Automatic Parallelization for GPUs

Thomas B. Jablin

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

Adviser: Professor David I. August

April 2013

Abstract

GPUs are flexible parallel processors capable of accelerating real applications. To exploit them, programmers rewrite programs in new languages using intimate knowledge of the underlying hardware. This is a step backwards in abstraction and ease of use from sequential programming. When implementing sequential applications, programmers focus on high-level algorithmic concerns, allowing the compiler to target the peculiarities of specific hardware. Automatic parallelization can return ease of use and hardware abstraction to programmers. This dissertation presents techniques for automatically parallelizing ordinary sequential C codes for GPUs using DOALL and pipelined parallelization techniques. The key contributions include: the first automatic data management and communication optimization framework for GPUs and the first automatic pipeline parallelization system for GPUs. Combining these two contributions with an automatic DOALL parallelization yields the first fully automatic parallelizing compiler for GPUs.

Acknowledgments

First, I'd like to thank my advisor David I. August. His contributions to our work by way of vision, intellectual support, and technical advice have been absolutely essential to my success as a graduate student. I will always remember the valuable lessons David has taught me about computer science, conducting research, writing persuasively, and his general philosophy of life. I am grateful for the opportunities he has provided.

I would also like to thank my readers Doug Clark and Scott Mahlke and my non-readers Margaret Martonosi and David Walker.

I'd like to thank all the members of the Liberty Research group for their camaraderie and support through my years at Princeton. I would like thank Guilherme Ottoni for his mentorship in my early years in the Liberty Group. I have always strived to meet his high standards of experimental methodology and personal integrity. Prakash Prabhu has been a frequent collaborator, running experiments, giving feedback, and providing supportive insight as the occasion required. Prakash writes bug-free code at an amazing speed, and has provided valuable and interesting feedback on this thesis. Nick Johnson is a great software architect, and I have learned tremendously from him. Our period of intense collaboration on the loop-sensitive alias analysis framework was the happiest and most intellectually productive period in my grad school career. Feng Liu has been valuable coauthor and a great and loyal friend. He is the one of the hardest working grad students I know, and I strive to match his dedication.

I would also like to thank the staff of the Computer Science department. The technical staff of the department has been extremely helpful. I'm also thankful for Melissa Lawson's sage counsel in navigating the complexities of grad student life.

Bob Dondero, Maia Ginsburg, and Donna Gabai, lecturers in courses I TA'd, have profoundly influenced the way I teach and the way I think about learning.

The compiler passes in this thesis are built on top of the LLVM compiler. The LLVM community has been very generous with their time, answering questions on IRC and responding to my bug reports. I have been consistently impressed with the high quality of the LLVM compiler infrastructure.

I am fortunate to have been born into a large and happy family. Without my parents' emotional support and wise advice, I would never have completed graduate school. I'm especially grateful for my grandparents' numerous trips to Princeton. It was a tremendous comfort to spend time with them. Their experience and optimism helped keep my problems in perspective. I'd also like to thank my brothers and sister: Jamie, Michael, Aaron, and Katie. A special thank-you must be expressed to Jamie. He has been a valuable and selfless coauthor on all of my papers and is presently finishing a PhD in computer science himself. I appreciate the time Jamie took away from his own research to help me meet paper deadlines.

Contents

Chapter 1

Introduction

Today, even entry-level PCs are equipped with GPUs capable of hundreds of GFLOPS performance. Real applications, rewritten to take advantage of GPUs, regularly achieve speedups between 4 and 100× [17, 25, 59]. Unfortunately, most applications do not use GPUs. The dominant paradigm for GPU-driven parallelization is laborious, requiring intensive effort to manually write and optimize code for GPUs. These high costs prevent widespread use of GPU parallelism.

GPU programming is difficult because GPU compilers have not provided adequate abstractions, forcing programmers to understand many low-level hardware details. Production GPU compilers require that programmers specify trade-offs between register spilling and number of parallel contexts, manage GPUs' complex memory hierarchy, efficiently communicate between CPU and GPU, and identify parallel work. GPU programmers are overwhelmed by implementation details. Consequently, high-performance GPU programming requires intimate knowledge of the GPU hardware. Furthermore, the lack of abstraction inhibits performance portability. Differences between GPU families or between different architectural generations in the same family can require substantial code revision for best performance.

For sequential CPU programming, even compilers for low-level programming languages like C provide a robust abstraction. C compilers optimize programs through instruction scheduling, register allocation, and vectorization to achieve high performance on diverse CPU architectures without programmer oversight or control. Sequential programmers apply architecture-specific optimizations only as a last resort when performance is critical and other options are exhausted. Consequently, sequential programmers focus on the high-level algorithmic issues and ignore the underlying architecture.

The sequential programming model offer excellent abstractions, but sequential CPU performance is no longer improving. The number of transistors per dollar per unit area continues to increase, but increased transistor counts no longer provide increased single-threaded performance due to power and thermal considerations. Since computer architects could not increase sequential performance, they rapidly increased parallel resources. Presently, these parallel resources are unused or underused by general-purpose applications due to the difficulties of parallel software development. GPU architectures are an extreme example of an architecture optimized for parallel resources at the expense of sequential performance and programmability.

Ideally, new developments in compiler technologies would provide simplicity, performance portability, and abstraction to parallel architectures. When superscalar architectures created opportunities for instruction-level parallelism (ILP), advances in computer architecture and compiler design allowed naïve programmers to benefit from ILP performance using sequential programming languages without understanding the underlying hardware.

Analyzing the features that made ILP successful and recreating them in the GPU context could enable ubiquitous transparent GPU parallelism for general purpose applications. Two problems inhibit transparent automatic parallelism for GPUs but

are solved or mitigated for superscalar processors: memory consistency and cyclic dependences. Superscalar processors resolve these problems using strong memory consistency guarantees and pipelining, respectively. To guarantee strong memory consistency, unicore superscalar processors provide a single shared memory and insure memory operations commit in program order. To manage cyclic dependences, superscalar architectures use pipelining and value forwarding to allow execution to begin before all of an instruction's dependences are satisfied.

To improve the ease of GPU programming, this dissertation presents a system for fully-automatic parallelization for C and C++ codes for GPUs. The system consists of a compiler and a run-time system. The compiler generates pipeline parallelizations for GPUs and the run-time system provides software-only shared memory. The main contributions are: the first automatic data management and communication optimization framework for GPUs and the first automatic pipeline parallelization system for GPUs.

With these components the compiler can automatically parallelize programs with recursive data-structures and complex unpredictable dependences previously thought unsuitable for GPUs. The system's fully-automatic performance is evaluated on a variety of applications and compared against the results of expert manual GPU parallelizations, where they exist.

1.1 Background

GPUs originated in the early 1980s as special purpose hardware devices designed to render three-dimensional graphics [15]. Consequently, it is remarkable that the highest performance general-purpose processor in many computers is the GPU. Three trends caused the current state of GPU architectures: integration, programmability,

and parallelism. The original GPU implementation [15, 16] consisted of twelve fixed-function vector processor chips connected in a pipeline. Rapidly increasing transistor counts eventually allowed a single chip to implement all the functions necessary for three-dimensional rendering. At the same time, graphics programmers demanded increasingly flexible hardware. Over time, each component in the original graphics pipeline gained generality, but retained different programming interfaces and different limitations. Eventually, computer-architects combined each of the semi-general components into a single unified shader [35]. Unified shaders improved programmability and ease of use by providing a single general interface to programmers and simplifying hardware design. Computer architects increased GPU architectures' parallel resources since their target application, three-dimensional rendering, is embarrassingly parallel. As a result of these three trends, modern GPUs are highly-parallel general-purpose architectures. GPUs lack robust memory consistency guarantees, flexible virtual memory, and fine grained synchronization primitives because these features are not needed for three dimensional rendering.

Even before the development of unified shaders and full generality, programmers began to see the performance potential of GPUs. For certain massively parallel problems, programmers were able to achieve impressive performance gains [58, 66, 67]. However, programming GPUs required translating programs into graphical operations, executing the operations on the GPU, and then reinterpreting the results of the graphical operations in terms of the original problem. At that time, writing programs for the GPU required deep knowledge of graphics, GPU hardware, and algorithms. The Brook for GPUs language and compiler dramatically eased GPU programming by automatically compiling from a C-like language to graphics primitives [12]. Brook for GPUs dramatically increased the level of program abstraction,

allowing programmers to focus on parallelization and performance tuning instead of translating programs into graphics primitives.

In 2007, NVIDIA released CUDA [44], a language and compiler derived from Brook for GPUs. CUDA bypasses the graphics APIs to target GPU hardware directly. In 2008, the Khronos Group standardized OpenCL, a standardized language for GPU programming [31]. However, GPU programming remains more complex than CPU programming.

1.1.1 Data Management and Communication Optimization

Parallelizing code for GPUs is difficult because CPUs and GPUs have separate memories, and neither has access to the other's memory. In order to share data-structures between CPU and GPU, programs must communicate data between CPU and GPU memories. In this work, *managing* data means determining what data to communicate between CPU and GPU memories to achieve a consistent program state. For performance, communication should follow acyclic patterns, since cyclic copying between CPU and GPU memories requires frequent synchronization and places communication latency on the program's critical path. *Optimizing* communication means replacing naïve cyclic communication patterns with efficient acyclic ones.

Data management presents a major problem for GPU parallelizations. The code in Listing 1 copies an array of strings to and from GPU memory, allocating and freeing memory as necessary. Almost every line of code in the example involves communication, not useful computation. The example code manages data by copying data between CPU and GPU memories using `memcpy`-style functions provided by the CUDA API [45]. Low-level `memcpy`-style pointer manipulation is notoriously difficult for programmers. For real codes, the hazards of manually copying to the GPU include

Listing 1: Manual explicit CPU-GPU memory management

```
    char *h_h_array[M] = {
      "as a book where men May read strange matters",
      ...
    };

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

    void bar(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

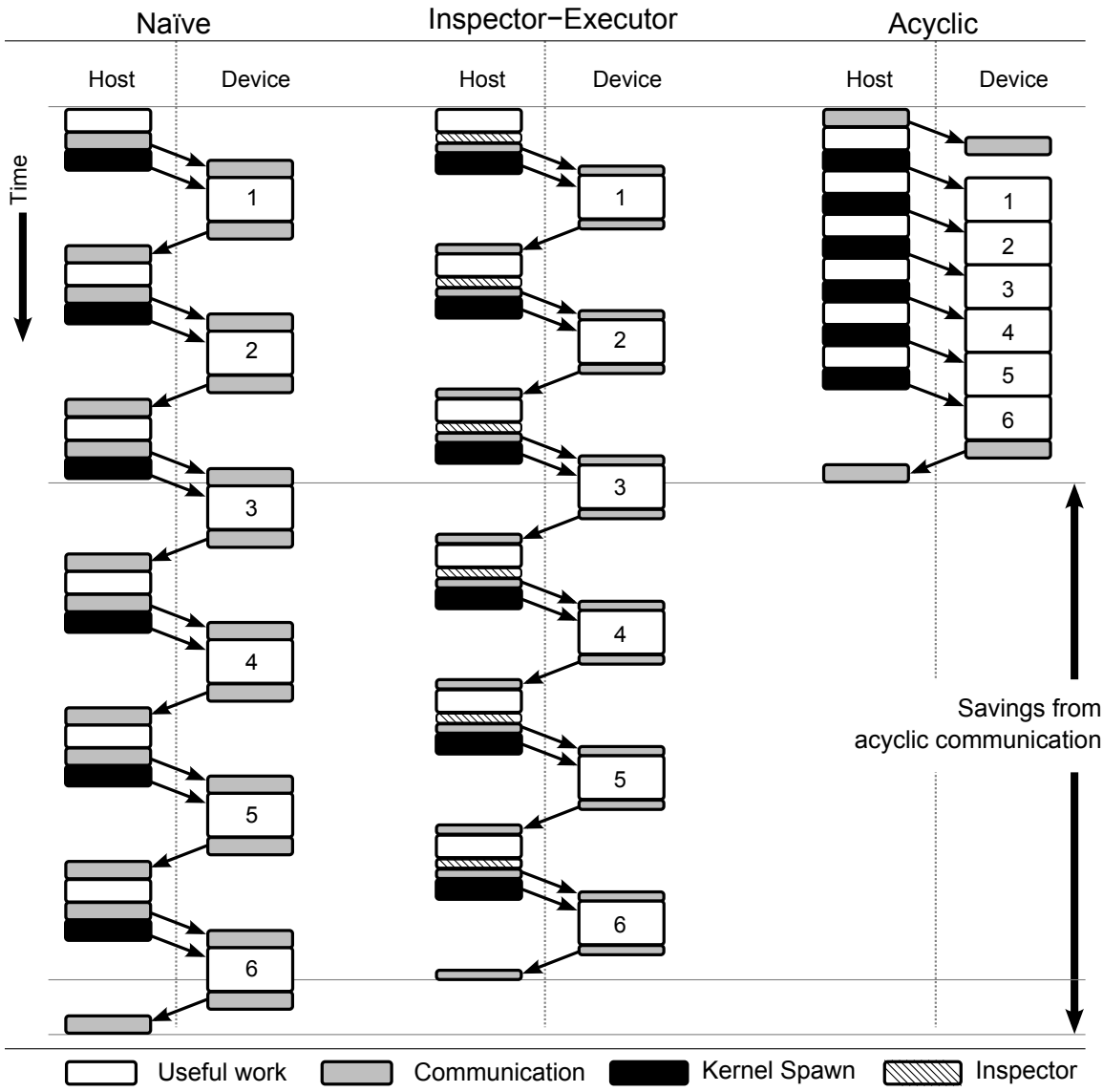


Figure 1.1: Execution schedules for naïve cyclic, inspector-executor, and acyclic communication patterns

subversive type casting, pointer aliasing, complex data-structures, dynamic memory allocation, and pointer arithmetic.

Acyclic CPU-GPU communication patterns are much more efficient than cyclic ones. Figure 1.1 shows an example program's execution schedule using cyclic and acyclic communication. For cyclic communication, communication latency is on the

Framework	Data Management	Comm. Opti.	Requires			Applicability				
			Annot.	TI	AA	CPU-GPU	Aliasing Pointers	Pointer Arithmetic	Max Indirection	Stored Pointers
JCUDA [73]	Annotat.	×	Yes	No	No	✓	×	×	∞	×
Named Regions [23, 36]	Annotat.	×	Yes	No	No	✓	×	×	1	×
Affine [64]	Annotat.	Annotat.	Yes	No	No	✓	×	×	1	×
IE [7, 40, 61]	Dynamic	×	Yes	No	No	×	×	×	1	×
CGCM [28]	Static	Static	No	Yes	Yes	✓	✓	✓	2	×
GMAC [20]	Annotat.	Dynamic	Yes	No	No	✓	✓	✓	∞	✓
DyManD [27]	Dynamic	Dynamic	No	No	No	✓	✓	✓	∞	✓

Table 1.1: Comparison between communication optimization and management systems (Annot: Annotation, TI: Type-Inference, AA: Alias Analysis)

program's critical path, and the program achieves limited parallelism between CPU and GPU execution. By contrast, the acyclic communication pattern keeps communication latency off the program's critical path and allows concurrent CPU and GPU execution.

In order to reduce errors and improve productivity, prior work proposes a variety of manual, semi-automatic, and fully-automatic techniques for data management and communication optimization. Table 1.1 summarizes the differences between various techniques. CGCM and DyManD will both be presented in this dissertation.

Data management techniques can be categorized as annotation-based, static, or dynamic. Annotation-based data management techniques require programmers to indicate when to transfer data-structures between CPU and GPU memories. These techniques hide the underlying complexity of the `memcpy`-style interface, but still require programmers to know what data should be copied between CPU and GPU memories and when to copy it.

By contrast, static and dynamic techniques automatically manage data without programmer effort. Using static or dynamic data management, programmers can write the example code in Listing 1 as Listing 2. Static data management techniques automatically transfer data between CPU and GPU memories, but rely on strong

Listing 2: Automatic implicit CPU-GPU memory management

```
    char *h_h_array[M] = {
      "as a book where men May read strange matters",
      ...
    };

□   __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

static analysis. When static analysis is too imprecise, the programmer must manage data. The quality of static analysis varies unpredictably between compilers. For example, the PGI compiler [64] can only manage communication for data-structures that are marked with the `restrict` keyword and accessed exclusively in an affine manner. If any either of these conditions are not met, the programmer must supplement PGI's static communication management with annotations or manual communication management.

Dynamic techniques manage data based on information determined at run-time. The information gathered at run-time is more accurate and complete than the results of static analysis. Consequently, dynamic communication management techniques can manage a wider variety of data-structures. For example, CGCM [28] can manage all pointer-free data-structures as well as data-structures with constant pointers to pointer-free data-structures.

To improve performance, most frameworks also offer communication optimization. Like data-management, the techniques can be classified as annotation-based, static, or dynamic. Just as with data management, dynamic communication is preferable.

Many data-management and communication optimization frameworks have restricted applicability for codes with pointers. The underlying problem is ensuring the CPU and GPU only dereference pointers to their respective memory-spaces. This problem is complicated for weakly typed languages like C and C++ in which it is undecidable whether a value is a pointer or not. CGCM partially solves this problem by using a sophisticated typing system to determine which values are pointers and to translate them when copying data between CPU and GPU memories. GMAC [20] and DyManD employ run-time libraries to allocate data-structures in CPU and GPU memories at matching addresses.

1.1.2 Pipeline Parallelism

Scientific codes already benefit from GPU's enormous parallel resources, but extending these results to general-purpose codes is challenging. GPUs are well suited loops with fine-grained independent iterations. Unfortunately, current GPU parallelization techniques are rarely applicable to non-scientific codes, due to frequent loop-carried dependences.

Pipeline parallelism extends the applicability of GPUs by exposing independent work units for code with loop-carried dependences. A pipeline consists of several stages distributed over multiple threads. Each stage executes in parallel with data passing from earlier to later stages through high-speed queues. Automatic pipeline parallelization techniques [46, 54, 55] construct pipelines from sequential loops by

partitioning instructions into different stages. Careful partitioning segregates dependent and independent operations. Stages with loop-carried dependences are called *sequential stages*; stages without loop-carried dependences are called *parallel stages*. Each iteration of a parallel stage can execute independently on different processors.

Manual pipeline parallelization techniques already target GPUs. Udupa et al. [69] use the StreamIt programing language [65] to manually parallelize the entire StreamIt benchmark suite for GPUs to achieve a speedup of over 5.5×. Many programmers consider explicit manual pipelining parallelization unnatural and therefore prefer automatic parallelization approaches.

Unfortunately, there are several difficulties in translating pipelining's success on multi-core and cluster architectures to CPU-GPU systems. All prior automatic pipelining techniques make assumptions about hardware that are incompatible with GPUs [26, 46, 54, 71]. Prior techniques assume a symmetric multiprocessing (SMP) architecture with shared memory and high-performance queues. CPU-GPU systems are asymmetric multiprocessing (AMP) architectures with a divided memory space and no hardware or software queue implementation. Each assumption presents an obstacle for pipelined parallelism on GPUs.

- Prior pipeline parallelization techniques assign instructions to stages based on the implicit assumptions that the performance of each thread is identical and the communication cost between stages is equal. AMP architectures invalidate this assumption. The naïve strategy of assigning sequential stages to fast CPU cores and parallel stages to numerous GPU cores ignores the significant performance impact of CPU-GPU communication.

- Prior code generation algorithms for pipelining synchronize to ensure that the parallelized program respects memory dependences. GPU systems do not support low-cost synchronization.

- Implementing pipeline parallelization on GPUs requires a queue between CPU and GPU threads and a queue between GPU threads. Some implementations assume specialized hardware queues [55]. Others use software queues based on x86-64's memory consistency model [53]. GPUs lack hardware queues or a robust memory consistency model necessary to implement software queues. Furthermore, there is no interface for communicating between the CPU and a running GPU thread.

Manual StreamIt parallelizations bypass these assumptions by adopting a very restrictive model of pipelining. Bringing the benefits of automatic pipeline parallelism to CPU-GPU architectures will require new automatic parallelization techniques.

The PBR technique discussed in this dissertation is the first automatic pipeline parallelization technique for GPUs. Compared to prior automatic pipelining implementations, PBR modifies both the partitioning algorithm and the code generation. The key observation behind PBR is that for pipeline parallelizations, there is a tradeoff between communication efficiency and computation efficiency. In the original automatic pipelining implementation, each non-branch instruction executes in exactly one thread. For example, when the result of a computation is used in multiple stages, it is computed once and communicated many times. However, if the computation is a pure function, each of the threads could execute the computation independently and thereby avoid communication. Executing the computation redundantly reduces communication overhead at the expense of computational efficiency. The original pipelining implementation assumes fast hardware queues and a relatively low core

count, so it never uses redundant computation to avoid communication. Modern GPUs have abundant parallel resources, but communication between cores on the GPU and between GPU and CPU is very expensive. Consequently, redundant computation is heavily favored.

PBR's partitioning algorithm partitions code into *redundant stages* in addition to parallel and sequential stages. Code in redundant stages can safely execute many times. By duplicating the redundant stages inside parallel and sequential stages, PBR dramatically reduces communication required to achieve efficient automatic pipelining on GPUs.

1.2 Dissertation Organization

This dissertation describes three techniques CGCM, DyManD, and PBR in chapter 2, 3, and 4 respectively. CGCM is the first fully automatic system for managing data and optimizing communication in CPU-GPU systems. CGCM eases manual GPU parallelizations and improves the applicability and performance of automatic GPU parallelizations. DyManD improves the applicability and performance of CGCM. Unlike CGCM, DyManD is able to manage recursive data-structures and is insensitive to the quality of alias analysis. Finally, PBR is the first fully automatic pipeline parallelization technique for GPUs. Chapter 5 examines the performance results for the CGCM, DyManD, and PBR and includes a case study demonstrating the parallelization of `KNNimpute`, an important bioinformatics application. Chapter 6 reviews related work. Chapter 7 assesses the impact of the work, discusses directions for future work, and concludes the dissertation.

Chapter 2

CPU-GPU Communication Management

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 [17, 25, 59]. 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 Data*. Generally, programmers manage data in CPU-GPU systems with `memcpy`-style functions. Manually managing data is tedious and error-prone. Aliasing pointers, variable sized arrays, jagged arrays, global pointers, and subversive type-casting make it difficult for programmers to copy the right data between CPU and

GPU memories. Unfortunately, not all data management is efficient; cyclic communication patterns are frequently orders of magnitude slower than acyclic patterns [42]. 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 chapter presents CPU-GPU Communication Manager (CGCM), the first fully automatic system for managing data and optimizing CPU-GPU communication. Automatically managing data and optimizing communication increases programmer efficiency and program correctness. It also improves the applicability and performance of automatic GPU parallelization.

CGCM manages data and optimizes communication using two parts, a run-time library and a set of compiler passes. To manage data, 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.

Framework	Data Management	Communication Optimization	Parallelism Extraction
CUDA [43]	Manual	Manual	Manual
OpenCL [31]	Manual	Manual	Manual
BrookGPU [12]	Manual	Manual	Manual
Baskararan et al. [6]	Manual	Manaul	Auto.
Leung et al. [37]	Manual	Manaul	Auto.
CUDA-lite [70]	Manual	Manaul	Auto.
JCUDA [73]	Annotation	×	Manual
GMAC [20]	Annotation	Auto.	Manual
Lee et al. [36]	Annotation	×	Auto.
PGI [64]	Annotation	Annotation	Annotation
CGCM [28]	Auto.	Auto.	Auto.
DyManD [27]	Auto.	Auto.	Auto.

Table 2.1: A taxonomy of related work showing data management, communication optimization, and parallelism extraction.

Table 2.1 shows a taxonomy of CPU-GPU data management techniques. No prior work fully automates CPU-GPU data management, but several semi-automatic techniques can manage data if programmers supply annotations [20, 36, 64, 73]. Some of these data management techniques are strongly coupled with automatic parallelization systems [36, 64]; others are not [20, 73]. Of the semi-automatic data management systems only GMAC optimizes CPU-GPU communications. Some prior automatic parallelization techniques require manual data management [6, 37, 70]. The earliest GPU parallelization systems feature manual parallelization and manual data management [12, 31, 43]. These systems remain the most popular. CGCM enables fully-automatic data management for manual and automatic parallelizations.

Data management is also a problem for distributed memory systems. Inspector-executor techniques automatically manage data for distributed memory systems [7, 40, 61] 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 27 programs, CGCM coupled with automatic parallelization shows a geomean whole program speedup of 4.95× over sequential CPU-only execution.

2.1 Motivation and Overview

CGCM avoids the limitations of prior work by employing a run-time support library and an optimizing compiler to automatically manage data 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 then optimizes communications to produce acyclic patterns. CGCM has two restrictions: 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 replace explicit CPU-GPU communication (Listing 1) with automatic communication (Listing 2). Replacing explicit CPU-GPU communication with CGCM yields dramatically shorter, simpler, clearer code and prevents several classes of programmer error.

Figure 2.1 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 2.2). The compiler inserts mapping functions to manage data (Section 2.3). Map promotion optimizes CPU-GPU communication by transferring data to the GPU early and keeping it there as long as possible

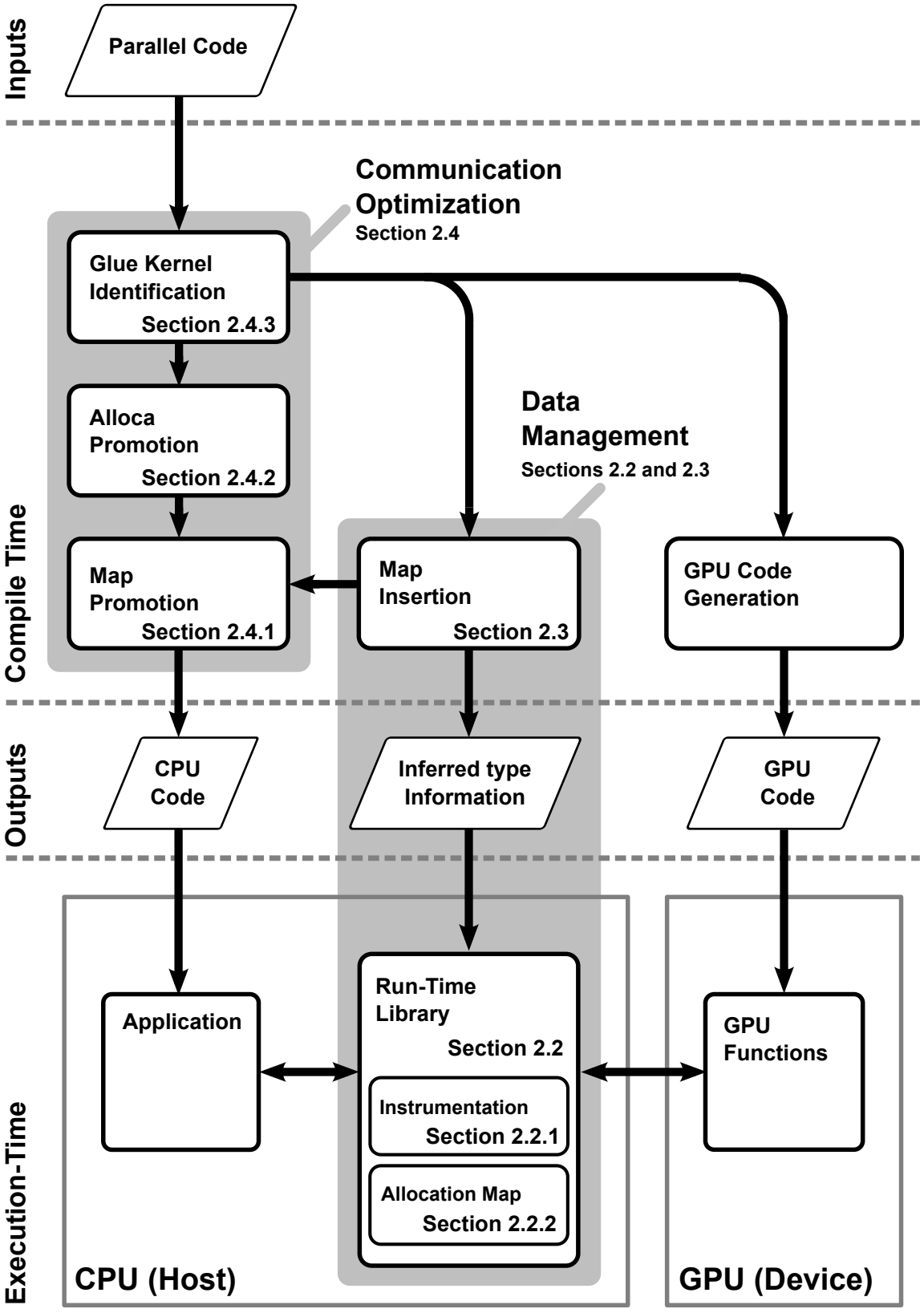


Figure 2.1: High-level overview of CGCM

(Section 2.4). Two enabling transformations, glue kernels and allocation promotion, improve map promotion's applicability.

2.2 Run-Time Library

The CGCM run-time library enables automatic CPU-GPU data management and communication optimization for programs with complex patterns of memory allocation and unreliable typing. To accomplish this, 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.

2.2.1 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 occupy 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.

Managing data at allocation unit granularity can correctly deal with pool-based custom allocators. Pool-based allocators `malloc` a large pool of memory and subsequently subdivide the pool into objects dynamically. Managing data at allocation unit granularity means the entire pool of objects will be transferred between CPU

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 2.2: CGCM's run-time library interface

and GPU memories as a single unit, even if only a single member is needed. There are two approaches to improving this situation. The first is to replace custom allocators with system default allocations. Berger et al. [9] show six out of eight applications with custom allocators perform better when the custom allocator is disabled. Alternatively, the custom allocator could be modified to call into the run-time library to report the size and location of allocation units within the pool.

2.2.2 CPU-GPU Mapping Semantics

Table 2.2 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

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

```
   char *h_h_array[M] = {
     "as a book where men May read strange matters",
     ...
   };

□  __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

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.

Algorithm 1: Pseudo-code for `map`

```
Require: ptr is a CPU pointer
Ensure: Returns an equivalent GPU pointer
info ← greatestLTE(allocInfoMap, ptr)
if info.refCount = 0 then
    if ¬info.isGlobal then
        info.devptr ← cuMemAlloc(info.size)
    else
        info.devptr ← cuModuleGetGlobal(info.name)
    cuMemcpyHtoD(info.devptr, info.base, info.size)
info.refCount ← info.refCount + 1
return info.devptr + (ptr − info.base)
```

- *Releasing* a CPU pointer decreases the corresponding GPU allocation unit's reference count, freeing it if necessary.

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, the compiler rewrites Listing 2 as Listing 3.

2.2.3 Design and 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

Algorithm 2: Pseudo-code for `unmap`

Require: ptr is a CPU pointer
Ensure: Update ptr with GPU memory
info ← greatestLTE(allocInfoMap, ptr)
if info.epoch ≠ globalEpoch ∧ ¬info.isReadOnly **then**
  cuMemcpyDtoH(base, info.devptr, info.size)
info.epoch ← globalEpoch

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.

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

Algorithm 3: Pseudo-code for `release`

Require: ptr is a CPU pointer
Ensure: Release GPU resources when no longer used
info $\leftarrow$ greatestLTE(allocInfoMap, ptr)
info.refCount $\leftarrow$ info.refCount $- 1$
if info.refCount $= 0 \wedge \neg$info.isGlobal **then**
  cuMemFree(info.devptr)

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.

2.3 Data Management

Data management 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 data. 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 data management compiler pass starts with sequential CPU codes calling parallel GPU codes without any data management. 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 if it 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 27 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.

2.4 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.

Algorithm 4: Pseudo-code for map promotion

```
forall the region ∈ Functions ∪ Loops do
    forall the 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)
```

The alloca promotion and glue kernels compiler passes improve the applicability of map promotion.

2.4.1 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

Listing 4: Listing 3 after map promotion

```
    char *h_h_array[M] = {
      "as a book where men May read strange matters",
      ...
    };

□   __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

`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.

2.4.2 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 promotion's 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 in order to calculate the size of the local variable earlier. Like map promotion, alloca promotion iterates to convergence.

2.4.3 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.

Chapter 3

Dynamically Manage Data

The applicability of CGCM is limited by its reliance on strong static analysis. To manage data, CGCM uses type-inference to statically determine the types of data-structures. Determining data-structures' types is necessary since CGCM handles pointer and non-pointer values differently. CGCM's static type-inference scheme characterizes data-structures as either arrays of pointers or arrays of non-pointers. Consequently, CGCM cannot automatically manage recursive data structures or data-structures with pointer and non-pointer types. To optimize communication, CGCM uses alias analysis to disprove cyclic dependences between code on the CPU and code on the GPU. Without cyclic dependences, cyclic communication is no longer necessary, so CGCM can safely optimize the program. CGCM requires static analysis (type-inference and alias analysis) because it manages data and optimizes communication at compile-time. The imprecision of static analysis limits CGCM's applicability and performance.

Dynamically Managed Data (DyManD) overcomes CGCM's limitations by combining dynamic analysis with CGCM-inspired efficient acyclic communication patterns. DyManD matches CGCM's performance without requiring strong alias analysis

and exceeds CGCM's applicability. DyManD creates the illusion of a shared CPU-GPU memory, allowing DyManD to manage complex and recursive data-structures which CGCM cannot. DyManD manages data automatically for both manual and automatic parallelizations.

DyManD's ability to manage and optimize recursive data-structures is crucial, since many general purpose and scientific applications use recursive data-structures like trees, linked lists, and graphs. The DOE, DARPA, and NSF believe next generation science requires graphs and other complex data-structures [22]. GPU programmers typically avoid recursive data-structures due to the difficulty of managing data and optimizing communication. By removing this difficulty, DyManD allows programmers to choose data-structures based on the problem domain.

DyManD's contribution over prior work is that it is the first fully-automatic CPU-GPU data management system to:

- support data-structures with pointer and non-pointer fields,
- support recursive data-structures,
- and be insensitive to alias analysis.

3.1 Motivation

To achieve performance on a CPU-GPU system, programs must manage data and optimize communication efficiently. Manual data management is difficult and error prone, and prior automatic data management is limited to simple data-structures. In this section, DyManD is motivated by comparison with two prior automatic techniques, inspector-executor [7, 40, 61] and CGCM [28]. Inspector-executor does not optimize communication, so its performance on GPUs is poor. CGCM requires strong alias analysis, but alias analysis is undecidable in theory and imprecise in practice.

Neither prior automatic technique manages complex recursive data-structures. DyManD efficiently manages complex data-structures without the limitations of type-inference or alias analysis.

CGCM is an automatic CPU-GPU data management and communication optimization system. To manage data, CGCM ensures that all live-in pointers to GPU functions are translated to equivalent GPU pointers. For correctness, CGCM copies data to the GPU at *allocation unit* granularity. An allocation unit comprises all bytes reachable from a pointer by well-defined pointer arithmetic. CGCM is only applicable to allocation units consisting entirely of pointer or non-pointer values. For non-pointer allocation units, CGCM copies the data to GPU memory without modification, but for pointer allocation units, CGCM iterates over the allocation unit, translating each CPU pointer to a GPU pointer. CGCM uses static type-inference to enforce this restriction due to C and C++'s subversive type casting.

CGCM automatically manages simple data-structures but has several important limitations due to its reliance on address translation and type-inference. To avoid translating GPU pointers back to CPU pointers, CGCM disallows storing pointers on the GPU. CGCM's simple type-inference is limited to scalar values, pointers to scalar values, and pointers to pointers to scalar values. It cannot infer the type of structures with pointers and non-pointers, higher-order pointers, or recursive data-structures. CGCM uses type-inference to differentiate pointer and non-pointer allocation units because it handles them differently. Even more sophisticated type-inference would fail in C and C++ due to frequent subversive type casting. Ideally, a data management system should be applicable to general purpose data-structures and arbitrary GPU functions without relying on imprecise static analysis.

In the area of manual GPU data management, prior work proposes several annotation-based systems [23, 36, 64, 73]. None of these systems handle pointer arithmetic,

aliasing inputs to GPU functions, or pointer indirection. The annotation-based techniques are limited to languages with strong type-systems [73], managing named regions [23, 36], or affine memory accesses [64].

GMAC [20] is a semi-automatic approach to data management and communication optimization. In GMAC, programmers annotate all heap allocations to indicate whether the allocated data is GPU-accessible. For manually annotated heap allocations, GMAC automatically generates efficient acyclic communication patterns. However, GMAC cannot manage stack allocations or global variables.

3.1.1 Prior Approaches to Communication Optimization

Acyclic CPU-GPU communication patterns are much more efficient than cyclic ones. For cyclic communication, communication latency is on the program's critical path, and the program achieves limited parallelism between CPU and GPU execution. By contrast, the acyclic communication pattern keeps communication latency off the program's critical path and allows parallel CPU and GPU execution.

To avoid cyclic communication, CGCM was introduced. Instead of copying data between CPU and GPU memories once per GPU function invocation, CGCM's communication optimization transfers data only once per program region. If a data-structure is not accessed by the CPU, CGCM copies it to GPU memory at the beginning of the code region and returns it to CPU memory at the end. CGCM uses static alias analysis to prove that the CPU will not access data-structures for the duration of a region.

Alias analysis quality strongly affects CGCM's ability to optimize CPU-GPU communication. Precise alias analysis is difficult to achieve in production compilers and remains an ongoing topic of research. CGCM's initial evaluation used a customized

Framework	Data Management	Comm. Opti.	Requires			Applicability				
			Annot.	TI	AA	CPU-GPU	Aliasing Pointers	Pointer Arithmetic	Max Indirection	Stored Pointers
JCUDA [73]	Annotat.	×	Yes	No	No	✓	×	×	∞	×
Named Regions [23, 36]	Annotat.	×	Yes	No	No	✓	×	×	1	×
Affine [64]	Annotat.	Annotat.	Yes	No	No	✓	×	×	1	×
IE [7, 40, 61]	Dynamic	×	Yes	No	No	×	×	×	1	×
CGCM [28]	Static	Static	No	Yes	Yes	✓	✓	✓	2	×
GMAC [20]	Annotat.	Dynamic	Yes	No	No	✓	✓	✓	∞	✓
DyManD [27]	Dynamic	Dynamic	No	No	No	✓	✓	✓	∞	✓

Table 3.1: Comparison between communication optimization and management systems (Annot: Annotation, TI: Type-Inference, AA: Alias Analysis)

alias analysis suite developed in tandem with CGCM. Consequently, the CGCM alias analysis gives precise results for the programs in the CGCM paper.

3.1.2 Relation of Prior Work to DyManD

Table 3.1 summarizes the differences between prior annotation-based manual data management systems, inspector-executor, CGCM, and DyManD. DyManD avoids the limitations of inspector-executor and CGCM by replacing static compile-time analysis with a dynamic run-time library. Static type-inference is unnecessary for DyManD since it does not translate CPU pointers to GPU pointers. By replacing standard allocation functions and modifying the GPU code generation, DyManD ensures that every allocation unit on the CPU has a corresponding allocation unit on the GPU at the same numerical address. Consequently, pointers copied to GPU memory point to equivalent allocation units in GPU memory without any translation. By avoiding pointer translation, DyManD removes the need for static type-inference.

DyManD dynamically optimizes communication, avoiding the need for static alias analysis. DyManD uses the page protection system to optimize communication by transferring data from GPU to CPU memory only when needed. To determine when a page is needed on the CPU, DyManD removes read and write privileges from the allocation units in CPU memory after copying them to GPU memory. If the CPU

accesses the pages later, the program will fault, and DyManD will transfer the affected allocation units back to CPU memory, mark the pages readable and writable, and continue execution. Cyclic communication is very infrequent in DyManD since data moves from GPU to CPU only if it is needed.

DyManD's communication optimization system is somewhat similar to software distributed shared memory (SDSM) [39] specialized for two nodes (the CPU and GPU). However, SDSMs rely on exception handling on *all* nodes to copy data on-demand. This scheme is unworkable on GPUs for two reasons. First, GPUs lack robust exception handling; the GPU equivalent of a segmentation fault kills all threads and puts GPU memory into an undefined state. Second, GPUs are presently unable to initiate copies from CPU memory. Consequently, DyManD conservatively copies data to GPU memory that *may* be accessed on the GPU, but copies data to CPU memory that *will* be accessed on the CPU. GMAC [20] also uses exception handling to optimize communication.

3.2 Design and Implementation

The DyManD data management and communication optimization system consists of three parts: a memory allocation system, a run-time library, and compiler passes. The memory allocation system ensures that addresses of equivalent allocation units on the CPU and GPU are equal, relieving the run-time system of the burden of translation. The run-time system dynamically manages data and optimizes communication. The compiler inserts calls to the memory allocation system and to the run-time library into the original program, and it generates DyManD compliant assembly code for the GPU. Table 3.2 summarizes DyManD's memory allocation and run-time library

Function prototype	Description
`blockAlloc(size)`	Allocate a block of memory at numerically equivalent addresses on the CPU and GPU.
`cuMemAlloc(size)`	CUDA driver API for allocating aligned memory on the GPU.
`map(ptr)`	Indicates `ptr` and any values it points to recursively may be used on the GPU.
`launch(gpuFunc)`	Launch a function on the GPU, copying data from CPU to GPU if necessary.
`dymandExceptionHandler(addr)`	Called when the CPU tries to access an allocation unit in GPU memory, copies the allocation unit to CPU memory.

Table 3.2: DyManD's run-time library and related functions from the CUDA driver API

interface. The remainder of the section will discuss the design and implementation of DyManD's memory allocator, run-time library, and compiler passes.

3.2.1 Memory Allocation

DyManD's memory allocation system keeps CPU and GPU versions of equivalent allocation units at numerically equivalent addresses in CPU and GPU memories. Using CPU addresses on the GPU without translation allows DyManD to avoid the applicability limitations of CGCM and inspector-executor. Address translation prevents prior work from managing data-structures with pointer and non-pointer fields and from managing data for GPU functions which store pointers.

The foundation of DyManD's memory allocation system is the `blockAlloc` function. The `blockAlloc` function (algorithm 5) allocates two blocks of memory, one on the CPU and a second on the GPU. The two blocks have the same size and address. Presently, there is no way to allocate memory at fixed GPU addresses. Therefore,

Algorithm 5: Pseudo-code for `blockAlloc`

Require: size is a multiple of page size
Ensure: Returns the address of equivalent allocation units in CPU and GPU memory

```
devptr ← cuMemAlloc(size)
addr ← devptr | MapMask
mmap(addr, size, MAP_FIXED)
return addr
```

`blockAlloc` first allocates GPU memory normally and then uses `mmap` to map a numerically equivalent address in CPU memory.

DyManD uses bitmasks to ensure that GPU allocations do not overlap with programs' static memory allocations. Static allocations start at low addresses so `blockAlloc` sets a high address bit to avoid overlapping static and dynamic allocations. A bitwise mask operation before each GPU memory access recovers the original GPU pointer. DyManD modifies code generation for the GPU to emit masking operations before load or store operations. When a pointer is compared or stored, the high bits are preserved. Consequently, storing and comparing pointers yields identical results on the CPU and GPU. From the programmer's perspective, addresses on the CPU and GPU are identical.

Allocation units come from dynamic allocations, from global variables, and from the stack. DyManD uses different techniques to manage allocation units depending on their source.

- For dynamic allocations, DyManD provides a customized version of `malloc`, `calloc`, and `realloc` based on `blockAlloc`. This implementation is similar to `mmap`-based `malloc` implementations [8, 41]. DyManD tracks all dynamic memory allocations.
- To manage global variables, a DyManD compiler pass replaces all global variables with equivalently sized dynamic allocations. To maintain program semantics,

DyManD allocates memory for global variables and copies any initial values before executing the `main` function.

- To manage stack allocations, a DyManD compiler pass replaces all escaping stack variables with dynamic allocations. The compiler pass ensures the dynamic allocations have the same scope and size as the original stack allocations. In general, escape analysis is undecidable, but in practice for stack variables, it is easily decidable.

3.2.2 Run-Time Library

DyManD's run-time library manages data and optimizes communication. For each allocation unit, the run-time maintains an ordered map from the base address to the size and state. The map can be used to determine if a pointer-sized value points within an allocation unit. The three states of an allocation unit are: CPU Exclusive (CPUEx), Shared, and GPU Exclusive (GPUEx). Allocation units in the Shared state may be accessed on the CPU but will become GPUEx on the next GPU function invocation. Figure 3.1 shows the state diagram for allocation units.

CPUEx to Shared via `map` All allocation units begin in the CPUEx state. In the CPUEx state, the CPU has exclusive access to the allocation unit. The `map` function (Algorithm 6) changes the state of CPUEx allocation units to Shared but does not copy the allocation unit to the GPU. The Shared state signifies that a specific allocation unit and any other allocation units it points to recursively should be copied to the GPU before invoking the next GPU function.

Shared to GPUEx via `launch` The run-time library's `launch` function (Algorithm 7) intercepts calls to GPU functions and copies data to the GPU. The `launch`

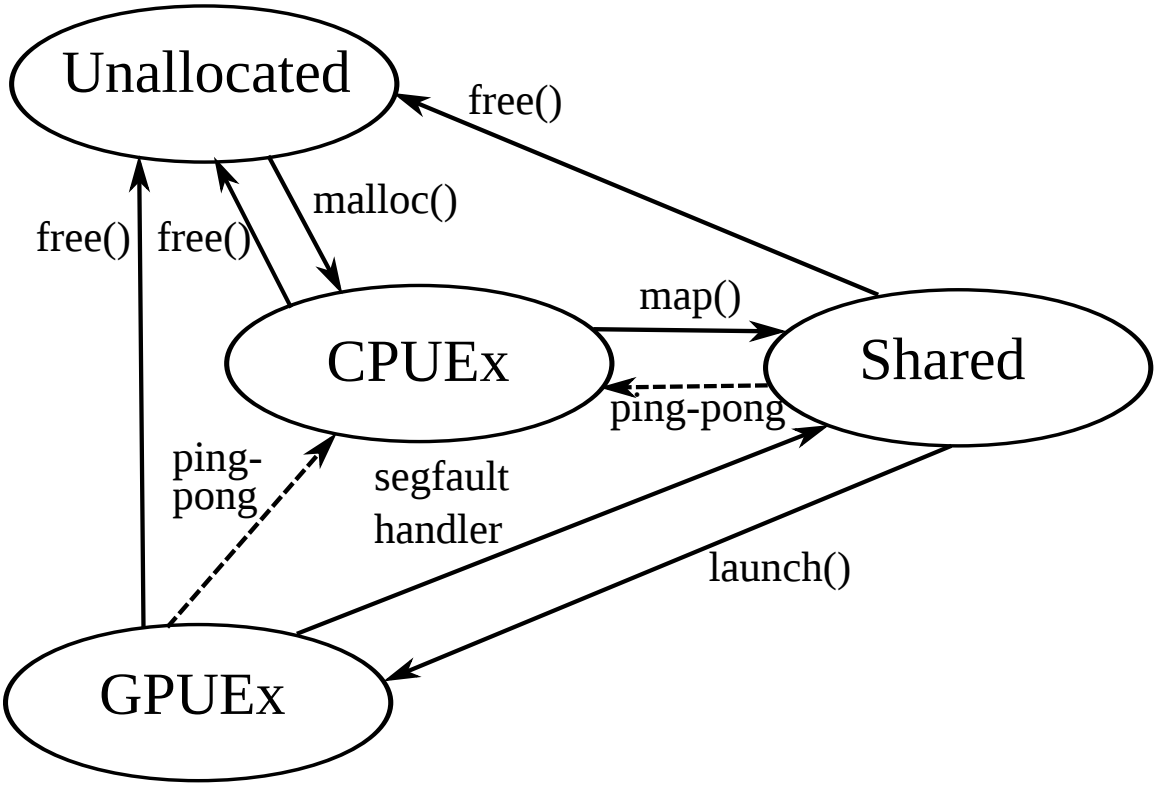


Figure 3.1: DyManD's state transition diagram for allocation units. The solid lines indicate transitions necessary for correctness. The dashed transitions improve performance heuristically, but are not necessary.

function selects a Shared allocation unit, copies it to GPU memory, and marks it GPUEx. After marking the allocation unit, `launch` scans the allocation unit for values that may be pointers. When a pointer is found, `launch` calls `map` with the new pointer and marks it Shared if it is not already. This is conservative, since non-pointer values that happen to point to valid addresses will cause unnecessary copying. Finally, `launch` calls `mprotect` to remove read and write permissions from the allocation unit's pages. Protecting pages prevents the CPU from accessing data in the GPUEx state. When no Shared allocation units remain, the GPU will have up-to-date versions of all allocation units it may access.

Algorithm 6: Pseudo-code for `map`

```
Require: ptr is a pointer sized value
Ensure: If ptr points to an allocation unit, mark all CPUEx allocation units
        sharing a page with ptr Shared
if ¬isPointer(ptr) then
    return
basePtr ← getBase(ptr)
forall the base ∈ getTransitiveClosure(basePtr, sharesPage) do
    if getState(base) = CPUEx then
        setState(base, Shared)
        push(sharedAllocs, base)
```

Algorithm 7: Pseudo-code for `launch`

```
Require: gpuFunc is a GPU function
Ensure: All Shared allocation units become GPU exclusive
while ¬empty(sharedAllocs) do
    base ← pop(sharedAllocs)
    size ← getSize(base)
    cuMemCopyHtoD(base, base, size)
    setState(base, GPUEx)
    foreach value ∈ loadAllValues(base, base + size) do
        if isPointer(value) ∧ getState(value) ≠ GPUEx then
            map(value)
    mprotect(base, size, PROT_NONE)
gpuFunc()
```

GPUEx to Shared via segfault handler The run-time library installs an exception handler (Algorithm 8) to detect accesses to pages in the GPUEx state. Touching any byte in a protected allocation unit triggers an exception. The exception handler copies the allocation unit back to CPU memory. For each allocation unit sharing a page with the faulting allocation unit, the exception handler restores read and write permissions, updates CPU memory, and marks the pages as Shared. DyManD preserves POSIX [49] semantics for access violations. When an access violation occurs

Algorithm 8: Pseudo-code for the exception handler which transfers allocation units back to the CPU on segmentation faults.

```
Require: ptr is the faulting address
Ensure: If ptr points to an allocation unit on the GPU, return it to the CPU
if ¬isPointer(ptr) ∨ getState(ptr) ≠ GPUEx then
    defaultSignalHandler()
    return
basePtr ← getBase(ptr)
forall the base ∈ getTransitiveClosure(basePtr, sharesPage) do
    size ← getSize(base)
    mprotect(base, size, PROT_READ | PROT_WRITE)
    cuMemcpyDtoH(base, base, size)
    setState(base, Shared)
```

to an address not protected by the run-time system, DyManD invokes the program's default exception handler.

The DyManD run-time system manages data and optimizes communication for complex recursive data-structures. The recursive nature of `launch` allows DyManD to successfully manage recursive data-structures with pointer and non-pointer fields. Additionally, the system naturally handles mapping the same allocation unit multiple times. If an allocation unit is live-in to a GPU function through multiple sources, it will only be transferred to the GPU once. By transferring data-structures from GPU to CPU memory only when necessary, the exception handler ensures a mostly acyclic communication pattern.

Shared and GPUEx to CPUEx via ping-pong heuristic DyManD has one additional state transition to improve performance by returning Shared data to the CPUEx state when it is no longer needed by the GPU. Sometimes a value enters the Shared state early in a program, and later the value is accessed on the CPU between two calls to GPU functions. In this case, the value will ping-pong between CPU and GPU memories even though it is never used on the GPU. To avoid this

problem, DyManD needs a way to restore Shared allocation units to the CPUEx state. It is unsound to mark *one* Shared allocation unit CPUEx since the GPU may still have a pointer to it. However, it is safe to transfer *all* allocation units off the GPU at once, restoring all allocation units to the CPUEx state. When ping-ponging is detected, the run-time library copies all GPUEx and Shared values back to the CPU, restores their read and write permissions, and marks them CPUEx. In practice, this heuristic resolves ping-ponging. If the run-time library detects that ping-ponging persists after intervening, it will not intervene again. This optimization improves the whole-program speedup of the `srad` program from 0.76× to 19.64×.

DyManD suffers from ping-ponging due to false sharing when allocation units frequently used on the GPU share a page with allocation units frequently used on the CPU. To avoid ping-ponging due to false sharing, the memory allocator uses three heuristics to arrange allocation units in memory. First, allocation units smaller than a page should never span a page boundary because this would force both pages to change state together. Second, allocation units larger than a page are always page aligned to prevent multiple large allocation units from transitioning together unnecessarily. Finally, allocation units are segregated by size since allocation units with a common size tend to transition from CPU to GPU memory as a group. For example, all the nodes of a binary tree will transition at once. Allocating them to the same page will not decrease performance.

3.2.3 Compiler Passes

The DyManD compiler's input is a program with CPU and GPU functions but without data management. For each GPU function, a DyManD compiler pass determines all live-in values. A value is live-in to a GPU function if it is passed to the GPU

function as an argument or if it is a global variable used by the GPU function or its callees. For each live-in value, the compiler pass inserts a call to DyManD's `map` function.

DyManD uses two compiler passes to create opportunities for dynamic communication optimization: alloca promotion and glue kernels. Both optimization techniques were initially used in CGCM [28].

Alloca promotion increases the scope of stack allocated values to improve optimization scope. Occasionally, programs will execute a loop in parallel on the GPU but allocate the loop's scratchpad arrays in CPU memory. Communication optimization fails since the stack allocated array falls out of scope between GPU function invocations. To remedy this situation, alloca promotion pre-allocates stack allocated arrays of predictable size, increasing their scope and allowing communication optimization.

Glue kernels prevent small sequential code regions from inducing cyclic communication. Sometimes a small sequential code region between two GPU functions uses an allocation unit that is on the GPU. The performance impact of the sequential code is trivial, but running it on the CPU induces cyclic communication which decreases performance. The glue kernel optimization transforms small sequential code regions into single threaded GPU functions. Surprisingly, the performance benefit of reduced cyclic communication outweighs the cost of single threaded execution on the GPU.

Chapter 4

Pipelining by Replication

The low applicability of GPUs and the high difficulty of writing efficient GPU code prevent widespread use of GPUs for parallel computation. Improving the applicability of GPU parallelizations can improve performance not only by accelerating the targeted loops, but also by avoiding CPU-GPU communication. Achieving high applicability for parallelizations on the GPU is critically important for performance because communication between CPU and GPU memories has high latency [20, 27, 28]. Even scientific applications contain irregular non-DOALL sections. If the non-DOALL sections access data-structures computed on the GPU, the communication latency of copying data between CPU and GPU memories will be on the program's critical path. Consequently, prior work has gone to great lengths to avoid cyclic communication on the program's critical path. For example, some implementations have improved performance by executing sequential code on the GPU to reduce communication [28].

Automatic pipeline parallelization techniques offer a compelling solution to increase both the applicability of GPUs and their ease of programming. Pipeline parallelism extends the applicability of GPUs by exposing independent work units for code with loop-carried dependences [69]. A pipeline consists of several stages. Each stage

executes in parallel with data passing unidirectionally from earlier to later stages through queues. Pipeline parallelization techniques [46, 54, 55] construct pipelines from sequential loops by partitioning instructions into different stages. Careful partitioning segregates dependent and independent operations. Stages with loop-carried dependences are called *sequential stages*, and stages without loop-carried dependences are called *parallel stages*. Each iteration of a parallel stage can execute independently on different processors.

Pipelining is a necessary but not sufficient component in an automatic parallelization framework. The primary benefit of pipelining is enabling scalable performance in combination with other automatic transformations [26]. Pipelining creates opportunities for DOALL [54] and LocalWrite [26] based parallelizations. Similarly, speculation [71], privatization [11], and parallel reductions enable pipelining. Consequently, pipelining is a stepping-stone towards automatic parallelization for general-purpose programs on GPUs, motivating further research on GPU-based speculation and privatization.

Implementing efficient pipelining for GPUs is challenging due to the limitations of GPU architectures. All prior automatic pipelining techniques [26, 46, 54, 71] assume a CPU-style execution environment with moderate core and thread counts, low-cost fine-grained synchronization primitives, simple and efficient queues for communication, and independent multi-threading. GPU architectures differ radically from this baseline. Specifically, GPUs have very high core and thread counts, high-cost coarse-grained synchronization primitives, no hardware or software queue support, and tightly coupled threads. The combination of these differences creates several challenges for a GPU pipelining implementation.

- Prior code generation algorithms for pipelining synchronize to ensure that the parallelized program respects memory dependences. GPU systems do not support low-cost or fine-grained synchronization.

- In the earliest CPU-based pipelines, threads communicate through specialized hardware queues [55]. GPUs lack hardware queues. Subsequent pipelining implementations use software queues [53]. However, software queues require either a robust memory consistency model or low-cost fined-grained synchronization instructions. GPUs presently lack these features.

- GPUs parallelizations typically use thousands of threads. The performance of software queues used in prior pipelining implementations falls precipitously at high thread counts [29].

- Prior pipelining algorithms assume all threads are completely independent. GPU architectures have best performance when groups of threads share a common control-flow path.

To resolve these issues, we present Pipelining by Replication (PBR), the first automatic CPU-GPU pipeline parallelization system. While current automatic parallelization techniques for GPUs only apply to embarrassingly parallel loops, PBR parallelizes loops with more complex control and data dependences. To demonstrate the correctness and performance of this pipelining implementation and its suitability as a framework for further research into automatic GPU parallelization, we present a detailed case study of the parallelization of the `em3d` program from the Olden benchmark suite [13]. After parallelization, `em3d` shows a speedup of 3.5× over best sequential execution. Additionally, PBR is applicable to 124 loops across 39 programs

that were not previously automatically parallelizable for GPUs. This parallelization result is in addition to the 554 DOALLable loops in the same programs.

4.1 Motivation

GPUs provide massively parallel resources, but current GPU hardware is heavily optimized for DOALL-style parallelizations. GPUs' weak memory model and lack of light-weight synchronization primitives make it difficult to implement more complex parallelization strategies. Pipeline parallelization exposes DOALL opportunities hidden in apparently sequential codes. PBR adapts pipeline parallelization to GPUs, expanding the GPUs' applicability. PBR consists of two parts: partitioning and code generation. Partitioning divides code into parallel and sequential stages and determines which loops can be profitably parallelized. Code generation takes partitions as input and generates code for efficient GPU execution.

4.1.1 Prior Approaches to Pipelining

In order to understand PBR, it is useful to review the most related work, the DSWP-MTCG family of automatic pipeline parallelization algorithms. The algorithms consist of two parts, a partitioner (DSWP [55]) and a code generator (MTCG [46]). The partitioner divides each loop iteration into a series of stages with communication proceeding from earlier stages to later stages. Given a partition, MTCG generates correctly parallelized code by inserting queues to communicate values from earlier stages to later stages as necessary. PS-DSWP generalizes DSWP by generating stages that can run in parallel across several threads.

Manual pipeline parallelization techniques already target GPUs. Udupa et al. [69] use the StreamIt programming language to manually pipeline parallelize the StreamIt

benchmark suite for GPUs. However, many programmers consider explicit manual pipeline parallelization unnatural and therefore prefer automatic parallelization approaches. The manual StreamIt parallelizations adopt a very restrictive model of pipelining. Specifically, StreamIt requires that each stage enqueue a constant number of values, determined at compile-time, per iteration. Consequently, StreamIt is inapplicable to programs with even moderately complex control dependences. Bringing the benefits of automatic pipeline parallelism to CPU-GPU architectures requires a more flexible approach.

4.1.2 Communication and Partitioning

Efficient communication between different stages in a pipeline is a major difficulty for GPU pipelining. In prior pipelining implementations, different stages communicate through high-bandwidth queues implemented in either hardware or software, with one queue allocated per thread. GPUs lack hardware queues. High-bandwidth software queues require either fine-grained lightweight synchronization or a robust memory consistency model. Unfortunately, GPUs lack these features as well.

Even if efficient software queues were possible on GPUs, they would not scale to typical GPU thread counts. Software queues achieve high bandwidth by amortizing the cost of synchronization operations over numerous enqueue operations. Figure 4.1 plots the bandwidth of a software queue implementation [29] versus the total number of bytes transmitted. The sustained bandwidth declines as the total number of bytes transmitted declines because the expensive synchronization operations are amortized over fewer enqueue operations. As thread count increases the number of bytes transmitted per queue falls, since each thread executes fewer iterations. Typical GPU

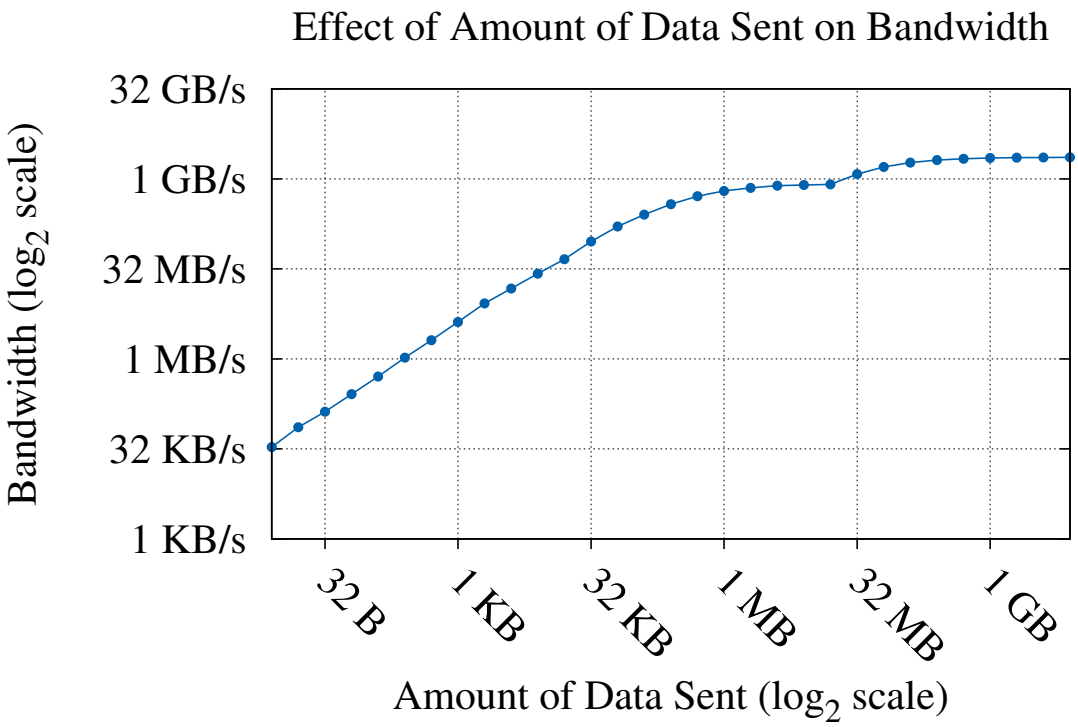


Figure 4.1: Average bandwidth of queues versus total bytes transmitted

parallelizations require thousands of threads, leading to very low numbers of bytes transmitted per queue and consequently very low queue bandwidths.

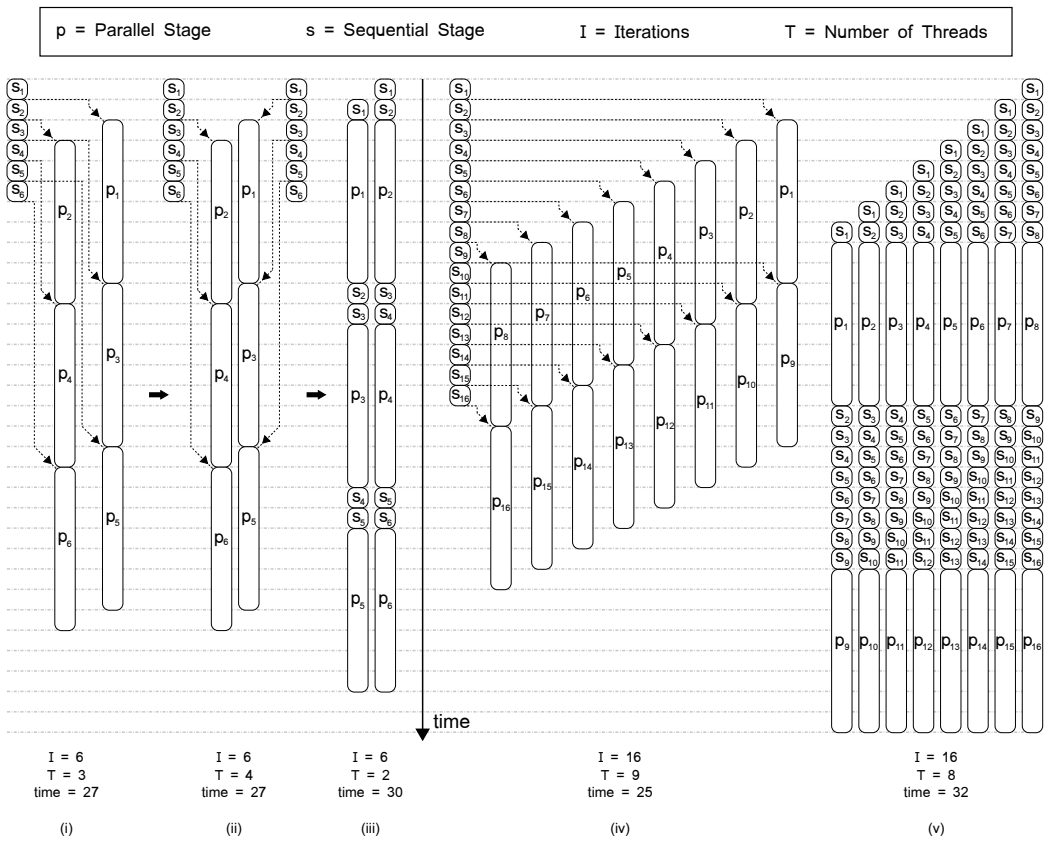
The key observation behind PBR is that for pipeline parallelizations, there is a trade-off between communication efficiency and computation efficiency. In the original automatic pipelining implementation [46], each non-branch instruction executes in exactly one thread. Consequently, a load instruction with uses in multiple stages may require considerable cross-thread communication. However, if the loaded value were constant, each of the threads could execute the load independently. This can be achieved by executing the load redundantly, reducing communication overhead at the expense of computational efficiency. Modern GPUs have abundant parallel resources

but communication between cores on the GPU is very expensive. Consequently, redundant computation is heavily favored for GPU architectures.

Figure 4.2a.i shows a timing diagram for a PS-DSWP parallelization consisting of a sequential stage followed by a parallel stage. Every iteration of the sequential stage executes in the same thread. The values computed in the sequential stage are communicated to one of two threads executing the parallel stage. In a PS-DSWP parallelization, each instruction executes in only one thread. In practice almost all sequential stages are side-effect free. Side-effect free code can execute multiple times without affecting the program semantics. For example, sequential stages are commonly used to iterate over a recursive data-structure, producing each element of the data-structure to a parallel stage for further processing.

Figure 4.2a.ii shows a timing diagram for an equivalent execution exploiting a side-effect free sequential stage. In the diagram, each iteration of the sequential stage executes in two different threads. Each thread executing a sequential stage communicates with a single thread executing a parallel stage. The first thread communicates the results of odd sequential stages to the first parallel-stage thread and the second thread communicates the results of even sequential stages to the second parallel-stage thread. Duplicating the sequential stage has the same performance as the PS-DSWP parallelization but requires an extra thread. Since each sequential stage communicates with only a single thread, merging each parallel stage with its private sequential stage would increase efficiency by avoiding communication.

In figure 4.2a.iii, the replicated sequential stages are merged with parallel stages to avoid communication; replicated computation replaces communication. For low thread counts, PS-DSWP-style parallelizations are more efficient. Increasing the thread count to eight (Figures 4.2a.iv and v) decreases the performance gap between communicating and replicating pipelines. As thread count approaches iteration count,



(a) Figures (i)-(iii) show the execution of the same 6 iterations of a loop, with s=1 and p=8: (i) PS-DSWP with 3 threads; (ii) PS-DSWP variation with 4 threads, where the extra thread is used for replication of the sequential stage so that even and odd iterations read from different threads; (iii) PBR example with 2 threads; Figures (iv)-(v) show the execution of the same 16 iterations of a loop: (iv) PS-DSWP with 9 threads; (v) PBR with 8 threads

$$\frac{sI + pI}{max(sI + p, \lceil \frac{pI}{T-1} \rceil + s)}$$

(b) PS-DSWP speedup equation over sequential

$$\frac{sI + pI}{sI + p + p\lfloor \frac{(I-1)}{T} \rfloor}$$

(c) PBR speedup equation over sequential

$$\lim_{T \to \infty} \frac{sI + p + p\lfloor \frac{(I-1)}{T} \rfloor}{max(sI + p, \lceil \frac{pI}{T-1} \rceil + s)} = \frac{sI + p}{sI + p} = 1$$

(d) PS-DSWP speedup equation over PBR as T$\to \infty$

$$\frac{10000 + 100 + \lceil \frac{100*(10000-1)}{3840} \rceil}{max(10000 + 100, \lceil \frac{100*10000}{3840-1} \rceil + 1)} = \frac{10361}{max(10100, 262)} = 1.03\times$$

(e) PS-DSWP speedup over PBR with I=10000, T=3840, s=1, p=100

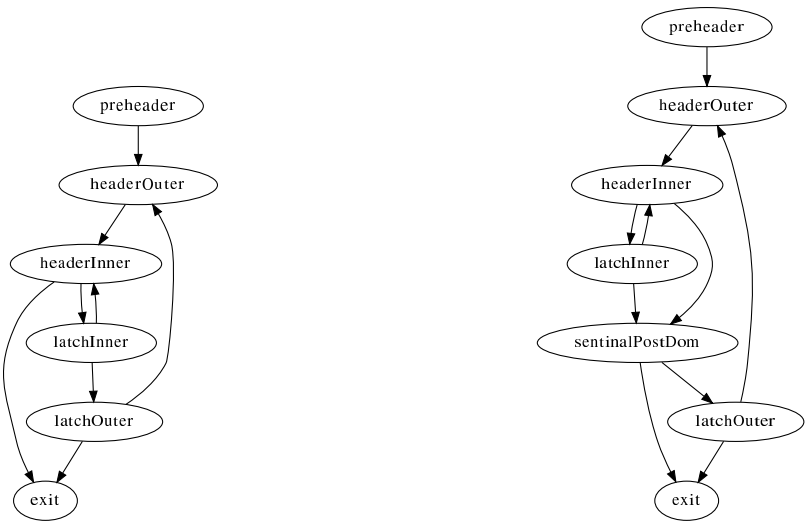
Figure 4.2: Comparison of timing diagram and speedup equations for PS-DSWP and PBR

the two pipelining techniques (Figures 4.2b and c) share a common performance limit (Figure 4.2d). Since GPU parallelizations typically use thousands of threads, the limit is approached very closely in practice (Figure 4.2e).

4.1.3 Code Generation

Generating efficient code for GPU architectures requires careful attention to control flow. In GPU architectures, groups of threads share a common program counter. GPUs can emulate independent control flow, but naïve code generation can yield disastrous performance consequences. *Control flow divergence* occurs when threads in a group do not behave identically at a branch. When a group of threads diverge, the GPU temporarily disables the threads that do not take the branch. The threads that take the branch execute until they reach the nearest post-dominator. Subsequently, the threads that took the branch are disabled, and the threads that did not take the branch execute until they reach the nearest post-dominator. Upon convergence, all threads continue execution. If the GPU encounters multiple divergent branches, the procedure repeats recursively until only a single thread remains and further divergence is impossible.

The performance impact of control flow divergence can be significant. On CUDA architectures a group of threads sharing a program counter is called a *warp*. A warp consists of thirty-two threads. A full warp of threads executes one instruction every other cycle. In the worst case of control flow divergence, only one thread in the warp executes an instruction every other cycle. Due to architectural limitations, threads cannot migrate from one warp to another, so the performance difference between ideal execution and worst case divergence is 32×.



(a) Original CFG

(b) Optimized for GPU Execution

Figure 4.3: Inner/Outer refers to the inner/outer-loop.
(a) Inner-loops whose immediate post-dominator is the function exit will have poor performance if control flow diverges between threads. (b) Insertion of a sentinel post-dominator node allows warp divergence in the inner loop to resynchronize after the inner loop invocation rather than at the function exit.

Unfortunately, control flow will never converge when the nearest post-dominator is the function exit. Figure 4.3a shows the CFG for a simple loop nest. Suppose each thread in a warp exits the inner loop on a different iteration. The nearest post-dominator of the inner-loop's backedge is the function's exit. Consequently, *any* control flow divergence on the inner-loop's backedge will cause the diverging thread to continue executing until the exit. The non-diverging threads must wait until the divergent thread exits to continue. However, if the threads converged after each inner loop iteration then the whole warp could execute the next iteration of the outer-loop in parallel. Figure 4.3b shows the same loop with the CFG transformed to create a non-exit post-dominator, which we call the *sentinel post-dominator*, for the inner-loop. This achieves thread convergence after each inner loop invocation.

Creating sentinel post-dominators is vital to PBR's performance. Typically, merged sequential stages have at least two distinct exits: one to execute a parallel stage and one to exit the function. The loop exit is always in an initial sequential stage because all instructions in a loop are control dependent on the branch that determines whether the function continues or exits.

4.1.4 Data Management

PBR uses the DyManD [27] data management and communication optimization framework. DyManD is an enabling technique for automatic GPU pipeline parallelism because it manages data and optimizes communication for programs with recursive data-structures, arbitrary casting, and unrestricted pointer arithmetic.

DyManD transfers data to GPU memory eagerly based on simple conservative static analysis and returns data to the CPU only when necessary using a dynamic demand-based scheme implemented using page protections. PBR uses an enhanced version of DyManD. Originally, DyManD waited for all executing GPU kernels to finish before exiting the program. However, since DyManD synchronizes when necessary to satisfy dependences between CPU and GPU, any values still being computed at program exit must be unused. By killing all current GPU tasks at program exit rather than synchronizing, enhanced DyManD enables a simple form of dynamic dead code elimination.

4.2 Design and Implementation

Figure 4.4 shows a diagram of the flow through the entire PBR system. The PBR system takes as input the original, unmodified, sequential C/C++ source code and produces a parallel GPU application. PBR makes use of a new technique, *malloc*

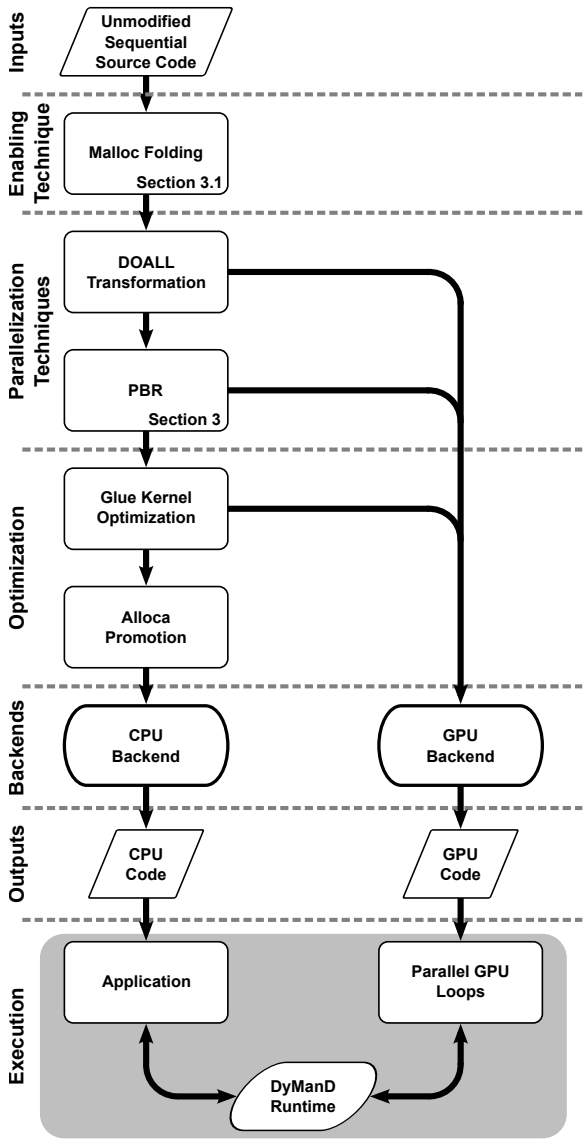


Figure 4.4: High level Overview of System

folding, to extend the applicability of automatic parallelization for GPUs to loops that dynamically allocate memory. GPU kernels cannot allocate memory. To overcome this, malloc folding calculates and preallocates memory required by a loop that will execute on the GPU.

Given DOALL transformation's high efficiency, all loops automatically identified as DOALLable are automatically DOALL parallelized. Next, PBR identifies and

Listing 9: Original version of hot loop in `make_neighbor` in `em3d` from the Olden Benchmark Suite.

```
for(cur_node = nodelist; cur_node;
    cur_node=cur_node->next) {

  cur_node->to_nodes =
    malloc(degree * (sizeof(node_t *)));

  for(j = 0; j < degree; ++j) {
    do {
      other_node = table[lrand48() % tablesz];
      for(k = 0; k < j; ++k)
        if(other_node == cur_node->to_nodes[k])
          break;
    } while(k < j);
    cur_node->to_nodes[j] = other_node;
    other_node->from_count++;
  }
}
```

transforms loops using pipeline parallelism. Subsequently, the glue kernel and alloca promotion optimizations from prior work [28] are applied to increase GPU execution efficiency. The resulting optimized program is finally run through CPU and GPU backends to lower the program to executable code. Finally, during program execution, the DyManD runtime [27] is used to automatically manage and optimize CPU-GPU communication.

4.2.1 Random Number Generation and Malloc Folding

Listing 9 shows the original sequential C version of `make_neighbors`. There are two apparent difficulties. First, the method calls `lrand48`, a deprecated POSIX [49] random number generator. Random number generators impede automatic parallelization due to cyclic dependences on the generator's internal state. By default, the PBR system replaces calls to `lrand48` and other well-known sequential random

number generators with independent parallel random number generators of equivalent strength. Prior work addresses the problem of random number generation by adding annotations to indicate either that calls to the generator commute [11, 51] or that the generator's internal state may be privatized.

The second difficulty is the call to `malloc`. Presently, the CUDA framework only allows small fixed quantity allocations of GPU global memory in kernels. By contrast, the DyManD framework allows memory allocated in CPU code to be used seamlessly in GPU kernels. The *malloc folding* transformation modifies the code to calculate the number of calls to malloc and the total number of bytes. These two numbers are used to call `mallocPool`. The `mallocPool` function allocates a region of memory that will not be freed until `free` has been called on pointers to the region once for each call to `malloc` in the original sequential code. In this way, the malloc folding transformation preserves semantics of the memory allocations with respect to `free`. Finally, calls to `malloc` are replaced with equivalent sized allocations from the newly created memory pool. Executing `mallocPool` on the CPU avoids calls to `malloc` inside otherwise parallelizable loops.

To automatically perform malloc folding, the PBR system gathers the transitive data and control dependences of the call to `malloc`. If dependence analysis shows these instructions do not modify memory (except the `malloc` itself) and are not affected by stores inside the original loop, they can be safely replicated. Replicating the transitive dependences and inserting them before the original loop creates a new loop that calls `malloc` the same number of times with the same values as the original loop. Malloc folding replaces calls to `malloc` in a skeleton version of the loop with bookkeeping code to calculate the total number of calls to malloc and the total number of bytes allocated. These values feed a call to `mallocPool` inserted into the original loop's preheader. Finally, calls to malloc in the original loop are replaced with code

Listing 10: Loop in `make_neighbor` after applying malloc folding.

```
int numAlloc = 0;
int numBytes = 0;
for(cur_node = nodelist; cur_node;
    cur_node=cur_node->next) {
  ++numAlloc;
  numBytes += degree * (sizeof(node_t *));
}

int8_t *malloc_pool =
  mallocPool(numAccess, numBytes);

for(cur_node = nodelist; cur_node;
    cur_node=cur_node->next) {
  cur_node->to_nodes = malloc_pool;
  malloc_pool += (degree * sizeof(node_t *));

  for(j = 0; j < degree; ++j) {
    do {
      other_node = table[lrand48() % tablesz];
      for(k = 0; k < j; ++k)
        if(other_node == cur_node->to_nodes[k])
          break;
    } while(k < j);
    cur_node->to_nodes[j] = other_node;
    other_node->from_count++;
  }
}
```

recording the current address of the pool and then incrementing it by the number of bytes "allocated." Listing 10 shows the `make_neighbors` function after the malloc folding transformation.

4.2.2 Partitioning

The goal of PBR's partitioner is to divide code in a target loop into a parallel stage and a replicable sequential stage. Algorithm 11 shows the pseudo-code for the partitioning algorithm. At a high level, the partitioner finds a candidate replicable sequential

Algorithm 11: Pseudo-code for partitioning algorithm

```
Input: loopInstSet: Set of all instructions in loop
Output: Set of instructions in sequential and parallel stages
seqInstSet = getAllInstWithCrossIterRegDep(loop)
foreach instruction i ∈ seqInstSet do
    footPrint = getMemoryFootPrint(i)
    if !isLoopInvariant(footPrint) then
        return false
parInstSet = loopInstSet - seqInstSet
foreach instruction i ∈ parInstSet do
    if hasCrossIterMemDep(i) then
        return false
return true
```

stage, checks that the candidate sequential stage is replicable, and then checks if the remaining instructions can form a parallel stage. To form the initial candidate stage, the partitioner identifies all instructions participating in cyclic register and control dependences. Verifying the candidate sequential stage requires checking that no instruction in the sequential stage writes to memory and that values loaded in the sequential stage are not modified by instructions in the parallel stage. Finally, the algorithm uses loop-sensitive dependence analysis to demonstrate that instructions in the parallel stage do not have cross-iteration memory dependences. Branches that control the loop's exits must appear in the sequential stage since they are necessarily cyclic. For the `make_neighbors` function, the sequential stage consists of only walking the linked list, and incrementing the `malloc_pool` variable. Invocations of the inner-loop can proceed in parallel.

Algorithm 12: Generic form of final GPU code generated by PBR.

```
foreach i = 0 → threadId do
    doOnlyRedundant()
while origLoopCondition do
    doParallelAndRedundant()
    foreach i = 0 → threadCount - 1 do
        doRedundant()
```

4.2.3 Code Generation

Code generation is relatively straightforward. The code-generation algorithm transforms the original sequential code for at most `threadId` iterations. After finishing the initial `threadId` iterations of the sequential stage, the program alternates between executing one iteration of a combined sequential-parallel stage and executing `threadNum` iterations of the sequential stage. All threads will exit in the same iteration since the partitioning ensures the loop exit is contained in the sequential stage and all threads execute every iteration of the sequential stage. Algorithm 12 shows the generic form of the generated code and Listing 13 shows the final parallel form of `make_neighbors`.

Listing 13: Loop in `make_neighbor` after applying malloc folding.

```
cur_node = nodelist;
for(i = 0; cur_node && i < threadId; ++i) {
  malloc_pool += (degree * sizeof(node_t *));
  cur_node=cur_node->next;
}

while(cur_node) {

  cur_node->to_nodes = malloc_pool;
  malloc_pool += (degree * sizeof(node_t *));

  for(j = 0; j < degree; ++j) {
    do {
      other_node = table[lrand48() % tablesz];
      for(k = 0; k < j; ++k)
        if(other_node == cur_node->to_nodes[k])
          break;
    } while(k < j);
    cur_node->to_nodes[j] = other_node;
    other_node->from_count++;
  }
  cur_node=cur_node->next;

  for(i = 0; cur_node && i < threadCount - 1;
      ++i) {
    malloc_pool += (degree * sizeof(node_t *));
    cur_node=cur_node->next;
  }
}
```

Chapter 5

Experimental Results

A single platform is used to evaluate CGCM, DyManD, and PBR. The performance baseline is an Intel Core 2 Quad clocked at 2.40 GHz with 4 MB 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 clocked at 1.40 GHz with 1,536 MB of global memory. The GTX 480 has 15 streaming multiprocessors with 32 CUDA cores each for a total of 480 cores. The CUDA driver version is 285.05.05. CGCM, DyManD, and PBR are all tuned for best performance on this reference platform.

The parallel GPU version of each program is always compared with the original single-threaded C or C++ implementation running on the CPU unless otherwise explicitly noted. All figures show whole program speedups, not kernel or loop speedups. For the automatic parallelizations, no programs are altered manually.

The sequential baseline compilations are performed by the clang compiler version 3.0 (trunk 139501) at optimization level three. The clang compiler produced SSE vectorized code for the sequential CPU-only compilation. The clang compiler does

not use automatic parallelization techniques beyond vectorization. The nvcc compiler release 4.0, V0.2.1221 compiled all CUDA C and CUDA C++ programs using optimization level three.

We use the same performance flags for all programs; no programs receive special compilation flags. 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 an open source PTX backend [57] performed all automatic parallelizations.

5.1 DyManD and CGCM Evaluation

DyManD is insensitive to alias analysis quality and more applicable than prior systems. To demonstrate DyManD's insensitivity to alias analysis, we compare the performance of DyManD and CGCM on a selection of 27 programs including all 24 programs tested in CGCM's original evaluation [28]. Since these programs are already applicable to CGCM, they cannot demonstrate DyManD's applicability improvements. Therefore, we manually parallelize three programs with recursive data-structures and compare the performance of manual data management and communication optimizations with DyManD's automatic data management.

To highlight CGCM's sensitivity to alias analysis quality, CGCM's performance is evaluated with no alias analysis, LLVM's production alias analysis [33], an alias analysis stack of three research-grade analyses [24, 34, 38], and perfect alias analysis performed manually.

The research-grade alias analysis stack consists of three analyses that are state-of-the-art in terms of both precision and scalability. These analyses are:

- Hardekopf and Lin's semi-sparse flow sensitive pointer analysis [24] which is inclusion-based, context insensitive, field sensitive, and flow sensitive.

- Lhoták and Chung's points-to analysis [38] which is context insensitive, semi-flow sensitive, and supports efficient strong updates.
- Lattner et al.'s pointer analysis [34] which is unification based, context sensitive, flow insensitive, and supports heap cloning.

5.1.1 Program Suites

We use different sets of programs to show DyManD's improved applicability and insensitivity to alias analysis relative to CGCM. To evaluate DyManD's performance on recursive data-structures, we compare DyManD with manual data management on manual parallelizations. We select three programs from the Olden benchmark suite [13] based on suitability for GPU parallelization and manually parallelized them using best practices. The Olden suite consists entirely of programs with recursive data-structures considered difficult to parallelize. The other programs in the suite were discarded because no suitable GPU parallelization could be found. Figure 5.1 shows the performance results for the selected Olden programs.

The alias analysis experiments consist of 27 programs drawn from the PolyBench [50], Rodinia [14], StreamIt [63], and PARSEC [10] benchmark suites. The 27 programs consist of all 24 programs in CGCM's original evaluation as well as three new programs selected from the same suites (`backprop`, `heartwall`, and `filterbank`). The PolyBench, Rodinia, and StreamIt suites have very few complex or recursive data-structures because the suites were designed for evaluating parallel compilers, architectures, and languages respectively.

PolyBench [5, 19] is a suite composed of 16 programs designed to evaluate implementations of the polyhedral model of DOALL parallelism in automatic parallelizing

compilers. Prior work demonstrates that kernel-type micro-benchmarks do not require communication optimization since they invoke a single hot loop once. The `jacobi-2d-imper`, `gemm`, and `seidel` programs have been popular targets for evaluating automatic GPU parallelization systems [6, 36]. Figure 5.2 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 the DOALL parallelizer ignores them. PARSEC consists of OpenMP parallelized programs for shared memory systems. The StreamIt suite features pairs of applications written in C and the StreamIt parallel programming language. Our simple DOALL parallelizer found opportunities in eight of the 12 Rodinia programs and from three selected programs from PARSEC and StreamIt suites. The 11 applications from Rodinia, StreamIt, and PARSEC are larger and more realistic than the PolyBench programs.

5.1.2 Applicability Results and Analysis

Figure 5.1 shows whole program speedup over sequential CPU-only execution for three manually parallelized Olden programs using manual data management or DyManD. Across all three benchmarks, manual data management did not confer a substantial performance advantage and was significantly more difficult to implement than automatic data management.

The `treeadd` program has the simplest data-structure, an unsorted binary tree implemented as a recursive data-structure. CGCM is inapplicable to `treeadd` because it contains a recursive data-structure and structures with pointer and non-pointer elements. In order to manage data, the programmer made a temporary copy of each node in the tree, replaced the copy's pointers with GPU pointers, transferred the copy

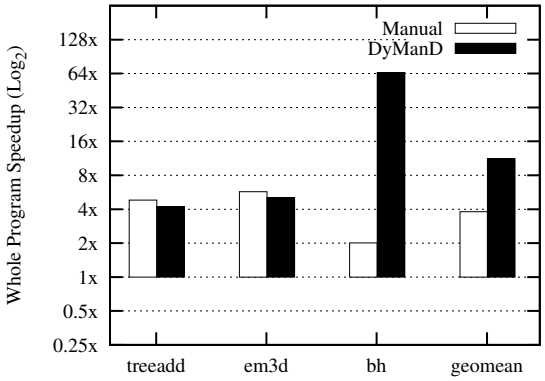


Figure 5.1: Whole program speedup over sequential CPU-only execution for manual parallelizations with manual and DyManD data management and communication optimization for programs with recursive data-structures.

to GPU memory, and freed the copy. The use of a temporary copy is unnecessary with DyManD because in DyManD, CPU and GPU pointers are equivalent. DyManD manages data by adding a call to `map` for the root of the binary tree before invoking the GPU function.

The `em3d` program uses two linked-lists to implement a many-to-many bipartite graph. Each node in the first linked-list contains an array of pointers to the second linked-list and vice-versa. Manual data management is somewhat more complicated than `treeadd` since identical pointers appear many times in the data-structure. To ensure each pointer is translated consistently, the programmer uses a map between CPU and GPU pointers. The manual data management performs a depth-first traversal starting from both roots of the bipartite graph. For each node in the graph, the programmer updates the map, uses the map to translate pointers in a temporary copy, transfers the copy to the GPU, and frees the copy. To manage data, DyManD inserts two calls to `map`, one for each root of the bipartite graph.

The `bh` program emulates Java-style object inheritance in C using careful data-structure layout and abundant casting. Although all subclasses are recursive data-structures, each subclass features different numbers and types of pointers at different structure offsets. In addition to the temporary copy and CPU to GPU pointer map used for `em3d`, the programmer must downcast abstract types to the appropriate subclasses. Manual data management requires the programmer to write custom code to translate each subclass. DyManD manages data by adding three calls to `map` before invoking the GPU function.

Surprisingly, in `bh` DyManD outperforms manual data management, even though both implementations transfer the same number of bytes in the same number of copies and use identical kernels. The performance difference is due to pointer translation. The programmer uses a temporary CPU copy to translate pointers, but DyManD does do not translate pointers. Ordinarily, the cost of the extra copy would be trivial, but the parallelized region is so much faster than the original sequential code that data management becomes a performance bottleneck.

5.1.3 Insensitivity Results and Analysis

Figure 5.2 shows whole program speedup over sequential CPU-only execution between DyManD and CGCM with no alias analysis, LLVM's production alias analysis, research-grade alias analysis, and perfect manual alias analysis. The figure's y-axis starts at 0.25× although some programs have lower speedups. Overall, DyManD's performance without alias analysis matches or exceeds CGCM's performance with production grade or research quality alias analysis.

For the PolyBench programs (`2mm` through `seidel`), the results indicate that the performance overhead of DyManD is comparable to CGCM even though DyManD has

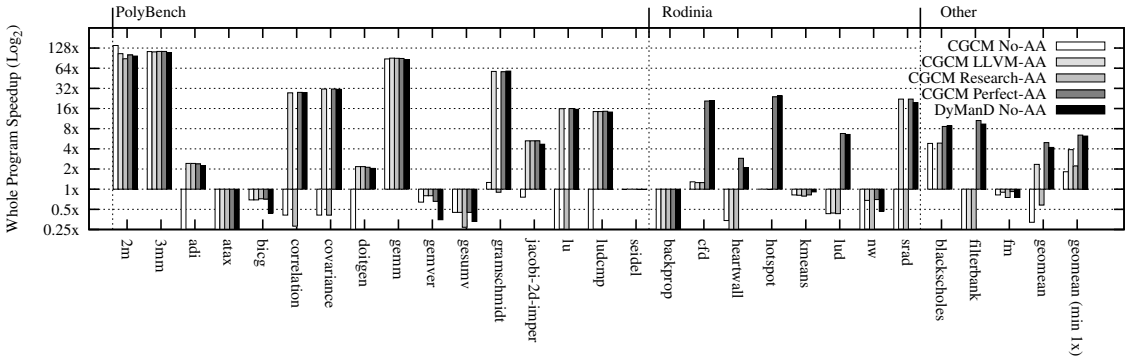


Figure 5.2: Whole program speedup over sequential CPU-only execution for CGCM with LLVM alias analysis, CGCM with custom alias analysis, and DyManD with no alias analysis.

a more complex run-time library. Differences in performance between DyManD and CGCM are usually due to the run-time overhead and not communication optimization because PolyBench has very few communication optimization opportunities. Most PolyBench programs consist of a single large GPU function that executes exactly once. Additionally, since the PolyBench programs do not dynamically allocate memory, very simple alias analysis can be precise. Consequently, the performance of DyManD and CGCM on the PolyBench suite is similar even with weak alias analysis.

The Rodinia, StreamIt, and PARSEC programs show more performance variability since these applications are more complex and require communication optimization for best performance. For these applications DyManD almost always performs better than CGCM with automatic alias analysis. Surprisingly, the research grade alias analysis system is not significantly superior to LLVM's production alias analysis system. LLVM's alias analysis was sufficient to optimize communication for `nw` and `srad`; the research alias analysis was not. The situation is reversed for `blackscholes` where LLVM's alias analysis is worse than the research grade implementation.

Across all the benchmarks, CGCM with perfect alias analysis outperforms DyManD very slightly. This reflects CGCM's lower run-time overhead. However, real

compilers do not have perfect alias analysis so DyManD performs better in practice. CGCM may be practical for languages that require less complex alias analysis such as FORTRAN or when programmer aliasing annotations are present. Nevertheless, DyManD's geomean overhead is 6.61% of whole program execution.

For programs where CGCM and DyManD are both slower than sequential execution, DyManD is almost always slower than CGCM. DyManD and CGCM's slowdowns are usually due to *necessary* cyclic communication between the CPU and GPU. DyManD and CGCM can only remove *unnecessary* cyclic communication. In CGCM, the program will copy data between CPU and GPU before and after every GPU function call. DyManD performs the same copies but must also frequently call into the operating system to protect and unprotect pages. Consequently, the performance penalty for cyclic communication is higher for DyManD than for CGCM.

5.2 PBR Evaluation

PBR is the first automatic GPU pipeline parallelization technique. Although pipeline parallelization typically requires enabling transformations such as reductions, speculation, and privatization to achieve scalable performance, pipelining alone is sufficient to achieve performance for `em3d`.

Automatic parallelization for `em3d` achieved a whole program speedup of 3.65×. The loop in the `make_neighbors` function accounts for 71.0% of the total execution time and had a speedup of 2.13×. The loop in `compute_nodes` accounts for 5.62% of the total execution time and had a speedup of 1.16×. Surprisingly, whole program speedup is greater than speedups of either parallelized loop. The reason is that results of many invocations of `compute_nodes` are never actually used on the CPU.

Benchmark	Program	Description	DOALL	PBR
-	otter	Theorem prover using first order logic	78	36
Olden	em3d	Simulates the propagation of electro-magnetic waves in a 3D object	0	6
Olden	bh	Solves the N-body problem using hierarchical methods	0	0
Olden	treeadd	Adds the values in a tree	0	0
Olden	bisort	Sorts by creating two disjoint bitonic sequences and then merging them	0	0
Olden	health	Simulates the Colombian health-care sytem	0	0
Olden	mst	Computes the minimum spanning tree of a graph	0	0
Olden	perimeter	Computes the perimeter of a set of quad-tree encoded raster images	0	0
Olden	power	Solves the Power System Optimization problem	0	0
Olden	tsp	Computes an estimate of teh best hamiltonian circuit for the traveling-salesman problem	0	0
Olden	voronoi	Computes the Voronoi Diagram of a set of points	0	0
ptrdist	ks	Kernighan-Schweikert graph partitioning tool	3	5
ptrdist	bc	GNU BC calculator	6	1
ptrdist	ft	Minimum spanning tree calculation	0	0
ptrdist	yacr2	Yet another channel routiner	4	3
rodinia	backprop	Machine-learning algorithm for a layered neural network	10	0
rodinia	bfs	Graph Traversal	0	0
rodinia	cfd	Unstructured grid finite volume solver for three-dimensional Euler equations	7	0
rodinia	heartwall	Tracks movement of mouse heart through ultrasound images	21	2
rodinia	hotspot	Thermal simulation for estimating processor temperature based floor plan and power measurements	2	0
rodinia	kmeans	K-means clustering algorithm	0	2
rodinia	leukocyte	Detects and tracks white-blood cells in medial imaging	128	31
rodinia	lud	Dense linear algebra solver	6	0
rodinia	nw	Global optimization method for DNA sequence alignment	3	0
rodinia	particlefilter	Statistical estimator of location given noisy location measurements	13	0
rodinia	srad	Diffusion algorithm based on partial differential equations used to remove speckles from images	6	1
rodinia	streamcluster	Solves online clustering problem	5	1
SPEC2000	179.art	Image Recognition / Neural Networks	21	2
SPEC2000	181.mcf	Combinatorial Optimization	1	2
SPEC2000	183.equake	Seismic Wave Propagation Simulation	3	0
SPEC2006	456.hmmer	Protein sequence analysis using profile hidden Markov models	183	30
stamp	kmeans	K-means clustering	3	0
StreamIt	filterbank	Creates a filter bank to perform multirate signal processing	7	0
StreamIt	audiobeam	Performs real-time beamforming on a microphone input array	17	2
StreamIt	bitonic	High performance sorting network	4	0
StreamIt	dct	Implements Discrete Cosine Transforms and Inverse Discrete Cosine Transforms	12	0
StreamIt	fft	Fast Fourier Transform kernel	3	0
StreamIt	fm	Software FM radio with equalizer	4	0
StreamIt	matmul-block	Blocked matrix multiply	4	0
-	**Total**		**554**	**124**

Table 5.1: Applicability of DOALL and Pipeline parallelism across 39 programs. The DOALL column shows the number of loops identified as DOALLable. The PBR column indicates the number of non-DOALLable loops parallelized by PBR.

Consequently, when the CPU reaches the end of the program, all pending work for the GPU is canceled and the program exits.

Table 5.1 shows the results of an applicability test across 39 benchmarks from a variety of benchmark suites. In total, 554 loops were identified as DOALL. Of the remaining loops in the programs, 124 were identified as being applicable to PBR.

Listing 14: The hottest loop in KNNimpute.

```
size_t gcnt = GenesIn.GetGenes();
for(unsigned i = 0; i < gcnt; ++i) {

  if(!( i % 100 ))
    printStatus();

  if((iOne = veciGenes[i]) == -1)
    continue;

  adOne = PCL.Get(iOne);
  for(unsigned j = i + 1; j < gcnt; ++j)
    if((iTwo = veciGenes[j]) != -1)
      Dat.Set(i, j, pMeasure->Measure( ... ));
}
```

5.3 KNNImpute Case Study

In order to demonstrate the strengths and limitations of the proposed system, we provide a case study showing how the KNNimpute program is automatically parallelized after only minor modifications by a programmer. KNNimpute is an important and influential tool in the field of bioinformatics with over a thousand citations in the last decade [52, 62, 68]. KNNimpute calculates values for missing data in gene expression microarray experiments. In bioinformatics, *impute* means to infer a missing experimental value based on empirical measurements of processes to which it contributes. KNNimpute uses weighted K-nearest neighbors to impute missing values for DNA microarray experiments.

Listing 14 shows pseudo-code for the hottest loop in KNNimpute, accounting for over 92% of total runtime. Both the outer and inner loops are DOALLable, but two difficulties prevent automatic parallelization. First, every hundredth iteration of the outer loop prints a status message to the console. Since the GPU cannot

perform IO, the code to print status message must run on the CPU. This prevents the compiler from parallelizing the outer loop for the GPU. The situation could be resolved through loop fission, since the contents of the status are independent of the loop's computations. However, the result would be printing all of the status messages before completing any of the computations. Rescheduling IO in this way is allowed by the C specification [1], but it clearly violates the programmer's intention of reassuring the user that the program is making progress. Alternatively, the program could parallelize only the inner loop, and allow the outer loop to run on the CPU, but the inner loop iterates only a few thousand times on reference inputs, so the overhead of invoking the GPU parallelization outweighs most of the benefit. Consequently, we removed the status messages in order to enable an outer-loop parallelization.

The second difficulty lies in the invocation of the `Message` function in the inner loop. `Message` is a virtual function which would be implemented using an indirect function call. Although, some recent GPUs have added support for indirect function calls, the GPU compiler backend used in this thesis cannot generate them [57]. This difficulty could be avoided by devirtualizing the function as in Java [21], but doing so requires dynamic recompilation to handle the case where new implementations of a virtual method are loaded dynamically. The programmer's intention in this function is to encourage reuse by allowing users to supply their own function for computing the pairwise correlation between two points. Implementations of `Measure` include functions based on Euclidean distance, Pearson correlation, Spearman rank correlation, and others. All these implementations are relatively simple pure functions, so the programmer intended `pMeasure` to be a functor. The original implementation is inefficient even for sequential compilation, since the `Measure` virtual function cannot be inlined. Inlining the `Measure` function would improve performance by avoiding call and return overhead in a tight loop and providing opportunities for specialization.

Listing 15: The hottest loop in KNNimpute after modifications.

```
size_t gcnt = GenesIn.GetGenes();
for(unsigned i = 0; i < gcnt; ++i) {

  if((iOne = veciGenes[i]) == -1)
    continue;

  adOne = PCL.Get(iOne);
  for(unsigned j = i + 1; j < gcnt; ++j)
    if((iTwo = veciGenes[j]) != -1)
      Dat.Set(i, j, Measure( ... ));
}
```

Instead of using virtual functions to implement functors, a programmer could use template-based metaprogramming to achieve greater performance without sacrificing flexibility [2]. To avoid indirect function calls, we modified the program by replacing `pMeasure->Measure` function with template parameter named `Measure`. Listing 15 shows the `KNNimpute`'s hottest loop after both modifications.

After performing these two simple modifications, the system is able to automatically parallelize `KNNimpute` for a whole program speedup of 1.5× over the best sequential compilation with the same modifications. `KNNImpute` is typically run once per experiment, and the results are saved and used in many analyses. The subsequent analyses are independent and can run in parallel on a cluster, so the `KNNImpute` program is a sequential bottleneck in an otherwise highly parallel computation. Consequently, parallelizing `KNNImpute` reduces the latency between running an experiment and examining the result and thereby accelerates the pace of science.

Chapter 6

Related Work

This chapter presents a summary of prior work related to CGCM, DyManD, and PBR. Since CGCM and DyManD are both data management and communication optimization techniques, their related work is described together and the related work for PBR follows in a separate subsection.

6.1 CGCM and DyManD Related Work

There are two techniques for managing data automatically: inspector-executor [7, 40, 61] and CGCM [28]. Inspector-executor systems manage data in clusters with distributed memory by inspecting program access patterns at run-time. Prior inspector-executor implementations are only applicable to simple array-based data structures. Some inspector-executor systems achieve acyclic communication when dependence information is reusable [56, 60]. This condition is rare in practice.

CGCM is the first fully-automatic data management and communication optimization system for GPUs. CGCM manages data using a combined run-time compile-time system. CGCM depends on compile-time type-inference to correctly transfer

data between CPU and GPU memories. The type-inference algorithm limits CGCM's applicability to simple array-based codes. Furthermore, CGCM depends on alias analysis for optimization, so the strength of alias analysis significantly affects overall performance.

DyManD does not require strong alias analysis for communication optimization and matches the performance of CGCM while achieving greater applicability. In contrast to CGCM, DyManD manages data and optimizes communication dynamically. For production compilers, DyManD is a more practical target than CGCM, since alias analysis is undecidable in theory and difficult to implement precisely and efficiently in practice.

Inspector-executor systems [56, 60] create specialized *inspectors* to identify precise dependence information among loop iterations. Salz et al. assume a program annotation to prevent unsound reuse [60]. Rauchwerger et al. dynamically check relevant program state to determine if dependence information is reusable [56]. The dynamic check requires expensive sequential computation for each outermost loop iteration. If the check fails, the technique defaults to cyclic communication.

CUDA 4.0's Unified Virtual Addressing (UVA) [45] also achieves a unified address space between CPU and GPU memories but has very different properties from DyManD. UVA allows programs to detect whether a value is a CPU pointer or a GPU pointer at run-time but does not facilitate data management or communication optimization. UVA distinguishes CPU pointers from GPU pointers by ensuring no valid address on the GPU is valid on the CPU and vice versa. By contrast, in DyManD, numerically equivalent addresses refer to equal size allocation units in CPU and GPU memories. From the perspective of the programmer, the DyManD run-time system keeps the contents of these allocation units identical.

Integrated GPUs, including CUDA and Fusion [3] devices, have the same data management and communication optimization problem as discrete devices. In most integrated GPUs, the CPU and GPU share the same physical memory. However, CPU-GPU communication still requires copying between memory allocated to the CPU and memory allocated to the GPU. Pinning memory renders it accessible to both CPU and GPU, but pinned memory has major limitations [3, 45]. Pinned memory is relatively scarce and requires programmers or compilers to determine which allocation units may be accessible on the GPU at allocation time. Additionally, pinned-memory cannot be swapped to disk so programs using pinned memory can adversely affect other programs running on the same computer.

Several semi-automatic systems exist that manage data using programmer annotations [20, 23, 36, 64, 73], but none handle recursive data structures. "OpenMP to GPGPU" [36] and hiCUDA [23] use annotations to automatically transfer arrays to GPU memory. JCUDA [73] uses the Java type system to transfer arrays to the GPU but requires the programmer to annotate whether parameters are live-in, live-out, or both. The PGI Fortran and C compiler [64] requires programmers to use the C99 `restrict` keyword to provide aliasing information. GMAC [20] requires annotations to manage specially marked heap allocations. Of all the semi-automatic techniques, only GMAC and the PGI accelerator optimize communication across GPU function invocations. GMAC's automatic communication optimization uses a page-protection based system similar to DyManD. For the PGI accelerator, optimizing communication requires additional programmer annotations.

Some automatic parallelization systems for GPUs require manual data management and communication optimization. CUDA-lite [70] 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" [6] and "A mapping path for GPGPU" [37] 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.

6.2 PBR Related Work

There are two main bodies of work related to PBR: pipeline parallelism techniques and automatic GPU parallelization techniques. No prior automatic technique adapts pipelined parallelism to GPUs.

DOPIPE [18, 47] is the first pipeline parallelization technique. Unlike later pipelining techniques, DOPIPE does not handle loops with control flow. DOPIPE generates parallel stages only when they correspond to a nested DOALLable loop in the original code.

DSWP generalizes DOPIPE by adding support for arbitrary control flow [55]. The original DSWP implementation is a manual technique limited to sequential stages and assumed hardware specialized hardware queues [55]. Subsequent implementations automate DSWP [46], add parallel stages [54], introduce speculation [71], and aid parallelization using programmer annotations [72]. By replacing the proposed hardware queues with a high performance software implementation, more recent publications demonstrate DSWP on commodity multi-core [53] and cluster [32] architectures. The earliest pipelining research describes pipeline parallelism as more general, but less scalable, than DOALL parallelism. Later works emphasize the pipelining's role as an enabling technique [26].

The StreamIt [65] programming language has built-in support for manual pipeline parallelization. Unlike DOPIPE and DSWP, StreamIt assumes shared memory, requiring programs to use queues for all communication between threads. The StreamIt compiler can compile and optimize parallel programs for GPUs [69].

All prior automatic GPU parallelization techniques only implement DOALL-style techniques. Several compilers implement semi-automatic GPU parallelizations, requiring programmers to add annotations to the targeted loops [36, 64]. "C-to-CUDA for Affine Programs" [6] and "A mapping path for GPGPU" [37] automatically transform program kernels into high performance CUDA C using the polyhedral model.

The idea of using the replicated stage to regenerate cross-iteration register dependences is similar to that of speculative prematerialization [74]. However, speculative prematerialization is only used to prematerialize cross-iteration register dependences which are defined in every iteration and therefore can be recomputed through pre-execution of at most one iteration. In contrast, PBR makes use of the massively parallel resources of the GPU to recompute cross-iteration register dependences for each loop iteration and is therefore not as constrained as prematerialization.

Chapter 7

Conclusion and Future Work

This dissertation presents an compiler-based automatic parallelization system for CPU-GPU architectures. The system is already capable of parallelizing many real applications. This chapter summarizes the impact of this system on research outside Princeton, highlights avenues for future work, and concludes the thesis.

7.1 Impact

Two publications by outside authors extend CGCM. Pai et al. propose X10CUDA+AMM, data management and communication optimization framework for the X10 parallel programming language [48]. X10CUDA+AMM extends CGCM by associating metadata with objects in order to track when the GPU's copy of the data becomes stale. The authors report a whole-program speedup 1.29× over their implementation of CGCM for X10. X10CUDA+AMM's implementation of metadata is not suitable for C, C++, or other weakly-typed languages.

PAR4ALL extends CGCM by increasing the granularity from allocation units to arrays [4]. Managing data at the array rather than the allocation unit granularity

reduces unnecessary communication, but requires that programmers abstain from subversive typecasting. The PAR4ALL framework achieves a geomean whole-program speedup of 4.43× over best sequential execution for the Rhodinia and Polybench suites.

7.2 Future Work

DyManD is sufficient to efficiently manage data and optimize communication for many programs, but not for all data-sets. In the current implementation, DyManD halts the program and prints an error message when GPU memory is exhausted. Typical GPUs have between 2 and 16GB of memory, but some scientific publications have larger working sets. Presently, GPUs do not support demand paging, but this may change in the near future. Even with GPU hardware that supports demand paging, new parallelization techniques may be required to minimize paging. One approach might be breaking large data-sets into chunks and then processing the data one chunk at a time. Automatically chunking data may require sophisticated data-flow analysis. Furthermore, determining the appropriate size for a chunk varies with the capabilities of the hardware and the behavior of the target program.

Automatic pipeline parallelization is an important stepping stone towards scalable automatic parallelization for general-purpose programs. Pipelining creates scalable performance opportunities in conjunction with other techniques such as speculation, privatization, and reductions. Implementing these techniques on GPUs may require extensions to hardware. For example, memory protections frequently enable speculation and privatization on CPUs, but current GPU architectures do not expose flexible interfaces for manipulating GPU memory. Furthermore, implementing sophisticated

low-overhead synchronization could enable high-speed queues and consequently enable a wider variety pipelining techniques on GPUs.

At present there is no way to communicate between the CPU and a running thread on the GPU. Creating hardware that supports this kind of pipelining would enable CPU-GPU pipelining where different CPUs cores execute sequential stages and the GPU executes parallel stages. Pipelining techniques are very insensitive to latency when communication is acyclic, so a CPU-GPU-CPU pipeline can be efficient as long as the initial and final CPU in the pipeline are different.

The applicability of the automatic DOALL and pipeline parallelizations presented in this thesis are limited by infrequent and spurious dependences. In multicore and cluster systems, speculative execution has also been used to enable automatic parallelization [32, 71, 53]. Unfortunately, these prior techniques rely on high-performance queues and flexible virtual memory systems. Neither are presently available on GPUs. Adapting shadow memory-based approaches to speculation for GPUs may be more fruitful [30].

7.3 Concluding Remarks

Despite GPUs enormous performance potential, they remain underutilized. Automatic compiler-based parallelism has the potential to radically reduce the difficulty of exploiting CPU-GPU architectures. This dissertation presents the CGCM, DyManD, and PBR techniques for automatically parallelizing ordinary sequential C and C++ codes for GPUs.

CGCM is the first fully automatic system for managing and optimizing CPU-GPU communication. 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 strong static analysis. The compiler breaks cyclic communication patterns by transferring data to the GPU early in the program and retrieving it only when necessary.

DyManD extends CGCM; by replacing static analysis with a dynamic run-time system, DyManD avoids the performance and applicability limitations of CGCM. CGCM's communication requires strong alias analysis and is very sensitive to analysis precision. By contrast, DyManD does not use alias analysis. DyManD consists of a run-time library and a set of compiler passes. The run-time library is responsible for managing data and optimizing communication while the compiler is responsible for code generation and creating optimization opportunities for the run-time. The run-time library manages data without requiring address translation since the DyManD memory allocator keeps equivalent allocation units at numerically equivalent addresses in CPU and GPU memories. The run-time library dynamically optimizes communication by using memory protections to return allocation units to the CPU only when necessary. DyManD outperforms CGCM equipped with production-quality and research grade alias analyses, achieving a whole program geomean speedup of 4.21x over best sequential execution versus geomean speedups of 2.35x and 1.28x, respectively, for CGCM.

PBR is the first fully automatic pipeline parallelization technique for GPUs. Automatic pipeline parallelization for GPUs increases GPU applicability for general-purpose codes. The key to enabling pipeline parallelization for the GPU is the use of redundant computation to avoid communication. Increasing computational redundancy decreases the overall amount of synchronization. This trade-off is favorable because the GPU has abundant parallel resources and limited synchronization primitives. Although automatic pipelining is only a stepping stone towards parallelizing

general purpose codes, it is already sufficient to parallelize `em3d` achieving a speedup of 3.65×.

This dissertation presents a fully-automatic parallelizing compiler supporting DOALL and pipeline parallelizations and using DyManD to manage data and optimize CPU-GPU communications. The compiler is able to achieve promising performance for many small but real applications.

Bibliography

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

[2] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[3] AMD. *AMD Accelerated Parallel Processing*, August 2011.

[4] M. Amini, F. Coelho, F. Irigoin, and R. Keryell. Static compilation analysis for host-accelerator communication optimization. In *The 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC'11)*, Sept. 2011.

[5] 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.

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

[7] 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.

[8] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35:117–128, November 2000.

[9] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[10] 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.

[11] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 69–84, 2007.

[12] 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.

[13] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 29–38, New York, NY, USA, 1995. ACM.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[15] J. H. Clark. Special Feature A VLSI Geometry Processor For Graphics. *Computer*, 13:59–68, 1980.

[16] J. H. Clark. The Geometry Engine: A VLSI Geometry System for Graphics. In *Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '82, pages 127–133, New York, NY, USA, 1982. ACM.

[17] 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.

[18] J. R. B. Davies. Parallel loop constructs for multiprocessors. Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, May 1981.

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

[20] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGPLAN Not.*, 45:347–358, March 2010.

[21] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, August 1996.

[22] Graph 500 specifications. http://graph500.org/specifications.html.

[23] T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM.

[24] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 226–238, New York, NY, USA, 2009. ACM.

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

[26] J. Huang, A. Raman, Y. Zhang, T. B. Jablin, T.-H. Hung, and D. I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization*, April 2010.

[27] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically Managed Data for CPU-GPU Architectures. In *Proceedings of the 2012 International Symposium on Code Generation and Optimization*, April 2012.

[28] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[29] T. B. Jablin, Y. Zhang, J. A. Jablin, J. Huang, H. Kim, and D. I. August. Liberty Queues for EPIC Architectures. In *Proceedings of the 8th Workshop on Explicitly Parallel Instruction Computing Techniques*, April 2010.

[30] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August. Speculative separation for privatization and reductions. *to appear Programming Language Design and Implementation (PLDI)*, June 2012.

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

[32] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[33] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[34] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[35] M. M. Leather and E. Demers. *Unified Shader*. United States Patent No. 7,796,133. ATI Technologies ULC, December 2003.

[36] 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.

[37] 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.

[38] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM.

[39] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, PODC '86, pages 229–239, New York, NY, USA, 1986. ACM.

[40] 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.

[41] O. Moerbeek. A new malloc (3) for openbsd. In *Proceedings of the 2009 European BSD Conference*, EuroBSDCon '09, 2009.

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

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

[44] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 3.1.1*, July 2010.

[45] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 4*, April 2011.

[46] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 105–118, 2005.

[47] D. A. Padua. *Multiprocessors: Discussion of some theoretical and practical problems*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, United States, November 1979.

[48] S. Pai and M. J. Thazhuthaveetil. "fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme". In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, Washington, DC, USA, 2012. IEEE Computer Society.

[49] POSIX.1-2008. The open group base specifications. (7), 2008.

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

[51] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[52] P. Prabhu, T. B. Jablin, A. Raman, Y. Zhang, J. Huang, H. Kim, N. P. Johnson, F. Liu, S. Ghosh, S. Beard, T. Oh, M. Zoufaly, D. Walker, and D. I. August. A survey of the practice of computational science. *Proceedings of the 24th ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

[53] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the*

Fifteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2010.

[54] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2008.

[55] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[56] 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.

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

[58] M. Rumpf and R. Strzodka. Using graphics cards for quantized fem computations. In *in IASTED Visualization, Imaging and Image Processing Conference*, pages 193–202, 2001.

[59] 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.

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

[61] 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.

[62] Google Scholar.
http://scholar.google.com/scholar?cites=3619060742417172543.

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

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

[65] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 12th International Conference on Compiler Construction*, 2002.

[66] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[67] C. Trendall and A. J. Stewart. General calculations using graphics hardware with applications to interactive caustics. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 287–298, London, UK, UK, 2000. Springer-Verlag.

[68] O. Troyanskaya, M. Cantor, G. Sherlock, P. Brown, T. Hastie, R. Tibshirani, D. Botstein, and R. B. Altman. Missing value estimation methods for dna microarrays. *Bioinformatics*, 17(6):520–525, 2001.

[69] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209, Washington, DC, USA, 2009. IEEE Computer Society.

[70] 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.

[71] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[72] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 389–400, 2010.

[73] 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.

[74] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.