UK e-mail: jp.mcallister@qub.ac.uk

[1][2], Block RAM (BRAM) [3][2] and Distributed RAM (DisRAM). However, realising high performance, low cost accelerators has traditionally required manual Register Transfer Level (RTL) design of datapath and memory architectures, a task rendered increasingly unproductive by the scale of modern FPGA.

High Level Synthesis (HLS) has been proposed fo high-productivity accelerator development by deriving RTL architectures from programs written in, for example, C, C++ and OpenCL [4]. However, in applications which handle large volumes of data, such as image or video processing, memory architecture is a critical design concern typically requiring multi-level hierarchies of on-chip and off-chip buffering. Deriving such structures to meet real-time performance requirements is an open HLS problem [4].

This paper presents an approach to overcoming this limitation. By extending foundation work in [5][6] a constructive [7] approach is presented which derives FPGA RTL accelerators, including all requisite on-chip and off-chip memory storage, from a C kernel specification to meet a user-defined throughput requirement. To the best of the authors' knowledge this is the first reported approach to achieve this capability. Specifically, by application of a prototype compiler to accelerators for Full Search Motion Estimation (FSME), Matrix Multiplication (MM), Fast Fourier Transform (FFT) and Sobel Edge Detection (SED), the following contributions are made:

i) A novel statement-oriented approach to deriving dataflow equivalents of C functions is presented.
ii) A novel synthesis approach for FPGA dataflow accelerators, including multi-level and off-chip memory, is presented.
iii) It is shown that automatic realisation of accelerators, including all memory resource, which meet user-defined performance requirements is enabled.
iv) Novel optimising transformations are presented which reduce memory capacity and FPGA resource cost by up to to 96% and 75% respectively.

Sections IV - V describe the proposed C-to-dataflow and dataflow-to-FPGA approaches and their application to FSME, MM, SED and FFT before Section VI describes dataflow transformation techniques for FPGA architecture optimisation and illustrates the cost-reduction capabilities of these techniques.

II. Background

Modern FPGA boast an abundance of diverse memory resources including off-chip DRAM, and on-chip BRAM and DisRAM realised using Look-Up Tables (LUTs). Each distinct memory resource offers a different capacity/rate combination as illustrated in Fig. 1 for Xilinx Virtex®-7 FPGA. For accelerators which demand access to large data objects at high data rates, this structure poses a difficult design problem. In order to offer sufficient capacity, off-chip DRAM is required, but high access rates necessitate on-chip BRAM and/or DisRAM. Supporting both large capacity and high access rates demands multi-level memory structures including both off-chip DRAM and multiple levels of on-chip BRAM/DisRAM, customised to the accelerator's behaviour and real-time performance.

Creating such structures with high performance and low cost has traditionally required manual RTL design in a HDL, a much more detailed design process than programming of multicore processors or GPUs in a language such as OpenCL. HLS has been proposed to enable accelerators to be synthesized from such languages, but HLS of

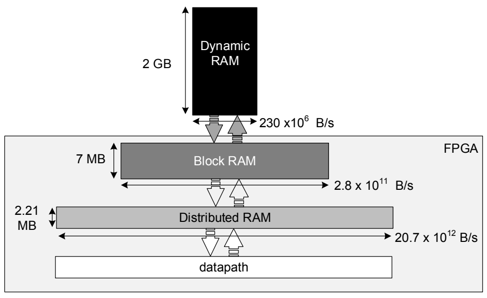


Fig. 1: Xilinx Virtex®-7 FPGA Memory Hierachy

complex memory structures is a known open problem [4]. Ideally, to support memory-intensive accelerators, HLS should enable a series of key autonomous capabilities:

1) Synthesize FPGA accelerators, including memory infrastructure, from naive-C kernel descriptions
2) Map application buffers to on-chip or off-chip RAM
3) Derive multi-level on-chip memory structures using both BRAM and DisRAM
4) Ensure that architecture derived satisfies user-defined performance requirements
5) Fulfill 1 - 4 automatically, without manual intervention.

Existing HLS approaches address some of these requirements, but not all. The MATLAB HLS tool in [8] relies on manual designer direction to realise BRAM-based variable storage, but cannot make such decisions autonomously nor target off-chip DRAM. Autopilot [4], Vivado HLS [9] and Synphony C offer HLS for C/C++ but suffer similar limitations; Vivado HLS can realise monolithic RAM components via manual user direction, but cannot do so autonomously nor target off-chip DRAM or form multi-level hierarchies. OpenCL HLS environments, such as Altera's OpenCL SDK and Xilinx's SDAccel [10] which accompany MPSoC-FPGA impose mandatory off-chip DRAM interface for a specific class of memories and leave realisations and targetting of all other memory resources to the designer to resolve manually - indeed, by using OpenCL as a specification model, manual memory handling is explicitly required [11].

Source-to-source compilers transform input programs for such tools to resolve some of these issues. The work in [12], [13], [14] exploits Polyhedral Intermediate Representations (IRs) [15] to realise data reuse buffers, thereby reducing external memory access rates. However, only a single level of reuse buffering is realised, neither multi-level nor off-chip RAM targetted and manual intervention is required in order to meet a real-time performance target.

A series of excellent HLS approaches in the literature also offer different capabilities. Compaan/LAURA [16] translates Static Affine Nested Loop Programs (SANLPs) in C or MATLAB to FPGA architectures realising Kahn Process Networks (KPNs) [17] derived via a Polyhedral IR, a theme which is maintained in ESPAM [16], [18]. These approaches can derive a wide range of architectures spanning networks of accelerators to multiprocessor architectures on FPGA. With respect to the memory synthesis problem ESPAM has even demonstrated the capability to target off-chip memory buffers [19]. However, to the best of the authors' knowledge, none of these approaches has addressed construction of custom hierarchies or automatic tuning of memory architecture to user-defined performance requirements. SystemCoDesigner [20] exploits the *FunState* [21] model of computation translating these to an FPGA architecture consisting of computing nodes communicating via FIFO queues. The FIFOs can take the form of BRAM or DisRAM but SystemCoDesigner does not explicitly address in which form these FIFOs should be realised nor off-chip or multi-level memory structures. Beyond this work, synthesis of memory structures for custom accelerators has an extremely long history with a large number of authors considering many aspects of process, including synthesis of IRs from C-like programs [22], [23], [16], [18] and synthesizing custom accelerators from IRs [24], [25], [26].

These works all offer promising features, such as polyhedral source-to-source compilers for automatic derivation of data reuse buffers [12], [13], [14] or to support manual mapping of KPN FIFOs to off-chip RAM [27] but, to the best of the authors' knowledge, no approach to date has demonstrated the capability to automatically allocate hybrid on/off-chip memory structures, determine the physcial nature of each on-chip buffer (i.e. BRAM or DisRAM) and bind application buffers thereon in order to meet a user-defined performance constraint. Indeed, the absence of any basic capability to create such architectures implies the need for constructive approaches, which can derive any kind of viable realisation, as a necessary foundation technology for HLS including sophisticated design space exploration and multi-objective optimisation.

This paper describes such an approach. Specifically, a constructive approach to deriving an FPGA RTL accelerator architecture with a prescribed real-time performance and including on-chip and off-chip memory, from a C specification is described in Sections III - VI.

III. Constructive Kernel Synthesis Approach

The proposed constructive synthesis process is illustrated in Fig. 2. The goal is to convert a C kernel to an RTL accelerator architecture such that a given real-time performance requirement, expressed as a number of iterations N of the kernel per second, is satisfied. The key design issue of concern in this work is realisation of memory resource for such accelerators. Initially (① in Fig. 2) an IR of the C kernel is derived and analysed in order to gauge the feasibility of satisfying N. In the case where N will not be satisfied, optimising transformations are applied to the IR via source-to-source transformations on the C kernel (③). In the case where the result does satisfy N, IR transformations are applied to reduce the cost of the final architecture, which is subsequently translated to RTL source (④). For the constructive process proposed, the synthesis and optimisation algorithms employed in ③ and ④ should avoid the use of complex, iterative design space exploration heuristics [7].

The form of the C input and the choice of IR are important considerations in developing such a process, with the former defining the breadth of functionality which may be expressed and the latter defining the capabilities of the synthesis process and the scope of the optimising transformations.

A number of works have shown how SANLPs are sufficiently powerful to express a broad class of signal, image and data processing operations whilst enabling sophisticated compiler analysis for accelerator optimisation for FPGA. Hence, in common with [16], [18], [13], [28], [29], [13], [14], [27], we propose the use of SANLP C. A SANLP representation of a matrix multiplication forming $C = A \times B$ where $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$ and $C \in \mathbb{R}^{m \times p}$ is shown in Pseudo-code 1.

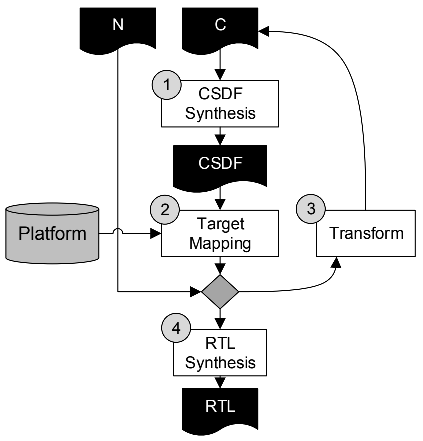


Fig. 2: C-CSDF-FPGA Synthesis

```
1: procedure MULTIPLYMATRICES($A, B$)
2:     for $i \leftarrow 1 : m$ do
3:         for $j \leftarrow 1 : p$ do
4:             $C[i][j] \leftarrow 0$
5:             for $k \leftarrow 1 : n$ do
6:                 $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
7:     return $C$
```

Pseudo-code 1: Matrix Multiplication

In order to determine the form of an effective IR, consider the purpose of the loop statements in Pseudo-code 1. The outermost loop successively isolates rows of A, the first level loop (iterator j) columns of B and the innermost loop scalar elements of these row/column subsets. The aggregate behaviour is a natural refinement of subsets of the kernel operands (A and B), to the scalar operands/results for the assignment statement in line 6. As such, with respect to data refinement and memory management, loops contain important information and should be considered operations with input operands and results, acting only to refine and schedule data. By hierarchical composition of such statements, the information necessary to realise that succession of refinements in a multi-level memory hierarchy is naturally represented. The chosen IR should maintain this information for synthesis. However, this information is not maintained in current IRs, which either pre-suppose a machine (and hence memory) architecture or which consider loop statements as representative of only the dependencies between invocations of their child statements.

Cyclo-static Dataflow (CSDF) models have been shown to be a suitable IR for SANLP programs [27]. A CSDF model is a graph $G = \{V, E\}$ where V is a set of *nodes* and E a set of *edges*; edges represent First-In First-Out (FIFO) queues of data *tokens* which connect ports on nodes. This syntax is summarised for a simple CSDF graph in Fig. 3.

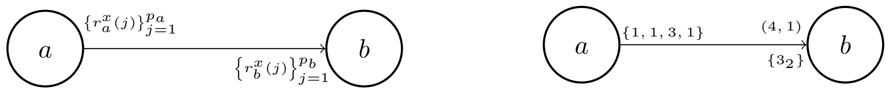


Fig. 3: CSDF Notation

Every node $v_i \in V$ executes a sequence of phases $\{f_i(j)\}_{j=1}^{p_i}$, such that the n^{th} invocation, or *firing*, of v_i performs the function of phase $f_i\left((n-1) \mod p_i + 1\right)$. On each firing, a node consumes (produces) a pre-defined number of tokens, known as that port's *rate*, via input (output) ports from the connecting FIFO edges. The rate of a port x takes the form $\{r_a^x(j)\}_{j=1}^{p_a}$ - i.e. the n^{th} firing of a consumes/produces $r_a^x\left((n-1) \mod p_a + 1\right)$ tokens via port x where each $r_a^x(j) \in \mathbb{Z}^+$. Rates are denoted adjacent to each port whilst token dimensions are denoted in circular braces[1].

Fig. 3b shows a two-actor network connected via a single edge. Actor a follows a sequence $\{f_1, f_2, f_3, f_4\}$, where on firing i the functionality of $f_{i \mod 4}$ is performed and the corresponding number of tokens produced through its single output port - so on the first firing 1 is produced, on the second 3, on the third 2 and on the fourth 4, before the sequence repeats. Similarly b follows a repeating consumption sequence of 3 on its first firing and 3 on its second, before repeating - for brevity we use the notation 3_2 to mean '3 repeated 2 times'. In this case, tokens are vectors of dimension $(4, 1)$.

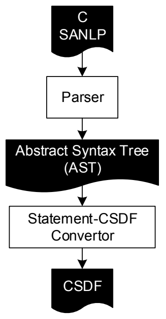
In CSDF, actors can manipulate both the form and value of data tokens as they flow along edges. If the hierarchical composition of loops, so important in refining and selecting data operands in the C kernel, could be captured using the CSDF token and rate semantics in a hierarchical CSDF IR, then it would be highly effective for derivation of accelerators including memory infrastructure. The proposed approach to converting a C SANLP to a CSDF IR is detailed in Section IV before Sections V and VI describe constructive CSDF accelerator synthesis and optimisation approaches.

IV. Deriving CSDF Models from C Kernels

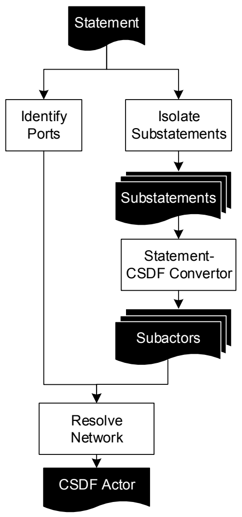
The CSDF IR should maintain the hierarchical composition of C kernel statements, in particular loop nests to support derivation of the multi-level accelerator memory architecture. This hierarchy is naturally maintained by Abstract Syntax Tree (AST) and hence it is proposed that CSDF IRs be derived via AST as illustrated in Fig. 4.

As shown, the AST is converted to a CSDF model using a Statement-CSDF convertor phase. The behaviour of this convertor is illustrated in the flowchart in Fig. 4b. An SANLP is interpreted as a single statement composed of a sequence of substatements $S = \{s_1, s_2, \cdots s_N\}$ of different kinds. This will be converted to a composite hierarchical CSDF actor (i.e. a network of interconnected sub-actors) with a single input port for each SANLP argument and an output for each result. The SANLP substatements are isolated and are themselves converted to

[1]Rates are omitted when $\{r_a^x(j)\}_{j=1}^{p_a} = \{1\}$, braces are omitted for single-phase sequences and token dimensions are omitted in the case of scalar tokens



(a) C Kernel Synthesis Process



(b) AST-CSDF Synthesis

Fig. 4: SANLP Synthesis Process

CSDF actors by recursively invoking the same statement-CSDF procedure. The resulting set of CSDF actors are then connected to one another and the set of ports identified to perform the function of the SANLP. Three classes of statements are recognised: *block*, *loop* and *conditional* statements.

A. Block and Assignment Statements

Block statements are imperative sequences of variable declarations and sub-statements; the AST itself is considered a block. In general substatements may be block, loop or conditional statements. The scope of a block statement extends between opening ({) and closing (}) delimiters or between successive loop/conditional statements otherwise. Block statements place no restrictions on the scope of variables defined therein, rather relying on the containing kernel, loop or conditional statements to manage variable scope. An illustrative block statement is shown in Pseudo-code 2.

```
1: procedure BLOCKSTATEMENT($a, b, e, f$)
2:     $c \leftarrow a + b$;
3:     $d \leftarrow c * e$;
4:     $c \leftarrow e * f$;
5:     return $c, d$
```

Pseudo-code 2: Block Statement

Pseudo-code 2 is composed entirely of assignment statements, which represent the leaves of the hierarchical kernel structure and are realised using primitive (i.e. non-hierarchical) CSDF actors. An assignment operation combines

k operands into a single result and hence the realising primitive actor hosts a set of $k+1$ ports $P = \{p_i\}_{i=1}^{k+1}$ - k inputs (one per operand) and a single output. The rate of each port x, $\{r^x(j)\} = \{1\}$.

Block statements are realised by composite actors with a single sub-actor for every sub-statement. In this case there are three sub-actors, one per assignment sub-statement. When synthesizing the composite, two issues must be addressed.

1) Arbitrating sub-actor access to operand/result variables.
2) Ensuring sub-statement dependencies are met.

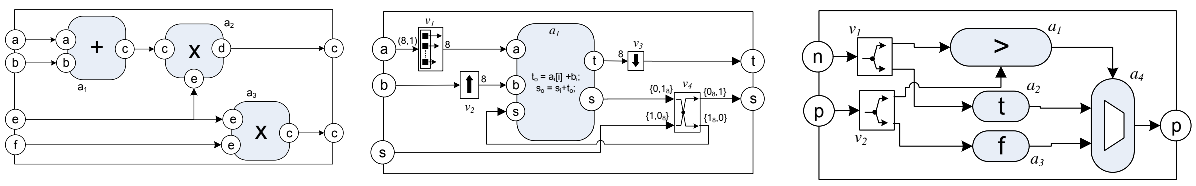
Block statements are realised by a composite CSDF actor with a single input (output) port for each variable read (written) as an block operand (result); the rate of each port x, $\{r^x(j)\} = \{1\}$. A single sub-actor realises each sub-statement with dependencies maintained by FIFO queues. According to this approach, the block statement in Pseudo-code 2 is realised using the dataflow node in Fig. 5a; note that this node is a Synchronous Dataflow (SDF) [30] specialisation of the generalised CSDF modelling approach.

B. Loop Statements

A loop statement is one which invokes a series of sub-statements numerous times, with the scope of the statement extending between the opening and closing delimiters for the statement ({ and }); all variables defined within a loop have local scope within the statement. On each iteration a subset of the variables read are selected as operands and a subset of the variables written selected as results; in a SANLP these subsets are defined by affine transformations of the loop variables. A sample loop statement is shown in Pseudo-code 3, with the equivalent CSDF actor shown in Fig. 5b.

```
1: procedure LoopStatement($\mathbf{m}[8], b, s, t$)
2:     $s \leftarrow 0$
3:     for $i \leftarrow 1 : 8$ do
4:         $t \leftarrow m[i] + b$
5:         $s \leftarrow s + t$;
6:     return $s$
```

Pseudo-code 3: Loop Statement



(a) Block Statement (b) Loop Statement (c) Conditional Statement

Fig. 5: CSDF Equivalents of C Statements

When converting loop statements to CSDF actors, three particular issues are of concern.

- Invoking the loop-body behaviour multiple times
- Managing loop-iterator dependent memory accesses
- Maintaining loop-carried dependencies

An input (output) port is instantiated for each variable read (written) as an operand (result) of the loop body, with the rate of each resulting port x, $r_x = 1$; hence, since all of $\mathbf{a}$, b, s are operands for operations within the loop of lines 3 - 5 in Pseudo-code 3, each of these results in an input port, with the variables used as results, s and t, resulting in output ports. The loop body (lines 4 and 5) is realised as a block sub-actor a_1 and invoked multiple times (8 in the case of Pseudo-code 3) by a series of peripheral actors inserted to ensure that the requisite data is provided for each invocation.

In an SANLP, the following may be determined statically, at compile time

- The number of loop iterations
- The set of sub-elements of array operands accessed in each iteration of the loop body

To enforce multiple invocations of the loop-body actor, input data are managed in one of two ways. Non-loop-indexed operands, such as b in Pseudo-code 3, are replicated via an *upsample* actor, v_2 in Fig. 5b which creates a copy of the token consumed for each invocation of the loop body; in the case of Fig. 5b, 8 copies are produced. For loop-indexed operands, such as $\mathbf{m}$, selection actors such as v_1 in Fig. 5b are used to produce a sequence of subsets of the input token, with the length of the sequence and the subsets inferred from the affine function of the loop iterators which address those operands. For results, the opposite behaviours occur - v_3, a *downsample* actor, consumes each of the values of t written by a_1, producing only the final value, whilst loop-indexed results, absent from Pseudo-code 3 are handled by a selective write actor which composes the scalar results into a vector.

Loop-carried dependencies, such as s in Pseudo-code 3 are managed using a cyclo-static *register* actor, v_4 in Fig. 5b. This actor has $n + 1$ phases, where n is the number of loop iterations. During the first phase an initial value is consumed from the appropriate input port. This is produced for consumption by the loop body statement, before being recycled back to the same statement n times. The final value is produced from the loop actor. Hence, this actor has two input ports and two outputs. The initialisation port (the lower port on the left hand side of v_4) consumes one token during the first phase and does not consume any subsequent tokens until the next firing of the loop actor. The intermediate values of s are produced by the lower right-hand side port of v_4, with intermediate values produced for each invocation of the body actor a_1 followed by a single firing with no tokens produced. During the first firing of v_4, the only token consumed is the initialise loop input token and hence no intermediate results are consumed from a_1 for the first firing of v_4, with a single intermediate value consumed during each of the final n firings. Finally, the only result written by the loop actor is the final intermediate result consumed and hence the final output port produces no tokens during the first n firings, with 1 produced during the final firing. Note that, whilst the loop statement actor is internally cyclo-static, it is externally SDF.

C. Conditional Statements

A conditional statement, a sample of which is shown in Pseudo-code 4, performs one of a set of sub-operations, depending on the result of some test condition. The statement scope is between opening ({) and closing (}) delimiters and variables declared within have local scope. A conditional statement is realised using a network of four actors:

1) An actor to evaluate the condition
2) An actor each to evaluate the true and false cases
3) An actor to select the appropriate result.

```
procedure CONDITIONALSTATEMENT($n, p$)
    if $n > p$ then
        $p = n$
    else
        $p = p$
    return $p$
```

Pseudo-code 4: Conditional Statement

For the conditional listing in Pseudo-code 4, the synthesized actor is shown in Fig. 5c. Unity-rate ports are synthesized for each operand and result; the actors v_1 and v_2 perform broadcasts, which duplicate the input token on each output port. In this case the condition specified is realised by actor a_1, which perform a 'greater than' comparison. Actors a_2 and a_3 realise the true and false condition statements, with a_4 selecting the appropriate result as dictated by the result of a_1. Note that, despite realising a data-dependent operation, this conditional actor is an SDF actor.

D. Network Synthesis

After synthesising CSDF models of each statement, FIFO queues are inserted to ensure sub-statement dependencies are maintained. To illustrate the result of this approach, Fig. 6 illustrates the CSDF IR derived form a C kernel which represents the memory accesses for FSME on 352×288 CIF frames, as described in Pseudo-code 5.

As shown, the hierarchical statement structure in the deep loop nest in lines 2 - 9 results in a deep hierarchical structure composed of a single actor with 6 levels of hierarchy. As Fig. 6 shows, this results in a sequence of six data formatting actors between the SANLP input and the assignment actor, the rightmost in Fig. 6. As shown, $v_{c,i}$ - $v_{c,n}$ successively refine C, of dimensions $(288, 352)$ by selecting 18 subsets of $(48, 384)$ ($v_{c,i}$), each of which are in turn decomposed into 22 subsets of $(16, 16)$ by $v_{c,j}$. Subsequently, each $(16, 16)$ frame is replicated 32 times by each of $v_{c,k}$ and $v_{c,l}$, before the $(1, 16)$ row vectors and their individual elements are extracted by $v_{c,m}$ and $v_{c,n}$ respectively. Similar behaviours are realised by the valve sequence transferring R.

Consider how this structure may be converted to an FPGA circuit. The assignment subactors a_n in Fig. 5a - Fig. 5c all operate solely on scalar tokens; the memory requirements for these actors is likely very small and may be

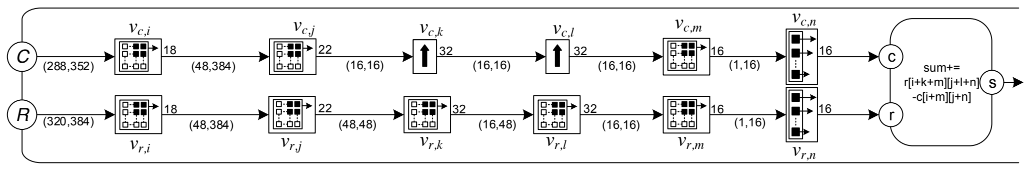


Fig. 6: Full Search Motion Estimation CSDF Graph

```
1:  procedure FSME($R, C$)
2:    for $i \leftarrow 0 : 17$ do
3:      for $j \leftarrow 0 : 21$ do
4:        for $k \leftarrow 0 : 31$ do
5:          for $l \leftarrow 0 : 31$ do
6:            $s \leftarrow 0$
7:            for $m \leftarrow 0 : 15$ do
8:              for $n \leftarrow 0 : 15$ do
9:                $s \leftarrow s + R[16i + k + m][16j + l + n] - C[i + m][j + n]$
10:   return $s$
```

Pseudo-code 5: Full Search Motion Estimation

realised using registers. However, the peripheral actors v_n in Fig. 5a - Fig. 5c operate on higher order data tokens, manipulating their form and rate but performing no computation on their values. Since they process multi-element tokens, memory costs are likely to be incurred and any memory-aware synthesis approach should focus on these actors, henceforth known as *valves*. Since these only reorder token elements they may be realised using controlled memories and hosted either on-chip or off-chip, allowing a synthesis tool to automatically determine which is required. Section V describes an approach for this purpose.

V. Constructive CSDF Synthesis on FPGA

In order to realise the CSDF IR as an FPGA accelerator, three key issues must be addressed:

1) Realising both the memory and computational aspects of the CSDF behaviour
2) Partitioning valves between on-chip and off-chip memory resources
3) Deriving multi-level on-chip memory structures
4) Ensuring the accelerator meets the user-defined performance requirement

A. FPGA Elements

FPGA accelerators have identical topology to the CSDF they realise, with each actor realised by a Processing Element (PE) and each valve by a Memory Element (ME) in a Globally Asynchronous Locally Synchronous (GALS) network via two-way handshaking connections enabling data-driven accelerator execution. Node-level control and global asynchrony are exploited intentionally to avoid the need for complex global synchronous control, which may impose a performance limiting effect on large scale accelerator structures. The structure of a PE is shown in Fig. 7.

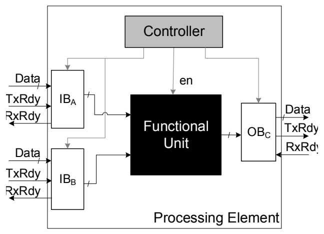


Fig. 7: FPGA Processing Element (PE)

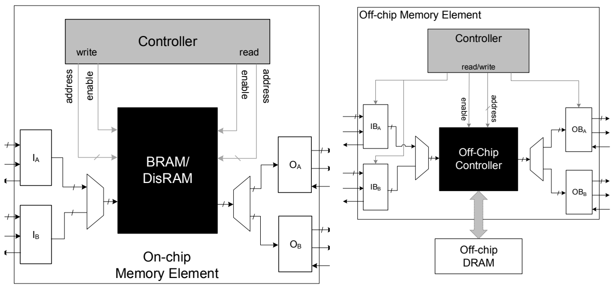
A PE is a parameterized RTL object composed of three main types of component.

- Buffers: Input Buffers (IBs) and Output Buffers (OBs).
- FU: the core functional unit
- Controller: sequences data from IBs to OBs via FU.

Buffers are attached to every port in order to satisfy the dataflow semantics of a PE: each buffer has sufficient capacity to store a single data frame and is parameterized such that its capacity is configured at compile-time given knowledge of the token type, rate and data type of the port to which it is attached. The FU is selected from a library of primitive assignment operators - these may hypothetically be pre-designed components in the case of complex sub-functions, but in all kernels analysed in paper the FU is realised using behavioural VHDL statements corresponding to the native assignment operators exploited in the C kernel. The controller enforces the dataflow consume/process/produce schedule.

Valves are realised using MEs in one of two forms depending upon whether on-chip or off-chip memory is targetted; both forms are illustrated in Fig. 8. In the case where on-chip storage is to be realised the per-port buffering of the PE is replaced by a large centralised buffer with sufficient capacity to house a single frame (i.e. the set of input tokens consumed during a firing of the valve) per input port. As shown in Fig. 8a, the on-chip ME memory component is realised using a dual-port RAM in the form of either BRAM or DisRAM. In this work, where the required capacity is less than 64 words, LUT-based DisRAM is employed, otherwise BRAM is used; it should be noted that this threshold is user-configurable so that the memory structure of the final architecture can be turned to the resources available on the target device. In the case where the valve is to be realised off-chip, the per-port buffering of the PE is retained, with the centralized buffer space evident in the on-chip ME replaced by an interface which arbitrates access to off-chip memory by source/sink PEs/MEs.

In both cases, the controller again enforces the sequential consume/produce dataflow schedule. Specifically for each input port x, the controller performs a j-phase bijective consumption function $c^x(j) : F^x \rightarrow \mathbb{Z}^+$, where $F^x = [1, r^x(j)] \times [1, m] \times [1, n]$ i.e. the triple which indexes the elements of F^x, the frame consumed at x is translated to an integer denoting the internal buffer location of that element[2]. For each output port y the elements of F^y are formed reading corresponding buffer elements via an injective produce function $p^y(j) : \mathbb{Z}^+ \rightarrow F^y$ where $F^y = [1, r^y(j)] \times [1, m] \times [1, n]$,



(a) On-Chip

(b) Off-Chip

Fig. 8: FPGA Memory Elements

B. Memory Partitioning and Synthesis

Since valves do not perform any arithmetic operations on token elements as they pass through, datapath circuitry is not required and these may be realised using managed memories, which may be hosted either on-chip or off-chip. It is required to determine the ultimate destination of each CSDF valve.

Note the general form of the CSDF ME kernel in Fig. 6 - a sequence of valves serving data to an actor which performs the appropriate computation. The memory synthesis problem, then, is this: to determine from such a sequence V, the subsequence V_{on} to be hosted on-chip and the subsequence V_{off} to be realised off-chip; note that $V_{on} \subseteq V$, $V_{off} \subseteq V$ where $V_{on} \cup V_{off} = V$ and $V_{on} \cap V_{off} = \emptyset$. The process of deriving V_{on} and V_{off} is shown in Algorithm 1.

Algorithm 1 Valve Chain Partitioning

```
1: procedure PartitionValves($V, r_{off}, N$)
2:     $V_{off} \leftarrow V, V_{on} \leftarrow \emptyset, i, j \leftarrow |V|, r \leftarrow 0$
3:     while $i > 0$ do
4:         $r = rate(v_i, N)$
5:         if $r > r_{off}$ then $j \leftarrow i$
6:         else
7:         $i \leftarrow i - 1$
8:     $V_{off} = \{v_k\}_{k=0}^{j-1}$, $V_{on} = \{v_k\}_{k=j}^{|V|}$
9:     return $V_{off}, V_{on}$
```

[2]For matrix tokens of order (m, n)

The procedure accepts as inputs V, r_{off}, the peak off-chip data rate and N, the desired number of iterations per second. Initially (line 2) the entire valve sequence is contained within V_{off} - i.e. hosted off-chip; this starting point is chosen so that MEs are hosted on-chip only when minimum data rates demand that be the case, hence minimising the amount of on-chip buffering. Given that the PE served by this sequence will likely demand a higher input data rate than that available using off-chip DRAM, the purpose is to determine the interface point between V_{off} and V_{on}.

To do so, each element $v_i \in V_{off}$ is analyzed, commencing with the final element. The total rate of production (r, in Bytes/s) of v_i is derived (line 4) by scaling the aggregate output rate by N. If $r > r_{off}$, v_i remains in V_{off}, otherwise it is marked as the interface point between V_{off} and V_{on}. After repeating for each element of V_{off}, the final value of j is used to define $V_{off} = \{v_k\}_{k=1}^{j-1}$, $V_{on} = \{v_k\}_{k=j}^{|V|}$ (line 8).

Note the purpose of this approach. Initially, all buffers are hosted off-chip and if off-chip memory access rates were sufficiently high as to support the required rate by a PE, no on-chip cost would result beyond that of the off-chip ME. As the achievable off-chip data rate decreases, greater numbers of the inner loops will be hosted on-chip but only those buffers where on-chip realisation is demanded by the required memory access data rates. Hence, this approach seeks to determine the minimum set of valves which must be hosted on-chip, thereby incurring the minimum FPGA resource cost whilst supporting real-time performance.

C. CSDF Analysis & Transformation

The one-to-one correspondence between CSDF topology and accelerator architecture and the use of pre-designed PE and ME components permits pre-characterisation of the performance and cost of these components in support of CSDF transformation to ensure that the prescribed performance requirements are met.

By default, the CSDF IR derived from the C SANLP instantiates a single actor for each substatement, with that single actor shared across all statement invocations. This approach will enable compact architectures but it is likely will have insufficient throughput to support the required number of kernel iterations N. Increased throughput may be enabled by increasing parallelism. Loop unrolling is a well-known and well-studied technique for increasing performance of accelerators synthesized from C SANLPs [28], [8], [13], [31]; in the interests of brevity we avoid discussing this here, but rather refer the reader to these works. In the context of this work, the primary challenge is determining the level of parallelism which needs to be induced in each loop in order to meet the desired real-time performance.

A n-level loop nest in the C specification, when synthesized to CSDF will result in sequences of valves $V = \{v_1, \cdots, v_n\}$. An unrolling procedure is proposed which determines the number of instantiations of each ME required in order to satisfy the real-time performance targets via loop unrolling, as described in Algorithm 2.

The unrolling procedure iteratively analyses each valve in the sequence V (lines 4 - 16 in Algorithm 2) with a view to ensuring that the requisite output data rate is achieved when synthesized. In the case where the anticipated output data rate after synthesis is lower than that demanded by the application, then the loop corresponding to the CSDF valve in question is unrolled, with the unroll factor determined by Algorithm 2.

Algorithm 2 Loop Unrolling via Valve Analysis

```
1: procedure UNROLL($V, N$)
2:     $r \leftarrow a \bmod b$
3:     $\mathbf{r} \leftarrow \mathbf{q}, u \leftarrow \mathbf{1}^{|V|}, i \leftarrow |V|$
4:     while $i > 0$ do
5:         $r_{et} = rate\,(v_i)$
6:         $r_d = N \cdot q_i \cdot m \cdot n \cdot r_{out}$
7:         $u_i \leftarrow \lceil \frac{r_d}{r_e} \rceil$
8:         if $u_i < r_e$ then
9:             $P = \left\{ p : p \in \mathbb{Z}^+ \text{ and } \frac{u_i}{r_o} \in \mathbb{Z}^+ \right\}$
10:            $u_i = \min \{ j : j \in P \text{ and } j > u_i \}$
11:        else
12:            $u_i = r_o$
13:        for $j \leftarrow i...|V|$ do
14:            $u_j \leftarrow \lceil \frac{u_j}{U} \rceil$
15:        $V = unroll(V, \mathbf{u})$
16:        $i \leftarrow i - 1$
```

Clearly, anticipating the data rate which may be achieved by a FPGA ME realising a CSDF valve requires an estimate of the clock rate of the final architecture. Accurate pre-synthesis estimation of FPGA clock rate is an open research challenge but the use of pre-designed PE and ME components allows ad-hoc estimation to be performed. Specifically, it is assumed that each PE and ME port can transfer only a single data element per clock cycle, regardless of the nature or rate of tokens consumed/produced by the equivalent CSDF actor; in the case where non-scalar tokens are transferred at a port, the consumption/production occurs over multiple clock cycles and, by convention, in row major order. Given pre-synthesis characterisation of the synthesize clock rate of each component and taking a conservative estimate of the composite clock rate (75% of lowest clock rate of all components) an estimate of the synthesized clock c can be derived.

The procedure in Algorithm 2 determines an unroll factor for each level in the loop nest $\mathbf{u}$ as follows. Each valve in $v_i \in V$ is considered in turn, $i = |V|$. The estimated outgoing data rate r_e, measured in words per second, is taken to be c, the anticipated clock rate (line 5), whilst the desired rate r_d is taken as the total number of words produced in an iteration of the graph schedule, scaled by the user-defined iteration rate N. The former term is determined by the product of q_i, the number of firings of v_i in an iteration of the schedule, m and n, the dimensions of the token produced[3] and r_{out}, the rate of the valve output. The required unroll factor u_i in order to satisfy r_d is then given by the ratio of r_d and r_e (line 7). When $u_i > r_{out}$, r_{out} is adopted as the unroll factor (line 12); otherwise

[3]assuming (m, n) matrix tokens - extension to other types is trivial.

it is taken to be the smallest integer factor of r_{out} which exceeds u_i (lines 9 and 10).

Where an unroll factor other than 1 results, the values of u for all subsequent valves are scaled downwards accordingly to ensure that the specific number of loop instances specified by u_i result. As a result of each stage of unrolling, the number and form of valves may change, influencing the results of subsequent unrolling calculations. Hence the graph is unrolled according to $\mathbf{u}$ before the next iteration commences.

Consider the effect of this process on the sequence of valves $v_{r,k} - v_{r,n}$ in Fig. 6. The estimated maximum output data rate from each of these, and the required rate for operation at 25 FPS for H.264, assuming 3-byte (r,g,b) pixels are described in Table I. Each of $v_{r,l} - v_{r,n}$ require unroll factors of 16 in order to support output data rates of 7.26 GBytes/s, as illustrated in Fig. 9 where 16 such valves are included. Whilst according to the initial configuration of the CSDF graph $v_{r,k}$ appears to require unrolling of the corresponding loop by a factor of 2 to enable the required 682 MB/s, the intermediate unrolling steps for $v_{r,l} - v_{r,n}$ result in a vastly increased data rate of 7.26 GBytes/s to service the input demand of all 16 instances of $v_{r,l}$. Accordingly, this loop too is unrolled by a factor 16.

edge	$v_{r,k}$	$v_{r,l}$	$v_{r,m}$	$v_{r,n}$
estimated rate (MB/s)	510	510	510	510
H.264 (MB/s)	682	7260	7260	7260

TABLE I: H.264 FSME Data Rates and Constraints

Note the general effect of this unrolling transformation - single actors in the original DFG have been replaced by multiple actors in the RTL architecture. In addition, in many cases the large tokens processed and stored by the original actors be decomposed into multiple smaller sub-tokens.

D. Experiments

To illustrate the capability of the proposed synthesis process, four memory-intensive processing kernels are addressed.

1) A 1024 point FFT on 16 bit fixed-point data targetting maximum possible throughput.
2) 25 FPS FSME on H.264 Level 1.3 CIF video employing 3-byte (r,g,b) pixels.
3) SED at 30 FPS on H.264 Level 3.1 720p frames employing 3-byte (r,g,b) pixels.
4) 128×128 16 bit fixed-point MM at 500 operations/s.

All cases target Alpha-Data ADM-XRC-5T1 hosting a Xilinx Virtex®-5 SX95T FPGA and 2 GB, 230 MB/s off-chip DRAM. The internal arithmetic employed by all accelerators is identical to that of the input data. According to the approach described in Section V-B the resulting partitioning of the two main valve sequences supplying operands to the SAD actor are mapped to off-chip and on-chip memory as illustrated in Fig. 9, along with the nature of each on-chip valve (B for BRAM, L for DisRAM).

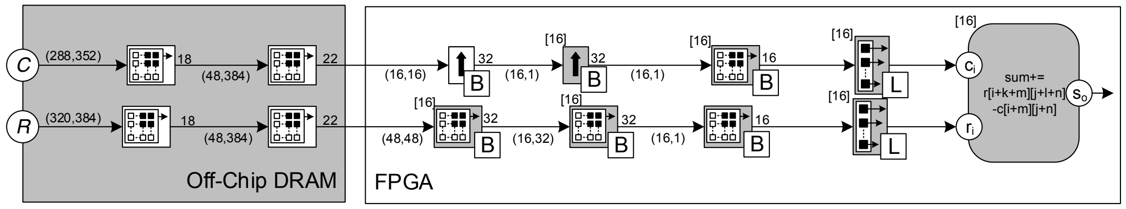


Fig. 9: FSME RTL Architecture

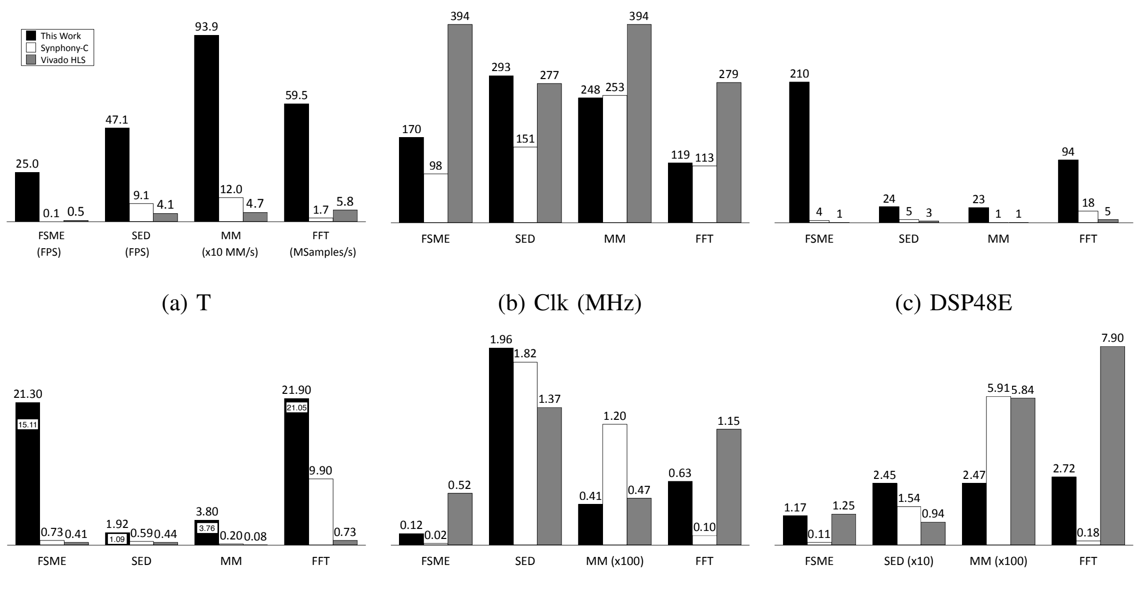
The performance and resource cost of this FSME architecture, as well as that derived via SynphonyC and Vivado HLS[4], is recorded in Fig. 10[5]. Note that, since the circuits derived by the commercial tools do not derive buffering for their input or output data and hence this cost in unreported for these but, rather, for benchmarking of the results produced.

Despite this, some obvious trends are apparent. The main trend which may be observed, in particular in Fig. 10a verifies the central novel claim of our work: all accelerators achieve the stated throughput targets and have been automatically derived to do so from a C kernel specification whilst, uniquely, actually deriving and including the requisite memory infrastructure. This is in contrast to the results produced by Vivado HLS and SynphonyC, which are much smaller but offer much lower performance; this is as a result of their inability or autonomously parallelise their input programs to meet user-defined performance requirements and their omission of the memory infrastructure required for their use.

It is also noticeable that the performance and scale of circuits produced by our tool are both, in general, much larger than those achieved by Synphony-C and Vivado HLS; this is a direct result of the automated loop unrolling parallelisation approach described in Section V-C - neither CatapultC nor Synphony-C are able to detect the need for such parallelisation nor apply loop unrolling autonomously, leading to the relatively low performance results. The increase in throughput over Synphony-C range between factors of 5.2 (SED) to up to two orders of magnitude (FSME) and factors of 10.3 (FFT) to 50 (FSME) when compared with Vivado HLS. These very substantial performance differentials and architecture make direct comparison of performance and cost difficult, but some comparative analysis is revealing. It is apparent that the DSP48E cost of the CSDF accelerators, relative to throughput, are highly competitive with those derived using Synphony-C and Vivado HLS. Fig. 10e shows that this work produces architectures which support higher throughput per DSP48E than Synphony-C in three of the four applications, despite the clock rate restriction experienced as a result of the generally increased scale of the accelerators. Synphony-C achieves superior DSP48E efficiency for MM due to its ability to combine these with LUTs to realise the core multiply-add operation, whilst at present all arithmetic components used in the CSDF approach exploit DSP48E-based arithmetic, leading to artificially high DSP48E cost. In addition Vivado HLS produces architectures with disproportionately higher performance per DSP48E in three applications, Fig. 10b

[4]Employing streaming interfaces

[5]Note that, for consistency, no manual manipulation or annotation of input source, nor manual direction, is given to any of the tools compared in this section



(a) T (b) Clk (MHz) (c) DSP48E

(d) LUTs ($\times 10^3$) (e) T/DSP (f) T/LUT

Fig. 10: Synthesis Results & Comparison

shows that these metrics are enabled by drammatically superior clock rates, underpinned by the much smaller circuit sizes.

Similarly, the accelerators derived via this work offer superior performance per LUT than Synphony-C in three of the four applications and one when compared to Vivado HLS. However, in Fig. 10d, the number of LUTs exploited as logic by this work are recorded within the bars. If LUT cost is considered to include only logic, a fairer comparison since neither Vivado HLS nor SynphonyC exploit LUTs as DisRAMs, CSDF T/LUT for FSME increases to 1.65, eclipsing that of Vivado HLS. This disproportionately lower LUT cost is despite the extra expense incurred for buffer control, omitted from the circuits produced by Synphony-C and Vivado HLS.

Neither Synphony-C nor Vivado-HLS address memory synthesis and hence incur no BRAM cost, making comparison of this metric impossible. However, as a point of comparison the 1024 point DFT accelerator produced by Spiral [32] demands 34 BRAMs on the same device.

In summary, the CSDF HLS approach has enabled real-time accelerators for FSME, SED, MM and FFT to be derived automatically from C, including all memory infrastructure. Further, despite deriving accelerators of much greater performance than SynphonyC and Vivado HLS and incurring extra cost for memory control, the efficiency with which FPGA resources are utilised are comparable and in many cases beyond that achieved by those commercial tools. It is also worth noting that, in addition, the algorithms employed to derive these structures (i.e. the statement-based DFG synthesis approach and Algorithms 1 and 2) have complexity which is linear in the number of statements in the original C source.

These results and observations are highly encouraging. However, they mask inefficiencies in the architectures derived, in particular with respect to utilisation efficiency of memory resources. These issues are addressed in

Section VI.

VI. CSDF Tranformation for Accelerator Optimisation

Since the number of CSDF valves and their capacities directly influences the number of MEs and their capacity in the FPGA architecture, direct manipulation and optimisation of the cost of the FPGA architecture may be enabled by applying transformations to the CSDF model which reduce the number of valves, and transformations to the valves themselves to reduce their cost, specifically with the goal of reducing the on-chip memory footprint. Two such constructive transformations are presented: *valve merging* and *valve compression*.

A. Valve Merging

Consider $\{v_{i,c}, \cdots, v_{n,c}\}$ in Fig. 6. Each of these valves is realised on-chip and each results in a distinct physical ME component, which incurs BRAM and LUT cost. However, the output produced by $v_{i,c}$ is a subset of the input to $v_{n,c}$; at no point is any new data introduced. For the purpose, then, of reducing the on-chip memory cost, it is reasonable to ask - can the entire sequence not be realised by a single valve?

In general, this is not the case - specifically in cases such as that in Fig. 6 as data approaches an actor valve capacity reduces whilst data rates increase. These objectives compete to influence the physical resource providing the buffering facility - in general larger capacity implies BRAM storage whilst higher data rates require DisRAM storage. However, consider the subsequences $\{v_{i,c}, v_{j,c}\}$ and $\{v_{k,c}, v_{l,c}, v_{m,c}\}$. All elements of each subsequence have the same output data rate; accordingly, there is no bandwidth benefit to employing multiple MEs, indeed to do so increases cost and latency. In order to minimise cost for a given data rate, merging such sequences into a single composite is proposed.

Recall from Section IV that valves manipulate the shape of data as it moves between CSDF actors. Consider two valves, v_1 and v_2, each of which has a single input and a single output, which are to be merged to form a composite v_3. This composite will respect the consumption pattern of v_1 and the production patterns of v_2, whilst maintaining the buffer storage of the former. Trivially, the consumption function of v_3, $c_3 = c_1$. Since the buffer contents of v_3 also match that of v_1, the production function p_3 may be derived by the composition of p_1, c_2 and p_2 i.e. $p_3 = p_2 \circ c_2 \circ p_1$. The merging process is described in Algorithm 3.

The merging process analyzes each element of the sequence in turn, starting at v_1 (line 2); the element in question is compared with its immediate successors. In the case where the output data rate from v_j is not smaller than that from v_i, then no overall data rate reduction results from any potential merging, which is then considered on the basis of estimated cost. Specifically, the total cost of the two disparate valves (c_s, line 6) and their combination (c_m, line 7) are estimated. Since as a result of merging the nature of the memory element may change from a LUT-based DisRAM to a BRAM, the latter of which is substantially more resource-scarce on modern FPGA than LUT resource, a simple weighted function quantifying the number of Equivalent LUTs (ELUTs) [33] is adopted to denote the '*cost*' of a realisation:

$$cost\,(v_i) = 1328b + l \tag{1}$$

Algorithm 3 Valve-Chain Merging

```
1: procedure MERGEVALVES($V$)
2:     $i \leftarrow 1$
3:     while $i < |V|$ do
4:         $j \leftarrow i + 1$
5:         if $rate\,(v_j.out) \geq rate\,(v_i.out)$ then
6:             $c_s \leftarrow cost(v_i) + cost(v_j)$
7:             $c_m \leftarrow cost(merge(v_i, v_j))$
8:             if $(c_m < c_s)$ then
9:                 $v_i \leftarrow merge(v_i, v_j)$
10:                $V \leftarrow \frac{V}{v_j}$
11:            else
12:                $i \leftarrow i + 1$
13:    return $V$
```

Here, b and l denote the estimated BRAM and LUT costs of the ME, derived directly from its capacity. When the merged cost is lower than the cost of the two separate valves, then the two are merged (line 9) and the process repeats; otherwise it restarts with the next valve in the sequence (line 12). Fig. 11 illustrates the effect of this optimisation on the structure of the FSME accelerator. As this shows, there has been a considerable effect in this case - the entire chain of on-chip buffers for both *C* and *R* has collapsed to a single bank of BRAM buffers since the output data rates of all of these buffers are identical.

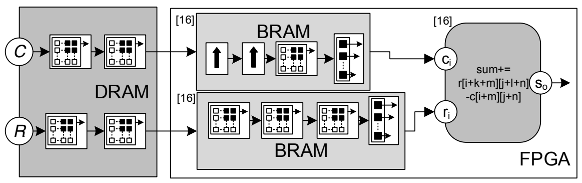
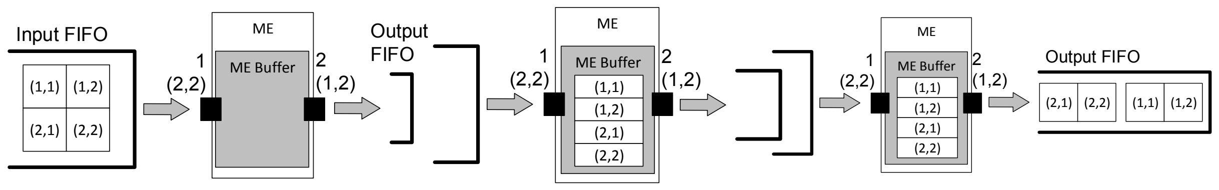


Fig. 11: Merged FSME Memory Architecture

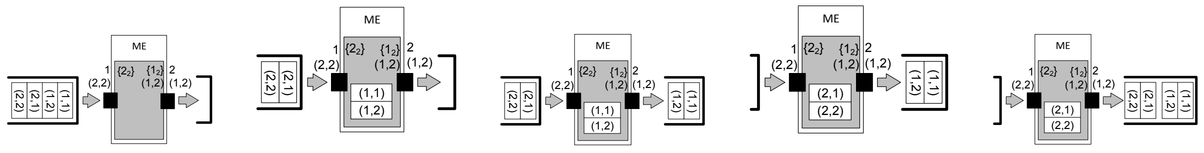
B. Valve Compression

1) Selective Storage Valves: As described in Section V, MEs contain sufficient memory storage to house a single data frame from each input port; however, in many instances this requirement is excessive. As a trivial example, consider a matrix - vector conversion valve illustrated in Fig. 12.

Fig. 12a - Fig. 12c illustrate the operation of this ME, assuming that SDF execution is maintained. Specifically, before firing (Fig. 12a) the entirety of the input matrix is available in the input FIFO, ready for consumption. The requirement for sequentlial consumption/operation/production behaviour in SDF dictates that the entirety of the



(a) DF: Pre-fire (b) DF: Post-consume (c) DF: Post-fire



(d) RTL: Pre-fire (e) RTL: p_1 (post-consume) (f) RTL post-p_1, pre-p_2 (g) RTL p_2 (post-consume) (h) RTL post-p_2

Fig. 12: Matrix-Vector Conversion: DF and Physical

input token is consumed before any output is produced, hence requiring storage for the entire matrix within the ME. When consumption is complete (Fig. 12b), the output vectors may be produced, with the resulting state of the system illustrated in Fig. 12c.

This approach requires a large amount of internal storage to store the entire input matrix. However, if it were known that the matrix would be incident on the ME in row major order, then the first output vector may be produced after only the first two elements of the input matrix were consumed. Then, since the elements within the ME internal buffer were no longer required, they may be overwritten by the last two elements of the input matrix, with the second output vector produced subsequently. This multi-phase behaviour may be captured precisely by an CSDF internal schedule. To enable this behaviour in an ME it is assumed:

- All input data frames are transferred into the ME in element-serial, row-major order
- Output tokens are atomic, input tokens are transparent

The latter assumption means that large input tokens are not considered atomic, so that each can be read over multiple phases of the internal ME CSDF schedule, whilst output tokens remain atomic, so that no elements of any output token is produced until all elements are available to be produced. Row major serialisation of data frames enforces their consumption as a sequence of elements; valves then operate in a cyclo-static manner with r_o phases, where $r_o \in \mathbb{Z}^+$ is the output rate. The input rate is a sequence of sets $\{x_n\}, n = 1, \cdots, r_o$, where x_i describes the triples of elements to be consumed during that firing. The output rate is $\{1_{r_o}\}$, i.e. a single token is produced. In the context of the matrix-vector conversion in Fig. 12, the behaviour in Fig. 12d - Fig. 12h may be enabled. Specifically, whilst the external dataflow of the RTL object if SDF, it operates an internal cyclo-static schedule which aggregates the integer production and consumption rates across multiple phases, each of which produces a single output token. In this case, since there are two output tokens, the number of internal phases is two, with the CSDF consumption and production sequences indicated in Fig. 12d - during each phase, two scalars are consumed and a single $(1, 2)$ vector produced. Hence, after consumption during the first phase p_1, the ME internal buffer contains only two scalars - enough only to produce the required output vector as illustrated in Fig. 12e. Once this has been produced at the end of p_1, the state of the input and output FIFOs and internal ME buffer are as shown in Fig. 12f. During the second phase p_2, the remaining input elements are consumed to overwrite the contents of the ME internal buffer (Fig. 12g), before production to end p_2 (Fig. 12h). The key observation is that the required quantity of internal ME buffering has been halved with respect to the strict SDF interpretation in Fig. 12a-Fig. 12c.

This simple example illustrates the valve compression concept - it reduces the quantity of internal ME buffering, without violating the dataflow semantics of the valve. The C kernel specification gives perfect visibility of the elements of the input data frame required for production of each output token at the level of individual token elements - this allows a compiler to record the set of elements required for definition of each output token. Given the word-serial nature of data consumption, ME controllers can then be derived which cease consumption when sufficient input data has been consumed, produce the corresponding output tokens and potentially overwrite elements which are not required for future output tokens.

Static analysis of the kernel from which the valve is derived allows three pieces of information to be derived:

1) $\{d_i\}$, the input frame elements required for definition of each token produced during each phase.
2) $\{e_i\}$, the input frame elements which may be discarded after each phase.
3) $\{x_i\}$, the elements consumed during each phase.

Assuming a p-phase CSDF actor, during phase i a set of input data words x_i are consumed and stored in ME buffer memory such that the aggregate of all data words consumed up to and including that firing contains at least d_i, i.e. $d_i \subseteq d_{i-1} \bigcup x_i$. If, after firing i, an element of d_i is not contained within $\{d_{i+1}, \cdots, d_{r_o}\}$, then the buffer space occupied by that element may be released. That is,

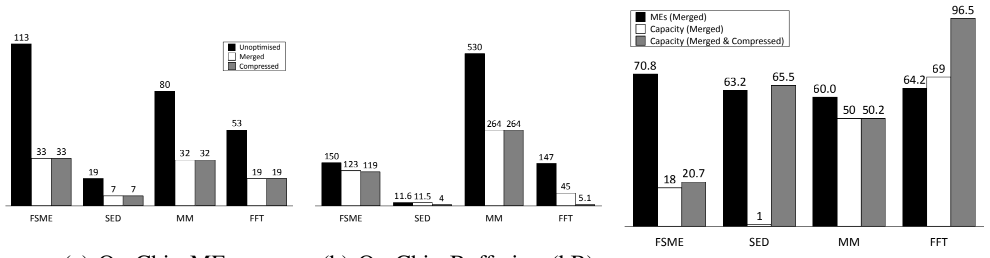
$$e_i = \frac{b_i}{\bigcup_{j=i+1}^{r_o} d_j} \tag{2}$$

where b_i denotes the set of elements in the internal buffer during phase i. Finally, x_i, the set of input elements to be consumed during phase i is given by

$$x_i := \begin{cases} \{0, \cdots, d_i\} & \text{if} \quad i = 1 \\ \{\max\{d_{i-1}\} + 1, \cdots, d_i\} & \text{if} \quad i < p \\ \{\max\{d_{i-1}\} + 1, \cdots, n\} & \text{otherwise} \end{cases} \tag{3}$$

where n denotes the index of the final data element in the input frame. The key memory capacity reduction step is the assignment of d_i to absolute memory locations which exploit the ability to overwrite elements included in e_i.

2) Destructive Consumption Valves: Whilst valve compression can reduce storage capacity by overwriting buffer elements which are no longer required, it assumes that, for at least one firing, every input data element is buffered. In general this may not be necessary. Where it is not, the input element may be consumed, but rather than being written into the ME buffer, is discarded.



(a) On-Chip MEs (b) On-Chip Buffering (kB) (c) Relative Reductions (%)

Fig. 13: Optimized Synthesis - On-Chip MEs and Buffering

In order to overcome this inefficiency and exploit any resulting reductions in buffer space, the ME must be able to selectively choose whether to store input data words in internal buffering, or to discard them. Actually, the information required to enable this behaviour is already available as a result of the kernel analysis performed for derivation of the selective storage valves. Specifically, in this case the storage of input elements is decoupled from their consumption, with only a subset $s_i \subseteq x_i$ written to internal buffer; s_i is given by:

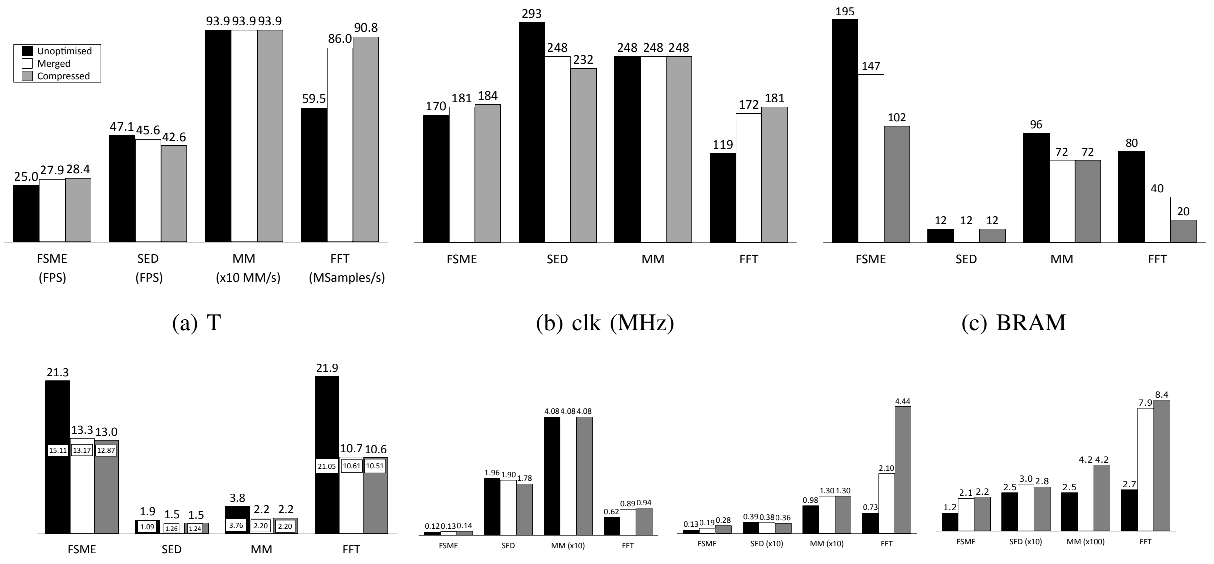
$$s_i = b_i \cap x_i \tag{4}$$

C. Experiments

The effect of the valve optimisations on the number of on-chip MEs and the total quantity of on-chip buffering for FSME, FFT, MM and SED is outlined in Fig. 13. As Fig. 13a shows, the number of MEs required to realise each operation is reduced substantially as a result of valve merging - Fig. 13c (MEs) shows reductions in the number of on-chip MEs by 70.8%, 63.2%, 60% and 64.2% for FSME, SED, MM and FFT respectively. However, these reductions are not universally accompanied by similar reductions in on-chip buffering capacity, reported in Fig. 13b. As Fig. 13c shows the relative reduction in on-chip buffering capacity exceeds that in the number of MEs in SED and FFT.

Fig. 14 illustrates the effect of these reductions on the performance, cost and efficiency of the resulting FPGA accelerators. As these are valve transformations, the number of DSP48E slices required is unchanged, but the reduced number and total capacity of MEs translates into substantial reductions in BRAM and LUT cost, as shown in Fig.14b and Fig.14c. Merging reduces BRAM cost by 24.6%, 25% and 50% for FSME, MM and FFT respectively, with valve compression increasing these savings further to 47.7% and 75% for FSME and FFT. Similarly, combinations of merging and compression reduce LUT cost by 39%, 21.9%, 41.6% and 51.6% across the four applications. The majority of these reductions come about from the removal of MEs in the merging step; it should be noted that each of the efficiency metrics now compare much more favourably to those of the circuits produced by Vivado-HLS or SynphonyC in Fig. 10.

In the case of SED, the throughput of the optimised solutions is lower than that of the original and whilst efficiency measured as throughput per unit resource is reduced in terms of DSP and BRAM resource, it remains



(a) T (b) clk (MHz) (c) BRAM

(d) LUT (e) T/DSP (f) T/BRAM (g) T/LUT

Fig. 14: Optimized Synthesis - Cost, Performance, Efficiency

that the slowest architecture is that which enables the performance requirement whilst incurring lowest resource cost. In all other cases efficiency is generally increased. Valve merging increases DSP efficiency by factors of 1.1 and 1.5 for FSME and FFT and BRAM (LUT) efficiency by 1.5 (1.8), 1.3 (1.7) and 2.9 (3.0) for FSME, MM and FFT. As a result of valve compression, FSME sees improvements in DSP, BRAM and LUT efficiency by respective factors of 1.1, 2.2 and 1.9 and FFT sees efficiency in the same resources increased by factors of 1.5, 6.1 and 3.2. It should be noted that this increases T/LUT beyond that achieved by SynphonyC and Vivado HLS for all applications except MM.

Many application-specific factors influence the cost reductions resulting from valve optimisations; however some general themes may be observed. The deeper the kernel loop nest, the greater the greater the scope of cost reduction via valve merging. Hence, for example, the six-level FSME loop nest and resulting four-ME chains on-chip offer much greater scope for optimisation than, for instance MM, which has a three-level loop nest with two hosted on-chip. Cost reductions in BRAM/DisRAM via valve compression are dependent on any capacity reduction being sufficiently large so as to remove entire components, since RAM components of wildly different capacities can incur the same on-chip cost.

It should, again, also be noted that the optimisations presented in this section all exhibit complexity linear in the number of DFG nodes and therefore in the number of statements in the C program from which they originated.

The capabilities of this constructive approach presented are highly encouraging and unique amongst HLS solutions for FPGA. They form a solid foundation upon which a comprehensive HLS approach may constructed which considers global design space exploration and multi-objective accelerator optimisation. The benefits of such an approach are implied by a number of the results presented here. For example, consider the SED and MM accelerators.

The constructive unrolling, partitioning and valve transformation approaches presented in this section and Section V have enabled real-time performance substantially beyond that of the target; these approaches are not sufficiently capable to trade this excess performance for reduced cost - iterative algorithms, such as simulated annealing or genetic algorithms are typically required in a situation such as this [7]. Performance/cost trade-offs are likely to arise by exploring, for instance, varying the partition point between on-chip and off-chip memory, assigning buffers to one of multiple banks of off-chip RAM, realising on-chip buffers of different sizes as either BRAM or DisRAM or realising arithmetic operations in PEs using DSP48e or LUTs, in an application specific manner. This requirement is prominent amongst other potential extensions, such as multi-kernel system optimisation and datapath-oriented optimisations, for instance recognising compound operations such as multiply-accumulate (MAC) and realising these using 1 DSP48e rather than two primitive FUs, are likely to have a strong impact on utilisation of these resources. Whilst the approach presented here does not address these standalone, it is the first to present a viable constructive process upon which the solutions may be developed.

VII. Conclusion & Future Work

HLS techniques for deriving custom FPGA computing architectures from software languages such as C, C++ and OpenCL, are emerging but are still limited in derivation of custom memory structures. This paper has presented a constructive C kernel synthesis approach which can derive FPGA accelerator architectures, including multi-level on-chip memory structures and off-chip RAM, such that a specified performance constraint may be achieved. By utilising a statement-oriented approach to derive CSDF IRs of the C kernel, memory hierarchies are exposed in such a way that these can be partitioned between off-chip and on-chip buffers and multi-level on-chip structures derived. Further, by automatically parallelising these structures via loop unrolling in the C kernel user-defined throughput targets may be satisfied in an automated fashion. To the best of the authors' knowledge, this is the first record of such a solution. Furthermore, it has shown how, by application of valve merging and compression transformations, performance and cost may be optimised whilst retaining real-time performance.

This approach lays a solid foundation for a more comprehensive HLS approach which support iterative design space exploration for derivation of more efficient results.

Acknowledgment

The authors wish to acknowledge the financial support for this work provided by the Department of Education and Learning Northern Ireland (DEL NI) and the UK Engineering and Physical Sciences Research Council (EPSRC).

References

[1] Xilinx Inc., *7 Series DSP48E1 Slice User Guide*, Aug. 2013.

[2] Altera Inc., *Stratix V Device Handbook*, Jan. 2014.

[3] Xilinx Inc., *7 Series FPGAs Memory Resources User Guide*, Jan. 2014.

[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473 –491, April 2011.

[5] M. Milford and J. McAllister, "Memory-centric VDF Graph Transformations for Practical FPGA Implementation," in *Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on*, Oct 2012, pp. 12–18.

[6] ——, "Automatic FPGA Synthesis of Memory Intensive C-based Kernels," in *Embedded Computer Systems (SAMOS), 2012 International Conference on*, July 2012, pp. 136–143.

[7] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[8] Banerjee, P. et al., "Overview of a Compiler for Synthesizing MATLAB Programs onto FPGAs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 3, pp. 312–324, March 2004.

[9] *Vivado Design Suite User Guide - High Level Synthesis*, Ug902 (v2013.2) ed., Xilinx Inc., Jun. 2013.

[10] Xilinx Inc., *SDAccel Development Environment*, Jul. 2015.

[11] Kronos OpenCL Workng Group, *The OpenCL Specification, Version 2.1*, Jan. 2015.

[12] J. Cong, P. Zhang, and Y. Zou, "Optimizing Memory Hierarchy Allocation with Loop Transformations for High-Level Synthesis," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 1229–1234.

[13] ——, "Combined Loop Transformation and Hierarchy Allocation for Data Reuse Optimization," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, Nov 2011, pp. 185–192.

[14] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based Data Reuse Optimization for Configurable Computing," in *Field Programmable Gate Arrays, 2013 ACM/SIGDA International Symposium on*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 29–38. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435273

[15] P. Feautrier, "Automatic Parallelization in the Polytope Model," in *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. London, UK, UK: Springer-Verlag, 1996, pp. 79–103. [Online]. Available: http://dl.acm.org/citation.cfm?id=647429.723579

[16] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, "System Design using Khan Process Networks: The Compaan/Laura Approach," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 1, Feb. 2004, pp. 340–345 Vol.1.

[17] G. Kahn, "The Semantics of Simple Language for Parallel Programming," in *Proc. IFIP Congress 74*. North-Holland Publishing Co., 1974, pp. 471–475.

[18] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and Automated Multiprocessor System Design, Programming, and Implementation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 3, pp. 542–555, March 2008.

[19] ——, "Efficient External Memory Interface for Multi-Processor Platforms Realized on FPGA Chips," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, Aug 2007, pp. 580–584.

[20] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SYSTEMCODESIGNER - An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications," *Design Automation of Electronic Systems, ACM Transactions on*, vol. 14, no. 1, pp. 1–23, 2009.

[21] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich, "FunState-An Internal Design Representation for Codesign," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 4, pp. 524–544, Aug. 2001.

[22] P. Lippens, J. Van Meerbergen, A. van der Werf, W. Verhaegh, B. McSweeney, J. Huisken, and O. McArdle, "PHIDEO: a Silicon Compiler for High Speed Algorithms," in *Design Automation. EDAC., Proceedings of the European Conference on*, Feb 1991, pp. 436–441.

[23] P. Lippens, J. Van Meerbergen, A. van der Werf, W. Verhaegh, and B. McSweeney, "Memory Synthesis for High Speed DSP Applications," in *Custom Integrated Circuits Conference, 1991., Proceedings of the IEEE 1991*, May 1991, pp. 11.7/1–11.7/4.

[24] J. Keinert and J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*. Springer, 2011.

[25] K. Denolf, M. J. G. Bekooij, J. Cockx, D. Verkest, and H. Corporaal, "Exploiting the Expressiveness of Cyclo-Static Dataflow to Model Multimedia Implementations," *EURASIP J. Adv. Sig. Proc.*, vol. 2007, 2007.

[26] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397 –408, Feb 1996.

[27] M. Bamakhrama, J. Zhai, H. Nikolov, and T. Stefanov, "A Methodology for Automated Design of Hard-Real-Time Embedded Streaming Systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, March 2012, pp. 941–946.

[28] T. Harriss, R. L. Walke, B. Kienhuis, and E. F. Deprettere, "Compilation From Matlab to Process Networks Realized in FPGA," *Design Autom. for Emb. Sys.*, pp. 385–403, 2002.

[29] E. Deprettere, T. Stefanov, S. Bhattacharyya, and M. Sen, “Affine Nested Loop Programs and their Binary Parameterized Dataflow Graph Counterparts,” in *Application-specific Systems, Architectures and Processors, 2006. ASAP ’06. Intl. Conf. on*, Sept 2006, pp. 186–190.

[30] E. Lee and D. Messerschmitt, “Synchronous Data Flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept 1987.

[31] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, “Combining Data Reuse With Data-Level Parallelization for FPGA-Targeted Hardware Compilation: A Geometric Programming Framework,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 3, pp. 305–315, March 2009.

[32] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, “Computer Generation of Hardware for Linear Digital Signal Processing Transforms,” *Design Automation of Electronic Systems, ACM Transactions on*, vol. 17, no. 2, pp. 15:1–15:33, Apr 2012.

[33] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, and D. Tullsen, “Application-Specific Customization of Parameterized FPGA Soft-Core Processors,” in *Computer-Aided Design, 2006. ICCAD ’06. IEEE/ACM International Conference on*, Nov. 2006, pp. 261–268.

Matthew Milford graduated in 2009 with an MEng in Electronic and Software Engineering from Queen’s University Belfast. Matthew’s dissertation focussed on Soft Processor Design for FPGA which formed the basis of a PhD investigating Compiler Design for Embedded Streaming Systems. Matthew currently works as a Software Engineer for Andor Technology in Belfast working closely with 3rd party and OEM customers such as Olympus, Nikon, Leica and Zeiss who are integrating Andor cameras into their life science microscopy systems.

John McAllister (S’02-M’04-SM’12) received the Ph.D. degree in Electronic Engineering from Queens University Belfast, U.K., in 2004. He is currently a member of academic staff in the Institute of Electronics, Communications and Information Technology (ECIT) at the same institution. His research focusses on embedded realisation of high performance signal processing architecture, in particular modelling languages, compilers and architectures for FPGA-based signal processing. He is a cofounder of Analytics Engines Ltd., a member of the Advisory Board to the IEEE Signal Processing Society Technical Committee on Design and Implementation of Signal Processing Systems (DISPS), an Associate Editor of IEEE TRANSACTIONS ON SIGNAL PROCESSING, Chief Editor of SigView and a member of the editorial board of Springer’s Journal of Signal Processing Systems.