# Performance Portability with the Chapel Language

Albert Sidelnik*, Saeed Maleki*, Bradford L. Chamberlain†, María J. Garzarán*, David Padua*

*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
†Cray Inc., Seattle, Washington, USA
Email: {asideln2,maleki1,garzaran,padua}@illinois.edu*, bradc@cray.com†

***Abstract*—It has been widely shown that high-throughput computing architectures such as GPUs offer large performance gains compared with their traditional low-latency counterparts for many applications. The downside to these architectures is that the current programming models present numerous challenges to the programmer: lower-level languages, loss of portability across different architectures, explicit data movement, and challenges in performance optimization.**

**This paper presents novel methods and compiler transformations that increase programmer productivity by enabling users of the language Chapel to provide a single code implementation that the compiler can then use to target not only conventional multiprocessors, but also high-throughput and hybrid machines. Rather than resorting to different parallel libraries or annotations for a given parallel platform, this work leverages a language that has been designed from first principles to address the challenge of programming for parallelism and locality. This also has the advantage of providing portability across different parallel architectures. Finally, this work presents experimental results from the Parboil benchmark suite which demonstrate that codes written in Chapel achieve performance comparable to the original versions implemented in CUDA on both GPUs and multicore platforms.**

## I. Introduction

In the last few years, systems of heterogeneous components, including GPU accelerator architectures, have become increasingly popular. This popularity has been driven by many emerging applications in client and HPC markets. Significant cost, power, and performance benefits are derived from executing these applications on systems containing both SIMD and conventional MIMD devices. For this reason, it is becoming increasingly common to find heterogeneous components from hand-held to large-scale [41].

Programmability and the ability to optimize for performance and power are considered major difficulties introduced by heterogeneous systems. These difficulties arise for two main reasons. First, with todays tools, it is necessary to use a different programming model for each system component: CUDA [28] or OpenCL [21] is often used to program GPU architectures, C or C++ extended with OpenMP [12] or Intel TBB [30] are used for conventional multicores, and MPI is used for distributed memory clusters. This results in an increased complexity in programming and porting across different architectures, as one must now fully develop and maintain separate copies of the code. The second difficulty is the need to schedule across device classes: the user must decide how to partition and correctly schedule the execution between the devices. This difficulty is typically compounded by each device having separate address spaces, forcing the user to take care of the allocation, deallocation, and communication of data across devices.

This paper builds on the native data parallel support of Chapel [8] to improve the programmability of heterogeneous systems containing GPU accelerator components, while retaining performance and portability across other architectures. Chapel is a high-level, general-purpose language with constructs for control of work distribution, communication, and locality. It includes support for both data and task parallelism, and allows for nested parallelism. Rather than rely completely on the compiler for performance optimizations, this work leverages Chapel's multiresolution philosophy of allowing a programmer to start with a high-level specification, then drop to a lower-level if the compiler is not providing sufficient performance. This gives expert programmers the ability to tune their algorithm's performance with capabilities similar to those of lower-level notations such as CUDA.

*Contributions and Evaluations:* The contributions of this paper are as follows:

- The presentation of a high-level and portable approach for developing applications with a single unified language, instead of libraries or annotations, that can target both GPU and multicore parallel architectures. With our approach, a single code can be used to efficiently target GPUs, traditional multicores, or a combination of both.
- Compiler transformations that map a high-level language onto GPU accelerator architectures. This also includes an algorithm for moving data automatically between a host and the device. These techniques would be applicable to other high-level languages, such as Python or Java, with the goals of targeting GPUs.
- A compiler algorithm is presented to generate code for multicores from our extended version of Chapel. In order to run efficiently, this technique extracts coarse-grain parallelism from code that contains both fine and coarse granularities.
- Experimental results to show that the performance of the hand-coded CUDA Parboil benchmarks [3] are

comparable to the Chapel implementation for both GPUs and multicores, with the Chapel code being simpler, shorter, and easier to read and maintain.

To measure the validity of the proposed approach, the Parboil benchmark suite was ported to Chapel. The Chapel compiler was then modified to generate the appropriate code based on the target architecture. This paper evaluates code generation techniques for both GPUs and multicores. The performance results show that Chapel generates codes that are competitive with the hand-tuned CUDA implementations for GPUs and their multicore counterparts.

*Outline:* This paper is organized as follows: Section II gives motivation for this work by providing a simple example. Section III describes background information on the Chapel language. Sections IV and V provide the implementation details for running on GPU and CPU architectures. Section VI discusses optimizations applied to the GPU. Section VII presents short Parboil benchmark examples written in Chapel used to target both CPUs and GPUs. Section VIII describes the initial results using the Parboil benchmark suite. Sections IX and X present related and future directions. Finally, conclusions are provided in Section XI.

## II. Motivation

As a motivating example, consider the STREAM Triad benchmark (part of the HPCC Benchmark Suite [25]), which computes a scaled vector addition. Figure 1 provides a comparison of different implementations of STREAM Triad. A CUDA implementation is provided in Figure 1(a), while Figure 1(b) is a Chapel implementation used to target a GPU. The comparison between them shows that the Chapel implementation has noticeably fewer lines of code and is more readable. This is achieved using Chapel distributions, domains, data parallel computations through the `forall` loop, and variable type inference [1], [8]. Furthermore, the Chapel implementation is easier to port. In fact, for the code in Figure 1(b), if the users wanted to target a multicore platform, they could either declare a different target distribution (shown in line 3) or simply set an environment variable specifying the target platform. To demonstrate portability, Figure 1(c) shows an implementation of STREAM Triad written for a cluster using a standard *Block* data distribution. The only difference between this implementation and the Chapel-GPU code in Figure 1(b) is the target distribution in line 3.

Figure 2 shows performance results for the STREAM Triad benchmark written in Chapel running on both a 32-node instance of the Cray XT4 supercomputer and a GPU with a problem size of $n = 85{,}983{,}914$. As the last two bars show, performance for the Chapel and CUDA implementations are equivalent. It is important to emphasize that for the cluster and Chapel-GPU results, only the declared distribution was changed. In contrast, the CUDA code does not support the same degree of portability. In addition to single-node multicores and GPUs, this code has run on large-scale cluster configurations achieving over 1.1TB/s in performance using 2048 nodes [10].

```
1  #define N 2000000
2  int main() {
3   float *host_a, *host_b, *host_c;
4   float *gpu_a, *gpu_b, *gpu_c;
5   cudaMalloc((void**)&gpu_a, sizeof(float)*N);
6   cudaMalloc((void**)&gpu_b, sizeof(float)*N);
7   cudaMalloc((void**)&gpu_c, sizeof(float)*N);
8   dim3 dimBlock(256);
9   dim3 dimGrid(N/dimBlock.x );
10  if( N % dimBlock.x != 0 ) dimGrid.x+=1;
11  set_array<<<dimGrid,dimBlock>>>(gpu_b,0.5f,N);
12  set_array<<<dimGrid,dimBlock>>>(gpu_c,0.5f,N);
13  float scalar = 3.0f;
14  STREAM_Triad<<<dimGrid,dimBlock>>>(gpu_b,
15      gpu_c, gpu_a, scalar, N);
16  cudaThreadSynchronize();
17  cudaMemCpy(host_a, gpu_a, sizeof(float)*N,
18      cudaMemcpyDeviceToHost);
19  cudaFree(gpu_a);
20  cudaFree(gpu_b);
21  cudaFree(gpu_c);
22 }
23 __global__ void set_array(float *a, float value,
24     int len) {
25  int idx = threadIdx.x+blockIdx.x*blockDim.x;
26  if(idx < len) a[idx] = value;
27 }
28 __global__ void STREAM_Triad(float *a, float *b,
29     float *c, float scalar, int len) {
30   int idx = threadIdx.x+blockIdx.x*blockDim.x;
31   if(idx < len) c[idx] = a[idx]+scalar*b[idx];
32 }
```

(a) STREAM Triad written in CUDA

```
1 const alpha = 3.0;
2 config const N = 2000000;
3 const space = [1..N] dmapped GPUDist(rank=1);
4 var A, B, C : [space] real;
5 B = 0.5;
6 C = 0.5;
7 forall (a,b,c) in (A,B,C) do
8       a = b + alpha * c;
```

(b) STREAM Triad written in Chapel for a GPU

```
1 const alpha = 3.0;
2 config const N = 2000000;
3 const space = [1..N] dmapped Block(boundingBox=[1..N]);
4 var A, B, C : [space] real;
5 B = 0.5;
6 C = 0.5;
7 forall (a,b,c) in (A,B,C) do
8       a = b + alpha * c;
```

(c) STREAM Triad written in Chapel for a cluster

Figure 1. Comparison of STREAM Triad implementations

## III. Background

This section presents a short overview of the Chapel programming language and its support for data parallelism. Chapel is a source-to-source compiler that generates C source code. For this work, Chapel will additionally generate

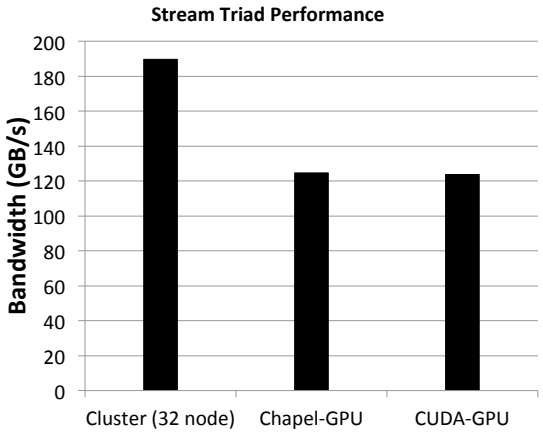



Figure 2. Results for the STREAM Triad benchmark comparing a cluster of multicores (Cray XT4 2.1 GHz Quad-Core AMD Opteron) and GPU (NVIDIA GTX280)

CUDA. A description of CUDA has been omitted, since it is readily available [5].

### A. Chapel Language Overview

Chapel is an object-oriented parallel programming language designed from first principles, rather than as an extension to an existing language. The base language supports iterator functions, OOP, generic programming, and takes advantage of type inference. Chapel was designed to facilitate programming of next-generation parallel machines. Along with X10 [13] and Fortress [6], Chapel grew out of the DARPA *High Productivity Computing Systems (HPCS)* program. Chapel support for data parallelism, index sets, and distributed arrays are derived from ZPL [34] and High Performance Fortran (HPF) [20]. Chapel's concepts of task parallelism and lightweight synchronization are derived from the Cray MTA/XMT's extensions to C and Fortran [31]. Lastly, Chapel supports interoperability with C and CUDA through C-style extern mechanisms. For interoperability with other languages such as C++, Fortran, or Python, the Babel interoperability tool [29] can be used.

### B. Domains and Distributed Arrays

The core component for data parallelism in Chapel is the concept of a *domain*, which is an extension to *regions* first described in ZPL. A domain is a language construct that describes an index space. These domains are a first-class ordered set of Cartesian indices that can have any arbitrary rank [1]. In addition to domains being iterated over by loops, they are used to describe the size and shapes of arrays. Consider the following example:

```
var D: domain(2) = [1..n, 1..n];
var A: [D] real;
```

Here `D` is a 2D domain and is initialized to contain the set of indices *(i,j)* with $i \in \{1, 2, .., n\}$ and $j \in \{1, 2, .., n\}$. The array `A` has its size defined by the domain, resulting in an $n \times n$ array.

### C. Data Parallelism in Chapel

Chapel has rich support for data parallel computation, making it ideal for SIMD-like architectures such as the GPU. The main construct for data parallelism in Chapel is the `forall` loop which iterates over the indices in a domain's index set or over a subset of the elements in an array. There is also built-in support for the *reduction* and *scan* operators. Chapel also allows users to define their own *reduction* and *scan* operations [15].

### D. Distributions (Built-in and User-defined)

Data distributions in Chapel are essentially a recipe that the compiler uses to map a computation and its associated data to the nodes where the program executes. Languages such as HPF and ZPL have support for distributed arrays, but the semantics of the distributions are hardwired in the compiler and runtime, leaving the programmer without enough flexibility to manipulate many forms of distributed data (such as sparse arrays). Similar to domains, distributions are first-class objects: they can be named, manipulated, and passed through functions.

Chapel provides a set of commonly-used distributions such as *Block* and *Cyclic*. Additionally, user-defined distributions [9], [11] enable the creation of a wide range of distributions that are application- or target-specific. User-defined distributions are developed directly in Chapel, typically using built-in features (e.g. classes, task parallelism, locales). This has the benefit that distribution developers can directly leverage the high-level facilities offered by the language rather than having to program in a lower-level language.

In order for users to write their own distribution, they must implement routines to fulfill the required interface. Interface components include the ability to create domains and arrays, wholesale assignment of index sets, iterators supporting sequential and parallel iteration over a domain, random access to elements of an array, and support for slicing and reindexing. If the user does not explicitly declare a distribution, Chapel will implicitly use a distribution that targets shared memory parallelism.

Chapel uses the `dmapped` keyword in order to map the domain's indices to the target architecture using the specified distribution. This approach is useful because Chapel distributions are designed so that the distributions can be swapped in order to change the implementation of a domain and its arrays without changing the code. This is illustrated by the examples in Figures 1(b) and 1(c), where the only difference is the target distribution on line 3. The advantage to this is that the code is cleaner and more portable: users do not need to maintain a separate code base depending on the target architecture.

## IV. Generating Code for GPU Accelerators

This section describes the Chapel language and compiler extensions that were added to target GPU platforms. Sections IV-A and IV-B discuss the Chapel GPU distribution and its associated support for domains and arrays. In Sections IV-C and IV-D, we describe support for data movement between the device and host and how code is executed on the GPU. Finally, Sections IV-E–IV-H will present code generation and other low-level facilities.

### A. GPU User-Defined Distribution

In this work, we have defined several distributions. `GPUDist` indicates that data must reside in the memory of the GPU and that computations will also be performed on the device. As shown later in this section, there are other user-defined distributions we have provided. In particular, they are used to support specialized GPU memory, as well as providing a different method of handling data movement.

In order to target the GPU, the user instantiates a `GPUDist` distribution class whose constructor takes the following arguments:

- **`rank`**: The dimensionality of the problem.
- **`blockSizeX`**, **`blockSizeY`**, **`blockSizeZ`** `[optional]`: The thread block size in the X, Y, and Z dimensions. These correlate to the equivalent thread block size dimensions used in CUDA. If the user does not explicitly initialize one of these values, they can be set using a heuristic approach based on the occupancy of the kernel and device. This is similar to the technique used in the Thrust library [2]. In the case of compiling for a multicore, this value is also useful, as will be shown in Section V.

### B. GPU Domains and Distributed Arrays

A GPU domain and its arrays are declared identically to those with standard distributions. When an array is declared with a GPU domain, `cudaMalloc()` [5] is invoked rather than the standard library `malloc()`.

In the following array declaration:

```
1 var gpuD = [1..n,1..n] dmapped GPUDist(rank=2);
2 var A: [gpuD] real;
```

line 1 defines a GPU distribution and domain with a rank of 2, while line 2 declares the 2D array `A` that is allocated on the GPU. Section IV-F describes how this technique applies to the other types of GPU memory.

### C. Data Movement

Since the GPU and the host have different address spaces, most GPU programming models require users to manage the movement of data. This complicates programming and reduces portability. To address this problem, we provide two methods of data movement in the Chapel code: implicit and explicit.

```
1 const space = [1..m] dmapped GPUDist(rank=1);
2 var input, output : [space] real;
3 input = ... // load input data
4 for 1..n {
5   forall j in space {
6     ...
7     output[j] = input[j];
8   }
9   ... = output;
10 }
```

Figure 3. Implicit Data Movement Example

*Implicit Data Movement:* In this approach, the programmer declares a single logical variable that can be accessed by the host and the device. The system automatically creates temporary storage and transfers the value(s) between the host and the GPU. The implicit data movement scheme is dependent on compiler analysis to determine when to move data. Section VI-A discusses our compiler algorithm that generates the implicit data movement code.

An example of Chapel code which utilizes implicit data movement is shown in Figure 3. Since the array `input` is read from inside of the `forall` loop, it is implicitly copied to the device before the `forall` is executed. After the loop completes execution, the array `output` is copied out to the host implicitly because it is modified inside of the `forall` loop. To the programmer, the arrays declared on line 2 are treated the same regardless of whether it is inside or outside of a `forall` loop. In other words, the arrays and their elements can be accessed or manipulated as a non-GPU array throughout the program.

*Explicit Data Movement:* For complete control of data movement between the host and device, the user can explicitly transfer the data. In this prototype, we only support synchronous data transfers.

Consider the example in Figure 4. Line 1 declares a domain and a new type of GPU distribution named `GPUExplicitDist()`. This distribution takes the same parameters as `GPUDist()`. On line 2, the user declares the corresponding host variables. On line 3, the GPU-specific arrays are declared using the distribution and domain from line 1. The assignment operation on line 6 performs the explicit data copy from host space into GPU space. After the parallel computation is complete, the assignment operation on line 11 performs the explicit copy of the results back to the host.

### D. Parallel Execution on the GPU

Chapel's `forall` loops that have been declared over a GPU domain enable parallel execution on GPUs. Each iteration of the loop corresponds to a light-weight GPU thread. Using the compiler-generated (or user-specified) block size, the compiler strip-mines the `forall` loop into block-sized units that correspond to thread blocks. As we

```
1  const space = [1..m] dmapped GPUExplicitDist(rank=1);
2  var h_input, h_output : [1..m] real;
3  var g_input, g_output : [space] real;
4  for 1..n {
5    h_input = ... // load input data
6    g_input = h_input;
7    forall j in space {
8      ...
9      g_output[j] = g_input[j];
10   }
11   h_output = g_output;
12   ... = h_output;
13 }
```

Figure 4. Explicit Data Movement Example

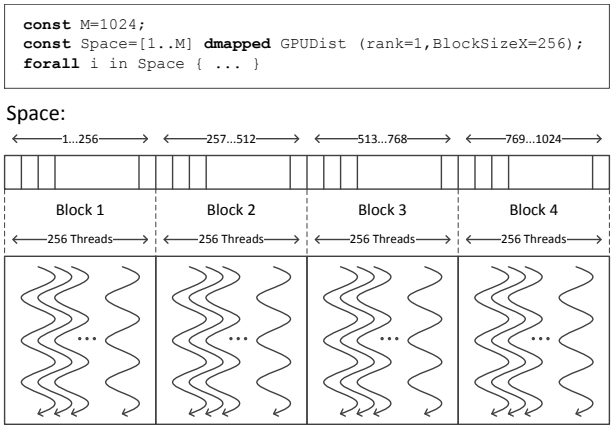



Figure 5. Mapping a Chapel 1D domain onto CUDA's thread blocks

will see in Section V, the compiler performs a similar transformation when targeting a multicore. Figure 5 is based on our previous STREAM Triad example of Figure 1(b). Here, `space` represents a distributed domain from 1 through `M`. Because `blockSizeX` = 256 and `M` = 1024, there are $\lceil \frac{1024}{256} \rceil = 4$ thread blocks for execution on the GPU. This block size provides the necessary information to map each iteration `i` of the `forall` loop onto a particular block and its associated thread.

### E. Code Generation for the GPU

A high-level view of the compilation process is shown in Figure 6. The Chapel source-to-source compiler takes as input a Chapel source file. When the compiler lowers a `forall` loop that iterates over a GPU domain, it will generate both C and CUDA source for the host and GPU. Otherwise, it will just generate C. The body of the `forall` is code generated as a CUDA kernel. By having the compiler analyze the body of the loop, it is able to determine whether any variables that are used in the loop are declared before the loop begins. In that case, the compiler will automatically pass them in as arguments to the kernel function. The host portion performs the thread block creation, along with passing the correct parameters into the CUDA kernel. As a

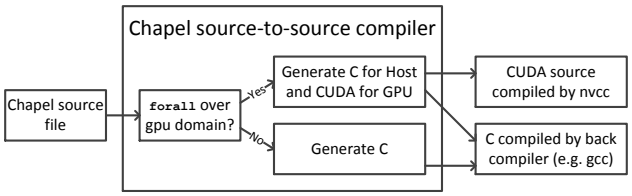



Figure 6. Overview of Chapel compilation process

final step, the generated GPU code (for both host and device) is compiled by NVIDIA's *nvcc* compiler, and the remaining code is compiled by the back-end compiler (e.g. *gcc*).

In addition to `forall` loops, Chapel supports the data parallel primitives *reduce* and *scan*, as mentioned in Section III-A. When the compiler determines that a `reduce` or `scan` operator uses data that have been declared on the GPU, the compiler makes a call to a highly-tuned library implementation of that operation. This is a similar approach as others have taken [24]. An example using the `reduce` primitive will be shown later in Section VII-A.

### F. Targeting Specialized GPU Memory Spaces

A common strategy to maximize performance for GPUs is to exploit the different physical memories in cases where locality exists [32]. On NVIDIA-based GPUs, a programmer has access to on-chip shared memory, read-only constant cache memory, and a read-only texture memory. The trade-off that occurs in using any of these specialized memories in Chapel is that the user compromises portability for performance. However, this GPU-specialized code can be transformed into code that runs on traditional processors by applying simple compiler transformations, as discussed in Section V and in other publications [37], [16], [38]. To use any of the specialized memories within the GPU, the user needs to declare their arrays using the following distributions:

*Shared Memory:* NVIDIA's GPUs offer an on-chip scratchpad memory that is user programmed to optimize for locality. Shared memory is faster than global memory and, unlike texture and constant cache memory, is writeable from the kernel. The data stored into shared memory is only visible by threads within the same thread block where the writing takes place. A compiler error occurs if the user attempts to write or read into their shared memory array when not executing a `forall`. In order to leverage this memory from Chapel, the user declares the distribution `GPUSharedDist()`.

*Constant Cache Memory:* Constant memory is used to hold constant values. This data is hardware-cached to optimize for temporal locality. There is only one cycle of latency when a cache hit occurs if all the threads in a warp access the same location. Otherwise, accesses to constant memory are serialized if threads read from different locations. With GPUs such as the NVIDIA GTX280, up to

```
1  const myspace = [1..m] dmapped GPUDist(rank=1);
2  const ccspace = [1..m] dmapped GPUCCDist(rank=1);;
3  var input: [ccspace] real;
4  var output: [myspace] real;
5  input = ... // load data into constant memory
6  for 1..n {
7    forall j in myspace {
8      ...
9      output[j] = input[j];
10   }
11   ... = output;
12 }
```

Figure 7. Constant Cache Example

64KB of data may be placed into constant memory. Data to be stored in constant memory must be declared with the `GPUCCDist()` distribution.

Figure 7 provides an example where the user would leverage the GPU constant memory within Chapel. On lines 1–4, we create the constant memory distribution, domain, and the respective constant memory array. Line 5 shows the array being loaded with data from the host. Finally, in line 9, the constant memory array is accessed, as with any typical array.

*Texture Cache Memory:* Similar to constant memory, texture memory is read-only and uses hardware caching for locality. Performance increases are seen when applications have spatial locality, as in stencil computations. The Chapel compiler does not yet support storage in this memory.

### G. Synchronization

CUDA uses `__syncthreads()` as a barrier between threads in a thread block. In Chapel, we take the approach that the programmer needs to also provide a similar synchronization primitive `thread_barrier()` for correct GPU code. While using a `thread_barrier()` is not needed to program traditional CPUs, Algorithm 1, as described in Section V, provides a compiler technique used to remove calls to `thread_barrier()` when targeting CPUs.

### H. GPU Low-Level Extensions

There are cases in which a user must leverage certain facilities offered only by the CUDA programming model. For example, CUDA provides fast math intrinsics that are implemented in hardware, such as `__fsinf()`, instead of the more accurate (but slower) `sin()`. In order to interoperate with these routines through Chapel, the compiler translates invocations to these routines into their software equivalent when targeting CPUs.

## V. Generating Code for Mutlicore

To address the portability argument, we present transformations necessary to take a single program text written with the GPU abstractions from the previous section and compile and execute it efficiently on a traditional multicore platform. We achieve this by taking advantage of coarse- and fine-grain parallelism that is exposed when targeting GPUs (i.e. CUDA thread blocks and threads within a thread block). Due to false-sharing issues and programming for locality, multicores are ideally suited for coarse-grain parallelism. For this, we assign adjacent thread blocks as work units to the processors. Therefore, we transform all `forall` loops that iterate over GPU domains to doubly-nested loops: the outer loop is a standard Chapel `forall` that iterates over blocks of threads, and the inner loop is a sequential `for` that iterates over threads within a block.

In order to preserve correctness, shared memory arrays are simply declared per-block as a non-distributed array. In other words, a non-distributed array is declared between the outer standard `forall` and the inner `for` loop.

This approach has two main advantages. First, Chapel handles the outer `forall` loop by distributing workloads evenly in a block-wise manner such that adjacent blocks are assigned to a core. Second, by serializing the inner loop, all calls to `thread_barrier()` can then be removed.

The disadvantage of the above approach is that serializing the inner loop that iterates over threads inside of a block is not always trivial. Algorithm 1 describes how we create an outer parallel loop $L_1$ and an inner sequential loop $L_2$. Algorithm 2 describes the technique used to serialize $L_2$. In the scenario where there are no calls to `thread_barrier()`, nothing is modified. If there is a `thread_barrier()` $barr$, which is not contained in any inner loop within $L_2$, we need to distribute the loop over the code before and after $barr$ in order to remove the barrier. If there is a variable $v$ declared in the original $L_2$ and it is accessed before and after $barr$, we need to move the declaration of $v$ before the distributed loops. If there is an assignment to $v$ that is dependent on `threadId`, that is accessed before and after $barr$, array expansion is applied to $v$ by the thread-block size. This is because each thread needs its own private copy of $v$. Array expansion is applied at most once for each $v$. On the other hand, if $barr$ is within an inner loop $L'$, we need to distribute $L_2$ around three sections of code: the code before $L'$, $L'$ itself, and the code after $L'$. Next, we can interchange $L'$ and the distributed $L_2$ since all the threads in a block must reach $barr$ an equal number of times. This interchange is valid since the outer loop can run in parallel. Finally, we call Algorithm 2 recursively to handle a deeper loop nest.

As an example demonstrating Algorithm 1, consider the simplified kernel of RPES from the Parboil Benchmark Suite shown in Figure 8. The code in Figure 8(a) shows the GPU-centric Chapel code, while Figure 8(b) shows the transformed Chapel code after performing the algorithm. First, the original `forall` loop is converted into another `forall` loop that iterates over thread blocks and a sequential `for` loop that iterates over threads within a block. Both the number of thread blocks and the number of threads within

**Algorithm 1:** Loop Transformation for Multicore

**Input**: List $forallList$ containing every `forall` loop with a GPU Distribution
**foreach** $L \in forallList$ **do**
  $L_1 \leftarrow$ Create standard Chapel `forall` loop that iterates over thread-blocks with loop index `blockId`;
  $L_2 \leftarrow$ Create sequential `for` loop that iterates over threads of a block with loop index `threadId`;
  **foreach** *shared array* $s \in L$ **do**
    Declare $s$ as a standard array in body of $L_1$;
  The body of $L$ becomes the body of $L_2$;
  $OutList \leftarrow$ Call `LoopDist(`$L2$`)`;
  Add $OutList$ to the body of $L_1$;

**Algorithm 2:** `LoopDist()` Function

**Input**: Sequential loop $L$
**Output**: List of Variable Declarations and Loops
**if** $\exists$ *barr* $\in L$ *where barr is a* `thread_barrier()` **then**
  List $VarDecl \leftarrow \emptyset$;
  **foreach** *variable declaration* $v \in L$ **do**
    **if** $\exists$ *an assignment to* $v$ *that is dependent on* `threadId` **and** $v$ *is accessed before and after barr* **and** $v$ *has not been expanded before* **then**
      $v_{exp} \leftarrow$expansion of $v$ by `blockSize`;
      $VarDecl \leftarrow VarDecl \cup \{v_{exp}\}$;
      Replace all occurrences of $v$ with $v_{exp}$[`threadId`];
    **else**
      $VarDecl \leftarrow VarDecl \cup \{v\}$;
  **if** $\exists$ *a loop* $L'$ *in the body of* $L$ **and** $barr \in L'$ **then**
    $DL_1 \leftarrow$ Loop with code before $L'$ surrounded by header of $L$;
    $DL_2 \leftarrow$ Loop with same header as $L$ with inner loop $L'$ as the body;
    $DL_2 \leftarrow$ Loop interchange $L$ with $L'$ in $DL_2$;
    $DL_3 \leftarrow$ Loop with code after $L'$ surrounded by header of $L$;
    **return** $< VarDecl,$ `LoopDist`$(DL_1),$ `LoopDist`$(DL_2),$ `LoopDist`$(DL_3) >$;
  **else**
    $DL_1 \leftarrow$ Distribute $L$ over code before $barr$;
    $DL_2 \leftarrow$ Distribute $L$ over code after $barr$;
    **return** $< VarDecl,$ `LoopDist`$(DL_1),$ `LoopDist`$(DL_2) >$;
**else**
  **return** $< L >$;

a thread block are computed based on the `blockSize` value specified in Section IV-A. Second, the shared memory array `Data` has to be declared just inside the new `forall` loop. Next, loop distribution of the sequential `for` loop is performed since there are calls to `thread_barrier()`. Finally, the two sequential loops are interchanged, and the inner call to `thread_barrier()` is removed.

```
1  var Data = [0..BLOCK_SIZE-1] dmapped GPUSharedDist();
2  forall gpuSpace_reduc {
3    const blid = getBlockID_x();
4    const thid = getThreadID_x();
5    Data[thid] = getData(blid, thid, ...);
6    thread_barrier();
7    for s in 0..5 do {
8      const i = 1 << (s * -1 + LOG_BLOCK_SIZE - 1);
9      if thid < i then
10       Data[thid] += Data[thid + i];
11     thread_barrier();
12   }
13   if thid == 0 then
14     ReductionSum[Offset + blid] = Data[0];
15 }
```

(a) Original code in Chapel with GPU `forall`

```
1  forall blid in 0..num_blocks-1 {
2    var Data = [0..BLOCK_SIZE-1];
3    for thid in 0..BLOCK_SIZE-1 {
4      Data[thid] = getData(blid, thid, ...);
5    }
6    for s in 0..5 {
7      for thid in 0..BLOCK_SIZE-1 {
8        const i = 1 << (s * -1 + LOG_BLOCK_SIZE - 1);
9        if thid < i then
10         Data[thid] += Data[thid + i];
11     }
12   }
13   for thid in 0..BLOCK_SIZE-1 {
14     if thid == 0 then
15       ReductionSum[Offset + blid] = Data[0];
16   }
17 }
```

(b) Translated code into Chapel multicore `forall`

Figure 8. Translation of a GPU `forall` into Multicore `forall`

## VI. Compiler Transformations and Optimizations

This section first presents a simple algorithm used to perform implicit data transfers between the host and device. Afterwards, we describe algorithms to optimize the generated GPU code.

### A. Implicit Data Transfers Between Host and Device

As mentioned in Section IV-C, implicit transfers between the host and GPU require compiler support. Algorithm 3 gives a conservative approach for determining and generating the code necessary to transfer the data. If an array that has been declared with a GPU distribution, is accessed within a `forall` loop, the compiler will compute the read and write sets of the array. If the read set is not empty, the compiler will generate code to copy the data into the device. Similarly, if the write set is not empty, the data is copied to the host after the loop completes.

In the previous example from Figure 3, the user never explicitly copies data between the device and host before calling the `forall` loop. To the user, the variable appears normal without the knowledge that it can only be used on the GPU. Based on the algorithm, the compiler will always copy data from the host to the device since `input` is read inside the `forall` loop. Also, the array `output` is written

**Algorithm 3:** Implicit Data Transfer

**Input**: Array $G$ declared with the GPU Distribution
**Input**: List $forallList$ containing every `forall` loop with a GPU Distribution
**foreach** $L \in forallList$ **do**
  **if** $USE(G) \neq \emptyset$ **then**
    Generate statement to copy $G$ from host to device before $L$ begins;
  **if** $DEF(G) \neq \emptyset$ **then**
    Generate statement to copy $G$ from device to host after $L$ completes;

to, causing the compiler to copy that data out to the host. As the example shows, since the `forall` loop is nested inside of a `for` loop, the array `input` is copied into the kernel redundantly. An improvement over the conservative approach taken here would be to analyze the complete program outside of the kernel to detect redundant copying.

### *B. Scalar Replacement of Aggregates and Dead Argument Elimination*

Compiling from a higher-level language like Chapel down to CUDA opens doors to possible optimizations. Chapel has support for multidimensional arrays with arbitrary index ranges. For this purpose, the Chapel compiler creates structures containing meta-data about the array, including start and end points, array strides, and a pointer to the raw data. The program has additional levels of indirection that it uses to look up the member variables of the structure, and therefore requires more memory operations than the typical array access in C. To avoid this increase, we perform scalar replacement of aggregates [27]. This technique flattens fields from a structure with single scalar elements. In particular, this is applied on all structures that are used within a `forall` loop that executes over an array or domain declared with a GPU distribution. The scalarized fields are then placed onto the formal argument list of the calling kernel routine. After this transformation is complete, we perform dead argument elimination on the original structures that were passed in as formals, as they are no longer necessary.

### *C. Kernel Argument Spilling to Constant Memory*

As a result of the previous optimization performed in Section VI-B, the number of formal parameters to the GPU kernel will likely have increased depending on the number of fields in the original structures. Because shared memory resources are reserved for arguments up to a maximum size of 256 bytes [5], there will be a greater performance impact as more arguments are passed. Additionally, if this size limit is exceeded, a back-end compiler error will be thrown. To get around this, Algorithm 4 describes a mechanism based on data-flow analysis that will spill scalar arguments into constant memory after a certain argument list

**Algorithm 4:** Spill Scalar Args Into Constant Mem

**Input**: List $argList$ containing each formal argument of the kernel function
**Input**: Spill threshold $threshold$
$totalSize \leftarrow 0$;
**foreach** $arg_i \in argList$ **do**
  **if** $sizeof(arg_i)+totalSize > threshold$ **and** $DEF(arg_i) = \emptyset$ **then**
    Declare constant memory variable $new_i$ outside of kernel;
    $new_i \leftarrow arg_i$;
    Remove $arg_i$ from $argList$;
    **foreach** $u_i \in USE(i)$ **do**
      $u_i \leftarrow new_i$;
  **else**
    $totalSize$+=`sizeof`$(arg_i)$;

size has been reached. In this algorithm, constant memory variables are generated with the `__constant__` modifier and are assigned from the host using the CUDA routine `cudaMemcpyToSymbol()`. In the unlikely event the algorithm is not able to spill enough of $argList$ into constant memory, the remainder of the arguments will be spilled into GPU global memory. The default threshold value for when to spill is set through a compiler flag (*--max-gpu-args=#*). It should be noted that, while this algorithm does not increase performance, it is necessary for correctness because of the limit on the number of arguments supported by the CUDA compiler.

## VII. Example Codes

The goal of this section is to discuss two examples and illustrate the portability of these codes across different parallel architectures. The examples are a 2D Jacobi method and a code to compute Coulombic Potential [35]. We present performance results for execution on a multicore, followed by results on a GPU. In the GPU case, we evaluated the two techniques of transferring data between the host and device discussed in Section IV. The hardware used for these experiments is the same as described in Section VIII-B.

### *A. 2D Jacobi*

Figure 9 illustrates the 2D Jacobi method that targets a GPU. This algorithm computes the solution of a Laplace equation over a 2D grid. The point of this code is to show an elegant high-level implementation of the algorithm rather than present the user with a low-level and highly-tuned implementation. Line 1 of the algorithm declares a GPU distribution with a rank of 2. Lines 4–5 declare two distributed domains, and lines 6–7 declare the associated arrays. Lines 12–16 are a parallel stencil computation for the GPU. Line 17 represents a maximum reduction. Finally, line 18 performs a sliced array copy of the inner domain `gPSpace`.

```
1  const gdist = new GPUDist(rank=2,
2                            blockSizeX=16,
3                            blockSizeY=16);
4  const gPSpace = [1..n, 1..n] dmapped gdist;
5  const gDomain = [0..n+1, 0..n+1] dmapped gdist;
6  var X, XNew : [gDomain] real;
7  var tempDiff : [gPSpace] real;

9  /* initialize data */
10 ....
11 do {
12   forall ij in gPSpace {
13     XNew[ij] = (X[ij+north] + X[ij+south] +
14                 X[ij+east] + X[ij+west]) / 4.0;
15     tempDiff[ij] = fabs(XNew[ij] - X[ij]);
16   }
17   delta = max reduce tempDiff;
18   X[gPSpace] = XNew[gPSpace];
19 } while (delta > epsilon);
```

Figure 9. Chapel Implementation of Jacobi 2D

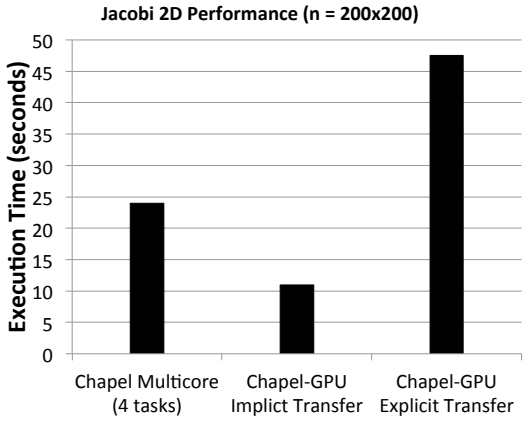



Figure 10. Performance of Jacobi 2D

Figure 10 shows the performance of this code. First, we show the performance on a multicore using 4 tasks. Then, the performance on a GPU is shown using both the implicit and explicit data transfer algorithm. It's important to note that, in this example, no lines of code were changed to the ported code between the GPUs and multicores. The Chapel GPU version of the code that uses the implicit data transfer algorithm shows degradation in performance due to the redundant data transfers that occur. This is the result of the conservative approach taken in the compiler algorithm.

### B. Coulombic Potential

The code for the Coulombic Potential (CP) application is shown in Figure 11. On lines 3–5, we declare two different GPU distributions. On lines 6–8, distributed domains are declared with the previously declared distributions. Lines 9 and 11 declare the input and output arrays. Lines 13–23 perform the parallel `forall` computation on the GPU. It should be mentioned that, on line 13, each iteration of the `forall` returns a two-tuple containing x and y coordinates.

```
1  const volmemsz = VOLSIZEX * VOLSIZEY;
2  const volmemsz_dom = [1..VOLSIZEY,1..VOLSIZEX];
3  const dst = new GPUDist(rank=1);
4  const totdst = new GPUDist(rank=2,
5          blockSizeX=BLOCKSIZEX, blockSizeY=BLOCKSIZEY);
6  const totspace = volmemsz_dom dmapped totdst;
7  const energyspace = [1..volmemsz] dmapped dst;
8  const atomspace = [1..MAXATOMS] dmapped dst;
9  var energygrid : [energyspace] real = 0.0;
10 /* initialize atominfo from input file */
11 var atominfo : [atomspace] float4 = ...;

13 forall (xindex,yindex) in totspace {
14   var energyval = 0.0;
15   var (coorx,coory) = (gspacing*xindex,
16                        gspacing*yindex);
17   for atom in atominfo {
18     var (dx,dy) = (coorx-atom.x, coory-atom.y);
19     var r_1 = 1.0 / sqrt(dx*dx + dy*dy + atom.z);
20     energyval += atom.w * r_1;
21   }
22   energygrid[yindex,xindex] += energyval;
23 }
```

Figure 11. Coulombic Potential in Chapel

Figures 12(a) and 13(a) present the results for CP running on both a GPU and a multicore. As in the previous example, no changes to the source code were made. These results will be discussed in more detail in Section VIII-C.

## VIII. Evaluation

This section presents our experimental results. In Section VIII-A, we describe the benchmarks used for the experiments. Section VIII-B describes our experimental setup. Section VIII-C evaluates the effectiveness of the compiler. For that, three types of evaluations are performed. First, we evaluate the GPU performance of the Chapel code compared with codes from the hand-coded Parboil benchmark suite. Second, we evaluate the performance of the same benchmarks on a traditional multicore platform. To compile and execute the Parboil CUDA code on a multicore, we use both the PGI CUDA-X86 [38] and Ocelot compilers [16]. The third evaluation estimates the *productivity* benefits of using Chapel. We use the difference in code size between the Chapel and CUDA source as the metric of *productivity*. This is a simple and not always compelling metric, but in this case, we believe that it gives a reasonable indication of the *productivity* advantage of using Chapel.

### A. Parboil Benchmarks

The Parboil benchmark codes used are Coulombic Potential (CP), MRI-FHD and MRI-Q [36], Rys Polynomial Equation Solver (RPES) [33], and the Two Point Angular Correlation Function (TPACF) [23]. The benchmark Sum of Absolute Differences (SAD) relies on texture memory, which our system does not support. Petri Net Simulation (PNS) was not ported due to time constraints.

In the case of the GPU evaluation, the Chapel codes that are compared use both implicit and explicit data transfers to

see what additional overhead results from the conservative implicit data transfer algorithm introduced in Section VI-A.

### B. Environmental Setup

For the GPU evaluations, each benchmark was run using an NVIDIA GTX280. The host code was executed on an Intel Quad-core 2.83GHz Q9550. For timing measurements, we used CUDA's kernel profiling mechanism (i.e. *CUDA_PROFILE=1*) that measures the execution time spent in the kernel along with execution time spent on data transfers between the host and the device.

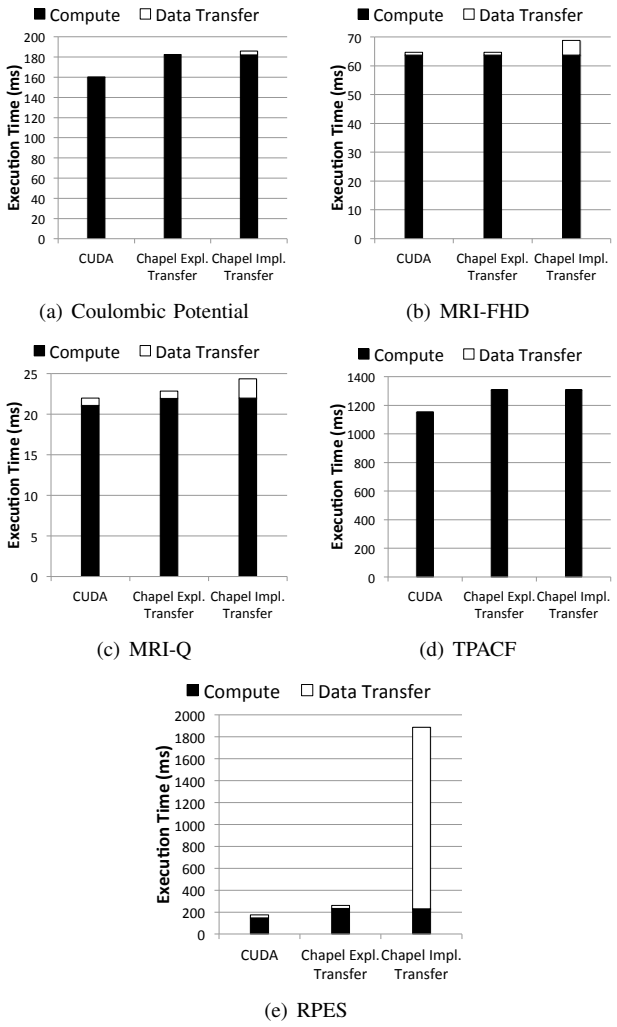
For the multicore evaluation, we use an Intel Quad-core 2.67GHz Nehalem i7 920, where each core supports two hyperthreads. The generated C code from Chapel was then compiled by the back-end PGI 11.8 C/C++ compiler. The CUDA codes were also compiled using the PGI CUDA-X86 compiler that generates OpenMP with the flags `-Mcudax86 -fast`. Similarly, the CUDA codes were compiled by the Ocelot compiler with optimization flag `optimizationLevel:full` being set.

### C. Experimental Results

Figures 12(a)–12(e) and 13(a)–13(e) demonstrate the performance, in execution time, of the Parboil benchmarks running on a GPU and multicore platform, respectively. Due to the difference in magnitude of times (milliseconds vs seconds) between GPU and multicore executions, we plot them separately and to different scales.

In Figures 12(a)–12(e), each bar is broken down into two portions: the total time spent performing data transfers vs. time performing the computation. In Figures 12(b)–12(e), the difference in compute performance was minimal between the CUDA and the Chapel implementations. When we look solely at the compute performance in Figures 12(a) and 12(d), the CUDA reference implementations have slightly better performance when compared with the Chapel implementations. In these cases, the performance difference was due to additional overhead, such as `for` loops being generated inefficiently. As the Chapel compiler matures, we expect these minor differences in compute performance to decrease.

The additional overhead due to the conservative implicit data transfer algorithm is apparent in Figures 12(b), 12(c), and 12(e). These three benchmarks demonstrate the deficiencies in our conservative approach for selecting which data to transfer into and out of the kernel. In the RPES algorithm, similar to the previous example in Figure 3, there is a parallel `forall` loop nested inside of a `for` loop. In the CUDA and Chapel explicit data transfer implementations, data is not transferred within the top-level `for` loop iterations; but, in the case of the implicit data transfer algorithm, the data is copied redundantly. The CP and TPACF algorithms in Figures 12(a) and 12(d) have an insignificant amount of overhead associated with the implicit data transfer scheme.



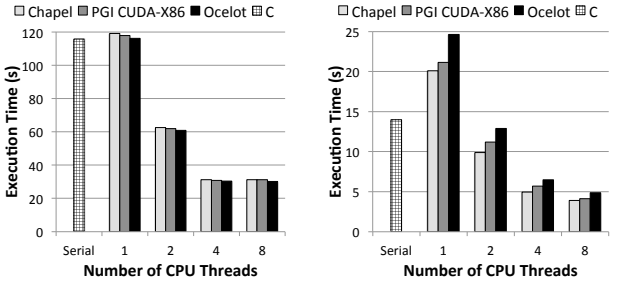

(a) Coulombic Potential (b) MRI-FHD

(c) MRI-Q (d) TPACF

(e) RPES

Figure 12. GPU Performance of the Parboil Benchmarks comparing Chapel to CUDA

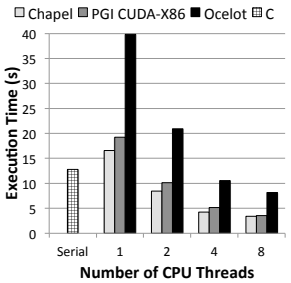
Figures 13(a)–13(e) present the original Parboil benchmarks running on a multicore platform. Here we are comparing a serial C implementation of the benchmark with that of the Chapel, PGI CUDA-X86, and Ocelot compilers. In the case of the Coulombic Potential (CP) benchmark in Figure 13(a), we see that the performance is similar for all three compilers. In the remaining Figures 13(b)–13(e), there is a noticeable performance difference between the three compilers. In particular, for Figure 13(e), we observed the PGI compiler deadlocking on the RPES benchmark when run with 8 threads. RPES and TPACF are the only benchmarks tested that use GPU shared memory and thus rely on thread barrier synchronization. One likely possibility for the difference in performance (and possibly correctness) between the compilers is that PGI and Ocelot might not be fully removing all possible barriers when targeting this code on a multicore. This would also explain why deadlocks



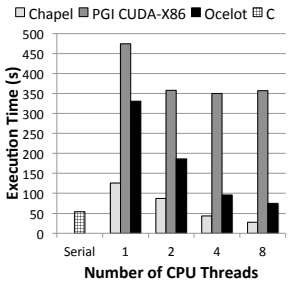

(a) Coulombic Potential



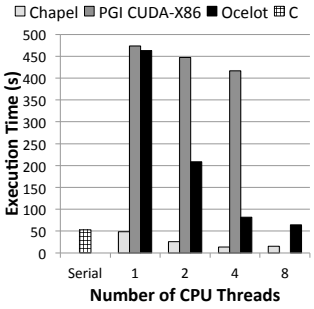

(b) MRI-FHD




(c) MRI-Q




(d) TPACF




(e) RPES

Figure 13. Multicore Performance of the Parboil Benchmarks

| Benchmark | # Lines (CUDA) | # Lines (Chapel) | % Shorter | # of Kernels |
|---|---|---|---|---|
| CP | 186 | 154 | 17.2% | 1 |
| MRI-FHD | 285 | 145 | 49.1% | 2 |
| MRI-Q | 250 | 125 | 50.0% | 2 |
| RPES | 633 | 504 | 20.4% | 2 |
| TPACF | 329 | 209 | 36.5% | 1 |

Table I
PARBOIL SOURCE CODE COMPARISON (CHAPEL VS CUDA)

occur on RPES when running with 8 threads.

Table I shows a comparison between the Chapel and CUDA implementations with the primary metric being the difference in lines of source code. In order to compute the total number of lines of code, we removed all comments, timing mechanisms, and blank lines. For all the benchmarks, the Chapel code was shorter by 17–50%. Not only is the code shorter, but as the examples in this paper show, it is cleaner and more elegant.

In summary, these results demonstrate that when using a language such as Chapel, we achieve performance that is comparable to that of the GPU-specific CUDA while making the code portable to execute efficiently on a traditional CPU platform. In addition, there are productivity and elegance gains in using Chapel over CUDA due to the lower amount of necessary code.

## IX. Related Work

Improving the programmability of accelerator architectures is currently an active area of research. The works of CUDA-X86 [4], MCUDA [37], and Ocelot [16] take the approach of having the programmer implement their algorithms in CUDA before having the compiler target a multicore platform. The approach that Chapel takes is different since it starts with a higher-level language that can be used to target GPUs, multicores, and clusters. There has also been some work providing language bindings to target the GPU [39], [22], but in these methods, the actual kernel is still expressed in CUDA or OpenCL.

Another approach that some have taken in translating their application code into the GPU accelerator space uses annotations or compiler directives on existing languages [19], [38], [7], [24], [26]. Chapel differs here in that it does not depend on annotations to induce the parallelism over a GPU, resulting in code we believe is more readable to the user. Additionally, the annotation-based approaches do not have as strong of support for high-level loop abstractions. This means a user of an annotation-based language needs to go in and decorate their loops when they want the code to be parallel. In contrast, Chapel's domains permit such things to be factored away in a more structured manner.

The work of OmpSs [17] addresses the issue of programming portability for heterogeneous multicore architectures. Our approach differs in that we use a single unified and high-level language to express computations that will execute on both GPUs and multicores. In their approach, they extend OpenMP and have the user provide implementation-specific kernels that will be mapped onto the respective devices. These approaches differ since the compute kernels are typically written as embedded CUDA or OpenCL.

X10 [14] and Habanero [40] both have shown support for GPUs, but they use different techniques to handle single-source portability across other architectures. They do not provide any mechanism for reverting to a multicore platform from a tuned GPU implementation. Also, there does not seem to be support for implicit data copies; programmers themselves have to explicitly perform the copies. Lastly, we could not find enough data to show how these approaches apply to larger scaled GPU applications.

## X. Future Directions

There are many areas of future direction for this research. For instance, one could strive to avoid exposing too much of the GPU low-level centric code to the programmer. While

this is beneficial to our compilation down to a multicore, it still forces the programmer to think in terms of CUDA in some aspects. For example, this includes relying on CUDA's synchronization primitives. One method to address this is to use the method in which programmers express their algorithms purely in terms of nested `forall`, `coforall`, and `for` loops, and leave it to the compiler to generate the necessary kernel from that. The second method, and possibly complementary approach, is to provide whole-array operation support. The Chapel language currently supports bulk-array operations that convert to parallel `forall` loops. We hope to expand on this by having the compiler optimize these operations through techniques such as loop fusion, where we can safely combine multiple `forall` loops into a single kernel.

As shown in Section VIII-C, the algorithm used for implicitly transferring data between the devices is too conservative, leaving room for improvement. One method for exploration would be to perform an interprocedural compiler analysis that looks across multiple `forall` loops and, based on the usage of the data, generates the necessary transfer code when required. The second alternative would be to incorporate the work done in the GMAC (Global Memory for Accelerators) project [18]. GMAC is library-based system that provides coherency between data on the device and on the CPU.

While Chapel already supports execution on traditional clusters containing multicores, the work presented here is primarily focused on single-node architectures containing a GPU. The next step of this research is to integrate support for clusters of single and multi-GPUs.

## XI. Conclusion

In this paper, we presented methods to increase programmer productivity by leveraging an emerging programming language built for parallelism and locality control. By utilizing Chapel's support for user-defined distributions, programmers are offered a concise and elegant approach to targeting GPU-based architectures. Additionally, we show that it is possible to be portable across multicore architectures and yet retain performance without resorting to different parallel libraries or language annotations such as pragmas or directives.

## Acknowledgments

The authors thank the anonymous reviewers, Michael Garland and Gregory Diamos of NVIDIA, and Steven Deitz from Microsoft, for their comments and answers to questions. This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001; National Science Foundation under Awards CNS 1111407, CCF 070226, and CSR-AES 0720593; DARPA under UHPC Contract Number HR0011-10-3-0007; a grant from Cray; and by a 2011-2012 NVIDIA Research Fellowship Award.

## References

[1] Chapel Specification 0.82. http://chapel.cray.com.

[2] CUDA Thrust Library. http://code.google.com/p/thrust.

[3] IMPACT Research Group. Parboil Benchmark Suite. http://impact.crhc.illinois.edu/parboil.php.

[4] Portland Group CUDA-X86. http://www.pgroup.com/resources/cuda-x86.htm.

[5] *NVIDIA CUDA C Programming Guide*, June 2011.

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[7] F. Bodin and S. Bihan. Heterogeneous Multicore Parallel Programming For Graphics Processing Units. *Sci. Program.*, 17(4):325–336, 2009.

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

[9] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, D. Iten, and V. Litvinov. Authoring User-Defined Domain Maps in Chapel. *Cray Users Group Conference (CUG)*, 2011.

[10] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi. HPC Challenge Benchmarks in Chapel. Technical report, Cray, Inc., 2009.

[11] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi. User-defined Data Distributions in Chapel: Philosophy and Framework. In *HotPar '10: Proc. 2nd Workshop on Hot Topics in Parallelism*, 2010.

[12] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach To Non-uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[14] D. Cunningham, R. Bordawekar, and V. Saraswat. GPU Programming in a High Level Language Compiling X10 to CUDA. In *X10'11 Workshop*, 2011.

[15] S. J. Deitz, D. Callahan, B. L. Chamberlain, and L. Snyder. Global-view Abstractions for User-defined Reductions and Scans. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 40–47, New York, NY, USA, 2006. ACM.

[16] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Compiler for Bulk-synchronous Applications in Heterogeneous Systems. In *PACT '10: The Nineteenth International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[17] A. Duran, E. Ayguad, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[18] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 347–358, New York, NY, USA, 2010. ACM.

[19] T. D. Han and T. S. Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61, New York, NY, USA, 2009. ACM.

[20] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

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

[22] A. Klöckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih. Pycuda: GPU Run-Time Code Generation for High-Performance Computing. *CoRR*, abs/0911.3456, 2009.

[23] S. D. Landy and A. S. Szalay. Bias and Variance of Angular Correlation Functions. *Astrophysical Journal*, 412(1):64–71, 1993.

[24] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, 2009.

[25] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, 2006.

[26] M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin. Performance Evaluation of GPUs Using the RapidMind Development Platform. In *SC '06: Proc. 2006 ACM/IEEE conference on Supercomputing*, page 181, 2006.

[27] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[28] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[29] A. Prantl, T. Epperly, S. Imam, and V. Sarkar. Interfacing Chapel with Traditional HPC Programming Languages. In *Fifth Conference on Partitioned Global Address Space Programming Model*, Galveston Island, Texas, Oct. 2011.

[30] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., 2007.

[31] M. Ringenburg and S.-E. Choi. Optimizing Loop-level Parallelism in Cray XMT (TM) Applications. In *Cray Users Group (CUG) 2009)*, Atlanta, GA, 2009.

[32] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[33] J. Rys, M. Dupuis, and H. F. King. Computation of Electron Repulsion Integrals Using the Rys Quadrature Method. *Journal of Computational Chemistry*, 4(2):154–157, 1983.

[34] L. Snyder. *A Programmer's Guide to ZPL*. MIT Press, 1999.

[35] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating Molecular Modeling Applications with Graphics Processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[36] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, and B. P. Sutton. Accelerating Advanced MRI Reconstructions on GPUs. In *CF '08: Proc. 5th conference on Computing frontiers*, pages 261–272, 2008.

[37] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 111–119, New York, NY, USA, 2010. ACM.

[38] M. Wolfe. Implementing the PGI Accelerator Model. In *GPGPU '10: Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50, 2010.

[39] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Euro-Par '09: Proc. 15th International Euro-Par Conference on Parallel Processing*, pages 887–899, 2009.

[40] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *LCPC*, volume 5898 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2009.

[41] X. Yang, X. Liao, K. Lu, Q. Hu, J. Song, and J. Su. The TianHe-1A Supercomputer: Its Hardware and Software. *J. Comput. Sci. Technol.*, 26(3):344–351, 2011.