http://eprints.gla.ac.uk/182352/

Deposited on: 19 March 2019

Smart-Cache: Optimising Memory Accesses for Arbitrary Boundaries and Stencils on FPGAs

Syed Waqar Nabi
School of Computing Science
University of Glasgow, Glasgow G12 8QQ
syed.nabi@glasgow.ac.uk

Wim Vanderbauwhede
School of Computing Science
University of Glasgow, Glasgow G12 8QQ
wim.vanderbauwhede@glasgow.ac.uk

Abstract—A key requirement for high performance on FPGAs is to maintain *continuous data streaming* from the DRAM. An impediment in many computations, especially in the scientific computing domain, is irregular stencils and boundary conditions, requiring memory accesses that are random, redundant, or both. To address this problem, we present *Smache*, a novel *smart-caching* framework that uses FPGA on-chip memory resources for optimising access for arbitrary stencil shapes and boundary conditions. We propose a combination of *stream* and *static* buffers, and it is the latter that allows arbitrarily large offsets in stencils. The architecture is complemented by a formal model for determining buffer configuration. We propose a hybrid use of the block and distributed RAM on the FPGA. The design is validated for a 2D grid, 4-point stencil with circular boundaries.

I. INTRODUCTION

The importance of streaming dataflow architecture for achieving high performance on FPGAs for scientific, multimedia and other HPC applications is well recognized. However, a considerable proportion of such models involve working with *stencils*, possibly with boundary conditions requiring access to locations with large *offsets* (or *reaches*), both of which lead to random and redundant accesses from the DRAM. This breaks the continuity of streaming, degrading performance. While there has been work on mitigating this issue, we observed the need of a more generic solution for dealing with arbitrary stencils and boundary conditions. For example, some scientific problems require stencil computations with circular boundary conditions that result in offsets as large as the entire grid-size itself, which may be many million bytes. There are no straightforward solutions that would allow one to maintain continuous streaming in such a scenario. In this paper, we present a novel *smart-caching* (Smache) architecture to enable continuous data streaming between global (i.e. off-chip DRAM) memory and an FPGA.

Our key contributions are: a novel hybrid architecture of *streaming* and *static* buffers, an algorithm to compute the buffer sizes, transparent on-chip double buffering with write-through logic, a proof-of-concept HDL implementation that uses a cost-model to balance the utilization of block-RAM (BRAM) and distributed-RAM (registers) on an FPGA, and a two-layer architecture customization.

The problem of streaming for stencil computations is a well explored topic. Cong et al's work [1] on an architecture for stencil computation acceleration on FPGAs is similar to our work with respect to our proposal of a hybrid use of registers and BRAMs. However, our use of static buffers, the buffer configuration algorithm and transparent on-chip double buffering feature is different. Reference [2] discusses an approach for optimizing streaming on the Maxeler[3] framework, and emphasizes the re-use of data by processing multiple time steps in one pass. This is pertinent work, but orthogonal to our proposal for optimizing off-chip access. The work presented in [4] similarly discusses optimizing date re-use by doing multiple time-steps in one pass. Sano et al. [5] propose an approach for optimizing memory access for high-performance computation in a multi-device scenario, using soft processors. Reference [6] presents an HDL template for 2D stencil code applications, that allows streaming. Reference [7] discusses this problem in the context of 3D stencils whose dimensions follows some constraints. Reference [8] propose the use of on-chip RAM for optimizing MPEG-4 applications. Reference [9] present a polyhedral model-based framework for iterative stencil loops. The use of OpenCL based HLS approach for stencil computation was discussed in [10]. Most of this work is complementary to ours, especially those dealing with multiple devices, or those that use polyhedral transformations or other approaches to group multiple *work-instance*[1] iterations (e.g. time loops of a scientific model) to maximize data re-use. Our proposal of using a combination of static and streaming buffers for arbitrary stencils shapes and boundary conditions is—as best as we know—a novel feature of our work.

II. A FORMAL MODEL FOR STREAM AND STATIC BUFFERING

A key novel contribution of our Smache architecture is its combination of *stream* and *static* buffers which together provide a general solution of arbitrary offsets required to create stencils. We have developed an algorithm for determining the configuration of these buffers for the general case. Given the following: a vector m of size N representing the content of the off-chip DRAM memory attached to an FPGA; and two iteration patterns p_i and p_o, each pattern is in general an ordered subset of a permutation of the sequence *0 .. N-1*, usually this would be a regular pattern such as contiguous or strided access, but for this discussion it can be more general; we define the streams $s_{\{i,o\}}$ as accesses to the array m:

$$\forall i \in [0, \#p_{\{i,o\}} - 1] \mid s_{\{i,o\}}[i] = m[\, p_{\{i,o\}}(i)\,]$$

For any given program performing computations on this stream, we can divide the computations based on the element of the stream on which they act. In practice this means splitting

---

[1]From OpenCL terminology; refers to one iteration over index space.

the computation per loop nest. Furthermore, given that we are working with stencils, usually a computation will not act on a single element only, but on a small subset of elements at a known offset from the given element. We call this subset the *stream tuple* or tuple for short.

We define two terms to quantify the memory access pattern for a computation: The ***range*** is the number of elements of the stream that participate in a given computation. We will also use the term to refer to the part of the stream comprising these elements. The ***reach*** is the difference between the largest and smallest offset from a given element in the stream to the elements in the stream tuple. For example assume an element of the stream that refers to $m[i]$, and its tuple $(m[i], m[i - 1], m[i + 1], m[i - k], m[i + k])$. Then the largest offset is k, the smallest $-k$, and the reach is $2k$.

For efficient performance we want to limit access to the DRAM and perform the necessary accesses as efficiently as possible. For that purpose we buffer elements from m in memory on the FPGA chip. This memory is very fast but it is a limited resource. We can buffer elements in two ways:

Stream buffering: For a given subset elements required for a given computation (a range of tuples), we can determine the reach of the tuple and create a circular window buffer (stream buffer) of the size of the reach. Then the tuple needed for the computation on each element of the stream is present in this buffer. Referring to the above example, for a given $s[i]$ the stream buffer would buffer all elements from $s[i - k]$ to $s[i + k]$.

Static buffering: For a given subset of elements required for a given computation, we can determine the range and create a static buffer of the size of this range to store an element from the tuple for all tuples in the range. For example, given the tuple above and the range above, the static buffer would be of size $N/m/4$ and could contain the elements $m[4.i + k], \forall i \in [N/m, 2N/m]$. A typical use case is that of circular boundary conditions in a 2D or 3D stencil-type computation: we can statically buffer the boundary values rather than have them in the stream.

The purpose of these two types of buffers is to make the best possible use of the FPGA internal memory: ideally we want to create a stream buffer for the complete stream. However, for some ranges of the stream the reach might be so large that this buffer can't fit in the available memory. We can reduce the reach for a range by storing part of a tuple in a static buffer, as long as the sum of sizes of all static buffers and the stream buffer fits in the on-chip memory. Note that we only ever need a single stream buffer, the one with the largest reach, because all other stream buffers will fit into this one.

We can formalise this approach as follows: Given the streams s_i and s_o, divided into k non-overlapping ranges r_j of size R_j; for each range we have a tuple t_j of n_j elements with a reach $max(t_j) - min(t_j)$. We can compute the cost in terms of stream buffer and static buffers using the algorithm listed in Algorithm 1.

III. SMACHE: A NOVEL HARDWARE ARCHITECTURE FOR STATIC AND STREAMING BUFFERS

We have designed a proof-of-concept HDL design that is generic and can work with arbitrary offsets, as well as different kinds of boundary conditions. To illustrate the architecture, we

---

Algorithm 1 Optimal buffer size calculation

```
for j in 0 .. k
    stream_j, static_j = calc_opt_sz(j)
end
tot = max_∀j(stream) + sum_∀j(static)
def calc_opt_sz(j)
    for i in 0 .. n_j − 1
        stream_i = max(t_{j,0} .. t_{j,i}) − min(t_{j,0} .. t_{j,i})
        static_i = i.R_j
        total_i = stream_i + static_i
    end
    i_opt = 0;  tot_opt = MAX_SZ
    for i in 0 .. n_j − 1
        if tot_i < tot_opt
            tot_opt = tot_i,  i_opt = i
        end
    end
end
```

will use a contrived example, and then discuss how the architecture can be generalized for arbitrary stencils boundaries. Our example is an 11×11 2D grid requiring a 4-point stencil, as shown in Figure 1(a). Circular boundaries at the horizontal edges, and open boundaries at the vertical edges, lead to a total of nine different stencil cases (4 corners, 4 edges, 1 non-boundary). This small example already captures a variety of non-trivial, irregular and asymmetric stencil cases, which is where we see our approach as making a contribution to the state of the art. Our objective is to maintain *continuous* and *contiguous* streaming from the DRAM. Stalling the stream from DRAM, or reverting to random accesses, affects the sustained memory bandwidth considerably, as our own work has also shown[11]. For small *reaches*, a conventional window or *stream* buffer would serve the purpose. This approach will not scale however, e.g. for circular boundaries reaching across the grid with potentially millions of elements. For such cases, the *static* buffers store a *fixed* set of elements rather than a moving window.

Figure 1(b) is a block diagram of our architecture showing the realization of the smart-cache technique for any grid that requires two static buffers, which would make this suitable for most 2D grids with circular boundaries. The data is read into the Smache module (large dotted rectangle), along with the index, the *work-instance*, and a *stall* signal to allow integration with e.g. the AXI4-Stream protocol. The Smache module is shown connected to a computation kernel, and the output from the kernel is fed back into the Smache module to update the static buffers for the next work-instance.

Static Buffers: A circular boundary condition leads to a *reach* spanning the entire array. For even modest-sized scientific models, storing entire arrays on-chip is simply not an option. Static buffers, our approach for storing stencil elements with large reaches was introduced in section II. They are stored on-chip, but unlike stream buffers, they don't hold the entire window of data from the current to the stencil index, but only the elements required to create the stencil. The memory required for static buffers is thus *independent* of the reach, which lets us work with arbitrarily large stencil reaches.

With reference to Figure 1(a), let's assume for the purpose of this example that we are unable to store the entire array on-chip due to resource constraints. So the stencil for elements in the top and bottom rows requires data that cannot be stored by the stream buffer. We cache these top and bottom rows into *static buffers* where they are maintained using a write-
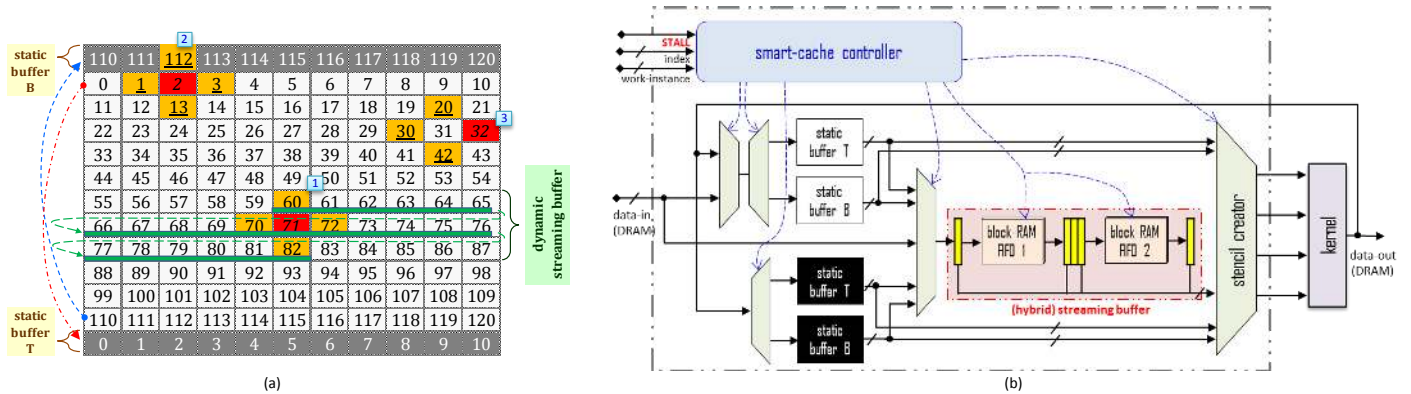
Fig. 1. *(a)* An 11×11 2D grid for a problem requiring a 4-point stencil (e.g. case 1), with circular horizontal boundaries(e.g. case 2), and open vertical boundaries (e.g. case 3). *(b)* Block diagram of the *smache* architecture that implements the static and streaming buffers (for a grid like one shown on the left), with double buffering of static buffers, and hubrid use of BRAMs and registers, shown connected to a computational kernel.

through policy. Figure 1(b) shows these static buffers. The two buffers T and B store the top and bottom rows of the array respectively. We also transparently incorporate double buffering (white and black buffers in the figure) approach, which are read and written concurrently, and swapped after every work-instance. The static buffers requires an additional *warm-up* work-instance iteration at the beginning. This warm-up cost is amortized over multiple work-instance iterations.

Stream Buffers and Hybrid use of registers and BRAM: Use of a moving-window or *stream* buffer is a well-understood technique for improving performance of stencil computations, and is part of our design. We propose a hybrid use of the two main types of on-chip memories on an FPGA: BRAM and registers. This allows flexibility in resource-constrained situations. We can statically predict which locations in the stream buffer will be accessed to create the stencil, and ensure these are placed in register memory, so that they can be accessed concurrently to create the stencil in a single cycle. The rest of the elements can be kept in one or more inter-leaved BRAMs which are accessed logically as a FIFO, but never require more than one concurrent read access. (Concurrent reads from the BRAMs infers a multi-port BRAM which can lead to synthesis of multiple identical BRAMs.) The relative utilization of registers vs BRAMs in the hybrid design varies between the extremes of using registers only for the entire stream buffer (Case-R), to using registers for those locations only that create the stencil, and BRAMs for all the rest (Case-H), which is the case shown in Figure 1(b).

Smache Controller: The Smache controller orchestrates the data movement across the buffers and creates the stencil tuple for the kernel. It is implemented as three concurrent finite statemachines (FSMs). **FSM-1:** pre-fetches data into the static buffers. **FSM-2:** gathers data from the static and streaming buffers, and emits the stencil tuple for the computation kernel. **FSM-3:** reads relevant data from the computation kernel, and updates static buffers.

Generalization of the Architecture: The Smache architecture is generic and compatible with arbitrary stencil shapes. To allow this adaptability, we can configure Smache at two levels. **Number of Static Buffers:** The number of static buffers needed for a particular problem can be determined from a static analysis of the stencil code as discussed in section II.

Configuration of Parameters: Once the number of static buffers is fixed, we can configure a set of parameters to specify any stencil shape within the constraints of the number of static buffers previously fixed.

Memory Utilization Cost Model for Design-Space Exploration: The Smache architecture uses scarce on-chip memory resources which are required by the computation kernel and the shell logic as well. For any design-space exploration (DSE) exercise, manual or automated, it is important to have a cost-model that reasonably predicts these resource requirements for Smache. This becomes especially more important in view of the *hybrid* approach we have proposed, that allows one to trade-off BRAM bits against Registers (and vice-versa). Fortunately, the Smache architecture lends itself to a simple memory resource cost-model. The details of the cost-model outside the scope of this paper. We have however shown estimated vs actual costs in section IV.

IV. EXPERIMENTAL RESULTS

Smache vs Baseline: We have implemented a prototype Smache architecture (Figure 1(b)) in Verilog-HDL for a simple 4-point averaging filter. For comparison, we created a baseline HDL design that does not have any stencil buffers. This means that each grid-point requires 4 words to be read from the global memory, which is $4\times$ more than what is required for the Smache architecture. The simulation results are shown in Figure 2. As expected, the Smache-based solution uses considerably less cycles—around 20% of the baseline—for the same computation as it eliminates redundant memory accesses. This is also indicated in the required global memory traffic which is around 40% of what is needed for the baseline case. So even though the Smache architecture synthesizes at a frequency that is lower than that for the baseline, we still get an overall $3\times$ simulated speed-up. The resource utilization of the baseline implementation was: 79 ALMs, 262 registers, and no BRAM bits; the Smache version used 520 ALMs, 1088 registers, and 1.5K BRAM bits, indicating the resource trade-off.

Hybrid Smache vs Register-Only Smache: We also compared two different versions of the Smache architecture to show the utility of the hyrbrization feature of our architecture. Case-R implements the entire streaming buffer in registers.

Case-H uses the hybrid design shown in Figure 1(b) which uses a combination of registers and BRAMs. The results are shown in Table I. The potential for trade-off is starkly demonstrated in a larger grid size of 1 million elements. In this scenario, Case-R consumes $66K$ registers and $131K$ BRAM bits, whereas the Case-H consumes only $1.5K$ Registers $196K$ BRAM bits. This variation in the utilization of the two types of on-chip memory resources can be exploited to meet design constraints.

Accuracy of Cost Model Estimates: We have developed a cost-model to estimate the memory resource utilization for the Smache architecture for a given problem definition and a certain mixture of registers and BRAM for the streaming buffer. We have verified the predicted costs against actual numbers from a full-synthesis for a Stratix-V device. The results are in Table I. As we can see, our predicted cost very closely tracks the actual resource utilization, and hence our cost expressions can easily be incorporated in a larger cost-model for design-space exploration.

V. CONCLUSION AND FUTURE WORK

FPGAs have started playing a role in mainstream HPC and big-data, albeit still a marginal one. In this context, the strength of FPGAs lie in their ability to provide high performance and energy-efficiency for *streaming* computations. However, stencil-based computation in the streaming paradigm, if done naively, would either break down the continuous data stream with repeated random accesses to the memory, or result in redundant data transfers. To optimise the throughput, we require contiguous accesses to DRAM, and re-use of data on-chip. We have proposed a caching architecture that is capable of working with arbitrary stencil shapes and boundary conditions, by using a combination of *static* and *streaming* buffers. The *streaming* buffers are for *nearby* stencil elements, and the *static* buffers stencil elements with very large reaches. We presented a formal model for buffer configuration, and showed how our streaming buffer architecture allows a hybrid use of registers and BRAMs to meet resource constraints. We presented simulation results on a prototype HDL implementation of our architecture, that

Problem		R_{sc}	B_{sc}	R_{sm}	B_{sm}	R_{total}	B_{total}
$11 \times 11_r$	Estimate	0	1408	800	0	800	1408
	Actual	0	1536	928	0	998	1536
$11 \times 11_h$	Estimate	0	1408	352	448	352	1856
	Actual	0	1536	355	512	425	2048
$1024 \times 1024_r$	Estimate	0	131072	65632	0	65632	131072
	Actual	0	131200	65670	0	66857	131200
$1024 \times 1024_h$	Estimate	0	131072	352	65280	352	196352
	Actual	0	131200	362	65536	1549	196736

TABLE I.   THE ESTIMATED VS ACTUAL UTILIZATION OF ON-CHIP MEMORY RESOURCES. HERE R REFERS TO REGISTERS, B TO BRAM, sc TO STATIC BUFFERS, sm TO STREAMING BUFFER.

clearly show the reduction of DRAM traffic and the potential for improved performance when our architecture is used. Our key future work is to completely automate the creation of the Smache architecture given a problem with a particular stencil shape and boundary conditions. We are also working to integrate our design with a commercial high-level FPGA programming tool, and get real-time results.
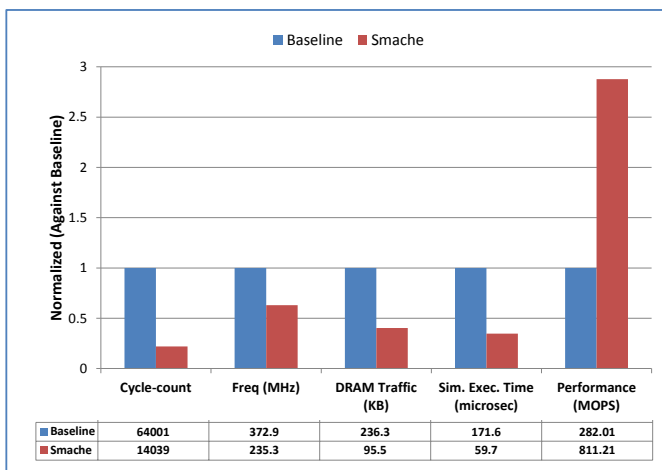

Fig. 2.   Performance comparison of Smache-based 4-point stencil design against baseline design that has no stencil buffering. The array size is 11×11, and the kernel is run 100 times. Cycle count is from simulation, and Frequency is from full synthesis for a Stratix-5 device. The DRAM traffic is calculated from the grid-size and the number of bytes accessed for each grid-point. The simulated execution time and performance in MOPS is calculated from the cycle count and frequency.

REFERENCES

[1] J. Cong, P. Li, B. Xiao, and P. Zhang, "An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[2] H. Fu, R. G. Clapp, O. Mencer, and O. Pell, "Accelerating 3d convolution using streaming architectures on fpgas," in *SEG Technical Program Expanded Abstracts 2009*. Society of Exploration Geophysicists, 2009, pp. 3035–3039.

[3] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Computing in Science Engineering*, vol. 14, no. 4, pp. 98–103, July 2012.

[4] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza, "A high-level synthesis flow for the implementation of iterative stencil loop algorithms on fpga devices," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 52.

[5] K. Sano, Y. Hatsuda, and S. Yamamoto, "Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 234–241.

[6] M. Schmidt, M. Reichenbach, and D. Fey, "A generic vhdl template for 2d stencil code applications on fpgas," in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2012 15th IEEE International Symposium on*. IEEE, 2012, pp. 180–187.

[7] M. Shafiq, M. Pericas, R. De la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguadé, "Exploiting memory customization in fpga for 3d stencil computations," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009, pp. 38–45.

[8] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 241–249, 2011.

[9] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, "A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops," in *2016 IEEE/ACM International Conference on Computer-Aided Design (IC-CAD)*, Nov 2016, pp. 1–8.

[10] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "Opencl-based fpga-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2016.

[11] S. W. Nabi and W. Vanderbauwhede, "Mp-stream: A memory performance benchmark for design space exploration on heterogeneous hpc devices," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 194–197.