LA-UR- 10-08340

Approved for public release;
distribution is unlimited.

Title:	Exploiting First-Class Arrays in Fortran for Accelerator Programming
Author(s):	Matthew J. Sottile Craig E. Rasmussen Wayne N. Weseloh Robert W. Robey Daniel Quinlan Jeffery Overbye
Intended for:	HIPS workshop at IPDPS 2011 (25th IEEE International Parallel & Distributed Processing Symposium)

Form 836 (7/06)

Exploiting First-Class Arrays in Fortran for Accelerator Programming

Matthew J. Sottile
Galois, Inc.
Email: matt@galois.com

Craig E Rasmussen,
Wayne N. Weseloh,
Robert W. Robey
Los Alamos National Laboratory
Email: rasmussn@lanl.gov

Daniel Quinlan
Lawrence Livermore National Laboratory

Jeffrey Overbey
Indiana University

***Abstract*—Emerging architectures for high performance computing often are well suited to a data parallel programming model. This paper presents a simple programming methodology based on existing languages and compiler tools that allows programmers to take advantage of these systems. We will work with the array features of Fortran 90 to show how this infrequently exploited, standardized language feature is easily transformed to lower level accelerator code. Our transformations are based on a mapping from Fortran 90 to C++ code with OpenCL extensions.**

I. Introduction

This paper presents a compiler-level approach for targeting a single program to multiple fundamentally different low level execution and programming models while allowing the application programmer to adopt a single high level programming model. We show that features of Fortran 90 for data parallel programming are well suited to automatic transformation to generate code specifically tuned for different low-level programming models such as OpenCL and CUDA. 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 from the programmer of working with tedious, error prone, low-level tools.

In the ideal situation, application programmers would like to adopt a programming model in which they write their application once and use automated tools to retarget it to many architectures. This has proven to be very challenging historically due to the subtle balance between high-level expressiveness of code and the performance of the lower-level code that is emitted by a compiler. This ideal high-level model that programmers work with should emphasize readability, maintainability, and close proximity in abstraction to the problem being solved. It should not be corrupted with details of specific target architectures solely for the purpose of single-system performance. For certain classes of applications, specifically those that map well onto a data-parallel programming model, we show that Fortran 90 contains language features that encourage high-level programming abstractions without sacrificing performance during low-level code generation.

At Los Alamos National Laboratory (LANL), as with many supercomputing facilities today, users have a wide variety of computer platforms from which to choose. The most common platform is made up of clusters of compute nodes with standard multi-core processors. An increasingly common feature is that some nodes also have accelerators that range from the IBM Cell processor (such as the LANL Roadrunner system) to a variety of GPUs from NVIDIA and AMD. Some nodes have hardware with vector instructions and others do not. The peak performance of these accelerated nodes often resides in the hundreds of gigaflops.

However, the performance that these accelerator architectures offer comes at a cost as processor architectures are trending toward multiple cores with instances of integrated accelerator units with user managed memory 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 are forced to adapt to the new systems. An especially challenging problem faced by application programmers 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 with unique memory systems and programming models. The fundamental question we address is what programming model and language constructs are best suited to span this set of new hardware designs.

A common theme amongst new processors is the emphasis on data parallel programming. This model is well suited to emerging architectures that are based on either vector processing or massively parallel collections of simple cores. A popular trend for programming systems such as GPU accelerators is the CUDA language from NVidia, or new languages like Chapel from Cray. A less intrusive approach that is well suited to legacy applications and languages are language extensions or libraries that allow programmers to avoid adopting entirely new languages. The recent OpenCL specification is intended to support this programming model, as are directive-based methods such as OpenMP or the Accelerator programming model from the Portland Group [1].

The problem with many of these choices is that they

expose too much detail about the machine architecture to the programmer. This is particularly true of CUDA and OpenCL. In CUDA, programmers must adapt their codes to fit the threading model used by NVidia GPUs, while OpenCL requires programmers to provide specially tuned versions of their code for different classes of machine. In both cases, the programmer is also responsible for explicitly managing memory, including staging of data back and forth from the host CPU and the accelerator device memory. While these models 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.

At this point in time it is not really possible to write once and run efficiently on the wide variety of computer platforms we have available. For some classes of applications, we believe that this goal is possible using language constructs already present in a popular mainstream scientific programming language – Fortran. A common, long-standing tongue-in-cheek response to new language developments in the scientific and high performance computing community is that a new language will arise to answer the needs of new systems, and it will be called Fortran. We believe that the work presented in this paper validates that notion – we need a new language to work with, and that language is Fortran.

A. Approach

This paper addresses the accelerator programmign problem by examining features in Fortran that allow programmers to express algorithms at a very high level that can be easily transformed by a compiler to run efficiently on a wide variety of platforms. In particular we consider computers based on GPUs and related accelerator processors.

We demonstrate that the array syntax of Fortran maps surprisingly well onto GPUs when transformed to OpenCL kernels. These Fortran language features include pure and elemental functions and array constructs like where and shift. In addition we add a few functions that enable a program to be deployed on machines with a hierarchy of processing elements, such as nodes employing GPU acceleration, *without requiring explicit declaration of parallelism within the program.* In addition the program uses entirely standard Fortran so it can be on a single core without concurrency. This work also is applicable to vendor-specific languages similar to OpenCL such as the NVidia CUDA language.

We provide (via Fortran interfaces in the ForOpenCL library) a mechanism to call the C OpenCL runtime and enable Fortran programmers to access OpenCL kernels. 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 [2] to develop these transformations. ROSE uses the Open Fortran Parser [3] to parse Fortran 2008 syntax and can output (unparse) to 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 OpenCL nodes. Furthermore, the Fortran array syntax maps directly to one, two, and three-dimensional thread groups in OpenCL.

Transformations for general Fortran procedures are not provided. 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.

We examine the performance of the Fortran data-parallel abstraction when transformed to OpenCL to run on GPU architectures. Since single node performance is often given as a reason for not using data-parallel constructs within Fortran, we consider the performance of serial data-parallel codes compared with the usage of explicit loop constructs.

We study automatic transformations and the performance for an application example that is typical of many applications that are based on finite-difference or finite-volume methods in computational fluid dynamics (CFD). The example is a simple shallow water model in two dimensions using finite volume methods with .... for time update.

An initial study was made for an important procedure in PAGOSA, a non-research, production-grade code at LANL completely written in data-parallel Fortran. We investigated automatically transforming this code to run on LANL's Petaflop Roadrunner computer (a hybrid mixture of AMD Opterons and IBM Cell processors). We demonstrated that a source-to-source compiler can automatically vectorize and parallelize (both within the Cell processor and across nodes) a small section of this code. Preliminary results showed a 12 times performance gain on the Cell processor over a traditional single core processor. A factor of 9 is gained from vectorizing and parallelizing the code to run on the Cell and the rest is gained from automatically overlapping inter-node communication with computation on the Cell.

B. Why Fortran?

Fortran is the oldest high-level programming language in continuous use since its introduction, and was developed to facilitate the translation of math formulae into machine code. Fortran was the first major language to use a compiler to translate from a high-level program representation to assembly language. Due to its age, it carries certain arcane baggage that later programming languages have evolved away from. As a result, Fortran has fallen into disfavor in certain programming circles. Modern versions of the language standardized in 1990, 1995, 2003, and 2008 have removed much of this legacy baggage, but these changes are not widely known. Modern Fortran exhibits features that are similar to other modern programming languages, and does

not mandate legacy features from decades old, deprecated versions of the language.

With the introduction of Fortran 90 (and later revisions to the standard), Fortran became a truly modern programming language. Much of the arcane baggage associated with it is actually no longer part of the language used by modern programmers, and exists in a deprecated form simply to support legacy codes. It is now modular and has many object-oriented features. It now includes a type system in which rich first-class array data types and corresponding syntax are part of the language, something that languages like C continue to lack. Furthermore, Fortran should be of interest to those studying parallel programming because of its functional and data parallel constructs and because of the coarray notation introduced in Fortran 2008. Unlike languages like C and C++, which are considered to be more modern, Fortran has become a truly parallel language with features added to recent language standards.

Yet, a likely question that one may pose is "*Why Fortran and not a more modern language like X?*" The recent rise in interest in concurrency and parallelism at the language level driven by multicore CPUs and manycore accelerators has driven a number of new language developments, both as novel languages and extensions on existing ones.

So perhaps it is time to replace Fortran with yet another computer language. The problems with replacing Fortran with an entirely new language are two fold: the economics of replacing the existing application base and the difficulty in obtaining programmer acceptance. It is estimated that replacing a major production application at Los Alamos National Laboratory would cost between 50 and 150 million dollars. In terms of programmer acceptance, there is always the "chicken and egg problem": programmers won't use a new language until they can expect good performance across a variety of platforms, and compiler vendors can't afford to produce quality compilers until there is a reasonable expectation of a market.

For scientific users, these new languages and language extensions present a challenge: how do developers effectively use them while avoiding rewriting their code and potentially growing dependent on a transient technology that will vanish tomorrow?

History has also shown that an investment in rewriting code does not guarantee success either, as seen in an effort at LANL to modernize a legacy Fortran code with the the newer C++ POOMA framework. This massive overhaul effort led to a code that was both slower and less flexible than the original Fortran [4]. Similar experiences have occurred in the past, notably during the devlopment of the Sisal language.

Fortran is unique in that it has contained language features that are well suited to modern architectures for a number of years. This should be unsurprising — Fortran was a primary language used to target systems such as the vector supercomputers and massively parallel systems of the 1970s and 1980s. These are the systems in which architectural features were developed that have led to single chip high performance architectures of interest today. Given that these new systems have features very similar to their predecessors, it is clear that the language features within Fortran for them are still relevant.

C. Comparison to Other Languages

A number of previous efforts have exploited data parallel programming at the language level to utilize novel architectuers, particularly in previous decades during the reign of vector and massively parallel computers in the high performance computing world. The origin of the array syntax that was adopted in Fortran 90 can be found in the APL language [5]. Fortran 90 differed from previous extensions of Fortran in that parallelism within whole-array operations was expressed at the expression level instead of via parallelism within explicit DO-loops (such as within IVTRAN for the Illiac IV).

The High Performance Fortran (HPF) extension of Fortran 90 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 [6]. 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 order of execution of the program (such as a sequence of loop iterations) can be flexible in order to allow parallel execution. 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 was likely in order to support parallel programming that wasn't easily mapped onto a pure data parallel model.

In some instances though, a purely data parallel model is appropriate 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 based on local offsets from singe indices. A simple 1D example is one in which an element is replaced with the average of its own value with 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)
  X(i) = (X(i-1) + X(i) + X(i+1)) / 3
end do
```

When shifts are employed, this can be expressed as:

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

Similar whole array shifting was used in higher dimensions for finite different codes within the computational physics community, especially at Los Alamos for codes targetting the CM-5 system that resided there until the late 1990s.

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. When the compiler is responsible for managing this intermediate storage, it has historically proven that they are inefficient and generate code that requires far more temporary storage than really necessary. This is not a flaw of the language constructs, but a sign of the lack of sophistication of the compilers with respect to their internal analysis to determine how to optimally generate this intermediate storage.

An interesting line of language research that grew out of the early work with HPF was that associated with the ZPL language work at the University of Washington ??. In ZPL, programmers adopt a simimlar global view of computation over arrays, but define their computations based on the local view of indices that participate in the update of each element of an array. For example, in the 1D averaging case stated above, a programmer would define the computation in terms of the update that occurs at each index in terms of relative offsets from that index:

II. Programming Model

The static analysis and source-to-source transformations used are very basic and only require the programmer to use a subset of Fortran that employs a data-parallel programming model. In particular, it encourages use of language features that were introduced and standardized in the Fortran 90 language specification. In this section we describe the set of four Fortran 90 features that our analysis and transformation method are based on. From these language constructs, we are able to easily transform them to a lower-level CUDA or OpenCL implementation.

Array notation: Fortran 90 introduced a rich array syntax that allows programmers to write statements that are in terms of whole arrays or subarrays, with data parallel operators to compute on the arrays. This has the benefit of avoiding explicit looping in the code and maintaining a high-level style that is closer to the original mathematical specification of the problem being solved. More importantly from a compilation perspective, this defers decisions about how to implement these whole-array operations to a compilation tool. Sophisticated loop analysis to identify parallelism within the loop are not necessary. When faced with a novel architecture such as modern GPUs that are ideally suited to data parallel programming models, the fit between Fortran arrays and these systems is quite clean.

Arrays are first class citizens in Fortran instead of a basic pointer to raw memory as in many other compiled languages. A Fortran array contains metadata associated with it that describes the array's shape and size. It should be noted that other languages that provide rich array data types and whole array operations may also be suitable targets for the transformations described in this paper. In addition to metadata about the structure of the arrays, operators are provided that allow programmers to write expressions based on the whole array instead of explicit indexing and application of operations to individual elements. 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 `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.

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. Elemental functions are intended to be used for data parallel programming, and as a result must be side

effect free (or, *pure*).

For example, the basic array operation shown above could be refactored into an 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 elemental functions to inline OpenCL functions. Thus there is no penalty for usage of elemental functions and provide a convient mechanism to express algorithms in simpler segments.

Pure procedures: Pure procedures, like elemental functions, must be free of side effects. Unlike 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 a candidates for running on accelerators in OpenCL. Currently our prototype tool transforms pure procedures to OpenCL kernels that *do not* call other procedures, except for elemental functions.

Shift functions: Many array-based algorithms require the same operation to be performed on each element of the array using the value of that element and some small set of neighboring cells. Often programmers implement these operations that are local to each element within the inner-loop of a set of nested FOR- or DO-loops using offsets relative to the current array index. An alternative to this local-view of the algorithm is to take a global view and write the algorithm in terms of the whole array. For example, consider a 1D array in which we wish to subtract the $(i-1)$th element from the ith for all elements. One way to look at this is that we are subtracting the entire array shifted by one element from itself.

Fortran provides a set of shifting operators that allow programmers to define operations based on shifted arrays. These intrinsic operators take an array, a dimension, and the amount by which it should be shifted (using the sign to indicate direction). By defining operations on entire arrays based on a global view of them shifted relative to each other, programmers can avoid explicit looping and potentially tricky index arithmetic. Furthermore, analysis of the extent of the set of shifted arrays in a given expression allows analysis tools to determine the amount of temporary or buffer storage necessary to hold intermediate values during whole array operations. With explicit loops, programmers must maintain this temporary storage manually.

Regions: Borrowing from ZPL, we introduce the concept of regions to Fortran. In Fortran, regions are expressed as pointers to a subsection of an existing array. Regions may refer to an interior portion of an array or the entire array.

The use of regions implies the use of the halo (or ghost) cell pattern where the size of an array is increased to provide extra array elements surrounding the interior portion of the array.

Regions 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. 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.

A. New Procedures

Unlike OpenCL, we require limited use of compiler directives, although two are used to enforce the semantics required by the programming model. These directives are:

- `$OFP CONTIGUOUS`: specifies that an dummy array variable is contiguous in memory. CONTIGUOUS is an attribute in Fortran 2008.
- `$OFP KERNEL`: specifies that a pure subroutine can be transformed to an OpenCL kernel.

Three new procedures (in addition to the intrinsic shift function) are defined in Fortran that are used in array-syntax operations. Each procedure takes a 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:

- `transfer_halo(array, halo)`: an impure subroutine that exchanges halo regions between nodes using MPI or Fortran coarrays. Array is the memory stored on the node representing the node-local portion of a virtual global array. Not used in this work.
- `interior(array, halo)`: a pure function that returns a copy of the interior portion of the array specified by halo. Array is local to a node.
- `region(array, halo)`: a pure function that returns a copy of the portion of the array specified by halo. Array is local to a node.

It should be noted that the two functions `interior` and `region` are pure and thus can be called from within a

pure kernel procedure. These two functions are part of the language recognized by the compiler and though the two functions return 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.

B. Parallelism

There are several advantages to this style of programming using array syntax, shifts, 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 functions return arrays by value. This is usually what the algorithm requires.
- Because pure and elemental function are free from side effects, it is easier for a compiler to schedule the work to be done in parallel.

An example of this style of programming in Fortran is shown in

```
Bz = Bz &
   + dt*(cshift(Ex,dim=2,shift=+1)-Ex)/dy &
   - dt*(cshift(Ey,dim=1,shift=+1)-Ey)/dx
```

This example is a solution to Maxwell's equations for the z component of the magnetic field using Fortran array syntax. Note that there are no explicit loops in this example. The operators +, −, and $*$ are applied to all of the elements of the three-dimensional arrays `Bz`, `Ex`, and `Ey`, individually. This is why data parallelism has been called collection-oriented programming by Blelloch [7], [8]. As the `cshift` function and the array-valued expressions all semantically returns a value, this style of programming is also similar to functional programming (or value-oriented programming).

Complete and very concise and elegant programs can be built with procedures similar to the example shown above. To aid this effort, Fortran supplies intrinsic functions like the array constructors (`CSHIFT`, `EOSHIFT`, `MERGE`, `TRANSPOSE`, ...), the array location functions (`MAXLOC` and `MINLOC`), and the array reduction functions (`ANY`, `COUNT`, `MINVAL`, `SUM`, `PRODUCT`, ...). To this set we add region functions described above.

This style of programming meets the requirements we have set for a programming model for developing applications suitable for acceleration. It allows the programmer to program at a very-high level of abstraction while providing the compiler with maximum flexibility in targeting the application for a particular hardware architecture. The data parallel programming model simultaneously meets the seemingly conflicting goals of maintainability, portability, and performance.

Unfortunately, this style of programming has never really caught on because when Fortran 90 was first introduced, performance was relatively poor and thus programmers shied away from using array syntax (even now, some are actively counseling against its usage because of performance issues [9]). 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 are maintainable and can achieve good performance on accelerator architectures.

C. Limitations

Only Fortran procedures are transformed into OpenCL kernels. The programmer must currently explicitly call these kernels from Fortran using 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.

Only elemental functions may be called from kernel functions. These include fortran functions that have an OpenCL analog and user-defined elemental functions.

A kernel procedure (specified by the `$OFP KERNEL` directive) must be pure and all array variables must be declared as contiguous. A kernel procedure may not call other procedures except for limited intrisic functions (primarily math), user-defined elemental function, and the `interior` and `region` functions.

Array sizes must be multiples of the local kernel size, `get_local_size(0)*get_local_size(1)`. This may be relaxed in the future.

III. Shallow Water Model

The numerical code used for this study is from a presentation at the NM Supercomputing Challenge [10]. 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 $dt \leqq (\sqrt{gh} + |u|)/dx$.

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 above. 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.

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

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

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

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

A. *Serial code examples*

The Fortran kernel procedure `wave_advance` for the shallow water code is declared as:

```
pure subroutine wave_advance(H,U,V,dx,dy,dt)
  real, dimension(:,:),intent(inout):: H,U,V
  real, intent(in) :: dx,dy,dt
  !$OFP CONTIGUOUS :: H,U,V
  !$OFP KERNEL :: wave_advance
end subroutine
```

where `H`, `U`, and `V` are start variables for height and x and y momentum respectively. The *OFP* compiler directives `CONTIGUOUS` and `KERNEL` are added because contiguous is Fortran 2008 syntax (not in current compilers) and kernel is an *OFP* extension.

Temporary arrays are required for the interior copies `iH`, `iU`, and `iV` of the state variables and for the flux quantities `Hx`, `Hy`, `Ux`, `Vx`, and `Vy` defined on cell faces. These temporary arrays are declared as,

```
real, allocatable, dimension(:,:) :: iH,iU,iV
real, allocatable, dimension(:,:) :: Hx,Hy,Ux
real, allocatable, dimension(:,:) :: Uy,Vx,Vy
```

Halos variables for the interior and cell faces are defined as

```
integer, dimension(4) :: halo,face_lt,face_rt
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 halos for the four faces each have a 0 in the definition. Thus the returned array copy will have size that is larger than the interior regions that use `[1,1,1,1]`. This is because there is one more cell face quantity than there are cells in a given direction.

The Lax-Wendroff update algorithm for the shallow-water model has two steps. The fluxes are first updated on cell faces, for example,

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

updates the x-direction flux for the height state variable.

Then second step uses these fluxes to update the interior regions for the state variables. 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]
iH = iH + (dt/dx) * ( region(Ux, face_lt) - &
                      region(Ux, face_rt) ) &
        + (dt/dy) * ( region(Vy, face_dn) - &
                      region(Vy, face_up) )
```

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

These simple code segments show how the shallow water model is implemented in standard Fortran (2003) using the data-parallel programming model described above. The resulting code is simple, consise, and easy to understand. However it does *not* necessarily perform well because of the temporary array variables, especially those produced by the `region` function. 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 seen below) that these temporary array copies are replaced by scalars in the transformed Fortran code so there is no performance penalty for using data-parallel statements as outlined.

IV. Source-To-Source Transformations

This section describes the transformations that take Fortran elemental and pure procedures as input and generate OpenCL kernels.

A. *OpenCL*

OpenCL [12] is an open language standard for developing applications for accelerators. The C-based language provides extensions for programming kernels that run on accelerator processing elements. The kernels are run by calling a C runtime library from the OpenCL host (normally the CPU).

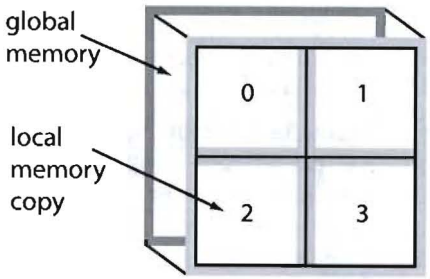

Figure 1. A schematic of global memory for an array and its copy stored in local memory for four thread groups.

Efforts to standardize a C++ runtime are underway and Fortran interfaces to the C runtime are described later.

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 fuction 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 memory, thread groups 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 if `region` or `shift` functions are called, for performance reasons it is often best if global memory accessed by a thread group is 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 emphasize that the hierarchical distribution of memory used on the OpenCL device shown in Figure 1 can be extended to include memory across MPI nodes as well. In this case, the virtual global array is represented by the background layer (with its halo) and its partitions stored in the 4 MPI nodes shown in the foreground.

Halo regions are constrained semantically so that they can not be written to by an OpenCL kernel because the *region* and *interior* functions return copies of the global memory. Thus once memory for a halo region has been transferred into global device memory by all of the host nodes running MPI (before the OpenCL kernel is run), memory is in a consistent state so that the kernels 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 read from the cache. On exit from a kernel, the local memory cache is copied back to global memory for all *interior* regions leaving global memory in a consistent state again.

B. ForOpenCL

This work describes transformations that automatically create OpenCL kernel functions from Fortran pure and elemental procedures. At present, these transformations will not transform an entire program. Users, for now, must explicitly replace calls to Fortran kernel procedures that run on the device, with calls to the OpenCL runtime that will run the kernel on the OpenCL host. While these host transformations are straightforward using ROSE, they are outside the scope of this paper.

In addition to the Fortran to OpenCL transformations, the ForOpenCL [3] library provides programmers with the ability to call the C OpenCL runtime from Fortran. ForOpenCL is a set of Fortran modules providing Fortran 2003 interface descriptions and classes that allow language interoperability with the OpenCL runtime.

V. Transformation Examples

This sections outlines the OpenCL equivalent syntax for portions of the Fortran shallow-water code described in the previous section. The notation uses uppercase for arrays and lowercase for scalar quantities. Interior and region array copies are denoted by an i or an r preceeding the array. For example, `iH = interior(H, halo)` is a local copy of interior region of array `H` (representing height) in the shallow water code.

1) interior and region functions: While the serial versions of the interior and regions functions return an array copy, in OpenCL, these functions return a scalar quantity based on the location of a thread in a thread group and the relationship of its location to the array copy in 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 halo. Thus `interior` and `region` are just inline OpenCL functions provided by the ForOpenCL library.

2) function and variable declarations: Fortran kernel procedures have direct correspondence with OpenCL equivalents. For example, `wave_advance` is transformed as `__kernel void wave_advance(__ global float * H, ..., float dt);`. The global state variables have local equivalents, e.g., `__local float`

H_local[MAX_LOCAL_SIZE], declared with the appropriate size and are copied to local memory by the compiler with inlined library functions. The flux array temporaries (region variables) are declared similarly but initialized to zero with inlined library functions. Interior variables are simple scalars, e.g., `float iH;`. Region variables cannot be scalar objects because regions are shifted and thus *shared* by threads within a thread group.

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

```
iH = iH + (dt/dx) * (region(Ux, face_lt) -
                     region(Ux, face_rt) )
        + (dt/dy) * (region(Vy, face_dn) -
                     region(Vy, face_up) );
```

Region variables are more complicated because they are arrays.

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

where `l = 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 compiler.

VI. Performance Measurements

Performance measurements were made comparing the transformed code with different versions of the serial shallow-water code. The serial versions include the original written in C and two Fortran versions: one using copy semantics for the regions and one using reference semantics. We also compare with a hand written OpenCL implementation that was optimized for local memory usage (no array temporaries). The accelerated measurments 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.

The performance comparison is shown in Figure 2 for varying array sizes (of the state variables). The time represents an average of 100 iterations of the outer time-advance loop calling the OpenCL kernel. This tight loop keeps the OpenCL kernel supplied with threads to take advantage of potential latency hiding by the NVIDIA GPU. Any serial code within this loop would reduce the measured values.

As can be seen in Figure 2, the transformed get very good results of roughly 65 times speedup over the best Fortran code and YYY speedup over the original C code. It comes with XXX % of the hand coded OpenCL version.

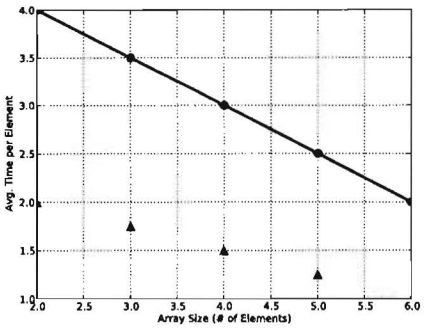

Figure 2. Performance comparison for varying array size.

VII. Conclusions

The sheer complexity of programming for clusters of many or multi-core processors with tens of millions threads of execution make the simplicity of the data parallel model attractive. Furthermore, the increasing complexity of todays applications (especially when convolved with the increasing complexity of the hardware) and the need for portability across hardware 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.

References

[1] The Portland Group, "PGI Fortran & C Accelerator Programming Model," Tech. Rep., March 2010. [Online]. Available: http://www.pgroup.com/resources/articles.htm

[2] D. Quinlan, "ROSE Compiler Infrastructure." [Online]. Available: http://rosecompiler.org/

[3] C. Rasmussen and M. Sottile, "Open Fortran Parser." [Online]. Available: http://fortran-parser.sourceforge.net/

[4] V. R. Basili, D. Cruzes, J. C. Carver, L. M. Hochstein, J. K. Hollingsworth, M. V. Zelkowitz, and F. Shull, "Understanding the High-Performance Computing Community: A Software Engineer's Perspective," *IEEE Software*, July/August 2008.

[5] A. D. Falkoff and K. E. Iverson, "APL language summary," *SIGPLAN Notices*, vol. 14, no. 4, 1979.

[6] 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.

[7] G. Blelloch and G. W. Sabot, "Compiling collection oriented languages onto massively parallel processors," *Journal of Parallel and Distributed Computing*, vol. 8, no. 2, February 1990.

[8] S. Rajopadhye, "LACS: a language for affine communication structures," INRIA, Research Report RR-2093, 1993. [Online]. Available: http://hal.inria.fr/inria-00074579/en/

[9] J. Levesque, "Fifty Years of Fortran," Panel Discussion, SuperComputing 2008, Reno, Nevada, 15 November 2008.

[10] R. W. Robey, R. Roberts, and C. Moler, "Secrets of supercomputing: The conservation laws, supercomputing challenge kickoff," LANL, Research Report LA-UR-07-6793, 21-23 October 2007.

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

[12] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf