Fortran High-Level Synthesis: Reducing the barriers to accelerating HPC codes on FPGAs

Gabriel Rodriguez-Canal [§1], Nick Brown[1], Tim Dykes[2], Jess Jones[2], and Utz-Uwe Haus[2]

[1]EPCC, The University of Edinburgh
[2]HPC/AI EMEA Research Lab, Hewlett Packard Enterprise

Abstract—In recent years the use of FPGAs to accelerate scientific applications has grown, with numerous applications demonstrating the benefit of FPGAs for high performance workloads. However, whilst High Level Synthesis (HLS) has significantly lowered the barrier to entry in programming FPGAs by enabling programmers to use C++, a major challenge is that most often these codes are not originally written in C++. Instead, Fortran is the lingua franca of scientific computing and-so it requires a complex and time consuming initial step to convert into C++ even before considering the FPGA.

In this paper we describe work enabling Fortran for AMD Xilinx FPGAs by connecting the LLVM Flang front end to AMD Xilinx's LLVM back end. This enables programmers to use Fortran as a first-class language for programming FPGAs, and as we demonstrate enjoy all the tuning and optimisation opportunities that HLS C++ provides. Furthermore, we demonstrate that certain language features of Fortran make it especially beneficial for programming FPGAs compared to C++. The result of this work is a lowering of the barrier to entry in using FPGAs for scientific computing, enabling programmers to leverage their existing codebase and language of choice on the FPGA directly.

Index Terms—FPGAs, Fortran, High Level Synthesis, HPC

I. INTRODUCTION

First introduced in the late 1950s, Fortran has been the lingua franca of scientific computing for over 60 years [1]. Whilst many alternative programming languages have come and gone, it has regained its popularity for writing high performance codes. Indeed, over 80% of the applications running on ARCHER2, a 750,000 core Cray EX which is the UK national supercomputer, are written in Fortran. This ubiquity has meant that there are many applications, both actively maintained and legacy, that use the language ranging from weather forecasting models used by the world's leading meteorology organisations [2] [3] to simulation codes used for modelling state of the art aircraft and jet engines [4] [5].

As the HPC community turns to exploring accelerators other than GPUs, for potential performance and energy usage advantages, Field Programmable Gate Arrays (FPGAs) have significant potential. Whilst FPGAs are yet to gain wide acceptance in HPC, there are a number of testbeds and smaller HPC systems that contain FPGAs. Numerous studies report that the ability to tailor the logic and select a bespoke on-chip memory configuration can be beneficial compared to general purpose architectures, especially for those codes which are memory bound [6] [7]. However, a major blocker is that currently, when using High Level Synthesis (HLS), one must write their codes in C or C++. Consequently there is a significant initial overhead for many HPC developers where they must first convert their Fortran code into C++, before getting anywhere near an FPGA. This not only requires expertise in another language, but furthermore is a time consuming and error prone process due to the differences between Fortran and C++, for instance in array index ordering (row major in C verses column major in Fortran), and default array start indices.

This paper describes work bringing Fortran programming to FPGAs by integrating Flang with AMD Xilinx's LLVM-based HLS back end. The objective is to lower the barrier to entry in porting existing HPC codes to FPGAs, and ultimately increase the accessibility of FPGAs to the HPC community. The rest of the paper is structured as follows, Section II describes the background to this work in more detail and surveys previous efforts bringing Fortran to FPGAs. Section III reports the hardware and software configurations used in this work, before Section IV then explores our Fortran HLS solution. A performance comparison of our work against the existing C/C++ HLS approach is undertaken in Section V, highlighting benefits of Fortran for writing HLS code. Section VI then draws conclusions and describes further work.

The contributions of this paper are:

1) Describing an approach to enabling direct support for Fortran on AMD Xilinx FPGAs through the Xilinx Vitis ecosystem.
2) A case study of leveraging AMD Xilinx's open source LLVM front end components to integrate new programming languages and tools into Vitis HLS, without changes needed to the standard Flang front end or Xilinx back end.
3) A comparison of performance between Fortran HLS and C/C++, demonstrating how Fortran specific language constructs, such as dynamic N-dimensional arrays, can provide an advantage over the existing HLS C/C++ flow.

II. BACKGROUND AND RELATED WORK

A common approach to programming FPGAs is to use C or C++ via HLS, with both AMD Xilinx and Intel providing their own HLS solutions. Several tools have been developed [8] [9] which aim to undertake automatic code transformations, often via bespoke MLIR dialects [10], in order to enable non-experts

---

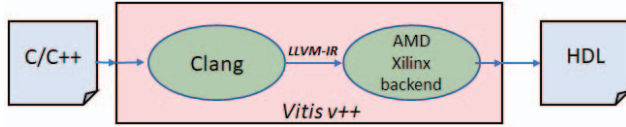§Corresponding author: gabriel.rodcanal@ed.ac.uk

Fig. 1: Default workflow of AMD Xilinx Vitis `v++`

to program FPGAs effectively. There are mixed results when using these tools, and code must be in C or C++ to start with, but it demonstrates the ability of third parties to develop their own transformations and then integrate with existing tools.

DACE [11] is an alternative approach which is more focused on HPC, where developers write code in Python and this is then transpiled to a variety of target architectures including FPGAs (C/C++ HLS for AMD Xilinx and OpenCL for Intel FPGAs). However, the limitation is that one must first undertake conversion of their code to integrate with DACE before running on FPGAs, similar to the disadvantages associated with having to first convert Fortran to C/C++ for HLS.

TyTra [12] is a compiler which synthesises Fortran to HDL and is intended as a platform for undertaking auto-optimisation steps to automatically convert the programmer's Von Neumann-based code into an efficient dataflow representation. Whilst, at the time of writing TyTra is not actively maintained, it is an interesting comparison point. The major disadvantages with TyTra is that one must rely on their own bespoke compiler resulting in risk (whether the compiler will continue to be maintained in the future), potentially poorer future performance (as they will not automatically benefit from the improvements made by vendors to their tool chains) and a lack of integration with the wider ecosystem (such as the Vitis profiling tools).

A. AMD Xilinx's LLVM-based HLS tooling

AMD Xilinx's HLS synthesis tool is based upon LLVM and, as illustrated in Figure 1, this comprises two parts; a front end built upon Clang [13] which compiles a programmer's C/C++ code to LLVM's Intermediate Representation format known as LLVM-IR, and a back end which synthesises LLVM-IR to the underlying HDL level for their FPGAs. Whilst much of Clang in Vitis HLS has remained unchanged, AMD Xilinx have added support for their own bespoke HLS pramas which correspond to additional constructs in the generated LLVM-IR.

AMD Xilinx have released their HLS Clang front end via the Illinois/NCSA open source licence, and this is kept up to date with each release of the Vitis suite, although their back end remains proprietary. Based upon these components it is possible to generate LLVM-IR and provide this to the AMD Xilinx back end for it to synthesise to HDL.

B. LLVM Flang

Flang [14] is the open source Fortran compiler provided as part of the LLVM compiler suite. Based around an active and growing open source community, this compiler was developed from the ground-up around five years ago and replaced the

previous Flang compiler in the main LLVM repository around two years ago. Built around MLIR, a framework in LLVM for defining bespoke IR dialects and mixing these together, Flang currently supports the Fortran 2003 standard with several features from later standards, such as Fortran 2008 and Fortran 2018, also supported and many others in development.

Ultimately, Flang is a front end for the Fortran language and after language specific lexing, parsing, and optimisation, generates LLVM-IR. This is then passed to a specific LLVM back end and built into the final executable, for instance targeting CPUs or GPUs.

III. EXPERIMENTAL SETUP

An AMD Xilinx U280 FPGA is used for the FPGA experiments conducted in this paper, at the default clock frequency of 300MHz. This contains an FPGA chip with 1.08 million LUTs, 4.5MB of on-chip BRAM, 30MB of on-chip UltraRAM, and 9024 DSP slices. This PCIe card also contains 8GB of High Bandwidth Memory (HBM2) and 32GB of DDR DRAM. All bitstreams are built for the U280 using Xilinx's Vitis framework version 2021.2 which at the time of writing is the latest compatible version for the U280. We use Flang version 16 throughout. Reported runtime is the execution time of the HLS kernel on the FPGA and excludes the overhead of data transfer to or from the device. Kernel execution time is gathered via profiling information on the OpenCL event.

For the CPU comparison runs, which are intended to provide context around the optimisations performed by the Clang and Flang front ends respectively, were undertaken on a single core of a Xeon Platinum (Cascade Lake) 8260M CPUs running at 2.40GHz. All reported performance times are based on wall clock time and numbers are averaged over 10 runs.

IV. FORTRAN FPGA PROGRAMMING VIA FLANG HLS

The behaviour of AMD Xilinx's Vitis `v++`, the tool that synthesises C/C++ code by calling HLS, can be altered to provide AMD Xilinx's LLVM-based back end externally generated LLVM-IR as long as the file has the extension `.xpirbc`. We leverage this approach to enable Fortran on Xilinx's FPGAs by connecting LLVM Flang to the AMD Xilinx back end via Flang's generated LLVM-IR, with the major objective being to leave LLVM Flang and the AMD Xilinx back end unchanged.

There are numerous challenges that must be overcome for AMD Xilinx's HLS back end to properly interpret the LLVM-IR. This is because Flang must conform to certain requirements when generating LLVM-IR based upon what is expected by the AMD Xilinx back end. In summary, these are:

1) The generated LLVM-IR must conform to LLVM version 7.
2) HLS pragmas must be lowered into AMD Xilinx's specific LLVM-IR metadata or directives.
3) HLS Streams are defined and managed with custom IR primitives generated by AMD Xilinx's custom Clang front end.

The challenge is that none of these requirements are directly met by Flang. For instance, at the time of writing the latest
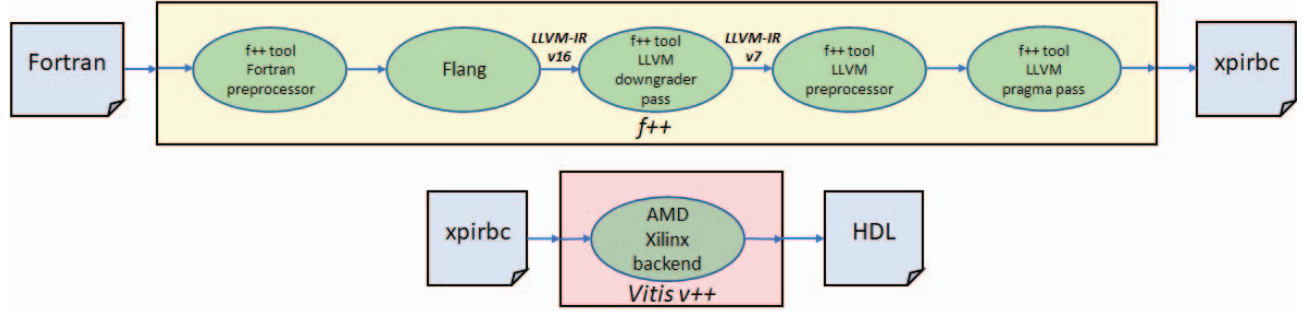
Fig. 2: Overall workflow of our Fortran, `f++`, tool where Fortran is converted into the *xpirbc* format by our tool and this is then provided to the AMD Xilinx back end which generates the HDL.

version of LLVM is version 16, in fact version 7 was from 2018 and is not supported by the latest Flang. Similarly, Flang has no internal knowledge of HLS streams or HLS annotations and as a result, the IR that is generated is not compatible with the back end unless these are explicitly added.

To address these issues, our approach places a layer between Flang and AMD Xilinx's back end which, as illustrated in Figure 2, performs passes and regular expressions on the Flang generated LLVM-IR to transform it into a form that is acceptable to the back end. These are all wrapped in our `f++` tool, which launches separate components of the Fortran HLS workflow.

A. Lowering the generated LLVM-IR to version 7

The differences between the latest versions of LLVM-IR and version 7 will either result in the LLVM-IR generated by Flang being rejected by the AMD Xilinx HLS back end, or accepted and subsequent erroneous results. The later can be especially challenging, for instance if the LLVM-IR contains the `byval` attribute for memory pointers, which is used by the latest LLVM version, then HLS will succeed and generate HDL, but after running the synthesis, place, and route stages the resulting bitstream will be incompatible with the Xilinx Runtime library, XRT. Whilst it might seen an obvious approach for Xilinx to update their LLVM back end to accept the latest LLVM-IR, this is non-trivial because of the differences between LLVM-IR version 7 and the current version. An example of this is that, as of version 16, LLVM-IR only supports opaque pointers at the IR level which are incompatible with HLS. Furthermore, the rapid pace of change in and around LLVM means that it is not always possible for mature products, such as AMD Xilinx's HLS, built upon this technology to track the latest bleeding-edge version.

Therefore, we developed a pass that will downgrade the LLVM-IR to version 7. This was initially based upon the prototype developed by the Khronos team [15] as part of their integration with SYCL. However, their work was built for Clang and only partially amends the generated LLVM-IR. For example, our pass also transforms procedure names into the format expected by the back end.Furthermore, to enable compatibility with Flang our pass also transforms procedure

argument names, which cannot begin with a number, but are generated as such by Flang. Finally, we strip the LLVM-IR of all metadata which does not conform to LLVM-IR version 7.

B. HLS pragmas

Our research prototype supports a subset of the most commonly used HLS pragmas, and it is trivial to add others following the same approach we have developed. It was decided to concentrate on those pragmas which are fundamental for controlling performance, namely the *pipeline*, *unroll*, *array_partition*, and *dataflow* pragmas, as well as the *interface* pragma which enables programmers to define the data interfaces of the generated HLS IP block. Crucially, all pragmas are handled in a similar way, and supporting a pragma involves the following steps:

1) Preprocessing to replace the pragma by a call to a placeholder Fortran subroutine with no arguments. This will lead to the Flang compiler generating a placeholder function call in the LLVM-IR, and we followed this approach because it does not introduce extra instructions before or after the call. Any arguments passed to the pragma are encoded as part of the subroutine name and this approach has been adopted because, if we were to pass them as arguments, then Flang will generate corresponding values in the IR, which makes it more difficult to then process and can in some circumstances result in bottlenecks in the generated HDL.

2) Run our bespoke LLVM pass which operates on the Flang generated LLVM-IR. Each placeholder subroutine call is identified and any arguments are decoded and extracted. The corresponding LLVM-IR constructs for the specific pragma that this corresponds to are inserted into the IR as appropriate.

3) Remove placeholder subroutine calls from the LLVM-IR

We followed this approach because it means that we can undertake pragma replacement via regular expressions in a preprocessing phase, rather than having to modify Flang itself to be aware of the pragmas. This would have required changes to Flang which would then diverge it from the main tree.

An important aspect to highlight, and a further challenge, is how the contention of instructions operate in loop trees in
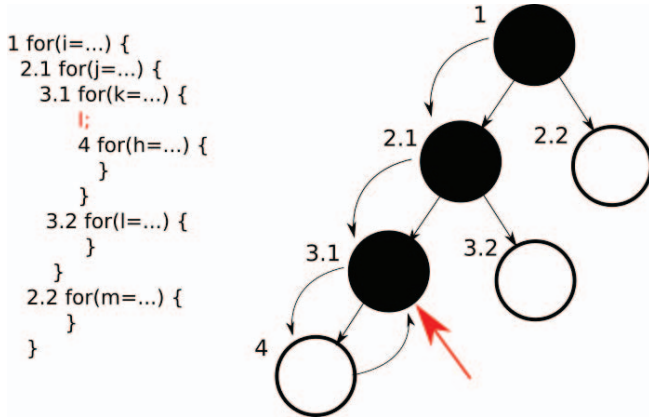
Fig. 3: Nested for loops and their loop tree, traversed with DFS with backtracking. Pointed by the red arrow, the loop containing instruction I. Black nodes contain the instruction.

```
1   proto_hls_stream(integer)
2
3   subroutine f(a, b)
4       use hls_stream
5
6       integer, dimension(100) :: a, b
7       type(HLSStream) :: s
8
9       set_hls_stream_type(s, integer)
10
11      do i = 1,100
12          call hls_write(a(i), s)
13      end do
14
15      do i = 1,100
16          b(i) = hls_read(s)
17      end do
18  end subroutine
```

Listing 1: Example Fortran program using our Fortran HLS Stream macro library.

LLVM-IR, as this is needed to correctly correlate the *pipeline* and *unroll* pragmas with the loops that they are operating over. In LLVM, given a loop L and any of its sub loops S and the instruction I, $I \in S \implies I \in L$. Therefore, our pass must identify the deepest sub loop in the loop tree that contains I using depth first search with backtracking. Otherwise, for nested loops, if the corresponding LLVM-IR was generated at the location of the place holder subroutine call, then the parent loop would be qualified in every case instead, altering the semantics of the program. Figure 3 illustrates the equivalence of a series of nested *for* loops and their corresponding loop tree, where we undertake this search for the annotated loop.

C. HLS streams and polymorphism

In AMD Xilinx's HLS, the C++ template mechanism is leveraged to implement the HLS streams library and this uses polymorphism to enable many different datatypes to be streamed. Whilst polymorphism is available from Fortran 2008 onwards, we decided to keep our tool backwards-compatible with Fortran 90 because that is the most common Fortran version used by HPC codes. Consequently, it was undesirable to encode any features that required later versions of Fortran. Therefore, we replicate the polymorphic behaviour with a library of preprocessor macros and preprocessing of the Fortran code using regular expressions.

Listing 1 sketches an example of Fortran code using our macro-based HLS Stream library. The stream functions provided in Vitis HLS (`read`, `write`, `empty` and `full`) are also provided in Fortran HLS. The module is instantiated in line 1 for the types that will be used with streams in the code, just an `integer` in this example, with twenty Fortran types and precisions supported overall by the macro. Listing 2 illustrates the macro expansion which will replace line 1 in Listing 1 by this preprocessor generated code.

The generated *hls_stream* module contains the user derived type `HLSStream` which enables Flang to check the correct use of the library at compile-time. For the types provided by the user in the macro function `proto_hls_stream`, the corresponding set of HLS stream subroutines are generated for each of these types inside the module. For instance, the *hls_lowered_write_integer* subroutine at line 8 in Listing 2 has been generated to accept an integer because that is the type provided to `proto_hls_stream`.

Similarly to the template instantiation mechanism in C/C++ HLS for typing streams, the macro at line 9 of Listing 1 is equivalent to `hls::stream<int> s`. This macro is expanded to `call set_depth_integer(s%data_integer, 0)` and has two purposes. Firstly, the type of the stream is added to a dictionary that keeps track of the stream types for each subroutine in the code. Secondly, the function call in the IR generated by Flang is replaced by the corresponding AMD Xilinx LLVM-IR primitive `@llvm.fpga.set.stream.depth`. This primitive is fundamental for AMD Xilinx's HLS back end to correctly handle streams, because otherwise it will attempt to implement these as Master AXI-4 interfaces.

This approach of representing stream polymorphic types as a Fortran derived type, similar to a *typedef* in C, with one field of the derived type being the type of the stream, is beneficial because we found that it ensures Flang generates similar LLVM-IR to Clang when accessing the member of a structure. This makes it more straightforward to locate the argument(s) of the AMD Xilinx's LLVM-IR primitive and/or the result of invoking such functions.

V. EVALUATION

We evaluate the work presented in this paper with TeaLeaf [16], an HPC mini-app that solves the linear heat conduction equation using 5 point stencils with implicit solvers. In total, this suite comprises 35 kernels which corresponds to 2587 lines of code ranging from Conjugate Gradient iteration to the Chebyshev method. Table I presents performance results

TABLE I: TeaLeaf mini-app kernels in HLS using C and Fortran (unoptimised) for the FPGA, also comparing against corresponding CPU performance.

Benchmark	Problem size	Alveo U280 FPGA		Xeon Platinum CPU	
		HLS C Runtime(s)	**HLS Fortran** Runtime(s)	**C** Runtime(s)	**Fortran** Runtime(s)
calc_2norm	480000	0.056	0.038	0.002	0.089
calc_residual	480000	0.196	0.215	0.008	0.087
cg_calc_p	480000	0.315	0.319	0.002	0.003
cg_calc_ur	480000	0.781	0.947	0.015	0.034
cg_calc_w	480000	0.198	0.201	0.009	0.006
cg_calc_w_norxy	480000	0.317	0.299	0.008	0.005
cg_init	480000	0.214	0.214	0.007	0.007
cheby_init	480000	0.850	0.845	0.025	0.093
cheby_iterate	480000	0.628	1.072	0.016	0.083
common_init	480000	0.711	0.631	0.025	0.093
field_summary	480000	0.105	0.054	0.008	0.095
finalise	480000	0.003	0.055	≈0.000	0.107
generate_chunk	480000	0.369	0.898	0.008	0.096
initialise_chunk	480000	0.116	0.143	≈0.000	0.048
jacobi_solve	480000	0.192	0.230	0.018	0.001
ppcg_calc_rrn	39677401	3.871	3.866	0.184	0.203
ppcg_calc_zrnorm	480000	0.059	0.059	0.002	0.086
ppcg_init	480000	0.343	0.509	0.009	0.109
ppcg_init_sd	480000	0.227	0.188	0.005	0.069
ppcg_inner	480000	0.754	0.727	≈0.000	0.106
ppcg_inner_norxy	480000	0.748	0.704	≈0.000	0.098
ppcg_pupdate	480000	0.043	0.044	0.002	0.084
ppcg_store_r	480000	0.043	0.044	0.002	0.086
ppcg_update_z	480000	0.043	0.044	0.002	0.099
set_field	480000	0.043	0.044	0.002	0.087
tea_block_init	480000	0.616	0.615	0.005	0.020
tea_block_solve	480000	0.098	0.267	≈0.000	0.026
tea_diag_init	480000	0.083	0.083	0.002	0.019
tea_diag_solve	480000	0.083	0.099	0.002	0.019
update_halo	480000	0.788	0.889	0.011	0.059
update_halo_cell	480000	0.001	0.001	≈0.000	0.008
update_in_halo_bt	480000	0.016	0.016	≈0.000	0.067
update_in_halo_cell_bt	480000	0.002	0.002	≈0.000	0.004
update_in_halo_cell_lr	480000	0.049	0.080	0.001	0.006
update_in_halo_lr	480000	0.874	0.898	0.011	0.067

```
1   module hls_stream
2       type(HLSStream)
3       end type
4
5       type(HLSStream_integer)
6           integer :: data_integer
7       end type
8       subroutine hls_lowered_write_integer(
9           input_data, stream_data)
10          integer, value :: input_data
11          integer :: stream_data
12      end subroutine
13      ...
14  end module
```

Listing 2: Expansion of `proto_hls_stream` macro. Only the generated hls_lowered_write_integer subroutine shown.

for the baseline implementation of these kernels on an AMD Xilinx Alveo U280 FPGA and a single core of a Xeon Platinum CPU. It should be highlighted that the standard deviation across the 10 averaged runtimes is negligible, only affecting the 4th decimal figure of the results, and hence this has been omitted for brevity. It is not our intention in this paper

to study the performance differences between architectures for these benchmarks and Table I reports a direct translation from the CPU to FPGA without any FPGA specific optimisations being undertaken because, in our first experiment, we are interested in the performance that our HLS Fortran tool can provide *out of the box*. Indeed, the comparison between FPGA and CPU performance is provided to highlight where there are performance differences resulting from the activities of the front ends of Clang and Flang, for instance undertaking language specific code-level optimisations.

Several high level observations can be made from the runtime results reported in Table I. Firstly, it can be seen that there are performance differences between the HLS C and HLS Fortran kernels, but in the main these differences are fairly small. Furthermore, there is no clear pattern around whether C or Fortran performs best with HLS. For instance, the *field_summary* Fortran kernel outperforms C, whereas the opposite is true for the *generate_chunk* kernel. Table II reports resource usage for these TeaLeaf kernels when using HLS C and HLS Fortran. The numbers in Table II are expressed as the percentage of overall resources that are used on the Alveo U280, and it can be seen that whilst the BRAM resource usage

14

between HLS C and HLS Fortran is fairly consistent, for the LUTs, FFs, and DSP slices it is more varied. In general terms, HLS Fortran tends to require more resources, especially LUTs, compared to HLS C but there are exceptions to this such as with the *cg_calc_p* kernel.

```
1   int x_dim = x_max − x_min;
2   int y_dim = y_max − y_min;
3   int w_x = x_max − x_min +
4       2 ∗ halo_exchange_depth + 1;
5   for(int k=0; k <= y_dim; k++) {
6       for(int j=0; j <= x_dim; j++) {
7           w[(k + halo_exchange_depth) ∗ w_x +
8               (j + halo_exchange_depth)] = ...
9       }
10  }
```

Listing 3: Access to a flattened 2D array with manually calculated offsets in C for *cg_calc_w_norxy* kernel.

Whilst the same algorithms from the TeaLeaf suite are being compiled using HLS C and HLS Fortran, the performance and resource usage differences between these flows reported in Tables I and II demonstrate that there are implications of whether one uses C or Fortran to drive HLS. This is interesting because, based upon the fact that thus far we are using Von-Neumann based algorithms, then one would expect the unoptimised nature of these codes to dominate and performance to be very similar or the same between HLS C and HLS Fortran. Indeed our objective in this work has been for HLS Fortran to match the performance of HLS C. However, clearly the results are more nuanced and we focus on two kernels to explore these differences in more detail. Starting with the *cg_calc_w_norxy* kernel which undertakes Conjugate Gradient, as shown in Table I, this performs better on the FPGA in Fortran compared to C. The reason for this is because of the explicit calculation of offsets for accessing the flattened 2-dimensional array, as shown in Listing 3. Vitis imposes a limitation that external arrays have a maximum of one dimension, in C this must be handled manually by the programmer but with Fortran it is possible to use N-dimensional arrays. This is because dynamic arrays require pointer to pointers in C and are not supported by Vitis HLS [17]. Arrays in Fortran are higher-level than in C, and-so during compilation Flang is able to calculate the offsets and flatten the dimensions of the arrays statically, rather than requiring this be undertaken at runtime, yielding an IR compatible with the aforementioned restriction. The programmer's Fortran code is illustrated in Listing 4, and this demonstrates an important advantage of using Fortran compared to C/C++ for programming FPGAs, where the additional expressiveness of Fortran, in this case via N-dimensional arrays, means that the compiler is able to undertake such offset calculations statically which are low level, time consuming, and can be error prone.

```
1   REAL(KIND=8),
2   DIMENSION(x_min−halo_exchange_depth:
3       x_max+halo_exchange_depth,
4       y_min−halo_exchange_depth:
5       y_max+halo_exchange_depth) :: w
```

```
6
7   do k=y_min,y_max
8       do j=x_min,x_max
9           w(j,k) = ...
10      end do
11  end do
```

Listing 4: Access to a 2D array in Fortran for *cg_calc_w_norxy* kernel.

The *ppcg_calc_rrn* kernel, which undertakes a reduction and whose runtime is reported in Table I, is also slightly faster using our Fortran-based approach compared to C on the FPGA. It is instructive to compare the runtime between C and Fortran on the FPGA against that of the CPU to understand whether performance differences come from these respective front ends. For instance, it can be seen that for the *cg_calc_w_norxy* kernel the Fortran runtime is lower than the C runtime for both the FPGA and CPU, whereas this is not the case for the *ppcg_calc_rrn* kernel where Fortran is slightly faster than C on the FPGA but slower on the CPU. It can be seen that there is no clear pattern, and just because one language is faster than the other on the FPGA then it does not mean that the same will be true on the CPU, or vice versa. Consequently, it can be deduced that it is the combination of both the front end and back end which is determining runtime performance and differences between languages are at the individual kernel level.

Results reported in Table I are unoptimised for the FPGA, and whilst this was an important baseline to understand the performance differences delivered for such codes, it is well known that Von-Neumann based codes must be transformed to their dataflow counterparts for best performance on an FPGA [18]. A key question was therefore whether support in our tool was rich enough to enable these same optimisations to be undertaken at the code level in HLS via Fortran, as is currently possible in C or C++. We focus on optimising the *cg_calc_w_norxy* and *ppcg_calc_rrn* kernels, with the results of optimising these kernels reported in Table III. We optimised the *cg_calc_w_norxy* Conjugate Gradient stencil kernel using shift registers. This technique replaces the strided memory accesses by sequential accesses to a shift register, where every element can be accessed in one cycle due to the initial prefetching of part of the array and a sliding window, where the access to new elements from memory is pipelined via the shift register operation [6]. It can be seen that this optimisation reduces the runtime of both the C and Fortran versions compared to their unoptimised counterparts.

The bottleneck with the *ppcg_calc_rrn* reduction kernel is in the floating point adder, which requires 7 cycles to complete a sum, therefore leading to an initiation interval of 7 because results from consecutive sum operations are required. We ameliorate this with partial sums, where Vitis pipelines the adder to undertake seven independent sum operations every 7 cycles instead [19], and then at the end adds these seven partial sums together to calculate the overall result.

There are two versions of the *ppcg_calc_rrn* kernel reported in Table III, *partial sums* uses the partial sum technique only

15

	HLS C % usage				HLS Fortran % usage			
Kernel	BRAM	LUT	FF	DSP	BRAM	LUT	FF	DSP
calc_2norm	0.10	0.32	0.14	0.19	0.10	0.43	0.16	0.16
calc_residual	0.10	0.50	0.30	0.32	0.10	0.64	0.35	0.29
cg_calc_p	0.10	0.52	0.22	0.32	0.10	0.49	0.22	0.27
cg_calc_ur	0.10	2.24	1.20	1.30	0.10	2.88	1.23	1.00
cg_calc_w	0.10	0.69	0.36	0.42	0.10	0.80	0.39	0.32
cg_calc_w_norxy	0.10	0.60	0.31	0.32	0.10	0.75	0.37	0.32
cg_init	0.10	1.82	1.09	1.26	0.10	2.19	0.95	0.81
cheby_init	0.10	2.15	1.31	1.43	0.10	2.50	1.19	0.98
cheby_iterate	0.10	2.31	1.36	1.46	0.10	2.85	1.31	1.10
common_init	0.10	2.29	1.42	1.13	0.10	2.82	1.53	0.79
field_summary	0.10	0.44	0.24	0.22	0.10	0.56	0.26	0.27
finalise	0.10	0.22	0.11	0.10	0.10	0.30	0.13	0.03
generate_chunk	0.10	1.05	0.54	0.32	0.10	1.63	0.74	0.70
initialise_chunk	0.10	0.91	0.33	0.12	0.10	1.07	0.46	0.22
jacobi_solve	0.10	0.63	0.37	0.35	0.10	0.85	0.44	0.32
ppcg_calc_rrn	0.10	0.37	0.17	0.19	0.10	0.45	0.20	0.23
ppcg_calc_zrnorm	0.10	0.35	0.15	0.19	0.10	0.46	0.18	0.16
ppcg_init	0.10	1.95	1.15	1.33	0.10	2.39	1.04	0.84
ppcg_init_sd	0.10	0.54	0.26	0.22	0.10	0.68	0.30	0.20
ppcg_inner	0.10	2.88	1.61	1.66	0.10	3.62	1.70	1.46
ppcg_inner_norxy	0.10	2.55	1.43	1.46	0.10	3.16	1.43	1.14
ppcg_pupdate	0.10	0.23	0.09	0.07	0.10	0.31	0.11	0.03
ppcg_store_r	0.10	0.23	0.09	0.07	0.10	0.31	0.11	0.03
ppcg_update_z	0.10	0.23	0.09	0.07	0.10	0.31	0.11	0.03
set_field	0.10	0.23	0.09	0.07	0.10	0.31	0.11	0.03
tea_block_init	0.10	0.78	0.35	0.48	0.10	0.89	0.35	0.35
tea_block_solve	0.10	1.23	0.82	0.93	0.10	1.30	0.65	0.61
tea_diag_init	0.10	0.26	0.38	0.07	0.10	0.36	0.42	0.07
tea_diag_solve	0.10	0.24	0.12	0.16	0.10	0.39	0.16	0.12
update_halo	0.20	1.95	0.44	0.20	0.74	2.76	0.65	0.31
update_halo_cell	0.20	1.35	0.32	0.20	0.20	1.40	0.40	0.31
update_in_halo_bt	0.74	1.48	0.34	0.13	0.74	1.67	0.42	0.17
update_in_halo_cell_bt	0.10	0.40	0.15	0.13	0.10	0.56	0.23	0.17
update_in_halo_cell_lr	0.10	0.39	0.18	0.13	0.10	0.50	0.22	0.13
update_in_halo_lr	0.74	1.52	0.35	0.13	0.74	1.63	0.38	0.13

TABLE II: Percentage of resource utilisation on the Alveo U280 FPGA for TeaLeaf mini-app kernels written in HLS using C or Fortran.

TABLE III: Optimised kernels in HLS C and HLS Fortran for the FPGA, and also a synthetic streaming kernel. Problem size of 480000 for *cg_calc_w_norxy*, 39677401 for *ppcg_calc_rrn*, and 32 million elements for Synthetic Streaming.

	Alveo U280 FPGA	
Benchmark	HLS C Runtime(s)	HLS Fortran Runtime(s)
cg_calc_w_norxy	0.200	0.222
ppcg_calc_rrn (partial sums)	0.630	0.631
ppcg_calc_rrn (dataflow)	0.820	0.535
Synthetic Streaming	0.131	0.132

and it can be seen that both Fortran and C deliver very similar performance on the FPGA, which is significantly faster than the unoptimised version in Table I. The second version of the *ppcg_calc_rrn* kernel, *dataflow*, uses the *dataflow* pragma and HLS streams to pipeline both the compute and interaction with external memory to load and store data.

This second optimised version of the *ppcg_calc_rrn* kernel, using dataflow regions and HLS streams, yielded surprising results, as it can be seen from Table III that the Fortran version is 53.27% faster than the C HLS code. Based upon the report generated by Vitis HLS, we observed that Vitis is unable to pipeline the reduction loop when using C, whereas it is able to do so in Fortran. Because of the failure of Vitis to pipeline the loop in C, the performance of the dataflow optimisation does not improve with respect to the partial sums. Whilst our kernels all ran at 300Mhz on the FPGA, it should also be noted that the maximum operating frequency reported by Vitis HLS is also different, up to 371.47 MHz possible with Fortran and 357.68 MHz for C. It is interesting to observe that small differences in the LLVM-IR resulting from different language front ends, can have a significant impact.

Finally, we developed a synthetic example, Synthetic Streaming in Table III, which sums two vectors from external memory as data is streamed in. This kernel relies on three dataflow regions connected via HLS streams; loading data from external memory, undertaking the element sum, and writing results back to external memory. Compared to the *ppcg_calc_rrn* kernel, these loops are much simpler and-so can be pipelined by Vitis when using either the C and Fortran flows, and both languages deliver similar performance.

Another important consideration is the resource utilisation that results from these tools when undertaking code optimisation, and whether one approach is more frugal than another. Table IV reports resource utilisation for the four optimised

TABLE IV: Percentage total resource usage in the optimised kernels and synthetic streaming in HLS C and HLS Fortran. Percentage overhead of Fortran also reported in brackets, in green the cases where Fortran leads to less resource utilisation.

Benchmark	HLS kernel usage C/Fortran (% total resources)			
	LUTs	FFs	BRAM	DSP
cg_calc_w _norxy	6.13 / 5.76	6.32 / 5.13	3.17 / 3.17	0.74 / 0.74
	(-6.04%)	(-18.83%)	(0.00%)	(0.00%)
ppcg_calc_rrn (partial sums)	0.65 / 0.60	0.34 / 0.30	0.40 / 0.40	0.19 / 0.22
	(-7.69%)	(-11.76%)	(0.00%)	(15.79%)
ppcg_calc_rrn (dataflow)	0.64 / 0.78	0.34 / 0.40	0.40 / 0.40	0.19 / 0.39
	(21.88%)	(17.65%)	(0.00%)	(105.26%)
Synthetic Streaming	0.29 / 0.22	0.10 / 0.08	0.15 / 0.10	0.00 / 0.00
	(-24.13%)	(-20.00%)	(-33.33%)	-

benchmarks that were explored in Table III and these can be compared against the resource utilisation for unoptimised kernels reported in Table II. For each benchmark and resource, the percentage of overall resource usage is reported for the C and then the Fortran code, with these separated by a slash. Underneath, in brackets, the percentage overhead of Fortran compared to C is reported, where green is used to highlight situations where Fortran resulted in reduced resource usage.

It can be seen in Table IV that for both *cg_calc_w_norxy* and *ppcg_calc_rrn* with partial sums kernels, Fortran requires fewer LUTs and FFs than C, with BRAM and DSP being comparable. With the synthetic streaming benchmark, where performance was comparable between the C and Fortran versions, it can be seen that Fortran also uses fewer resources.

Fortran requires resources than C for the *ppcg_calc_rrn* with dataflow kernel, especially for DSP slices, however the hardware that is generated is fundamentally different between these versions due to the lack of pipelining on the reduction loop in C which led to lower performance. Therefore, overall, these utilisation results demonstrate that Fortran is competitive against C for resource utilisation when optimising the kernel and, in some cases, such as synthetic streaming, can result in fewer resources with no loss in performance.

VI. CONCLUSIONS AND FURTHER WORK

The work described in this paper enables programmers to leverage Fortran when programming AMD Xilinx FPGAs, which is especially important for HPC workloads as many of these are written in Fortran. Whilst Fortran programmers still need to transform their Von-Neumann algorithms to a dataflow representation via algorithmic restructuring and apply appropriate HLS pragmas, crucially our tool avoids the time consuming and error prone preliminary step of converting code into C/C++. This, combined with the fact that code remains in one single language which HPC developers are familiar with, is beneficial in encouraging HPC developers to explore accelerating their Fortran codes using FPGAs. The tool that we have described connects LLVM's Flang with the AMD Xilinx HLS back end, without any modification required to

either Flang or the back end. As a result, the programmer is, apart from our conversion utility, using standard tools that are maintained by the wider community. There is no divergence from LLVM Flang or the AMD Xilinx HLS back end, and programmers benefit from enhancements made both in Flang by the community and in the HLS back end by AMD Xilinx. Furthermore, wider features of the Vitis ecosystem, such as profiling, are also made available to the Fortran programmer.

We have compared performance of our approach against the existing C/C++ Vitis HLS tooling. It was observed that, in the main, performance between these tools is fairly comparable, although there are some differences for instance with Fortran where the programmer can leverage higher level features, such as N-dimensional arrays, that improves expressiveness and enables the compiler to undertake additional optimisations. The major benefit of our approach is that we unlock the ability for large HPC applications that are written in Fortran to be run on FPGAs, and as we demonstrated these pragmas currently supported by our tool enables these codes to be optimised for FPGAs, with additional pragmas and constructs being trivial to add to the flow if required.

An objective has been for our HLS Fortran flow to match the performance of HLS C so that HPC programmers are able to accelerate their Fortran codes on FPGAs without any reduction in performance compared to leveraging C/C++. However, it can be observed that some HLS Fortran kernels execute slightly slower than their HLS C versions and consequently, for further work, optimisation at the LLVM-IR level would be beneficial. For instance, the *cg_calc_w_norxy* Fortran kernel performs slightly worse than its C counterpart and we believe that this is because of the use of long (64-bit) integers for array indicies in the IR generated by Flang. Currently these require integer conversion from the programmer's 32-bit indicies to 64-bit in the IR, and this could be optimised by our tool to avoid such conversions. It would also be interesting to apply our code to programming the AI engines that are present on the Versal, potentially providing a unified programming interface between the PL and AI engines. Because AMD Xilinx have already developed MLIR dialects for their AI engine compiler, and internally Flang uses MLIR via the FIR dialect, it should be possible to connect these tools and leverage our HLS Fortran flow. Furthermore, it would be interesting to explore generalising our tool, for instance to support other programming languages, such as Python, Rust or Julia, to be integrated with the AMD Xilinx back end.

We intend to make our tool open source and available at *https://fpga.epcc.ed.ac.uk/community/fortran.html* .

REFERENCES

[1] W. Brainerd, "The importance of Fortran in the 21st century," *Journal of Modern Applied Statistical Methods*, vol. 2, no. 1, p. 3, 2003.

[2] P. E. Strazdins, M. Kahn, J. Henrichs, T. Pugh, and M. Rezny, "Profiling methodology and performance tuning of the met office unified model for weather and climate simulations," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 2011, pp. 1322–1331.

[3] N. Brown, M. Weiland, A. Hill, B. Shipway, C. Maynard, T. Allen, and M. Rezny, "A highly scalable Met Office NERC Cloud model," *arXiv preprint arXiv:2009.12849*, 2020.

[4] B. Yang, X. Fang, L. Zhang, F. Zhuang, M. Bi, C. Chen, G. Li, and X. Wang, "Applicability of empirical models of isentropic efficiency and mass flow rate of dynamic compressors to jet engines," *Progress in Aerospace Sciences*, vol. 106, pp. 32–42, 2019.

[5] O. Lehmkuhl, A. Lozano-Durán, and I. Rodriguez, "Active flow control for external aerodynamics: from micro air vehicles to a full aircraft in stall," in *Journal of Physics: Conference Series*, vol. 1522, no. 1. IOP Publishing, 2020, p. 012017.

[6] N. Brown, "Accelerating advection for atmospheric modelling on Xilinx and Intel FPGAs," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 767–774.

[7] M. A. Dávila-Guzmán, R. G. Tejero, M. Villarroya-Gaudó, and D. S. Gracia, "An Analytical Model of Memory-Bound Applications Compiled with High Level Synthesis," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 218–218.

[8] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 741–755.

[9] S. Curzel, S. Jovic, M. Fiorito, A. Tumeo, F. Ferrandi *et al.*, "Higher-Level Synthesis: experimenting with MLIR polyhedral representations for accelerator design," in *IMPACT 2022: 12th International Workshop on Polyhedral Compilation Techniques*, 2022, pp. 1–10.

[10] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[11] A. N. Ziogas, T. Schneider, T. Ben-Nun, A. Calotoiu, T. De Matteis, J. de Fine Licht, L. Lavarini, and T. Hoefler, "Productivity, portability, performance: data-centric Python," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[12] S. W. Nabi and W. Vanderbauwhede, "Automatic pipelining and vectorization of scientific code for FPGAs," *International Journal of Reconfigurable Computing*, vol. 2019, pp. 1–12, 2019.

[13] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008, pp. 1–20.

[14] L. J. Kedward, B. Aradi, O. Čertík, M. Curcic, S. Ehlert, P. Engel, R. Goswami, M. Hirsch, A. Lozada-Blanco, V. Magnin *et al.*, "The state of Fortran," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 63–72, 2022.

[15] L. Forget, G. Harnisch, R. Keryell, and F. de Dinechin, "A single-source C++ 20 HLS flow for function evaluation on FPGA and beyond." in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2022, pp. 51–58.

[16] S. McIntosh-Smith, M. Martineau, T. Deakin, G. Pawelczak, W. Gaudin, P. Garrett, W. Liu, R. Smedley-Stevenson, and D. Beckingsale, "TeaLeaf: A mini-application to enable design-space explorations for iterative sparse linear solvers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 842–849.

[17] A. XILINX, *Vitis High-Level Synthesis User Guide-UG1399 (v2022.2)*. June, 2022.

[18] N. Brown, "Porting incompressible flow matrix assembly to FPGAs for accelerating HPC engineering simulations," in *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 2021, pp. 9–20.

[19] J. Hrica, "Floating-point design with Vivado HLS," *Xilinx Application Note*, 2012.

18