Using the High Productivity Language Chapel to Target GPGPU Architectures

Albert Sidelnik María J. Garzarán
David Padua
Department of Computer Science
University of Illinois
{asideln2,garzaran,padua}@illinois.edu

Bradford L. Chamberlain
Cray Inc.
bradc@cray.com

Abstract

It has been widely shown that GPGPU architectures offer large performance gains compared to their traditional CPU counterparts for many applications. The downside to these architectures is that the current programming models present numerous challenges to the programmer: lower-level languages, explicit data movement, loss of portability, and challenges in performance optimization.

In this paper, we present novel methods and compiler transformations that increase productivity by enabling users to easily program GPGPU architectures using the high productivity programming language Chapel. Rather than resorting to different parallel libraries or annotations for a given parallel platform, we leverage 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 being portable across distinct classes of parallel architectures, including desktop multicores, distributed memory clusters, large-scale shared memory, and now CPU-GPU hybrids. We present experimental results from the Parboil benchmark suite which demonstrate that codes written in Chapel achieve performance comparable to the original versions implemented in CUDA.

1. Introduction

In the last few years, systems of heterogeneous components, including GPGPU accelerator architectures, have become increasingly popular. This popularity has been driven by many emerging applications in consumer and HPC markets [25]. Significant cost, power, and performance benefits are derived from executing these applications on systems containing both SIMD and conventional MIMD devices. For this reason, the interest in heterogeneous systems is not a passing fad. It is instead likely that many systems, from hand-held to large-scale [15], will soon, or already do, contain heterogeneous components.

Programmability and the ability to optimize for performance and power are considered major difficulties introduced by heterogeneous systems such as those containing GPUs which are co-processors that must be activated from conventional processors. Heterogeneity is also important for performance since conventional processors perform much better in certain classes of computations, particularly irregular computations. 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 [24] or OpenCL [17] are often used to program GPGPU architectures, while C or C++ extended with OpenMP [9] or Intel TBB [26] are used for conventional multicores, and MPI is used for distributed memory clusters. This results in a loss of portability across different parallel architectures, as one must fully port and maintain separate copies of the code to run on the different architectures. The second reason 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 movement of device data.

In this paper we build on the parallel programming language Chapel's [6] native data parallel support and provide compiler techniques to increase 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 built from the ground up in order to increase programmer productivity, while allowing control of work distribution, communication, and locality. It includes support for parallel models such as data-, task-, and nested parallelism. Rather than rely completely on the compiler for performance optimizations, we leverage Chapel's multiresolution philosophy of allowing a programmer to start with an extremely high-level specification (in this case, with Chapel's array language support) and drop to lower levels if the compiler is not providing sufficient performance. This allows expert programmers the ability to tune their algorithm's performance with similar capabilities as a lower-level model such as CUDA.

Evaluations and Contributions We evaluate the performance and programmability of our compiler prototype against applications from the Parboil benchmark suite.[1] Because the applications in Parboil are performance-tuned and handed-coded in CUDA, they make an ideal comparison since the goal of this work is to increase programmer productivity without sacrificing performance.

The contributions of this paper are as follows:

- We present a high-level and portable approach to developing applications on GPU accelerator platforms with a single unified language, instead of libraries or annotations, that can target multiple classes of parallel architectures. This includes the introduction of a user-defined distribution for GPU accelerators.
- We introduce compiler transformations that map a high-level language onto GPU accelerator architectures. This also includes a conservative algorithm for implicitly moving data between a host and the accelerator device. These techniques would be applicable to other high-level languages such as Python or Java with the goals of targeting accelerators.
- Results demonstrate that performance of the hand-coded implementations of the Parboil benchmark written in CUDA are comparable to the low-level Chapel implementation which is simpler, and easier to read and maintain.

[1] http://impact.crhc.illinois.edu/parboil.php

```
#define N 2000000
int main() {
 float *host_a, *host_b, *host_c;
 float *gpu_a, *gpu_b, *gpu_c;
 cudaMalloc((void**)&gpu_a, sizeof(float)*N);
 cudaMalloc((void**)&gpu_b, sizeof(float)*N);
 cudaMalloc((void**)&gpu_c, sizeof(float)*N);
 dim3 dimBlock(256);
 dim3 dimGrid(N/dimBlock.x );
 if( N % dimBlock.x != 0 ) dimGrid.x+=1;
 set_array<<<dimGrid,dimBlock>>>(gpu_b,0.5f,N);
 set_array<<<dimGrid,dimBlock>>>(gpu_c,0.5f,N);
 float scalar = 3.0f;
 STREAM_Triad<<<dimGrid,dimBlock>>>(gpu_b,
     gpu_c, gpu_a, scalar, N);
 cudaThreadSynchronize();
 cudaMemCpy(host_a, gpu_a, sizeof(float)*N,
     cudaMemcpyDeviceToHost);
 cudaFree(gpu_a);
 cudaFree(gpu_b);
 cudaFree(gpu_c);
}
__global__ void set_array(float *a, float value,
    int len) {
 int idx = threadIdx.x+blockIdx.x*blockDim.x;
 if(idx < len) a[idx] = value;
}
__global__ void STREAM_Triad(float *a, float *b,
    float *c, float scalar, int len) {
  int idx = threadIdx.x+blockIdx.x*blockDim.x;
  if(idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

Figure 1. STREAM Triad written in CUDA

```
1 config const N = 2000000;
2 const mydist = new dist(new GPUDist
3                (rank=1, tbSizeX=256));
4 const space : domain(1) distributed
5                mydist = [1..N];
6 var A, B, C : [space] real;
7 (B, C) = (0.5, 0.5);
8 const alpha = 3.0;
9 forall (a,b,c) in (A,B,C) do
10      a = b + alpha * c;
```

Figure 2. STREAM Triad written in Chapel for a GPU

```
config const N = 2000000;
const mydist = new dist(new Block
               (bbox=[1..N]));
const space : domain(1) distributed
               mydist = [1..N];
var A, B, C : [space] real;
(B, C) = (0.5, 0.5);
const alpha = 3.0;
forall (a,b,c) in (A,B,C) do
     a = b + alpha * c;
```

Figure 3. STREAM Triad written in Chapel for a multicore

Outline This paper is organized as follows: Section 2 gives motivation for this work. Section 3 describes background information on Chapel and the GPU architecture. Sections 4 and 5 provide the implementation details for running on a GPU. In Section 6, we present some example Chapel codes that target the GPU accelerator. Section 7 describes our initial results using the Parboil benchmark suite. Sections 8 and 9 present related and future work. We provide conclusions in Section 10.

2. Motivation

As a motivating example, consider the implementations of the STREAM Triad benchmark from the HPCC Benchmark Suite [21] in Figures 1–3. The comparison between the reference CUDA implementation in Figure 1 and the Chapel code for a GPU in Figure 2 clearly shows that the Chapel code has significantly fewer lines of code, and is simpler and more readable. This is achieved using Chapel distributions, domains, data parallel computations through the *forall* statement, and variable type inference [3, 6]. Furthermore, the Chapel implementation is easier to port. In fact, by only changing the lines of code that specify the data distribution, we could target different platforms, such as multicore, distributed memory, clusters of GPUs, or any hybrid combination of these. As an example of this, Figure 3 shows the STREAM benchmark written to run on a muticore platform where the only lines changed are the ones that declare the type of distribution (lines 2–3). While it is certainly possible to compile CUDA for other platforms, the language verbosity makes it less attractive as a general purpose parallel language.

To demonstrate portability, Figure 4 shows performance results for the STREAM benchmark running on a 32-node instance of the Cray XT4 supercomputer and a GPU for a problem size of $n = 85{,}983{,}914$. In addition to multicores and clusters, this code has run on large scale configurations achieving over 1.1TB/s in performance using 2048 nodes [7]. In comparing the performance of STREAM written for the GPU, we see that it matches the equivalent implementation written in CUDA. It's important to re-emphasize that for the cluster and Chapel-GPU bar, we used the same Chapel code where only the distribution was changed, whereas the CUDA code does not support the same degree of portability.

3. Background

This section presents a short overview of the programming language Chapel with the primary focus on data-parallelism, as we leverage this when targeting GPU accelerator implementations. Additionally we describe Nvidia's CUDA programming model, which is the language generated by our compiler.

3.1 Chapel Language Overview

Chapel is an object-oriented parallel programming language designed from first principles, rather than an extension to any existing language. The base language supports modern features such as iterators, OOP, type inference, and generic programming. Chapel was designed to improve parallel programmability and productivity in next-generation parallel machines. Chapel, along with X10 [10] and Fortress [4], is part of the DARPA *High Productivity Computing Systems (HPCS)* program. Support for data parallelism, index sets, and distributed arrays are derived from ZPL [29] and High Performance Fortran [16]. Chapel's concepts of task parallelism and lightweight synchronization are derived from Cray MTA's extensions to C and Fortran [1]. Lastly, Chapel supports interoperability to languages such as C, Fortran, and CUDA through C-style extern mechanisms.

3.1.1 Domains and Distributed Arrays

A core component for data parallelism in Chapel is the concept of *domains*, which are an extension to *regions* first described in ZPL.

A domain describes an index space, where it has a rank and an order on its elements. Furthermore, a domain is used to describe the size and shape of an array. In addition to dense rectilinear domains/arrays, Chapel supports a number of other domain types including sparse and unstructured. These domains are a first-class ordered set of Cartesian indices that can have any arbitrary rank [3].

Consider the following example:

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

D is defined as a 2D domain and is initialized to contain the set of indices *(i,j)* for all *i* and *j* such that $i \in 1,2,...,n$ and $j \in 1,2,...,n$. The array `A` is declared against the domain resulting in an $n \times n$ array.

3.1.2 Chapel 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 a physical architecture where computation executes. Past languages such as HPF and ZPL have had support for distributions, but the semantics of the distributions were tightly coupled with 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, cyclic, and block-cyclic as found in HPF. Additionally, one can use Chapel's user-defined distributions [8] to extend or write their own Chapel distribution. In order for a user to write their own distribution, a required interface for all the given routines must be implemented. 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.

In lines 2–3 of Figure 2, a user-defined GPU distribution is declared. Lines 4–5 show a distributed domain declared against the distribution. If we want to target a different platform, we only need to declare a different distribution, as shown in Figure 3, lines 2–3.

3.1.3 Data Parallelism in Chapel

Chapel has rich support for data parallel computation, making it ideal for SIMD-like architectures such as the GPU. Some of its data parallel features include *parallel iterations*, *array slicing*, *array operations*, and *reductions and scans*.

The main form of *data parallel iteration* in Chapel is made through the `forall` statement. This statement can be used to iterate over all of the indices in a domain's index set or over all of the elements of an array. *Array slicing* is the use of a domain to refer to a subset of an array's elements. This domain can be sized differently than the original domain used to declare the array. Array slicing can be beneficial in stencil computations, especially for dealing with boundary conditions [13]. Similar to Fortran 90, Chapel supports bulk *array operations* including the whole-array assignment, and scalar promotion of operators and functions against arrays. This is applicable only when the dimensions of arrays conform in size and dimensionality. There is built-in support for the standard primitives of *reduction and scan* that apply operators to combine expressions either into a scalar or an array. Chapel also supports the ability for a user to provide their own user-defined reduction and scan operations [11].

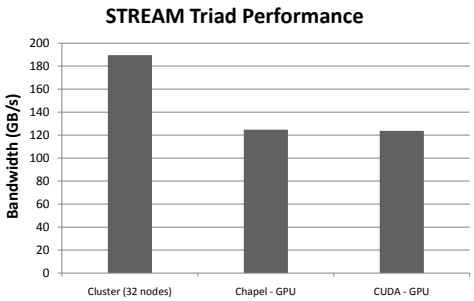


Figure 4. Results for the STREAM Triad benchmark comparing a cluster of multicores (Cray XT4 2.1 GHz Quad-Core AMD Opteron) and GPU (Nvidia GTX280)

3.2 Overview of the CUDA Programming Model and GPU Architecture

There are numerous programming models currently available that target GPU architectures. Our prototype compiler generates CUDA code.

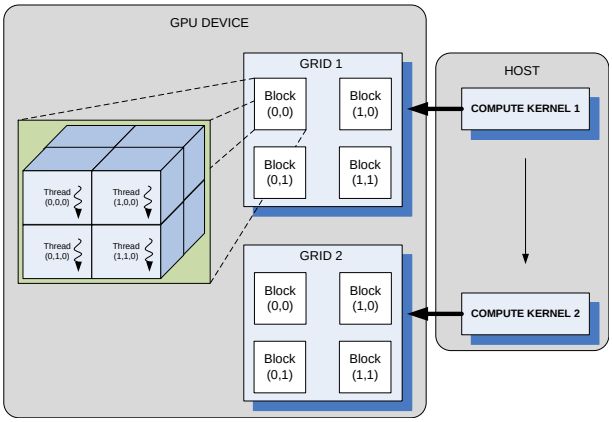


Figure 5. The mapping of compute kernels to the CUDA thread grid layout

CUDA adopts a "single program, multiple data" (SPMD) programming model. CUDA is a C subset with extensions. It exposes a two-level hierarchy of parallelism consisting of thread blocks and thread grids that are laid out in a Cartesian coordinate structure. Every thread is mapped to a particular location within a thread block, and each thread block is mapped to a location in the grid of thread blocks. Figure 5 shows a kernel routine mapped to a unique grid consisting of a Cartesian grid of 1D or 2D thread blocks, each with their own block ID coordinate. Each thread block then consists of a 1-to-3D space of individual threads represented by their thread ID in each dimension. Other kernel routines in the program can be mapped to other grid sizes. Before a programmer invokes the GPU kernel, the number of total thread blocks (in X and Y dimensions), and the number of threads within a thread block (also in X, Y, and Z dimensions) need to be specified.

For a user to access data that has been declared on the host, the data needs to be explicitly copied onto the device, and then vice-versa if the host program needs the newly computed results.

A simple example is shown in Figure 1. Lines 5-7 and 19-21 allocate and free the GPU arrays respectively. Lines 8-9 specify the block and grid sizes. Lines 23-32 show the actual routines that are executed on the GPU device. These routines are invoked on lines

11-12 and 14. In order to use the computed values in host space, the results need to be copied using the `cudaMemCpy()` routine shown in line 17.

To keep data consistent, CUDA provides a set of primitives, including atomic operations, to guard against race conditions. In particular, programmers can use `__syncthreads` as a barrier, but this barrier is restricted only for threads within a thread block. CUDA currently does not provide any easy means of a global barrier among different thread blocks.

4. Generating Code for GPGPU Accelerators

This section describes our approach for making Chapel target GPGPU platforms.

4.1 GPU User-Defined Distribution

As mentioned earlier, we utilize Chapel's user-defined distributions and implement the required routines which are part of the distribution interface. Some of the necessary interface routines include those for the allocation and deallocation of arrays residing on the GPU device, the execution of parallel iteration over a domain and arrays as part of the `forall` statement, the indexing of an array, array slicing, whole array operations, and the support for data transfers between a host and its device. So that a user can use a GPU distribution, we define a `GPUDist()` distribution class that takes the following arguments:

`rank` An integer value specifying the dimensionality of the problem.

`gridSizeX`, **`gridSizeY`** A set of integer values specifying the number of thread blocks in the grid along the X or Y dimension. These arguments correlate to CUDA's notion of grid dimensions.

`tbSizeX`, **`tbSizeY`**, **`tbSizeZ`** A set of integer values specifying the thread block size in the X, Y, or Z dimension. These also correlate to the equivalent thread block size dimensions as used in CUDA.

4.2 GPU Domains and Distributed Arrays

A GPU domain and its arrays are declared similarly to those declared against standard distributions. When an array is declared against a GPU domain, the specified memory allocation routines that are part of the distribution interface are properly invoked (in this case `cudaMalloc()` [2] rather than the standard library `malloc()`).

Consider the following array declaration:

```
1 const mydist = new dist(new GPUDist(rank=2));
2 var gpuD: domain(2) distributed mydist=[1..n,1..n];
3 var A: [gpuD] real;
```

Line 1 defines a GPU distribution with a rank of 2, while lines 2 and 3 declare a 2-D domain and array that are allocated on the GPU.

4.3 Data Movement

Since the GPU typically has a distinct address space from the host, most GPU programming models require users to manage the movement of data. This decreases programmability by making it more difficult for the user and leads to the loss of portability on other parallel platforms. To address this problem, we provide two methods of data movement in the Chapel code: implicit and explicit.

Implicit Data Movement In this approach, the programmer declares a single set of data that can be accessed by the host and the device. The system automatically creates temporary storage and transfers the data between the host and the GPU. The implicit data movement scheme is dependent on compiler analysis to determine when to move data. An example of Chapel code which utilizes implicit data movement is shown in Figure 6. To the programmer, the `input` array declared on line 4 is treated the same no matter if it is inside or outside of a `forall` loop. In other words, the array and its elements can be accessed or manipulated as any typical array would throughout the program. In Section 5.3 we discuss our compiler algorithm that generates the implicit data movement code.

```
1 const mydist = new dist(new GPUDist(rank=1,
2                                     tbSizeX=256));
3 const space: domain(1) distributed mydist = [1..m];
4 var input, output : [space] real;
5 input = ... // load input data
6 for 1..n {
7   forall j in space {
8     ...
9     output(j) = input(j);
10  }
11  ... = output;
12 }
```

Figure 6. Implicit Data Movement Example

Explicit Data Movement If the user wants lower-level flexibility, they can explicitly transfer the data and thus give the programmer complete control of movement and overlapping of data and computation [33].

Consider the example in Figure 7. On line 1, we declare a new type of GPU distribution named `GPUExplicitDist()`, which takes in the same parameters as seen with `GPUDist()`. On line 2, the distributed domain is declared as we have seen for the other distributions. On line 3, the user declares the corresponding host variables. On line 4, the GPU-specific arrays are declared against the distribution and domain declared on lines 1 and 2. The assignment operation on line 7 performs the explicit data copy from host space into GPU space. After the parallel computation is complete, line 13 copies the results back onto the host.

```
1 const mydist = new dist(new GPUExplicitDist(rank=1,
2                                 tbSizeX=256));
3 const space: domain(1) distributed mydist = [1..m];
4 var h_input, h_output : [1..m] real;
5 var g_input, g_output : [space] real;
6 for 1..n {
7   h_input = ... // load input data
8   g_input = h_input;
9   forall j in space {
10    ...
11    g_output(j) = g_input(j);
12  }
13  h_output = g_output;
14  ... = h_output;
15 }
```

Figure 7. Explicit Data Movement Example

4.4 Parallel Execution on the GPU

As discussed in Section 3.1.3, one method to enable data parallel execution is through `forall` loops. Here, each iteration of the loop represents a light-weight GPU thread. Figure 8 is based on our previous STREAM example. Here, `space` represents a distributed domain with a range from 1 through `m`. Because `tbSizeX` = 4

and m = 1024, there are $\lceil \frac{1024}{4} \rceil = 256$ thread blocks available for execution on the GPU.[2] This provides the necessary information to map each iteration i of the forall loop into a particular block and its associated thread.

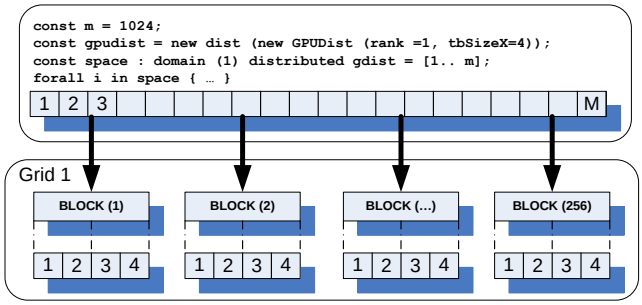


Figure 8. The mapping of a Chapel 1D domain onto CUDA's thread blocks

4.5 Code Generation for the GPU

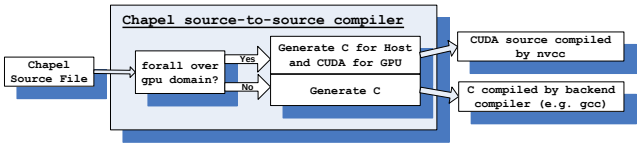


Figure 9. Overview of Chapel compile process

To generate code that executes on the GPU we take advantage of Chapel's support of user-defined distributions and data parallel computations through the forall statement. A high-level view of the compile process is shown in Figure 9. The Chapel source-to-source compiler takes as input a Chapel source file. Depending on the type of distribution declared, when the compiler analyzes a forall loop, it will either generate C source if it's a non-GPU parallel execution, or it will generate both C and CUDA source for the host and GPU respectively. In other words, a forall that targets a GPU distribution will be translated by the compiler so that the host portion performs the thread block creation along with passing in the correct parameters into the CUDA kernel. The body of the forall is contained within the CUDA kernel. As a 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 respective backend compiler (e.g. *gcc*).

In addition to forall expressions, Chapel supports the data parallel primitives *reduce* and *scan* as mentioned in Section 3.1.3. To provide these primitives on a GPU, we follow the approach others have taken [20] by calling a highly-tuned library implementation of the reduction or scan operation. This approach allows for a set of the typical associative operations across different primitive data types. An example using the *reduce* primitive will be shown later in Section 6.1.

4.6 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 [27]. 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 from using any of these specialized memories in Chapel is that the user gives up portability for performance. However, this GPU-specialized code can be transformed into code that runs on traditional processors by applying simple compiler transformations as described in MCUDA [32]. To use any of these specialized memories, the user needs to declare their arrays against the following distributions:

Shared Memory Nvidia's CUDA-capable GPUs offer an on-chip scratch pad memory that is user-programmed to optimize for locality. Compared with global memory, shared memory has the advantage of offering much faster access times. Unlike texture and constant cache memory, shared memory is writable, but only from within the parallel execution on a GPU. The scope of the data loaded into shared memory is only visible by threads within a thread block, and not across different thread blocks. 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(). The syntax to access the shared memory array is the same to access any array in Chapel.

Constant Cache Memory Constant memory is a form of read-only memory typically used to hold constant values. This data is hardware-cached to optimize for temporal locality when threads are accessing commonly accessed data such as constants. Even though constant memory is located off-chip in the device's DRAM, 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 64KB of data may be placed into constant memory. In order to use constant memory from Chapel, the user must declare the distribution named GPUCCDist(). Afterward, any arrays that need to be placed into constant memory must be copied into arrays declared against GPUCCDist() through an assignment operation.

Based on our previous example from Figure 10, we show how a user would leverage the GPU constant memory within Chapel. On lines 3 and 5–6, we create the constant memory distribution, domain, and the respective constant memory array. Line 8 shows the array being loaded with data from the host. Finally, on line 12, the constant memory array is read in as any typical array.

```
1  const mydist = new dist(new GPUDist(rank=1,
2                                      tbSizeX=256));
3  const ccdist = new dist(new GPUCCDist(rank=1));
4  const myspace: domain(1) distributed mydist=[1..m];
5  const ccspace: domain(1) distributed ccdist=[1..m];
6  var input: [ccspace] real;
7  var output: [myspace] real;
8  input = ... // load data into constant memory
9  for 1..n {
10   forall j in myspace {
11     ...
12     output(j) = input(j);
13   }
14   ... = output;
15 }
```

Figure 10. Constant Cache Example

Texture Cache Memory Similar to constant memory, texture memory is a read-only memory that has hardware caching for locality. Major performance increases are seen when applications have spatial locality, as in stencil computations. The Chapel compiler will, but does not yet, support this feature. The implementation of this feature would be similar to how constant memory is already

[2] Block size chosen is only for demonstration purposes as there is no performance benefit to sizes this small.

supported in terms of code generation and user-defined distribution support. Some of the benchmarks we will see in Section 7 are implemented using texture memory. As a temporary workaround, we replace the texture memory arrays with standard arrays.

4.7 Synchronization

As explained in Section 3.2, CUDA uses `__syncthreads()` as a barrier between threads in a thread block. We are currently implementing a compiler algorithm that performs the automatic placement of the synchronization without requiring programmer intervention. The algorithm is designed to guarantee that, in the presence of control flow, all the paths meet in the synchronization point. For this, the compiler needs to compute the dominance frontier at these join points. For the benchmarks where synchronization is necessary, we manually insert a call to a `thread_barrier()`, which is later translated to `__syncthreads()` in the generated CUDA code.

4.8 GPU Low-Level Extensions

There are certain cases where a user must leverage certain facilities only offered by the CUDA programming model. For example, CUDA provides to the user 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 user can either rely on a provided Chapel math library that links to the CUDA routines or can explicitly `extern` the necessary routine.

5. Compiler Optimizations

This section describes some of the algorithms that the Chapel compiler performs to either generate necessary accelerator code, or to enable higher performance from the generated accelerator code. This also gives the back-end GPU compiler (e.g. *nvcc*) code that is easier to optimize.

5.1 Scalar Replacement of Aggregates and Dead Argument Elimination

Compiling from a higher-level language such as Chapel down to CUDA opens doors to possible optimizations. Since Chapel is much higher-level than C, it has support for non-zero based multidimensional arrays. 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, leading to a decrease in a memory bandwidth. Since memory bandwidth is a typical limiting factor on accelerator codes, we reduce bandwidth pressure by performing scalar replacement of aggregates [23]. This technique replaces 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.

5.2 Kernel Argument Spilling to Constant Memory

As a result of the previous optimization of scalar replacement of aggregates, the number of formal parameters to the GPU kernel will likely have increased depending on the number of fields from the original structures. Because shared memory resources are reserved for arguments up to a maximum size of 256 bytes [2], there will be more of a performance impact with the more arguments that are passed. Additionally, if this size limit is exceeded, a compiler error will get thrown. To get around this, Algorithm 1 describes a mechanism based on data-flow analysis that will spill scalar arguments into constant memory after a certain argument list length has been reached. In lines 6–7, constant memory variables are declared with the `__constant__` modifier and are copied into from the host using the CUDA routine `cudaMemcpyToSymbol()`. The default threshold value on whether to spill is set through the 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 limited number of arguments that are imposed by the CUDA compiler.

Algorithm 1: Spill scalar arguments into constant memory

Input: List $argList$ containing each formal argument of the kernel function
Input: Spill threshold $threshold$
1 **foreach** i *in* $argList$ **do**
2 Compute Def Map DM_i for i;
3 Compute Use Map UM_i for i;
4 **if** *location of* $i \geq threshold$ **then**
5 **if** $DM_i = \emptyset$ **then**
6 Declare constant memory variable new_i outside of kernel;
7 $new_i \leftarrow i$;
8 Remove i from $argList$;
9 **foreach** u_i *in* UM_i **do**
10 $u_i \leftarrow new_i$;

5.3 Implicit Data Transfers Between Host and Device

As mentioned in Section 4.3, implicit transfers between the host and the GPU require compiler support. Algorithm 2 gives a conservative approach for determining and generating the necessary code to transfer the data. If the passed-in array has been declared against a GPU distribution, 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 kernel. If the write set is not empty, the compiler will copy the data out after the kernel completes.

Algorithm 2: Implicit Data Transfer

Input: Array $GPUArray$ declared with the GPU Distribution
1 Compute $GPUArray$ Use Map UM;
2 Compute $GPUArray$ Def Map DM;
3 **if** $UM \neq \emptyset$ **then**
4 Generate statement to copy $GPUArray$ from host to device before kernel invocation;
5 **if** $DM \neq \emptyset$ **then**
6 Generate statement to copy $GPUArray$ from device to host after kernel returns;

When we revisit the example from Figure 6, the user never explicitly copies data between the device and host before calling the `forall` loop. To the user, data is treated as a normal array without the detailed knowledge that it can only be used on the GPU. Based on the algorithm, the compiler will always copy data from the host onto the device since `input` is read in the `forall` loop which becomes a GPU kernel. Also, the array `output` is written 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.

6. Example Codes

The goal of this section is to illustrate that oftentimes only a small number of changes are required to port codes across diverse platforms. We make use of two sample codes: a 2-D Jacobi method, and Coulombic Potential [30]. We present performance results for execution on a multicore and a GPU. In both cases, the only difference in code is the declared distribution. For the GPU performance, we present results using the two approaches to transferring data between the host and device as discussed in Section 4. The hardware used for these experiments are the same as described in Section 7.2.

6.1 2-D Jacobi

Figure 11 demonstrates 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 highly-tuned for performance implementation. Line 1 of the algorithm declares a GPU distribution with a rank of 2. Lines 6–8 declare two distributed domains, with lines 10–11 declaring the associated arrays. On lines 16–20, we perform our parallel stencil computation on the GPU, with line 21 performing a maximum reduction also on the GPU. Finally, line 22 performs a sliced array copy of the inner domain `gPSpace`.

```
1  const gdist = new GPUDist(rank=2,
2                            tbSizeX=16,
3                            tbSizeY=16);
4  const PSpace = [1..n, 1..n],
5        BigDom = [0..n+1, 0..n+1];
6  const gPSpace : domain(2) distributed
7                          gdist = PSpace;
8  const gDomain : domain(2) distributed
9                          gdist = BigDom;
10 var X, XNew : [gDomain] real;
11 var tempDiff : [gPSpace] real;

13 /* initialize data */
14 ....
15 do {
16   forall ij in gPSpace {
17     XNew(ij) = (X(ij+north) + X(ij+south) +
18                 X(ij+east) + X(ij+west)) / 4.0;
19     tempDiff(ij) = fabs(XNew(ij) - X(ij));
20   }
21   delta = max reduce tempDiff;
22   X(gPSpace) = XNew(gPSpace);
23 } while (delta > epsilon);
```

Figure 11. Chapel Implementation of Jacobi 2D

In Figure 12 we measure the performance of the 2D Jacobi method, by having three grouped comparisons. First, we measure the performance of the algorithm on a multicore using 4 tasks. We then compare the performance of the algorithm on a GPU using both the implicit and explicit data transfer algorithm. It's important to note that, in this example, only the line declaring the distribution was changed between the multicore and GPU run. We can see from the results that only the GPU case using explicit data transfers performs better than multicore version. 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.

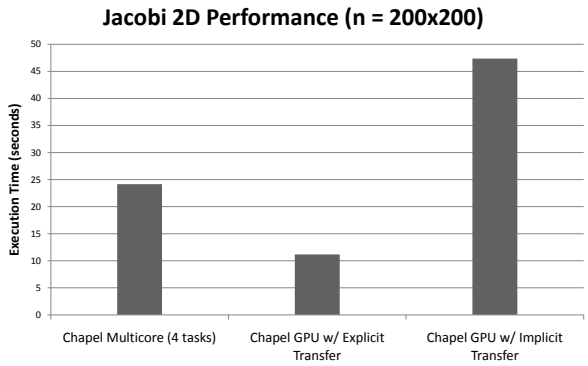


Figure 12. Performance of Jacobi 2D

6.2 Coulombic Potential

The code for the Coulombic Potential (CP) application is shown in Figure 13. On lines 2 and 4, we declare two different GPU distributions. On lines 5, 7, and 9, distributed domains are declared against the previously declared distributions. Lines 13 and 15 declare our input and output arrays. Lines 15–24 performs the parallel `forall` computation on the GPU. It should be mentioned that on line 15, each iteration of the `forall` returns a two-tuple containing x and y coordinates.

Figure 14 presents the results for CP running on both a GPU and on a multicore. As in the previous example, the only change in source code was in the distribution declaration. As the results show, both GPU implementations run about 1.85x faster than the multicore implementation using 4 tasks. The additional time spent in the implicit transfer version was negligible in this case.

```
1  const volmemsz_dom = [1..VOLSIZEY,1..VOLSIZEX];
2  const gdst = new dist(new GPUDist(rank=2,
3         tbSizeX=BLOCKSIZEX, tbSizeY=BLOCKSIZEY));
4  const gdst2 = new dist(new GPUDist(rank=1));
5  const space : domain(2) distributed
6          gdst = volmemsz_dom;
7  const energyspace : domain(1) distributed
8          gdst = volmemsz_dom;
9  const atomspace : domain(1) distributed
10         gdst2 = [1..MAXATOMS];
11 var energygrid : [energyspace] = 0.0;
12 /* initialize atominfo from input file */
13 var atominfo : [atomspace] float4 = ...;

15 forall (xindex,yindex) in space do {
16   var energyval = 0.0;
17   var (coorx,coory) = (gspacing*xindex,
18                        gspacing*yindex);
19   for atom in atominfo {
20     var (dx,dy) = (coorx-atom.x, coory-atom.y);
21     var r_1 = 1.0 / sqrt(dx*dx + dy*dy + atom.z);
22     energyval += atom.w * r_1;
23   }
24   energygrid(yindex, xindex) += energyval;
25 }
```

Figure 13. Coulombic Potential in Chapel

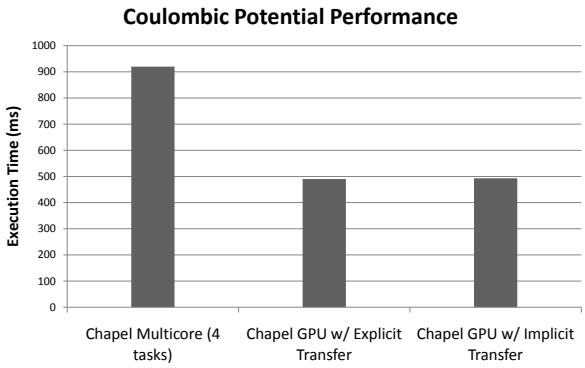


Figure 14. Performance of CP

7. Evaluation

7.1 Parboil Benchmarks

In this section, we evaluate the performance of the Parboil benchmark suite, which was performance-tuned and hand-coded for CUDA. We compare the CUDA implementation with the same benchmarks ported into Chapel. The Parboil codes that we look at are Coulombic Potential (CP), MRI-FHD and MRI-Q [31], Rys Polynomial Equation Solver (RPES) [28], and the Two Point Angular Correlation Function (TPACF) [19]. The benchmark Sum of Absolute Differences (SAD) relies on texture memory which we do not currently have results for as texture memory support is currently on going work. Petri Net Simulation (PNS) was not ported due to time constraints. 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 5.3. The version of CP discussed in this section is different from the version shown in Section 6.2. The CP used here uses constant memory and explicit loop unrolling which was not applied in the previous case. It's also important to note that, in the following results, we don't explicitly measure the impact of the previously discussed scalar replacement of aggregates and kernel argument spilling algorithms. We have seen that in the case of very simple codes such as STREAM, there is a 50% increase in performance by having those optimizations enabled. Therefore, we leave those optimizations enabled when measuring the benchmarks.

7.2 Environmental Setup

All of the benchmarks were run on a Nvidia GTX280 GPU with the CUDA 2.3 compiler as the backend for the Chapel generated source code. The Nvidia GTX280 consists of a number of streaming multiprocessors (SM), with each SM containing 8 streaming processors (SP). Additionally, each SM contains a 16KB scratchpad memory, an 8KB read-only constant cache, and an 8KB texture cache. Each SM executes groups of threads, commonly referred to as warps, in batches of 32, where the same instruction is applied to every thread inside of a warp in a SIMD-like fashion.

The host code was executed on an Intel Quad-core 2.83GHz Q9550 using Linux 2.6.30. 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.

7.3 Experimental Results

Figures 15(a)–15(e) demonstrate the performance of the benchmarks. Each bar is broken down into two portions: the total time spent performing data transfers and time performing the computation. In Figures 15(c)–15(e), the difference in compute performance

Parboil Benchmark	# Lines (CUDA)	# Lines (Chapel)	# of Kernels
CP	186	154	1
MRI-FHD	285	145	2
MRI-Q	250	125	2
RPES	633	504	2
TPACF	329	209	1

Table 1. Parboil Benchmark Source Code Comparison (Chapel vs CUDA)

was minimal between the CUDA and the Chapel implementations. When we look solely at the compute performance in Figures 15(a) and 15(b), we see that the CUDA reference implementations have better performance when compared with both of 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 should expect these changes in compute performance to decrease.

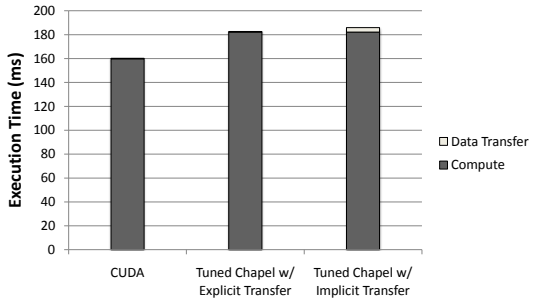
We now look at the overhead due to the conservative implicit data transfer algorithm. Figures 15(a), 15(c), 15(d), and 15(e) exhibit extra overhead, with the RPES algorithm showing the highest amount of overhead. These four benchmarks demonstrate the deficiencies in our conservative approach for selecting which data to transfer into and out of the kernel. In the RPES algorithm, there is a parallelizable `forall` loop nested inside of a `for` loop. In the CUDA and Chapel explicit data transfer implementations, data is not transferred across the top-level `for` loop iterations, but in the case of the implicit data transfer algorithm this occurs. The TPACF algorithm in Figure 15(b) has no overhead associated with the implicit data transfer scheme, thus appearing negligible in the results.

Table 1 shows an additional 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 in the source files, we removed all comments and any of the timing mechanisms. While only looking at the total number of lines is not a precise measurement in productivity, it should show to the reader that these benchmarks when ported to Chapel required less code.

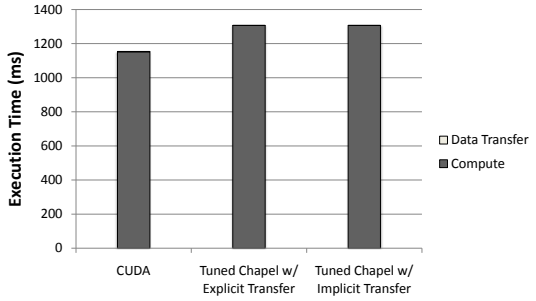
These results demonstrate that when using a language such as Chapel, we achieve performance that is comparable to that of a GPU specific language such as CUDA, while making the programmability easier for the user.

8. Related Work

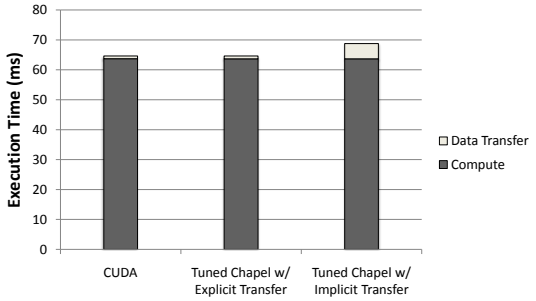
Improving the programmability of accelerator architectures is currently an active area of research. The works of MCUDA [32] and Ocelot [12] take the approach of having the programmer implement their algorithms in CUDA, and then having the compiler target a multicore platform. The Chapel approach differs these compilers in that Chapel starts with higher-level language, especially when compared with CUDA or OpenCL. This allows for more general purpose code, compared with the strictly data-parallel code that one is forced to use when writing CUDA. Another approach that some have taken in translating their application code into the GPU accelerator space is through the use of annotations or compiler directives on existing languages [5, 14, 20, 22, 34]. Our approach differs in that we do not depend on annotations to induce the parallelism over a GPU, resulting in what we believe to be a cleaner approach. There has been some work providing language bindings to target the GPU [18, 35], but in essence, the compute kernels typically are written as embedded CUDA, which does not increase programmability. Lastly, natural competitors to Chapel are the languages X10 and Fortress, but to the best of our knowledge, neither have publications or official releases supporting GPGPU support.



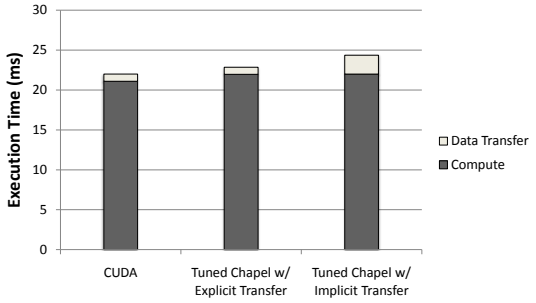
(a) Coulombic Potential



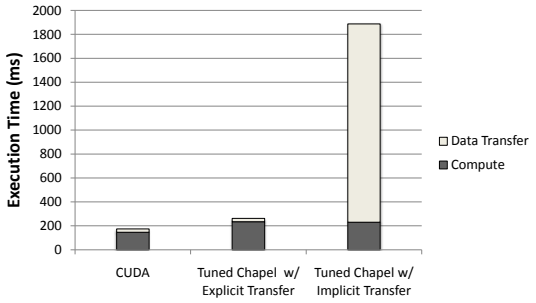
(b) TPACF



(c) MRI-FHD



(d) MRI-Q



(e) RPES

Figure 15. Performance of the Parboil Benchmarks comparing Chapel to CUDA

9. Future Work

As an extension to user-defined distributions, we are working to increase portability by introducing the concept of a high-level "meta"-distribution (e.g. `metaDist()`) that is an aggregate distribution consisting of lower-level distributions. This new meta-distribution gets instantiated through a compiler flag or a system run-time check that determines which one of the distributions should be used based on the hardware where the code is running. This meta-distribution allows for full portability by having just a single source code that runs efficiently in different platforms, by having the programmer specify all the appropriate distributions for the different targets, or by using a default distribution when one does not exist for the target platform.

One weakness that we are looking to alleviate deals with exposing too much of the GPU low-level centric code to the programmer. This includes still relying on CUDA's thread blocks, grids, etc. One possible method to fix this and increase programmability is to raise the level of abstraction and add array operation support. Having support for these operations will increase programmability, especially when targeting these devices. Currently the language 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 fuse multiple `forall` loops into a single kernel.

Our algorithm for implicitly transferring data between the devices is too conservative (as shown in Section 7.3) and that there is room for improvement. We are investigating new optimizations (either runtime or at compile time) to increase the performance of the implicit scheme so that it is much closer to the explicit scheme. One possible method that we are exploring to prevent redundant/unnecessary copies would be to perform a compiler analysis that looks across multiple forall statements and based on the usage of the data, the compiler can generate the necessary transfer code when required.

Lastly, we are investigating compiler techniques for automatically detecting the ideal forms of GPU memory to target based on locality analysis. For example, if there is sufficient locality, the compiler should be able to automatically generate code to use shared memory rather than global.

10. Conclusion

In this paper, we presented a methods to increase programmer productivity, by leveraging a new programming language built for parallelism and locality control to target GPU-based architectures. In addition to GPUs, we show that it is possible to be portable across distinct parallel architectures, and yet retain performance without resorting to different parallel libraries or language annotations such as pragmas or directives.

Acknowledgments

This material is based upon work supported by the National Science Foundation under award CCF 0702260, by Cray Inc. under Agreement No. Cray-SRA-2010-01696, and a 2010-2011 Nvidia Research Fellowship Award.

References

[1] *Cray MTA-2 Programmer's Guide*. 2005.

[2] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2009.

[3] Chapel Specification 0.795, April 2010. http://chapel.cray.com.

[4] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt.

The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

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

[6] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[7] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. HPC Challenge Benchmarks in Chapel. Technical report, Cray, Inc., 2009.

[8] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. User-defined Data Distributions in Chapel: Philosophy and Framework. In *HotPar '10: Proc. 2nd Workshop on Hot Topics in Parallelism*, 2010.

[9] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek 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.

[11] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence 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.

[12] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan 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.

[13] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, and David Padua. Writing Productive Stencil Codes with Overlapped Tiling. *Concurr. Comput. : Pract. Exper.*, 21(1):25–39, 2009.

[14] Tianyi David Han and Tarek 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.

[15] Paul Henning and Andrew B. White Jr. Trailblazing with Roadrunner. *Computing in Science and Engineering*, 11:91–95, 2009.

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

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

[18] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda: GPU Run-Time Code Generation for High-Performance Computing. *CoRR*, abs/0911.3456, 2009.

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

[20] Seyong Lee, Seung-Jai Min, and Rudolf 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.

[21] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, 2006.

[22] Michael D. McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance Evaluation of GPUs Using the RapidMind Development Platform. In *SC '06: Proc. 2006 ACM/IEEE conference on Supercomputing*, page 181, 2006.

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

[24] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[25] James C. Phillips and John E. Stone. Probing Biomolecular Machines With Graphics Processors. *Commun. ACM*, 52(10):34–41, 2009.

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

[27] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei 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.

[28] 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.

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

[30] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating Molecular Modeling Applications with Graphics Processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[31] Samuel S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton. Accelerating Advanced MRI Reconstructions on GPUs. In *CF '08: Proc. 5th conference on Computing frontiers*, pages 261–272, 2008.

[32] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei 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.

[33] Sundaresan Venkatasubramanian and Richard W. Vuduc. Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU systems. In *ICS '09: Proc. of the 23rd int. conference on Supercomputing*, pages 244–255.

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

[35] Yonghong Yan, Max Grossman, and Vivek 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.