Systematically migrating an operational microphysics parameterisation to FPGA technology

James Stanley Targett
and Wayne Luk
Department of Computing
Imperial College London
{jst10,w.luk}@imperial.ac.uk

Michael Lange
and Olivier Marsden
European Centre for Medium-Range
Weather Forecasts (ECMWF)
{Michael.Lange,Olivier.Marsden}@ecmwf.int

***Abstract*—Efficient utilisation of computational resources is one of the critical challenges in Numerical Weather Prediction (NWP) due to tight time constraints and the complexity of the numerical models. Enabling hardware acceleration is therefore of vital interest to the weather and climate modelling community. In this paper, we describe a systematic process to migrate physical parameterisations from sequential code for CPU execution into FPGA designs; a set of conditions that the code must be able to satisfy for the process to be suitable (Single Pass Exclusive Mutability of Current Cell); and describe the steps required to automate the process. We showcase the migration of a cloud microphysics parameterisation which forms a significant portion of the runtime in operational weather models, and show that the design produces results using an order of magnitude less energy than CPU and GPU implementations while achieving significantly higher throughput. We show that when incorporated inside a larger NWP system, a further throughput increase by a factor of five is possible.**

I. Introduction

Numerical weather and climate prediction poses one of the grand challenges for the upcoming era of exascale computing. Forecast accuracy is limited by model resolution, which in turn is perpetually limited by available computing resources and energy budgets. To achieve a step-change in predictive capabilities for operational weather forecasts, a significant reduction in time-to-solution and energy-to-solution for high-resolution computational models is of vital importance. Efficient utilisation of high-performance computing (HPC) resources is, therefore, a key prerequisite for significant improvements to forecasting skill [1][2]. The low energy footprint of dataflow computing architectures, such as Field Programmable Gate Arrays (FPGA), and their potential for highly scalable throughput-driven HPC systems are therefore of great interest to the weather and climate modelling community.

Dataflow architectures, however, represent a very fundamental change in programming model to traditional von Neumann architectures, such as CPUs or GPUs, with key elements not directly expressible in traditional programming languages used in HPC today [3]. Operational numerical weather prediction (NWP) systems, on the other hand, often contain large amounts of C/C++ or Fortran code that have been hand-optimised for a single architecture and HPC system, and often include a wide variety of coding patterns. It is therefore of crucial importance to analyse the applicability of the dataflow computing paradigm for a variety of model components, including components that often contain highly bespoke code, to evaluate the feasibility and potential gains offered by FPGA and dataflow architectures. Where dataflow architectures show gains over competing architectures, incorporating them into an operational system will require tools to convert source code produced by scientists into optimised FPGA designs.

Operational weather and climate codes commonly consist of multiple complex components that have often been optimised for individual HPC architectures over a long period of time. In the case of the Integrated Forecasting System (IFS), the operational model at the European Centre for Medium-Range Weather Forecasts (ECMWF), no clear hotspots emerge in performance profiles. This causes the performance engineering focus to shift to individual components of the model.

One class of components is the set of physical parameterisations. These are expected to contribute upwards of a third of IFS's operational cost in future configurations with increased resolution [4]. As currently employed in the IFS, these follow a common computational pattern. One of these physical parameterisations is the *cloud* microphysics *sc*heme, CLOUDSC. This has been extracted as a standalone dwarf (mini-application) as part of the ESCAPE project [4];

Our contributions in this paper are:

- the first FPGA design implementing CLOUDSC;
- the systematic process used to migrate CLOUDSC to FPGA, a set of conditions the code must be able to satisfy for the process to be suitable (SPEMCC), and we show that process which is automatable and applicable to other physical parameterisations;
- and performance comparisons between our FPGA implementation of CLOUDSC and reference CPU and GPU implementations.

We show an increase in energy efficiency compared to GPU of $3.9\times$ and $7.5\times$ for double and single precision, respectively, while achieving an increase in throughput of $2\times$ and $2.8\times$ respectively. In single precision, we show that, given more bandwidth, the FPGA could increase its throughput to $12.5\times$ that of the GPU while further increasing energy efficiency. This is achieved despite eschewing floating point optimisations so that correctness can be shown through bit-identical results.

2576-2621/21/$31.00 ©2021 IEEE
DOI 10.1109/FCCM51124.2021.00016

II. Background

A. IFS and Physical Parameterisations

Global numerical weather models are limited to grid resolutions and time steps that preclude the explicit resolution of all of the physical processes affecting meteorological atmospheric quantities. Accordingly, the effect of these processes must be modelled or parameterised.

As currently employed in the IFS, these follow a common computational pattern, where large amounts of data parallelism are available between individual columns of vertically aligned grid points. Within a column, large amounts of computational work are required for each grid point, often including point-wise solutions of small non-linear systems. In addition, a non-linear control flow with a high branching factor, as well as a large number of local stack variables per routine, result in high register pressure and sensitivities to memory latency on CPU, and low occupancy on GPU accelerators. This is similar to the challenges for atmospheric chemistry models described in [5]. As a result, both CPU and GPU architectures are primarily limited by poor register and cache utilisation. These are drawbacks that dataflow FPGA designs intrinsically avoid.

B. The cloud micro-physics parameterisation

Cloud formation and precipitation are among the processes that happen at small spatial and time scales. As well as allowing precipitation to be estimated, the cloud scheme is an essential component in the correct representation of radiative and latent heating in the atmosphere. A brief description of the cloud scheme in the IFS follows.

The IFS cloud scheme was originally written by Tiedtke [6], and significantly extended by Forbes et al. [7]. It provides equations for five prognostic variables, water vapour, cloud liquid water, cloud ice, rain and snow. The prognostic equation for each of the variables includes source and sink terms resulting from the physical processes which affect them.

It is of the form:

$$\frac{\partial q_i}{\partial t} = s_i + \frac{1}{\rho}\frac{\partial \rho V_i q_i}{\partial z}$$

where q_i is the specific water content for category i ($i = 1$ for cloud liquid droplets, $i = 2$ for rain, and so on), s_i is the net source or sink of q_i through microphysical processes, and the last term represents the sedimentation of q_i with fall speed V_i.

The coupled Partial Differential Equations are integrated implicitly in time, source and sink terms are separated into fast and slow terms relative to the model time step size, with fast terms treated implicitly, and the vertical advection term is treated implicitly.

With these choices, the equations can be written as:

$$\frac{q_\alpha^{n+1} - q_\alpha^n}{\Delta t} = A_i + \sum_{\beta=1}^{m} B_{\alpha\beta} q_\beta - \sum_{\beta=1}^{m} B_{\beta\alpha} q_\alpha + \frac{\rho_{k-1} V_\alpha q_{\alpha,k-1}^{n+1} - \rho V_\alpha q_\alpha^{n+1}}{\rho \Delta z}$$

where the superscript n refers to the current time step, $m = 5$ is the number of prognostic equations, and the $k-1$ subscript refers to a term taken at the model level above the current level k, k being implicitly present for all terms of the equation.

This scheme is run at every time step for all the columns in the atmospheric grid. The grid is distributed across MPI [8] tasks, with one task typically receiving on the order of $10^3 - 10^4$ columns with current grid and job sizes.

C. CLOUDSC Dwarf implementation

The cloud microphysics dwarf was chosen as a representative of physical subgrid-scale parameterisations in the IFS since it exhibits a complex set of coding patterns found across a range of routines in the IFS physics. It also exhibits a set of performance characteristics, commonly found in physical parameterisations across different NWP models, that often define the porting and parallelisation strategy [9][10]:

- Large amount of data parallelism across horizontal dimensions, with data dependencies along vertical columns due to the vertical advection term.
- High operational and arithmetic intensity with multiple coupled physical processes modelled per grid point.

While in the current operational configuration, CLOUDSC contributes over 7% of the operational cost of IFS [9].

The original code consists of approximately 3500 lines of Fortran2003 code. It contains a driver routine that invokes a kernel over sets of columns in a thread-parallel fashion, with each thread processing a user-defined number of columns at a time. The code has been optimised for CPU architectures by allocating all variables with a common, user-defined block-stride (memory blocking) to allow manual tuning of the working set for different cache sizes.

Despite a variation in the number of floating point operations (Flops) for each column due to value-driven conditional statements, a nominal average number of Flops-per-column has been established via performance counters during the original creation of the dwarf. Based on this, CLOUDSC has an average operational intensity of 3.35 Flops/byte (single precision) and 1.67 Flops/byte (double precision).

III. Migrating CLOUDSC to FPGA

A. Iteration Space

One of the fundamental changes when converting code from a traditional instruction-based architecture to a dataflow processor that implements the numerical computations directly in hardware is that the flow of data into the chip and between individually synthesised kernels needs to be well defined as a set of data streams [11], [12].

For an application programmer, this means that the largest iteration space over which a traditional subroutine executes needs to be expressed as a single conceptual loop to which a hardware-synthesised kernel can be mapped. All array arguments to that routine that share a common set of dimensions then define the input and output data streams. In our example, CLOUDSC operates on a variably large number of independent columns, each containing 137 vertical levels.

The presence of vertical derivative variables, however, means that we are processing array variables with an extent of 138 computational cells in each column.

Within the CLOUDSC kernel, two classes of loops can be identified:

- Short loops used for operations on vectors and matrices, such as solving a system of fluxes between the five prognostic variables in each vertical layer. These loops will be unrolled in hardware.
- Long loops over groups of columns, or over vertical layers within columns, that map to the data dimensions of input and output arrays. These grid dimensions can be collapsed to define a combined iteration space over all cells in the grid that spans all argument dimensions.

The latter loops define the overall iteration space for porting, which implies that the CLOUDSC kernel will initially be expressed as a single kernel. As a result, all computations to process a single grid cell will be performed in a fully pipelined datapath, resulting in the outputs for a single cell being generated at every clock cycle once the hardware pipeline is filled. In order to create this single dataflow kernel, loop fusion is applied to the original kernel code to ensure a single loop body exists that encompasses all computations for a single cell and defines the correct offsets along the stream dimensions.

After determining the bounds of the iteration space, the order in which this space is to be streamed into the FPGA must be decided. The data dependency between cells is due to the vertical advection term. This term is produced near the end of a cells pipeline and is required near the start of the pipeline for the next cell in the same column. This entails that the majority of the instruction pipeline for processing a single cell needs to complete before iterating to the next cell in a column.

The longest chain of computation within a column from a value computed in one cell to the same value in the next cell is around 100 floating point operations. With our compilers default rate of inserting registers into the pipeline, this results in a minimum of around 2500 cycles between the computation of adjacent cells in a column. This rules out iterating through one column before moving to the next.

On the other hand, between calculation and reuse, each cell receives 39 floating point values (2.5 kb for double precision) from the previous cell in its column. Storing these values off-chip would be prohibitively expensive as we expect the design to be bandwidth limited. If we were to iterate through all cells on a vertical level before moving to the next, our design would be limited to a maximum number of columns due to limited on-chip memory.

Therefore, we adopt an iteration order created by partitioning the space into blocks of columns. Within each block, we iterate through level-by-level before moving onto the next block. A lower bound on block size is the number of pipeline stages along the dependency path. A larger block size would lead to wasting buffer space, so we keep the block size close to this bound.

B. Single Pass Exclusive Mutability of Current Cell

Before porting, we reorganise the code to simplify the process. We require:

- that all variables are either read-only or associated with a single cell (1),
- that only one cell can have its variables modified at a time (2, Exclusive Mutability),
- that each cell's variables can only be modified together, and only once (3, Single Pass),
- and that cells can only access variables from another cell if the other cell has already been modified (4).

Single Pass (3) implies that there is an iteration structure covering all columns and cells, within which all modifications to cell's variables are made. More strongly, we will require that this is in the form of two layers of outer loop over the cells and columns, with a counter for each. To enforce (1), we will require that all variables that are written to must either be enclosed within the scope of both of the outer loops, or be an element of an array that is indexed by both the cell and column indexes. To enforce (4), we will require that arrays indexed by cell can only be accessed using the cell counter from the outer loop plus a non-positive integer.

These properties make the code susceptible to porting to a streaming design since the contents of the outer loops defines a pipeline that calculates the results for any single cell. We refer to this code meeting these conditions as being "Single Pass Exclusive Mutability of Current Cell" (SPEMCC) compliant. CLOUDSC was made SPEMCC compliant though loop fusion and loop reordering of the long loops, modifying the scope of variables, promoting column indexed arrays to also be cell indexed, and shifting the positions in which values are stored within arrays.

C. Conditional Execution

Within CPU code, conditional blocks are used. When the condition is false, the statements inside are not executed. Within a dataflow design, the potentially required calculation must be performed regardless of the value of the condition. If the condition is true, the result can then be applied; otherwise it is simply discarded. Where nested conditionals exist, the problem of wasted computation is compounded. This means that special cases can be very expensive on an FPGA.

Within CLOUDSC, some conditionals are within short loops and only depend on that loop's counter and a compile-time constant value. Since we will unroll these loops, the condition is tested at compile time and we can avoid unnecessary computation on the FPGA.

Another pattern is for two mutually exclusive blocks to compute similar calculations and apply the result to different variables. In this case, we can share resources between the two blocks and avoid unnecessary computation.

Excluding these two types of conditional block, CLOUDSC conditionals are nested up to four deep.

A notable top-level conditional block covers almost all of the computation and causes almost no work to be done for

the first 14 cells in each column. On a CPU, these cells, corresponding to around 10% of the total cell count, can be quickly skipped over, whereas in this design the FPGA must compute all the results regardless.

Only 2.5% of floating point operations occur nested within three conditionals, and only 0.2% within four conditionals.

D. Porting to a Dataflow Description Language

The next step is to convert the SPEMCC compliant code into a dataflow description language. In our implementation, we use MaxJ [12]. The steps to do this are as follows: the shell must be set up with boilerplate code; I/O streams must be created, along with counters to keep track of the iteration order; variable definitions must be modified to express streams of the correct type; index accesses to variables are removed where they refer to the current cell and replaced with stream offsets where they refer to a previous cell; where dependencies exist, a stream must be set up for use and later referenced from its final use; finally, conditional statements must be replaced with variables capturing the result of their evaluation, and assignments within the conditional block must be modified to use the ternary operator so as not to affect the results when the condition is false.

E. Breaking up the Kernel

The FPGA we target is a Xilinx Virtex UltraScale+ VU9P FPGA (16 nm). The VU9P chip consists of three Super Logic Regions (SLRs) stacked on top of each other. Some attention to the crossing points between the SLRs is required since the number of crossings is limited, additional resources are required to manage the crossing, and timing is more challenging to meet for crossing signals. The double precision version of our design uses more than 90% of some resources and so must be spread out over all three SLRs. As a single kernel, this requires signals to pass to all three SLRs to keep the stream in lockstep. This makes it difficult for the compiler to build the design and meet timing.

Therefore, we split the kernel into three pieces to spread between the SLRs, using buffers between them to ease timing requirements. To reduce the number of crossings, we split the kernel where communication can be minimised. Unfortunately, the majority of the kernel operates on three five-by-five matrices to perform the LU solve for the five prognostic quantities. Passing any of these matrices between kernels would be too costly. Therefore the design is split into the following three kernels: *setup*, *solve*, and *diagnostics*. The *solve* kernel is by far the largest and uses the majority of resources.

F. I/O

In the generated set of kernels, each cell is only processed once, with additional scalar values used and generated for individual columns. The overall kernel I/O is as follows: Each cell takes in 46 floating point (FP) inputs and produces 23 FP outputs. Each column takes two FP inputs and two boolean inputs and produces one FP output. Each cell also reuses seven of the FP input values from the previous cell in its column and produces 32 FP values to be used in the next cell in its column.

To efficiently move data on to and off of the FPGA, variables are grouped into vectors of 8 or 16 elements. As there are 46 per-cell inputs, our choice of vector sizes causes us to have to input 48 elements per cell. This leaves two unused elements for per-cell input.

The two per-column floating point inputs are always positive. This allows their sign bit to be repurposed to transfer the two per-column boolean inputs. In order to avoid implementing extra I/O channels, the two unused per-cell input elements are used for the per-column data.

As there are 23 per-cell outputs, this gets rounded up to 24, leaving one unused element. This is used for the single per-column output. The use of only 8- and 16-wide vectors causes a 4.8% overhead in I/O bandwidth usage due to unused elements.

Our design targets a Maxeler MAX5 card. In addition to the VU9P FPGA, this contains 48 GB of DDR4 RAM, split across three DIMMs, and a PCIe port for connection to its host machine. The PCIe connection is a Gen3 x16 connection that can theoretically reach a throughput of 15.75 GB/s in each direction. In practice, however, less than a quarter of this is typically achievable.

In order to be less constrained by bandwidth, we will use a setup that makes use of the on-card DRAM to illustrate the potential performance of the FPGA. Data will be considered to be resident on the DRAM, thereby avoiding the PCIe as a bottleneck. The on-card DRAM consists of three DIMMs that can each deliver a bandwidth of 15 GB/s. Each of the SLRs contains a memory controller that controls one of the DIMMs.

As the design is expected to be bandwidth limited, access to each of the three DIMMs should be as balanced as possible. The total off-chip I/O is 72 floats per cycle, so a target of 24 floats per cycle transferred to or from each DIMM should be aimed for. The *setup* kernel takes 24 floating point inputs and produces no output. The *solve* kernel takes 8 floating point inputs and produces 10 floating point outputs. The *diagnostics* kernel takes 16 floating point inputs and produces 14 outputs. Load balancing and vector grouping is achieved by moving the I/O for 8 inputs from the *diagnostics* kernel to the *solve* kernel, and then simply passing them along in the connection between kernels. The I/O for two of the outputs are moved from the *solve* kernel to the *diagnostics* kernel. This leads to each kernel having a total of 24 floating point inputs/outputs per cycle grouped into vectors of 8 or 16.

G. Validation: Bit identity of results

The accuracy of the CLOUDSC kernel is validated by comparing the results of a reference run on the host CPU to the offloaded FPGA output. For full validation, bit identity is desirable, particularly in the context of offloading components of large and complex meteorological codes.

To maintain bit identity, we deliberately eschewed the use of any floating point optimisations for FPGA. Despite this, we do not produce bit identical results to the reference run due

to the different implementations of `exp` and `log` functions between the system libm and Maxeler's KernelMath library.

To show that these differences are the only causes of non-bit-identical results between the port and the reference, the KernelMath implementations of `exp` and `log` are ported to C. After applying these modifications to the reference C kernel the two implementations produce bit-identical results.

H. Performance Model

The expected performance can be estimated by computing both I/O-limited and compute-limited performance numbers and taking the minimum of the two.

The I/O performance can be found by identifying the path which is most limited by bandwidth and then calculating according to Eq. 1, in which bw is the limiting bandwidth, x is the number of floats to be transferred, p is the number of bytes per floating point value (4 in single precision or 8 in double precision), and n_{io} gives the performance in numbers of columns executed per second.

$$bw \,/\, (138 \times x \times p) = n_{io} \tag{1}$$

bytes/s cells/column floats/cell bytes/float columns/s

For our design, the three connections to the DIMMs are expected to be equally limiting. For each of these, the expected bandwidth is 15 GB/s and 24 floats are transferred per cell.

For compute, the limiting factor is design feasibility and achievable frequency on the platform. A clock rate of C MHz limits performance to:

$$C \times 1 \,/\, 138 = n_{compute} \tag{2}$$

cycles/s cell/cycle cells/column columns/s

By setting these two formulas equal to each other, we can obtain the value of C that gives the minimum clock rate required to make use of all the available bandwidth. In the model, increasing the clock rate above this will not improve performance since the design will be I/O limited and the kernels will stall, waiting for input. However, in reality, it is likely that to achieve maximum performance the clock rate will need to be higher than predicted due to a variety of factors.

Given this model, we can predict that the double precision design can compute 566 thousand columns per second if a clock rate of 78.2 MHz is reached, while the single precision design can compute 1132 thousand columns per second if a clock rate of 156 MHz is reached.

I. Building the Designs

We build the designs using MaxCompiler version 2019.1 and Vivado version 2018.3. The design was successfully built in single precision, but in double precision, hardware congestion was a problem. This problem was overcome by noting that the achievable frequency of these designs is far below the MAX5 maximum frequency, allowing the pipeline to be coarsened through the use of MaxCompiler's *Pipelining Factor*, thereby reducing hardware requirements of the designs. With this modification, the double precision build was successful.

Based on design characteristics, additional optimisations for throughput can be carried out. The single precision designs use less than a third of the chip, allowing three copies of the design to be implemented simultaneously on the chip. However, since performance is already bandwidth-limited, adding more copies of the design will not improve performance. The hypothetical performance that could be achieved if sufficient bandwidth were available can be examined by connecting the same inputs to all three copies and discarding the outputs of two of them. This optimised single-precision design was successfully built.

For both the single and double precision designs, the designs can be built at clock frequencies higher than those predicted by the performance model. However, due to bandwidth limitations, this increased frequency should not increase performance. A hypothetical situation where more bandwidth was available can again be examined by reusing the inputs. Where the design did accept a new input on all cycles, it is modified to accept new inputs only on odd cycles. The design thus uses each input twice. If the previous input values are reused as is, an unrealistic power saving will result since flip-flop updates would not be needed [13]. In order to prevent this, the reused inputs are taken from three cycles prior. To achieve double the throughput of the designs with the input reuse, clock frequency must be doubled. This design is successfully built in both single and double precision with twice the clock frequency.

In single precision, the three kernel copies design can be combined with the inputs reuse optimisation. However, hardware contention precludes doubling the clock frequency. The maximum achieved clock for this design is 220 MHz. This should allow an approximately 40% increase in computational work over the three copy design.

The resource usage of these designs is shown in Table I.

Precision	Clock Frequency (MHz)	Uses Inputs Twice	Three Copies	Pipelining Factor	Logic Utilisation (%)	DSP Blocks (%)	BRAM18 Usage (%)	URAM Usage (%)
Double	80			0.3	48.32	90.80	45.56	21.15
Double	160			0.3	48.34	90.80	45.56	21.15
Double	160	✓		0.3	48.60	90.80	50.05	26.98
Single	160			0.3	20.46	31.93	26.76	10.42
Single	160			1	28.27	31.93	29.21	13.44
Single	315	✓		1	28.62	31.93	31.92	16.77
Single	160		✓	0.3	49.60	95.53	50.74	33.54
Single	220	✓	✓	0.4	53.83	95.53	59.42	44.79

TABLE I: Resource Usage

IV. Systematic Method and Automation

While an FPGA implementation of CLOUDSC may be able to achieve higher performance than the Fortran or C version, it could not become used in a production system while it is manually converted from another language as this process is far too time consuming and prone to error. As models are continually improved by weather and climate scientists,

Algorithm 1 Overall process of generating hardware from fortran

1: Enforce of Exclusive Mutability of Current Cell
2: Determine block size
3: Port to Dataflow Description Language
4: Break kernel into smaller chunks
5: Determine I/O groupings
6: Build bitstream and generate host executable

the codebase must be in a language appropriate for those scientists. MaxJ and other hardware languages cannot be this language as they require specialist knowledge and obscure the code with unnecessary details that the scientists should not be concerned with. This necessitates some form of automation that allows the automatic conversion of the code to MaxJ or a similar language if FPGAs are ever to be used to accelerate a production system.

The systematic process we followed is described in Algorithm 1. In following the process, we found no steps that could not be automated. Therefore we expect to be able to automate the whole process. This would allow other physical parameterisations within IFS and future versions of CLOUDSC to be easily accelerated on FPGAs without expert input.

The first step is to modify the code to make it SPEMCC compliant (Section III-B). While enforcing SPEMCC on any given code may be a difficult task, CLOUDSC is representative of the physical parameterisations within IFS. The transformations that were required for CLOUDSC could be detected by pattern matching and high-level data dependency analysis, and so we believe that these types of transformations could be automatically applied to other physical parameterisations to make them SPEMCC compliant.

The second step is to determine the size of each block of columns. In MaxJ, we can use the *AutoLoopOffset* feature to automatically find and use the lowest possible value.

The third step is to convert the code to a dataflow description language. Once the code is SPEMCC compliant, this task can be automated. For MaxJ, this can be done by following the steps in Section III-D.

The fourth step is to break up the kernel into smaller chunks (Section III-E). This is a graph partitioning problem and can use the tools associated with that problem [14]. After the original kernel is split, the method for determining the block size through AutoLoopOffset will no longer work correctly. Therefore the block size must be extracted before this step and stated explicitly instead.

The fifth step is to determine the I/O groupings and where the data is coming from or sent to (Section III-F). If per-column inputs/outputs are small in number, we can consider them as per-cell inputs/outputs. If floating point inputs/outputs are labelled as non-negative, they can be used to transfer boolean inputs/outputs to/from the same kernel. Remaining boolean inputs can be grouped together. Each kernel can be associated with an SLR, and each kernel's inputs and outputs can each be considered a separate group. To balance the required bandwidth from each SLR according to the available bandwidth, inputs can be moved to preceding kernels and outputs can be moved to succeeding kernels.

The final step is to build the bitstream and generate the host executable file. This requires setting a clock rate and running the compilation toolchain. We can start with the clock rate suggested by the performance model. If the build is unsuccessful we can reduce the clock rate and/or modify parameters, such as pipelining factor, in a trial-and-error process until the build is successful. It is possible that the build will not be successfully if, for example, the design is too large to fit on the chip.

V. Performance evaluation

The MAX5 accelerator card (16 nm) was hosted on a machine with two Intel Xeon CPU E5-2643 v4 (14 nm) running at 3.40 GHz, each with 6 cores (12 threads), and 128 GB of DDR4 RAM running at 2400 MHz. The host runs on CentOS Linux release 7.4.1708 and uses MaxelerOS 2019.1 to communicate with the MAX5 card. The host code was compiled with gcc version 4.8.5.

The CPU reference runs were performed on a single socket of a Cray XC40 node with an Intel Xeon EP E5-2695 V4 (14 nm) "Broadwell" with 18 cores clocked at 2.1 GHz. The original Fortran code was compiled with Cray CCE 8.7.7 with compiler flags identical to operational ECMWF software builds. The code is parallelised via OpenMP and run with two pinned hyperthreads per physical core.

The GPU reference runs were performed on a single NVidia Quadro GV100 Volta (12 nm) with 32 GB on-card memory using the PGI 19.5 compiler. The GPU-enabled code was ported using OpenACC offload directives and optimised for GPU execution [9].

On the MAX5 FPGA and NVidia GPU systems, data transfers to the accelerator cards were excluded from performance timings since the CLOUDSC kernel is known to be limited by data transfers to and from accelerator devices connected via PCIe 3.0. The rationale for this is that CLOUDSC is originally part of a large set of physical parameterisations that are optimised within a single parallel region for CPU execution. As a result, several of the required input fields are temporary values that would be device-resident in any operational setting, where the combined work-to-data-transfer ratio would be more favourable to accelerator execution.

A. Power measurements

An approximate power usage comparison is undertaken between the three architectures for the CLOUDSC code. It was not possible to install power monitoring hardware on the Cray and the GPU systems, and there is no single software-based power-monitoring solution that covers all three systems, so the comparison does not aspire to be definitive.

Measurements of the MAX5 were made with the host's cooling set to maximum and both PSUs of the host supplied from an ATEN PE810G power monitor. For each test, we ran the application continuously for over one minute to warm up and then took throughput and power readings over at least five minutes. Readings were taken every two seconds and

Device	Data Location	Clock Frequency (MHz)	Uses Inputs Twice	Pipelining Factor	Thousand Columns Per Second	Proportion of perf. model	Dynamic Power Usage (W)	Energy Usage Per Column (μJ)
XC40	Host	2100		-	205.1	-	118	575.3
V100	GPU	1627		-	264.4	-	85	321
FPGA	DFE	80		0.3	289	51.2%	28.6	98.8
FPGA	DFE	160		0.3	531	93.8%	43.8	82.4
FPGA	DFE	160	✓	0.3	1070	94.5%	77.2†	72.2†

TABLE II: Double Precision Performance Results.
† indicates the addition of predicted bandwidth power/energy cost.

Device	Data Location	Clock Frequency (MHz)	Uses Inputs Twice	Three Copies	Pipelining Factor	Thousand Columns Per Second	Proportion of perf. model	Dynamic Power Usage (W)	Energy Usage Per Column (μJ)
XC40	Host	2100			-	326	-	120	368
V100	GPU	1627			-	383	-	76	198
FPGA	DFE	160			0.3	1070	94.5%	27.6	25.7
FPGA	DFE	160			1	1062	93.8%	28.1	26.5
FPGA	DFE	315	✓		1	2140	94.5%	48.2†	22.5†
FPGA	DFE	160		✓	0.3	3228	92.8%	70.8†	21.9†
FPGA	DFE	220	✓	✓	0.4	4778	99.9%	94.3†	19.7†

TABLE III: Single Precision Performance Results.
† indicates the addition of predicted bandwidth power/energy cost.

averaged over the whole period. The system's static power was measured over a ten minute period with the idle bitstream on the MAX5 card. The initial five minute period was discarded.

For the designs that simulate additional physical memory bandwidth, we acknowledge that the additional bandwidth would lead to increased power requirements. To approximately quantify these, we build a hardware design with all the I/O to/from DRAM but with none of its compute synthesised. Power usage of this design is compared to the idle system power usage. The difference between these values provides an estimate of the power usage increase for the extra bandwidth. We use this estimate to extrapolate the power and energy for the designs with simulated extra bandwidth reported in Tables II and III.

For the GPU and the Cray XC40 runs, the power measurements are full-board and full-node power, respectively. These readings are supplied by vendor-specific diagnostics tools. Power consumption was measured while the systems were idle and under load. The difference is reported. On the Cray, performance counters were sampled before the benchmark execution and coarsely sampled during the parallel loops [15]. On the NVidia GPU, power consumption was read using `nvidia-smi` [16].

B. Double precision results

Performance results for double precision runs are shown in Table II. An initial build with a clock frequency of 80 MHz, just above the performance model target of 78.2 MHz, achieves a throughput higher than the CPU and GPU while using significantly lower power, despite falling short of the predictions of our performance model. Doubling the frequency to 160 MHz allows us to achieve significantly improved throughput, which is approximately $2\times$ faster than the GPU and more than $2.5\times$ faster than the CPU.

Importantly, the throughput of the ported FPGA kernel is memory-bandwidth limited and therefore, higher performance should be achievable with higher bandwidth. This is confirmed by reusing the inputs to simulate higher available bandwidth, which leads to a doubling in throughput: $4\times$ faster than the GPU and more than $5\times$ faster than the CPU.

The dynamic power consumption of the FPGA chip itself is significantly lower than either CPU or GPU architecture. Considering the energy consumption per computed column for the bandwidth-mimicking benchmark, including estimated memory power consumption, an energy efficiency improvement of $4.4\times$ over GPU and $8\times$ over CPU can be achieved.

C. Single precision results

The single precision results shown in Table III further demonstrate the memory bandwidth performance limitation. Using the same 160 MHz clock frequency, a throughput similar to that of the dual-input double precision runs is obtained, resulting in a speed-up of $2.8\times$ over GPU and $3.3\times$ over the single-socket CPU. As before, the FPGA kernel performance is still bound by the available memory bandwidth since reusing input data indeed doubles the achievable throughput. The energy usage per column of the FPGA chip is more than $7\times$ lower than the GPU for all the implementations and is over an order of magnitude lower compared to the CPU.

Moreover, due to the low resource usage of the single precision kernel design on the chip, an additional performance improvement can be achieved by supplementing the pipeline parallelism inherent to FPGAs with data parallelism by synthesising three copies of the datapath design on the chip. Assuming that bandwidth is sufficient to keep the three copies busy, this can increase performance by an additional factor of three since each kernel copy can process data independently.

The combined configuration of input reuse and design replication satisfies our performance model to 99.9%. By scaling the performance prediction for DRAM runs from 156 Hz to 220 Hz, we expect a single datapath design to achieve a throughput of 1596 columns per second, corresponding to 4788 columns per second for the three-copy design. The close match between prediction and benchmark runs underlines the memory bandwidth limitation of the FPGA kernel.

D. Xilinx U280 Accelerator Card

The VU9P FPGA we have used in this study is not the most powerful FPGA available. A Xilinx U280 accelerator card contains an XCU280 (16 nm) with 32% more DSP slices than a VU9P, and also has high bandwidth memory with a total available bandwidth of 460 GB/s. Given the number of copies of our design that we can fit on the VU9P was strongly limited by available DSP, it is reasonable to project that four copies of the single precision datapath could be instantiated on the device. As our designs were also strongly limited by available bandwidth, the more than ten times

increase in off-chip bandwidth available on the U280 would increase performance greatly. From our performance model, this bandwidth would enable 12.5 million columns to be computed per second, but would require an unrealistic 428 MHz to achieve this. If a modest clock rate of 200 MHz was achieved, 5797 thousand columns per second could be computed, $17.8\times$ and $15.1\times$ better than the CPU and GPU used in our performance evaluation, respectively.

VI. Evaluation and discussion

The performance and power efficiency metrics presented in the previous section clearly demonstrate significant advantages achieved through optimised FPGA execution for the CLOUDSC kernel. In this section, we aim to provide some context to the achieved performance results, evaluate the potential of FPGAs and dataflow computing as HPC accelerators and a programming model for weather and climate codes, and discuss the systematic method and its automation.

A. FPGA acceleration for physical parameterisations

The parallelism afforded in atmospheric physics kernels due to the lack of inter column dependencies, along with a single direction of the dependencies within each column, make the code suitable for pipelined implementation on FPGAs by exploiting a similar data streaming and iteration scheme as described in section III-A. Despite the complex nature of the computations and the high branching factor in the original code, the data path of the converted source remained manageable once the kernel was split into individual subsections. The redundant computation introduced by evaluating both branches of any elementary conditional in the kernel increases the resource usage of the chip, but it has no direct impact on throughput since the data streaming frequency can be retained. As such, our design demonstrates that complex physical parameterisations can be accelerated using FPGA.

As demonstrated in Section V-C, the performance of the resulting FPGA kernel is limited by the available memory bandwidth on the accelerator card. Moreover, since the validation of the ported kernel was based on bit-identity with CPU results, further optimisations that might affect exact bitwise replication have been avoided. If this strict validation requirement was lifted, it is conceivable that additional FPGA-specific optimisation strategies could improve the achievable throughput of the chip further while maintaining scientific accuracy of the results. Such optimisations could include: 1) Optimisations such as the use of Tri-Adders to reduce the data path complexity. 2) Removal of divisions where possible as these are expensive to perform on FPGA. 3) Reduction of precision for computation down to the precision required for results with the required accuracy. 4) Reduction of precision in I/O could allow reduction of bandwidth usage. Computations inside kernels could be maintained in higher precision. 5) Computation and/or I/O performed in fixed point.

B. Bandwidth limitations

Maximum achievable performance for the CLOUDSC kernel on the MAX5 system has been shown to be strongly limited by available memory bandwidth. Optimisations described here, and tested as far as possible with ersatz bandwidth equivalents, suggest that performance would scale up to a factor of four, with increasing memory bandwidth. Dataflow computing has effectively allowed a problem which is register- and core-bound on traditional architectures to be transformed into a situation limited by memory bandwidth.

For the particular case of CLOUDSC, and more generally for meteorological physical parameterisations, kernel IO would not necessarily be to RAM but could instead be to another parameterisation. This would, in practice, reduce the memory bandwidth required to fully utilise the FPGA chip. Chaining of parameterisations could be on-chip if hardware resources are sufficient, or between chips in systems where FPGAs are connected with very high bandwidth.

C. Systematic Porting Process

We have shown the process used to migrate CLOUDSC to FPGA and described the steps towards its automation. CLOUDSC is only one of the physical parameterisations that together are expected to make up 38% of IFS's operational cost under CPU and GPU architectures. Due to the common computational patterns within the physical parameterisations, we expect that an automated version of the process we describe will allow these parameterisations to be migrated to FPGA with competitive performance. This would allow scientists to continue to develop the models in the original source code while benefiting from FPGA acceleration.

VII. Conclusion

In this paper we described a systematic method for migrating microphysical parameterisations from sequential CPU code to high performance FPGA designs. The method is suitable for paramaterisations that follow the common computation pattern found within IFS. More broadly, we described the specific set of conditions, SPEMCC, that the code must be able to comply with for it to be migrated. We described the potential for automating this method.

We showcased the method through migrating CLOUDSC to run on a Maxeler MAX5 card and achieved speed-ups of up to $2.8\times$ over a GPU (NVidia Volta) and up to $3.3\times$ over a CPU (Intel Broadwell) for single precision runs, as well as speed-ups of $2\times$ over GPU and $2.5\times$ over the CPU in double precision. Coupled with a significant increase in energy efficiency ($4.4\times$ over GPU and $8\times$ over CPU in double precision).

Acknowledgements

This work was carried out with support from the EuroExa project (grant agreement no. 754337), funded by the European Union's Horizon 2020 Research and Innovation Programme. The support of the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS, Grant Reference EP/L016796/1) and the Maxeler University Programme is gratefully acknowledged. Many thanks also to B. Reuter for help with preparation of the manuscript.

References

[1] T. C. Schulthess, P. Bauer, N. Wedi, O. Fuhrer, T. Hoefler, and C. Schär, "Reflecting on the goal and baseline for exascale computing: A roadmap based on weather and climate simulations," *Computing in Science Engineering*, vol. 21, no. 1, pp. 30–41, Jan 2019.

[2] P. Bauer, T. Quintino, N. Wedi, A. Bonanni, M. Chrust, W. Deconinck, M. Diamantakis, P. Düben, S. English, J. Flemming, P. Gillies, I. Hadade, J. Hawkes, M. Hawkins, O. Iffrig, C. Kühnlein, M. Lange, P. Lean, O. Marsden, A. Müller, S. Saarinen, D. Sarmany, M. Sleigh, S. Smart, P. Smolarkiewicz, D. Thiemert, G. Tumolo, C. Weihrauch, and C. Zanna, "The ecmwf scalability programme: Progress and plans," no. 857, 02 2020. [Online]. Available: https://www.ecmwf.int/node/19380

[3] O. Pell and O. Mencer, "Surviving the end of frequency scaling with reconfigurable dataflow computing," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, pp. 60–65, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2082156.2082172

[4] A. Müller, W. Deconinck, C. Kühnlein, G. Mengaldo, M. Lange, N. Wedi, P. Bauer, P. K. Smolarkiewicz, M. Diamantakis, S.-J. Lock, M. Hamrud, S. Saarinen, G. Mozdzynski, D. Thiemert, M. Glinton, P. Bénard, F. Voitus, C. Colavolpe, P. Marguinaud, Y. Zheng, J. Van Bever, D. Degrauwe, G. Smet, P. Termonia, K. P. Nielsen, B. H. Sass, J. W. Poulsen, P. Berg, C. Osuna, O. Fuhrer, V. Clement, M. Baldauf, M. Gillard, J. Szmelter, E. O'Brien, A. McKinstry, O. Robinson, P. Shukla, M. Lysaght, M. Kulczewski, M. Ciznicki, W. Piątek, S. Ciesielski, M. Błażewicz, K. Kurowski, M. Procyk, P. Spychala, B. Bosak, Z. P. Piotrowski, A. Wyszogrodzki, E. Raffin, C. Mazauric, D. Guibert, L. Douriez, X. Vigouroux, A. Gray, P. Messmer, A. J. Macfaden, and N. New, "The escape project: Energy-efficient scalable algorithms for weather prediction at exascale," *Geoscientific Model Development*, vol. 12, no. 10, pp. 4425–4441, 2019. [Online]. Available: https://www.geosci-model-dev.net/12/4425/2019/

[5] M. Alvanos and T. Christoudias, "GPU-accelerated atmospheric chemical kinetics in the ECHAM/MESSy (EMAC) Earth system model (version 2.52)," *Geoscientific Model Development*, vol. 10, no. 10, pp. 3679–3693, 2017. [Online]. Available: https://www.geosci-model-dev.net/10/3679/2017/

[6] M. Tiedtke, "Representation of clouds in large-scale models," *Monthly Weather Review*, vol. 121, no. 11, pp. 3040–3061, 1993. [Online]. Available: https://doi.org/10.1175/1520-0493(1993)121<3040:ROCILS>2.0.CO;2

[7] R. Forbes, A. Tompkins, and A. Untch, "A new prognostic bulk microphysics scheme for the ifs," no. 649, p. 22, 09 2011. [Online]. Available: https://www.ecmwf.int/node/9441

[8] W. Gropp, *MPI (Message Passing Interface)*. Boston, MA: Springer US, 2011, pp. 1184–1190. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_222

[9] H. Xiao, M. Diamantakis, and S. Saarinen, "An openacc gpu adaptation of the ifs cloud microphysics scheme," no. 805, 2017. [Online]. Available: https://www.ecmwf.int/node/17320

[10] V. Clement, S. Ferrachat, O. Fuhrer, X. Lapillonne, C. E. Osuna, R. Pincus, J. Rood, and W. Sawyer, "The claw dsl: Abstractions for performance portable weather and climate models," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '18. New York, NY, USA: ACM, 2018, pp. 2:1–2:10. [Online]. Available: http://doi.acm.org/10.1145/3218176.3218226

[11] R. Duncan, "A survey of parallel computer architectures," *Computer*, vol. 23, no. 2, pp. 5–16, Feb 1990.

[12] Maxeler, "Programming mpc systems," Tech. Rep., 2013. [Online]. Available: https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf

[13] H. M. Vo, "Optimizating power consumption using multi-bit flip-flop technique in tetris game on fpga," in *Proceedings of the International Conference on System Science and Engineering (ICSSE)*, ser. ICSSE 17. IEEE, 2017, pp. 530–533. [Online]. Available: https://doi.org/10.1109/ICSSE.2017.8030930

[14] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[15] S. J. Martin and M. Kappel, "Cray xc30 power monitoring and management," in *Cray User Group Conference Proceedings*, 2014.

[16] NVIDIA, "Nvidia system management interface," Tech. Rep. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface