Automatic CPU-GPU Communication Management and Optimization

Thomas B. Jablin Prakash Prabhu James A. Jablin †
Nick P. Johnson Stephen R. Beard David I. August

Princeton University, Princeton, NJ
{tjablin, pprabhu, npjohnso, sbeard, august}@cs.princeton.edu

†Brown University, Providence, RI
jjablin@cs.brown.edu

Abstract

The performance benefits of GPU parallelism can be enormous, but unlocking this performance potential is challenging. The applicability and performance of GPU parallelizations is limited by the complexities of CPU-GPU communication. To address these communications problems, this paper presents the first fully automatic system for managing and optimizing CPU-GPU communcation. This system, called the CPU-GPU Communication Manager (CGCM), consists of a run-time library and a set of compiler transformations that work together to manage and optimize CPU-GPU communication without depending on the strength of static compile-time analyses or on programmer-supplied annotations. CGCM eases manual GPU parallelizations and improves the applicability and performance of automatic GPU parallelizations. For 24 programs, CGCM-enabled automatic GPU parallelization yields a whole program geomean speedup of 5.36x over the best sequential CPU-only execution.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Experimentation, Performance

Keywords GPU, communication, management, optimization

1. Introduction

Currently, even entry-level PCs are equipped with GPUs capable of hundreds of GFLOPS. Real applications, parallelized to take advantage of GPUs, regularly achieve speedups between 4x and 100x [8, 10, 20]. Unfortunately, parallelizing code for GPUs is difficult due to the typical CPU-GPU memory architecture. The GPU and CPU have separate memories, and each processing unit may efficiently access only its own memory. When programs running on the CPU or GPU need data-structures outside their memory, they must explicitly copy data between the divided CPU and GPU memories.

The process of copying data between these memories for correct execution is called *Managing Communication*. Generally, programmers manage CPU-GPU communication with `memcpy`-style functions. Manually managing CPU-GPU communication is tedious and error-prone. Aliasing pointers, variable sized arrays, jagged arrays, global pointers, and subversive typecasting make it difficult for programmers to copy the right data between CPU and GPU memories. Unfortunately, not all communication management is efficient; cyclic communication patterns are frequently orders of magnitude slower than acyclic patterns [15]. Transforming cyclic communication patterns to acyclic patterns is called *Optimizing Communication*. Naïvely copying data to GPU memory, spawning a GPU function, and copying the results back to CPU memory yields cyclic communication patterns. Copying data to the GPU in the preheader, spawning many GPU functions, and copying the result back to CPU memory in the loop exit yields an acyclic communication pattern. Incorrect communication optimization causes programs to access stale or inconsistent data.

This paper presents CPU-GPU Communication Manager (CGCM), the first fully automatic system for managing and optimizing CPU-GPU communication. Automatically managing and optimizing communication increases programmer efficiency and program correctness. It also improves the applicability and performance of automatic GPU parallelization.

CGCM manages and optimizes communication using two parts, a run-time library and a set of compiler passes. To manage communication, CGCM's run-time library tracks GPU memory allocations and transfers data between the CPU memory and GPU memory. The compiler uses the run-time library to manage and optimize CPU-GPU communication without strong analysis. By relying on the run-time library, the compiler postpones, until run-time, questions that are difficult or impossible to answer statically. Three novel compiler passes for communication optimization leverage the CGCM run-time: map promotion, alloca promotion, and glue kernels. Map promotion transforms cyclic CPU-GPU communication patterns into acyclic communication patterns. Alloca promotion and glue kernels improve the applicability of map promotion.

The contributions of CGCM over prior work are:

- The first fully automatic CPU-GPU communication *management* system.
- The first fully automatic CPU-GPU communication *optimization* system.

Figure 1 shows a taxonomy of CPU-GPU communication management techniques. No prior work fully automates CPU-GPU communication, but several semi-automatic techniques can manage communication if programmers supply annotations [12, 24, 26]. Some of these communication management techniques are strongly coupled with automatic parallelization systems [12, 24]; others are not [26]. None of the semi-automatic communication systems optimize CPU-GPU communications. Some prior automatic parallelization techniques require manual communication [3, 13, 25]. The earliest GPU parallelization systems feature manual parallelization and manual communication [6, 11, 16]. These systems remain the most popular. CGCM enables fully-automatic communication management for manual and automatic parallelizations.

Communication management is also a problem for distributed memory systems. Inspector-executor techniques automatically manage communication for distributed memory systems [4, 14, 22]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

Copyright © 2011 ACM 978-1-4503-0663-8/11/06. . . $10.00

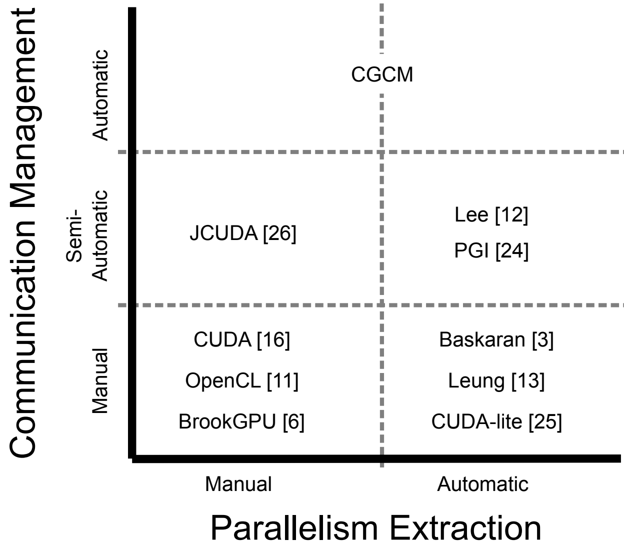

Figure 1. A taxonomy of related work showing automatic and manual communication and parallelization as independent axes.

but have not been used for CPU-GPU systems. Inspector-executor techniques can reduce the number of bytes transferred, but the overall communication pattern remains cyclic.

We have coupled CGCM with an automatic parallelizing compiler to produce a fully automatic GPU parallelization system. To compare a strong cyclic communication system against CGCM's acyclic communication, we adapted inspector-executor to GPUs. Across 24 programs, CGCM coupled with automatic parallelization shows a geomean whole program speedup of 5.36x over sequential CPU-only execution versus 0.92x for inspector-executor.

This paper provides a detailed description of the design and implementation of CGCM and presents an evaluation of the system. Section 2 describes background information about the challenges of CPU-GPU communications and limitations of related work. Section 3 describes the run-time support library. Section 4 explains how the compiler automatically inserts calls to the CGCM library. The map promotion, alloca promotion, and glue kernel techniques for optimizing CPU-GPU communication appear in Section 5. Section 6 interprets the performance results for 24 programs taken from the PARSEC [5], StreamIt [23], Rodinia [7], and PolyBench [17] benchmark suites. This section also compares CGCM to an idealized inspector-executor system. Section 7 discusses related work, and Section 8 concludes.

2. Motivation

The divided CPU-GPU memories create communication management and optimization difficulties that motivate CGCM. Managing communication means copying data between CPU and GPU memories to ensure each processing unit has the data it needs. Manual communication management is time-consuming and error-prone, and semi-automatic communication management in prior work has limited applicability. Optimizing communication means transforming communication patterns to remove cyclic dependencies. Optimizing communication is vital for program performance, but increases the difficulty of communication management. Lack of optimized communication limits the performance of prior semi-automatic communication management frameworks. CGCM is the first fully-automatic communication management and optimization system.

Listing 1: Manual explicit CPU-GPU memory management

```
  char *h_h_array[M] = {
    "What so proudly we hailed at the twilight's last gleaming,",
    ...
  };

□ __global__ void kernel(unsigned i, char **d_array);

  void foo(unsigned N) {
    /* Copy elements from array to the GPU */
▨   char *h_d_array[M];
▨   for(unsigned i = 0; i < M; ++i) {
▨     size_t size = strlen(h_h_array[i]) + 1;
▨     cudaMalloc(h_d_array + i, size);
▨     cudaMemcpy(h_d_array[i], h_h_array[i], size,
▨                cudaMemcpyHostToDevice);
▨   }

    /* Copy array to the GPU */
▨   char **d_d_array;
▨   cudaMalloc(&d_d_array, sizeof(h_d_array));
▨   cudaMemcpy(d_d_array, h_d_array, sizeof(h_d_array),
▨              cudaMemcpyHostToDevice);

□   for(unsigned i = 0; i < N; ++i)
■     kernel<<<30, 128>>>(i, d_d_array);

    /* Free the array */
▨   cudaFree(d_d_array);

    /* Copy the elements back, and free the GPU copies */
▨   for(unsigned i = 0; i < M; ++i) {
▨     size_t size = strlen(h_h_array[i]) + 1;
▨     cudaMemcpy(h_h_array[i], h_d_array[i], size,
▨                cudaMemcpyDeviceToHost);
▨     cudaFree(h_d_array[i]);
▨   }
  }
```

□ Useful work ▨ Communication ■ Kernel spawn

Listing 2: Automatic implicit CPU-GPU memory management

```
  char *h_h_array[M] = {
    "What so proudly we hailed at the twilight's last gleaming,",
    ...
  };

□ __global__ void kernel(unsigned i, char **d_array);

  void foo(unsigned N) {
□   for(unsigned i = 0; i < N; ++i) {
■     kernel<<<30, 128>>>(i, h_h_array);
    }
  }
```

□ Useful work ▨ Communication ■ Kernel spawn

2.1 Communication Management

Communication management presents a major difficulty for manual and automatic GPU parallelizations. Current GPU programming languages, such as CUDA and OpenCL, require manual communication management using primitive `memcpy`-style functions.

Framework	Comm. Opti.	Requires Annotations	Applicability						Acyclic Communication
			CPU-GPU	Aliasing Pointers	Irregular Accesses	Weak Type Systems	Pointer Arithmetic	Max Indirection	
JCUDA [26]	×	Yes	✓	✓	✓	×	×	∞	No
Named Regions [12]	×	Yes	✓	✓	×	✓	×	1	No
Affine [24]	×	Yes	✓	×	×	✓	×	1	With Annotation
Inspector-Executor [4, 14, 22]	×	Yes	×	×	✓	✓	×	1	No
CGCM	✓	No	✓	✓	✓	✓	✓	2	After Optimization

Table 1. Comparison between communication systems

Manually copying complex data-types from CPU memory to GPU memory is tedious and error-prone. Listing 1 shows how a CUDA programmer might copy an array of strings to and from the GPU, allocating and freeing memory as necessary. Almost every line of code in the listing involves communication management and not useful computation. Furthermore, the programmer must manage buffers and manipulate pointers. Buffer management and pointer manipulation are well-known sources of bugs.

Automatic communication management avoids the difficulties of buffer management and pointer manipulation, improving program correctness and programmer efficiency. However, automatically managing communication based on compile-time static analysis is impossible for general C and C++ programs. In C and C++, any argument to a GPU function could be cast to a pointer, and an opaque pointer could point to the middle of data-structures of arbitrary size. Prior work restricts its applicability to avoid these difficulties [12, 24, 26].

Table 1 compares the applicability of prior communication management techniques. Prior GPU techniques [12, 24, 26] require programmer annotations, do not handle the full generality of pointer arithmetic and aliasing, create cyclic CPU-GPU communication patterns by default, and do not optimize communication. Consequently, prior automatic communication management techniques have limited applicability and frequently yield poor performance. To gain acceptance in the GPU programming community, automatic communication management techniques must overcome these limitations.

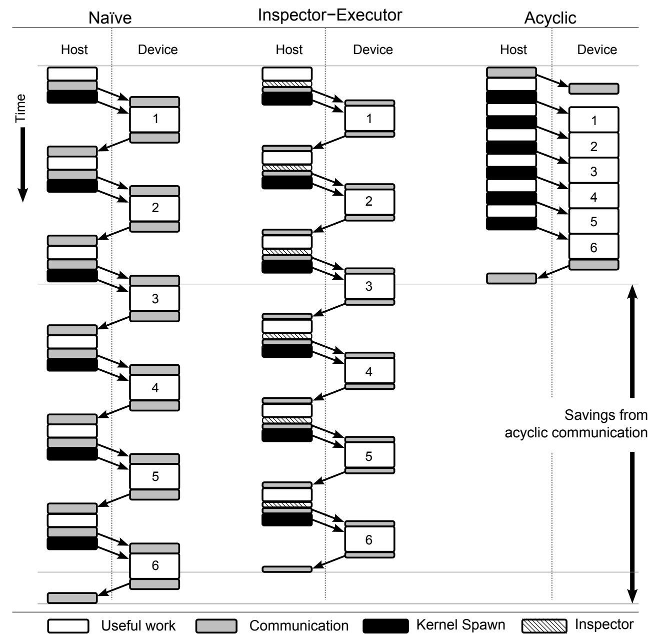

Figure 2. Execution schedules for naïve cyclic, inspector-executor, and acyclic communication patterns

2.2 Communication Optimization

Figure 2 shows execution schedules for three communication patterns: a naïve cyclic pattern, an inspector-executor pattern, and an acyclic pattern. In the naïve schedule, data transfers to and from GPU memory create cyclic dependencies, forcing the CPU and GPU to wait for each other. GPU programmers understand cyclic dependencies dramatically increase execution time. Nevertheless, all prior automatic GPU communication systems generate cyclic communication [12, 24, 26]. Cyclic communication patterns prevent these systems from efficiently parallelizing complex programs that launch many GPU functions.

Inspector-executor systems manage communication in clusters with distributed memory [4, 14, 22]. The inspector-executor approach breaks loops into an inspector, a scheduler, and an executor. The inspector simulates the loop to determine which array offsets the program reads or writes during each iteration. After the inspection, a scheduler assigns loop iterations to cluster nodes and transfers the appropriate data. Executors on each cluster node compute loop iterations in parallel. The inspector-executor schedule in Figure 2 outperforms the naïve communication schedule because the benefit of communicating only the needed array elements exceeds the cost of the inspection. However, in real programs the cost of sequential inspection may exceed the benefits of parallel execution. Generally, inspector-executor is cyclic since it transfers only the needed bytes for a single iteration at a time. Acyclic inspector-executor variants require annotations or strong static analysis [18, 21].

Figure 2 shows the execution schedule for acyclic CPU-GPU communications. Removing cyclic communication avoids the latency of back-and-forth communication and allows the CPU and GPU to work in parallel. The performance benefit of acyclic communication is significant.

2.3 Overview

CGCM avoids the limitations of prior work by employing a run-time support library and an optimizing compiler to automatically manage and optimize CPU-GPU communication, respectively. The run-time library determines the size and shape of data-structures during execution. The compiler uses the run-time library to manage memory without strong analysis and then optimizes communications to produce acyclic patterns. CGCM has two restrictions:

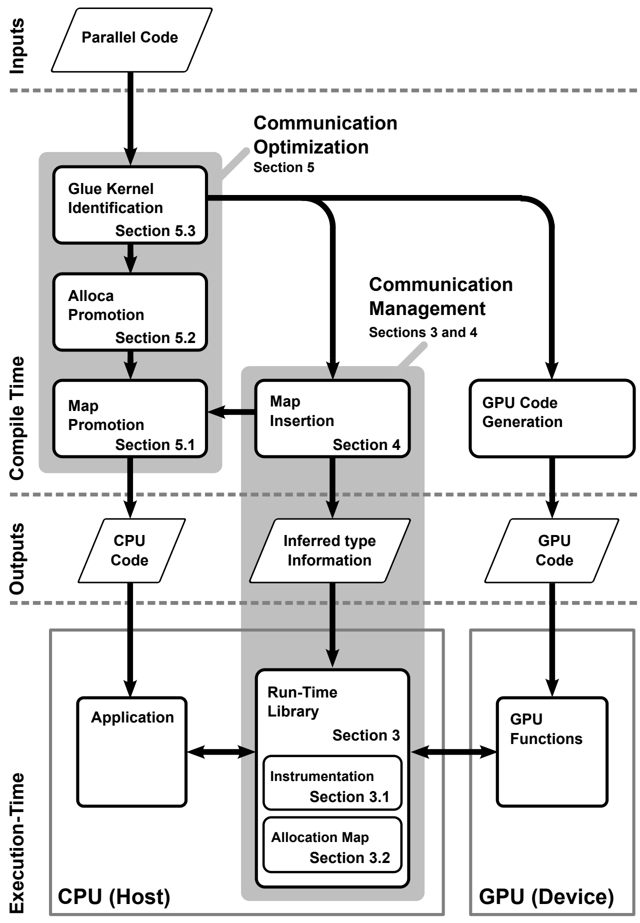

Figure 3. High-level overview of CGCM

CGCM does not support pointers with three or more degrees of indirection, and it does not allow pointers to be stored in GPU functions. Using CGCM, programmers can write the example code in Listing 1 as Listing 2. Replacing explicit CPU-GPU communication with CGCM yields dramatically shorter, simpler, clearer code and prevents several classes of programmer error.

Figure 3 shows a high-level overview of CGCM's transformation and run-time system. The run-time library provides *mapping* functions which translate CPU pointers to equivalent GPU pointers (Section 3). The compiler inserts mapping functions to manage CPU-GPU communication (Section 4). Map promotion optimizes CPU-GPU communication by transferring data to the GPU early and keeping it there as long as possible (Section 5). Two enabling transformations, glue kernels and allocation promotion, improve map promotion's applicability.

3. Run-Time Library

The CGCM run-time library enables automatic CPU-GPU communication management and optimization for programs with complex patterns of memory allocation and unreliable typing. To accomplish this feat, the run-time library correctly and efficiently determines which bytes to transfer. For correctness, the run-time library copies data to the GPU at *allocation unit* granularity. A pointer's allocation unit comprises all bytes reachable from a pointer by valid pointer arithmetic. Using the concept of allocation units, the run-time library can support the full semantics of pointer arithmetic without strong static analysis. A one-to-one mapping between allocation units in CPU memory and allocation units in GPU memory allows the run-time library to translate pointers.

3.1 Tracking Allocation Units

Unlike inspector-executor systems which manage memory on a per-byte or per-word granularity, CGCM manages memory at the granularity of *allocation units*. CGCM determines which bytes to transfer by finding allocation information for opaque pointers to the stack, heap, and globals. In C and C++, an allocation unit is a contiguous region of memory allocated as a single unit. Blocks of memory returned from `malloc` or `calloc`, local variables, and global variables are all examples of allocation units. All bytes in an array of structures are considered part of the same allocation unit, but two structures defined consecutively are different allocation units. Transferring entire allocation units between CPU and GPU memories ensures that valid pointer arithmetic yields the same results on the CPU and GPU, because the C99 programming standard [1] stipulates that pointer arithmetic outside the bounds of a single allocation unit is undefined.

Copying an allocation unit between CPU and GPU memories requires information about the allocation unit's base and size. The run-time library stores the base and size of each allocation unit in a self-balancing binary tree map indexed by the base address of each allocation unit. To determine the base and size of a pointer's allocation unit, the run-time library finds the greatest key in the *allocation map* less than or equal to the pointer. Although allocation information for global variables is known at compile-time, stack and heap allocations change dynamically at run-time. The run-time library uses different techniques to track the allocation information for global, stack, and heap memory.

- To track global variables, the compiler inserts calls to the run-time library's `declareGlobal` function before `main`. Declaring addresses at run-time rather than at compile-time or link-time avoids the problems caused by position independent code and address space layout randomization.
- To track heap allocations, the run-time library wraps around `malloc`, `calloc`, `realloc`, and `free`. These wrappers modify the allocation map to reflect the dynamic state of the heap at run-time.
- To track escaping stack variables, the compiler inserts calls to `declareAlloca`. The registration expires when the stack variable leaves scope.

3.2 CPU-GPU Mapping Semantics

Table 2 lists each function in the run-time library and its arguments. The run-time library contains functions that translate between CPU and GPU pointers. The three basic functions are `map`, `release`, and `unmap`. Each of these functions operates on opaque pointers to CPU memory.

- *Mapping* a pointer from CPU to GPU memory copies the corresponding allocation unit to GPU memory, allocating memory if necessary. The run-time library employs reference counting to deallocate GPU memory when necessary. Mapping a pointer from CPU to GPU memory increases the GPU allocation unit's reference count.
- *Unmapping* a CPU pointer updates the CPU allocation unit with the corresponding GPU allocation unit. To avoid redundant communication, the run-time library will not copy data if the CPU allocation unit is already up-to-date. Since only a GPU function can modify GPU memory, `unmap` updates each allocation unit at most once after each GPU function invocation.
- *Releasing* a CPU pointer decreases the corresponding GPU allocation unit's reference count, freeing it if necessary.

Function prototype	Description
`map(ptr)`	Maps from host to device pointer, allocating and copying memory if necessary. Increases the allocation unit's reference count.
`unmap(ptr)`	Maps to host memory if the allocation unit's epoch is not current. Updates the allocation unit's epoch.
`release(ptr)`	Decreases the reference count of the allocation unit. If the reference count is zero, frees resources.
`mapArray(ptr)`	Maps from host to device pointer, allocating and copying memory if necessary. Increases the allocation unit's reference count.
`unmapArray(ptr)`	Maps to host memory if the allocation unit's epoch is not current. Updates the allocation unit's epoch.
`releaseArray(ptr)`	Decreases the reference count of the allocation unit. If the reference count is zero, frees resources.
`declareAlloca(size)`	Allocates memory on the stack and registers it with the run-time library.
`declareGlobal(name,ptr,size,isReadOnly)`	Registers a global with the run-time library.

Table 2. CGCM's run-time library interface

Listing 3: Listing 2 after the compiler inserts run-time functions (unoptimized CGCM).

```
   char *h_h_array[M] = {
     "What so proudly we hailed at the twilight's last gleaming,",
     ...
   };

□  __global__ void kernel(unsigned i, char **d_array);

   void foo(unsigned N) {
□    for(unsigned i = 0; i < N; ++i) {
▩      char **d_d_array = mapArray(h_h_array);
■      kernel<<<30, 128>>>(i, d_d_array);
▩      unmapArray(h_h_array);
▩      releaseArray(h_h_array);
     }
   }
```

□ Useful work ▩ Communication ■ Kernel spawn

Algorithm 1: Pseudo-code for `map`

Require: ptr is a CPU pointer
Ensure: Returns an equivalent GPU pointer
info $\leftarrow$ greatestLTE(allocInfoMap, ptr)
if info.refCount = 0 **then**
    if $\neg$info.isGlobal **then**
        info.devptr $\leftarrow$ cuMemAlloc(info.size)
    else
        info.devptr $\leftarrow$ cuModuleGetGlobal(info.name)
    cuMemcpyHtoD(info.devptr, info.base, info.size)
info.refCount $\leftarrow$ info.refCount + 1
return info.devptr + (ptr $-$ info.base)

Algorithm 2: Pseudo-code for `unmap`

Require: ptr is a CPU pointer
Ensure: Update ptr with GPU memory
info $\leftarrow$ greatestLTE(allocInfoMap, ptr)
if info.epoch $\neq$ globalEpoch $\wedge$ $\neg$info.isReadOnly **then**
    cuMemcpyDtoH(base, info.devptr, info.size)
info.epoch $\leftarrow$ globalEpoch

Each of the primary run-time library functions has an array variant. The array variants of the run-time library functions have the same semantics as their non-array counterparts but operate on doubly indirect pointers. The array mapping function translates each CPU memory pointer in the original array into a GPU memory pointer in a new array. It then maps the new array to GPU memory. Using run-time library calls, Listing 2 can be rewritten as Listing 3.

3.3 Implementation

The `map`, `unmap`, and `release` functions provide the basic functionality of the run-time library. The array variations follow the same patterns as the scalar versions.

Algorithm 1 is the pseudo-code for the `map` function. Given a pointer to CPU memory, `map` returns the corresponding pointer to GPU memory. The `allocaInfoMap` contains information about the pointer's allocation unit. If the reference count of the allocation unit is non-zero, then the allocation unit is already on the GPU. When copying heap or stack allocation units to the GPU, `map` dynamically allocates GPU memory, but global variables must be copied into their associated named regions. The `map` function calls `cuModuleGetGlobal` with the global variable's name to get the variable's address in GPU memory. After increasing the reference count, the function returns the equivalent pointer to GPU memory.

The `map` function preserves aliasing relations in GPU memory, since multiple calls to `map` for the same allocation unit yield pointers to a single corresponding GPU allocation unit. Aliases are common in C and C++ code and alias analysis is undecidable. By handling pointer aliases in the run-time library, the compiler avoids static analysis, simplifying implementation and improving applicability.

Algorithm 3: Pseudo-code for `release`

```
Require: ptr is a CPU pointer
Ensure: Release GPU resources when no longer used
info ← greatestLTE(allocInfoMap, ptr)
info.refCount ← info.refCount − 1
if info.refCount = 0 ∧ ¬info.isGlobal then
    cuMemFree(info.devptr)
```

The pseudo-code for the `unmap` function is presented in Algorithm 2. Given a pointer to CPU memory, `unmap` updates CPU memory with the latest state of GPU memory. If the run-time library has not updated the allocation unit since the last GPU function call and the allocation unit is not in read only memory, `unmap` copies the GPU's version of the allocation unit to CPU memory. To determine if the CPU allocation unit is up-to-date, `unmap` maintains an epoch count which increases every time the program launches a GPU function. It is sufficient to update CPU memory from the GPU just once per epoch, since only GPU functions alter GPU memory.

Algorithm 3 is the pseudo-code for the `release` function. Given a pointer to CPU memory, `release` decrements the GPU allocation's reference count and frees the allocation if the reference count reaches zero. The `release` function does not free global variables when their reference count reaches zero. Just as in CPU codes, it is not legal to free a global variable.

4. Communication Management

CPU-GPU communication is a common source of errors for manual parallelization and limits the applicability of automatic parallelization. A CGCM compiler pass uses the run-time library to automatically manage CPU-GPU communications. For each GPU function spawn, the compiler determines which values to transfer to the GPU using a liveness analysis. When copying values to the GPU, the compiler must differentiate between integers and floating point values, pointers, and indirect pointers. The C and C++ type systems are fundamentally unreliable, so the compiler uses simple type-inference instead.

The communication management compiler pass starts with sequential CPU codes calling parallel GPU codes without any CPU-GPU communication. All global variables share a single common namespace with no distinction between GPU and CPU memory spaces. For each GPU function, the compiler creates a list of live-in values. A value is live-in if it is passed to the GPU function directly or is a global variable used by the GPU.

The C and C++ type systems are insufficient to determine which live-in values are pointers or to determine the indirection level of a pointer. The compiler ignores these types and instead infers type based on usage within the GPU function, ignoring usage in CPU code. If a value "flows" to the address operand of a load or store, potentially through additions, casts, sign extensions, or other operations, the compiler labels the value a pointer. Similarly, if the result of a load operation "flows" to another memory operation, the compiler labels the pointer operand of the load a double pointer. Since types flow through pointer arithmetic, the inference algorithm is field insensitive. Determining a value's type based on use allows the compiler to circumvent the problems of the C and C++ type systems. The compiler correctly determined unambiguous types for all of the live-in values to GPU functions in the 24 programs measured.

For each live-in pointer to each GPU function, the compiler transfers data to the GPU by inserting calls to `map` or `mapArray`. After the GPU function call, the compiler inserts a call for each live-out pointer to `unmap` or `unmapArray` to transfer data back to the CPU. Finally, for each live-in pointer, the compiler inserts a call to `release` or `releaseArray` to release GPU resources.

Algorithm 4: Pseudo-code for map promotion

```
forall region ∈ Functions ∪ Loops do
    forall candidate ∈ findCandidates(region) do
        if ¬pointsToChanges(candidate, region) then
            if ¬modOrRef(candidate, region) then
                copy(above(region), candidate.map)
                copy(below(region), candidate.unmap)
                copy(below(region), candidate.release)
                deleteAll(candidate.unmap)
```

Listing 4: Listing 3 after map promotion

```
    char *h_h_array[M] = {
      "What so proudly we hailed at the twilight's last gleaming,",
      ...
    };

□   __global__ void kernel(unsigned i, char **d_array);

    void foo(unsigned N) {
■     mapArray(h_h_array);
□     for(unsigned i = 0; i < N; ++i) {
■       char **d_d_array = mapArray(h_h_array);
■       kernel<<<30, 128>>>(i, d_d_array);
■       releaseArray(h_h_array);
      }
■     unmapArray(h_h_array);
■     releaseArray(h_h_array);
    }
```

□ Useful work ■ Communication ■ Kernel spawn

5. Optimizing CPU-GPU Communication

Optimizing CPU-GPU communication has a profound impact on program performance. The overall optimization goal is to avoid cyclic communication. Cyclic communication causes the CPU to wait for the GPU to transfer memory and the GPU to wait for the CPU to send more work. The map promotion compiler pass manipulates calls to the run-time library to remove cyclic communication patterns. After map promotion, programs transfer memory to the GPU, then spawn many GPU functions. For most of the program, Communication flows one way, from CPU to GPU. The results of GPU computations return to CPU memory only when absolutely necessary. The alloca promotion and glue kernels compiler passes improve the applicability of map promotion.

5.1 Map Promotion

The overall goal of map promotion is to hoist run-time library calls out of loop bodies and up the call graph. Algorithm 4 shows the pseudo-code for the map promotion algorithm.

First, the compiler scans the region for promotion candidates. A region is either a function or a loop body. Each promotion candidate captures all calls to the CGCM run-time library featuring the same pointer. Map promotion attempts to prove that these pointers point to the same allocation unit throughout the region, and that the allocation unit is not referenced or modified in the region. If successful, map promotion hoists the mapping operations

out of the target region. The specific implementation varies slightly depending on whether the region is a loop or a function.

For a loop, map promotion copies `map` calls before the loop, moves `unmap` after the loop, and copies `release` calls after the loop. Map promotion copies the `map` calls rather than moving them since these calls provide CPU to GPU pointer translation. Copying `release` calls preserves the balance of map and release operations. Inserting `map` calls before the loop may require copying some code from the loop body before the loop.

For a function, the compiler finds all the function's parents in the call graph and inserts the necessary calls before and after the call instructions in the parent functions. Some code from the original function may be copied to its parent in order to calculate the pointer earlier.

The compiler iterates to convergence on the map promotion optimization. In this way, map operations can gradually climb up the call graph. Recursive functions are not eligible for map promotion in the present implementation.

CGCM optimizes Listing 3 to Listing 4. Promoting the initial `mapArray` call above the loop causes the run-time library to transfer `h_h_array`'s allocation units to the GPU exactly once. The subsequent calls to `mapArray` inside the loop do not cause additional communication since the GPU version of the allocation units is already active. Moving the `unmapArray` call below the loop allows the run-time to avoid copying allocation units back to CPU memory each iteration. The optimized code avoids all GPU to CPU communication inside the loop. Spawning GPU functions from the CPU is the only remaining communication inside the loop. The final result is an acyclic communication pattern with information only flowing from CPU to GPU during the loop.

5.2 Alloca Promotion

Map promotion cannot hoist a local variable above its parent function. Alloca promotion hoists local allocation up the call graph to improve map promotions applicability. Alloca promotion preallocates local variables in their parents' stack frames, allowing the map operations to climb higher in the call graph. The alloca promotion pass uses similar logic to map promotion, potentially copying code from child to parent to calculate the size of the local variable earlier. Like map promotion, alloca promotion iterates to convergence.

5.3 Glue Kernels

Sometimes small CPU code regions between two GPU functions prevent map promotion. The performance of this code is inconsequential, but transforming it into a single-threaded GPU function obviates the need to copy the allocation units between GPU and CPU memories and allows the map operations to rise higher in the call graph. The glue kernel optimization detects small regions of code that prevent map promotion using alias analysis and lowers this code to the GPU.

Interrelationships between communication optimization passes imply a specific compilation schedule. Since alloca promotion and glue kernels improve the applicability of map promotion, the compiler schedules these passes before map promotion. The glue kernel pass can force some virtual registers into memory, creating new opportunities for alloca promotion. Therefore, the glue kernel optimization runs before alloca promotion, and map promotion runs last.

6. Evaluation

CGCM is applicable to all 101 DOALL loops found by a simple automatic DOALL parallelizer across a selection of 24 programs drawn from the PolyBench [17], Rodinia [7], StreamIt [23], and PARSEC [5] benchmark suites. The named region technique [12] and inspector-executor [4, 14, 22] were only applicable to 80. Without communication optimizations, many programs show limited speedup or even dramatic slowdown with automatic communication management. By optimizing communication, CGCM enables a whole program geomean speedup of 5.36x over best sequential CPU-only execution.

6.1 Experimental Platform

The performance baseline is an Intel Core 2 Quad clocked at 2.40 GHz with 4MB of L2 cache. The Core 2 Quad is also the host CPU for the GPU. All GPU parallelizations were executed on an NVIDIA GeForce GTX 480 video card, a CUDA 2.0 device with 1,536 MB of global memory and clocked at 1.40 GHz. The GTX 480 has 15 streaming multiprocessors with 32 CUDA cores each, for a total of 480 cores. The CUDA driver version is 3.2 release candidate 2.

The parallel GPU version is always compared with the original single threaded C or C++ implementation running on the CPU, even when multithreaded CPU implementations are available. All figures show whole program speedups, not kernel or loop speedups. All program codes are compiled without any alterations.

The sequential baseline compilations are performed by the clang compiler version 2.9 (trunk 118020) at optimization level three. The clang compiler produced SSE vectorized code for the sequential CPU-only compilation. The clang compiler at optimization level three does not use automatic parallelization techniques beyond simple vectorization.

The nvcc compiler release 3.2, V0.2.1221 compiled all CUDA C and C++ programs using optimization level three.

CGCM uses the same performance flags to manage and optimize communication for all programs. The optimizer runs the same passes with the same parameters in the same order for every program. A simple DOALL GPU parallelization system coupled with CGCM and an open source PTX backend [19] performed all automatic parallelizations.

6.2 Benchmark Suites

PolyBench [2, 9] is a suite composed of programs designed to evaluate implementations of the polyhedral model of DOALL parallelism in automatic parallelizing compilers. Prior work on automatic GPU parallelization reports impressive performance on kernel-type micro-benchmarks without communication optimization. The `jacobi-2d-imper`, `gemm`, and `seidel` programs have been popular targets for evaluating automatic GPU parallelization systems [3, 12]. The simple DOALL parallelizer found opportunities in all of the PolyBench programs, and CGCM managed communication for all GPU functions. Figure 4 shows performance results for the entire PolyBench suite.

The Rodinia suite consists of 12 programs with CPU and GPU implementations. The CPU implementations contain OpenMP pragmas, but CGCM and the DOALL parallelizer ignore them. The simple DOALL parallelizer found opportunities in six of the 12 Rodinia programs and from one selected program from the PARSEC and StreamIt benchmark suites. CGCM managed communications for all functions generated by the DOALL parallelizer. The StreamIt benchmark suite features pairs of applications written in C and the StreamIt parallel programming language. PARSEC consists of OpenMP parallelized programs for shared memory systems. The eight applications from Rodinia, StreamIt, and PARSEC are larger and more realistic than the PolyBench programs.

6.3 Results

Figure 4 shows whole program speedup over sequential CPU-only execution for inspector-executor, unoptimized CGCM, opti-

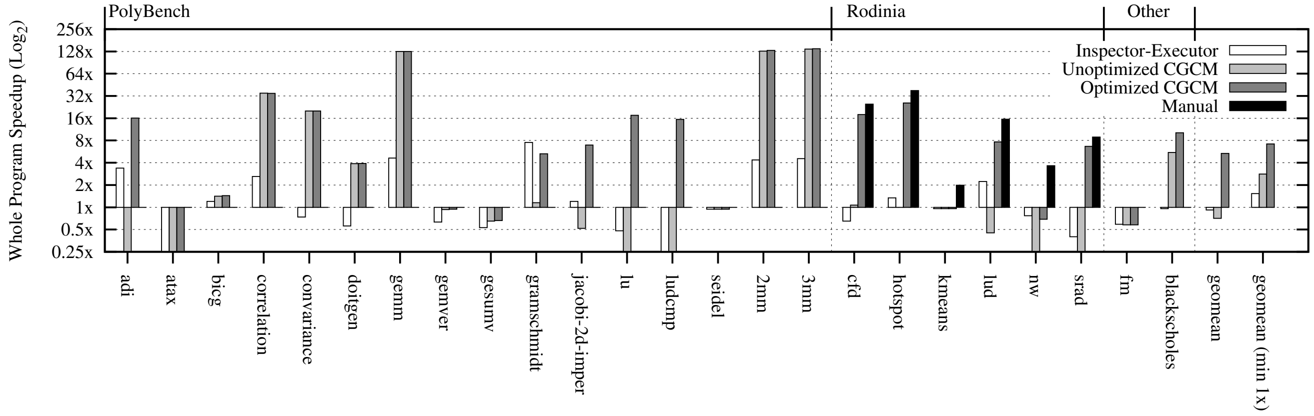

Figure 4. Whole program speedup of inspector-executor, unoptimized CGCM, optimized CGCM, and manual parallelizations over sequential CPU-only execution.

Program	Suite	Limiting Factor	Performance Study				Applicability Study				Manual Prior Work
			GPU		Communication		Total Kernels	CGCM	Inspector-Executor	Named Regions	
			UnOpti.	Opti.	UnOpti.	Opti.					
adi	PolyBench	GPU	0.02%	100.00%	99.98%	0.00%	7	7	7	7	
atax	PolyBench	Comm.	0.28%	0.28%	98.20	98.44%	3	3	3	3	
bicg	PolyBench	Comm.	4.36%	4.46%	72.38%	74.15%	2	2	2	2	
correlation	PolyBench	GPU	87.49%	87.39%	10.17%	10.12%	5	5	5	5	
covariance	PolyBench	GPU	77.12%	77.28%	18.61%	18.43%	4	4	4	4	
doitgen	PolyBench	GPU	87.48%	87.52%	11.29%	11.20%	3	3	3	3	
gemm	PolyBench	GPU	73.49%	73.76%	19.69%	19.49%	4	4	4	4	
gemver	PolyBench	Comm.	4.06%	4.10%	88.21%	89.36%	5	5	5	5	
gesummv	PolyBench	Comm.	6.17%	6.29%	86.17%	86.74%	2	2	2	2	
gramschmidt	PolyBench	Comm.	1.82%	8.37%	98.18%	90.91%	3	3	3	3	
jacobi-2d-imper	PolyBench	GPU	7.20%	95.97%	92.82%	3.32%	3	3	3	3	
seidel	PolyBench	Other	0.01%	0.01%	0.59%	0.59%	1	1	1	1	
lu	PolyBench	GPU	0.41%	88.05%	99.59%	7.02%	3	3	2	2	
ludcmp	PolyBench	GPU	1.23%	87.38%	98.10%	4.13%	5	5	3	3	
2mm	PolyBench	GPU	75.53%	77.25%	17.96%	18.25%	7	7	7	7	
3mm	PolyBench	GPU	78.75%	79.29%	17.86%	17.85%	10	10	10	10	
cfd	Rodinia	GPU	4.65%	77.96%	85.90%	0.16%	9	9	3	3	[7]
hotspot	Rodinia	GPU	2.78%	71.57%	92.60%	0.89%	2	2	1	1	[7]
kmeans	Rodinia	Other	0.65%	0.00%	10.84%	0.05%	2	2	2	2	[7]
lud	Rodinia	GPU	3.77%	63.57%	91.56%	0.39%	6	6	1	1	[7]
nw	Rodinia	Other	0.00%	2.44%	100.00%	24.19%	4	4	2	2	[7]
srad	Rodinia	Other	0.00%	27.08%	100.00%	6.20%	6	6	1	1	[7]
fm	StreamIt	Other	0.00%	0.00%	0.00%	0.00%	4	4	4	4	
blackscholes	PARSEC	Other	1.74%	3.23%	45.84%	0.96%	1	1	0	0	

Table 3. Summary of program characteristics including: program suite, limiting factor for performance, the contributions of GPU and communication time to total execution time as a percentage, the number of applicable kernels for the CGCM, Inspector-Executor, and Named Region management techniques, and a citation for prior manual parallelizations.

mized CGCM, and a manual parallelization if one exists. The figure's y-axis starts at 0.25x, although some programs have lower speedups. Table 3 shows additional details for each program. The geomean whole program speedups over sequential CPU only execution across all 24 applications are 0.92x for inspector-executor, 0.71x for unoptimized CGCM, and 5.36x for optimized CGCM. Taking the greater of 1.0x or the performance of each application yields geomeans of 1.53x for inspector-executor, 2.81x for unoptimized CGCM, and 7.18x for optimized CGCM.

Before optimization, most programs show substantial slowdown. The `srad` program has a slowdown of 4,437x and `nw` has a slowdown of 1,126x. By contrast, `ludcmp`'s slowdown is only 4.54x. After optimization, most programs show performance improvements and none have worse performance. However, several fail to surpass the CPU-only sequential versions. For comparison, we simulate an idealized inspector-executor system. The inspector-executor system has an oracle for scheduling and transfers exactly one byte between CPU and GPU for each accessed allocation unit. A compiler creates the inspector from the original loop [4]. To measure performance ignoring applicability constraints, the inspector-executor simulation ignores its applicability guard. CGCM outperforms this idealized inspector-executor system. The disadvantages of sequential inspection and frequent synchronization were not overcome by transferring dramatically fewer bytes.

Figure 4 shows performance results for automatic parallelization coupled with automatic communication management. Across all 24 applications, communication optimizations never reduce performance. This is a surprising result since the glue kernel optimization has the potential to lower performance, and CGCM's implementation lacks a performance guard. Communication optimization improves the performance for five of the sixteen PolyBench programs and six of the eight other programs. For many PolyBench programs, the outermost loop executes on the GPU, so there are no loops left on the CPU for map promotion to target. Therefore, optimization only improve performance for six of the 16 PolyBench programs.

Table 3 shows the number of GPU kernels created by the DOALL parallelizer. For each DOALL candidate, CGCM automatically managed communication correctly without programmer intervention. Unlike CGCM, the parallelizer requires static alias analysis. In practice, CGCM is more applicable than the simple DOALL transformation pass.

The table also shows the applicability of named regions [12] and inspector-executor management systems [4, 14, 22]. Affine communication management [24] has the same applicability as named regions, but a different implementation. The named region and inspector-executor techniques require that each of the live-ins is a distinct named allocation unit. The named regions technique also requires induction-variable based array indexes. The named region and inspector-executor systems are applicable to 66 of 67 kernels in the PolyBench applications. However, they are applicable to only 14 of 34 kernels from the more complex non-PolyBench applications. Although inspector-executor and named region based techniques have different applicability guards, they both fail to transfer memory for the same set of kernels.

Table 3 shows the GPU execution and communication time as a percent of total execution time. The contributions of CPU execution and IO are not shown. This data indicates the performance limiting factor for each program: either GPU execution, communication, or some other factor (CPU or IO). GPU execution time dominates total execution time for 13 programs, ten from PolyBench and three from other applications. The simpler PolyBench programs are much more likely to be GPU performance bound than the other more complex programs. GPU-bound programs would benefit from more efficient parallelizations, perhaps using the polyhedral model. Communication limits the performance of five programs, all from PolyBench. The only application where inspector-executor outperforms CGCM, `gramschmidt`, falls in this category. Finally, six programs, one from PolyBench and five from elsewhere, are neither communication nor GPU performance bound. Improving the performance of these applications would require parallelizing more loops. Two of the applications that are neither GPU nor communication-bound, `srad` and `blackscholes`, outperform sequential CPU-only execution. These applications have reached the limit of Amdahl's Law for the current parallelization.

The manual Rodinia parallelizations involved complex algorithmic improvements. For example, in `hotspot` the authors replace the original grid-based simulation with a simulation based on the pyramid method. Surprisingly, the simple automatic parallelization coupled with CGCM is competitive with expert programmers using algorithmic transformations. Table 3 explains why. Programmers tend to optimize a program's hottest loops, but ignore the second and third tier loops which become important once the hottest loops scale to thousands of threads. Automatic GPU parallelizations substitutes quantity for quality, profiting from Amdahl's Law.

7. Related Work

Although there has been prior work on automatic parallelization and semi-automatic communication management for GPUs, these implementations have not addressed the problems of fully-automatic communication management and optimization.

CUDA-lite [25] translates low-performance, naïve CUDA functions into high performance code by coalescing and exploiting GPU shared memory. However, the programmer must insert transfers to the GPU manually.

"C-to-CUDA for Affine Programs" [3] and "A mapping path for GPGPU" [13] automatically transform programs similar to the PolyBench programs into high performance CUDA C using the polyhedral model. Like CUDA-lite, they require the programmer to manage memory.

"OpenMP to GPGPU" [12] proposes an automatic compiler for the source-to-source translation of OpenMP applications into CUDA C. Most programs do not have OpenMP annotations. Furthermore, these annotations are time consuming to add and not performance portable. Their system automatically transfers *named regions* between CPU and GPU using two passes. The first pass copies all named annotated regions to the GPU for each GPU function, and the second cleanup pass removes all the copies that are not live-in. The two passes acting together produce a communication pattern equivalent to unoptimized CGCM communication.

JCUDA [26] uses the Java type system to automatically transfer GPU function arguments between CPU and GPU memories. JCUDA requires an annotation indicating whether each parameter is live-in, live-out, or both. Java implements multidimensional arrays as arrays of references. JCUDA uses type information to flatten these arrays to Fortran-style multidimensional arrays but does not support recursive data-types.

The PGI Fortran and C compiler [24] features a mode for semi-automatic parallelization for GPUs. Users target loops manually with a special keyword. The PGI compiler can automatically transfer *named regions* declared with C99's `restrict` keyword to the GPU and back by determining the range of *affine* array indices. The `restrict` keyword marks a pointer as not aliasing with other pointers. By contrast, CGCM is tolerant of aliasing and does not require programmer annotations. The PGI compiler cannot parallelize loops containing general pointer arithmetic, while CGCM preserves the semantics of pointer arithmetic. Unlike CGCM, the PGI compiler does not automatically optimize communication across GPU function invocations. However, programmers can use an optional annotation to promote communication out of loops. Incorrectly using this annotation will cause the program to access stale or inconsistent data.

Inspector-executor systems [18, 21] create specialized *inspectors* to identify precise dependence information among loop iterations. Some inspector-executor systems achieve acyclic communication when dynamic dependence information is reusable. This condition is rare in practice. Salz et al. assume a program annotation to prevent unsound reuse [21]. Rauchwerger et al. dynamically

check relevant program state to determine if dependence information is reusable [18]. The dynamic check requires expensive sequential computation for each outermost loop iteration. If the check fails, the technique defaults to cyclic communication.

8. Conclusion

CGCM is the first fully automatic system for managing and optimizing CPU-GPU communication. CPU-GPU communication is a crucial problem for manual and automatic parallelizations. Manually transferring complex data-types between CPU and GPU memories is tedious and error-prone. Cyclic communication constrains the performance of automatic GPU parallelizations. By managing and optimizing CPU-GPU communication, CGCM eases manual GPU parallelizations and improves the performance and applicability of automatic GPU parallelizations.

CGCM has two parts, a run-time library and an optimizing compiler. The run-time library's semantics allow the compiler to manage and optimize CPU-GPU communication without programmer annotations or heroic static analysis. The compiler breaks cyclic communication patterns by transferring data to the GPU early in the program and retrieving it only when necessary. CGCM outperforms inspector-executor systems on 24 programs and enables a whole program geomean speedup of 5.36x over best sequential CPU-only execution.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We also thank Helge Rhodin for generously contributing his PTX backend. Additionally, we thank the anonymous reviewers for their insightful comments.

This material is based on work supported by National Science Foundation Grants 0964328 and 1047879, and by United States Air Force Contract FA8650-09-C-7918. James A. Jablin is supported by a Department of Energy Office of Science Graduate Fellowship (DOE SCGF).

References

[1] ISO/IEC 9899-1999 Programming Languages – C, Second Edition, 1999.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1991.

[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction (CC)*, 2010.

[4] A. Basumallik and R. Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23, 2004.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. 2009.

[8] D. M. Dang, C. Christara, and K. Jackson. GPU pricing of exotic cross-currency interest rate derivatives with a foreign exchange volatility skew model. *SSRN eLibrary*, 2010.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *International Journal of Parallel Programming (IJPP)*, 1992.

[10] D. R. Horn, M. Houston, and P. Hanrahan. Clawhmmer: A streaming HMMer-Search implementation. *Proceedings of the Conference on Supercomputing (SC)*, 2005.

[11] Khronos Group. *The OpenCL Specification*, September 2010.

[12] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the Fourteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[13] A. Leung, N. Vasilache, B. Meister, M. M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, pages 51–61, 2010.

[14] S.-J. Min and R. Eigenmann. Optimizing irregular shared-memory applications for clusters. In *Proceedings of the 22nd Annual International Conference on Supercomputing (SC)*. ACM, 2008.

[15] NVIDIA Corporation. *CUDA C Best Practices Guide 3.2*, 2010.

[16] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 3.0*, February 2010.

[17] L.-N. Pouchet. PolyBench: The Polyhedral Benchmark suite. http://www-roc.inria.fr/ pouchet/software/polybench/download.

[18] L. Rauchwerger, N. M. Amato, and D. A. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming (IJPP)*, 26:537–576, 1995.

[19] H. Rhodin. LLVM PTX Backend. http://sourceforge.net/projects/llvmptxbackend.

[20] 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 *Proceedings of the Thirteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[21] J. Saltz, R. Mirchandaney, and R. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40, 1991.

[22] S. D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of the Conference on Supercomputing (SC)*. IEEE Computer Society Press, 1994.

[23] StreamIt benchmarks. http://compiler.lcs.mit.edu/streamit.

[24] The Portland Group. PGI Fortran & C Accelator Programming Model. White Paper, 2010.

[25] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-m. W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In *Proceeding of the 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[26] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Springer-Verlag, 2009.