# ForOpenCL: Transformations Exploiting Array Syntax in Fortran for Accelerator Programming

Matthew J. Sottile
Galois, Inc.
Portland, OR 97204
mjsottile@computer.org

Craig E Rasmussen
Los Alamos National Laboratory
Los Alamos, NM 87545
crasmussen@lanl.gov

Wayne N. Weseloh
Los Alamos National Laboratory
weseloh@lanl.gov

Robert W. Robey
Los Alamos National Laboratory
brobey@lanl.gov

Daniel Quinlan
Lawrence Livermore National Laboratory
dquinlan@llnl.gov

Jeffrey Overbey
University of Illinois at Urbanna-Champaign
overbey2@illinois.edu

## ABSTRACT

Emerging GPU architectures for high performance computing are well suited to a data-parallel programming model. This paper presents preliminary work examining a programming methodology that provides Fortran programmers with access to these emerging systems. We use array constructs in Fortran to show how this infrequently exploited, standardized language feature is easily transformed to lower-level accelerator code. The transformations in ForOpenCL are based on a simple mapping from Fortran to OpenCL. We demonstrate, using a stencil code solving the shallow-water fluid equations, that the performance of the ForOpenCL compiler-generated transformations is comparable with that of hand-optimized OpenCL code.

## 1. INTRODUCTION

This paper presents a compiler-level approach for targeting a single program to multiple, and possibly fundamentally different, processor architectures. This technique allows the application programmer to adopt a single, high-level programming model without sacrificing performance. We suggest that existing data-parallel features in Fortran are well-suited to applying automatic transformations that generate code specifically tuned for different hardware architectures using low-level programming models such as OpenCL. For algorithms that can be easily expressed in terms of whole array, data-parallel operations, writing code in Fortran and transforming it automatically to specific low-level implementations removes the burden of creating and maintaining multiple versions of architecture specific code.

The peak performance of these newer accelerator architectures can be substantial. Intel expects a teraflop for the SGEMM benchmark with their Knights Ferry processor while the performance of the M2090 NVIDIA Tesla processor is in the same neighborhood [7]. Unfortunately the performance that many of the new accelerator architectures offer comes at a cost. Architectural changes are trending toward multiple heterogeneous cores and less of a reliance on superscalar instruction level parallelism and hardware managed memory hierarchies (such as traditional caches).

These changes place a heavy burden on application programmers as they work to adapt to these new systems. An especially challenging problem is not only how to program to these new architectures — considering the massive scale of concurrency available — but also how to design programs that are portable across the changing landscape of computer architectures. How does a programmer write one program that can perform well on both a conventional multicore CPU *and* a GPU (or any other emerging many-core architectures)?

A directive-based approach, such as OpenMP or the Accelerator programming model from the Portland Group [14], is one solution to this problem. However, in this paper we take a somewhat different approach. A common theme amongst the new processors is the emphasis on data-parallel programming. This model is well-suited to architectures that are based on either vector processing or massively parallel collections of simple cores. The recent CUDA and OpenCL programming languages are intended to support this programming model.

The problem with OpenCL and CUDA is that they expose too much detail about the machine architecture to the programmer [15]. The programmer is responsible for explicitly managing memory (including the staging of data back and forth between the host CPU and the accelerator device) and specifically taking into account architectural differences (such as whether the architecture contains vector units). While these languages have been attractive as a method for early adopters to utilize these new architectures, they are less attractive to programmers who do not have the time or resources to manually port their code to every new architecture and programming model that emerges.

## 1.1 Approach

We demonstrate that a subset of Fortran map surprisingly well onto GPUs when transformed to OpenCL kernels. This data-parallel subset includes: array syntax using assignment statements and binary operators, array constructs like WHERE, and the use of pure and elemental functions. In addition, we provide new functions that explicitly take advantage of the stencil geometry of the problem domain we consider. Note that this subset of the Fortran language is implicitly parallel. This programming model *does not require explicit declaration of parallelism within the program.* In addition, programs are expressed using entirely standard Fortran so it can be compiled for and executed on a single core without concurrency.

Transformations are supplied that provide a mechanism for converting Fortran procedures written in the Fortran subset described in this paper to OpenCL kernels. We use the ROSE compiler infrastructure[1] to develop these transformations. ROSE uses the Open Fortran Parser[2] to parse Fortran 2008 syntax and can generate C-based OpenCL. Since ROSE's intermediate representation (IR) was constructed to represent multiple languages, it is relatively straightforward to transform high-level Fortran IR nodes to C OpenCL nodes. This work is also applicable to transformations to vendor-specific languages, similar to OpenCL, such as the NVIDIA CUDA language.

Transformations for arbitrary Fortran procedures are not attempted. Furthermore, a mechanism to transform the calling site to automatically invoke OpenCL kernels is not provided at this time. While it is possible to accomplish this task within ROSE, it is considered outside the scope of this paper. However, ForOpenCL provides via Fortran interfaces a mechanism to call the C OpenCL runtime and enable Fortran programmers to access OpenCL kernels generated by the supplied transformations.

We study the automatic transformations for an application example that is typical of stencil codes that update array elements according to a fixed pattern. Stencil codes are often employed in applications based on finite-difference or finite-volume methods in computational fluid dynamics (CFD). The example described later in this paper is a simple shallow-water model in two dimensions using finite volume methods. Stencil-like patterns appear in a number of other contexts as well. In image processing, they appear in convolution-based algorithms in which small kernels are convolved with an image to implement denoising, edge detection, and other common operators. Similar stencil operators appear in general signal processing applications as well.

Finally, we examine the performance of the Fortran data-parallel abstraction when transformed to OpenCL to run on GPU architectures. The performance of automatically transformed code is compared with a hand-optimized OpenCL version of the shallow-water code.

We do not perform any additional analysis of the code to identify parallelism beyond that present in the data parallel operations that we focus on in this paper. Additional program analysis methods may be investigated to study their applicability in future versions of this work.

# 2. PROGRAMMING MODEL

A question that one may pose is "*Why choose Fortran and not a more modern language like X for programming accelerator architectures?*" The recent rise in interest in concurrency and parallelism at the language level driven by multicore CPUs and many-core accelerators has driven a number of new language developments, both as novel languages and extensions on existing ones. However, for many scientific users with existing codes written in Fortran, new languages and language extensions to use novel new architectures present a challenge: how do programmers effectively use them while avoiding rewriting code and potentially growing dependent on a transient technology that will vanish tomorrow? In this paper we explore the constructs in Fortran that are particularly relevant to GPU architectures.

In this section we present the Fortran subset employed in this paper. This sub-setting language will allow scientific programmers to stay within the Fortran language and yet have direct access to GPU hardware. We start by examining how this programming model relates to developments in other languages.

## 2.1 Comparison to Prior Fortran Work

A number of previous efforts have exploited data-parallel programming at the language level to utilize novel architectures. The origin of the array syntax adopted by Fortran in the 1990 standard can be found in the APL language [6]. These additions to Fortran allowed parallelism to be expressed with whole-array operations at the expression level, instead of via parallelism within explicit DO-loops, as implemented in earlier variants of the language (e.g., IVTRAN for the Illiac IV).

The High Performance Fortran (HPF) extension of Fortran was proposed to add features to the language that would enhance the ability of compilers to emit fast parallel code for distributed and shared memory parallel computers [10]. One of the notable additions to the language in HPF was syntax to specify the distribution of data structures amongst a set of parallel processors. HPF also introduced an alternative looping construct to the traditional DO-loop called FORALL that was better suited for parallel compilation. An additional keyword, INDEPENDENT, was added to allow the programmer to indicate when the loop contained no loop-order dependencies that allowed for parallel execution. These constructs are similar to DO CONCURRENT, an addition to Fortran in 2008.

Interestingly, the parallelism features introduced in HPF did not exploit the new array features introduced in 1990 in any significant way, relying instead on explicit loop-based parallelism. This restriction allowed the language to support parallel programming that wasn't easily mapped onto a pure data-parallel model. The SHADOW directive introduced in HPF-2, and the HALO in HPF+ [1] bear some similarity to the halo region concept that we discuss in this paper.

In some instances though, a purely data-parallel model is ap-

[1] http://www.rosecompiler.org/

[2] http://fortran-parser.sf.net/

propriate for part or all of the major computations within a program. One of the systems where programmers relied heavily on higher level operations instead of explicit looping constructs was the Thinking Machines Connection Machine 5 (CM-5). A common programming pattern used on the CM-5 (that we exploit in this paper) was to write whole-array operations from a global perspective in which computations are expressed in terms of operations over the entire array instead of a single local index. The use of the array shift intrinsic functions (like CSHIFT) were used to build computations in which arrays were combined by shifting the entire arrays instead of working on local offsets based on single indices. A simple 1D example is one in which an element is replaced with the average of its own value and that of its two direct neighbors. Ignoring boundary indices that wrap around, explicit indexing will result in a loop such as:

```
do i = 2,(n-1)
  Xnew(i) = (X(i-1) + X(i) + X(i+1)) / 3
end do
PTR_SWAP(Xnew, X, tmp_ptr)
```

In this loop, it is necessary to manually implement a double buffering scheme in order to avoid mixing values computed in the current loop with values from a prior execution of the loop. When shifts are employed, this can be expressed as:

```
X = (cshift(X,-1) + X + cshift(X,1)) / 3
```

The semantics of the shift intrinsic and operators like + applied to the whole array make it unnecessary to manually implement the double buffering scheme found in the loop above. Similar whole array shifting was used in higher dimensions for finite difference codes within the computational physics community for codes targeting the CM-5 system. Research in compilation of stencil-based codes that use shift operators targeting these systems is related to the work presented here [3].

The whole-array model was attractive because it deferred responsibility for optimally implementing the computations to the compiler. Instead of relying on a compiler to infer parallelism from a set of explicit loops, the choice for how to implement loops was left entirely up to the tool.

Unfortunately, this had two side effects that have limited broad acceptance of the whole-array programming model in Fortran. First, programmers must translate their algorithms into a set of global operations. Finite difference stencils and similar computations are traditionally defined in terms of offsets from some central index. Shifting, while conceptually analogous, can be awkward to think about for high dimensional stencils with many points. Second, the semantics of these operations are such that all elements of an array operation are updated as if they were updated simultaneously. In a program where the programmer explicitly manages arrays and loops, double buffering techniques and user managed temporaries are used to maintain these semantics. Limited attention to optimizing memory usage due to this intermediate storage by compilers has led to these constructs seeing little adoption by programmers.

An interesting line of language research that grew out of HPF was that associated with the ZPL language at the University of Washington [5] and Chapel, an HPCS language developed by Cray [4]. In ZPL, programmers adopt a similar global view of computation over arrays, but define their computations based on regions, which provide a local view of the set of indices that participate in the update of each element of an array. A similar line of research in the functional language community has investigated array abstractions for expressing whole-array operations in the Haskell language in the REPA (regular, shape-polymorphic, parallel array) library [8].

## 2.2 Fortran Language Subset

The static analysis and source-to-source transformations used in this work require the programmer to use a language subset that employs a data-parallel programming model. In particular, it encourages the use of array notation, elemental functions, and pure procedures. From these language constructs, we are able to easily transform Fortran procedures to a lower-level OpenCL kernel implementation.

### *Array notation*

Fortran has a rich array syntax that allows programmers to write statements in terms of whole arrays or subarrays, with data-parallel operators to compute on the arrays. Array variables can be used in expressions based on whole-array operations. For example, if A, B, and C are all arrays of the same rank and shape and s is a scalar, then the statement

```
C = A + s*B
```

results in the element-wise sum of A and the product of s times the elements of B being stored in the corresponding elements of C. The first element of C will contain the value of the first element of A added to the first element of c*B. Note that no explicit iteration over array indices is needed and that the individual operators, plus, times, and assignment are applied by the compiler to individual elements of the arrays independently. Thus the compiler is able to spread the computation in the example across any hardware threads under its control.

### *Pure elemental functions*

An elemental function consumes and produces scalar values, but can be applied to variables of array type such that the function is applied to each and every element of the array. This allows programmers to avoid explicit looping and instead simply state that they intend a function to be applied to every element of an array in parallel, deferring the choice of implementation technique to the compiler. Pure elemental functions are intended to be used for data-parallel programming, and as a result must be side effect free and mandate an intent(in) attribute for all arguments.

For example, the basic array operation shown above could be refactored into a pure elemental function,

```
pure elemental real function foo(a, b, s)
  real, intent(in) :: a, b, s
  foo = a + s*b
end function
```

and called with

```
C = foo(A, B, s)
```

Note that while `foo` is defined in terms of purely scalar quantities, it can be *applied* to arrays as shown. While this may seem like a trivial example, such simple functions may be composed with other elemental functions to perform powerful computations, especially when applied to arrays. Our prototype tool transforms pure elemental functions to inline OpenCL functions. Thus there is no penalty for usage of pure elemental functions and they provide a convenient mechanism to express algorithms in simpler segments.

It should be noted that in Fortran 2008, the concept of impure elemental functions was introduced. This requires that elemental functions now need to be explicitly labelled as pure to ensure that they are side effect free.

### *Pure procedures*

Pure procedures, like pure elemental functions, must be free of side effects. Unlike pure elemental functions that require arguments to have an `intent(in)` attribute, they may change the contents of array arguments that are passed to them. The absence of side effects removes ordering constraints that could restrict the freedom of the compiler to invoke pure functions out of order and possibly in parallel. Procedures and functions of this sort are also common in pure functional languages like Haskell, and are exploited by compilers in order to emit parallel code automatically due to their suitability for compiler-level analysis.

Since pure procedures don't have side effects they are candidates for running on accelerators in OpenCL. Currently our prototype tool only transforms pure procedures to OpenCL kernels that *do not* call other procedures, except for pure elemental functions, either defined by the user or intrinsic to Fortran.

## 2.3 New Procedures

Borrowing ideas from ZPL, we introduce a concept of a region to Fortran with a set of functions that allow programmers to work with subarrays in expressions. In Fortran, these functions return a copy of or a pointer to an existing array or array section. This is unlike ZPL, where regions are analogous to index sets and are used primarily for address resolution within an array without dictating storage related behavior. The functions that we propose are similar in that they allow a programmer to deal with index regions that are meaningful to their algorithm, and automatically induce a halo (or ghost) cell pattern as needed in the implementation generated by the compiler, where the size of an array is implicitly increased to provide extra array elements surrounding the interior portion of the array. It is important to note, however, that all memory allocated by the programmer must explicitly contain the extra array elements in the halo.

Region functions are similar to the shift operator as they can be used to reference portions of the array that are shifted with respect to the interior portion. However, unlike the shift operator, regions are not expressed in terms of boundary conditions and thus don't explicitly *require* a knowledge of, nor the application of, boundary conditions locally (global boundary conditions must be explicitly provided by the programmer outside of calls to kernel procedures). Thus, as will be shown below, regions are more suitable for usage by OpenCL thread groups which access only local subsections of an array stored in global memory.

ForOpenCL provides two new functions that are defined in Fortran and are used in array-syntax operations. Each function takes an integer array halo argument that specifies the number of ghost cells on either side of a region, for each dimension. For example `halo = [left, right, down, up]` specifies a halo for a two-dimensional region. These functions are:

- `region_cpy(array, halo)`: a pure function that returns a copy of the interior portion of the array specified by halo.
- `region_ptr(array, halo)`: an impure function that returns a pointer to the portion of the array specified by halo.

It should be noted that the function `region_cpy` is pure and thus can be called from within a pure kernel procedure, though it should be noted that `region_ptr` is impure because it aliases the array parameter. However as will be shown below, the usage of `region_ptr` is constrained so that it does not introduce side effects in the functions that call it. These two functions are part of the language recognized by the compiler and though `region_cpy` returns a copy of a portion of an array *semantically*, the compiler is not forced to actually make a copy and is free to enforce copy semantics through other means. In addition to these two new functions, ForOpenCL provides the compiler directive, `$OFP PURE, KERNEL`, which specifies that a pure subroutine can be transformed to an OpenCL kernel and that the subroutine is pure except for calls to `region_ptr`. These directives are not strictly necessary for the technique described in this paper, but aid in automated identification of specific kernels to be transformed to OpenCL. A directive-free implementation would require the transformation tool be provided the set of kernels to work via a user defined list.

## 2.4 Advantages

There are several advantages to this style of programming using array syntax, regions, and pure and elemental functions:

- There are no loops or index variables to keep track of. Off by one index errors and improper handling of array boundaries are a common programming mistake.
- The written code is closer to the algorithm, easier to understand, and is usually substantially shorter.
- Semantically the intrinsic function `region_cpy` returns an array by value. This is usually what the algorithm requires.

- Pure elemental functions are free from side effects, so it is easier for a compiler to schedule the work to be done in parallel.

Data parallelism has been called collection-oriented programming by Blelloch [2]. As the `cshift` function and the array-valued expressions all semantically return a value, this style of programming is also similar to functional programming (or value-oriented programming). It should be noted that the sub-setting language we employ goes beyond pure data parallelism by the use of pure (other than calls to `region_ptr`) subroutines and not just elemental functions.

Unfortunately, this style of programming has never really caught on because when array syntax was first introduced in Fortran, performance of codes using these features was relatively poor and thus programmers shied away from using array syntax (even recently, some are actively counseling against its usage because of performance issues [12]). Thus the Fortran community was caught in a classic "chicken-and-egg" conundrum: (1) programmers didn't use it because it was slow; and (2) compilers vendors didn't improve it because programmers didn't use it. A goal of this paper is to demonstrate that parallel programs written in this style of Fortran can achieve good performance on accelerator architectures.

## 2.5 Restrictions

Only pure Fortran procedures are transformed into OpenCL kernels. This restriction is lifted slightly to allow calls to `region_ptr` from within a kernel procedure. The programmer must explicitly call these kernels using Fortran interfaces in the ForOpenCL library (described below). It is also possible, using ROSE, to modify the calling site so that the entire program can be transformed, but this functionality is outside the scope of this paper. Here we specifically examine transforming Fortran procedures to OpenCL kernels. Because OpenCL support is relatively new to ROSE, some generated code must be modified. For example, the `__global` attribute for kernel arguments was added by hand.

It is assumed that memory for all arrays reside on the device. The programmer must copy memory to and from the device. In addition, array size (neglecting ghost cell regions) must be multiples of the global OpenCL kernel size.

Array variables within a kernel procedure (specified by the `$OFP PURE, KERNEL` directive) must be declared as contiguous. A kernel procedure may not call other procedures except for limited intrinsic functions (primarily math), user-defined elemental functions, and the `region_cpy` and `region_ptr` functions. Future work will address non-contiguous arrays (such as those that result from strided access) by mapping array strides to gather/scatter-style memory accessors.

Array parameters to a kernel procedure must be declared as either intent(in) or intent(out); they cannot be intent(inout). A thread may read from an extended region about its local element (using the `region_cpy` function), but can only write to the single array element it owns. If a variable were intent(inout), a thread could update its array element before another thread had read from that element. This restriction requires double buffering techniques.

# 3. SHALLOW WATER MODEL

The numerical code used for this work is from a presentation at the NM Supercomputing Challenge [13]. The algorithm solves the standard 2D shallow water equations. This algorithm is typical of a wide range of modeling equations based on conservation laws such as compressible fluid dynamics (CFD), elastic material waves, acoustics, electromagnetic waves and even traffic flow [11]. For the shallow water problem there are three equations with one based on conservation of mass and the other two on conservation of momentum.

$$\begin{aligned} h_t + (hu)_x + (hv)_y &= 0 \quad \text{(mass)} \\ (hu)_t + (hu^2 + \tfrac{1}{2}gh^2)_x + (huv)_y &= 0 \quad (x\text{-momentum}) \\ (hv)_t + (huv)_x + (hv^2 + \tfrac{1}{2}gh^2)_y &= 0 \quad (y\text{-momentum}) \end{aligned}$$

where h = height of water column (mass), $u$ = x velocity, $v$ = y velocity, and $g$ = gravity. The height $h$ can be used for mass because of the simplification of a unit cell size and a uniform water density. Another simplifying assumption is that the water depth is small in comparison to length and width and so velocities in the z-direction can be ignored. A fixed time step is used for simplicity though it must be less than $dt \leqq dx/(\sqrt{gh} + |u|)$ to fulfill the CFL condition.

The numerical method is a two-step Lax-Wendroff scheme. The method has some numerical oscillations with sharp gradients but is adequate for simulating smooth shallow-water flows. In the following explanation, $U$ is the conserved state variable at the center of the cell. This state variable, $U$ = $(h, hu, hv)$ in the first term in the equations below. $F$ is the flux quantity that crosses the boundary of the cell and is subtracted from one cell and added to the other. The remaining terms after the first term are the flux terms in the equations above with one term for the flux in the x-direction and the next term for the flux in the y-direction. The first step estimates the values a half-step advanced in time and space on each face, using loops on the faces.

$$\begin{aligned} U^{n+\frac{1}{2}}_{i+\frac{1}{2},j} &= (U^n_{i+1,j} + U^n_{i,j})/2 + \frac{\triangle t}{2\triangle x}\left(F^n_{i+1,j} - F^n_{i,j}\right) \\ U^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} &= (U^n_{i,j+1} + U^n_{i,j})/2 + \frac{\triangle t}{2\triangle y}\left(F^n_{i,j+1} - F^n_{i,j}\right) \end{aligned}$$

The second step uses the estimated values from step 1 to compute the values at the next time step in a dimensionally unsplit loop.

$$U^{n+1}_{i,j} = U^n_{i,j} - \frac{\triangle t}{\triangle x}(F^{n+\frac{1}{2}}_{i+\frac{1}{2},j} - F^{n+\frac{1}{2}}_{i-\frac{1}{2},j}) - \frac{\triangle t}{\triangle y}(F^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} - F^{n+\frac{1}{2}}_{i,j-\frac{1}{2}})$$

## 3.1 Fortran implementation

Selected portions of the data-parallel implementation of the shallow water model are now shown. This code serves as input to the ForOpenCL transformations described in the next section. The interface for the Fortran kernel procedure `wave_advance` is declared as:

```
subroutine wave_advance(dx,dy,dt,H,U,V,oH,oU,oV)
  !$OFP PURE, KERNEL   :: wave_advance
  real, intent(in)     :: dx,dy,dt
  real, dimension(:,:) :: H,U,V,oH,oU,oV
  contiguous  :: H,U,V,oH,oU,oV
  intent(in)  :: H,U,V
  intent(out) :: oH,oU,oV
  target      :: oH,oU,oV
end subroutine
```

where `dx, dy, dt` are differential quantities in space $x, y$ and time $t$, `H, U`, and `V` are state variables for the height and $x$ and $y$ momentum respectively, and `oH, oU, oV` are corresponding output arrays used in the double buffering scheme. The *OFP* compiler-directive attributes `PURE` and `KERNEL` indicate that the procedure `wave_advance` is to be transformed as an OpenCL kernel and that it must be pure, other than for any pointers used to reference interior regions of the output arrays.

Temporary arrays are required for the quantities `Hx, Hy, Ux, Vx`, and `Vy`, that are defined on cell faces. Also, the pointer variables, `pH, pU`, and `pV`, are needed to access and update interior regions of the output arrays. As these pointers are assigned to the arrays `oH, oU, oV`, these output arrays must have the target attribute, as shown in the interface above. The temporary arrays and array pointers are declared as,

```
real, allocatable, dimension(:,:) :: Hx, Hy, Ux
real, allocatable, dimension(:,:) :: Uy, Vx, Vy
real, pointer,     dimension(:,:) :: pH, pU, pV
```

Halo variables for the interior and the cell faces are declared and defined as

```
integer, dimension(4) :: face_lt, face_rt, halo
integer, dimension(4) :: face_up, face_dn

halo    = [1,1,1,1]
face_lt = [0,1,1,1];  face_rt = [1,0,1,1]
face_dn = [1,1,0,1];  face_up = [1,1,1,0]
```

Note that the halo definitions for the four faces each have a 0 in the initialization. Thus the returned array copy will have a size that is larger than any interior region that uses the full halo `[1,1,1,1]`. This is because there is one more cell face quantity than there are cells in a given direction.

The first Lax-Wendroff step updates state variables on the cell faces. Assignment statements like the following,

```
Hx = 0.5*( region_cpy(H,face_lt) +   &
           region_cpy(H,face_rt) )   &
     + (0.5*dt/dx)                   &
     * (region_cpy(U,face_lt) - region_cpy(U,face_rt))
```

are used to calculate these quantities. This equation updates the array for the height in the $x$-direction. The second step then uses these face quantities to update the interior region, for example,

```
face_lt = [0,1,0,0];  face_rt = [1,0,0,0]
face_dn = [0,0,0,1];  face_up = [0,0,1,0]

pH = region_ptr(oH, halo)

pH = region_cpy(H, halo)
       + (dt/dx) * ( region_cpy(Ux, face_lt) -   &
                     region_cpy(Ux, face_rt) )   &
       + (dt/dy) * ( region_cpy(Vy, face_dn) -   &
                     region_cpy(Vy, face_up) )
```

Note that face halos have been redefined so that the array copy returned has the same size as the interior region.

These simple code segments show how the shallow water model is implemented in standard Fortran using the data-parallel programming model described above. The resulting code is simple, concise, and easy to understand. However it does *not* necessarily perform well when compiled for a traditional sequential system because of suboptimal use of temporary array variables, especially those produced by the function `region_cpy`. This is generally true of algorithms that use Fortran shift functions as well, as some Fortran compilers (e.g., gfortran) do not generate optimal code for shifts. We note (as shown below) that these temporary array copies are replaced by scalars in the transformed Fortran code so there are no performance penalties for using data-parallel statements as outlined. However, there is an increased memory cost due to the double buffering required by the kernel execution semantics.

# 4. SOURCE-TO-SOURCE TRANSFORMATIONS

This section provides an brief overview of the ForOpenCL transformations that take Fortran elemental and pure procedures as input and generate OpenCL code. Elemental functions are transformed to inline OpenCL functions and subroutines with the `PURE` and `KERNEL` compiler directive attributes are transformed to OpenCL kernels.

## 4.1 OpenCL

OpenCL [9] is an open-language standard for developing applications targeted for GPUs, as well as for multi-threaded applications targeted for multi-core CPUs. The kernels are run by calling a C runtime library from the OpenCL host (normally the CPU). Efforts to standardize a C++ runtime are underway and Fortran interfaces to the C runtime are distributed in the ForOpenCL library.

An important concept in OpenCL is that of a thread and a thread group. Thread groups are used to run an OpenCL kernel concurrently on several processor elements on the OpenCL device (often a GPU). Consider a data-parallel statement written in terms of an elemental function as discussed above. The act of running an OpenCL kernel can be thought of as having a particular thread assigned to each instance of the call to the elemental function as it is mapped across the arrays in the data-parallel statement. In practice, these threads are packaged into thread groups when they are run on the device hardware.

Device memory is separated hierarchically. A thread instance has access to its own thread memory (normally a

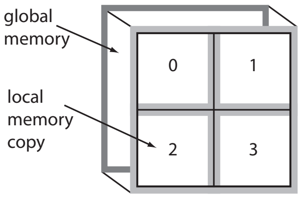

**Figure 1: A schematic of global memory for an array and its copy stored in local memory for four thread groups.**

set of registers), threads in a thread group to OpenCL local memory, and all thread groups have access to OpenCL global memory. When multiple members of a thread group access the same memory elements (for example in the use of the `region_cpy` function in the calculation of face variable quantities shown above), for performance reasons it is often best that global memory accessed and shared by a thread group be copied into local memory.

The *region* and *halo* constructs easily map onto the OpenCL memory hierarchy. A schematic of this mapping is shown in Figure 1 for a two-dimensional array with a 2x2 array of 4 thread groups. The memory for the array and its halo are stored in global memory on the device as shown in the background layer of the figure. The array copy in local memory is shown in the foreground divided into 4 *local* tiles that partition the array. Halo regions in global memory are shown in dark gray and halo regions in local memory are shown in light gray.

We point out that the hierarchical distribution of memory used on the OpenCL device shown in Figure 1 is similar to the distribution of memory across MPI nodes in an MPI application. In the case of MPI, the virtual global array is represented by the background layer (with its halo) and partitions of the global array are stored in the 4 MPI nodes shown in the foreground. Our current and future work on this effort includes source-to-source transformations to generate MPI code in addition to OpenCL in order to deal with clusters of nodes containing accelerators. This work is outside the scope of this paper.

Halo regions obtained via the `region_cpy` function (used with intent(in) arrays) are constrained semantically so that they can not be written to by an OpenCL kernel. The `region_cpy` function returns a copy of the region of the array stored in global memory and places it in local memory shared by threads in a thread group. Thus once memory for an array has been transferred into global device memory by the host (before the OpenCL kernel is run), memory is in a consistent state so that all kernel threads are free to read from global device memory. Because the local memory is a copy, it functions as a software cache for the local thread group. Thus the compiler must insert OpenCL barriers at proper locations in the code to insure that all threads have written to the local memory cache before any thread can start to read from the cache. On exit from a kernel, any local memory explicitly stored in register variables by the compiler (memory accessed via the `region_cpy` function) is copied back to global memory for all intent(out) arrays. Recall that a thread may only write to its own intent(out) array element, thus there are no race conditions when updating intent(out) arrays.

## 4.2 Transformation examples

This section outlines the OpenCL equivalent syntax for portions of the Fortran shallow-water code described in Section 3. The notation uses uppercase for arrays and lowercase for scalar quantities. Variables temporarily storing quantities for updated output arrays (declared as pointers in Fortran) are denoted by a `p` preceding the array name. For example, the Fortran statement `pH = region_ptr(oH, halo)` is transformed as a scalar variable declaration representing a single element in the output array `oH`.

### *4.2.1 Region function*

While the Fortran version of the `region_cpy` function semantically returns an array copy, in OpenCL this function returns a scalar quantity based on the location of a thread in a thread group and the relationship of its location to the array copy transferred to local memory. Because we assume there is a thread for every element in the interior, the array index is just the thread index adjusted for the size of the halo. Thus `region_cpy` is just an inline OpenCL function and is provided by the ForOpenCL library.

### *4.2.2 Function and variable declarations*

Fortran kernel procedures have direct correspondence with OpenCL equivalents. For example, the `wave_advance` interface declaration is transformed as

```
__kernel void
wave_advance(float dx, ..., __global float * H, ...);
```

The intent(in) arrays have local equivalents that are stored in local memory and are declared by, for example,

```
__local float H_local[LOCAL_SIZE];
```

These local arrays are declared with the appropriate size and are copied to local memory by the compiler with an inlined library function. The array temporaries defined on cell faces are declared similarly while interior pointer variables are simple scalars, e.g., `float pH`. Intent(in) array variables cannot be scalar objects because regions may be shifted and thus *shared* by threads within a thread group.

### *4.2.3 Array syntax*

Array syntax transforms nearly directly to OpenCL code. For example, interior pointer variables are particularly straightforward as they are scalar quantities in OpenCL,

```
pH = region_cpy(H, halo)
       + (dt/dx) * ( region_cpy(Ux, face_lt) -
                     region_cpy(Ux, face_rt) )
       + (dt/dy) * ( region_cpy(Vy, face_dn) -
                     region_cpy(Vy, face_up) );
```

| Array width | F90 | GPU (16x8) | Speedup |
|---|---|---|---|
| 16 | 0.025 ms | 0.017 ms | 1.5 |
| 32 | 0.086 | 0.02 | 4.3 |
| 64 | 0.20 | 0.02 | 10.0 |
| 128 | 0.76 | 0.036 | 21.1 |
| 256 | 3.02 | 0.092 | 32.8 |
| 512 | 12.1 | 0.32 | 37.8 |
| 1024 | 49.5 | 1.22 | 40.6 |
| 1280 | 77.7 | 1.89 | 41.1 |
| 2048 | 199.1 | 4.82 | 41.3 |
| 4096 | 794.7 | 19.29 | 41.2 |

**Table 1: Performance measurements for the shallow-water code. All times reported in milliseconds.**

Allocated variables are more complicated because they are arrays.

```
Hx[i] = 0.5 * (region(H_local, face_lt)+ ...);
```

where `i = LX + LY*(NLX+halo(0)+halo(1)))` is a local index variable, `LX = get_local_id(0)` is the local thread id in the $x$ dimension, `LY = get_local_id(1)` is the local thread id in the $y$ dimension, `NLX = get_local_size(0)` is the size of the thread group in the $x$ dimension, and the `get_local_id` and `get_local_size` functions are defined by the OpenCL language standard.

## 5. PERFORMANCE MEASUREMENTS

Performance measurements were made comparing the transformed code with different versions of the serial shallow-water code. The serial versions included two separate Fortran versions: one using data-parallel notation and the other using explicit looping constructs. We also compared with a hand-written OpenCL implementation that was optimized for local memory usage (no array temporaries). The accelerated measurements were made using an NVIDIA Tesla C2050 (Fermi) cGPU with 2.625 GB GDDR5 memory, and 448 processor cores. The serial measurements were made using an Intel Xeon X5650 hexacore CPU with 96 GB of RAM running at 2.67 GHz. The compilers were gfortran and gcc version 4.4.3 with an optimization level of -O3.

Several timing measurements were made by varying the size of the array state variables. The performance measurements are shown in Table 1. An average time was obtained by executing 100 iterations of the outer time-advance loop that called the OpenCL kernel. This tight loop kept the OpenCL kernel supplied with threads to take advantage of potential latency hiding by the NVIDIA GPU. Any serial code within this loop (not present in this study) would have reduced the measured values.

The transformed code achieved very good results. In all instances, the performance of the transformed code was within 25% of the hand-optimized OpenCL kernel. Most of the extra performance of the hand-optimized code can be attributed to the absence of array temporaries and to packing the three state variables `H, U,` and `V` into a single vector datatype.

While we did not have an OpenMP code for multi-core comparisons, the transformed OpenCL code on the NVIDIA C2050 was up to 40 times faster than the best serial Fortran code executing on the host CPU.

## 6. CONCLUSIONS

The sheer complexity of programming for clusters of many or multi-core processors with tens of millions threads of execution makes the simplicity of the data-parallel model attractive. The increasing complexity of today's applications (especially in light of the increasing complexity of the hardware) and the need for portability across architectures make a higher-level and simpler programming model like data-parallel attractive.

The goal of this work has been to exploit source-to-source transformations that allow programmers to develop and maintain programs at a high-level of abstraction, without coding to a specific hardware architecture. Furthermore these transformations allow multiple hardware architectures to be targeted without changing the high-level source. It also removes the necessity for application programmers to understand details of the accelerator architecture or to know OpenCL.

## 7. ACKNOWLEDGMENTS

This work was supported in part by the Department of Energy Office of Science, Advanced Scientific Computing Research.

## 8. REFERENCES

[1] S. Benkner, G. Lonsdale, and H. Zima. The HPF+ Project: Supporting HPF for Advanced Industrial Applications. In *Proceedings of Euro-Par'99*, 1999.

[2] G. Blelloch and G. W. Sabot. Compiling collection oriented languages onto massively parallel processors. *Journal of Parallel and Distributed Computing*, 8(2), Feb 1990.

[3] R. G. Brickner, K. Holian, B. Thiagarajan, and S. L. Johnsson. Designing a stencil compiler for the connection machine model CM-5. Technical Report LA-UR-94-3152, LANL, 1994.

[4] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[5] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and L. Snyder. The High-Level Parallel Language ZPL Improves Productivity and Performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[6] A. D. Falkoff and K. E. Iverson. APL language summary. *SIGPLAN Notices*, 14(4), 1979.

[7] M. Feldman. Intel touts manycore coprocessor at supercomputing conference. *HPC-Wire*, 2011.

[8] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of ICFP 2010*, 2010.

[9] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[10] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[11] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics, 2002.

[12] J. Levesque. Fifty Years of Fortran. Panel Discussion, SuperComputing 2008, Reno, Nevada.

[13] R. W. Robey, R. Roberts, and C. Moler. Secrets of supercomputing: The conservation laws, supercomputing challenge kickoff. Research Report LA-UR-07-6793, LANL, 2007.

[14] The Portland Group. PGI Fortran & C Accelerator Programming Model. Technical report, March 2010.

[15] M. Wolfe. How We Should Program GPGPUs. *Linux Journal*, 2008.