From Tensor Algebra to Hardware Accelerators: Generating Streaming Architectures for Solving Partial Differential Equations

Francis P. Russell, James Stanley Targett, Wayne Luk
Department of Computing, Imperial College London, London, SW7 2AZ, UK
{francis.russell02,james.targett10,w.luk}@imperial.ac.uk

***Abstract*—Hardware accelerators are attractive targets for running scientific simulations due to their power efficiency. Since, large software simulations can take person years to develop, it is often impractical to use hardware acceleration, which requires significantly more development effort and expertise than software development. We present the design and implementation of a proof-of-concept compiler toolchain which enables rapid prototyping of hardware finite difference solvers for partial differential equations, generated from a high-level domain specific language. Multiple fields, grid staggering and non-linear terms are supported. We demonstrate that our approach is practical by generating and evaluating hardware designs derived from the heat and simplified shallow water equations.**

I. Introduction

Large-scale scientific simulations are intensive both in terms of computation and in power consumption. Reconfigurable architectures are a promising target to run such simulations due to their high power efficiency. However, these architectures have not been widely adopted.

A major obstacle to using reconfigurable hardware for scientific simulation is the high complexity of developing hardware designs which can take an order of magnitude longer time than the corresponding software design [1]. Additionally, a developer requires both a significant understanding of hardware design and of the underlying mathematics of the simulation they are implementing. If scientists wish to make alterations to the model, these changes again require hardware expertise and will most likely take longer than the corresponding changes in a software implementation. Without sophisticated tools, hardware acceleration will never be practical for many scientific simulations.

In the software domain, great success in solving similar issues has been achieved though the use of domain-specific languages (DSLs). Using a DSL description, scientists can define and alter scientific models in a notation familiar to them. Meanwhile, compiler experts can develop a toolchain that generates optimised code for these equations, without needing to understand the subtleties of the underlying mathematics.

In this work, we present an initial attempt to bring this methodology to finite difference solvers implemented in reconfigurable hardware. We present the design, approach and implementation of a compiler that given partial differential equations (PDEs) specified using a DSL, generates a hardware design for a finite difference solver.

We make the following contributions:

- A novel approach compiling tensor algebra through multi-level DSLs to reconfigurable hardware designs;
- A prototype tool implementing the proposed approach based on Haskell and targeting Maxeler platforms;
- Evaluation of the approach based on designs for a simplified version of the shallow water equations and the heat equation, illustrating the DSLs and the performance and power consumption of the resulting designs.

II. Background

The Unified Form Language (UFL), adopted by the FEniCS project [2], is used to specify finite element variational forms and provides the input to a compiler that manipulates high-level mathematical expressions and generates optimised C output. The DSL is embedded in Python and provides abstractions at the level of tensor algebra. Calculations of basis functions and quadrature tabulation are performed inside the compiler automatically.

There has been significant research into the use of DSLs for improving FPGA design productivity [3], ranging from those that expose a general compute model to those that target specific applications such as packet filtering, image and signal processing. For example, the Delite [4] compiler framework has been used to transform functional descriptions to hardware designs via pattern matching. Our work primarily targets optimisations that require finite difference domain specific knowledge while addressing hardware synthesis, whereas Delite targets more general optimisations.

Schmitt et al. have explored using DSLs to generate multi-grid solvers for FPGAs [5]. The DSLs appear to be primarily used to encode Multigrid solver algorithms rather than specification of high-level equations themselves.

Maxeler's MaxGenFD library [6] targets high-performance stencil implementations for finite difference problems. It does not attempt to generate stencils from high-level descriptions, does not support non-linear terms, and boundary conditions support is specific to seismic problems. We are not aware of any work that attempts to target finite difference hardware designs from a high-level description.

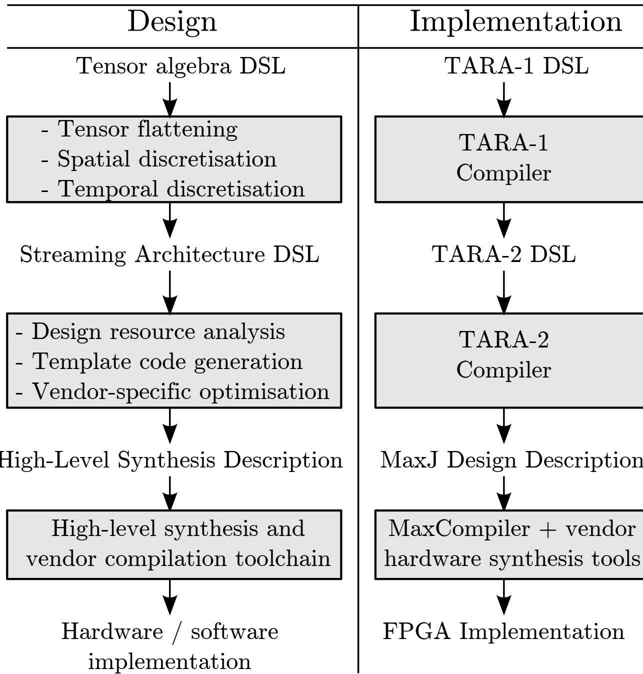


Fig. 1. The conceptual design flow we target and the specific implementation we present.

Although not a focus of this paper, extensive research exists on the optimisation of stencil operations on FPGAs which is relevant to the future development of our hardware design back end [7], [8], [9].

III. Approach

Our approach consists of two levels of DSLs. A top-level DSL designed to permit expressions of finite difference solvers in a notation familiar to scientists (TARA-1) and a lower-level internal DSL intended to target streaming architectures (TARA-2). Both our conceptual and implementation design flow are shown in Figure 1. Ideally, the user of our compiler toolchain does not need to be aware of TARA-2, and all compilers in the workflow are invoked automatically by the build system. In practice, enabling access to TARA-2 and other lower-level representations may enable optimisations not yet implemented automatically to be performed manually, depending on user expertise. In addition, having the TARA-1 compiler output its internal representations during execution has proven useful for verifying the correctness of its rewrites.

The TARA-1 *DSL*

The specification language for our top-level DSL is inspired by the Unified Form Language [2] (UFL) developed for the FEniCS project for specifying finite-element discretisations of PDEs. For a consistent syntax, it resembles Python, but has no imperative features.

We show a simple example of the TARA-1 DSL for the heat equation in Figure 2.

The types that can be declared are as follows:

```
# Constants
h = NamedLiteral(name="h", value=0.014)
dt = NamedLiteral(name="dt", value=0.1)
alpha = NamedLiteral(name="alpha", value=1e-4)
n = NamedLiteral(name="n", value=253)

# Field and update
u = Field(name="heat", rank=0)
heat_eq = Equation(Dt(u), alpha * div(grad(u)))

# Boundary conditions and solve
source = BoundaryCondition(u, 1,
  subdomains=["top", "bottom", "left"])
sink = BoundaryCondition(u, 0,
  subdomains=["right"])
step = Solve(name="step", spatial_order=1,
  temporal_order=1, equations=[heat_eq],
  boundary_conditions=[source, sink], delta_t=dt)

# Mesh
m = Mesh(name="HeatSolver", dim=2, fields=[u],
  solves=[step], spacing=[h,h], dimensions=[n, n])
```

Fig. 2. The heat equation specified in the TARA-1 DSL. A heat source is placed at the top, left and bottom edges and the right edge acts as a sink.

Field Fields are tensor-valued quantities that vary over a mesh. Although their rank (scalar, vector etc.) is defined, dimension is not. The components of a field can be staggered in each mesh direction which is required by some numerical methods to achieve stability (Figure 5).

BoundaryCondition These enable Dirichlet or Neumann boundary conditions to be specified for subsets of the mesh's edge domain.

MeshConstant These are tensor-valued quantities that do not vary over a mesh, but may be changed between mesh updates. These can be used to specify quantities that may vary across runs.

NamedLiteral These are scalar-valued quantities that will never change. Although constants can be declared directly, using a NamedLiteral allows their identity to be preserved throughout the code generation pipeline.

Equation These specify how to calculate the temporal derivative for a given field. Constants and other fields can be referenced, and derivatives applied to fields.

Solve Solves contain equations to update at least one field on a mesh. Solves specify the accuracy with which fields and their spatial derivatives must be evaluated, the temporal accuracy of field updates and any boundary conditions to be applied.

Mesh Meshes are discretisations of physical space on which fields are defined. Meshes possess a list of fields and the associated solves for those fields. Meshes specify the problem dimension rather than fields, enabling equations to be written dimension agnostically. Grid-point spacing and mesh size are also specified which are important for leveraging optimisations when these values are known constants.

In the TARA-1 DSL, we facilitate the expression of field updates using vector calculus, providing operators such as grad (∇) and div ($\nabla\cdot$). This makes specifications both more

concise and less error-prone since vector calculus operators may expand to relatively complex scalar expressions. Additionally, vector calculus operators have physical interpretations which are obscured when flattened to scalar derivatives.

Transforming to a streaming architecture

We now describe the mechanism by which our DSL can be transformed to a streaming architecture design.

Tensor Algebra Flattening: In this step, a scalar expression is derived for each element of the tensor expression forming the right hand side of a field update.

- Field and constant references are expanded to tensors of references to their scalar elements.
- Tensor algebra operators such as *grad* and *div* are expanded to tensors of derivative operators then applied to their operands.
- The rank of the resulting expression is checked against the left-hand side of the field updates.

After this step, a symbolic expression has been computed for each scalar element of the temporal derivative of each field. In all expressions, spatial derivatives are applied to scalar valued-expressions.

Discretising spatial derivatives: In this step, spatial derivatives are transformed into discretised expressions referring to the mesh on which the equations are being solved. This involves the transformation of the symbolic expressions constructed in the previous step:

1) For each scalar expression, derivatives are pushed to the bottom of the tree of each expression. This is done through application of the sum, product and quotient rules. Since only fields can have non-zero spatial derivatives, all derivative operations end up applied to either fields or derivatives of fields.
2) Expressions representing zero or more spatial derivatives applied to an element of a field are rewritten to a new type of terminal which incorporates the number of derivatives taken in each dimension.
3) Based on the order of spatial accuracy and degree of derivative required in each dimension, Lagrange polynomials are constructed over symbolic field grid-point values. These are then symbolically differentiated and partially evaluated at a location dependent on grid-staggering.
4) The terminals constructed in step 2 are replaced by the stencil expressions derived in step 3.

We draw attention to two important aspects. Firstly, grid-staggering [10]. This is a technique used to avoid discretisation errors in velocity-pressure simulations. Typically, values for a scalar quantity will be stored in the centre of a grid but velocity components will be defined on cell faces. As any tensor component of a field may be staggered in multiple dimensions, a vector field in d dimensions has 2^{d^2} possible staggerings (512 for a 3D vector field).

The stencil generated for a field reference depends not just on the order of spatial accuracy required, but the relative staggering between the field being referenced and the field being updated on the left-hand side of an expression. For example, given a height field h, and a velocity field whose components are u and v, calculating the value of v could require different stencils depending on whether it is being referenced from an expression updating v, h or itself. For this reason, it is important to maintain abstractions above the stencil level to adequately support finite difference techniques.

Secondly, we note that it is sometimes assumed that stencil-type updates are merely a weighted combination of surrounding points. While this supports linear operators, more complex equations contain non-linear terms, requiring more general expression support; the convective acceleration term of Navier-Stokes is an example of this.

Deriving time-stepping: We target explicit finite difference. We automatically derive Adams-Bashforth co-efficients for temporal accuracy of a specified order by symbolically integrating the Lagrange polynomial constructed over the previously computed temporal derivatives of a field.

Boundary conditions: TARA-1 provides a way to associate Dirichlet or Neumann boundary conditions to edge domains. Currently, the only edge domains supported are the edges of the mesh itself. General support of boundary conditions typically requires solving a linear system to determine how values should be assigned to ghost points, or determining the coefficients of alternative stencils that are used for updates in proximity of a boundary condition. Higher-order boundary conditions are especially problematic from a performance perspective since they may require significant resources (if implemented in hardware) but are used infrequently.

Due to the complexity and potential impact of such design choices, we have chosen to first implement boundary conditions for low-order problems, which we know can be generated efficiently. Dirichlet and Neumann boundary conditions are translated to a set of directives that specify a subset of a mesh row or column to apply an operation to which consists of a simple update to a constant value or offset adjacent value. This approach is temporary, with the longer-term aim to symbolically derive appropriate expressions for boundary conditions and incorporate them directly into the update stencil.

Design synthesis from the TARA-2 *DSL*

We treat the step of converting the discretised model into a hardware design as a distinct step. We have created an independent tool for this process that can be used outside of the overall tool-chain. The input for this tool chain is a second DSL (TARA-2) designed to be restrictive enough to encourage inputs conducive to creating performant hardware but permissive enough to support most inputs for 2D stencil finite difference calculations.

We assume the existence of a single 2D regular grid. We then require a description of the state that must be preserved across time steps. This state consists only of a list of scalars that are defined at each point on the grid. While the input to the tool-chain may have described a mesh where some variables are spatially staggered or are non-scalar quantities,

this design forces all the fields to be aligned to a single grid and transformed to scalars for the purpose of computation.

As the single grid now contains all of the data, the process of creating a hardware design is simplified – the design can step through the single grid, updating all the fields simultaneously.

Next, we require a description of how each field is updated. This is done by specifying a scalar expression used to update each point in the field (the stencil update), with all updates treated as happening in parallel. These statements ignore boundary conditions as these are handled separately. Fields may be referenced in the assignment, but when used they represent the value that the field has at the point being updated before the current update occurred. Values of fields at other locations in the grid can be accessed through use of the *Offset* operator. This takes the field and a relative position, and returns the value of the field at the location relative to the current point. The *Current* operator returns the value of its argument would take if computed after the update has finished, facilitating some expression reuse across variable updates. In this way, we support arbitrary stencil-like operations including those that involve multiple fields and non-linear terms.

We permit alternative update expressions to be used for the initial model updates, and an expression which can be used to initialise each cell variable. This is required for spin-up when using higher-order time stepping schemes, as previous derivatives have not yet been computed. Since the initial value generation and update steps require negligible compute, these are performed on the CPU.

Finally, we require a mechanism for applying boundary conditions. Boundary conditions at the higher level are translated to a set of update operations that can be implemented efficiently in hardware. Each operation is described by which grid-points it applies to and what action occurs at that location. The location is described as either a single row or column in the grid, and a start and end index within that group. Boundary conditions are applied after the stencil update step is complete. The operation may simply replace the value with a constant, or use an offset operation applied to a neighbouring point value to calculate a new value. Though this approach works for low-order boundary conditions, a different approach may be necessary for higher-order scenarios.

Using the description above, we can generate a hardware design. From the way the DSL is constructed, it becomes simple to devise a kernel that streams in the fields in grid order, and then streams out the fields after a single update. These kernels can be chained together such that n kernels produce the state after n updates.

We can therefore produce a template hardware design for models described by this DSL. This includes the kernel, host communication, and host code. The information included in the DSL can then be incorporated into this template to generate the working hardware design. The final step is to run the hardware design through the standard hardware and software compilation tools to generate an executable file.

TARA-2 is still general enough to permit code generation for multiple architectures (C is also implemented) and also provides a level at which optimisations may be implemented that could not be applied further along the toolchain. Finite difference stencil expressions are amenable to common-subexpression elimination (CSE) optimisations that target properties of polynomials and would not be applied by default in either software compilers or hardware synthesis tools. Maintaining coefficients as rationals also enables more effective CSE than is possible after conversion to floating point representations [11]. At this step, it would be possible to extract expensive operations from field expressions (e.g. trigonometric functions) and convert them to constant field expressions. Such an optimisation is beyond the scope of a high-level synthesis tool, especially if the expressions must be evaluated on the host at run-time before hardware execution.

IV. Implementation

We have implemented the approach described in Section III in a prototype compiler written in Haskell. For parsing TARA-1 we use the Parsec parser combinator library. For manipulation of symbolic expressions we use our own representation with sum and product representations based on maps from expressions to coefficients or powers, respectively. This was inspired by GiNaC [12].

Our top-level compiler is capable of generating expressions involving tensor fields of arbitrary rank and dimension, and calculates discretised expressions for approximating derivatives and explicit time-stepping for arbitrary orders of accuracy.

Throughout our symbolic manipulation, we retain coefficients as rational values wherever possible. Previous work [11] has shown that this has two benefits: firstly, we do not incur any form of floating point rounding errors during our expression manipulation; secondly, maintaining the exact values of coefficients facilitates more effective CSE which is beneficial for reducing hardware resource requirements. Currently our TARA-2 compiler does not directly support rationals or common sub-expression elimination, but could be modified to do so in future.

Our main limitations currently surround the treatment of boundary conditions. As mentioned in Section III, we limit ourselves to lower order boundary conditions that can be translated to a set of directives that are efficiently implementable by the TARA-2 compiler.

Our TARA-2 compiler takes an input designed to be efficiently compilable to streaming architectures. FPGAs are targeted via the Maxeler hardware synthesis toolchain and conventional CPUs via C. Due to our description's specificity, we expect that we could generate efficient GPU implementations via OpenCL or CUDA, and hardware implementations via other high-level synthesis tools (e.g. Vivado HLS) in future.

The TARA-2 DSL is intended to support scientific simulations, but is much more low-level than TARA-1. It is implemented as an embedded subset of Haskell. For the heat equation example in Figure 2, we provide an example of the TARA-2 DSL in Figure 3. The main properties include:

```
heat = CellVariable "heat"
heat_dt0 = CellVariable "heat_dt0"

model = do
  setWidth (253 + 3)
  setHeight (253 + 3)
  addUpdate "heat" (heat + Current heat_dt0 * 0.1)
  addUpdate "heat_dt0" (25.0 / 49.0 *
    (heat * (-4.0) + Offset heat (-1) 0 +
    Offset heat 0 (-1) + Offset heat 0 1 +
    Offset heat 1 0))
  setBC "heat" Vertical 1 1 254 (SetValue 1.0)
  setBC "heat" Horizontal 254 1 254 (SetValue 1.0)
  setBC "heat" Horizontal 1 1 254 (SetValue 1.0)
  setBC "heat" Vertical 254 1 254 (SetValue 0.0)
```

Fig. 3. Excluding boilerplate, the TARA-2 DSL showing declarations and updates for the heat equation.

No tensor support Scalar-valued fields are defined through the definition of **CellVariable**s. Tensor-valued fields must be flatted to their scalar components, simplifying indexing.

Static allocation All values associated with a cell including any derivatives that need storage must be given a name, making it trivial to synthesize appropriate data structures. These are declared with the **CellVariable** keyword. The **setWidth** and **setHeight** keywords define the mesh size. Currently these must be constant which enables simplifications of indexing in hardware.

General stencil syntax Updates are specified for variables using the **addUpdate** keyword to declare how each scalar element of a field will be calculated. Such expressions can use both mesh offsets and references to other variables. These expressions are constructed using the **Offset** keyword.

No time-stepping There is no concept of time-stepping at this level so the bookkeeping to manage access to the last n derivatives of a field by a time-stepping scheme must be handled by declaring appropriate **CellVariable**s. The distinction between values calculated during the previous or current iteration is maintained however, to facilitate reuse of common sub-expressions between field updates. The **Current** keyword can be used to access expressions for **CellVariable**s at the current iteration.

No discretisation Aspects of temporal and spatial discretisation such as time-step size and grid-point interval must be handled by incorporating these values where necessary in the expressions supplied to the DSL. Staggered fields are handled by generating the appropriate expressions.

Boundary conditions Boundary conditions are mapped to a fixed set of primitives that can be synthesised efficiently on the FPGA. The **setBC** keyword specifies that a range of the entries of a particular row or column should be overwritten with a specified expression. **SetValue** defines a constant whereas **NegateValueAwayFromZero** negates the adjacent value in the away-from-zero direction. Other directives exist, but we do not expand upon them. We intend to switch to the current update syntax for them in future.

```
-- Similarly for u_0 (x) and u_1 (y) fields
h = CellVariable "h"
h_dt0 = CellVariable "h_dt0"
h_dt1 = CellVariable "h_dt1"

model = do
  setWidth (128 + 3)
  setHeight (128 + 3)
  -- Directives for h and u_1 omitted
  addUpdate "u_0" (u_0 +
    u_0_dt0 * (-40.0) / 3.0 +
    u_0_dt1 * 25.0 / 6.0 +
    Current u_0_dt0 * 115.0 / 6.0)
  addInitialUpdate "u_0" 2 (u_0 +
    Current u_0_dt0 * 10.0)
  addUpdate "u_0_dt0" (h * 7.6640625e-5 +
    Offset h 1 0 * (-7.6640625e-5) + u_0 *
    (u_0 + (-Offset u_0 (-1) 0)) * (-7.8125e-6) +
    Offset u_1 0 (-1) * (u_0 +
    (-Offset u_0 0 (-1))) * (-7.8125e-6))
  addUpdate "u_0_dt1" u_0_dt0
  setBC "u_0" Vertical 0 1 129 NegateValueAwayFromZero
  setBC "u_0" Vertical 129 1 129 NegateValueTowardsZero
  setBC "u_0" Horizontal 129 0 129 (SetValue 0.0)
  setBC "u_0" Horizontal 1 0 129 (SetValue 0.0)
```

Fig. 4. A redacted form of the TARA-2 DSL showing height-field declarations and updates for the shallow water equations. Updates and declarations for the velocity field have been omitted.

The output of the TARA-2 compiler includes both an input program to the Maxeler high level synthesis toolchain (which is then compiled to a hardware design), and host code which allows a CPU to interface with an FPGA and use the synthesised bitstream. C output is also supported which is used for our CPU benchmarks.

V. Evaluation

To evaluate our compiler, we compile and execute an FPGA design for two different PDEs. First, the heat equation:

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u \tag{1}$$

A representation of this equation in the TARA-1 DSL is shown in Figure 2 and the TARA-2 DSL is shown in Figure 3. Boundary conditions are applied so that the left, top and bottom edges of the mesh are a heat source and the right edge is a sink. We run all experiments on a mesh of 253×253 cells, and employ first-order accurate spatial discretisation and time-stepping. Euler timestepping requires calculation but not storage of the temporal derivative, so for our benchmarks, we manually modify Figure 3 to inline the *heat_dt0* value directly into the update expression. This optimisation can easily be done automatically, but is done manually due to the way our TARA-1 compiler currently outputs time-stepping expressions.

Our second PDE is a simplified version of the shallow water equations (SSWE). Specifically, we neglect the kinematic viscosity term, but note that the equation remains non-linear.

```
# Definitions (g, H and dt omitted)
u = Field(name="v", rank=1,
  spatial_staggering="dimension")
h = Field(name="h", rank=0)
nu = NamedLiteral(name="nu", value=0.0)
hx = NamedLiteral(name="hx", value=128000.0)
nx = NamedLiteral(name="nx", value=128)

V_Eq = Equation(Dt(u), nu * div(grad(u))
  -dot(u, grad(u)) - g * grad(h))
H_Eq = Equation(Dt(h), -div((h + H) * u))
V_BC = BoundaryCondition(u, [0, 0])
H_BC = BoundaryCondition(Dn(h), 0)

TimeStep = Solve(name="step", spatial_order=1,
  temporal_order=3, equations=[V_Eq, H_Eq],
  boundary_conditions=[V_BC, H_BC], delta_t=dt)

sigmax = pow(pos[0] - (nx * hx) / 2.0, 2) /
  pow(3.0 * nx * hx / 20.0, 2)
sigmay = pow(pos[1] - (nx * hx) / 2.0, 2) /
  pow(3.0 * nx * hx / 20.0, 2)

m = Mesh(name="shallow_water", dim=2, fields=[u, h],
  solves=[TimeStep], spacing=[hx, hx],
  dimensions=[nx, nx],
  initial=[("h", 100 * exp(-sigmax - sigmay))])
```

Fig. 5. A TARA-1 DSL description showing declarations and updates for a simplification ($\nu = 0$) of the shallow water equations. A no-slip boundary condition is applied to the velocity at the edge of the domain. The normal derivative of the height field is set to 0 at the edges. A function is specified that is used to initialise the height field so that it has a central peak.

$$\frac{\partial h}{\partial t} = -\nabla \cdot ((H + h)v) \tag{2}$$

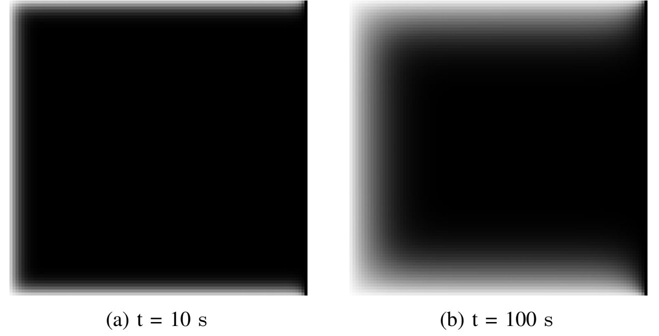
$$\frac{\partial v}{\partial t} = -g\nabla h - u \cdot \nabla u \tag{3}$$

We show the TARA-1 DSL representation of the simplified shallow water equations in Figure 5. We use a grid-size of 128×128 cells, with first-order accurate spatial discretisation and third-order accurate temporal discretisation (Adams-Bashforth). The horizontal and vertical velocity components are staggered in their respective directions, giving an Arakawa C-grid scheme [10].

We compile our DSL descriptions to hardware designs and execute on actual hardware. Details of hardware resource utilisation are shown in Table I. We also show performance results of both our hardware builds and software implementations generated from our TARA-2 DSL descriptions in Table II.

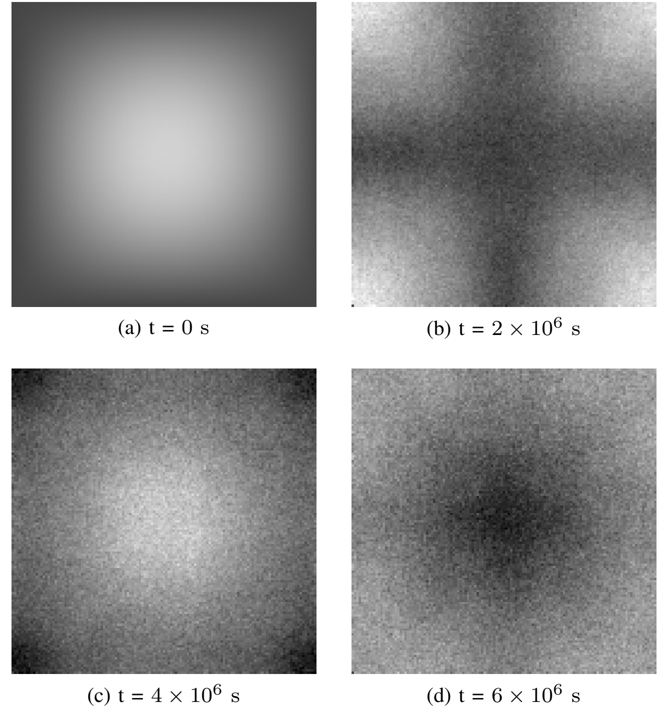
We show field renderings from the heat conduction and simplified shallow water executions in Figures 6 and 7, respectively.

All performance tests are run on a Maxeler MaxStation containing a Maxeler Max3 DFE (data-flow engine) with a Xilinx Virtex-6 SX475T FPGA (40 nm), a quad-core Intel Core i7-870 processor (45 nm), and 16 GiB of DDR3-1600 RAM. Static power requirements are measured at 95.6 W. Static and dynamic power measurements are taken using an ammeter placed in the path of the host machine's power supply. When compiling FPGA bitstreams we use MaxCompiler 2015.2.



(a) t = 10 s (b) t = 100 s

Fig. 6. Renderings of the heat field generated by executing a hardware design for our heat solver generated from a DSL description. The top, left and bottom boundaries act as a heat source and the right boundary as a sink.



(a) t = 0 s (b) t = 2×10^6 s (c) t = 4×10^6 s (d) t = 6×10^6 s

Fig. 7. Renderings of the height field generated by executing a hardware design for our simplified shallow water solver generated from a DSL description. The initial conditions define a curved peak of water in the centre of the region which reflects off the edges of the domain.

We attempt to build designs with different numbers of kernels at 150 MHz and choose the largest design that successfully compiles. We then build that design with a range of clock rates (multiples of 10 MHz) to determine the highest achievable clock rate. In a fully automated process, these values could be chosen using exponential search. During place and route we build up to 9 cost tables for each design, and try a lower clock rate if timing is not passed. For SSWE we build with both the maximum number of kernels that fit on the device, and with 1 fewer kernel. For heat conduction we build the largest number of kernels that fit and is a multiple of 5, and with 5 fewer kernels. We use a smaller number of kernels since a

Design	LUTs	DSPs	BRAM	Number of Kernels	Clock Frequency (MHz)
SSWE (FPGA, 11 mantissa, 8 exponent)	203422 (68.4%)	1008 (50.0%)	567 (26.6%)	14	200
SSWE (FPGA, 24 mantissa, 8 exponent)	183947 (61.8%)	1840 (91.3%)	567 (26.6%)	10	190
Heat (FPGA, 11 mantissa, 8 exponent)	210309 (70.7%)	360 (17.9%)	444 (20.9%)	90	190
Heat (FPGA, 24 mantissa, 8 exponent)	202177 (67.9%)	1200 (59.5%)	299 (18.7%)	75	150

TABLE I
HARDWARE RESOURCE UTILISATION OF OUR HEAT AND SIMPLIFIED SHALLOW WATER SOLVERS. EACH DESIGN IS PIPELINED SUCH THAT MULTIPLE SUCCESSIVE TIMESTEPS ARE EVALUATED SIMULTANEOUSLY WITH EACH KERNEL HANDLING A SINGLE TIMESTEP UPDATE.

Design	Time / cell update (µs)	Time improvement	Dynamic Power (W)	Energy / cell update (nJ)	Energy improvement	GFLOP/s
SSWE (FPGA, 11 mantissa, 8 exponent)	6.15	6.634x	93.74	0.576	22.7x	200.8
SSWE (FPGA, 24 mantissa, 8 exponent)	11.1	3.675x	100.1	1.111	11.8x	111.3
SSWE (CPU, single precision, single-core)	326	0.125x	40.26	13.12	1x	3.790
SSWE (CPU, single precision, ideal 8-core)	40.8	1x	N/A	N/A	N/A	30.32
Heat (FPGA, 11 mantissa, 8 exponent)	3.86	4.171x	89.3	0.344	14.7x	118.8
Heat (FPGA, 24 mantissa, 8 exponent)	5.86	2.747x	85.3	0.500	10.1x	78.29
Heat (CPU, single precision, single-core)	129	0.125x	39.32	5.07	1x	3.556
Heat (CPU, single precision, ideal 8-core)	16.1	1x	N/A	N/A	N/A	28.45

TABLE II
PERFORMANCE AND ENERGY UTILISATION RESULTS FOR THE HEAT AND SIMPLIFIED SHALLOW WATER SOLVERS RUNNING ON THE CPU IN SINGLE PRECISION, AND ON FPGA HARDWARE IN SINGLE AND REDUCED PRECISION. RESULTS WERE COLLECTED OVER AT LEAST A MINUTE OF WALL-CLOCK TIME.

design with fewer kernels can achieve higher performance if it achieves a higher clock rate. Each kernel corresponds to a single timestep update so the number of kernels in a design indicates the number of timesteps being processed in parallel in the pipeline.

We make use of the FPGA's flexibility to support non-standard precision by building designs with 11 mantissa bits which maintains numerical stability over extended runs (several simulated days). We use single precision and also build hardware at single precision to allow for a direct comparison. Mesh data is stored *off-chip* in DRAM located on the Maxeler board and therefore are not limited to the available BRAM on the FPGA. Mesh sizes may be arbitrary (so long as the mesh data is large enough to fill the pipeline and fit in DRAM). Since mesh data is written to/from DRAM, simulation does not require any significant communication with the host. Designs are built so that they continuously send mesh data to the host system, which is used to produce the renderings in Figures 6 and 7.

Our CPU implementation is single-threaded and generated from the TARA-2 description. The structure of the code is consistent with finite difference implementations using hand-written stencil updates but higher performance would likely be achieved if a stencil compiler such as Pochoir [13] could be leveraged. Code is compiled using version 12.1.4 of the Intel C compiler with the '`-Ofast -no-prec-div -ipo`' flags. For time comparisons we show artificial speed results for an 8-core implementation where each core achieves the same performance as the single-core implementation (perfect scaling).

Updating a single cell in the simplified shallow water equations (SSWE) model requires 72 floating-point operations (36 multiplications, 28 additions and 8 subtractions). We find that the 131×131 simplified shallow water hardware implementation is able to update 2790 million cells per second (200.8 GFLOP/s), whereas the single-core CPU is able to update 52 million cells per second (3.79 GFLOP/s).

For the heat model we use a grid with 256×256 grid points. There are 7 FLOPs per cell update (2 multiplications and 5 additions). The hardware implementation is able to update 16978 million cells per second (118.8 GFLOP/s) whereas the single-core CPU is able to update 508 million cells per second (3.556 GFLOP/s)

For SSWE, the FPGA requires 97.4 W of dynamic power, hence 0.576 nJ per cell update, whereas the single-core CPU requires 40.26 W of dynamic power, hence 13.12 nJ per cell update. For the heat model the FPGA requires 89.3 W of dynamic power (0.344 nJ per cell update), whereas the single core CPU requires 39.32 W of dynamic power (5.07 nJ per cell update).

VI. DISCUSSION AND INSIGHTS

The design of existing software DSLs for code generation of PDEs inspires the design of our own top-level DSL. However, through the construction of our compilation toolchain, we have gained knowledge of hardware-specific concerns affected by the implementation of our DSLs.

Since the evaluation of stencils is the most computationally intensive part of finite difference schemes, these are typically targeted for code generation. In software, boundary conditions are more likely to be handled by hand-written code. However, requiring a scientist to edit lower-level representations defeats the purpose of our DSL. In addition, we have chosen to synthesise hardware for boundary conditions rather than handling them via communication with the host system. Doing

so would potentially limit the performance of designs depending on the latency of communication, particularly in cases where the iteration rate is valued over mesh size. Requiring a scientist to edit a hardware description of a boundary condition implementation is particularly undesirable. Hence we find that a high-level specification of boundary conditions is an important requirement for our DSL.

The TARA-1 DSL specifically allows specification of mesh size and grid-spacing. These can be chosen to either be constant or runtime-specified values. Choosing constant values for grid spacing allows us to specialise kernels and reduce hardware requirements. Specifying a constant grid size simplifies the hardware required to perform indexing operations and allows determining the exact memory required for grid buffering, again enabling a reduction in logic. Specialising these values in software has little benefit and so are typically handled as variables. We have deliberately designed our DSL to incorporate the specification of mesh properties which would typically not be included in a finite difference stencil description. Enabling flexibility in specification of these and other values makes it possible for the user to choose between run-time flexibility or exploiting hardware-specific optimisations, depending on requirements.

VII. Conclusion

We have described the design principles and implementation of a toolchain that can take a high-level specification of a finite difference problem and synthesize a hardware design for a solver.

This approach reduces the time taken to modify the key aspects of a finite difference hardware solver such as equation terms, spatial and temporal order of accuracy, field staggering and boundary conditions, to a matter of seconds, with no hardware-specific knowledge required. Without this approach, it would typically this would require someone with both knowledge of hardware synthesis and the underlying mathematics to make these changes in hours or days. Our proof-of-concept toolchain demonstrates that it is practical to rapidly iterate on such designs, significantly increasing productivity.

We have presented examples of the TARA-1 and TARA-2 DSLs for heat conduction and for simplified shallow water test-cases, along with the rationale behind their design. Through a multi-level compiler toolchain, we generate hardware designs and show hardware utilisation metrics, performance and power consumption. Pipelining enables high compute intensity to be achieved on computations that are typically memory-bound, demonstrating that our approach is also practical from a performance perspective.

Through development of the toolchain, we gain much understanding of the requirements of these DSLs when used to target hardware designs, in particular the need for flexibility in aspects that allow hardware designs to be specialised and the need to fully encompass certain problem characteristics such as mesh spacing and time-stepping scheme inside our DSL.

Future work includes support for three-dimensional problems, generation of boundary conditions for higher-order approximations, applying our approach to larger scale models and developing debugging support for synthesized hardware [14]. With FPGA hardware now available in the cloud, multi-FPGA implementations are of particular interest.

Acknowledgement

This work is supported in part by the European Union Horizon 2020 research and innovation programme (grant number 671653), by the UK EPSRC (EP/N031768/1, EP/I012036/1, EP/L00058X/1 and EP/K503733/1), by the Maxeler University Programme, by the HiPEAC NoE, by Intel, and by Xilinx.

References

[1] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D. J. Lee, "Real-time optical flow calculations on FPGA and GPU architectures: A comparison study," in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, Apr. 2008, pp. 173–182.

[2] M. S. Alnæs, A. Logg, K. B. Ölgaard, M. E. Rognes, and G. N. Wells, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *ACM Trans. Math. Softw.*, vol. 40, no. 2, pp. 9:1–9:37, Mar. 2014.

[3] N. Kapre and S. Bayliss, "Survey of domain-specific languages for FPGA computing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2016, pp. 1–12.

[4] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. D. Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 651–665.

[5] C. Schmitt, M. Schmid, F. Hannig, J. Teich, S. Kuckuk, and H. Köstler, "Generation of multigrid-based numerical solvers for FPGA accelerators," in *Proceedings of the 2nd International Workshop on High-Performance Stencil Computations (HiStencils)*, A. Größlinger and H. Köstler, Eds., pp. 9–15.

[6] Maxeler Technologies, "MaxgenFD white paper," https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxGenFD.pdf, 2012, [Online; accessed 24-April-2018].

[7] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 695–705, Mar. 2014.

[8] G. Deest, N. Estibals, T. Yuki, S. Derrien, and S. Rajopadhye, "Towards Scalable and Efficient FPGA Stencil Accelerators," in *IMPACT'16 - 6th International Workshop on Polyhedral Compilation Techniques, held with HIPEAC'16*, Prague, Czech Republic, Jan. 2016.

[9] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 26:1–26:25, Aug. 2017.

[10] R. J. Purser and L. M. Leslie, "A semi-implicit, semi-lagrangian finite-difference scheme using hligh-order spatial differencing on a nonstaggered grid," *Monthly Weather Review*, vol. 116, no. 10, pp. 2069–2080, 1988.

[11] F. P. Russell and P. H. J. Kelly, "Optimized code generation for finite element local assembly using symbolic manipulation," *ACM Trans. Math. Softw.*, vol. 39, no. 4, pp. 26:1–26:29, Jul. 2013.

[12] C. Bauer, A. Frink, and R. Kreckel, "Introduction to the GiNaC framework for symbolic computation within the C++ programming language," *Journal of Symbolic Computation*, vol. 33, no. 1, pp. 1–12, 2002.

[13] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 117–128.

[14] J. Goeders and S. J. E. Wilton, "Effective FPGA debug for high-level synthesis generated circuits," in *24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–8.