Reducing FPGA Memory Footprint of Stencil Codes through Automatic Extraction of Memory Patterns

Robert Szafarczyk
School of Computing Science
University of Glasgow, Glasgow, UK
robert.szafarczyk@glasgow.ac.uk

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

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

***Abstract*—FPGAs are attractive for scientific high-performance computing due to their potential for high performance-per-Watt. Stencil codes in scientific applications are difficult to optimize on FPGAs, because of redundant, non-contiguous memory accesses to relatively low bandwidth DRAM. In this paper, we present an algorithm to aggressively reduce on-chip block RAM (BRAM) and off-chip DRAM utilisation of stencil codes running on FPGAs. The algorithm extracts memory accesses from computational pipelines and removes all redundant intermediate arrays, including those used for stencil buffering, by trading DRAM accesses for computation. The algorithm is based on rewrite-rules on a strict functional representation derived from Fortran code and generates provably correct, optimized code.**

Typical FPGA implementations store the stencil window in on-chip shift registers implemented in BRAMs; we use only DRAM and optimize the memory accesses instead. Our approach dramatically reduces BRAM usage so that the domain size is only limited by available DRAM. We report a drop of 78% and 18% in BRAM usage in 3-D and 2-D stencil codes compared to a manual implementation using shift registers while staying competitive in performance or even improving performance-per-Watt.

***Index Terms*—stencils, compiler rewrite rules, dataflow, HLS**

I. Introduction

High-Level Synthesis (HLS) has made FPGAs more accessible for software developers, making them a viable platform for the acceleration of High-Performance Computing (HPC) workloads, because of the promise of higher energy efficiency. Most existing HPC climate models (and many new ones) are written in Fortran. Fortran is a highly suitable language for domain scientists to express the underlying physics, and the use of Fortran isn't a problem when targeting supercomputers consisting of CPUs. Porting to GPUs is non-trivial but feasible [15]; porting such code to FPGAs, however, is very challenging. Manually rewriting large code bases in an HLS language is usually not feasible (e.g. the Weather Research and Forecasting Model (WRF) comprises more than a million lines of Fortran code).

We have developed a Fortran-to-OpenCL compiler primarily targeting GPUs and have adapted it to translate compute-intensive legacy Fortran code for acceleration on FPGAs. In our toolchain[1], we represent compute-intensive stencil-based finite difference codes in an intermediate dataflow functional language (TyTraCL), allowing provably correct transformations at the dataflow level, without changing computations.

[1]https://github.com/wimvanderbauwhede/RefactorF4Acc

Stencils are difficult to optimise for FPGAs because the data needed for the stencil reach (stencil window buffer) often don't fit into on-chip memory (BRAM), especially in a 3-D domain. In that case, stencil buffers need to be stored in off-chip memory, making memory accesses costly in terms of performance and energy consumption.

In this paper, we propose a novel compiler-based optimisation which drastically reduces the DRAM and BRAM memory footprint of stencil codes on FPGAs. Our contributions are:

- A rewrite-rules based algorithm, which extracts stencil memory accesses from deep computational pipelines and removes redundant intermediate arrays.
- Evaluation of our approach on two stencil codes, showing a reduction in FPGA resources compared to a hand implementation while staying competitive in performance and supporting larger domains. We also show that our approach provides better performance-per-Watt and lower latency than the original Fortran CPU implementation.

II. Background & Related Work

A. Related Work

HLS compilers from FPGA vendors automatically apply many memory transformations to reduce the number of DRAM accesses and keep the computational pipeline fed with data [4], [19]. The authors in [5], [16] present a set of optimization guidelines for HLS FPGA programming, including memory access transformations. Extracting memory from compute pipelines is mentioned as an important optimization for stencil codes. Mainstream HLS compilers often fail to perform this automatically without explicit annotations from the programmer [8]. Our approach extracts these memory accesses in a fully automatic and provably-correct way.

Others have used the approach of separating computations from the data flow. Halide [11] is a C++ DSL for image processing which lets the programmer specify separately what to compute and how to schedule the required data movement. Although the primary target of Halide programs are GPUs and CPUs, recent work has presented HeteroHalide, which targets FPGAs [9]. The programmer can choose from several backends, including SODA [3], which optimizes stencil programs for FPGAs. Compared to HeteroHalide, our Fortran-to-FPGA approach doesn't require explicit programmer annotations, and

1946-1488/22/$31.00 ©2022 IEEE
DOI 10.1109/FPL57034.2022.00033

the transformation we present in this paper isn't studied by them. SODA can be used in HeteroHalide to perform an automated design space exploration (DSE) to find optimal unroll factors, number of iterations, and shift-register sizes. Our intermediate representation is also amenable to an automated DSE, as we have demonstrated previously [10]. HeteroHalide targets image processing stencils, which usually work on a smaller domain and stencil window compared to stencils in scientific computing.

The DaCe framework implements code transformations at the dataflow level, generating HLS code for FPGAs [2]. DaCe programs are usually written in Python/NumPy, with their data flow between side-effect free functions represented as a flow-based Stateful Dataflow Multigraph (SDFG). This effectively decouples data movement from compute. Their approach to rewriting programs is based on graph transformations. In [6], the authors extend DaCe with StencilFlow – a framework for accelerating stencil computations on FPGAs. StencilFlow automatically generates shift-registers between stencils and applies several vendor-specific optimizations during code generation. Our algorithm removes the need for buffering in-between stencils. We have found that using shift-registers, which consume considerate BRAMs, restricts the supported domain size for stencil codes operating in more than two dimensions.

B. TyTraCL: A Functional Coordination Language

In prior work, we developed a source-to-source compiler toolchain which turns legacy Fortran 77 code into type-safe Fortran 95 [12]. This allows the code to be auto-parallelized to target GPUs [13]. Internally, we use a functional coordination language (TyTraCL), inspired by Haskell, to describe the program at a dataflow level [14]. Effectively, the Fortran code is transformed into a functional language which describes the data flow. The main building blocks of TyTraCL are:

- fixed-size vectors representing Fortran arrays,
- scalarized kernel functions,
- higher-order functions *map* and *fold* to express loops.
- a *stencil* function to express stencil-based accesses (for a point in a vector, it returns its stencil points),
- functions combining and splitting of vectors and stencils.

In this work, we target SYCL as our backend [17]. SYCL abstracts away most of the boilerplate needed in OpenCL, making it an easier target for our source-to-source compiler. For example, SYCL can automatically schedule host-device data transfer by analysing the memory dependency graph, including overlapping data transfer and computation.

III. Memory Reduction

The key algorithmic contribution of this paper is a set of program transformations, implemented as type-driven rewrite rules on the TyTraCL code, which eliminate intermediate arrays from the code. For each output array, we can extract a chain of higher-order functions producing the values in that array. We can then pattern-match on predefined chains of higher-order functions involving stencils, and change the function chains to different, equivalent chains, such that for the final stencil in a stencil chain, the neighbouring points are always recomputed, instead of being stored in buffers.

There are two main cases to consider. The simpler case is when we have a sequence of dependency-free loops that compute values and store the results in intermediate arrays, to be used by the following loops. In such a case, it is easy to see that we can merge the loops into a single loop and remove the intermediate arrays. In TyTraCL code, this is particularly straightforward: a sequence of *map* calls

```
v_1 = map loop_kernel_1 v_0
v_2 = map loop_kernel_2 v_1
v_3 = map loop_kernel_3 v_2
```

can be replaced by a single *map* call on the composed function (`.` is the function composition operator)

```
v_3 = map (loop_kernel_3 . loop_kernel_2
          . loop_kernel_1) v0
```

and thus v_1 and v_2 are eliminated. This is one example of a type-driven rewrite rule.

The much more complex case is when some of the loops operate on the intermediate arrays via stencil access patterns. It is not possible to simply compose the functions because of the intervening *stencil* call: although it may seem that we can rewrite

```
v_1 = map loop_kernel_1 v_0
v_1_s = stencil s_1 v_1
v_2 = map loop_kernel_2 v_1_s
```

as

```
v_2 = map loop_kernel_2
 (stencil s_1 (map loop_kernel_1 v_0))
```

and so eliminate v_1, this is not actually the case because the array of stencil patterns still needs to be created. However, we can solve this by moving the stencil up to the first array. We do this by introducing a new operation, *maps*, with its corresponding rewrite rule:

```
stencil s (map f) = map (maps f) (stencil s)
```

With this new operation, we can rewrite the example as

```
v_2 = map loop_kernel_2
 (map (maps loop_kernel_1) (stencil s_1 v_0))
v_2 = map (loop_kernel_2 . (maps loop_kernel_1))
          (stencil s_1 v_0))
```

What *maps* does is to apply f for every point in the stencil s. In this way, we can replace intermediate arrays with additional computations. Because stencil codes are usually memory bandwidth limited, this does not deteriorate the performance, on the contrary, it can even lead to improved performance, as shown in our evaluation.

A further operation *scomb*, with its own rewrite rule, lets us combine stencils:

```
stencil s_2 (stencil s_1 v_1)
 = stencil (scomb s_2 s_1) v_1
```

TABLE I: Resource usage of evaluated FPGA kernels: a manual implementation, direct translation, and our reduced approach.

Code	Approach	BRAMs	ALMs	REGs	DSPs	Frequency (MHz)	Initiation interval	Max. domain size	DRAM usage relative to direct	Domain size bottleneck
velfg	manual	1,834	197,674	498,219	216	182	1	$100 \times 100 \times 90$	$0.25\times$	BRAM
	direct	821	158,256	220,571	217	204	1	$900 \times 900 \times 90$	$1\times$	DRAM
	reduced	467	35,803	89,890	194	254	1	$1,900 \times 1,900 \times 90$	$0.25\times$	DRAM
sw2d	manual	516	5,015	56,140	63	249	1	$15,000 \times 15,000$	$0.73\times$	DRAM
	direct	587	59,351	95,751	76	237	1	$12,500 \times 12,500$	$1\times$	DRAM
	reduced	449	29,971	103,945	258	251	1	$15,000 \times 15,000$	$0.73\times$	DRAM

There are several more rules to deal with grouping and ungrouping of vectors etc., but the above rules are the key ones. For the sake of brevity, we gloss over reductions (*fold*), as they are simpler to handle than *map*s because a reduction results conceptually in a scalar. In general, the final program consists of a number of folds and a final map on composed functions on composed stencils on groups of input vectors:

```
r_1 = fold comp_kernel_1 comp_stencil_inputs_1
r_2 = ...
outputs = map (comp_kernel_2 r_1 r_2 ...)
 comp_stencil_inputs
```

The composed kernel functions only perform scalar operations and therefore all intermediate arrays are eliminated.

IV. Evaluation

A. Methodology

Our approach is specific to automatic acceleration of legacy scientific Fortran codes, targeting programs where multiple heterogeneous stencils are chained together – a common pattern in climate models. It is the shift-register (delay buffers), commonly found in-between the stencil applications on FPGAs, that our rewrite rules aim to eliminate. We evaluated our approach on two stencil codes found in mainstream physics simulators written in Fortran:

- *velfg*: a 3D force calculation kernel from the Large Eddy Simulator (LES) for Urban Flows, a hurricane simulator [18]. The program consists of 3 stencil applications, and a combined 138 single precision operations per grid point.
- *sw2d*: a 2-D shallow water model (wave simulator) from [7]. The program consists of 4 stencil applications, and a combined 90 single precision operations per grid point.

Prior work allows us to make the code type safe, and extract computational kernels in the form of side-effect free functions [12]. From this point, we applied our rewrite rules to remove intermediate arrays. We evaluate three kernel types on an Intel Arria 10 GX 1150 FPGA board (8 GB DRAM, 64 MB on-chip capacity):

- *direct*: a direct syntactic translation from Fortran to SYCL without applying rewrite rules. Basic FPGA idioms have been applied to ensure an initiation interval of 1. Intermediate arrays are stored in DRAM.
- *reduced*: a compiler based translation from Fortran to SYCL using our memory reduction rewrite rules. Intermediate arrays are replaced by additional computation.
- *manual*: a manually implemented kernel using best practices [5], [16]: separate kernels for memory accesses and stencil applications, one streaming memory access per array, pipes kernel communication, shift-registers for the stencil windows in-between stencil applications.

B. Resource usage

The main goal of using our rewrite rules is to reduce the usage of FPGA memory resources, most importantly BRAMs. We demonstrate the results in table I.

velfg BRAM: We report a 78% reduction in BRAMs when comparing our *reduced* approach to the *manual* implementation. The *manual* implementation uses a prohibitively large amount of BRAMs to store the stencil reach in on-chip memory. There are 21 such shift-registers in total, which quickly become a size bottleneck in a 3D domain; the domain size $100 \times 100 \times 90$ cannot be increased. The BRAM consumption in the *reduced* approach doesn't scale with the domain size.

velfg DRAM: Our *reduced* approach automatically removes all 18 redundant intermediate arrays in DRAM. The same arrays were also removed in the *manual* approach. The intermediate arrays found in the original Fortran code were not removed in the *direct* translation to SYCL. Thus, the *direct* approach supports a smaller domain size of $900 \times 900 \times 90$ compared to $1,900 \times 1,900 \times 90$ achieved by our *reduced* approach – a $4\times$ reduction in DRAM usage from using our rewrite rules.

sw2d BRAM: Our *reduced* approach uses 24% fewer BRAMs than a *direct* Fortran to SYCL port. There were only 3 redundant arrays to remove in the *sw2d* code (compared to 18 in *velfg*), making the reduction in BRAMs smaller. There is a 18% reduction in BRAM usage when comparing our *reduced* approach to the *manual* implementation (for the same $15,000 \times 15,000$ domain). There are 12 shift-registers in total in the *manual sw2d* implementation, and the stencil reaches in a 2-D domain don't grow as fast (DRAM becomes the bottleneck before BRAM). We conclude that our approach doesn't offer big resource savings for 2-D stencil codes where the domain size is limited by DRAM.

sw2d DRAM: In the *sw2d* code, there were only 3 redundant intermediate arrays to remove by our *reduced* approach. This still increased the supported domain size from $12,500 \times 12,500$ to $15,000 \times 15,000$, compared to a *direct* FPGA port. The *manual* implementation equally supports a domain size of $15,000 \times 15,000$.

C. Performance

Figure 1a shows that our *reduced* approach is competitive in performance with the *manual* implementation, while reducing BRAM utilization and supporting larger domain sizes. We measured throughput of all approaches in $points/ms$, where $point$ is a single 32-bit float in the 2-D or 3-D domain.

velfg: Our *reduced* approach has the same performance as the *manual* implementation (1% difference). A *direct* port from Fortran to SYCL has poor performance on the FPGA - $9.7\times$ worse than the *reduced* and *manual* approach.

sw2d: The *manual* implementation has $1.14\times$ better performance than our *reduced* approach. A *direct* port has again very poor performance on the FPGA. The *reduced* approach performs worse than the *manual* implementation because the *sw2d* has a significantly lower compute intensity – DRAM access has a bigger impact on performance.

D. Power efficiency

We verify that our *reduced* approach achieves better performance-per-Watt compared to the original Fortran code running on a single CPU[2].

Methodology: We ran all versions of the *velfg* code for 1000 iterations (the runtime was at least 15 minutes in all cases). We measured the whole FPGA board power (including DRAM) using on-board power sensors accessed through the 'fpgainfo' tool available in the Intel OPAE FPGA driver stack [1]. For the CPU measurement, we used Intel's Running Average Power Limit (RAPL) technology measuring only the core power consumption (excluding caches and DRAM), and subtracting the baseline power consumed at idle. Thus, the performance-per-Watt for a single core will be better than for all cores, even when assuming linear multi-core speedup.

Results: In figure 1b we compare the energy efficiency of all FPGA approaches to the original *fortran* code running on a single CPU[2]. In the *velfg* code, our *reduced* approach on FPGA is $8\times$ more power efficient than *fortran* on CPU, and $1.28\times$ more efficient than the *manual* implementation on FPGA. Finally, a *direct* Fortran to SYCL port on FPGA has worse power efficiency than the original CPU code.

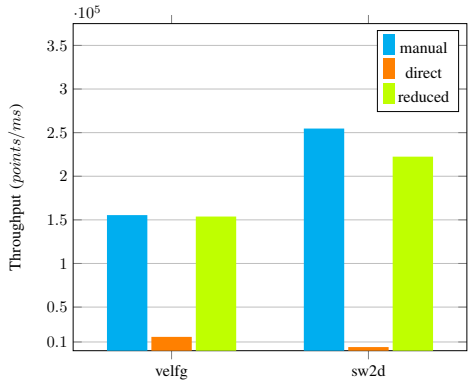
In the *sw2d* code, our *reduced* approach on FPGA is $3.8\times$ more power efficient than *fortran* on CPU. The *manual* implementation has the best performance-per-Watt; $1.16\times$ better than our *reduced* approach. A *direct* Fortran to FPGA port has much worse power efficiency than the CPU code.

We conclude that our memory reduction approach is more beneficial to stencil codes operating in a 3-D (or higher) domain, where one can scale the supported domain size much more once BRAM buffers are removed. For smaller stencils, the rewrite rules don't give any measurable benefit beyond a small domain size increase, as demonstrated by the *sw2d* exemplar.

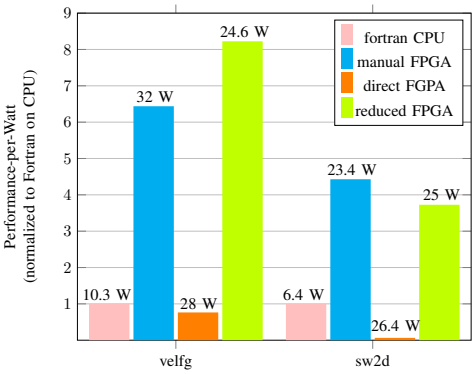
V. Conclusion

We have presented an algorithm to remove intermediate arrays from stencil codes. We have incorporated this algorithm

[2]Intel® Xeon® Platinum 8260, with GNU Fortran (GCC) 4.8.5 with -Ofast



(a) Performance of the FPGA approaches (number of processed domain points per ms). Higher is better.



(b) Energy efficiency of the FPGA approaches compared to the original Fortran CPU code. Measured in thousands-of-points per Watt, and normalized to Fortran. The throughput of the base (Fortran) is 6947 $points/ms$ for velfg, and 7789 $points/ms$ for sw2d. Average Wattage is annotated for each approach.

Fig. 1: Throughput and performance-per-Watt of a manual FPGA implementation, a direct Fortran-to-SYCL translation, and our memory reduced approach.

in our compilation toolchain from legacy Fortran code to SYCL, with the effect of significantly reducing FPGA memory resource usage. Our compiler expresses dataflow, parallelism and stencil accesses in a functional, domain-specific language (TyTraCL). We use a set of formal rewrite rules on the primitives of the language to rewrite the program. The algorithm extracts stencil point accesses from deep computational pipelines and moves them to the start of the pipeline, an exchanges the intermediate arrays used for stencil updates for additional computation. From the transformed TyTraCL program we can generate Fortran, OpenCL or SYCL.

Acknowledgements

This work was partly supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/L00058X/1. We thank Intel for access to FPGAs through Intel DevCloud and The Edinburgh Parallel Computing Centre for access to their FPGA cluster.

REFERENCES

[1] Open Programmable Acceleration Engine Intel. https://opae.github.io/. Accessed: 2022-02-14.

[2] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. Soda: Stencil with optimized dataflow architecture. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18. Association for Computing Machinery, 2018.

[4] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. From opencl to high-performance hardware on fpgas. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534, 2012.

[5] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1014–1029, 2021.

[6] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. *StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems*, page 315–326. IEEE Press, 2021.

[7] Jochen Kämpf. *Ocean modelling for beginners: using open-source software*. Springer Science & Business Media, 2009.

[8] Tobias Kenter, Gopinath Mahale, Samer Alhaddad, Yevgen Grynko, Christian Schmitt, Ayesha Afzal, Frank Hannig, Jens Förstner, and Christian Plessl. Opencl-based fpga design to accelerate the nodal discontinuous galerkin method for unstructured meshes. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 189–196, 2018.

[9] Jiajie Li, Yuze Chi, and Jason Cong. Heterohalide: From image processing dsl to efficient fpga acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 51–57. Association for Computing Machinery, 2020.

[10] Syed Waqar Nabi and Wim Vanderbauwhede. Automatic Pipelining and Vectorization of Scientific Code for FPGAs. *International Journal of Reconfigurable Computing*, 2019:7348013, November 2019. Publisher: Hindawi.

[11] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, jun 2013.

[12] Wim Vanderbauwhede. Making legacy Fortran code type safe through automated program transformation. *The Journal of Supercomputing*, 78(2):2988–3028, February 2022.

[13] Wim Vanderbauwhede and Gavin Davidson. Domain-specific acceleration and auto-parallelization of legacy scientific code in FORTRAN 77 using source-to-source compilation. *Computers & Fluids*, 173:1–5, September 2018.

[14] Wim Vanderbauwhede, Syed Waqar Nabi, and Cristian Urlea. Type-Driven Automated Program Transformations and Cost Modelling for Optimising Streaming Programs on FPGAs. *International Journal of Parallel Programming*, 47(1):114–136, February 2019.

[15] Wim Vanderbauwhede and Tetsuya Takemi. An analysis of the feasibility and benefits of gpu/multicore acceleration of the weather research and forecasting model. *Concurrency and Computation: Practice and Experience*, 28(7):2052–2072, 2016.

[16] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. Energy efficient scientific computing on fpgas using opencl. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 247–256, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Michael Wong, Nevin Liber, Sanzio Bassini, Andrew Richards, Mark Butler, Jeff McVeigh, Brandon Cook, Hideki Sugimoto, Cyril Cordoba, Thomas Fahringer, and et al. Sycl - c++ single-source heterogeneous programming for acceleration offload. https://www.khronos.org/sycl/, Jan 2014.

[18] Toshiya Yoshida, Tetsuya Takemi, and Mitsuaki Horiguchi. Large-eddy-simulation study of the effects of building-height variability on turbulent flows over an actual urban area. *Boundary-Layer Meteorology*, 168(1):127–153, 2018.

[19] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, and Satoshi Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, volume 2016. IEEE, November 2016.