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

Abstract—**FPGAs are attractive for scientific high-performance computing due to their potential for high performance per Watt. However, stencil codes in scientific applications are difficult to optimize on FPGAs, because of redundant, non-contiguous memory accesses to relatively low bandwidth DRAM. In the numerical weather prediction and climate modelling domain, most applications are written in legacy Fortran, making the use of FPGAs challenging. Our research focuses on the compilation of legacy numerical simulation codes to accelerators such as GPUs and FPGAs.**

In this paper, we present a novel 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 formal rewrite rules on a strict functional representation derived from the original Fortran code and generates provably correct, optimized code.

Best HLS practices store the stencil window in on-chip shift registers implemented in BRAMs; we use only DRAM and optimize the memory accesses instead. 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. In 3-D stencil codes using shift registers on FPGAs, the bottleneck for the problem size is often the BRAM; our approach dramatically reduces BRAM usage so that it is only limited by available DRAM. Compared to a direct translation from Fortran, our algorithm can reduce DRAM usage by $4.5\times$ and $1.4\times$ in our exemplars, allowing to increase the domain size from $900\times900\times90$ to $1900\times1900\times90$ in the tested 3-D stencil code. The performance per Watt improvement over CPU execution is on average $6\times$, demonstrating the feasibility of creating high-performance FPGA code by automatic compilation of legacy scientific Fortran codes.

Index Terms—**stencil code, compiler rewrite rules, dataflow, FPGA HLS**

I. Introduction

High Level Synthesis (HLS) has made FPGAs more accessible for software developers. FPGAs are an attractice platform for acceleration of High Performance Computing (HPC) workloads, especially climate model simulations, because of the promise of higher energy efficiency.

Most existing climate models (and many new ones) are written in Fortran. Fortran is a highly suitable language for domain scientist 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 [1]; porting such code to FPGAs is very challenging. Manually rewriting large code bases in a HLS language is usually not feasible. For example 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 compiler toolchain[1], we represent compute intensive stencil-based finite difference codes in an intermediate dataflow functional language (Anonymous), allowing rapid and provably correct code transformations at the dataflow level, without changing actual computations.

Stencils are difficult to optimise for FPGAs because the data needed for the stencil reach (stencil window buffer) often doesn'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.

The key contributions of this work are:

- An algorithm based on provably-correct rewrite rules which automatically extracts stencil memory accesses from deep computational pipelines, and removes redundant intermediate arrays.
- Evaluation of our approach on two real-world stencil codes, showing a reduction in FPGA resource usage compared to a hand implementation, while staying competitive in performance and supporting larger domain sizes.
- Verification that our Fortran-to-FPGA compiler based 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 [18] [8]. For instance, Intel's HLS compiler synthesizes memory accesses into different Load Store Unit (LSU) versions depending on the detected access pattern - enabling coalescing, pipelining, prefetching, or caching. Our work is complementary to these efforts, in that it makes inferring more optimal LSUs easier.

[1] Anonymous

The authors in [9] and [15] present a large set of optimization guidelines for HLS FPGA programming, including memory access transformations. Extracting memory from computational pipelines is explicitly mentioned in [9] as an important optimization for stencil codes: "By extracting accesses to external memory from the computational logic, we enable compute and memory accesses to be pipelined and optimized separately". Mainstream HLS compilers often fail to perform this automatically, without explicit annotations from the programmer [12]. Our approach extracts these memory accesses in a fully automatic and provably-correct way.

Others have used the approach of separating computations from data flow. Halide [14] is a commercially used C++ DSL for image processing which lets the programmer separately specify what to compute and how to schedule the required data movement. Altough the primary target of Halide programs are GPUs and CPUs, recent work has presented HeteroHalide which targets FPGAs. The DaCe framework, similar to this work, implements code transformations at the dataflow level, generating HLS code for FPGAs [6]. 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, an approach which we have also taken. Their approach to rewriting programs is based on graph transformations and differs from ours.

In [10], the authors present StencilFlow – an End-To-End DaCe based framework for accelerating stencil computations on FPGAs. StencilFlow automatically generates shift-registers between stencils connected by OpenCL channels, and applies several vendor specific optimizations during code generation. Our algorithm extracts stencil memory accesses from computational pipelines, instead of using buffering to increase locality of on-chip memory. 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, especially when using a smaller FPGA. DaCe offers a productive framework to write new, FPGA accelerated code in several front-ends like scientific Python; our goal is to enable automatic acceleration of legacy scientific Fortran code bases.

B. Anonymous: 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 [2]. This allows the code to be auto-parallelized to target GPUs [3]. In our toolchain, we use a functional coordination language (Anonymous) inspired by Haskell to describe the program at a dataflow level [4]. Effectively, the Fortran code is transformed into a pure functional language which describes the data flow and parallelism. The main building blocks of Anonymous are:

- Fixed-size vectors representing Fortran arrays,
- scalarized kernel functions (ELEMENTAL and PURE in Fortran terms),
- higher-order functions *map* and *fold* to express parallel loops.
- a *stencil* function to express stencil-based memory access patterns,
- functions for combining and splitting of vectors and stencils.

For illustration, a part of the Anonymous code for the *velfg* example used in the paper:

```
-- velfg_map_133
s4 = [23409,46665]
v_s_1 = stencil s4 v_0
s5 = [46665,46816,46817]
w_s_1 = stencil s5 w_0
s6 = [23560,46816]
u_s_1 = stencil s6 u_0
(cov7_1,cov8_1,diu7_1,diu8_1,nou7_1,nou8_1)
 = unzipt (map (velfg_map_133 (dzn_0,dx1_0,dy1_0)))
 (zipt (u_s_1,w_s_1,v_s_1))
```

This code describes a nested loop with different stencil accesses on three arrays. Because this representation is pure and functional, we can use it for provably correct program transformations.

C. SYCL

In this work we make use of the SYCL [16] C++ abstraction layer over OpenCL – a standard for heterogeneous programming based on C++ abstractions developed by the Khronos Group. In many respects, SYCL is similar to OpenCL: it includes command queues, Single Work-Item (SWI) and NDRange kernel invocations, pipes (channels), and other OpenCL constructs. The advantage of SYCL is the many quality of life improvements for the programmer (e.g. less boilerplate code, more powerful abstractions thanks to C++ meta-programming). At the same time, it is fully compatible with existing OpenCL code. Some features help the compiler produce better code, e.g. memory accessors to inform the compiler about the direction of memory accesses (similar to *intent* in Fortran), finer grained parallelism control or built-in parallel operations like reductions.

There are SYCL implementations from the two major FPGA manufacturers: DPC++ from Intel and triSYCL from Xilinx. We use Intel's DPC++ compiler in this work. When compiling for FPGAs, DPC++ generates LLVM IR, translates it to SPIR-V, and uses the OpenCL backend. The final bitstream is generated using aoc from the Intel Quartus tools.

III. Memory Reduction

The key algorithmic contribution of this paper is a set of program transformations, implemented as type-driven rewrite rules on the Anonymous code, which eliminate intermediate arrays from the code. There are two main cases. 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 Anonymous 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))
```

and finally as

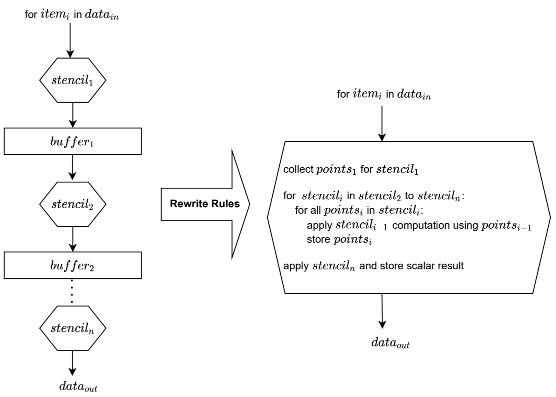
```
v_2 = map (
 loop_kernel_2 .
 (maps loop_kernel_1))
 (stencil s_1 v_0))
```

What *maps* does is to apply f for every point in the stencil s. In this way, we can replace intermediate arrays by 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

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

lets us combine stencils. 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 in fact 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:



for $item_i$ in $data_{in}$
$stencil_1$
$buffer_1$
$stencil_2$
$buffer_2$
$stencil_n$
$data_{out}$
Rewrite Rules
for $item_i$ in $data_{in}$
collect $points_1$ for $stencil_1$
for $stencil_i$ in $stencil_2$ to $stencil_n$:
for all $points_i$ in $stencil_i$:
apply $stencil_{i-1}$ computation using $points_{i-1}$
store $points_i$
apply $stencil_n$ and store scalar result
$data_{out}$

Fig. 1. Conceptual view of the memory reduction rules' effect. Hexagons represent kernels, rectangles represent memory. On the left is a traditional stencil chain implementation, with intermediate buffers in DRAM or FPGA shift-registers. On the right is the transformed code: multiple kernels fused into one, for each $item_i$ in $data_{out}$, every point for every $stencil_i$ is computed.

```
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, making the use of benchmark suits, such as Rodinia, [7] difficult. Our algorithm targets 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. The Rodinia structured grid codes have just one stencil application and no intermediate stencil buffers. Thus, we evaluate 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 [17]. The program consists of 3 stencil applications, and a combined 138 single precision operations per grid point.
- *sw2d*: a 2D shallow water model (wave simulator) from [11]. 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 [2]. From this point, we applied our rewrite rules to remove intermediate arrays between stencil applications. We evaluate 3 SYCL kernels:

- *direct*: a direct syntactic translation from Fortran to SYCL C++ 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 C++ using our memory reduction rewrite rules. Intermediate arrays are replaced by additional computation.
- *manual*: a manually implemented kernel using best practices [9], [15]: separate kernels for memory accesses and stencil applications, one streaming memory access per array, SYCL pipes for communication between kernels, shift-registers (delay buffers) for the stencil reaches, non stencil arrays stored in private memory, and systolic array pattern applied to reduce fanout. Intermediate arrays are stored in BRAM.

We used an Intel Arria 10 GX 1150 FPGA board with 8 GB of dual channel DDR4 memory (maximum theoretical memory bandwidth of 34 GB/s), and 2713 MK20 RAM blocks (65.7 Mb on-chip capacity). The SYCL compiler version was dpcpp 2022.0.0. We measured kernel execution time using SYCL events, ignoring the memory transfer times. In real simulations, the kernel would be run for thousands of iterations, making the memory transfer overhead negligible.

B. Resource usage

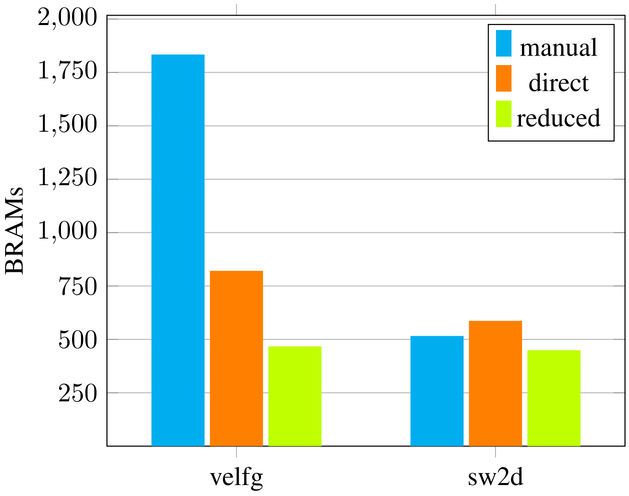


Fig. 2. Block RAM (BRAM) consumption of a *manual* implementation (with pipes and shift registers), a *direct* Fortran to SYCL translation, and a translation using our memory *reduced* approach. Lower is better. BRAM usage at the maximum supported domain for each approach (see table I). The velfg stencil code is left, the sw2d is right.

The main goal of using our rewrite rules is to reduce the usage of FPGA memory resources, most importantly BRAMs. We demonstrate this in figure 2; table I shows a more detailed resource usage report. The reported numbers are for the maximum supported domain of each approach, and do not include the resources used to implement the OpenCL BSP (this number is the same for all kernel versions).

velfg BRAM: We report a 78% reduction in BRAMs when comparing our *reduced* approach $(1,900 \times 1,900 \times 90$ domain) to the *manual* implementation ($100 \times 100 \times 90$ domain). Our *reduced* approach consumes the same amount of BRAM for all domain sizes. The *manual* implementation uses a prohibitively large amount of BRAMs to store the stencil reach in on-chip memory. Following best practices, the array values are streamed from memory on SYCL pipes and stored in shift-registers between two stencil applications. There are 21 such shift-registers in total. Because the stencil reach grows fast in the 3D *velfg* domain, this is a bottleneck; the domain size $100 \times 100 \times 90$ cannot be increased without running out of BRAMs. Our *reduced* approach $(1,900 \times 1,900 \times 90$ domain) also consumes 44% fewer BRAMs than a *direct* Fortran to SYCL port ($900 \times 900 \times 90$ domain). This is a direct consequence of the rewrite rules. In both approaches, most BRAMs are used to implement Load Store Units (LSUs) to DRAM (no BRAMs are used for user defined on-chip buffers). In practice, our rewrite rules hoist all DRAM stencil accesses from computational pipelines to the beginning of the kernel. Next, all stencil loads from the same array are merged in a single loop nest. This helps the SYCL compiler to infer a sequential memory accesses, where possible, resulting in fewer but wider LSUs. The removal of redundant intermediate DRAM arrays (and the corresponding accesses) further decreases the LSU number.

velfg DRAM: Our *reduced* approach automatically removes all 18 redundant intermediate arrays in DRAM. The same arrays were also removed manually in the *manual* approach, thus the *reduced* and *manual* approach have the same DRAM utilization. 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 $4.5\times$ reduction in DRAM usage per domain item from using our rewrite rules. The *manual* approach is bottlenecked by BRAM and supports only a domain of $100 \times 100 \times 90$.

sw2d BRAM: Our *reduced* approach uses 24% fewer BRAMs than a *direct* Fortran to SYCL port. The reasons for this are the same as for the *velfg* code: fewer but wider LSUs, and removed LSUs that would otherwise be used to access redundant arrays. 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 the grow much slower ($O(n)$ compared to $O(n^2)$ in *velfg*). This means that fewer BRAMs are needed to implement the shift registers in the *manual* approach. We conclude that our approach doesn't offer big resource savings in this 2D stencil code when the domain size is limited by DRAM. If the *sw2d* 2D stencil code would become bottlenecked by BRAM again (e.g. by using an FPGA board with 32 GB of DRAM, or using a smaller FPGA with fewer BRAMs), our *reduced* approach would again support a larger domain size than the *manual* implementation. We still have the advantage that our *reduced*

TABLE I

RESOURCE USAGE OF EVALUATED FPGA KERNELS: A MANUAL IMPLEMENTATION, DIRECT TRANSLATION FROM FORTRAN TO SYCL, TRANSLATION TO SYCL USING OUR MEMORY REDUCTION APPROACH. USAGE SHOWN FOR THE LARGEST SUPPORTED DOMAIN SIZE.

Code	Approach	BRAMs	ALMs	REGs	DSPs	Frequency (MHz)	Initiation interval	Max. domain size	DRAM usage (B) per domain point	Domain size bottleneck
velfg	manual	1,834	197,674	498,219	216	182	1	$100 \times 100 \times 90$	24	BRAM
	direct	821	158,256	220,571	217	204	1	$900 \times 900 \times 90$	108	DRAM
	reduced	467	35,803	89,890	194	254	1	$1,900 \times 1,900 \times 90$	24	DRAM
sw2d	manual	516	5,015	56,140	63	249	1	$15,000 \times 15,000$	32	DRAM
	direct	587	59,351	95,751	76	237	1	$12,500 \times 12,500$	44	DRAM
	reduced	449	29,971	103,945	258	251	1	$15,000 \times 15,000$	32	DRAM

approach is fully compiler-based, as opposed to the *manual* implementation.

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 – a $1.4\times$ reduction in DRAM usage per domain item coming from our rewrite rules. The *manual* implementation is also free of redundant arrays, equally supporting a domain size of $15,000 \times 15,000$.

C. Performance

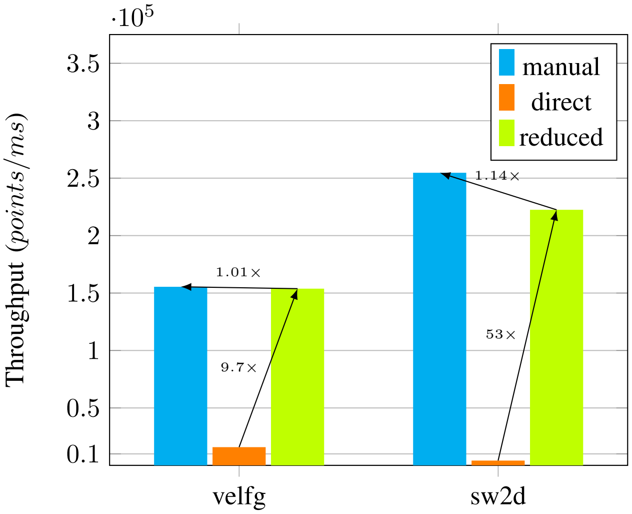


Fig. 3. Performance of a manual implementation, a direct Fortran to SYCL translation, and a translation using our memory reduced approach. Higher is better. Throughput is an average across all supported domains. The arrows show the improvement of using our approach over a direct syntactic translation (orange to green), and the improvement of a fully manual implementation over our approach (green to blue).

Figure 3 shows that our compiler-based *reduced* approach is competitive 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 2D or 3D domain. We tested six different domain sizes and show the average throughput (*velfg* from $100 \times 100 \times 90$ to $1,900 \times 1,900 \times 90$; *sw2d* from 1000×1000 to $15,000 \times 15,000$). The *manual velfg* kernel was tested only on $100 \times 100 \times 90$, because it didn't fit on the FPGA for larger domains.

velfg: Our compiler-based *reduced* approach has practically the same performance as the *manual* implementation (1% difference). We also see that 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 poor performance on the FPGA, with our *reduced* approach showing a $53\times$ improvement. The *reduced* approach performs less well than the *manual* implementation because the *sw2d* has a significantly lower compute intensity, with DRAM access having a bigger impact on performance. While our *reduced* approach has a far better memory access pattern than a *direct* port, it still has more DRAM accesses than the *manual* implementation, which relies on on-chip shift-registers to access the stencil points.

D. Power efficiency

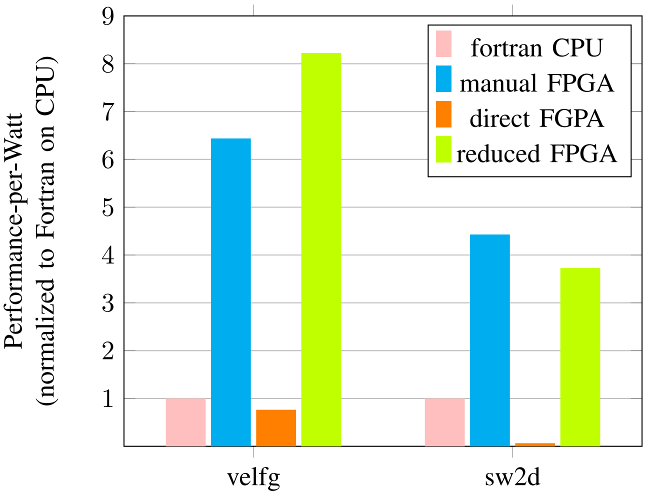


Fig. 4. Efficiency of the original Fortran CPU code, a manual FPGA kernel implementation, a direct Fortran-to-SYCL translation, and our memory reduced translation. Measured in thousands-of-points per Watt, and normalized to fortran. The throughput of the base (fortran) is 6947 $points/ms$ for velfg, and 7789 $points/ms$ for sw2d. See table II for Wattage.

TABLE II
POWER CONSUMPTION OF THE ORIGINAL FORTRAN CODE ON CPU AND THE EVALUATED FPGA KERNELS: A MANUAL IMPLEMENTATION, DIRECT TRANSLATION FROM FORTRAN TO SYCL, TRANSLATION TO SYCL USING OUR MEMORY REDUCTION APPROACH.

Code	Approach	Domain	Iterations	Run Time (s)	Avg. Power (W)	Total Energy (J)
velfg	fortran	$900 \times 900 \times 90$	1,000	6,823	10.26	70,004
	manual	$100 \times 100 \times 90$	100,000	1,133	31.95	36,223
	direct	$900 \times 900 \times 90$	1,000	5,402	28.01	151,293
	reduced	$900 \times 900 \times 90$	1,000	1,222	24.55	30,028
sw2d	fortran	$10,000 \times 10,000$	1,000	9,169	6.39	58,580
	manual	$10,000 \times 10,000$	1,000	1,987	23.36	46,411
	direct	$10,000 \times 10,000$	1,000	25,670	26.36	676,580
	reduced	$10,000 \times 10,000$	1,000	2,030	25.01	50,780

A major incentive behind using FPGAs for HPC codes is the promise of better energy 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 on a $900 \times 900 \times 90$ domain (except the *manual* approach, which used a $100 \times 100 \times 90$ domain); and all versions of the *sw2d* code on a 10000×10000. Both codes ran for 1000 iterations, resulting in a minimum of 15 minutes runtime. Every 0.1 seconds during runtime, 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 [5]. The reported wattage is the average across all measurements during the execution of the kernel. We followed the same approach for CPU measurements, running the code for 1000 iterations with measurements every 0.1 seconds. Intel's Running Average Power Limit (RAPL) technology allows power measurements of the core, uncore and DRAM part of the chip for a single OS process [13]. To be conservative, we measured *only core power* (excluding caches and DRAM), and we subtracted power consumed at idle from the power consumed during kernel runtime. If our power measurements are skewed, they are skewed in favour of the CPU.

Results: Table II shows detailed power consumption results for our experiment. In figure 4 we compare 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. The *manual velfg* implementation uses considerable more FPGA resources than the other two FPGA approaches, resulting in a higher wattage ($31.95W$ compared with $24.55W$ for the *reduced* approach). Finally, a *direct* Fortran to SYCL port 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; it is $1.16\times$ more efficient than our *reduced* approach. Again, a *direct* Fortran to SYCL port has much worse power efficiency than the original CPU code.

V. CONCLUSION

We have presented a novel algorithm to entirely remove intermediate arrays from stencil codes. We have incorporated this algorithm in our compilation toolchain from legacy Fortran code to SYCL, with the effect of significantly reducing FPGA memory resource usage, both off-chip and on-chip, without compromising performance.

Our compiler expresses dataflow, parallelism and stencil accesses in a domain-specific language, Anonymous. 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, thus eliminating the intermediate arrays used for stencil updates. From the transformed Anonymous program we can generate Fortran, OpenCL or SYCL code.

From an FPGA perspective, this means that intermediate stencil buffers, typically BRAM-based and thus a very limited resource, are eliminated.

Our approach is performance competitive with manually optimized FPGA kernels, while allowing to target larger domain sizes. Compared to a direct Fortran-SYCL FPGA syntactic translation, our approach has an order of magnitude better performance. Finally, we have shown that our automatic porting of legacy Fortran to FPGAs results in an at least $6\times$ improvement in power efficiency.

ACKNOWLEDGMENT

REFERENCES

[1] Anonymous.

[2] Anonymous.

[3] Anonymous.

[4] Anonymous.

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

[6] 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.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), mar 2018.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.