Hindawi
International Journal of Reconfigurable Computing
Volume 2019, Article ID 7348013, 12 pages
https://doi.org/10.1155/2019/7348013

Research Article

Automatic Pipelining and Vectorization of Scientific Code for FPGAs

Syed Waqar Nabi and Wim Vanderbauwhede

School of Computing Science, University of Glasgow, Glasgow, UK

Correspondence should be addressed to Syed Waqar Nabi; syed.nabi@glasgow.ac.uk

Received 4 May 2019; Revised 4 August 2019; Accepted 8 October 2019; Published 18 November 2019

Academic Editor: John Kalomiros

There is a large body of legacy scientific code in use today that could benefit from execution on accelerator devices like GPUs and FPGAs. Manual translation of such legacy code into device-specific parallel code requires significant manual effort and is a major obstacle to wider FPGA adoption. We are developing an automated optimizing compiler TyTra to overcome this obstacle. The TyTra flow aims to compile legacy Fortran code automatically for FPGA-based acceleration, while applying suitable optimizations. We present the flow with a focus on two key optimizations, automatic *pipelining* and *vectorization*. Our compiler frontend extracts patterns from legacy Fortran code that can be pipelined and vectorized. The backend first creates fine and coarse-grained pipelines and then automatically vectorizes both the memory access and the datapath based on a cost model, generating an OpenCL-HDL hybrid working solution for FPGA targets on the Amazon cloud. Our results show up to 4.2× performance improvement over baseline OpenCL code.

1. Introduction

Acceleration devices for high-performance computing (HPC) and scientific computing are becoming increasingly heterogeneous. There is a general consensus that no single type of device–CPU, GPU, or FPGA–will be best suited across the entire range of scientific applications. GPUs are already well-established as a practical alternative to conventional CPUs for accelerating scientific applications. A considerable proportion of supercomputers in the top 500 list contains GPU accelerators. FPGAs are a more recent addition to this canvas, and in spite of significant improvements in their performance and programmability in recent years, they are still far from widespread adoption as mainstream acceleration devices.

A key challenge that applies in a lesser or greater extent to all accelerators is writing parallel, high-performance code customized for performance specifically on that device. The challenge is all the more acute for FPGAs, which are notoriously difficult to program. Improvements in FPGA logic capacity as well as high-level synthesis (HLS) programming frameworks such as Altera's (Intel's) AOCL, Xilinx's SDAccel, and Maxeler have played an important role in their transition from peripheral, embedded, or prototyping only devices to first-order desktop accelerators. However, FPGAs have failed to make the kind of inroads in HPC that GPUs have made. This is, in part at least, due to the fact that until very recently there were no practical high-level programming platforms for FPGAs, and even with their introduction, it is still a challenging task to write high-performance code. While heterogeneous programming languages like OpenCL provide *code portability*, they are not *performance portable* across devices. For example, [1] report that "even though OpenCL is functionally portable across devices, direct ports of GPU-optimized code do not perform well compared with kernels optimized with FPGA-specific techniques such as sliding windows. However, by exploiting FPGA-specific optimizations, it is possible to achieve up to 3.4x better power efficiency. . .." Our own previous work has shown that even very simple OpenCL kernel code can lead to very different performance profiles when moving from one FPGA framework to another [2].

It is our contention that such programming and optimization challenges will remain a hurdle to the adoption of acceleration devices–especially FPGAs–in mainstream HPC, and that high-level programming frameworks like OpenCL should themselves be targets for still higher level compilers that can work with sequential, unoptimized legacy code, in which case one could truly have *performance portability*.

We propose an optimizing compiler framework that uses *Type Transformations* (TyTra) to explore FPGA-specific optimizations for a given application and automatically generates the implementation code from legacy Fortran code, leading to the desired code and performance portability. It applies these type transformations on a high-level, functional representation of the kernel (the code to be accelerated on the FPGA), extracted from the Fortran code and then uses a cost model for evaluating the search-space for an optimized solution.

A flowchart of the TyTra framework is shown in Figure 1. The frontend refactor Fortran 77 codes into modern, maintainable, extensible, and accelerator-ready Fortran code (available at https://github.com/wimvanderbauwhede/RefactorF4Acc). We can then generate OpenCL code targeted at GPUs (available at https://github.com/wimvanderbauwhede/AutoParallel-Fortran) or FPGAs (available at https://github.com/wimvanderbauwhede/Fortran-to-OpenCL-FGPA). For FPGA targets specifically, we invoke a more involved optimization pass that translates the OpenCL code to an intermediate representation (IR) (available at https://github.com/waqarnabi/ocl2tir) and then generates low-level hardware description language (HDL) code from it. It is this optimizing route in our flow that we will discuss in the paper, with a focus on two key optimizations: pipelining and vectorization.

We briefly discuss the frontend of the TyTra flow in Section 4. We present our view of the requirements of optimizing code on FPGAs in Section 5, leading into the main contribution of this paper in Section 6, the TyTra backend compiler (TyBEC) and the automatic pipelining and vectorization it enables. We present the evaluation of our approach in the next section before concluding the paper. We start by reviewing some related work.

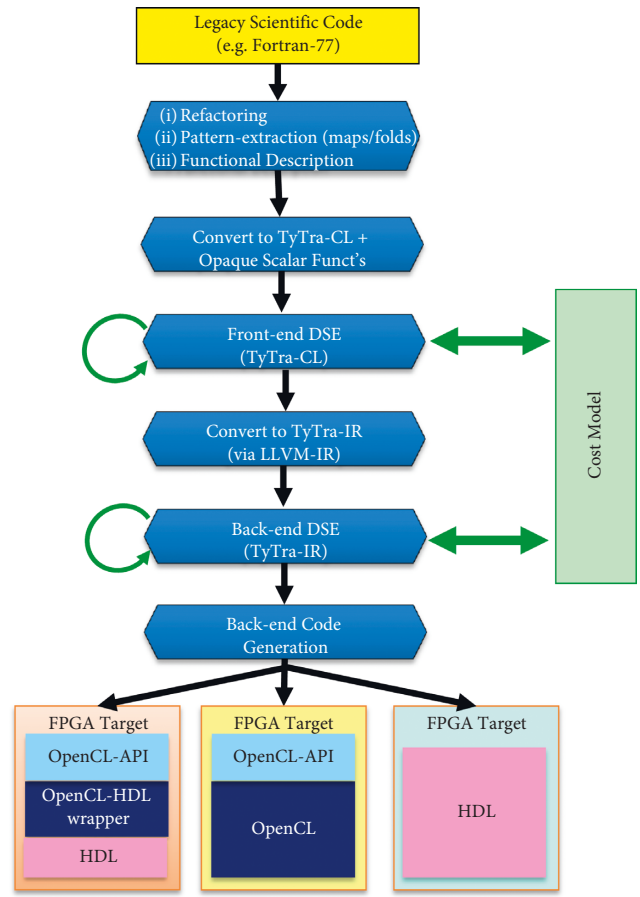

Figure 1: The TyTra optimizing compiler framework. The starting point is Fortran 77 scientific code, though there can be other possible entry points as well. There are a number of backend code-generation options, with this paper's focus on hybrid OpenCL-HDL route, geared towards deployment on the Amazon cloud's F1 instances.

2. Related Work

The viability of FPGAs as *mainstream* acceleration devices for scientific computing is well established in literature. As an example, reference [3] presents a suitability analysis of FPGAs for heterogeneous HPC platforms, using the Berkeley 13 dwarfs as a reference. They found FPGAs to be suitable for 5 of the 13 dwarves, but noted that they are difficult to use for nonspecialized designers and emphasized the importance of more abstraction and less customization. Reference [4] demonstrates the suitability of FPGAs, programmed via OpenCL, for implementing partial differential equation (PDE) based scientific models. The same article, however, showing an OpenCL kernel written with differences in syntax and compiler hints for two different FPGA target devices/vendors, supports our contention that OpenCL code is not performance portable.

There are a number of commercial tools available that provide a high-level programming route for accelerating scientific code on FPGAs. Maxeler [5] is a good example, which provides a Java metaprogramming model for describing computation kernels and connecting data streams between them. It has been used for accelerating applications from various domains, e.g., scientific and financial modelling. Altera (now Intel) OpenCL or AOCL [6] is the implementation of the OpenCL heterogeneous parallel programming framework for Intel FPGAs. While it is based on the OpenCL standard, it has vendor-specific optimization extensions. Xilinx similarly has its own OpenCL implementation called SDAccel [7]. Like AOCL, SDAccel is based on the OpenCL standard and also proposes custom optimizations to improve performance.

There have been other studies that are motivated in a way similar to ours, that is, by the need for a higher abstraction design entry than conventional high-level languages. For example, algorithmic skeletons have been proposed to separate algorithm from architecture-specific parallel programming [8]. SparkCL is an attempt to bring increasingly diverse architectures, including FPGAs, into the familiar

Apache Spark framework [9]. Another route for increasing the design abstraction is to use domain-specific languages (DSLs), and there are numerous examples for FPGAs, e.g., *FSMLanguage* for designing FSMs [10] and *CLICK* for networking applications [11].

A work that is quite similar to ours is the *Geometry of Synthesis* project [12]. It proposes design entry in a functional language paradigm, leading to generation of RTL code for FPGAs. It does not have automatic generation and evaluation of architectural design variants though. There has also been work on exploring vectorization for FPGA pipelines for specific applications (for example, see [13]). Automatic pipelining of high-level code is now possible with commercial tools like Xilinx's SDAccel, Intel's AOCL, and Maxeler, and some tools allow vectorization as well, though it has to be manually programmed or hinted via pragmas. Automatic pipelining and vectorization of high-level code based on a cost model, one that can work with legacy Fortran code, is entirely novel as best as we know.

While not the focus of this paper, a key first step of our flow is refactoring legacy Fortran 77 code to make it more *accelerator friendly*. There are a number of similar refactoring tools available for Fortran, though Fortran 77 is supported by very few. ROSE framework (http://www.rosecompiler.org/index.html) from LLNL [14] is probably the most well known, which relies on the Open Fortran Parser (OFP) (http://fortran-parser.sourceforge.net/). This parser claims to support the Fortran 2008 standard. Furthermore, there is the language-Fortran (https://hackage.haskell.org/package/language-fortran) parser which claims to support Fortran 77 to Fortran 2003. A refactoring framework which claims to support Fortran 77 is CamFort [15]; according to its documentation, it supports Fortran 66, 77, and 90 with various legacy extensions. An eclipse-based interactive refactoring tool Photran [16] supports FORTRAN 77-2008.

3. TyTra Frontend

We will briefly discuss this frontend of the TyTra flow here for completeness; more details are in [17], where we assess its correctness, completeness, and capability.

FORTRAN 77 can be both computationally efficient as well as programmer efficient, allowing the programmer to write code quickly without being too strict about it. However, it becomes very difficult to maintain and port very quickly as a result of this. We aim to make our refactored code modern, maintainable, and extensible.

The requirements in mind were very different compared with today's languages when FORTRAN 77 was designed, especially in terms of avoiding bugs. Some specific features, now unacceptable in modern languages, are: implicit typing, subroutine arguments intended access absent, and absence of a module system, required both for extensibility and maintainability.

The TyTra frontend compiler converts all nonprogram code units into modules. These are then used with an explicit export (only) declaration. There are many more refactorings applied by our compiler, such as rewriting label-bases loops as do-loops etc.

The common feature of the vast majority of current accelerators is that they have a separate memory space, usually physically separate from the host memory. Furthermore, the common offload model is to create a "kernel" subroutine (either explicitly or implicitly) which is run on the accelerator device.

It is extremely important to separate the memory-spaces of the host and the kernel when generating code for modern accelerators. Since Fortran 77 uses global variables liberally, our frontend converts them to subroutine arguments across the entire call tree of the program.

Our goal is to convert legacy Fortran 77 code into parallel code so that the computation can be accelerated on FPGAs. We use a three-step process.

First, the above refactorings give us a modern, maintainable, extensible, and accelerator-ready Fortran 95 codebase. This gives an excellent starting point for many of the other existing tools, e.g., the generated code can be conveniently parallelized using OpenMP or OpenACC annotations. We want, however, to provide an end-to-end solution to the user that does not require any annotations.

The next step in our process is identifying data-level parallelism present in the code in the form of *maps* and *folds*. The terms, *map* and *fold*, are from the functional programming domain and refer to ways of performing an operation on all elements of a list. These constructs are broadly equivalent to loop nests with and without dependencies, and as Fortran is loop-based, our analysis is actually an analysis of loops and dependencies. Internally, though our representation uses the functional programming model where *map* and *fold* are higher-order functions (functions operating on other functions), extracted from the bodies of the loops; we thus raise the abstraction level of our representation, making it independent of both the original code and the final code to be generated. We then apply a number of rewrite rules for map and fold based functional programs (broadly speaking equivalent to loop fusion or fission) to optimise the code.

The third step involves the backend of our framework, the focus of the rest of this paper, where we use patterns extracted to ensure that our kernels are guaranteed to be both pipelineable and vectorizeable. Then, using our cost model, we generate optimized, synthesizable code for Xilinx FPGAs on the Amazon cloud F1 instances.

4. Transforming Scientific Applications for Performance on FPGAs: A Perspective

The potential to get good performance and energy efficiency on FPGAs is widely recognized, but coupled with the realization that achieving the potential is not trivial. It is our view that, in the context of the domain of scientific computing, the guidelines for creating architectures on FPGAs that give high throughput can be summed up as follows:

(1) Create a deep and custom pipeline, fine-grained as well as coarse-grained

(2) Coalesce (vectorize) global memory accesses

(3) Vectorize the pipelined datapath

(4) Minimize data stalls on these pipelines

(5) Maximize throughput by replicating appropriate functional units

(6) Minimize random access to external memory by optimizing stencil computations

(7) Use optimized numerics where possible

(8) Use vendor-optimizations where suitable

The design-space exploration (DSE) in TyTra is informed by the view summarized above and is carried out at two abstractions: the *frontend* and the *backend*. In this work, we highlight the backend, specifically two key optimizations: pipelining and vectorization (points 1–3).

4.1. Pipelining. FPGAs consist of a fine-grained reconfigurable fabric that can be customized for a given application. If the underlying application is amenable to pipeline parallelism, then high throughput on FPGAs can be achieved by creating deep, custom pipelines. Pipelining can be done at three hierarchical levels.

4.1.1. Pipelining inside Instructions. This refers to atomic datapath instructions like floating point operations that require multiple clock cycles to complete. It is important to pipeline them to achieve a high operational frequency on the FPGA and also to maintain throughput. Such pipelined functional units are available both in the academia and via vendor tools. For example, we use *FloPoCo* [18], an open-source tool, to generate pipelined functional units for our solution.

4.1.2. Pipelining across Instructions. To get good performance, pipelining instructions inside computation kernels is crucial. This requires a data-dependence analysis to ensure data hazards are avoided. We refer to such pipelines in this work as *fine-grained* pipelining.

4.1.3. Pipelining across Kernels. While most FPGA HLS tools would automatically pipeline at the fine-grained level, it is important to pipeline at this *coarse-grained* level as well to get viable performance on FPGAs for large scientific problems. Such pipelining obviates the need of using the FPGA external DRAM (i.e., *global memory* in OpenCL terminology) to communicate between kernels.

4.2. Vectorization. To optimize throughput and utilize a target device at its maximum or near-maximum potential, the external memory interface should ideally be operating at or near saturation. This is typically achieved by *coalescing* access to memory [19] and/or using multiple memory banks concurrently [13]. The purpose is to read multiple array indices concurrently as a single, wider data word. This coalescing can also be called *vectorizing* the memory access. In the previous work [2], we adapted a synthetic memory performance benchmark to show the sustained bandwidth to global memory of various devices (benchmark available at https://github.com/waqarnabi/mp-stream), and we showed that vectorization achieved up to 8.5× memory bandwidth increase over the baseline. For any application that is *memory-bound*, this improvement in memory bandwidth will translate to an improved throughput for the overall application. To complement the vectorized memory access, the application's datapath can also be vectorized, though one must ensure data hazards are avoided.

We will now discuss pipelining and vectorization optimizations in the context of the TyTra backend.

5. TyTra Backend (TyBEC) Compiler

The TyTra backend is designed to be compatible with a variety of frontend entry routes and is composed of a custom intermediate representation (IR) language, a parser, a scheduler, a cost/performance model, and finally an FPGA code generator.

5.1. The TyTra Intermediate Language. The TyTra Intermediate Language (TIR) is the interface provided by the TyTra backend, to which a number of possible frontends can be coupled, one of which was shown in Figure 1. However, for the purpose of this discussion, how one arrives at a TIR description of the problem is not relevant.

The TIR description of a problem lies halfway between the frontend and backend optimizations that together seek to identify the optimal design variant. *Optimal* in this context means that the kernel has been pipelined, ideally to achieve a throughput of one cycle per output, and then vectorized to go beyond this throughput until we either saturate the memory bandwidth (memory wall) or run out of FPGA resources (compute wall). There are some optimizing transformations that take place in the frontend DSE phase. These transformations relate primarily to finding specific computation patterns such as maps and folds and connecting them in a coarse-grained dataflow graph. Once the design variant generated by the frontend has been specified in the TIR, the backend optimizations can be applied, and FPGA implementation code can be generated. This context informs the underlying model of computation and specific requirements for our custom IR.

5.1.1. Model of Computation: Kahn Process Networks. The TIR syntax is quite similar to the LLVM-IR; however, its underlying model of computation is entirely different, as it models a *dataflow* machine. The Kahn process network (KPN) is a suitable abstraction to use, though we apply some additional requirements and constraints on it.

KPNs were first introduced by Gilles Kahn when presenting a simple language for parallel programming [20]. The use of the KPN abstraction for modelling architectures for FPGAs is not a novel concept (e.g., see [21, 22]). The key features of this abstraction, which make it very suitable for use as the underlying model for our IR, are as follows (direct quotes in the list are from [20]):

(i) Processes (or nodes) communicate via *unbounded* first-in first-out (FIFO) queues.

(ii) All processes run forever.

(iii) If any process were to stop for an external reason, the whole system will stop.

(iv) Communication lines (or edges or channels) are the only way in which processes communicate.

(v) The time taken by edges to transmit information can be unpredictable, but always finite.

(vi) At any given time, a process is either computing or waiting for data on one of its input edges.

(vii) A process can have its own memory or state, so it is a "function from the histories of its input lines to the histories of its output lines." This means nodes that fold (or reduce) information are possible.

(viii) Writes to a channel are *nonblocking*, while reads are *blocking*.

(ix) KPN process is *monotonic*: it "need not have all of its inputs to start computing, since future input concerns only future output." Monotonicity allows pipeline parallelism.

While these features of the KPN make it a suitable abstraction for our purpose of modelling pipeline parallelism on FPGAs, they cannot be used as is. The key departure required from this abstraction is replacing *unbounded* FIFOs (which are not possible in a real system) with *bounded* FIFOs. Additionally, nonblocking writes are not suitable with finite FIFOs, and we introduce blocking writes. However, the introduction of blocking writes and finite FIFOs can lead to deadlocks unless safe bounds for the sizes of the FIFOs are derived properly [23]. We derive safe FIFO bounds by statically scheduling all the nodes in our dataflow graph discussed in more detail later.

5.1.2. Syntax and Semantics of the TIR. The TIR is strongly and statically typed, and all computations are expressed using Static Single Assignment (SSA). The syntax is based on the LLVM-IR, but the semantics are built on top of a *streaming* paradigm, suitable for inferring pipelined datapaths on an FPGA target.

Static typing is a requirement for synthesizing an FPGA design at compile time. Strong and static typing together provide the basis for our static cost model that underpins the TyTra flow. The FPGA code-generator too requires explicit typing. TIR datatypes are mapped to LLVM datatypes wherever possible and follow the same general scheme for naming datatypes. However, LLVM-IR does not differentiate between signed and unsigned data which are required for TIR. Also, TIR allows custom and nonstandard datatypes, which in fact is one of the reasons for creating our own IR. TIR supports arrays and vectors, again following the syntax of LLVM-IR.

A design is constructed by creating a hierarchy of IR *functions*, which may be considered equivalent to *modules* in an HDL like Verilog. However, these functions are described at a higher abstraction than the *register transfer* level typical for an HDL. The TyTra backend parses the TIR description and extracts a dataflow architecture from it.

The TIR is neither a subset nor a superset of the LLVM-IR, but an independent language that is inspired by it. There are many features of LLVM-IR that TIR does not support, and at the same time, there are a number of extensions in the TIR that are alien to LLVM-IR. Under the hood, these languages have a fundamentally different view of the machine as discussed earlier, which is eventually the fundamental difference. Other ways in which TIR departs from the LLVM-IR semantics are: all variables are data *streams*, arguments represent ports for connectivity between peer or parent-child functions, there is custom instruction for *splitting* and *merging* nodes, there is a specialized syntax for creating *offset* streams for stencils, we can have custom datatypes with nonstandard widths, and there is an extended syntax to express the creation and consumption of data streams from/to memory. Further discussion of the TIR's syntax and semantics is outside the scope of this paper, but interested readers can refer to [24]. We also show TIR for two example problems later in Section 6.

5.2. Scheduling and Pipelining. Recall from the previous section that our extension to the KPN requires us to have bounded FIFOs, which are large enough to ensure there is no deadlock in the presence of blocking writes. Hence, although the hardware realization of our nodes is based on asynchronous hand-shaking with back-pressure (based on the AXI-Stream protocol), we do need to determine safe bounds for all inferred buffers. This is why static scheduling of the dataflow graph (DFG) is essential. Moreover, the TyTra flow is predicated on the availability of a performance model that can predict the latency and throughput of each node, which too requires a static scheduler, even if in practice there is no centralized scheduling controller in the synthesized FPGA circuit.

The scheduling algorithm of TyBEC is based on the KPN model of computation. The SSA syntax of the TIR lends itself to a straightforward extraction of the pipelined dataflow from primitive instructions in *leaf* functions. We have adopted an *As Soon As Possible* (ASAP) algorithm for scheduling the instructions. This can be suboptimal in terms of resource usage [25]. More sophisticated algorithms are possible, which exploit reuse of functional units across instructions to save resources. This however would require a fundamental change in the architecture from the current one with a distributed scheduling mechanism, to a centralized one where a controller would orchestrate the reuse of resources. This is a line of investigation we mean to pursue in the future.

Functions can be hierarchical as well, reflected by hierarchical nodes in the DFG. Each hierarchy is captured by the same extended KPN model that we have discussed earlier. The resultant DFG translates to dataflow *pipelines* on the FPGA, which are reflected in the generated HDL. This pipeline parallelism is, as we discussed earlier, an essential component of the FPGA-oriented optimizations we wish to

apply. Both the coarse-grained pipelining of kernels (hierarchical nodes) and fine-grained pipelining of instructions (leaf nodes) are achieved by this hierarchical scheduling process, illustrated in Figure 2. This aspect of our work may be contrasted with tools like SDAccel, where coarse-grained pipelines across kernels have to be explicitly modelled using OpenCL *pipe* semantics.

A finer level of pipelining, where we pipeline multicycle primitive instructions like floating point operations, is achieved by using pipelined functional units generated from the FloPoCo tool [18].

5.3. Vectorization. Vectorization is a well explored optimization for improving performance and has been explored for FPGAs as well. This is similar to, though not exactly the same as, the vectorization optimization in CPUs. The way we use this term, "vectorization" refers to *both* vectorization of the datapath, and the coalescing of memory accesses. In the TyTra backend, once the kernels have been pipelined, the design can be vectorized automatically. Memory-access vectorization results in coalesced transactions, which can help achieve operation at or near memory bandwidth saturation, as we have shown in our earlier work [2]. Vectorized datapath ensures that the overall throughput is not limited by the computation. Together, these automatic vectorization operations enable our backend to push the solution closer to both the memory and computation limits of the device, which is where we ideally want to be. The key novelty in our flow is the complete automation of the process of vectorizing the memory access and the datapath.

5.3.1. Automatic Detection of Vectorizable Loops in Serial Code. Vectorizing a loop by a factor N_V implies concurrent execution of N_V iterations of the loop. Loops can be vectorized only if they have been unrolled, and if there are no *loop-carried dependencies* with reaches smaller than the vectorization width N_V. The TyTra flow is based on extracting *map* and *fold* patterns from serial scientific code, as shown in Figure 1 and then *scalarizing* them before passing them on to the backend. *Scalarization* here implies replacing index-based array accesses with scalar variables, effectively subsuming loops that iterate over arrays. Since the (ostensibly) scalar variables actually refer to data streams, the semantics of the program are preserved, though that does require some metainformation to be carried through by the compiler, e.g., the size of the loops. An example of this transformation in Figure 3 shows two versions of an update() function from a scientific model. The code on the left is a conventional loop-based function, converted by our frontend (while we show the loop-based function written in C to emphasize the transformation, the frontend actually uses loop-based Fortran code as the source and emits scalarized C as shown.) to the scalarized version.

These transformations convert loops iterating over arrays to *scalar* kernels operating on *streams*. The advantage of this frontend transformation is that we get vectorization opportunities at the backend for free. Loop-unrolling is no longer required, as the frontend has ported the code into the streaming dataflow domain. More importantly, the frontend transformations guarantee that the kernels exposed to the backend do not have any loop-carried dependencies and thus are vectorizable to arbitrary widths. The TyTra flow is in fact conservative in exposing vectorization opportunities. This is because our flow only vectorizes *mappable* loops that have no loop-carried dependencies, whereas vectorization is possible even if they are there, as long as their reach is larger than the vector width.

5.3.2. Using a Cost Model for Identifying the Optimum Vector Width. The vectorization of kernels results in additional resource usage on the FPGA. By using our cost model, we can estimate the resource usage for different vectorization options. Then, we limit the vector width to the maximum that we can fit within the target FPGA's resources. Currently, our backend supports vector widths of 1, 2, 4, 8, and 16 (for OpenCL compatibility), but there is no innate reason for limiting our flow to these vectorization factors.

The cost model is discussed in detail in [24], but we present its brief outline here. Figure 4 shows how the cost model is used in our flow. The TIR description of the design variants is fed into the cost model, along with a description of the target. This description is in the form of its available resource, memory bandwidth profile, and the cost of various primitive instructions on that device. The cost model then accumulates the resource requirements for the entire device and also estimates its performance after scheduling all instructions and functions. It then estimates the performance of all variants (in case of the examples in this paper, variants are generated by varying the vectorization factor) and generates OCL-HDL hybrid code for the chosen one.

5.4. Generating OpenCL-HDL Hybrid Implementation for FPGA (F1) Instances on the Amazon Cloud. We considered a number of options for implementing the design generated by the TyTra flow on an FPGA, as shown in Figure 1, finally converging on an OpenCL-HDL hybrid for F1 instances on the Amazon cloud. Commercial vendors like Xilinx, Intel, and Maxeler all provide such a hybrid programming route. The HLS abstraction can be used to handle the *shell* logic conveniently, and kernel datapaths can be expressed at a lower abstraction (register-transfer level or *RTL*), e.g., in Verilog HDL. We avoid the need to generate complex RTL code for shell logic yet maintain much more control over optimizations for the kernel pipeline. Our hybrid approach is more amenable to performance and cost prediction than HLS-only routes. The generated kernel pipeline is more performance portable across FPGA vendor tools and devices, as no vendor-specific pragmas and optimizations are used. We do need to generate vendor-specific shell code, but that follows a standard template with little variation across designs.

Amazon's EC2 F1 instances on the cloud provide a suitable way of accessing the latest FPGA hardware as well as tools [26]. *FPGA AMI* machine images are available, which come prebuilt with FPGA development and runtime tools based on Xilinx's SDAccel and Vivado frameworks. These

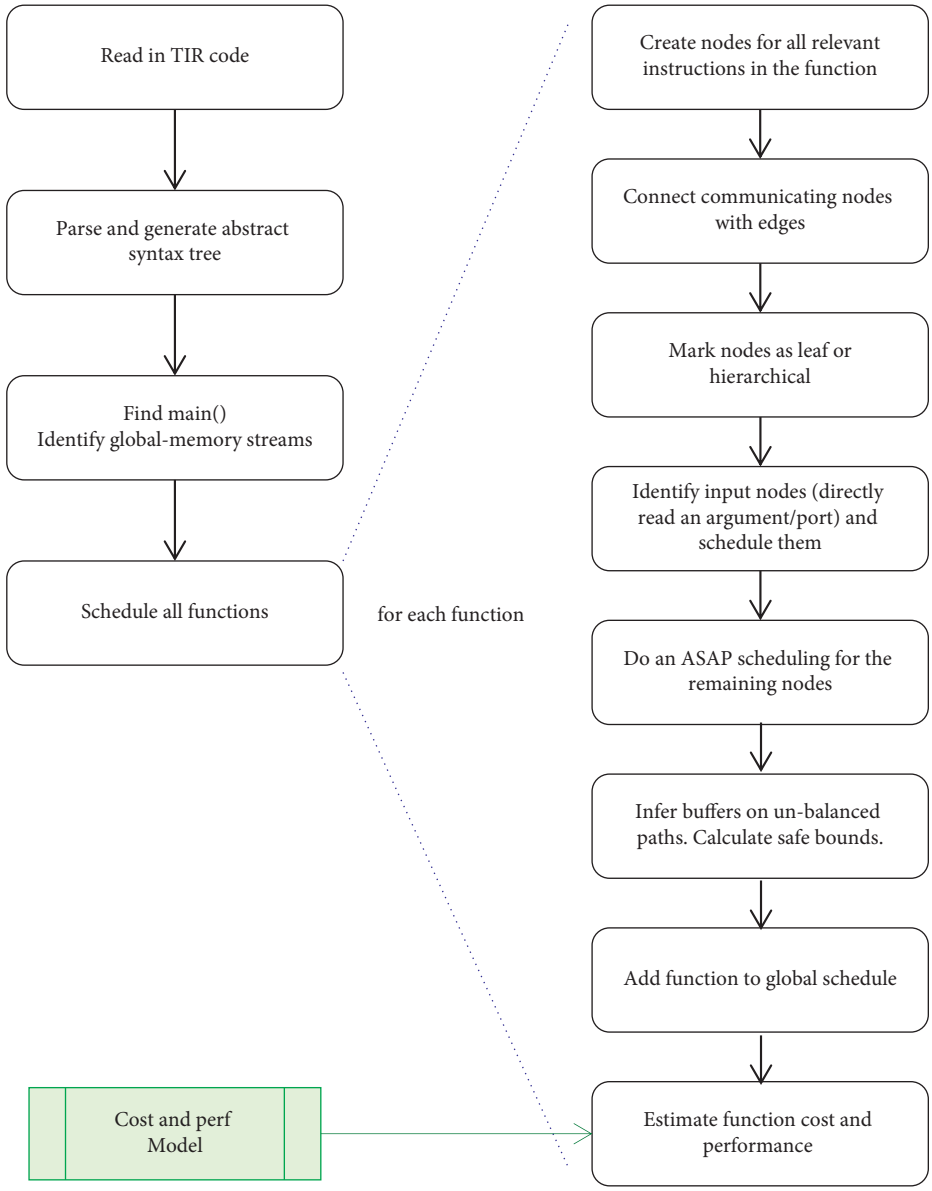

Figure 2: The TyTra backend scheduler. It reads in the TIR description of the problem, which has a syntax similar to LLVM-IR's, using the SSA (single static assignment) format. The output of the scheduler is the dataflow graph of the problem, with buffers inferred if needed (e.g., see Figures 6 and 9), which is then used to estimate performance, as well as generate synthesizable Verilog HDL.

```
/* The original function with loops */
void updates ( host_t *h, host_t *hzero, ...){
  for (int j=0; j<= ROWS-1; j++) {
    for (int k=0; k<=COLS-1; k++) {
      h[j*COLS + k]    = hzero[j*COLS + k]
                       + eta   [j*COLS + k];
      wet[j*COLS + k] = 1;
      if ( h[j*COLS + k] < hmin )
            wet[j*COLS + k] = 0;
      u[j*COLS + k] = un[j*COLS + k];
      v[j*COLS + k] = vn[j*COLS + k];
    }
  }
}
```

```
/* The "scalarized" function with streams */
void update_map_24 (float *h_j_k, float hzero_j_k, ...){
     *h_j_k = hzero_j_k+eta_j_k;
     *wet_j_k = 1;
   if ((*h_j_k)<hmin)     *wet_j_k = 0;
     *u_j_k = un_j_k;
     *v_j_k = vn_j_k;
   }
```

Figure 3: An illustration showing conversion of a loop-based function (left) to its *scalarized* version (right), where the "scalars" are effectively *streams* of data. Metainformation extracted by the compiler, e.g., the sizes of the streams, is carried through to the backend in order to preserve semantics.

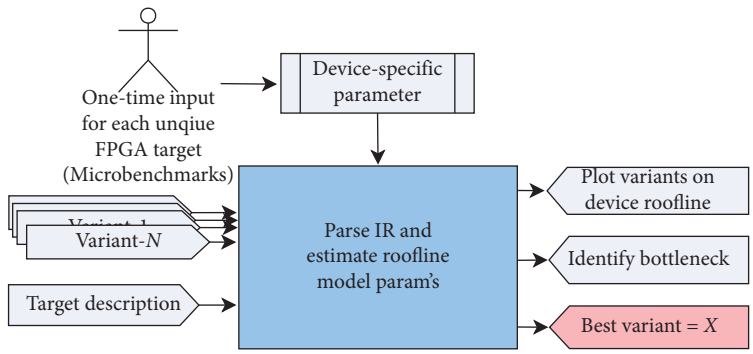

Figure 4: The use-case of the cost model that is integrated inside the TyTra flow's backend. It is used to estimate the resource usage and performance of variants being explored in the design-space.

tools provide various design-entry options. From our vantage point, the utility of these platforms is that we can work with the latest hardware and tool versions, and we can experiment with our hybrid HLS(OpenCL)–HDL(Verilog) flow.

To integrate HDL kernels with the shell logic provided by Xilinx's SDAccel tool, they need to be compatible with the AXI protocol: AXI4 for DDR read and write controllers; AXI4-Stream for transferring data streams to and from these controllers and also peer communication; and AXI4-Lite for control information exchange with the host [27]. The DDR (AXI4) and control (AXI4-Lite) interface logic is incorporated by using template code provided by SDAccel. This reduces the kernel pipeline compatibility requirement to the AXI4-*Stream* protocol for interfacing with the memory controllers. Figure 5 demonstrates this setup.

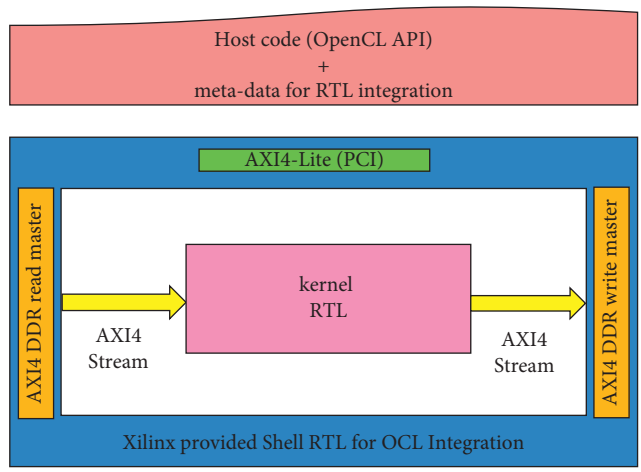

Figure 5: The hybrid OpenCL-HDL code-generation setup. TyTra generated kernel pipeline design in Verilog HDL is integrated with the OpenCL-based SDAccel framework, using AXI protocols.

6. Evaluation

We evaluate our approach with two examples, first through a synthetic barebones kernel and then on a scientific code simulating the *Coriolis* force.

6.1. A Synthetic Example. The simple, contrived example creates a coarse-grained pipeline with integer arithmetic operations. This translates to a single cycle throughput, and single path dataflow. The TIR and DFG of this problem are shown in Figure 6. Note that the DFG is generated automatically as part of the backend scheduling and code-generation. HDL code is also generated by the backend. It can be viewed by running the backend on the TIR (the prototype TyTra backend compiler is available at https://github.com/waqarnabi/tybec).

This illustrative example highlights the backend automatic pipelining and vectorization optimizations that are the focus of this work. We generated code for all vector widths supported by our flow, in order to demonstrate the effect of vectorization in this paper. In practice, we would follow the following simple algorithm for converging on the vector width:

(1) Create a design variant with the maximum allowable vector width (currently 16 words)

(2) Estimate resource utilization

(3) If estimated resources are more than available on target FPGA, step down to the next available vector width

(4) Repeat 1–3 until design is predicted to fit by the cost model (we aim to use less than 80% of target FPGA resources, as in our experience, beyond this threshold designs typically fail to synthesize)

The results of our experiments are shown in Figure 7. Since this is a small example, the maximum available vectorization factor of 16 was possible within the available resources, and that is the variant selected. Our results show an almost 4.2× speedup over the scalar baseline for the maximum vectorization. The speedup is sublinear and indicates a memory bottleneck. This bottleneck could be mitigated by using multiple memory banks if available.

The corresponding resource trade-off can be seen in Figure 8. Other than DSP units, all resources show sublinear scaling due to the almost uniform usage of logic in the *shell* of the design across all variants.

```
define void @kernel_A ( i32 %ka_vin0, i32 %ka_vin1, i32 %ka_vout) pipe {
  i32 %local1   = add i32 %ka_vin0, %ka_vin1
  i32 %local2   = add i32 %ka_vin0, %ka_vin1
  i32 %local3   = sub i32 %local1, %local2
  i32 %local4   = add i32 %local1, %local3
  i32 %ka_vout  = add i32 %local1, %local4}

define void @kernel_B ( i32 %kb_vin, i32 %kb_vout) pipe {
  i32 %kb_vout  = add i32 %kb_vin, %kb_vin}

define void @kernel_C ( i32 %kc_vin, i32 %kc_vout) pipe {
  i32 %kc_vout  = mul i32 %kc_vin, %kc_vin }

define void @kernel_D ( i32 %kd_vin, i32 %kd_vout) pipe {
  i32 %kd_vout  = add i32 %kd_vin, %kd_vin }

define void @kernelTop  ( i32 %kt_vin0, i32 %kt_vin1, i32 %kt_vout) pipe {
  call @kernel_A ( i32 %kt_vin0, i32 %kt_vin1, i32 %vconn_A_to_B)
  call @kernel_B ( i32 %vconn_A_to_B, i32 %vconn_B_to_C)
  call @kernel_C ( i32 %vconn_B_to_C, i32 %vconn_C_to_D)
  call @kernel_D ( i32 %vconn_C_to_D, i32 %kt_vout) }

define void @main () {
 %vin0  = alloca [NLinear x i32], addrspace(1)
 %vin1  = alloca [NLinear x i32], addrspace(1)
 %vout  = alloca [NLinear x i32], addrspace(1)
 %vin0_stream = streamread i32, i32*  %vin0
 %vin1_stream = streamread i32, i32* %vin1
 streamwrite i32 %vout_stream, i32* %vout
 call @kernelTop ( i32 %vin0_stream, i32 %vin1_stream, i32 %vout_stream)
}
```

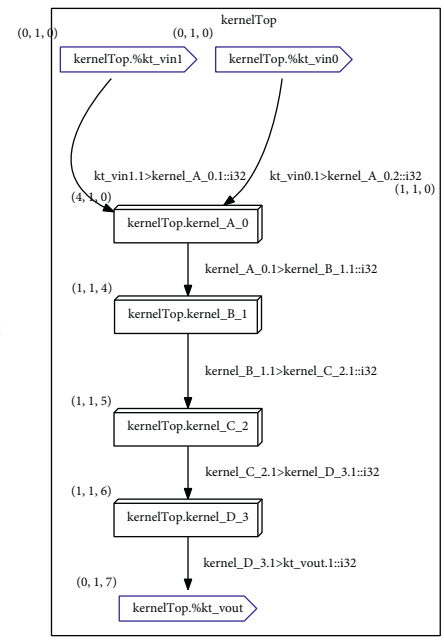

FIGURE 6: TIR code and DFG of the synthetic example. The TIR shows 4 kernels connected in a coarse-grained pipeline in a *top* kernel, which is connected to global memory streams in the *main* function. The DFG is generated by the backend from the TIR, and only the top-level kernel is shown here. The tuple of three integers with each node is the scheduling parameters (latency, firing-interval, and start-delay) inferred from the code and used by the backend for scheduling and RTL code generation.

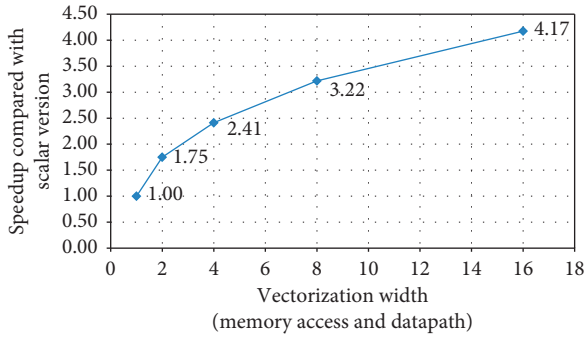

FIGURE 7: Speedup achieved over nonvectorized OpenCL baseline for various vector widths, for the first example. The TyTra solutions are OpenCL-HDL hybrids, and the complete host API, shell, and kernel code for all variants is generated automatically from TIR description.

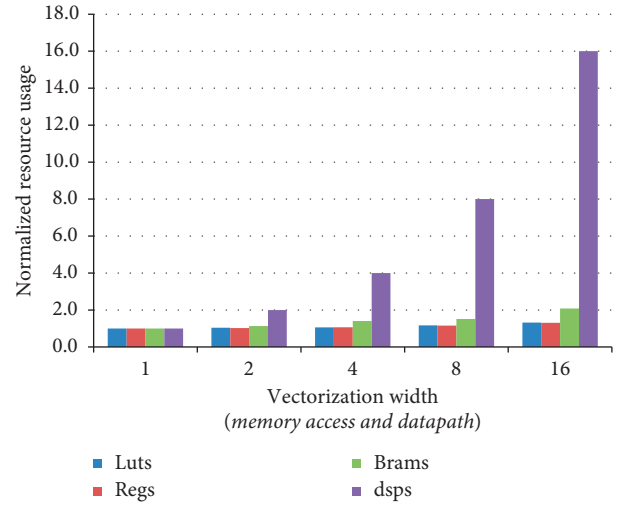

FIGURE 8: Resource usage for various OpenCL-HDL hybrid vectorized versions, normalized against resource usage for nonvectorized OpenCL baseline for the first example.

6.2. Simulating the Coriolis Force. This second example is based on Fortran code for modelling the Coriolis force that accompanies a text on ocean modelling [28]. The code predicts the pathway of nonbuoyant fluid parcels in a rotating fluid subject to the Coriolis force. The kernel is computed over a two-dimensional grid for a certain number of time steps. At each time step, the kernel reads the velocities and positions of each grid point and updates them. That is, at each time step, it reads 4 floating point numbers and writes 4 floating point numbers. The Fortran code is shown in Figure 9. The equivalent TyTra-IR code and dataflow graph (top kernel only) as generated by the TyTra backend are shown in Figures 10 and 11, respectively.

The TyTra backend generates a baseline RTL (and OpenCL wrappers) that is pipelined and without vectorization. It then generates vectorized versions as well, as long as the predicted cost fits in the target device. In this example as well, like the previous one, all possible vectorization factors up to 16 can be accommodated. The resource cost prediction of one design variant takes in the order of 0.1 seconds. This makes the design-space exploration fairly quick when we consider the vectorization optimization in isolation as there are a limited number of possible variants.

```
!time loop
DO n = 1,ntot
  !space loop
  DO i = 1, maxX*maxY
    ! velocity predictor
    un(i) = (u(i)*(1-beta)+alpha*v(i))/(1+beta)
    vn(i) = (v(i)*(1-beta)-alpha*u(i))/(1+beta)

    ! predictor of new location
    xn(i) = x(i) + dt*un(i)/1000
    yn(i) = y(i) + dt*vn(i)/1000

    ! updates for next time step
    u(i) = un(i)
    v(i) = vn(i)
    x(i) = xn(i)
    y(i) = yn(i)
  END DO
END DO
```

Figure 9: Fortran code of the Coriolis example.

```
;--velocity update kernel
define void @coriolis_ker0 ( data_t %u, data_t %v, data_t %un, data_t %vn) pipe {
  data_t %mul   = fmul data_t %u, ONE_MINUS_BETA
  data_t %mul1  = fmul data_t %v, ALPHA
  data_t %add   = fadd data_t %mul, %mul1
  data_t %un    = fdiv data_t %add, ONE_PLUS_BETA
  data_t %mul4  = fmul data_t %v, ONE_MINUS_BETA
  data_t %mul5  = fmul data_t %u, ALPHA
  data_t %sub6  = fsub data_t %mul4, %mul5
  data_t %vn    = fdiv data_t %sub6, ONE_PLUS_BETA
}
;--position update kernels
define void @coriolis_ker1_subker0 ( data_t %x, data_t %un, data_t %xn ) pipe {
  data_t %mul = fmul data_t %un, DT
  data_t %div = fdiv data_t %mul, 1000.0
  data_t %xn  = fadd data_t %x, %div
}
define void @coriolis_ker1_subker1  ( data_t %y, data_t %vn, data_t %yn) pipe {
  data_t %mul1 = fmul data_t %vn, DT
  data_t %div2 = fdiv data_t %mul1, 1000.0
  data_t %yn   = fadd data_t %y, %div2
}
;--top kernel connecting the two sub-kernels
define void @kernel_top(  data_t %u, data_t %v, data_t %x, ... ) pipe {
  call @coriolis_ker0 (data_t %u, data_t %v, data_t %un_local, data_t %vn_local)
  call @coriolis_ker1_subker0 ( data_t %x, data_t %un_local, data_t %xn)
  call @coriolis_ker1_subker1 ( data_t %y, data_t %vn_local, data_t %yn)
  data_t %un = load data_t %un_local
  data_t %vn = load data_t %vn_local
}

;--main: declared device global (DRAM) memory arrays, defines streams to/from it
;--    and "calls" the top kernel
define void @main () {
  %u  = alloca [SIZE x data_t], addrspace(1)
  ... ;--other device memory arrays
  %u_stream = streamread data_t, data_t* %u, !tir.stream.type !streamld, !tir.stream.size !SIZE
  ...;--other input streams from device memory
  streamwrite data_t %un_stream, data_t* %un, !tir.stream.type !streamld, !tir.stream.size !SIZE
  ...;--other output streams to device memory

  ;--call the top level kernel and pass it the streams and the constant
  call @kernel_top ( data_t %u_stream, data_t %v_stream, ...)
  ret void
}
```

Figure 10: TIR code of the Coriolis example. It shows 3 kernels connected in a coarse-grained pipeline in a *top* kernel, which is connected to global memory streams in the *main* function (not shown).

The actual performance for variants is shown in Figure 12. Note that there is an important difference between this and the first example. The first example had 2 integer inputs and 1 integer output. When the inputs, a total of 64 bits, were vectorized by the maximum factor of 16, it was still within maximum data width allowed by SDAccel (which is 1024 bits). This second example, however, had an input (and output) total width of 128 (32×4) bits, so it can only be vectorized up to a factor of 8, which is reflected in the results. In the future, we plan to incorporate multiple memory interfaces and banks into our design, which would allow us to exploit this vectorization feature to its full potential.

An interesting observation here is that the performance peaks at 2.7× baseline, at a vectorization factor of 4, with vectorization to a factor of 8 showing virtually no improvement. This example has a wider total input width of 128 (32×4) bits, as opposed to 64 bits for the previous one. Since the DDR memory bus for the target FPGA platform is 512 bits wide, it is saturated at a vectorization factor of 4 already. Using multiple concurrent memory interfaces and banks should allow us to go beyond this saturation limit.

Another observation is that the performance profile is virtually unchanged across different grid sizes and number of time steps. This shows performance gains of vectorization scaling well with the problem size.

The resource usage for all these variants is shown in Figure 13, which shows the expected increase for increasing the vectorization factor. Compared with the first example, this is a larger kernel with resource heavy floating point units, leading to the kernel having a proportionally larger share of the resources versus the shell logic. This is the reason vectorization scales up the resources much more than the first example.

7. Conclusion

FPGAs are fast-becoming mainstream accelerator devices for a variety of HPC applications. Writing optimized programs for FPGAs remains a challenge though, even with the availability of HLS tools. We are developing an optimizing compiler framework called *TyTra*, where we propose to use a combination of transformations and optimizations to automatically generate FPGA implementations from serial, legacy scientific codes. In this paper, we have presented two key optimizations that are part of this framework, pipelining and vectorization, the latter applied to both the external (DDR) memory accesses as well as the kernel datapath. We discussed briefly how we transform legacy serial Fortran code to kernels with *map* patterns, suitable for pipeline parallelism as well as vectorization. We highlighted our custom IR language-based backend that can be used to express the variants in our design space and which schedules computations based on an extended KPN-based machine model, finally emitting an OpenCL-HDL hybrid implementation. Evaluation of our approach on two examples showed performance gains between 2.7× and 4.2×. Extending our solution to exploit multiple memory banks concurrently can be reasonably expected to achieve further performance gains.

Exploiting such vectorization opportunities when accelerating HPC code on FPGAs is essential; otherwise, we may be operating far below optimal performance. Our flow, because it is based on a sophisticated frontend analysis and a cost model-based backend code-generation framework, can give these performance gains automatically.

There are a number of complimentary lines of investigation that we are still pursuing. Further optimizations at the frontend and backend of our framework in addition to pipelining and vectorization could further improve

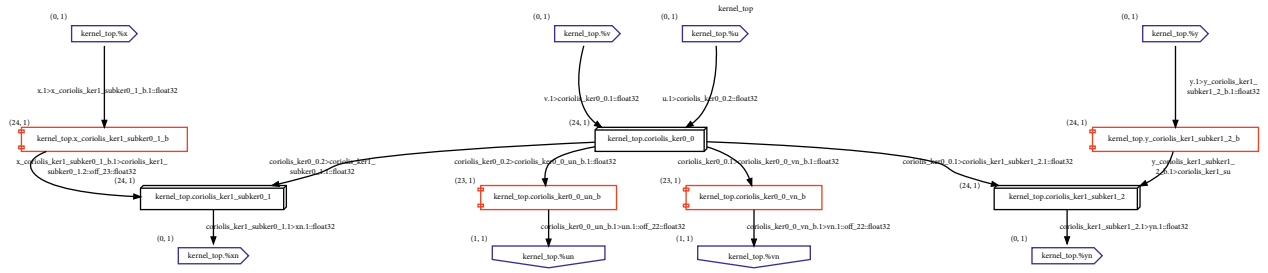

FIGURE 11: The DFG of the Coriolis example generated from the TIR, showing only the top-level kernel. The tuple of integers with each node is the scheduling parameters (latency and firing-interval) used by the backend for scheduling and RTL code generation. The red boxes (the boxes with two small stubs) are inferred buffers for synchronization and deadlock avoidance.

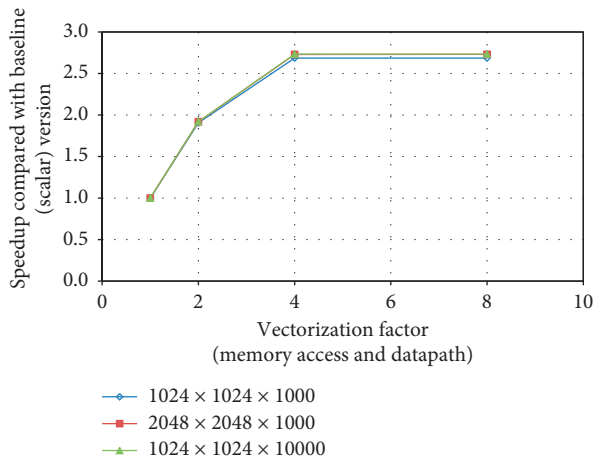

FIGURE 12: Speedup achieved over nonvectorized OpenCL baseline plotted against vectorization factor, for the second example (Coriolis). The TyTra solutions are OpenCL-HDL hybrids, and the complete host API, shell, and kernel code for all variants is generated automatically from TIR description. The speedup is calculated for 3 different grid sizes and time steps (legend shows *dimension1* × *dimension2* × *time steps*).

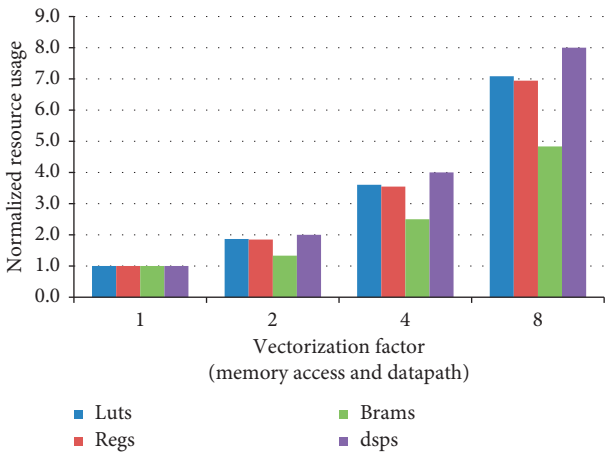

FIGURE 13: Resource usage for various OpenCL-HDL hybrid vectorized versions, normalized against resource usage for nonvectorized OpenCL baseline, for the second example (simulating the Coriolis force).

performance. Until now, we have made models from domain of fluid dynamics the focus for our test cases, and such models are innately amenable to finding mappable loops and hence to *streaming*. We are investigating extending the application domain to deep learning neural networks, which too lends itself to a streaming architecture, but requires closer integration of *folds* in addition to maps, which is an on-going work. We are also in the process of integrating all stages of the flow into a single framework, which we hope will contribute to mainstreaming of FPGAs as HPC accelerators.

Data Availability

The TytTra backend compiler has been deposited in a Github repository at https://github.com/waqarnabi/tybec. This is an on-going work, so the authors should be contacted if any issues.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors acknowledge support of the EPSRC for this work carried out as part of the TyTra project (no. EP/L00058X/1).

References

[1] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with FPGAs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16*, pp. 409–420, Piscataway, NJ, USA, November 2016.

[2] S. W. Nabi and W. Vanderbauwhede, "MP-STREAM: a memory performance benchmark for design space exploration on heterogeneous HPC devices," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 194–197, Vancouver, British Columbia, Canada, May 2018.

[3] F. A. Escobar, X. Chang, and C. Valderrama, "Suitability analysis of FPGAs for heterogeneous platforms in HPC," *IEEE*

Transactions on Parallel and Distributed Systems, vol. 27, no. 2, pp. 600–612, 2016.

[4] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. 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*, pp. 247–256, New York, NY, USA, February 2017.

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

[6] T. Czajkowski, U. Aydonat, D. Denisenko et al., "From OpenCL to high-performance hardware on FPGAs," in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 531–534, Oslo, Norway, August 2012.

[7] Xilinx, *The Xilinx SDAccel development environment*, 2014 https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[8] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[9] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala, "SparkCL: a unified programming framework for accelerators on heterogeneous clusters," CoRR, abs/1505.01120, 2015.

[10] J. Agron, *Domain-Specific Language for HW/SW Co-design for FPGAs*, pp. 262–284, Springer, Berlin, Heidelberg, Germany, 2009.

[11] C. Kulkarni, G. Brebner, and G. Schelle, "Mapping a domain specific language to a platform FPGA," in *Proceedings of the 41st Annual Design Automation Conference, DAC '04*, pp. 924–927, New York, NY, USA, June 2004.

[12] D. B. Thomas, S. T. Fleming, G. A. Constantinides, and D. R. Ghica, "Transparent linking of compiled software and synthesized hardware," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1084–1089, Grenoble, France, March 2015.

[13] M. Weinhardt and W. Luk, "Memory access optimization and ram inference for pipeline vectorization," in *Proceedings of the International Workshop on Field Programmable Logic and Applications*, pp. 61–70, Glasgow, UK, August 1999.

[14] C. Liao, D. J. Quinlan, T. Panas, and B. R. De Supinski, "A rose-based openmp 3.0 research compiler supporting multiple runtime libraries," in *Proceedings of the International Workshop on OpenMP*, pp. 15–28, Beijing, China, June 2010.

[15] D. Orchard and A. Rice, "Upgrading fortran source code using automatic refactoring," in *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools, WRT '13*, pp. 29–32, New York, NY, USA, October 2013.

[16] J. Overbey, S. Xanthos, R. Johnson, and B. Foote, "Refactorings for fortran and high-performance computing," in *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, pp. 37–39, St. Louis, MO, USA, May 2005.

[17] W. Vanderbauwhede and G. Davidson, "Domain-specific acceleration and auto-parallelization of legacy scientific code in fortran 77 using source-to-source compilation," *Computers & Fluids*, vol. 173, pp. 1–5, 2018.

[18] F. De Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[19] SDAccel optimization recommendations, 2019, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1207-sdaccel-optimization-guide.pdf.

[20] G. Kahn, "The semantics of a simple language for parallel programming," *Information Processing*, vol. 74, pp. 471–475, 1974.

[21] H. Nikolov, T. Stefanov, and E. Deprettere, "Modeling and FPGA implementation of applications using parameterized process networks with non-static parameters," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pp. 255–263, Napa Valley, CA, USA, April 2005.

[22] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: a FPGA based flexible coarse grain architecture design paradigm using process networks," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pp. 1–7, Long Beach, CA, USA, March 2007.

[23] T. M. Parks, *Bounded Scheduling of Process Networks*, Technical report, California University Berkeley Department of Electrical Engineering and Computer Sciences, Berkeley, CA, USA, 1995.

[24] S. W. Nabi and W. Vanderbauwhede, "FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis," *Journal of Parallel and Distributed Computing*, vol. 133, pp. 407–419, 2017.

[25] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "Scheduling algorithms for automated synthesis of pipelined designs on FPGAs for applications described in MATLAB," in *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, vol. 17, pp. 85–93, San Jose, CA, USA, 2000.

[26] Amazon EC2 F1 instances, 2019, https://aws.amazon.com/ec2/instance-types/f1/.

[27] AMBA, AXI and ACE Protocol Specification, 2011, https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf.

[28] J. Kämpf, *Ocean Modelling for Beginners: Using Open-Source Software*, Springer Science & Business Media, Berlin, Germany, 2009.