chips

MDPI

Article

Domain Specific Abstractions for the Development of Fast-by-Construction Dataflow Codes on FPGAs

Nick Brown †

EPCC, University of Edinburgh, Edinburgh EH8 9BT, UK; n.brown@epcc.ed.ac.uk
† Current address: The Bayes Centre, The University of Edinburgh, 47 Potterrow, Edinburgh EH8 9BT, UK.

Abstract: FPGAs are popular in many fields but have yet to gain wide acceptance for accelerating HPC codes. A major cause is that whilst the growth of High-Level Synthesis (HLS), enabling the use of C or C++, has increased accessibility, without widespread algorithmic changes these tools only provide correct-by-construction rather than fast-by-construction programming. The fundamental issue is that HLS presents a Von Neumann-based execution model that is poorly suited to FPGAs, resulting in a significant disconnect between HLS's language semantics and how experienced FPGA programmers structure dataflow algorithms to exploit hardware. We have developed the high-level language Lucent which builds on principles previously developed for programming general-purpose dataflow architectures. Using Lucent as a vehicle, in this paper we explore appropriate abstractions for developing application-specific dataflow machines on reconfigurable architectures. The result is an approach enabling fast-by-construction programming for FPGAs, delivering competitive performance against hand-optimised HLS codes whilst significantly enhancing programmer productivity.

Keywords: FPGAs; dataflow; high level synthesis; Lucent; programming models

Citation: Brown, N. Domain Specific Abstractions for the Development of Fast-by-Construction Dataflow Codes on FPGAs. *Chips* **2024**, *3*, 334–360. https://doi.org/10.3390/chips3040017

Academic Editor: Gaetano Palumbo

Received: 31 July 2024
Revised: 25 September 2024
Accepted: 29 September 2024
Published: 4 October 2024

Copyright: © 2024 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Field Programmable Gate Arrays (FPGAs) are extremely popular in embedded computing but have yet to gain wide acceptance for High Performance Computing (HPC) workloads. The flexible nature of FPGAs, where the electronics can be configured to represent many different circuits has been shown to provide significant energy efficiency benefits [1] and also improved performance for select applications [2] due to the specialisation that they can provide over and able general purpose CPUs and GPUs. Given that heterogeneous computing using GPUs has delivered very significant successes to HPC and the HPC community has accepted, at large, the role of accelerators, it is, therefore, somewhat surprising that whilst FPGAs have been highly successful in many other fields, they are yet to gain wide acceptance for HPC workloads. However, energy efficiency, an area where FPGAs often excel, is becoming of increasing importance to HPC as many supercomputing centres look to decarbonise their workloads.

There have, in fact, been numerous efforts exploring the role of FPGAs for HPC workloads [2–4], and whilst FPGAs have been demonstrated to benefit certain classes of HPC codes, for instance, those which are memory bound [5], a major limiting factor has been programmability. Whilst technologies such as AMD Xilinx's Vitis [6] and Intel's Quartus Prime Pro [7] have lowered the barrier to entry for software developers, enabling programmers to write their codes in C or C++ via High-Level Synthesis (HLS), it is still challenging to develop highly optimised codes for FPGAs. This requires significant expertise and time to undertake algorithmic level dataflow optimisations [8], resulting in significant performance differences between the initial and optimised versions of kernels on FPGAs which can take many months to tune.

One could summarise that current generation FPGA programming ecosystems enable software developers to write codes that are correct-by-construction, but not fast-by-construction. Instead, we believe the programming model should enable code to be written,

by construction, to suit the dataflow style. There should also be a rich set of abstractions so the compiler retains enough knowledge about a programmer's intentions which it can then use when making tricky, low-level decisions. In this paper, we explore abstractions where programmers are encouraged to structure their code in a style that provides a wealth of information to the compiler, with the aim of demonstrating whether this can enable software developers to write fast-by-construction codes for FPGAs. The ultimate aim of this work is to empower software developers to be able to easily and conveniently leverage FPGAs for their high-performance workloads.

This paper is structured as follows; in Section 2 we explore the programming problem further before introducing, in Section 3, our proposed execution model. Section 4 then describes the key abstractions in Lucent, our dataflow programming language used as a vehicle to explore the execution model and associated abstractions, before tieing these together and using them to describe an example in Section 5 and a discussion around the implementation of our compiler in Section 6. In Section 7 we study performance and programmer productivity for a number of kernels against other approaches, before describing related work in Section 8 and concluding in Section 9.

The novel contributions of this paper are as follows:

- Description of how custom dataflow computing machines can enable large-scale theoretical concurrency compared to the imperative Von Neumann model, and an Application Specific Dataflow Machine (ASDM)-based conceptual execution model.
- Exploration of an appropriate set of programming abstractions and language semantics enabling the convenient expression of high-level declarative algorithms that can be transformed into a dataflow architecture.
- Demonstration that a declarative language, using our abstractions and presenting the progression of time as a first-class concern, delivers comparable, sometimes better, performance to state-of-the-art approaches whilst being significantly simpler to program.

2. Motivation: The Challenges with Von Neumann

By empowering software developers to write FPGA code in C or C++ and synthesising this to the target hardware description language, HLS has significantly reduced the barrier to entry in obtaining code physically running on an FPGA. However, a very significant performance gap often exists between this initial version and the final code which has been optimised for a dataflow architecture [1,9,10]. This is because the optimisation steps required to close this performance gap are complex and time-consuming, requiring substantial expertise. The underlying issue is that not only are imperative languages such as C or C++-based upon an inherently theoretical sequential model, but they are also designed to map onto a Von Neumann execution model which is at odds with how reconfigurable architectures, such as FPGAs, operate. Considering the denotational semantics of imperative languages, the program's current state, σ, is represented as a function mapping storage locations (e.g., variable names) to their values. This is illustrated in Figure 1, and under the imperative model state transformations are undertaken via explicit language operations, such as assignment. This classical denotational representation of an imperative language's state means that at any one point in time, there will only be one single update to the state, in this example updating memory location a from 1 to $1'$.

Imperative languages were developed explicitly with the Von Neumann architecture in mind, where it is natural to represent such state, σ, as a table in memory and provide instructions that operate upon and manipulate this central table (memory). The Von Neumann view of fetching instructions based upon the program counter and then executing these forms the programmer's underlying imperative language execution model. However, this inherently sequential model is at odds with FPGAs because these are instead built on dataflow principles and when using HLS the tooling will represent this state as a dataflow graph which results in a significant disconnect between the hardware and programmer's execution model. Numerous approaches including DaCe [11], MaxJ [12], and visual tools like Simulink have been proposed where programmers explicitly design the dataflow

in terms of nodes running concurrently connected via streams that pass data from one to the next. However, this direct *stream-style* of programming is low-level and akin to programming Von Neumann architectures by working at the level of the program counter and individual registers. Consequently, HLS is still dominant when writing software targeting FPGAs.

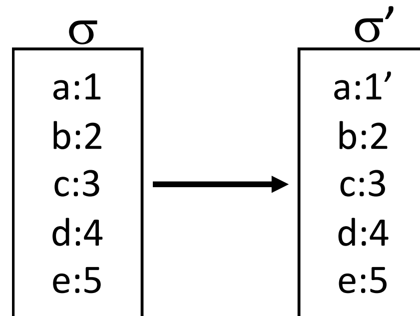

Figure 1. Illustration of update to program state, σ, for classical imperative languages which is fundamentally sequential.

3. Embracing Computational Concurrency via Custom Dataflow Machines.

Instead, our view is that for performance on a dataflow architecture, it is preferable to provide a naturally concurrent approach to the update of a program's state. At each unit of time, all elements are updated, and this is illustrated in Figure 2, which exposes a maximum amount of theoretical concurrency, as the entirety of the program's state is updated per unit of time. However, to realise this requires:

1. An architecture with significant amounts of raw concurrency.
2. A way of representing the state that supports efficient concurrent updates.
3. Programming abstractions that provide consistency across concurrent updates and avoid interference between them.

These three requirements are not easy for a traditional Von Neumann-based architecture such as a CPU or GPU to accomplish, and although the vectorisation of arithmetic operations on CPUs and GPUs could be argued as a limited example, there are constraints on this such as the operations supported or amount of flexible concurrency exposed.

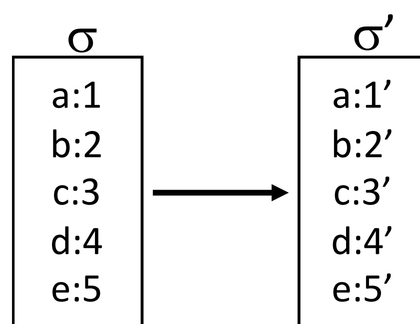

Figure 2. Illustration of update to program state, σ, for dataflow architectures which is fundamentally concurrent.

Dataflow architectures are better suited for providing this concurrency of state update, where the ability for operations to run in parallel provides the raw concurrency which addresses requirement one from the above list and the state can be represented as data flowing between the constituent operations meeting requirement two. Then, to provide an execution model for requirement three we considered research undertaken between the 1970s and early 1990s around general purpose dataflow machines [13]. Ultimately, these did not gain ubiquity, not least due to the inherent trade-offs and disadvantages with fixed general-purpose dataflow designs; however, the reconfigurable nature of FPGAs enables

the operations, flow of data, and granularity to be bespoke to a specific application. Put simply, a custom dataflow computing machine can be generated which is entirely suited to providing this concurrency of state update on an algorithm by algorithm basis.

3.1. Application Specific Dataflow Machine Execution Model

While the imperative language programmer only very rarely programs Von Neumann-based architectures at the assembly/machine code low-level, this still provides them with a conceptual execution model presenting a representation of *how* their code is executed. Likewise in our approach, to concurrently process and represent the program's state, we provide the FPGA programmer with an explicit dataflow execution model focused around building a custom computing machine that we call an Application Specific Dataflow Machine (ASDM). In the ASDM model all operations updating elements of state (and there might be many such operations for each element) are undertaken concurrently, with the flow of data between operations representing the overarching state itself. Consequently programmers view their code as being executed by a set of concurrently running operations, each consuming sequences of input data and transforming these to a sequence of output data. It is our assertion that by providing this high-level representation of how code will be executed then, given appropriate language constructs, a programmer's code can be effectively translated by the compiler to this custom form of hardware computing machine.

An illustration of the ASDM execution model is provided in Figure 3, where there are five elements of state, a to e for some example code. There are five operations, all running concurrently and external data, for instance, provided by the host, are streaming into the operations for states a and b, and state element e is streamed out as the overall result. Broadly, this corresponds to Figure 2, where each element of state is updated concurrently, but crucially instead of being represented by a central table in memory as per the Von Neumann model in Figure 1, the state is distributed across the components of the dataflow machine.

The programmer's conceptual execution model is of operations running concurrently (a producer) streaming their resulting element of state to all operations that consume that state element. Time is explicit and progresses in quanta, which is an abstract atomic unit of time, with the guarantee that an update at one time quantum will not arrive at a consumer operation until the next time quantum. The progression of time is a first-class concern to the programmer and data are never at rest, with the overarching state of the ASDM (and hence the program) continually changing from one time quantum to the next.

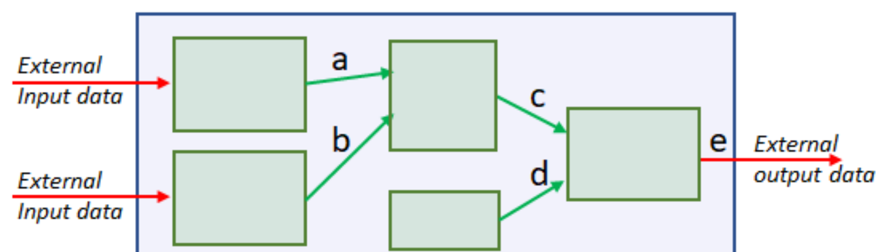

Figure 3. Illustration of programmer's abstract execution model based on ASDM, where a to e represent elements of state for some example code.

3.2. Building upon the Foundations of Lucid

One of the more well-known dataflow languages developed for general-purpose dataflow CPUs was Lucid [14], which contained numerous novel dataflow abstractions. Being a declarative rather than imperative language the programmer provides their logic in code with control flow inferred by the compiler. Whilst Lucid never ran on a real-world dataflow machine or FPGA, there have been some derivatives such as Lustre [15] and Esterel [16] developed in the 1990s which were aimed at programming FPGAs. However, these derivatives were targeted at the efficient implementation of electronic circuits,

rather than the algorithmic level of HPC software accelerated on FPGAs by exploiting and embracing dataflow.

When manually writing highly optimised HLS codes for FPGAs, we realised that the underlying concepts in Lucid were applicable and we found ourselves informally following these general underlying ideas to achieve the best performance. Consequently, whilst Lucid was designed for fixed, general purpose, dataflow machines, the abstractions that it provides were used as a starting point for our work here. Lucid itself is very much a product of the 1980s in its syntax and semantics, and whilst there are a large number of differences in our approach to Lucid, we are using the abstractions provided by Lucid as a foundation for our work, building upon these to suit the reconfigurable nature of FPGAs and providing rich information to the compiler.

4. The Lucent Dataflow Language

Building on our approach to providing the programmer with a conceptual execution model of *how* their dataflow code will run described in Section 3, we developed the Lucent dataflow language which aims to effectively support the concurrent state update of Figure 2. Each member of the state, known as a sequence because it continually changes over time, can only have one producer but multiple consumers to ensure consistency. The most natural way to represent this was by following the declarative paradigm where the programmer declares the logic of their algorithm and each individual member of state, and from this, the compiler can determine the exact control flow steps required. The entire update of the program's state occurs at each unit of time, known as a time quantum, and abstractly each quantum contains two parts, a work and commit phase. This is illustrated in Figure 4 for three time quanta, where during the work phase calculation operations are undertaken concurrently across the ASDM but these must all be completed globally before the commit phase makes these state changes visible. Such a two-stage approach ensures time quanta are atomic, which is important for consistency as it guarantees that values being worked with are those from the start of the time quantum rather than updates filtering through mid-quantum.

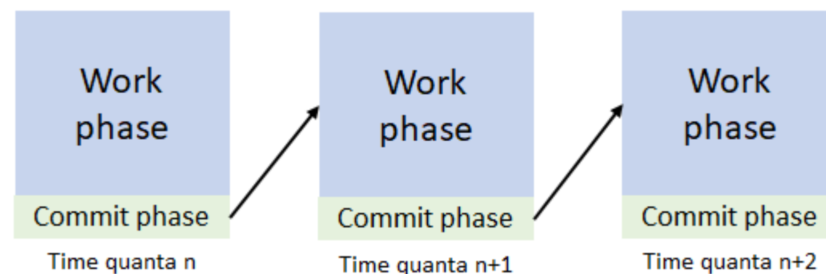

Figure 4. Illustration of the work and commit phase across time quanta, where the updates per quantum are not made visible until the commit phase.

The objective of Lucent is to provide a set of programming abstractions that conveniently map between the programmer's mathematical model (the sequences comprising the state that is being updated each time quantum) and their execution model (the physical realisation of this in hardware by the ASDM). Whilst it is not the purpose of a paper to give a full tutorial-like description, instead we use a simple factorial calculation as a running example to build up and highlight our key concepts and contributions.

4.1. Filters

Section 3.1 introduced the Application Specific Dataflow Machine (ASDM) execution model and described the concept of operations continually running and generating results each time quantum. From a code perspective, Listing 1 illustrates the first step in developing a factorial dataflow kernel with the filter, *mykernel*, which is marked as *external* meaning that it can be called from the host. A filter is a logical collection of sequence update declarations,

and in this example is defined as producing a sequence of output integers, one per time quantum, and the declaration *mykernel=12* results as per Figure 5 in the value *12* being written to the output sequence every time quantum until infinity. Even from this extremely simple example, it can be seen how the progression of time is a first-class concern, as the declaration defines the value of a sequence *over time*, i.e., here *mykernel* holds the value 12 over time.

Listing 1. Lucent filter that streams out 12 infinitely.

```
1  external filter mykernel:int() where:
2      mykernel=12
```

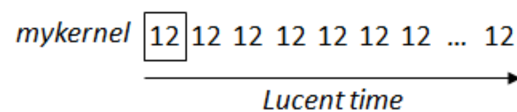

Figure 5. Values of the mykernel sequence over time (the boxed value denotes the initial value).

To express the factorial calculation the language must support calling other filters, working with input sequences of data, and undertaking conditionals, and these are illustrated in Listing 2. Marked *external*, the *mykernel* filter is the entry point, defined to produce a sequence of output integers, and accepts an integer input sequence *in*. There is a *factorial* filter in Listing 2 which is mocked up which does not yet implement the factorial calculation as further language constructs are still needed for that. This mocked-up *factorial* filter accepts an *n* input sequence and line 2 contains a conditional examining, at every time quantum, the value held in the *in* sequence. If this is end-of-data, *EOD*, then that is written to the output; otherwise, the result of the *factorial* filter is written out, whose input argument is the sequence *in*.

Listing 2. Example of working with input streams, calling a filter, and conditionals.

```
1  external filter mykernel:int(in:int) where:
2      mykernel=if (in == EOD) then EOD else factorial(in) fi
3
4  filter factorial:int(n:int) where:
5    seq:int=n*10
6    factorial=seq
```

Filters are similar to functions in other languages and provide convenience, for example in Listing 2 the same result could be achieved at line 3 without the *factorial* filter by the expression *in*10*. The declaration of *seq* at line 5 is contrived but illustrates an internal sequence, which is declared and then used subsequently in the filter. This highlights another important aspect because Lucent is a declarative, rather than imperative language and the ordering of lines 5 and 6 does not matter because, unlike the imperative model, the programmer is not explicitly specifying the order in which their code is running based on program order. Instead, this is based on declarations updating sequences which comprise the state of the program. The programmer is designing their code and relying on the compiler to undertake the low-level decisions based on the rich source of information, such as instruction scheduling, with the guarantee that the compiler will organise operations such that the code makes progress each time quantum.

EOD is a special token used to signify the end of data and this is automatically included at the end of all external input sequences. In Lucent we define a filter as terminating when *EOD* is written to its output, and writing *EOD* to the output of the external filter will terminate the entire kernel. Unless there is an explicit check for *EOD* by the programmer, any operation involving *EOD* will produce *EOD* as the result which enables convenient propagation of the token. Whilst the check for *EOD* is good practice at Line 2, strictly speaking it is not needed because *EOD* in the *in* sequence will flow through the *factorial* filter.

4.2. Manipulating Values over Time

A major contribution of Lucid was the followed-by, *fby*, operator. This defines that a value is followed by another value in the subsequent time quantum, enabling one to manipulate values over time. For instance, *my_seq:int* = 0 *fby* 1 *fby* 2 declares that *my_seq* is a sequence of type integer which is the value *0* at time quantum zero, value 1 at time quantum one, and then value 2 at every subsequent time quantum. The statement *my_seq:int* = 0 *fby my_seq* + 1 declares that *my_seq* is 0 at the first time quantum and then *my_seq* + 1 in subsequent time quantum, effectively a counter, and this is illustrated in Figure 6.

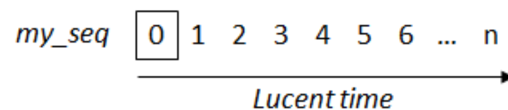

Figure 6. Value held in the *my_seq* sequence based upon the *my_seq:int* = 0 *fby my_seq* + 1 declaration, implementing a counter.

We adopted *fby* in Lucent, and in this example of a counter, integer addition requires one cycle, so whilst the value at one time quantum relies on the value at the previous time quantum (because it is an accumulation), it can be achieved within a single cycle. However, if *my_seq* were a different number representation, such as a floating point, then several cycles might be required for the addition. For example, the AMD Xilinx Alveo U280 single precision floating point requires eight cycles, and results in a spatial dependency between iterations that stalls progress. Effectively, if written naively as a single addition then the pipeline must wait those eight cycles before progressing onto the next number, or else the accumulated value will be inconsistent as the addition at the previous iteration is not yet completed.

Based on the data type, the Lucent compiler will detect such potential issues and generate different target codes to ameliorate them. For example, a common optimisation for spatial dependencies [17] is to replicate the pipeline and work in cycles of *n* iterations, where each replica is updating a separate variable which is all combined at the end. By working in cycles, one avoids these spatial dependencies each iteration; however, low-level optimisations such as these add additional complexity to the HLS code. By contrast, this is hidden from the Lucent programmer and in our approach can be handled by the compiler due to the high-level declarative nature of Lucent.

4.2.1. Intermediate Dataflow Operators

In Lucid and derivatives such as Lustre and Esterel, there were numerous additional dataflow operators hard-coded into the language which enabled programmers to manipulate values over time in different ways. However, the language constructs described in Sections 4.1 and 4.2 contain all the necessary building blocks for encoding these as Lucent filters, rather than as part of the core language. This not only keeps the language implementation simpler but furthermore illustrates that with a small number of operators, it is possible to build more advanced intermediate dataflow support.

One such operator is *as soon as*, which checks the value in a boolean stream at each time quantum and will stream out the value in the input sequence as soon as this is true followed by *EOD*. This construct is needed for the factorial calculation to generate the result once the calculation has reached its conclusion. Listing 3 illustrates the factorial filter implemented in Lucent using *asa*, although there is still one omission that means it will not yet quite work as required. At line 13 of Listing 3, if the value held in the *expr* sequence is true then the corresponding element in the *input* sequence at this time quantum is streamed out followed by *EOD*.

Listing 3. Lucent implementation of the as-soon-as filter.

```
1  external filter mykernel:int(in:int) where:
2    mykernel=if (in == EOD) then EOD else factorial(in) fi
3
4  filter factorial:int(n:int) where:
5    ctr:int = 1 fby ctr + 1
6    fac_calc:int = 1 fby fac_calc * ctr
7    factorial = asa(ctr==n, fac_calc)
8
9  filter asa:int(expr:boolean, input:int) where:
10   asa=if (expr == EOD or input == EOD) then:
11         EOD
12       else:
13         if (expr) then input fby EOD else NONE fi
```

If *expr* is false at Line 13 of Listing 3 then the *NONE* token is written. This is the second special token (the other being EOD) and is used to represent an empty value. It is important because progress must be made each time quantum, and therefore, some value must be present in the sequence. However, there are situations, as with the *asa* filter, where at times this should be an empty value to be ignored by the rest of the code as a no-operation. It can be thought of as a bubble in the data being streamed which, similarly to the *EOD* token, will be propagated by arithmetic and logical operators and ultimately ignored. This *NONE* token is not present in Lucid or other derivatives but was required as we formalised the rules around state progression for the execution model.

4.3. Nested Time via Time Dimensions

The language constructs described so far have assumed that time is constant and progressing at the same rate for all filters. However, this can be overly simplistic and exhibit several disadvantages. Firstly, it can be desirable for a filter to pause an input sequence and consume the same value over many time quanta, for instance, iterating a calculation over some input number before continuing and this is required for our factorial calculation. Secondly, a filter might need to build up some internal state and then reset it at some point later before further computations. Thirdly, an algorithm might conceptually work in iterations, calculating a series of values per iteration which are fed to the next iteration as a block.

To handle these short comings the original Lucid and derivatives contained numerous operators and capture rules for pausing and restarting streams of data. However the semantics of these were rather complex and in the 1990s the concept of time dimensions was proposed in [18,19]. Time dimensions meant that the progression of time could vary between program components and in [19] the authors postulated that this could significantly simplify Lucid by making redundant numerous operators. This time dimension work was only ever theoretical, and as far as we are aware, never implemented in a programming language, but we have built upon this to enable Lucent to address the limitations described above.

Time dimensions can be imagined as the nesting of state, and this is illustrated in Figure 7, where for each time quantum of *mykernel* the *nested* filter runs through until completion. Between time quanta of *mykernel*, the *nested* filter is reset and starts afresh, and for each time quantum of *mykernel* the same input values are provided from *mykernel* to *nested* filter. The factorial example requires pausing the input sequence of numbers, iterating over a specific value to calculate the result before moving onto the next input. This exhibits the first and second challenges described above, pausing the input sequence and maintaining intermediate results, and Listing 4 provides the complete factorial example illustrating the concept of time dimensions in Lucent. A time dimension *d* is defined and the filter *factorial* marked as being part of this. If the input sequence *in* to the *factorial* filter contains *1,2,3,4,5,EOD* then without time dimensions the input to *factorial* would progress

each time quantum until at time quantum six *EOD* would be passed, and this is the problem with the preceding factorial version in Listing 3.

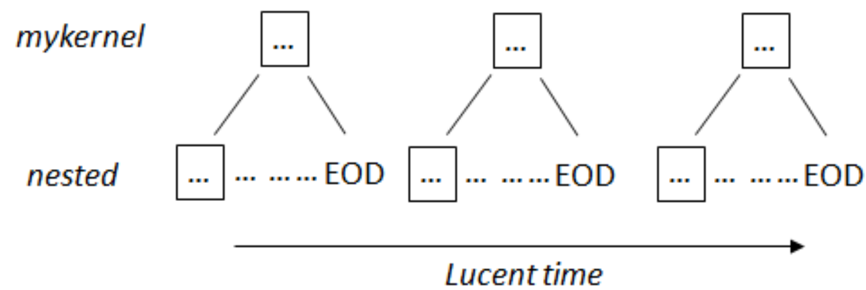

Figure 7. Example of progression for two filters, *mykernel* and *nested*, where the *nested* filter is in separate time dimensions and runs until completion for each time quanta of the outer *mykernel* filter.

The time dimension on the filter *factorial* solves this because, from the perspective of the *mykernel* filter operating in the default time dimension, the *factorial* filter will run to completion in one atomic time quantum. However, in each of these outer quanta, the *factorial* filter runs for *d*, inner dimension time quanta. Consequently, the kernel's output would be *1,2,6,24,120,EOD*, with each number representing the value streamed from the *factorial* filter at the end of its execution in each outer, default, time quantum. Incidentally, whilst *EOD* is streamed from the *factorial* filter for each of the outer time quanta, the semantics of the language means that the outer time dimension will consume this.

Listing 4. Example of time dimension applied to factorial filter.

```
1  timedimension d
2
3  external filter mykernel:int(in:int) where:
4    mykernel=if (in == EOD) then EOD else factorial(in) fi
5
6  filter d.factorial:int(n:int) where:
7    ctr:int = 1 fby ctr + 1
8    fac_calc:int = 1 fby fac_calc * ctr
9    factorial = asa(ctr==n, fac_calc)
```

From the perspective of the default time dimension, *factorial* is running till completion in one single time quantum. Therefore, it consumes exactly one value from the input sequence and streams out one resulting value per outer time quantum. Within the *d* time dimension, the *factorial* filter will run for a number of *d* time quanta, and the same *in* value streamed in for each of these *d* quanta until the outer, default, time dimension progresses to the next quantum. The semantics of the filter itself are unchanged, for instance, it will terminate when *EOD* is written to its output, and such termination of filters in a time dimension will signify one atomic time quantum from the perspective of the caller, here the default time domain. The *factorial* filter will restart between outer time quanta, resetting all internal states.

4.4. Exploiting the Type System

Lucid and derivatives were implicitly typed; however, FPGAs require that the type is known in order to allocate appropriate resources. Furthermore, explicit typing enables the convenient customisation of data representation and provides more information upon which the compiler can operate. Therefore, an explicit typing approach is adopted in Lucent, and Listing 5 illustrates an example of exploiting the type system for the factorial example, where at line 1 the programmer defines a type variable, *longlong*, to be an integer of 128 bits. There are two aspects to highlight, firstly, the existence of type variables, which are similar to typedefs in C, and can be used as any other type in the language. Secondly,

type annotations, which are provided as arguments in square braces to the type and will override default settings. The *mykernel* external filter of Listing 5 at line 4 provides an example, where the programmer is supplying additional annotations to the type of the output and input sequences. This hints where in external memory the data should be located, DDR-DRAM has been selected for the output stream and bank 2 of HBM for the *in* input stream.

It can also be seen in Listing 5 that the *factorial* filter outputs a sequence of type *longlong*, the type variable defined at line 1. There are implicit type conversions occurring between 32-bit integers and longlong, for example, streaming of *longlong* from the *factorial* filter at line 10, which is written at line 5 to the output of the *mykernel* filter of type int.

Listing 5. Type system to specialise data representation.

```
1   type longlong:=int[precision=128]
2
3   timedimension d
4   external filter mykernel:int[storage="dram"](in:int[storage="hbm", bank=2]) where:
5     mykernel=if (input == EOD) then EOD else factorial(in) fi
6
7   filter d.factorial:longlong(n:int) where:
8     ctr:int = 1 fby ctr + 1
9     fac_calc:longlong = 1 fby fac_calc * ctr
10    factorial = asa(ctr==n, fac_calc)
```

The language provides primitive types including integers, floating point, fixed point, and boolean with the programmer able to specialise details of these as necessary. A benefit of type variables, such as *longlong* in Listing 5, is that the type configuration is in one single place and changing these details is trivial for the programmer. Whilst there are mechanisms in HLS, such as typedefs or templating, it can still require explicit programming, such as converting from floating to fixed point, which can increase code complexity. Furthermore, HLS often requires the use of pragmas to specialise in specific decisions around types that can be unwieldy. Instead, in Lucent this is all handled by the type system.

Leveraging the type system also enables us to address one of the potential limitations of the ASDM execution model, namely that there is only one streaming output per filter. This restriction simplifies the language's semantics but can be overly restrictive where programmers might wish to stream out multiple values from a filter. Instead of modifying the execution model which would add complexity, the type system is used. Consequently, a *multistream* type is provided, which packages multiple sequences together. From the perspective of the execution model, language semantics and the compiler, there is still only one output being streamed from the filter, but crucially because this is typed as a *multistream* it is composed of multiple sequences.

Listing 6 illustrates the use of the *multistream* type for the factorial example, where the *factorial* filter streams out both the factorial result and the original input number that has been factorised. At line 8 the *factorial* filter is declared to stream out a *multistream* of two numbers, *val* which is the longlong factorial result and *num* which is a (32-bit) integer. Members can be referred to by name or positional index, for instance, the declaration *factorial->val* at line 11 declares the value of the *val* member of the multistream and *factorial[1]* at line 12 the second member. Members can be written to and read from independently.

Listing 6. The multistream type enables a filter to stream out multiple sub-streams.

```
1   type longlong:=int[precision=128]
2
3   timedimension d
4   external filter mykernel:int(input:int) where:
5     result:multistream[longlong, int]=factorial(input)
6     mykernel=if (input == EOD) then EOD else result[0] fby result[1] fi
7
8   filter d.factorial:multistream[val:longlong, num:int](n:int) where:
9     ctr:int = 1 fby ctr + 1
10    fac_calc:longlong = 1 fby fac_calc * ctr
11    factorial−>val=asa(ctr==n, fac_calc)
12    factorial[1]=asa(ctr==n, n)
```

4.4.1. The List Type

Multi-dimensional lists are also provided as a type and Listing 7 illustrates their use, where filter *example1* continually streams out a list of four members, 1, 2, 3, 4, each time quantum. The *example2* filter uses an internal sequence, *a* to store the list with the *storage* type annotation to control which on-chip memory location is used for data allocation. In this second example, the *at* operator selects the element at list index 1 which is then streamed out. The third example in Listing 7 implements a simple shift buffer where the tail, *tl*, operator creates a new list with every element apart from the first, and list concatenation, *::*, combining the two lists. In this example, the tailed list is combined with a list containing the value in the *input* sequence at that time quantum. Whilst Lucent's semantics are that lists are immutable, with a new list created whenever modifier operators such as tail or concatenation are issued, the compiler will optimise this to avoid list copying and allocation ensuring such operators are implemented efficiently and minimise the storage space required.

Listing 7. Example of the list type and operators.

```
1   filter example1:list[int,4]() where:
2     example1=[1,2,3,4]
3
4   filter example2:int() where:
5     a:list[int, 4, storage=''uram'']=[1,2,3,4]
6     example2=at(a, 1)
7
8   filter example3:list[double,3](input:double) where:
9     example3=if (input == EOD) then EOD else tl(example3) :: [input] fi
```

There are two reasons why the size of lists must be explicit; first, the tooling needs to know the data size to map to the appropriate number of on-chip memory resources such as BRAM blocks; second, it provides rich information which the compiler can use to optimise access. This second point is important because on-chip memories are, at most, dual-ported with a maximum of two concurrent accesses per cycle. When encountering this, HLS programmers must explicitly partition memory and determine the shape and the size of these. This adds complexity and by contrast due to the higher level nature of the programmer's code, Lucent can identify memory access patterns during compilation and undertake such optimisations.

Logical or arithmetic operations performed on lists will be undertaken concurrently, in the same time quantum, for each pair of elements and this enables a convenient approach to vectorisation of operations. For instance, undertaking a multiplication over two lists of eight double floating-point elements would undertake eight double-precision floating-point multiplications per time quantum, resulting in eight double results in a single quantum. This use of lists is a convenient way of parallelising operations over data.

An example is illustrated in Listing 8, where the filter is defined to accept two streams, *a* and *b*, which are typed to be lists of doubles of size 8. Consequently, each time quanta a list with eight doubles will be streamed in and the operation, $mykernel = a * b$, at line 2 will undertake concurrent multiplication of all pairs of elements, resulting in a list of eight doubles streamed out per time quanta. Vectorisation via lists can be used throughout Lucent code, and as an aside the compiler will automatically pack input data from the host into these lists and unpack output data, so no host side data changes would be required in Listing 8. Multi-dimensional lists also enable non-concurrent accesses on external data, where these can be declared as lists and manipulated via list operators, for example using *at* to retrieve elements at specified indexes from external memory.

Listing 8. Concurrency via list vectorisation.

```
1  external mykernel:list[double,8](a:list[double,8], b:list[double,8]) where:
2    mykernel=a * b
```

4.4.2. Generics

The as-soon-as, *asa*, filter was illustrated in Section 4.2.1 and describes a filter that provides intermediate dataflow support built upon the foundations in the core language. There are a variety of these filters in the *core* module, which is automatically imported into a users code. Furthermore, there are several other modules which provide utility functionality such as *euclid* which provides the Euclidean geometry abstraction atop of sequences of data and *buffer* which provides different types of data storage and buffering.

As described, the type of data and static size of lists must be known at compile time so that appropriate structures on the FPGA can be synthesised. However, this static requirement severely limits the ability to write reusable generic filters, for instance, across different types. To this end, we have added support for several generic type abstractions in Lucent and these are illustrated in Listing 9.

The first of these is the ability to accept sequences that are typed generically. For example, in Listing 9 the code for the *example1* filter accepts the sequence *a* which it can be seen has a type of <*T*>. The angular braces indicate a generic and the token, in this case *T*, the generic type. The type of the *b* input sequence is a generic called V. These generic types can then be referred to in the filter, for instance, the type of the *c* sequence is T and the filter streams out a sequence of type T. These types are resolved by the compiler for calls into the filter, for instance, *example1(1,2)* would set *T* and *V* to both be 32-bit integers, whereas for *example1(1.9,2)* then *T* is a double precision floating point number and *V* is a 32-bit integer because the compiler defaults to integer constants being 32-bit and float constants are double precision. Separate concrete implementations of this filter will exist for different permutations of input sequence types. It is only after a concretisation pass by the compiler that checks are performed to ensure that operators between sequences are permitted, for instance, raising an error if the type of *T* and *V* are incompatible in the addition of line 3 in Listing 9.

Listing 9. Examples of generics in Lucent.

```
1  filter example1:<T>(a:<T>, b:<V>) where:
2    c:T=a*7
3    example1=c+b
4
5  filter example2:<T><<N, S>>(in:list[<T>, 4]) where:
6    data:list[<T>, N, storage=S]=in
7    example=at(data, 2)
```

The *example2* filter in Listing 9 illustrates the passing of constant values to a filter. Here, the double angular braces indicate that *N* and *S* are constants that will be provided to the filter by the caller. The primary objective of this is to enable the generic sizing of lists,

but these can then be used freely in the filter body more generally. For example, the call *example«4, "lutram"»([1.0, 2.0, 3.0, 4.0])* would concrete *N* to be the constant 4, *S* to be the string lutram, and the generic type *T* is a 32-bit integer. Effectively, for *example2*, this will size the list *data* to be four elements of type 32-bit integer and allocated in LUTRAM rather than the default BlockRAM.

It is also possible to pass filters as arguments to other filters as first-class values, again these are all concreted during compilation with a separate implementation of the filter for each permutation of generics. This passing of filters is useful as it enables the programmer to inject some specific functionality into a generic filter.

5. 2D Jacobi: Bringing the Concepts Together

Our approach encourages programmers to express their dataflow code in a high-level manner that can be efficiently transformed to ASDM by the compiler. Listing 10 illustrates solving LaPlace's equation for diffusion in 2D using Jacobi iteration. From the perspective of the outer time dimension, an entire Jacobi iteration operates over all the data within a single outer atomic time quantum. Hence, the use of the *d* time dimension on the *iteration* filter in Listing 10, and it can be seen that the *input* and *data* sequences are decorated with the *d* time dimension also. This places them within the *d* time dimension, where progression of that dimension will progress these sequences, and from one outer time quantum to the next these sequences will reset to the beginning.

Listing 10 also illustrates flexibility around types and numeric representation, where at line 4 we define our own bespoke fixed point type, *myfixed*, of size 28 bits and 20 bits allocated for the fractional component. The programmer has written the *jacobi* external filter to represent numbers in single precision floating point, with the *myfixed* type used for the Jacobi calculation and shift buffer. Mixing these datatypes is allowed and numeric conversion is handled by the compiler and ultimately maps to the corresponding data format conversion constructs in HLS. Whilst one could in theory support any arbitrary primitive types, a current restriction is that these must correspond to a type in HLS. For example, HLS only supports half, single and double precision floating points, this is also supported in Lucent, but because of this restriction, other floating point precision such as quarter or quadruple are not supported.

Listing 10. Illustration of solving LaPlace's equation for diffusion in 2D using Jacobi iteration.

```
1   import buffer
2   import euclid
3
4   type myfixed:=fixed[precision=28, fraction=20]
5   timedimension d[streamsize=16384]
6
7   filter d.iteration:myfixed(in:myfixed) where:
8     iteration=calc(shiftbuffer<<128, 128>>(in))
9
10  filter calc:myfixed(in:list[myfixed,3,3]) where:
11    calc=if (in == EOD) then:
12           EOD
13         elif (ishalocell<<128,128>>(in)) then:
14           at(in,1,1)
15         else:
16           0.125 * (at(in,0,0) + at(in,0,1) + at(in,0,2) + at(in,1,0) + at(in,1,2) + at(in,2,0) + at(in,2,1)
                 + at(in,2,2))
17
18  external filter jacobi:float(d.input:float) where:
19    ctr:int=0 fby ctr+1
20    d.data:float=iteration(input) fby iteration(data)
21    jacobi=if (ctr == 10000) then unpack(data) fby EOD fi
```

It can be seen in line 20 that there is a dependency on the *iteration* filter, where results from one call to *iteration* are passed to the next. The compiler can efficiently implement this using ping-pong buffers and *streamsize* must currently be provided to the time dimension definition as it defines the amount of temporary storage, in number of elements, required (in future we hope to remove the need for this). The *unpack* filter (located in the always imported core module) called at line 21 unpacks the *data* sequence into the current, outer, and time dimension, streaming results out after 10,000 completed Jacobi iterations followed by *EOD* to terminate.

It should be noted that the *shiftbuffer* and *ishalocell* filters are provided in the *buffer* and *euclid* modules, respectively, with the *shiftbuffer* filter implementing a 2D shift buffer following similar principles to the 1D shift buffer, demonstrated in Listing 7. As described in Section 4.4.2, the <<*128,128*>> syntax passes specific concrete constants, in this case *128* to generic filters and this is required in this case as the size of the lists implementing the 2D shift buffer must be known at compile time.

6. Language Implementation

Whilst our main objective in this paper is to describe the programming abstractions that underlie our approach, the compiler itself is important to highlight, albeit briefly, as it transforms the programmer's logic into hardware. The Lucent compiler currently targets the AMD Xilinx Alveo family of FPGAs and leverages AMD Xilinx's Vitis framework. This is illustrated in Figure 8, where Lucent source files are first combined with appropriate module files via the pre-processor. The *core* module is always imported, and other modules such as *euclid* and *buffer* are imported based on the *import* keyword. This is then provided to the Lucent compiler which will transform the Lucent code into the corresponding C++ HLS device code, along with all the necessary configuration files, host level, and build support. The device HLS code, along with configuration files are then provided to AMD Xilinx's Vitis tooling, along with the Lucent runtime, which generates the RTL and then progresses this through synthesis, placement and routing to ultimately generate a bitstream that will run on the FPGA hardware. The Lucent runtime is a collection of header files that can be called from the device C++ HLS code and provides common functionality such as data packing and unpacking for external writing and reading, stream replication for connecting generated dataflow regions together, and data type conversion. Furthermore, emulation in both software and hardware modes is supported by our flow.

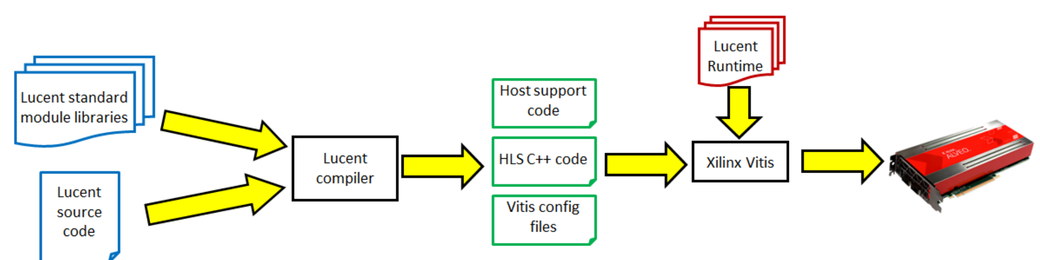

Figure 8. Illustration of our compilation tool-chain flow, from the programmer's source file to the target FPGA.

Within the compiler itself, after preprocessing the user's code the compiler then parses the source to form an Abstract Syntax Tree (AST) which is then transformed into a Directed Acyclical Graph (DAG). In the DAG each instance of an identifier such as sequences, filters and types variables are shared between the different parts of the tree that reference it. This DAG is then walked four times in separate phases, which involve the following:

1. Bind generic filters and types to actual values, bind filters passed as arguments and resolve overloaded filters. These all result in separate filters in the DAG, one for each permutation in the user's code, enabling independent type-specific optimisations and transformations to be applied in later phases.

2. Work at the individual operation level, resolving information such as storage location based on annotations to types and whether filter calls are across time dimensions. Also, determine how some operations should be implemented, for instance, maintaining minimal data copying whilst providing the immutability property for lists.
3. Inter-operation transformation and optimisation, for example, to identify spatial dependencies or conflicts on memory ports. Reorganisation of the DAG's structure is undertaken if appropriate and a check is inserted to ensure that the time quantum's split between the work and commit phase is consistent. There is no explicit synchronisation point for this split, but instead, if this phase determines the interaction of operations results in inconsistencies it adds a temporary value.
4. C++ HLS code is generated for the constituent parts and these are linked by transforming into an Intermediate Representation (IR) that represents the individual dataflow regions, connections between them and specific HLS configuration settings. This represents the concrete C++ code for each individual part, combined with an abstract view of how the different parts of the dataflow algorithm will be connected.

We generate the dataflow-based IR described above because, based upon this abstract representation, it is then much easier to determine the appropriate ordering of calls to dataflow functions, whether an output stream serves as input to multiple regions (in that case it must be duplicated) and identify any stream consumption consistency issues. Furthermore, on this IR it is also easier to determine which operations can be packaged together and what operations must be separated. This IR also contains tokens that represent configuration items specific to HLS, such as the partitioning of an array or whether a loop is to be pipelined, and these ultimately end up as HLS pragmas.

This IR is then lowered to HLS C++ once the aforementioned operations have been undertaken upon it, at this stage optimisations such as loop unrolling or automatic memory partitioning are also applied based on potential issues identified within the structure of the dataflow code. The Vitis HLS tooling is driven by appropriate pragmas, and a challenge was how to provide correct implementation (e.g., ensuring timing is met) and enable HLS optimisations to be undertaken effectively. Much of this depends on the structure of the code that is generated and has required significant experimentation, not least because HLS C++ was not designed as a target intermediate code. An example of this is termination, because in HLS one dataflow region cannot produce and consume streams from another specific dataflow region, or put another way, there can not be a two-way exchange of data between dataflow regions. This is a challenge because the external filter determines the termination of the code; however, this will inevitably consume values from other filters but it must also instruct these to terminate.

Our solution to this limitation imposed by HLS is illustrated in Figure 9, where there is a separate *health watcher* Compute Unit (CU), and this accepts an input AXI stream and outputs an AXI stream. Each dataflow region has an additional output stream (blue arrows in Figure 9) and these are combined by a dataflow region (the blue circle in Figure 9) which streams this out to the health watcher. The Lucent kernel CU, which contains the dataflow regions, accepts the health watch's output AXI stream (green arrows in Figure 9), this is then split into separate HLS streams by the green circle dataflow region in Figure 9 which are then fed into individual dataflow regions. With this structure, a dataflow region can issue the termination token to the health watcher which will then stream this back and distribute it amongst all the dataflow regions. Whilst this is a workaround to enable tokens to be sent to preceding dataflow regions, the health watcher could also be useful more generally in the future to capture aspects such as erroneous situations and issue logging or recovery.

Within an FPGA family, there are variations such as memory and chip size. Consequently, platform-specific configuration is provided to the Lucent compiler which describes the target FPGA in detail such as the available memory spaces and number of cycles required for different operations. This platform-specific information is then used by the compiler to make effective decisions about how to structure and generate the target code.

Currently, our compiler only supports AMD Xilinx Alveo FPGAs; however, only the activities on the dataflow IR contain AMD Xilinx-specific activities, and the runtime also abstracts much of the vendor-specific support that is required. Therefore, whilst it is still further work, much of this can be leveraged to target other FPGAs such as those from Intel.

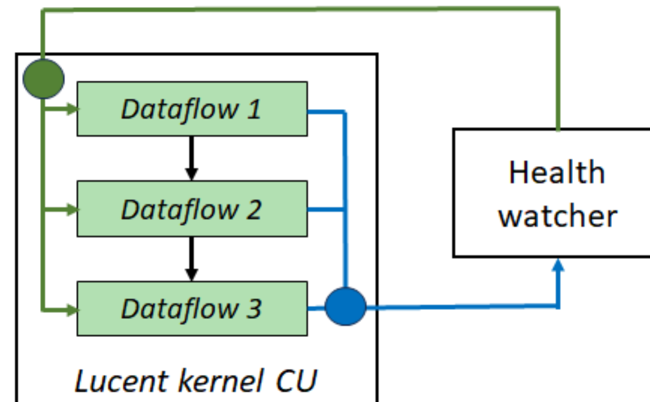

Figure 9. Illustration of a separate health watcher Compute Unit (CU) that enables loop back of termination signal to preceding dataflow stages.

7. Performance and Productivity Evaluation

In this section, we explore the performance and productivity of our approach on an AMD Xilinx Alveo U280 FPGA clocked at 300 MHz, using Vitis version 2021.2. For comparison, the CPU is a 24-core 8260M Cascade Lake Xeon Platinum (with CPU code threaded via OpenMP). All CPU and host codes were compiled with GCC version 10.2.0 and with -O3 optimisation level. For the BLAS experiments, we compared against the Vitis Library version 2021.2. All reported results are averaged over three runs. Card level power measurements on the FPGA are captured via XRT, and via RAPL on the CPU.

7.1. Foundational BLAS Operations

Basic Linear Algebra Subprograms (BLAS) [20] are a set of linear algebra operations forming the cornerstone of many computational kernels. In their open source Vitis library [21], AMD Xilinx have produced their own BLAS routines which we compare against, in addition to versions developed in DaCe [11] and HLS unoptimised for dataflow. The latter is a baseline and representative of HPC software developers without extensive FPGA optimisation knowledge.

As an example, we provide the code for the dot product in Listing 11 where it can be seen that the programmer only needs to write four lines of code for this kernel, with the Lucent compiler then determining the most efficient way to implement this. For comparison, this code is 32 lines in the Vitis library.

Listing 11. Dot product Lucent code (scalar version).

```
1  filter dot_product:double(a:double, b:double) where:
2    accumulated:double=a*b fby accumulated+(a*b)
3    dot_product=if (a == EOD or b == EOD) then:
4                      accumulated fby EOD
```

Table 1 summarises the runtime of five common BLAS operations (in double precision) on an AMD Xilinx Alveo U280. The Vitis library can vectorise operations and we include results for scalar runs (no vectorisation) and vectorisation. The Vitis library supports up to 16-way vectorisation which provides the best performance and is the configuration used, and we adopted (eight-way) vectorisation in Lucent using the list type. It can be seen that the scalar Lucent code significantly out-performs the Naive HLS code and scalar Vitis library code, where the Lucent compiler is generating more efficient dataflow code.

Performance of the vectorised Vitis library and vectorised Lucent code is much closer with the simpler Lucent code performing comparatively. With DaCe the axpy, gemv, and gemm benchmarks are written manually in the Stateful DataFlow multiGraph (SDFG) format, explicitly defining the dataflow graph and low level attributes of this, whilst the dot product and l2norm benchmarks are normal Python code with some function level decoration but placing more emphasis on DaCe to undertake code conversion. For the GEMM implementation DaCe uses a highly tuned systolic vectorised implementation from [11], although 438 lines of code, and in Lucent we developed a systolic version with (vect) and without (scalar) vectorisation at 46 lines of code. The Vitis library and naive GEMM implementations do not use a systolic approach.

Table 1. Performance details of BLAS routines. Problem size of 10,000,000 for dot product, axpy, and l2norm, 5000 for gemv, and 1000 for gemm.

Routine	Naive (ms)	DaCe (ms)	Vitis Library (ms)		Lucent (ms)	
			Scalar	Vect	Scalar	Vect
dot product	286.15	167.25	265.65	31.84	87.42	15.31
axpy	61.52	34.13	259.64	38.45	41.92	8.32
l2norm	247.37	167.04	136.75	19.83	56.96	18.96
gemv	422.41	83.90	401.25	55.76	154.65	31.56
gemm	2598.84	42.63	2187.49	173.74	356.32	65.32

There are several reasons for the performance differences observed in Table 1. The first is the level of concurrency provided by the algorithm, where multiple elements can be processed in parallel, and it can be seen in Table 1 that the vector implementations outperform their scalar counterparts. The second is how well the HLS kernel is optimised to avoid stalling, for example, that the iteration interval (II) for pipelined loops, which is the frequency of iteration processing, is one and data dependencies are removed. Thirdly, the compute part of the kernel is continually fed with data, for instance, ensuring that proceeding dataflow stages are generating results each cycle and that data are being most effectively read from and written to external memory. It was our hypothesis that the compiler, by consuming a rich amount of high-level information, could effectively undertake transformations that optimise for the second and third points. Indeed, the third point is the reason why Lucent is undertaking eight-way vectorisation rather than 16-way, as conducted by the Vitis library, for these routines. This is because, following best practice [22], for efficiency, the Lucent compiler generates HLS code that always reads from and writes to external memory in chunks of 512 bits, which is eight double precision numbers.

Table 2 reports the energy usage, in Joules, of each of the BLAS routines implemented across the different technologies. The energy is calculated based on the average power draw and runtime. The power draw, which is between 27 and 30 Watts, is fairly constant across the experiments. Consequently, it can be seen that the most significant distinguishing feature of energy efficiency is performance. Indeed, the Lucent implementations are competitive against those using other technologies.

Table 2. Energy usage of BLAS routines. Problem size of 10,000,000 for dot product, axpy, and l2norm, 5000 for gemv, and 1000 for gemm.

Routine	Naive (J)	DaCe (J)	Vitis Library (J)		Lucent (J)	
			Scalar	Vect	Scalar	Vect
dot product	4.83	4.83	7.52	0.95	2.56	0.46
axpy	1.76	0.95	7.37	1.09	1.13	0.25
l2norm	7.01	4.69	3.40	0.59	1.65	0.54
gemv	12.12	2.42	11.79	1.57	4.45	0.88
gemm	75.63	1.25	63.66	5.14	10.26	1.94

Table 3 reports LUT and BRAM HLS kernel resource usage for each of these routines for scalar versions of the kernel across the technologies. It can be seen that, generally, the naive version uses the least resources with more optimised implementations requiring more. The Lucent implementation tends to require the most resources and there are two reasons for this, firstly the fact that each filter is terminated dynamically via *EOD* which requires additional complexity to check for this condition and terminate gracefully. Secondly, the health watcher (whose resource usage for Lucent we include in these figures) described in Section 6 also adds some overhead. However, unlike the other kernels the Lucent version does not require recompilation for different problem sizes which, given the long compile times associated with FGPA development, is a significant benefit. The resource usage of these single HLS kernel BLAS routines is small, which makes the additional overhead of the Lucent implementation of greater relative relevance.

Table 3. Resource usage details of scalar BLAS routines with a problem size of 10,000,000 for dot product, axpy, and l2norm, 5000 for gemv, and 1000 for gemm.

Routine	Naive		DaCe		Vitis Library		Lucent	
	BRAM	**LUTs**	**BRAM**	**LUTs**	**BRAM**	**LUTs**	**BRAM**	**LUTs**
dot product	0.17%	0.31%	0.45%	0.88%	0.54%	0.42%	0.86%	0.71%
axpy	0.17%	0.32%	0.56%	1.27%	0.51%	0.33%	0.89%	0.45%
l2norm	0.38%	0.11%	0.69%	1.05%	0.11%	0.56%	0.72%	0.95%
gemv	0.23%	0.25%	0.94%	2.21 %	0.56%	0.50%	0.93%	1.72%
gemm	0.76%	3.21%	5.37%	7.28%	1.27%	9.69%	3.45%	6.55%

7.2. Application Case Study: Atmospheric Advection

The Met Office NERC Cloud model [23] is a high-resolution atmospheric model used by the Met Office and the wider scientific community for undertaking atmospheric simulations. Advection, which is the movement of particles through the air due to kinetic effects, accounts for around 40% of the overall runtime and the PW advection scheme was previously accelerated on FPGAs in [24] and then [2]. The computational algorithm is a 3D stencil operating over three fields U, V, and W which are wind in the x, y and z dimensions. In total, this kernel requires 66 double-precision floating point operations per grid cell.

Figure 10 illustrates the hand-optimised dataflow design from [2], where the boxes are HLS dataflow regions and the arrows are HLS streams that connect these. We implemented a version of this scheme in Lucent and this code is somewhat similar, albeit more complicated, than the 2D Jacobi example in Section 5. For instance, in this advection Lucent code we leverage 3D shift buffers, and there are three separate buffers and compute regions for each of the U, V, and W fields. Furthermore, the compute regions themselves contain more complicated mathematics between elements in the stencil which is not symmetric. However, in comparison to the 2D Jacobi example in Section 5, this advection benchmark does not operate in iterations so time dimensions are not required.

We compared our version in Lucent against the hand-optimised HLS code from [2], the original advection kernel on the 24-core Cascade Lake CPU, and a naive version in HLS which is directly based on the CPU version without any dataflow optimisation. These results are reported in Table 4 where *single* refers to a single CPU core or FPGA kernel and *entire* to all 24 CPU cores or as many kernels that will fit onto the FPGA. It can be seen that the naive HLS version compares very poorly against all other versions and the CPU, and due to this very poor single-kernel performance, we did not scale up the number of kernels. Furthermore, the naive version contained more lines of code than the Lucent and CPU versions, with the additional lines in the naive version tending to be pragmas that direct the HLS tooling around interfaces to enable the code to compile. Comparing single kernel performance between the hand-optimised HLS and Lucent versions is most revealing, where the hand-optimised HLS code is around 4% faster than the Lucent code,

however, at 598 lines compared with 23 lines, the Lucent code is 26 times shorter and much simpler for the programmer to develop and maintain.

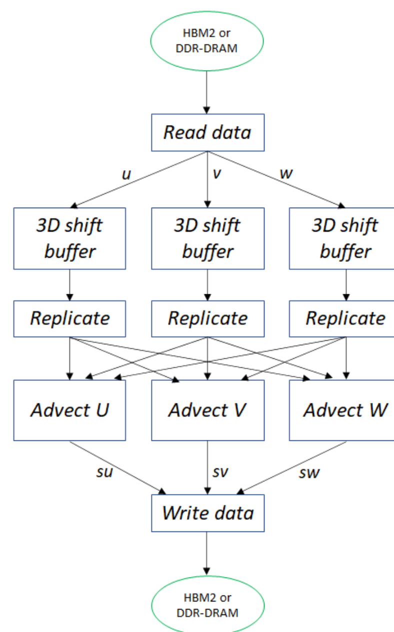

Figure 10. Sketch of hand-optimised dataflow design of advection HLS kernel from [2].

Table 4. Performance (in GFLOPs) and lines of code for PW advection kernels over problem size of 16 million grid cells.

Description	GFLOPs *(Single)*	GFLOPs *(Entire)*	Lines of Code
Xeon Platinum CPU	2.09	15.20	21
Naive HLS	0.018	-	32
hand-optimised HLS	14.50	80.22	598
Lucent	14.02	67.19	23

Considering performance over the entire chip, it can be seen that both the Lucent and hand-optimised HLS versions significantly outperform the 24-core CPU. However, hand-optimised HLS code is faster than Lucent because six hand-optimised HLS kernels can fit onto the FPGA compared to only five Lucent kernels. To explore this further, the LUT and BRAM resource usage for each implementation is reported in Table 5, where it can be seen that the resource usage for the Lucent version is higher than the hand-optimised HLS and that it is the BRAM that is the limitation here. In theory, the BRAM usage of the Lucent version should just fit and use 99% of the FPGA's BRAM; however, in practice, the tooling fails during placement with this configuration.

Table 5. Resource usage for FPGA PW advection kernels.

Description	LUT Usage *(Single)*	BRAM Usage *(Single)*	LUT Usage *(Entire)*	BRAM Usage *(Entire)*
Naive HLS	2.55%	8.65%	-	-
hand-optimised HLS	3.78%	14.40%	22.68%	86.42%
Lucent	5.23%	16.51%	26.15%	82.55%

The average power usage and power efficiency of different implementations of this benchmark are reported in Table 6. It can be seen that broadly the FPGA draws similar

average power for all implementations, although there is a difference when moving from single to multiple kernels. Furthermore, the Lucent implementation draws slightly less average power over multiple kernels compared to the hand-optimised version, but this is most likely because it is comprised of five rather than six compute units. When predicting power efficiency on the FPGA, the performance results in Table 4 act as a fairly accurate gauge. It can be observed that the power efficiency of both single hand-optimised HLS and Lucent implementations is similar, but the improved performance of six rather than five kernels results in better power efficiency for the hand-optimised HLS implementation when running across the FPGA.

Table 6. Average power usage and power efficiency (in GFLOPs/Watt) for PW advection kernels over problem size of 16 million grid cells.

Description	Single		Entire	
	Power *(Watts)*	**Efficiency *(GFLOPs/Watt)***	**Power *(Watts)***	**Efficiency *(GFLOPs/Watt)***
Xeon Platinum CPU	65.55	0.03	172.56	0.09
Naive HLS	32.56	0.0006	-	-
hand-optimised HLS	33.39	0.43	46.54	1.72
Lucent	33.55	0.42	44.23	1.52

7.3. Application Case Study: Credit Default Swap

Credit Default Swap (CDS) is a common financial instrument that provides an insurance policy against the non-repayment of loans where, for a premium, a third party takes on the risk of loan non-repayment. Quantitative finance is the use of mathematical models to analyse financial markets and securities, and CDS simulations are one frequent activity to ensure that CDS sellers make informed decisions around policies and premiums, and this is part of the wider quantitative finance domain which uses computing to model financial transactions [25]. CDS calculates *spread* which is the annual amount in basis points that the CDS protection buyer should pay the protection seller, and dividing this number by 100 expresses this as the percentage of the loan itself. A CDS engine simulates many different options and details [26] based upon three main inputs which are the interest (a list of percentages of interest payable on the loan) and hazard rates (the likelihood that the loan will default by a specific point in time) which both are constant during execution, and a vector of options which are processed one by one. Each of these three elements comprises two numbers, the point in time which is expressed as the fraction of a year and the value itself.

Unlike the advection kernel explored in Section 7.2, this algorithm is not stencil based and-so explores our approach when applied to a different pattern of operation. The code uses a double precision floating point throughout and follows the high-level dataflow design illustrated in Figure 11. Filters in blue iterate over a value multiple times before outputting a value, and in Lucent are represented using time dimensions.

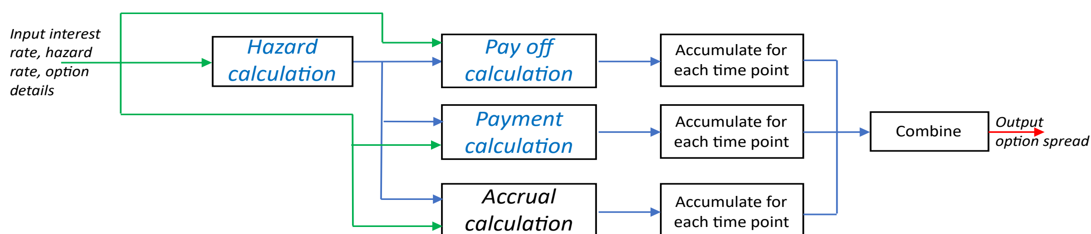

Figure 11. Illustration of Credit Default Swap (CDS) dataflow design.

We compare our Lucent CDS implementation against the CPU, AMD Xilinx's open-source Vitis library implementation [21], and a hand-optimised HLS kernel from [27]. Performance is measured in the number of options that are processed per second, and Table 7 reports performance and lines of code comparison. *Single* is running over a single CPU core or FPGA kernel, and *entire* over all 24 CPU cores or six FPGA kernels. The Vitis library implementation performs poorly compared to others because it promotes flexibility of integration over dataflow performance. The hand-optimised HLS approach is the fastest, however, it is also the largest and most complex. By contrast the Lucent programmer has written far less code (nine times less than the hand-optimised HLS code) and this productivity has cost 3% performance for a single kernel and 4% for multiple kernels.

Table 7. Performance & lines of code for CDS engine implementations.

Description	Options/sec *(Single)*	Options/sec *(Entire)*	Lines of Code
Xeon Platinum CPU	8738	75,823	102
AMD Xilinx's Vitis library	3462	16,071	252
hand-optimised HLS	27,675	114,115	742
Lucent	26,854	107,801	83

Table 8 details the average power usage and power efficiency for the different CDS engine implementations. It can be seen that the Xeon Platinum CPU draws significantly more power than the U280 FPGA, which is in agreement with the advection benchmark of Section 7.2, and even the *entire* configuration of six kernels on the FPGA draws less power than a single CPU core. We observe a similar pattern to the advection benchmark, where power efficiency is largely governed by performance and even though the FPGA draws less power than the CPU, AMD's Vitis library is less power efficient because it is significantly slower. The higher performance of the hand-optimised HLS and Lucent implementations, coupled with the low power draw of the FPGA, results in these being the two most power efficient implementations. However, in comparison to the power usage results of the advection benchmark in Table 6, it can be seen in Table 8 that the CDS engine implementation draws marginally more power for a single compute unit, but less power when running with multiple compute units across the FPGA. Whilst the differences between these benchmarks are fairly small, it does demonstrate that there is some variablility on a benchmark by benchmark basis.

Table 8. Average power usage and power efficiency for CDS engine implementations

Description	Single		Entire	
	Power *(Watts)*	Efficiency *(Options/Watt)*	Power *(Watts)*	Efficiency *(Options/Watt)*
Xeon Platinum CPU	65.98	132	175.39	432
AMD Xilinx's Vitis library	35.72	97	38.09	422
hand-optimised HLS	35.86	771	37.38	3052
Lucent	35.26	762	39.88	2703

The resource usage of the FPGA CDS implementations is reported in Table 9, where it can be seen that this is largely comparable across the implementations, although the Lucent version requires slightly more resources which is consistent with previous benchmarks. It is only possible to run using six FPGA kernels, not because of resource usage directly, but because we are exhausting the number of available AXI ports in the U280 shell. A maximum of 32 AXI ports are provided by the U280 shell and these are for all HLS compute units, with this limitation most likely driven by a single AXI crossbar connecting to all HLS IP blocks in the block design. For our HLS implementations, each kernel requires five separate AXI interface ports, and whilst it would be possible to reduce this number

per kernel by bundling ports together, it is only possible to transmit one piece of data per cycle on each AXI port. Therefore, whilst bundling would enable more HLS kernels to be leveraged, it also would mean that data can not be transmitted concurrently for the ports that are bundled together, thus significantly reducing single-kernel performance.

Table 9. Resource usage for CDS engine implementations.

Description	LUT Usage *(Single)*	BRAM Usage *(Single)*	LUT Usage *(Entire)*	BRAM Usage *(Entire)*
AMD Xilinx's Vitis library	9.16%	2.54%	54.96%	15.23%
hand-optimised HLS	9.52%	2.54%	56.86%	15.23%
Lucent	10.01%	2.73%	60.6%	16.38%

In conclusion, there is a small performance penalty when using Lucent compared to hand-optimised HLS code; however, the code is far simpler. This is the same for power efficiency, where the marginally better performance of the hand-optimised HLS implementation provides slightly better power efficiency than the Lucent kernel. However, the hand-optimised code has taken a long time to develop and requires considerable expertise in all the optimisation techniques, which is one of the major reasons for a large number of lines of code. Whilst our approach incurs a small power and resource usage penalty in comparison to the hand-optimised HLS code, in reality, few people are likely to undertake the work required in manually tuning their HLS code to such a degree and instead would rely on AMD's Vitis library. These situations would result in worse performance and power efficiency than the CPU.

By contrast, the Lucent code is far shorter than the other FPGA implementation, and even slightly shorter than the CPU code, because as mentioned, based upon the richness of information provided the compiler is able to conduct much of the heavy lifting around automatic optimisation. Whilst we observe power usage differences between the advection and CDS engine benchmarks, it is interesting to observe that for a single benchmark the power usage between implementations for a single compute unit on the U280 is fairly flat. Even when scaling to multiple kernels, this only incurs a marginal increase in power usage and this demonstrates that when considering power efficiency it is in fact performance on the FPGA, so the kernel runs as quickly as possible, which is the most important factor. This aligns closely with the objectives of Lucent, where the natural way in which to express algorithms is the fast way, and by leveraging an approach that is both correct and fast by construction one not only optimises for performance but also power and energy efficiency.

7.4. Limitations

Thus far we have concentrated on the strengths of our approach, but the design decisions undertaken have meant that there are some trade-offs. In abstracting low-level details from the programmer, our compiler makes specific choices based on prioritising performance over other aspects such as power or area. This was seen in the atmospheric advection case study (Section 7.2) where when the number of kernels was scaled up then one less Lucent kernel was able to fit on the FPGA compared against hand-optimised HLS code. One reason for this increased resource usage is supporting the *EOD* token for filter termination. This is a powerful approach as it provides dynamic termination which is determined at runtime. In comparison to other approaches, such as DaCe, when using Lucent the problem size is not a compile time constant, and therefore, the code does not need to be recompiled between configurations. Given the long build times involved in FPGA bitstream generation, this is very beneficial but it has been observed does incur some overhead. Development of a compiler optimisation pass to determine whether termination needs to be dynamic throughout, or in fact, whether some termination of some filters could be handled statically would likely provide benefit. Furthermore, the use of the type system is a way in which the programmer could provide hints to the compiler via annotations to direct it in the decisions that it makes.

Another limitation is that programmers must learn a new language and paradigm. This is a major ask of developers and whilst we would argue that, compared with the effort required to write high-performance HLS code this is time well spent, the reality is that many FPGA programmers are unlikely to switch. However, we see a major contribution of this work being the underlying dataflow execution model and language abstractions, which are language agnostic to drive the development of code-level dataflow abstractions that can be applied to existing languages. These could be encoded as Domain Specific Languages (DSLs) inside existing languages, such as Python, or even as an Intermediate Representation (IR) within the compiler itself.

There are some limits to the expressivity of our approach, for instance, the lack of loops makes it more verbose to iterate through data in multiple dimensions and hence the *euclid* Lucent-provided module as illustrated in Listing 10 to abstract this. The suitability of the dataflow architecture and underlying conceptual ASDM execution model forms the major limitation as to what algorithms are suitable to be expressed in this way. There have been a number of HPC codes ported to FPGAs and the examples described in this paper illustrate a variety of access and compute patterns in use; however, algorithms that are very tightly coupled with extensive feedback loops will be less suited to our approach, although they would also be less suited to a dataflow architecture.

8. Related Work

Instead of presenting the programmer with a dataflow execution model and abstractions, an alternative is to place the emphasis on improved compiler technology in extracting the essence of dataflow from existing Von Neumann-based languages. The Merlin compiler [28] optimised HLS code by requiring the programmer to leverage a small number of Merlin-specific pragma annotations which would then assist in identifying key patterns and transforming these based upon known optimisation techniques. However, our argument is that C or C++ code is such a long way from the dataflow model that the work that the compiler has to conduct in optimising these is significant, and instead, it makes more sense to provide improved abstractions to the programmer. This is illustrated by [28] by the fact that there is only one application considered, a logistic regression. MultiLevel Intermediate Representation (MLIR) [29] is a form of IR, and ScaleHLS [30] leverages MLIR to capture dataflow patterns from HLS code via a series of MLIR dialects. These place emphasis on the compiler to extract dataflow and make appropriate decisions, rather than our approach where the programmer encodes it. ScaleHLS shows promise and can complement our model and abstractions as improved compiler technology could then further assist the Lucent compiler in undertaking optimal low-level decisions.

Stencil-HMLS [31] leverages MLIR to automatically transform stencil-based codes to FPGAs. Driven by extracting stencils from existing programming languages [32] and Domain Specific Languages [33], this work operates upon the MLIR stencil dialect [34] to generate resulting code structures that are highly tuned for FPGAs and then provided to AMD Xilinx's HLS tool at the LLVM-IR level. This work demonstrates that based upon domain-specific abstractions, in this case, stencils, one is able to leverage the knowledge and expertise of the FPGA community to transform these abstract representations into an efficient dataflow form. Furthermore, it was also demonstrated that energy usage and efficiency are largely driven by performance on the FPGA. However, this approach is only for stencils, whereas our approach is general, and there is a large semantic gap between the stencil mathematical representation and target optimised dataflow code. This places significant pressure on the transformation and is somewhat tied to the specific stencil in question. In contrast, our approach is much more general and can handle many different types of applications. By constraining the programmer to write their code in a specific manner using the ASDM execution model and abstractions present in Lucent, our approach closes the semantic gap and makes the job of generating efficient target code more straightforward for the compiler.

DaCe [11] is a middle ground between IR and dataflow programmer abstraction, and we compared the performance of BLAS operators in DaCe on the U280 against our approach in Section 7.1. Dataflow algorithms can be expressed directly in DaCe's Stateful DataFlow multiGraph (SDFG) by the programmer, and this tends to provide the best performance but can be verbose and complex (for instance, the DaCe gemm implementation used in Section 7.1 from [11] is 438 lines of code) and requires lower level details such as the type of storage and size of stream buffers. It is often preferable for programmers to reason about their code mathematically rather than operationally, and it is also possible to transform existing codes into DaCe IR, which works well for libraries such as Numpy [35]. However, for general-purpose user codes, there is a semantic gap between DaCe IR and a user's Python code which must be bridged. The dot product and l2norm benchmarks in Section 7.1 followed this approach for DaCe, which resulted in poor performance because of unresolved spatial dependencies in the generated code causing stalls. In contrast, our approach requires the user to write their code directly using our languages and ASDM execution model, with the view that by doing so their code is in the correct form for a dataflow architecture such that the compiler can more easily undertake the low-level decisions required.

Approaches such as hlslib [36] and Maxelor's MaxJ [12] aim to deliver dataflow abstractions encoded as a class library but the programmer is still using a Von Neumann language. For instance, the MaxJ programmer uses Java to encode their application based on pre-provided abstractions which are then synthesised by Maxelor's compiler. Whilst MaxJ provides a rich set of class-based dataflow mechanisms, compared to our declarative approach the programmer must still drive their FPGA code via a Von Neumann-based language and retains responsibility for making numerous low-level decisions with detailed knowledge of the underlying architecture. Maxelor's NBody simulation code [37] is an example where the code is verbose with several complex concerns such as the number of ports and access mode for on-chip memory. Working with Von Neumann constructs, such as variables and loops, programmers must structure loops appropriately to avoid dependencies and set options such as the pipeline factor to translate Java to hardware efficiently.

The Clash [38] functional language is a hardware description language heavily based around Haskell and, unlike our approach, Clash transforms to VHDL. However, Clash is designed for a different type of workload and is much lower level than our approach, for instance, having been used to develop a RISC-V core [39]. Including hardware features such as signals, the clock, and reset Clash programmers are working much more at the circuit level than Lucent, and furthermore, only primitive Haskell types are supported which cannot be configured as in Lucent. Whilst there is a version of matrix multiplication provided in Clash, this is integer only because the language does not support floating point arithmetic numeric representation. By contrast, Lucent is much more aimed at the algorithmic level, to enable developers to abstractly develop fast by construction dataflow algorithms with the ASDM execution model in mind. Similar to Clash, Lava [40] is a Haskell module for FPGA design and unlike Clash is embedded within Haskell which, on the one hand, means that the programmer need not learn a new language but on the other hand is restricted by the assumptions made in Haskell and the fact that static analysis can not be performed, which Clash addresses.

There are Domain Specific Languages (DSLs) providing abstractions for specific problem domains for FPGAs [41]. An example is [42] for stencil-based algorithms on FPGAs and [43] for image processing. Whilst these show promise, compared to our general purpose approach, they are limited to a specific class of problem domain and algorithm like Stencil-HMLS [31]. Instead in this work, we have explored a general-purpose dataflow model and programming abstractions that can encode a wide variety of applications.

9. Conclusions and Future Work

In this paper, we have explored a conceptual execution model and dataflow abstractions that aim to empower software developers to write fast-by-construction dataflow codes

for FPGAs. Basing the programmer's execution model on bespoke Application Specific Dataflow Machines (ASDM), we present a language called Lucent which builds on some of the concepts developed in Lucid, and Lucent was then used as a vehicle to explore appropriate abstractions for programming FPGAs. The underlying dataflow abstractions and declarative nature of Lucent have been described, as well as how these can be used for building more complex abstractions, and how the type system provides additional flexibility and data customisation. We also described the concept of time dimensions, which until this point had only been studied theoretically, and by adopting this notion were able to provide improved expressivity in a logical and convenient manner.

We compared performance for several BLAS kernels, a real-world atmospheric advection scheme, and quantitative finance CDS against hand-optimised HLS, AMD Xilinx's Vitis open source library, DaCe, and CPU baseline. In all cases finding that Lucent performs comparatively, or better, than the Vitis library and DaCe, and is comparable against hand-optimised code, but is also much simpler. The next step is to target Intel's Stratix-10 FPGAs which will enable portability across FPGA vendors. It would also be interesting to port Lucent to other architectures, such as AMD Xilinx's AI engines [44] which are an example of coarse-grained reconfigurable architectures and at the time of writing, have been both packaged with their FPGA solutions and also with their main-stream AMD CPUs via Ryzen AI. It would also be interesting to expand the range of number representations that are supported in Lucent, for instance, Posits or coupling with custom IP blocks that provide their own numeric representation. The ability of FPGAs to flexibly provide support to handle processing these numbers in hardware is a potential advantage of them, and Lucent could be useful in hiding the complexity around leveraging these in codes.

It is generally tough to convince programmers to learn a new language; however, Lucent is really a vehicle to explore and develop the concepts and abstractions introduced in this paper. Such concepts could then be encoded in existing tools or languages. For instance, developing an MLIR dialect based upon Lucent, in which existing language and algorithmic patterns can then be targeted during compilation would be beneficial. This would likely significantly close the semantic gap between imperative, Von Neumann-based, languages and the dataflow paradigm, effectively acting as a stepping stone when lowering from these imperative languages to FPGAs.

We conclude that, by basing their conceptual execution model on an ASDM where all elements of the program state are updated concurrently in a consistent manner, and using a declarative language and the abstractions presented, software developers are able to develop fast-by-construction codes for FPGAs more productively and obtain high performance by default.

Funding: This research was funded by the ExCALIBUR EPSRC xDSL project grant number EP/W007940/1. This research was supported by an RSE personal research fellowship.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Acknowledgments: The authors acknowledge support from EPCC, including use of the NEXTGenIO system, which was funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 671951. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

Conflicts of Interest: The authors declare no conflicts of interest

Abbreviations

The following abbreviations are used in this manuscript:

HPC	High Performance Computing
FPGA	Field Programmable Gate Array
HLS	High Level Synthesis
ASDM	Application Specific Dataflow Machine
CPU	Central Processing Unit
GPU	Graphical Processing Unit
AST	Abstract Syntax Tree
DAG	Directed Acyclical Graph
IR	Intermediate Representation
BLAS	Basic Linear Algebras Subprograms
CDS	Credit Default Swap
SDFG	Stateful DataFlow multiGraph

References

1. Brown, N. Exploring the acceleration of Nekbone on reconfigurable architectures. In Proceedings of the 2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC), Atlanta, GA, USA, 13 November 2020; pp. 19–28.
2. Brown, N. Accelerating advection for atmospheric modelling on Xilinx and Intel FPGAs. In Proceedings of the 2021 IEEE International Conference on Cluster Computing (CLUSTER), Portland, OR, USA, 7–10 September 2021; pp. 767–774.
3. Karp, M.; Podobas, A.; Kenter, T.; Jansson, N.; Plessl, C.; Schlatter, P.; Markidis, S. A High-Fidelity Flow Solver for Unstructured Meshes on Field-Programmable Gate Arrays. In Proceedings of the 2022 International Conference on High Performance Computing in Asia-Pacific Region, Virtual Event, Japan, 11–14 January 2022 ; pp. 125–136.
4. Ma, Y.; Cao, Y.; Vrudhula, S.; Seo, J.S. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 45–54.
5. Firmansyah, I.; Changdao, D.; Fujita, N.; Yamaguchi, Y.; Boku, T. Fpga-based implementation of memory-intensive application using opencl. In Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, Nagasaki, Japan, 6–7 June 2019; pp. 1–4.
6. Xilinx. Vitis Unified Software Platform Documentation. Available online: https://docs.amd.com/v/u/2020.2-English/ug1416-vitis-documentation (accessed on 28 September 2024).
7. Intel. Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide. Available online: https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html (accessed on 28 September 2024).
8. de Fine Licht, J.; Blott, M.; Hoefler, T. Designing scalable FPGA architectures using high-level synthesis. In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Vienna, Austria, 24–28 February 2018; pp. 403–404.
9. Cong, J.; Fang, Z.; Kianinejad, H.; Wei, P. Revisiting FPGA acceleration of molecular dynamics simulation with dynamic data flow behavior in high-level synthesis. *arXiv* **2016**, arXiv:1611.04474.
10. Fraser, N.J.; Lee, J.; Moss, D.J.; Faraone, J.; Tridgell, S.; Jin, C.T.; Leong, P.H. FPGA implementations of kernel normalised least mean squares processors. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **2017**, *10*, 1–20. [CrossRef]
11. Ben-Nun, T.; de Fine Licht, J.; Ziogas, A.N.; Schneider, T.; Hoefler, T. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 17–22 November 2019; pp. 1–14.
12. Tech, M. *Multiscale Dataflow Programming*; Technical Report; Maxeler Technologies: London, UK, 2015.
13. Veen, A.H. Dataflow machine architecture. *ACM Comput. Surv. (CSUR)* **1986**, *18*, 365–396. [CrossRef]
14. Wadge, W.W.; Ashcroft, E.A. *Lucid, the Dataflow Programming Language*; Academic Press: London, UK, 1985; Volume 303.
15. Halbwachs, N.; Caspi, P.; Raymond, P.; Pilaud, D. The synchronous data flow programming language LUSTRE. *Proc. IEEE* **1991**, *79*, 1305–1320. [CrossRef]
16. Berry, G. A hardware implementation of pure Esterel. *Sadhana* **1992**, *17*, 95–130. [CrossRef]
17. Brown, N. Porting incompressible flow matrix assembly to FPGAs for accelerating HPC engineering simulations. In Proceedings of the 2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC), St. Louis, MO, USA, 15 November 2021; pp. 9–20.
18. Faustini, A.A.; Jagannathan, R. *Multidimensional Problem Solving in Lucid*; SRI International, Computer Science Laboratory: Tokyo, Japan, 1993.
19. Ashcroft, E.A.; Faustini, A.A.; Wadge, W.W.; Jagannathan, R. *Multidimensional Programming*; Oxford University Press on Demand: Oxford, UK, 1995.
20. Lawson, C.L.; Hanson, R.J.; Kincaid, D.R.; Krogh, F.T. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw. (TOMS)* **1979**, *5*, 308–323. [CrossRef]
21. Xilinx. Vitis Libraries. Available online: https://github.com/Xilinx/Vitis_Libraries (accessed on 28 September 2024).

22. Xilinx. Vitis—Optimising Performance. Available online: https://docs.amd.com/v/u/2020.1-English/ug1416-vitis-documentation (accessed on 28 September 2024).
23. Brown, N.; Lepper, A.; Weil, M.; Hill, A.; Shipway, B.; Maynard, C. A directive based hybrid met office nerc cloud model. In Proceedings of the Second Workshop on Accelerator Programming Using Directives, Online, 16 November 2015; p. 7.
24. Brown, N. Exploring the acceleration of the Met Office NERC cloud model using FPGAs. In Proceedings of the International Conference on High Performance Computing, Frankfurt, Germany, 16–20 June 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 567–586.
25. Dempster, M.A.H.; Kanniainen, J.; Keane, J.; Vynckier, E. *High-Performance Computing in Finance: Problems, Methods, and Solutions*; CRC Press: Boca Raton, FL, USA, 2018.
26. Hull, J.; Basum, S. *Options, Futures and Other Derivatives*; Prentice Hall: Upper Saddle River, NJ, USA, 2009.
27. Brown, N.; Klaisoongnoen, M.; Brown, O.T. Optimisation of an FPGA Credit Default Swap engine by embracing dataflow techniques. In Proceedings of the 2021 IEEE International Conference on Cluster Computing (CLUSTER), Portland, OR, USA, 7–10 September 2021; pp. 775–778.
28. Cong, J.; Huang, M.; Pan, P.; Wang, Y.; Zhang, P. Source-to-source optimization for HLS. In *FPGAs for Software Programmers*; Springer: Cham, Switzerland, 2016; pp. 137–163._8 [CrossRef]
29. Lattner, C.; Amini, M.; Bondhugula, U.; Cohen, A.; Davis, A.; Pienaar, J.; Riddle, R.; Shpeisman, T.; Vasilache, N.; Zinenko, O. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv* **2020**, arXiv:2002.11054.
30. Ye, H.; Hao, C.; Cheng, J.; Jeong, H.; Huang, J.; Neuendorffer, S.; Chen, D. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. *arXiv* **2021**, arXiv:2107.11673.
31. Rodriguez-Canal, G.; Brown, N.; Jamieson, M.; Bauer, E.; Lydike, A.; Grosser, T. Stencil-HMLS: A multi-layered approach to the automatic optimisation of stencil codes on FPGA. In Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, Denver, CO, USA, 12–17 November 2023; pp. 556–565.
32. Brown, N.; Jamieson, M.; Lydike, A.; Bauer, E.; Grosser, T. Fortran performance optimisation and auto-parallelisation by leveraging MLIR-based domain specific abstractions in Flang. In Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, Denver, CO, USA, 12–17 November 2023; pp. 904–913.
33. Bisbas, G.; Lydike, A.; Bauer, E.; Brown, N.; Fehr, M.; Mitchell, L.; Rodriguez-Canal, G.; Jamieson, M.; Kelly, P.H.; Steuwer, M.; et al. A shared compilation stack for distributed-memory parallelism in stencil DSLs. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, San Diego, CA, USA, 27 April–1 May 2024; Volume 3, pp. 38–56.
34. Gysi, T.; Müller, C.; Zinenko, O.; Herhut, S.; Davis, E.; Wicky, T.; Fuhrer, O.; Hoefler, T.; Grosser, T. Domain-specific multi-level IR rewriting for GPU: The Open Earth compiler for GPU-accelerated climate simulation. *ACM Trans. Archit. Code Optim. (TACO)* **2021**, *18*, 1–23. [CrossRef]
35. Ziogas, A.N.; Schneider, T.; Ben-Nun, T.; Calotoiu, A.; De Matteis, T.; de Fine Licht, J.; Lavarini, L.; Hoefler, T. Productivity, portability, performance: Data-centric Python. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, MO, USA, 14–19 November 2021; pp. 1–13.
36. de Fine Licht, J.; Hoefler, T. hlslib: Software engineering for hardware design. *arXiv* **2019**, arXiv:1910.04436.
37. Maxeler. N-Body Simulation. Available online: https://github.com/maxeler/NBody (accessed on 28 September 2024).
38. Baaij, C.; Kooijman, M.; Kuper, J.; Boeijink, A.; Gerards, M. C? ash: Structural descriptions of synchronous hardware using haskell. In Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, Lille, France, 1–3 September 2010; pp. 714–721.
39. Cox, D. Where Lions Roam: RISC-V on the VELDT. Available online: https://github.com/standardsemiconductor/lion (accessed on 28 September 2024).
40. Bjesse, P.; Claessen, K.; Sheeran, M.; Singh, S. Lava: Hardware design in Haskell. *ACM Sigplan Not.* **1998**, *34*, 174–184. [CrossRef]
41. Kapre, N.; Bayliss, S. Survey of domain-specific languages for FPGA computing. In Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, Switzerland, 29 August–2 September 2016; pp. 1–12.
42. Kamalakkannan, K.; Mudalige, G.R.; Reguly, I.Z.; Fahmy, S.A. High-level FPGA accelerator design for structured-mesh-based explicit numerical solvers. In Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Portland, OR, USA, 17–21 May 2021; pp. 1087–1096.
43. Stewart, R.; Duncan, K.; Michaelson, G.; Garcia, P.; Bhowmik, D.; Wallace, A. RIPL: A Parallel Image processing language for FPGAs. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **2018**, *11*, 1–24. [CrossRef]
44. Gaide, B.; Gaitonde, D.; Ravishankar, C.; Bauer, T. Xilinx adaptive compute acceleration platform: Versaltm architecture. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019; pp. 84–93.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.