Data Reuse Analysis for Automated Synthesis of Custom Instructions in Sliding Window Applications

Georgios Zacharopoulos, Giovanni Ansaloni, Laura Pozzi

Faculty of Informatics
University of Lugano (USI)
Switzerland
{georgios.zacharopoulos, giovanni.ansaloni, laura.pozzi}@usi.ch

ABSTRACT

The efficiency of accelerators supporting complex instructions is often limited by their input/output bandwidth requirements. To overcome this bottleneck, we herein introduce a novel methodology that, following a static code analysis approach, harnesses data reuse in-between multiple iteration of loop bodies to reduce the amount of data transfers. Our methodology, building upon the features offered by the LLVM-Polly framework, enables the automated design of fully synthesisable and highly-efficient accelerators. Our approach is targeted towards sliding window kernels, which are employed in many applications in the signal and image processing domain.

1. INTRODUCTION

Application-Specific Instruction Set Processors (ASIPs) are digital architectures that support hybrid HW-SW execution. They comprise a CPU and tightly or loosely coupled hardware accelerators, where critical parts of computation can be accelerated.

A large body of research exists in the identification of computational kernels [2, 4, 6, 7], i.e. detecting automatically, from source code, applications hotspots to be implemented in HW. Hardware accelerated kernels are then exposed to software as a single and complex *Custom Instruction (CI)*. These works traditionally target dataflow computation, and mostly do not tackle control flow nor memory transfers — a limitation that we attempt to overcome in this paper.

Alongside CI identification, an equally important dimension is the *automatic synthesis* of such instructions from a high-level (e.g.: C language) description, as the efficiency of the resulting HW implementation makes a tangible difference in performance. High Level Synthesis (HLS) [5, 10, 15, 17], aims exactly at performing this SW to HW translation efficiently. While existing high-level synthesis tools are generally effective in automatically applying optimisations in data-flow in order to produce good designs, the optimisation *of control-flow and of memory transfers* still poorly translates automatically to hardware.

Complex CIs, spanning entire (possibly nested) loops, commonly exhibit a high degree of *data reuse*, i.e.: the intersection between the sets of input data values being processed in subsequent iterations is quite large. Effective exploitation of data reuse greatly lowers the required bandwidth to and from accelerators, thus minimising data transfer overheads and, ultimately, increasing their performance. The application of data reuse analysis to guide CI synthesis is still in its infancy. In fact, state of the art methods either rely on manually rewriting the source code before high-level synthesis [15] or on source-to-source translation [11] [12], and are therefore poorly integrated in HLS toolchaines.

This work attempts to bridge this gap. It presents a compiler-driven framework, based on the LLVM Polly [14] library, able to identify data reuse opportunities in computational kernels in order to guide the automatic synthesis of complex hardware accelerators. Such accelerators aptly exploit unrolling and pipelining, by embedding a local storage holding elements which are re-used across iterations.

We herein target sliding window applications, common in the image processing field. In such domain, a transformation is applied to each element of a large two-dimensional input array (a frame) according to the values in a small neighbourhood (a window). We consider large, but constrained, input/output links as the main architectural constraint in the design of the CI accelerators. Such arrangement is usually supported by commercial ASIP platforms, e.g.: the Tensilica Xtensa processor [13] supports CIs interfaced with 512-bits interconnects.

2. RELATED WORK

In the state of the art in automatic identification of custom instructions, research has so far focused mostly on accelerating data-flow [4, 6], disregarding the opportunities offered by the optimization of memory accesses. An exception is provided by papers [2, 7] where the authors acknowledge that custom instructions that include storage can provide larger speedup than those which attempt to accelerate dataflow only. However, both these papers focus on the identification

IMPACT 2017
Seventh International Workshop on Polyhedral Compilation Techniques
Jan 23, 2017, Stockholm, Sweden
In conjunction with HiPEAC 2017.
http://impact.gforge.inria.fr/impact2017

of the custom instructions, and do not present a high-level synthesis methodology to automatically and efficiently implement them as hardware accelerators.

In the context of sliding window applications, this challenge has been addressed both by research effort and commercial tools. The smart buffers [5] generated by the ROCCC compiler [17] allow to automatically detect data reuse opportunities, but, as opposed to our work, don't have the flexibility to interface with interconnects of varying width. The methodology described in [10] employs reuse buffers spanning multiple frame columns, which pose a significant area overhead. Both [5] and [10] are not able to combine hardware unrolling and pipelining, which are instead jointly supported by our framework. An alternative approach, described in [3], is also resource-intensive, as it requires the storage of large parts of a frame being processed inside the custom hardware. In [18], the authors propose an analytical method to gather microarchitectural parameters for sliding-window applications on FPGAs. Their design however ultimately needs to be manually implemented and hence the work neglects high level synthesis aspects. The commercial Vivado_HLS High-Level Synthesis tool requires extensive manual effort at the source code (C) level to instantiate reuse buffer. Conversely, our approach relies on automated code analysis to derive the characteristics of a target application.

As opposed to [11], [12] and [10], which rely on source-to-source transformations to expose optimisation opportunities for HLS synthesis, we detect and exploit these benefits directly. In [11] and [10], only single-datapath designs are considered, neglecting the benefits offered by wide interconnects and multiple datapaths. These opportunities are leveraged in [12], which adopts a strategy based on super-pixels and large internal buffers. In our methodology, instead, only the data elements belonging to an overlapping set of windows have to be locally stored, resulting in more area-efficient accelerators. In [11] and [10] data reuse is exploited only across two consecutive iterations of the innermost loops. We increase the scope (and the performance benefits) of data reuse analysis by also considering the opportunities present across iterations of the external loop.

Our framework allows efficient processing of a two- dimensional input data of pre-determined size. Our contribution is therefore agnostic with respect to optimisations, such as loop tiling and communication coalescing [1], that aim at reshaping such boundaries.

3. METHODOLOGY

In order to generate our custom-storage accelerators we proceed with a two-steps methodology, where first we identify and then we synthesise the part of computation to be implemented in HW. The phases of identification and synthesis are depicted in Figure 1, along with the evaluation procedure we follow.

Here, we first briefly describe how the part of computation to be accelerated can be identified (Section 3.1). The optimisation of data accesses is essential to derive efficient CI implementation for window applications. In this light, in

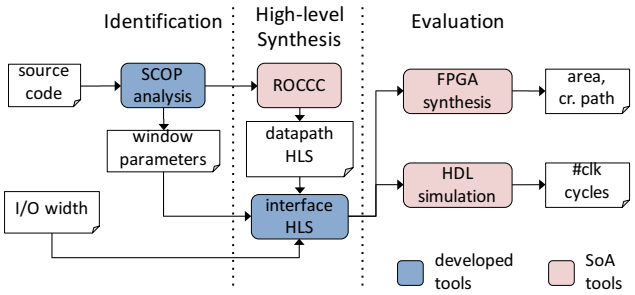

Figure 1: Block scheme of our framework. Code analysis retrieves the parameters of SCoP kernels, which are employed, along with input/output constraints, to guide the automated design of highly efficient hardware accelerators.

Section 3.2 we discuss data reuse opportunities in our target domain and, in Section 3.3, how they can be automatically harnessed with static code analysis. Finally, we provide details on the resulting hardware implementations.

3.1 Custom Instructions Identification

Figure 2 shows the data flow graph (DFG) and the control flow graph (CFG) for the Sobel Filter application kernel. The Sobel Filter is a sliding-window application where a window of 3x3 moves across an image, with a stride of 1 (horizontally and then vertically), and eight of the nine pixels in each window are then used to compute an output. The corresponding DFG is depicted in the inner loop, with eight loads and one store (red nodes), and dataflow computation in between. State of the art identification algorithms traditionally focus on dataflow only. Given a number of inputs and outputs that can be exchanged between processor and

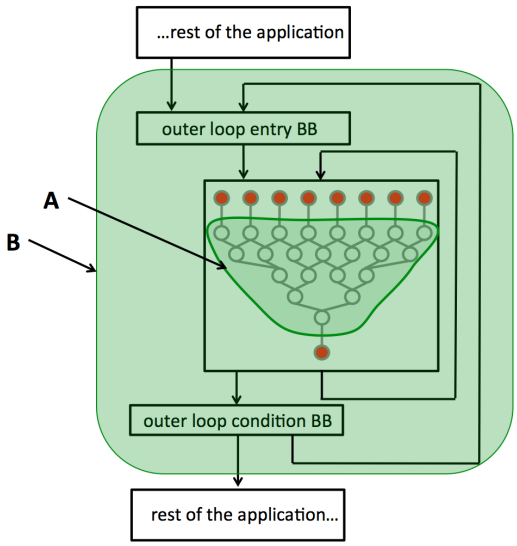

Figure 2: State of the art identification algorithm traditionally focus on dataflow computation only (subgraph A). In this paper, instead, the control flow graph (CFG) of the application is also analysed, in search for static control parts (SCoPs) — subgraphs of the CFG where the flow of control is known *statically*. The whole SCoP region (subgraph B) is hence identified and selected as accelerator.

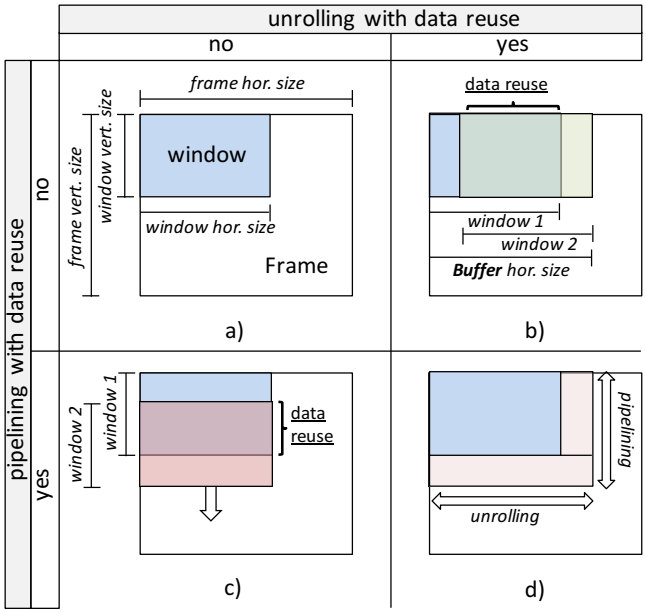

Figure 3: At each iteration, sliding window applications process a subset of the input data (a). The managed set of subsequent iterations present a high degree of overlap, both in the horizontal (b) and vertical (b) dimensions. Our framework automatically leverages both, maximising data reuse (d).

accelerators, identification algorithms such as [4, 6] would select the dataflow computation as a suitable custom instruction (subgraph A in the figure), leaving loads and store nodes out, as forbidden.

Conversely, we propose in this work to identify the entire loop (subgraph B) as a candidate CI, including loads and stores. The rationale motivating our strategy is two-fold. First, larger CI, possibly leading to higher speed-ups, can be identified. Second, costly memory accesses can be minimised by employing optimisations whose scope encompasses multiple loop iterations and that aggressively exploit data reuse opportunities (discussed in Section 3.2). Hence, custom instruction identification is in this paper essentially moved *beyond* the basic block boundary, by selecting the whole CFG subgraph as a custom instruction.

In order to identify code sections corresponding to sliding window kernels, the control flow graph (CFG) of the application is analysed, looking for loop nests. Then, the LLVM Polly [14] library is used to verify whether the CFG structure of nests is a SCoP (Static Control Part), which is a *subgraph of the CFG where the flow of control is known statically.* If so, its polyhedral model, derived by Polly, is derived to provide the SCoP parameters required for its hardware implementation, as detailed in Section 3.3.

3.2 Data Reuse Opportunities

Our framework leverages both hardware unrolling and pipelining to achieve a high degree of inter-iteration data reuse. Figure 3 illustrates the reuse concept from a high-level perspective, assuming, without loss of generality, that the sliding window moves first in the vertical direction and then, at the end of each frame column, in the horizontal one.

Algorithm 1 LLVM Analysis Pass - Data Reuse Analysis

```
1: function RunOnRegion()
2:     getAnalysis(ScopInfo)
3:     scop = getScop()
4:     RunOnScop(scop)
5:
6: function RunOnScop(scop)
7:     LI = getLoopInfo()
8:     SE = getSE()
9:     if L == OutermostLoop then
10:        getTripCountForOutermostLoop()
11:        getStrideForOutermostLoop()
12:    else if L == InnerMostLoop then
13:        getTripCountForInnermostLoop()
14:        getStrideForInnermostLoop()
15:        getReadMemoryAccesses()
16:        ComputeDistancesForReadAccesses()
17:        ComputeWindowSize()
```

Window applications proceed by computing output values from a subset of an entire frame, localized in a small two-dimensional block. It is therefore possible to limit the internal storage of the local memory of the accelerator to the data used in a single window, called the *managed set*, resulting in a compact hardware implementation (Figure 3a).

Nonetheless, by adopting a larger local memory, the data required by multiple windows can be stored at the same time. We observe (Figure 3b) that the managed sets of horizontally adjacent windows are highly overlapping, only differing by a number of columns of elements equal to the vertical stride of the application. Multiple (overlapping) managed sets can therefore be supported with little overhead in the size of the local buffer. Each window enclosed in the buffer can then be processed in parallel by a different datapath, implementing unrolling with data reuse in hardware.

Data reuse is also present in the vertical dimension. In fact, the managed set of subsequent iterations differ by a number of rows equal to the horizontal stride (Figure 3c). This source data reuse can be efficiently harnessed by hardware pipelining, implementing the local storage as a row-wise shift register.

Our framework supports both types of data reuse concurrently (Figure 3d). In this setting, updating the managed set entails the transfer from main memory of the data corresponding to a buffer row (as opposed to a window row in Figure 3c). We use this added degree of freedom to tailor both the local storage structure and the number of implemented datapaths according to the input/output width of the communication interface (i.e.: number of data elements that can be concurrently read or written).

3.3 Data Reuse Analysis

To automate the analysis of data reuse in SCoPs, information about the window size, the stride and the frame size must be collected from the applications source code. The window size defines the access pattern within the innermost body of the loop. The innermost and outermost loop stride is the value of the induction variable increase for the in-

nermost and the outermost loop respectively. Finally, the frame size is defined as the iteration space in which the sliding window is moving. To obtain these values, we developed a compiler analysis pass, building on the capabilities offered by the LLVM Polly framework [14]. Application-specific parameters are then considered in conjunction with architecture constraints (input/output width) to automatically synthesise efficient SCoP accelerators.

The Analysis Pass that we have developed iterates over regions of the application functions identified as Static Control Parts (SCoPs) by Polly. As reported in Algorithm 1, for each SCoP Loop and Scalar Evolution (SE) information are extracted from the current body of the loop, by using the analysis passes provided by LLVM. Loop information provides the loop depth, and thus whether a loop is the innermost or the outermost one in a SCoP. SE information includes the loop trip count method, which computes the iteration space for each loop. This information enables the computation of the frame size.

The SCoP horizontal and vertical stride is calculated by the *getStrideForLoop* function, which takes as argument the basic block corresponding to the loop body. We developed it by leveraging the *getStride* method included in the LLVM ScopInfo Analysis pass and functions included in the Integer Set Library (`isl`) [16].

Finally, the read memory accesses residing in the innermost body of the loop are identified by using `isl` functions within our own *ComputeDistancesForReadAccesses* function. We compute the distance (or delta) of each of these accesses with respect to the first identified one. From the access pattern, the window size is computed as its minimum enclosing rectangle.

3.4 Hardware implementation

The parameters retrieved with SCoP analysis (horizontal and vertical window size, stride, domain) and the characteristics of the interconnect (input and output width) are employed to derive efficient CI implementation with local storage and data reuse.

As shown in the block scheme in Figure 4, accelerators implementations embed multiple combinatorial datapaths[1], each executing one iteration of the loop body of the target application. Their input interface embeds a local storage, whose horizontal size corresponds to the available input data width of IN_W data elements, while its vertical size is equal to the vertical dimension of the application window v. It is implemented as a $IN_W * v$ shift register, operating in the vertical (top-down) direction. During execution, the first row of the shift register is filled with input data in each clock cycle. A subset of the elements stored in the shift register is connected to each of the different datapaths according to their managed sets, e.g.: the first one having inputs corresponding to the buffer columns ranging from 0 to $h-1$ (the horizontal size of the window). Figure 4 illustrates such scheme for the simple case of $IN_W = h + 1$.

[1]While we consider only combinational datapaths in the experiments presented in Section 5, our methodology can be also applied to multi-cycle datapaths.

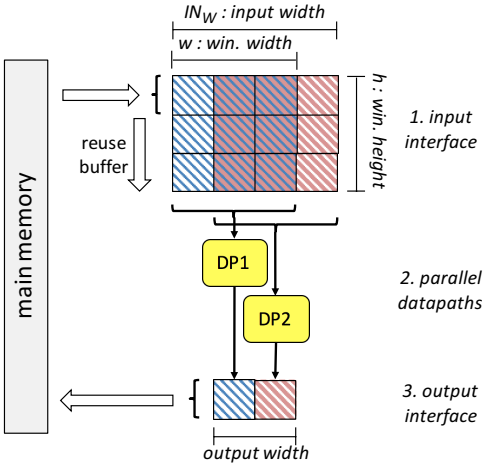

Figure 4: The automatically generated accelerators include multiple datapaths, executing the loop body of two-dimensional SCoPs.

	Pipelining data reuse	Unrolling data reuse	Parametric I/O width
Our	Yes	Yes	Yes
Vivado	Manual	No	Yes
ROCCC	Yes	No	No

Table 1: Summary of the features of the accelerators generated by our framework, compared with state-of-the-art tools.

At the start of operations, v rows are stored in the shift register before activating the datapaths logic. Afterwards, this activation is performed for each new row, discarding the last (topmost) line and storing a new one in the first (bottom) position of the shift register. At the completion of a vertical slide of the window through the frame, a new one is started, increasing the horizontal displacement of the buffer by $IN_W - h + 1$ elements.

Finally, since no reuse opportunities are present for outputs, the output interface simply concatenates the values generated by the datapaths, and transfers them as a single and wide memory access.

4. EXPERIMENTAL SETUP

We evaluated the performance of accelerators generated by our methodology with respect to equivalent ones generated by the ROCCC [17] and Vivado_HLS high level synthesis tools. Table 1 provide a high-level summary of the main features of each framework.

Vivado_HLS directives allow to (manually) reshape input and output vectors to interface with wide input and output connections comprising multiple data elements, but has limited support for data reuse. In more detail, [15] describes how source code must be (manually) restructured to obtain a pipelined implementation, but does not mention how unrolling (with data reuse) can be performed on the modified code. In our comparative evaluation, we consider both implementations optimised only by means of directives (named Vivado_norew in the following experiments) as well as ones deriving from an extensive rewrite of the source code (Vi-

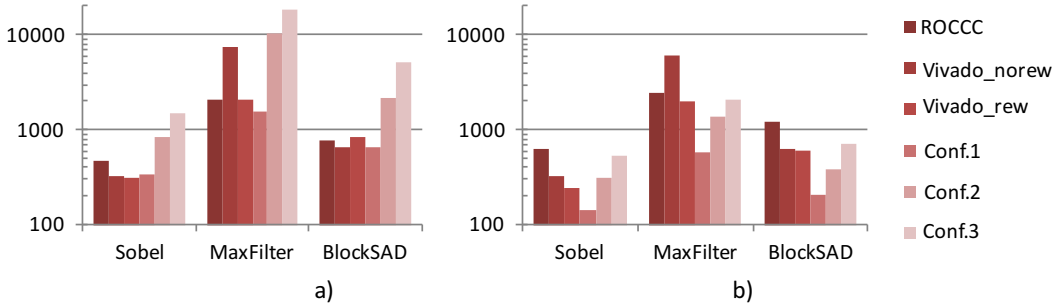

Figure 5: Comparison of required resources for our generated systems and for baseline approaches: LUTs (a) and Flip-Flops (b). ROCCC implementations also employ 2 BRAM elements.

Benchmark	Window size	Input width, #DPs Conf.1	Conf.2	Conf.3
SAD	4x4	4, 1	8, 5	16, 13
Max. Filter	8x8	8, 1	16, 9	24, 17
Sobel	3x3	3, 1	8, 6	16, 14

Table 2: Benchmarks characteristics and employed configurations. Conf.1 presents a single datapath, while Conf.2 and Conf.3 have a moderate to large number of datapaths. Window sizes and input widths are expressed in bytes.

vado_rew). ROCCC, similarly to [11] and [10], automatically detects opportunities for a pipelined data reuse, but faces has limited support for unrolling, resulting in accelerators which only embed a single datapath. Additionally, it cannot generate systems with unaligned accesses to multiple (contiguous) data elements, required in sliding window applications to take advantage of large I/O widths. In our framework, those three features are automatically leveraged, *without requiring any code rewriting.*

For all experiments, the number of clock cycles employed to process a frame was retrieved from HDL simulations. For implementation, we targeted a Xilinx Virtex7 FPGA. We employed low-level synthesis to measure the length of critical paths and the amount of required resources (flip-flops and look-up tables).

5. EXPERIMENTS

In this section, we comparatively evaluate the benefit of our approach, from a performance and resource usage viewpoint. We employed benchmarks with varying window sizes, as summarised in Table 2. BlockSAD, a hot-spot from Block Motion Search in H264, is used to find the similarity between 4x4 blocks in neighbouring regions of different frames. Our implementation is adapted from the JM H264 reference [9]. Maximum Filter computes the brightest pixel among neighbours in 8x8 blocks. Sobel is a popular method for edge detection. In all cases, similarly to [10], we considered 100x100 frames, with each pixel encoded as a single byte. Also described in Table 2 are the three considered configurations for each benchmark, employing either a single datapath (Conf.1) or multiple ones (Conf.2, Conf.3).

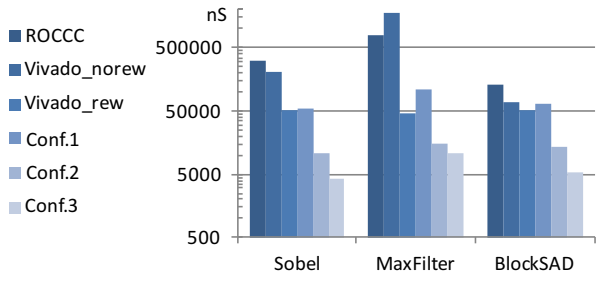

Figure 6: Execution time to process a 100x100 frame.

Execution Time. Execution times are reported in Figures 6 for the target FPGA. The first consideration that can be derived from these results is on the effectiveness of data reuse. ROCCC systems, which transfer a single pixel per data access but implement pipelining, have similar performance with respect to Vivado_norew ones, which can read a window row of pixels per access but do not exploit data reuse. Secondly, Conf.1 accelerators — even though they do not require code modifications — are as efficient as Vivado_rew ones, as they both present wide input/output widths as well as pipelining with data reuse. Finally, unrolling with reuse, that is supported only by our framework, dramatically decreases run-times, with an order-of-magnitude speed up on average between Conf.1 and Conf.3. The other state of the art tools *cannot provide a comparable solution with such low execution time.*

Required resources. Figure 5 reports the amount of resources required by our generated accelerators, compared with the considered baseline alternatives. Unsurprisingly, accelerators featuring a high number of datapaths (Conf.3) require more resources than single-datapaths alternatives (Conf.1, Vivado). Nonetheless, the area increase is sublinear with respect to the amount of datapaths, as the size of the input buffer only grows slightly to support a higher degree of parallelism. In fact, results highlight that complex accelerators employ a substantial amount of combinatorial logic (implemented with LUTs in FPGAs), but are competitive with other HLS frameworks regarding the amount of required memory resources (Flip-Flops in FPGAs).

As illustrated in Figure 7, the speedup improvements due to parallel datapaths compare favourably with the area overheads: in the case of the BlockSAD benchmark, for example, Conf.3, which embeds 13 parallel datapaths, requires 6x more LUTs with respect to the Vivado_rew implementation, but at the same time results in a 9.2x speedup. Again, it is important to note that state of the art tools do not support unrolling with data reuse, but stop at the level of speedup that can be achieved by single-datapath solutions.

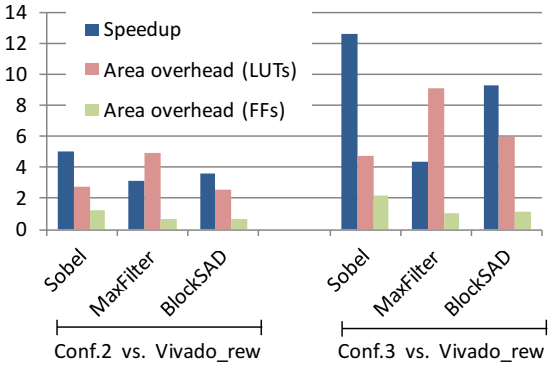

Figure 7: Comparison of multi-datapath and optimised Vivado implementations.

Discussion. Our framework *brings High-Level Synthesis one step closer to mimicking manual hardware-designer decisions.* The Conf.3 accelerator for BlockSAD, presented above, is essentially the same as the one designed manually by Hameed et. al [8], in a paper aimed at manually designing accelerators for the H264 application. The authors have indeed chosen to invest area for as many as 16 BlockSAD datapaths in parallel, in order to 1) maximise speedup and reuse, and 2) exploit the high bandwidth present between processor and accelerator, in their Cadence Tensilica Xtensa processor [13]. The present work mimics the rationale followed there, but is able to do so *automatically.* HLS state of the art tools can automate some of the decisions taken by our framework, but not all — in particular, they cannot automatically and jointly exploit unrolling and pipelining while considering reuse, and hence deliver the levels of speedup provided here.

6. CONCLUSIONS

This paper has presented a methodology, based on static code analysis, to automatically optimise the high level synthesis of accelerators dedicated to sliding window applications. Our framework leverages data locality, typical of the target domain, by exploiting data reuse in conjunction with aggressive hardware unrolling and pipelining. It results in an order-of-magnitude performance improvement with respect to state-of-the-art methodologies, without requiring manual modifications of the source code of applications.

7. REFERENCES

[1] C. Alias, A. Darte, and A. Plesco. Optimizing remote accesses for offloaded kernels: application to high-level synthesis for FPGA. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 575–580, Mar. 2013.

[2] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi. Automatic identification of application-specific functional units with architecturally visible storage. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 212–217, Mar. 2006.

[3] Y. Dong, Y. Dou, and J. Zhou. Optimized generation of memory structure in compiling window operations onto reconfigurable hardware. In *Reconfigurable Computing: Architectures, Tools and Applications, ARC*, pages 110–121, 2007.

[4] E. Giaquinta, A. Mishra, and L. Pozzi. Maximum convex subgraphs under I/O constraint for automatic identification of custom instructions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(3):483–494, 2015.

[5] Z. Guo, B. Buyukkurt, and W. A. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. In *Proceedings of the 2004 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 249–256, 2004.

[6] G. Gutin, A. Johnstone, J. Reddington, E. Scott, and A. Yeo. An algorithm for finding input-output constrained convex sets in an acyclic digraph. *J. Discrete Algorithms*, 13:47–58, 2012.

[7] M. Haaß, L. Bauer, and J. Henkel. Automatic custom instruction identification in memory streaming algorithms. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 1–9, Oct. 2014.

[8] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. *Commun. ACM*, 54(10):85–93, 2011.

[9] HHI. *H.264/AVC Reference Software, http://iphome.hhi.de/suehring/tml/*, 2014.

[10] W. Meeus and D. Stroobandt. Automating data reuse in high-level synthesis. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1–4, Mar. 2014.

[11] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the 2013 ACM/SIGDA 21st International Symposium on Field Programmable Gate Arrays*, pages 29–38, Feb. 2013.

[12] M. Schmid, O. Reiche, F. Hannig, and J. Teich. Loop coarsening in C-based high-level synthesis. In *Proceedings of the 26th International Conference on Application-specific Systems, Architectures and Processors*, pages 166–173, July 2015.

[13] Xtensa customizable processors: http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.

[14] C. L. Tobias Grosser, Armin Groesslinger. Polly - Performing polyhedral optimizations on a low-level intermediate representation. In *Parallel Processing Letters*, Apr 2012.

[15] F. Valina. Implementing memory structures for video processing in the Vivado HLS tool. In *Xilinx*, Sept. 2012.

[16] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[17] J. R. Villarreal, A. Park, W. A. Najjar, and R. Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *Proceedings of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 127–134, 2010.

[18] H. Yu and M. Leeser. Automatic sliding window operation optimisation for FPGA-based computing boards. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 76–88, April 2006.