Dynamically Managed Data for CPU-GPU Architectures

Thomas B. Jablin James A. Jablin[†] Prakash Prabhu Feng Liu David I. August

Princeton University, Princeton, New Jersey, USA
[†]Brown University, Providence, Rhode Island, USA

ABSTRACT

GPUs are flexible parallel processors capable of accelerating real applications. To exploit them, programmers must ensure a consistent program state between the CPU and GPU memories by managing data. Manually managing data is tedious and error-prone. In prior work on automatic CPU-GPU data management, alias analysis quality limits performance, and type-inference quality limits applicability. This paper presents Dynamically Managed Data (DyManD), the first automatic system to manage complex and recursive data-structures without static analyses. By replacing static analyses with a dynamic run-time system, DyManD overcomes the performance limitations of alias analysis and enables management for complex and recursive data-structures. DyManD-enabled GPU parallelization matches the performance of prior work equipped with perfectly precise alias analysis for 27 programs and demonstrates improved applicability on programs not previously managed automatically.

1. INTRODUCTION

Codes parallelized for GPUs routinely yield speedups between 4x and 100x for real applications [9, 15, 29]. Unfortunately, parallelizing code for GPUs is difficult because programmers must explicitly ensure a consistent program state for data-structures shared between CPU and GPU memories. To share data-structures between CPU and GPU, programs must communicate data between CPU and GPU memories. *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.

Manually sharing data-structures between CPUs and GPUs is tedious and error-prone. Semi-automatic techniques, such as Global Memory for Accelerators (GMAC) [11], ease this burden but require programmers to add annotations. Automatically managing data and optimizing communication removes the difficulty completely. Two automatic data management systems exist: Inspector-Executor (IE) [4, 22, 31] and CPU-GPU Communication Management (CGCM) [16].

IE dynamically manages data but does not optimize communication. In IE, a compiler generates an *inspector* for each parallelized code region. The inspector loads the data needed by a parallelized region and transfers the data to the appropriate memory space. Parallel *executor* functions use data sent by the inspector. IE is intended for distributed memory clusters and is unsuitable for GPUs because it cyclically copies data between CPU and GPU memories for each GPU function. In prior work, IE adapted to GPUs yields a whole program slowdown compared to sequential CPU-only execution due to this frequent cyclic communication [16].

CGCM is explicitly designed to manage data and optimize communication for GPUs. 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 dependencies between code on the CPU and code on the GPU. Without cyclic dependencies, 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.

To address the limitations of IE and CGCM, we introduce Dynamically Managed Data (DyManD). DyManD combines IE-inspired dynamic analysis with CGCM-inspired efficient acyclic communication patterns. DyManD matches CGCM's performance without requiring strong alias analysis and exceeds the CGCM and IE's applicability. DyManD creates the illusion of a shared CPU-GPU memory, allowing DyManD to manage complex and recursive data-structures which IE and 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 [12]. 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.

We evaluate how DyManD's insensitivity to type inference and alias analysis leads to improved applicability and performance. To demonstrate alias analysis insensitivity, we compare DyManD's performance with no alias analysis to CGCM's equipped with various alias analyses. The alias analyses tested with CGCM include: no alias analysis, LLVM's production-quality alias analysis, a stack of three recently published research-grade alias analysis, and perfect alias analysis supplied manually. DyManD exceeds CGCM's performance with all automatic alias analysis techniques and matches CGCM's performance equipped with human insight.

DyManD's applicability improvements are evaluated by measuring the performance of 27 programs. For three programs CGCM and IE cannot manage, we perform detailed case-studies comparing DyManD's performance and ease of use with manual communication. These case studies demonstrate that using DyManD is significantly easier and less error prone than manual data management. The results of the performance evaluation indicate that DyManD is as efficient as manual data management.

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.

Section 2 motivates the need for dynamic CPU-GPU data management and optimization. Section 3 describes DyManD, the proposed dynamic communication system. Section 4 evaluates DyManD's performance, using prior work as a baseline. Section 5 surveys prior work, and Section 6 concludes.

2. 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, IE [4, 22, 31] and CGCM [16]. IE 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.

2.1 Prior Approaches to Data Management

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.

Listing 1: Manual explicit CPU-GPU memory management

```
    char *h_h_array[M] = {
      "The woods are lovely, dark and deep.",
      ...
    };

□   __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

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 [24]. Low-level `memcpy`-style pointer manipulation is notoriously difficult for programmers. For real codes, the hazards of manually copying to the GPU include subversive type casting, pointer aliasing, complex data-structures, dynamic memory allocation, and pointer arithmetic. In order to reduce errors and improve productivity, IE and CGCM were introduced. Unfortunately, neither technique can manage parallelized code using complex data-structures.

IE manages data for distributed memory systems but has been adapted to CPU-GPU systems [16]. The inspector function computes all addresses that will be accessed by the parallel executor function. The inspector function must be side-effect free; otherwise the executor function will start in the wrong program state. Consequently, IE is only applicable to parallel code regions with side-effect free address computation. Storing pointers is a side-effecting operation that may affect address computation, since a stored pointer may be loaded and used in an address computation. Modifying array indices stored in memory is forbidden in IE for the same reason.

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 type 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 [13, 19, 33, 34]. 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 [34], managing named regions [13, 19], or affine memory accesses [33].

GMAC [11] 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.

2.2 Prior Approaches to Communication Optimization

Acyclic CPU-GPU communication patterns are much more efficient than cyclic ones. Figure 1 shows an example program's execution schedule using cyclic and acyclic communication. 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

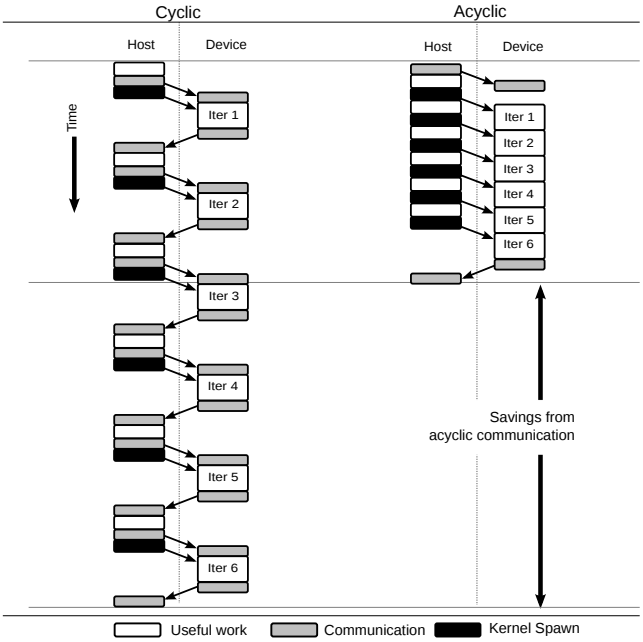

Figure 1: Execution schedules for cyclic and acyclic communication patterns (Iter : Iteration)

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

2.3 Relation of Prior Work to DyManD

Table 1 summarizes the differences between prior annotation-based manual data management systems, IE, CGCM, and DyManD. DyManD avoids the limitations of IE 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

Framework	Data Management	Comm. Opti.	Requires			Applicability				
			Annot.	TI	AA	CPU-GPU	Aliasing Pointers	Pointer Arithmetic	Max Indirection	Stored Pointers
JCUDA [34]	Annotat.	×	Yes	No	No	✓	×	×	∞	×
Named Regions [13, 19]	Annotat.	×	Yes	No	No	✓	×	×	1	×
Affine [33]	Annotat.	Annotat.	Yes	No	No	✓	×	×	1	×
IE [4, 22, 31]	Dynamic	×	Yes	No	No	×	×	×	1	×
CGCM [16]	Static	Static	No	Yes	Yes	✓	✓	✓	2	×
GMAC [11]	Annotat.	Dynamic	Yes	No	No	✓	✓	✓	∞	✓
DyManD	Dynamic	Dynamic	No	No	No	✓	✓	✓	∞	✓

Table 1: Comparison between communication optimization and management systems (Annot: Annotation, TI: Type-Inference, AA: Alias Analysis)

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) [21] 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 [11] also uses exception handling to optimize communication.

3. 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 2 summarizes DyManD's memory allocation and run-time library interface. The remainder of the section will discuss the design and implementation of DyManD's memory allocator, run-time library, and compiler passes.

3.1 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 IE. 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 2) 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, `blockAlloc` first allocates GPU memory normally and then uses `mmap` to map a numerically equivalent address in CPU memory.

Algorithm 2: 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

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 [5, 23]. 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.

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

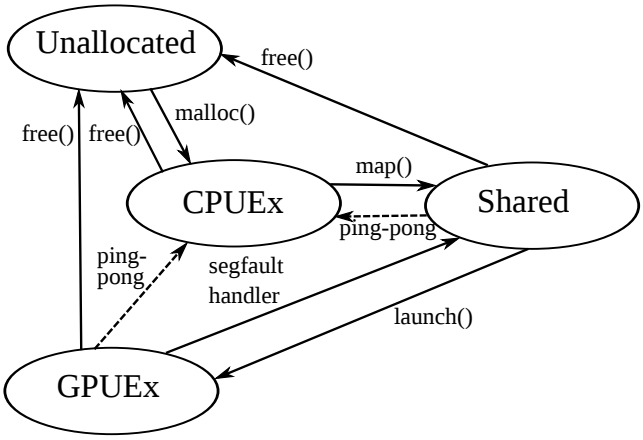

Figure 2: DyManD's state transition diagram for allocation units. The solid lines indicate transitions necessary for correctness. The dashed transitions improve performance heuristically, but are not necessary.

- 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 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 2 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 3) 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.

Algorithm 3: 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 4: 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()
```

Shared to GPUEx via `launch`. The run-time library's `launch` function (Algorithm 4) intercepts calls to GPU functions and copies data to the GPU. The `launch` 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 5: 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)
```

GPUEx to Shared via segfault handler. The run-time library installs an exception handler (Algorithm 5) 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 [25] semantics for access violations. When an access violation occurs 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 0.76x to 6.73x.

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.

3.3 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 [16].

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.

4. 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 in CGCM's original evaluation. 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 [17], an alias analysis stack of three research-grade analyses [14, 18, 20], 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 [14] is inclusion-based, context insensitive, field sensitive, and flow sensitive.
- Lhoták and Chung's points-to analysis [20] is context insensitive, semi-flow sensitive, and supports efficient strong updates.
- Lattner et al.'s pointer analysis [18] is unification based, context sensitive, flow insensitive, and supports heap cloning.

4.1 Experimental Platform

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 4.0 release candidate 2. Both CGCM and DyManD are tuned for best performance on this reference platform.

The parallel GPU version is always compared with the original single threaded C or C++ implementation running on the CPU. 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 130127) 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 [28] performed all automatic parallelizations.

4.2 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 [7] 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 3 shows the performance results for the selected Olden programs.

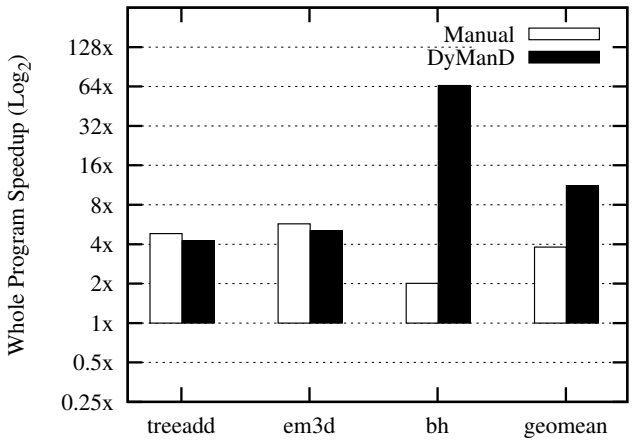

Figure 3: Whole program speedup over sequential CPU-only execution for manual parallelizations with manual and DyManD data management and communication optimization for programs with recursive data-structures.

The alias analysis experiments consist of 27 programs drawn from the PolyBench [26], Rodinia [8], StreamIt [32], and PARSEC [6] 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 [2, 10] 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 [3, 19]. Figure 4 shows performance results for the entire PolyBench suite.

The Rodinia suite consists of 12 programs with CPU and GPU implementations. The CPU implementations contain OpenMP pragmas, but 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.

4.3 Applicability Results and Analysis

Figure 3 shows whole program speedup over sequential CPU-only execution for three manually parallelized Olden programs

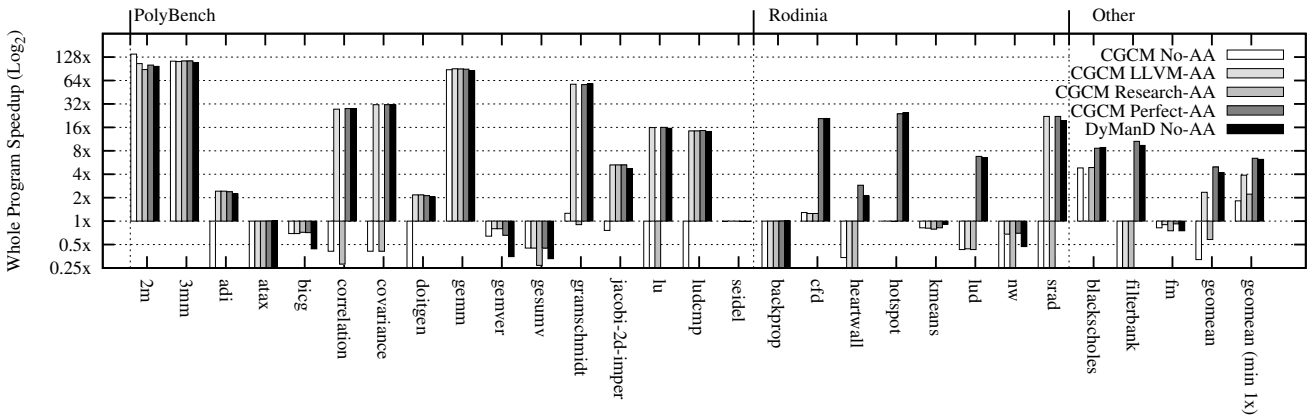

Figure 4: Whole program speedup over sequential CPU-only execution for CGCM with LLVM alias analysis, CGCM with custom alias analysis, and DyManD with no alias analysis.

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 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.

4.4 Insensitivity Results and Analysis

Figure 4 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 0.25x 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 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. RELATED WORK

There are two techniques for managing data automatically: IE [4, 22, 31] and CGCM [16]. IE systems manage data in clusters with distributed memory by inspecting program access patterns at run-time. Prior IE implementations are only applicable to simple array-based data structures. Some IE systems achieve acyclic communication when dependence information is reusable [27, 30]. 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 strongly 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.

CUDA 4.0's Unified Virtual Addressing (UVA) [24] 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 [1] 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 [24, 1]. Pinned memory is relatively scarce and requires programmers or compilers to decide 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 [11, 13, 19, 33, 34], but none handle recursive data structures. "OpenMP to GPGPU" [19] and hiCUDA [13] use annotations to automatically transfer arrays to GPU memory. JCUDA [34] 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 [33] requires programmers to use the C99 `restrict` keyword to provide aliasing information. GMAC [11] 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.

6. CONCLUSION

DyManD is the first dynamic data management and communication optimization system. By replacing static analysis with a dynamic run-time system, DyManD avoids the performance limitations of IE and 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 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.

7. ACKNOWLEDGMENTS

We thank the Liberty Research Group for their support and feedback during this work. We also thank Helge Rhodin for generously contributing his PTX backend. Finally, we thank the anonymous reviewers for their insightful comments. This work is supported by NSF Grants CCS-0964328 and OCI-1047879, DARPA Contract FA8750-10-2-0253, USAF Contract FA8650-09-C-7918, the DOE OS Graduate Fellowship, and a Google Graduate Fellowship. All opinions, findings, conclusions, and

recommendations expressed throughout this paper are those of the Liberty Research Group and do not necessarily reflect the views of our supporters.

8. REFERENCES

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

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2011.

[17] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

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

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

[27] 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.

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

[29] 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.

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

[31] 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.

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

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

[34] 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.