Fast and Efficient Automatic Memory Management for GPUs using Compiler-Assisted Runtime Coherence Scheme

Sreepathi Pai R. Govindarajan Matthew J. Thazhuthaveetil
Supercomputer Education and Research Centre,
Indian Institute of Science, Bangalore, India.
{sree@hpc.serc, govind@serc, mjt@serc}.iisc.ernet.in

Abstract

Exploiting the performance potential of GPUs requires managing the data transfers to and from them efficiently which is an error-prone and tedious task. In this paper, we develop a software coherence mechanism to fully automate all data transfers between the CPU and GPU without any assistance from the programmer. Our mechanism uses compiler analysis to identify potential stale accesses and uses a runtime to initiate transfers as necessary. This allows us to avoid redundant transfers that are exhibited by all other existing automatic memory management proposals.

We integrate our automatic memory manager into the X10 compiler and runtime, and find that it not only results in smaller and simpler programs, but also eliminates redundant memory transfers. Tested on eight programs ported from the Rodinia benchmark suite it achieves (i) a 1.06x speedup over hand-tuned manual memory management, and (ii) a 1.29x speedup over another recently proposed compiler–runtime automatic memory management system. Compared to other existing runtime-only and compiler-only proposals, it also transfers 2.2x to 13.3x less data on average.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Run-time environments, Optimization

Keywords GPU, Memory Management, Data Transfers, Automatic, Software Coherence

1. Introduction

GPUs are now widely used in high performance computing. Three of the current top 5 supercomputers in the world [20] are accelerated by GPUs. Performance obtained on GPUs is often an order of magnitude greater than that obtainable on modern multicore CPUs [13]. A key aspect of GPU performance is the use of local, high-bandwidth *GPU memory* that is distinct from CPU memory. Programs that run on the GPU, called *kernels*, can only access this memory. All data consumed or produced by them must therefore reside in GPU memory. Programs on the CPU which share data with GPU kernels must use explicit transfers between CPU and GPU memories in order to communicate. Current programming models for the GPU, such as CUDA [14] and OpenCL [10], require the programmer to manually perform allocations of GPU memory and transfers of shared data between the CPU and GPU. This manual memory management is tedious, error-prone and a source of performance and portability issues. CUDA 4.0 [15] unifies CPU and GPU memory into a single 64-bit address space ("unified virtual address"), but the GPU and CPU still cannot directly access each other's memory and programmers must continue to perform explicit memory transfers.

CUDA and OpenCL are based on C and C++ and are fairly low-level. Some higher-level languages support programming GPUs directly without the use of CUDA or OpenCL. In particular, the partitioned global address space (PGAS) languages X10 [3] and Chapel [2] already support programming the GPU directly. These languages have built-in support for expressing parallelism, asynchronous communication and execution, and notions of remote and local data. Therefore GPUs can be integrated into these programming languages in a more natural fashion than is possible with CUDA or OpenCL. However, neither X10 nor Chapel fully abstracts CUDA. In X10CUDA [21] (the X10 compiler and runtime for CUDA) the programmer is still required to manually perform GPU memory allocations and transfers. GPU memory management therefore continues to be a source of programming difficulty and potential performance problems even in these languages.

Although there are several proposals that can automate CPU–GPU memory management [6, 8, 9, 11], our analysis (Section 3) shows that all of them introduce redundant transfers. These transfers increase the execution time of programs when compared to the use of hand-tuned manual transfers. Thus, the problem is not only to automate data transfers, but do so in an *efficient* manner. Our proposal differs from past work in the following ways:

- We present the design of the first *full* software coherence mechanism for automating transfers between the CPU and GPU.
- Designed as a compiler–runtime hybrid scheme, it requires no OS or hardware support (unlike [6, 9, 17]), and is therefore widely applicable to accelerators beyond GPUs.
- Importantly, our design treats the GPU and CPU as peers, by keeping full coherence state for both, allowing it to avoid redundant transfers. Hence it outperforms all existing proposals and is also highly competitive with manual memory management.

We integrate our automatic memory management system into X10CUDA, making the following specific contributions:

- We describe compiler analyses, code transformations and runtime support for AMM, a fast and efficient software coher-

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

```
var J:Rail[Float]! = Rail.make[Float](...);

val J_cuda = Rail.makeRemote[Float](gpu, ...);
val E_c = Rail.makeRemote[Float](gpu, ...)
...
for (iter=0; iter< niter; iter++){
    for (...) {
      for(...) {
        // read J
      }
    }

    finish J.copyTo(..., J_cuda, ...);

    //Run GPU kernels
    srad_cuda_1(gpu, E_c, ..., J_cuda, ...);
    srad_cuda_2(gpu, E_c, ..., J_cuda, ...);

    finish J.copyFrom(..., J_cuda, ...);
}
```

Listing 1. Code excerpt from Rodinia's srad ported to X10CUDA with manual memory management

ence scheme that performs automatic memory management for GPUs. We implement the scheme as *X10CUDA with Automatic Memory Management* (X10CUDA+AMM).

- On programs from the Rodinia Benchmark suite, X10CUDA+AMM obtains a geometric mean speedup of 1.06x over the original programs that use hand-tuned programmer-specified memory management.
- X10CUDA+AMM is 1.29x faster than CGCM [8], a recent automatic memory management proposal.
- Comparison with ADSM [6] and OpenMPC [11] reveals that they are slower by 1.32x and 3.72x than the original programs. On average (geomeans), they also transfer 2.2x and 13.3x more data than AMM.

The rest of the paper is organized as follows. We compare manual memory management in X10CUDA to automatic memory management in X10CUDA+AMM in Section 2. We then investigate a naive compiler-based automatic memory management system in Section 3 and demonstrate that it performs poorly because of large number of redundant transfers. Section 4 presents an overview of our scheme, while the compiler analyses, code transformations and runtime support are described in Section 5. We describe our port of CGCM to X10CUDA, resulting in X10CUDA+CGCM in Section 6. The results of our evaluation of X10CUDA+CGCM and X10CUDA+AMM are presented in Section 7. Other relevant related work is discussed in Section 8.

2. Background

Programs for NVIDIA GPUs are written in a C++ dialect called CUDA. CUDA code is structured as *kernels* which execute atomically on the GPU. A kernel's data must reside in GPU memory, which is a distinct address space from that of the CPU. The CUDA API provides functions which a programmer can use to create copies of shared data on the GPU and keep them updated.

X10CUDA [21] provides an X10 dialect of CUDA. Kernels are written as X10 Closures and are translated by the X10CUDA compiler to CUDA code. A GPU is exposed to the programmer as an X10 *Place* which is an X10 construct used to distribute data and schedule computations. Data is transferred using the X10 *Rail* type which is a C array-like type whose contents must reside entirely in a single Place. The Rail type provides methods to create Rails in GPU Places and to transfer data between a CPU-side Rail and a GPU-side Rail.

Listing 1 shows an excerpt from the Rodinia [4] srad benchmark program that we have ported to X10CUDA with the original hand-tuned memory management statements suitably translated, but intact. First, `Rail.makeRemote` is used to create GPU-side copies of CPU-side arrays (akin to CUDA's `cudaMalloc()`). In srad, J_cuda is the GPU copy of J and must therefore be kept in sync with it. There is no CPU-side equivalent for E_c because it is private to the GPU and communicates intermediate data between the two kernels (`srad_cuda_1` and `srad_cuda_2`). It appears in CPU code because GPU memory must be allocated by the CPU as kernels lack the ability to allocate memory.

```
var J:Rail[Float]! = Rail.make[Float](...);
val E_c = Rail.make[Float](...)
...
for (iter=0; iter< niter; iter++){
    for (...) {
      for(...) {
        // read J
      }
    }

    //Run GPU kernels
    srad_cuda_1(E_c, ..., J, ...);
    srad_cuda_2(E_c, ..., J, ...);
}
```

Listing 2. Rodinia's srad in X10CUDA with implicit memory management

Next, the programmer inserts the necessary memory transfers. The $J.copyTo$ and $J.copyFrom$ statements copy to and from the GPU respectively (akin to CUDA's `cudaMemcpy()`). The `srad_cuda_1` kernel reads J_cuda and writes E_c. Therefore, the programmer updates J_cuda on the GPU using the `copyTo` call on J. The `srad_cuda_2` kernel reads from E_c and writes to J_cuda, so the `copyFrom` call is used to update the CPU-side version J. No calls are made to transfer the E_c array since it is never read or written to on the CPU.

Listing 2 shows the same srad program without explicit memory management. In this *implicit* version, no copies of Rails on the GPU are created by the programmer. Nor are memory transfers explicitly specified. The compiler and runtime automatically create copies on the GPU and keep them updated by performing memory transfers when required. This version of srad is therefore much easier to write and involves considerably less book-keeping by the programmer. By removing explicit GPU-specific memory transfers from the source code, we have also increased the portability of the code.

This increase in convenience, productivity and portability cannot come at the expense of performance. In well-written hand-tuned programs, memory transfers are usually not the primary determinants of performance and occupy only a small fraction of execution time. Inefficient automatic memory management can unfortunately change this, as we shall see in the next section.

3. The Need for Efficient Memory Management

Figure 1 shows the total data transferred between the CPU and GPU by a naive automatic scheme for eight applications from the Rodinia [4] benchmark suite written in implicit X10CUDA. The results are normalized to the total data transferred by the original X10CUDA program where transfers are specified manually by the programmer.

This naive automatic scheme transfers the read set into the GPU when a kernel is invoked, and copies the write set to the CPU immediately after the kernel ends[1]. It is easy to implement in a compiler and has been used in an initial version of Chapel [19]

[1] In both the manual and naive versions, the granularity of transfer is an entire Rail (array).

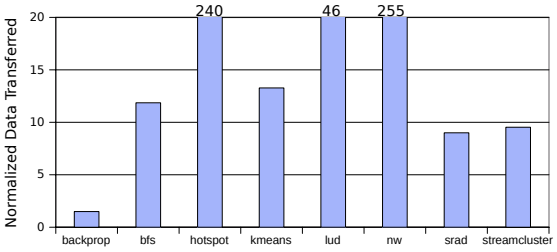

Figure 1. Total data transferred by a naive automatic memory transfer scheme normalized to that transferred by explicitly-specified transfers

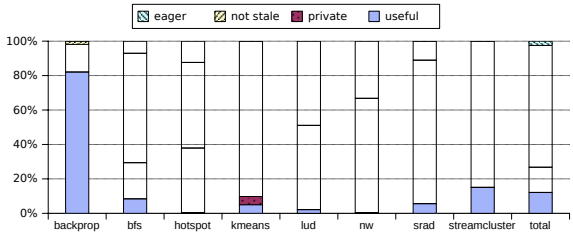

Figure 2. Breakup of data transferred by the naive automatic memory manager

for the GPU and with minor variations in the OpenMP to GPU compiler [12]. The scheme keeps the data always coherent in CPU memory, but it performs a large number of redundant transfers that cause it to transfer 1.5x to 255x more data than manually-specified transfers. We classify these redundant data transfers into three main categories described below and plot this information in Figure 2.

Transfers of non-stale data On kernel invocation, the naive scheme often transfers data which is not stale on the GPU. This happens, for example, when the CPU acts as a read-only consumer of data produced by the GPU kernel. The original copy of such data is in GPU memory and does not need to be transferred back. The naive scheme does not track the state of shared data making these transfers hard to avoid. Consider also the case of the `streamcluster` benchmark which uses a randomized CPU algorithm that only occasionally writes to shared data between kernel invocations. Thus, for some invocations, the GPU will have a non-stale copy of the data since the CPU has not written to it, but for others, it will have to update its copy of the data from the CPU. A pure compiler-only scheme would end up transferring data redundantly every time. The redundant transfers in this case are referred to as "not stale" in Figure 2. A scheme can prevent non-stale redundant transfers by only transferring data that has changed.

Eager transfers of data The naive scheme transfers data eagerly back to the CPU immediately after the kernel finishes. This can lead to redundant transfers, especially in loops that invoke kernels. Consider the following extract from the main loop of the `lud` benchmark, where all the functions invoked are GPU kernels:

```
for (...) {
  lud_diagonal_gpu(m, ...)
  lud_perimeter_gpu(m, ...);
  lud_internal_gpu(m, ...);
}
lud_diagonal_gpu(m, ...);
```

As there is no CPU-side code that reads or writes shared data inside the loop, it is sufficient to transfer the matrix `m` once at the beginning and once at the end of the loop. In this case, even the transfer at the end of the loop can be postponed to after the `lud_diagonal_gpu` outside the loop has executed. The naive

Scheme	Only Non-stale to Stale		Prevents Eager	
	C2G	G2C	C2G	G2C
Naive	N	Y	Y	N
ADSM [6] (Lazy)	Y	N	Y	Y
ADSM [6] (Rolling)	Y	N	N	Y
OpenMPC [11]	N	Y	Y	N
CGCM [8]	N	Y	Y	N
DyManD [9]	N	N	N	P
AMM	Y	Y	Y	Y

Table 1. A comparison of the ability of automatic memory management systems to eliminate redundant transfers. *Legend: P = Partial, C2G = CPU-to-GPU, G2C = GPU-to-CPU*

scheme fails to utilize the opportunity presented by such loops to reduce the number of transfers since it always transfers data before kernel invocation and immediately after the kernel ends. Such loops are not the only source of such redundant transfers. Calls to GPU kernels with no intervening CPU-side reads or writes, like the calls to `srad_cuda_1` and `srad_cuda_2` in Listing 2, also cause redundant transfers in schemes that transfer data eagerly[2]. A scheme can avoid eager transfers by only initiating transfers when it knows the data will be read.

Transfers of private GPU-only data The naive scheme also transfers private GPU-only data between the CPU and GPU. This is data that is resident on the GPU and never read or written by the CPU. It is used by GPU kernels to communicate intermediate data between themselves (e.g., E_c in Listing 2) or across multiple invocations of the same kernel. Since a GPU kernel cannot allocate GPU memory on its own, the CPU must perform the allocation on its behalf and pass the kernel the pointer to the allocated memory. In an automatic scheme, a private GPU array would end up being represented by a CPU array that is never read or written by CPU code, but it would still feature in the read and write sets of GPU kernels. Not knowing that the CPU never reads or writes this array, the naive memory manager would keep this array coherent in CPU memory with transfers that are redundant. In `srad`, of the six same-sized arrays allocated on the GPU, *five* are private to the GPU and are only used for communication between the two kernels. Avoiding redundant transfers of GPU-only private data requires identification of GPU-only private data, but a scheme can also take advantage of the fact that the CPU does not read or write GPU-only private data.

So, of the total data transferred by the naive memory manager, only 12% is useful in that the transfer updates a stale copy of the data. Private data constitutes 14%, eager transfers about 2.5%, while over 70% of data transferred simply overwrites a non-stale copy (Figure 2).

We conclude that an automatic memory management scheme can eliminate redundant data transfers by: (i) not doing eager data transfers, (ii) ensuring that data is only transferred from a non-stale copy to a stale copy, (iii) transferring data only if it will change the destination copy. In practice, the third requirement can be costly to implement, so we assume data that was written to has its value changed as well.

Table 1 evaluates existing automatic memory management schemes against the first two requirements. These schemes span from pure compiler-only schemes (OpenMPC) to compiler–runtime schemes (CGCM, DyManD) and fully runtime schemes (ADSM). We see that *all* current systems fail to eliminate redundant transfers. For example, ADSM does not track changes on the GPU, so it always copies back data from the GPU leading to non-stale transfers. Similarly, compiler-based OpenMPC and CGCM are conservative and transfer modified data eagerly from the GPU back to the CPU

[2] In `srad`'s case, these eager transfers are actually of private GPU-only data, and therefore accounted as "private" in Figure 2.

even when there is no CPU reader. Some of DyManD's transfers, too, are redundant non-stale transfers, because it tracks ownership of data but not whether data has changed or not. It also exhibits eager transfers, because although, like ADSM, it only transfers data back to the CPU on a read or a write, on occasion it brings back all GPU data to the CPU leading to eager transfers. We describe and evaluate these schemes quantitatively in Section 7.

To address these inefficiencies, we propose AMM, a compiler-assisted runtime coherence scheme for automatic GPU memory management. Our scheme reduces redundant data transfers by (i) tracking data staleness, (ii) identifying and preventing transfers of private GPU-only data, and (iii) transferring data in a lazy fashion.

4. Overview of Automatic Memory Management using Software Coherence

Since the CPU and GPU have distinct address spaces, programmers are forced to maintain copies of shared data in each address space. Without a hardware coherence mechanism, these copies must be kept coherent by software, usually by programmers manually performing data transfers. An automatic software coherence mechanism is therefore a natural alternative to manual GPU memory management. Such a mechanism would detect accesses to shared data and automatically schedule data transfers whenever accesses to stale data are detected. Thus, instead of statically inserting memory transfers to keep data coherent, we insert *coherence checks* to ensure that data accessed is not stale. These checks cause the runtime to trigger data transfers dynamically as required.

In this section, we present an overview of how GPU memory is automatically managed using our software coherence scheme in our implementation of X10CUDA+AMM. We discuss what our scheme keeps coherent, how it checks the coherence status of data on the CPU and GPU, and how it initiates data transfers. We then apply our scheme to the `srad` benchmark and demonstrate how it eliminates redundant transfers. The details of compiler analyses, transformations and runtime support required to implement our software coherence scheme outlined here are discussed in Section 5. For simplicity of presentation, we assume a system with one GPU in the text, however, our scheme is trivially extensible to multiple GPUs.

4.1 Implementing Coherence for Rails

The primary data type that must be kept coherent in an X10CUDA program is the Rail. X10CUDA+AMM kernels also accept primitive types, but these are passed by value. In general, CUDA kernels can accept arbitrary pointer parameters, e.g., to structures in GPU memory. However, X10CUDA does not yet support passing structures to its kernels. Nevertheless, our description of implementing coherence for Rails can be generalized easily to other non-primitive types.

We choose to implement software coherence at the granularity of an entire Rail. This granularity is also used by the manual versions of almost all of the benchmarks we tested. In these benchmarks, GPU kernels are written expecting Rails to be transferred in their entirety before and after kernel execution. We speculate that this is the result of the large parallelism available on the GPU, the atomic nature of kernel executions and the inability to transfer data mid-kernel to the GPU.

We associate with each Rail state information indicating its coherence status ("stale", "not stale") on each device (CPU, GPU) as well as addresses of the buffers on each device if allocated. This state is maintained by the runtime as described in Section 5.3. All Rails start out as not stale on all devices until the first write. Then the device that performed the write owns the copy of the Rail, and the other copies are marked as stale.

4.2 Maintaining Coherence on the GPU

A GPU kernel cannot transfer data while executing, so inserting coherence checks into kernels would be futile – we cannot update stale GPU copies mid-kernel execution. Therefore, X10CUDA+AMM must check and update all the Rails in a kernel's read set before it begins execution.

When a kernel is invoked, checks inserted by our compiler cause our coherence mechanism to update all the stale Rails in the kernel's read set from their CPU copies. If this is the first time a particular Rail is being transferred to the GPU, our coherence mechanism also allocates memory on the GPU for it before the transfer.

The coherence status of all the Rails written to by the kernel must also be updated on the CPU. Compiler-inserted calls to our coherence mechanism mark the CPU copy of each Rail in the kernel's write set as stale.

To prevent eager transfers, our approach *does not* transfer the non-stale GPU copy of a Rail back to the CPU immediately after the kernel finishes. Transfer to the CPU of a Rail updated on the GPU is initiated only when a compiler-inserted check on the CPU detects a stale access to that Rail.

4.3 Maintaining Coherence on the CPU

Unlike GPU code, where a kernel's entry and exit points are convenient points to insert coherence checks, there are no clearly defined boundaries in CPU code where checks to maintain coherence can be inserted.

Our coherence mechanism therefore tracks all CPU reads and writes to every Rail in the program. Each read or write of a Rail in CPU code is preceded by a coherence check inserted by our X10CUDA+AMM compiler. If the check detects a stale Rail, it updates the Rail from its GPU copy and sets its coherence status to not stale, so redundant transfers are avoided.

Since checking every read or write would be inefficient, we develop a compiler analysis that conservatively identifies a set of reads and writes that need to have checks inserted before them (Section 5.2.1). Only these reads or writes trigger coherence checks. To prevent overhead from redundant execution of checks at runtime, we also optimize the placement of these checks (Section 5.2.2).

4.4 Putting it all together

We demonstrate how our software coherence scheme triggers data transfers and avoids redundant transfers in the `srad` benchmark, whose high-level description was given in Section 2.

Figure 3 shows the flow of control during `srad`'s execution. We annotate the edges with changes in coherence status for the two Rails J and E_c. The initial part of `srad` reads an image and initializes J. It then executes a loop (Listing 2) that consists of a CPU-side reduction of J and consecutive calls to the two CUDA kernels `srad_cuda_1` and `srad_cuda_2`.

Inside the loop, J and E_c experience changes in their coherence status on CPU and GPU due to the code execution on these devices. Data transfers are only triggered based on the coherence status of the Rails. These transfers occur at kernel boundaries (for CPU to GPU transfers) and before the actual read/write (for GPU to CPU transfers). The elimination of redundant transfers happens as follows.

Elimination of non-stale data transfers The Rail J is initialized by the CPU causing its GPU copy to be marked as stale. The first GPU kernel, `srad_cuda_1` reads J, so in the first iteration of the loop our coherence mechanism transfers it to the GPU. The second GPU kernel, `srad_cuda_2`, writes to J causing the CPU copy to be marked as stale. When the reduce node inside the loop attempts to read J, the compiler-inserted check before the read of J operation

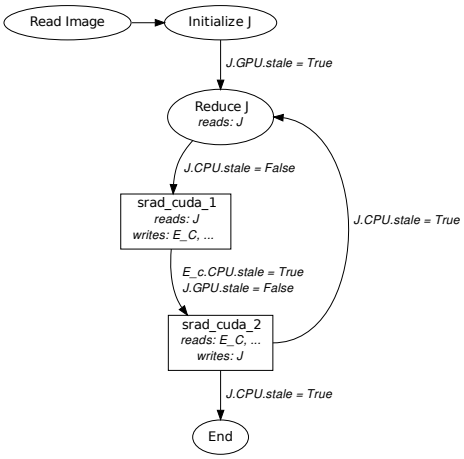

Figure 3. srad runtime control flow. The labels on the edges denote changes in coherence status for the Rails J and E_c on a per-device (CPU/GPU) basis. J is never stale on the GPU after the first iteration of the loop, a fact used by the runtime to eliminate redundant transfers of J to the GPU.

detects the stale access, and a transfer is initiated to update the CPU copy of J. There is no CPU code in the loop that writes to J, so the copy on the GPU is never stale. Hence, in subsequent iterations, transfers guarded by the coherence checks avoid the transfer and no transfers of J to the GPU occur in later iterations of the loop.

Although the redundant transfer in this example can be eliminated by purely compiler-based approaches (e.g., using loop peeling transformation), there are situations (e.g. in streamcluster as explained in Section 3) where runtime techniques are indeed needed to avoid some redundant data transfers.

Elimination of private GPU-only data transfers In srad, E_c is private GPU-only data. It starts out as not stale on the GPU and CPU, and since it is never written by the CPU, the first invocation of `srad_cuda_1` causes a GPU memory allocation for it but no transfer takes place since it is not stale on the GPU. This GPU kernel writes to E_c so the CPU-held copy of E_c is marked as stale. However, no CPU code ever reads E_c hence there are no stale accesses to it and hence no transfers back to the CPU are triggered by the coherence mechanism. Thus, transfers of private GPU-only data are completely avoided.

Elimination of eager transfers Since the software coherence mechanism only transfers data when there are stale accesses, there are no eager transfers. If the srad loop consisted only of kernel calls and the reduction code was eliminated, no reads or writes from the CPU would occur, and data would not be transferred back to the CPU during execution of the loop.

As shown above, our software coherence scheme automatically manages shared data between the GPU and CPU and avoids redundant transfers.

4.5 Coherence Checks versus Data Transfers

Earlier compiler-based approaches [2, 8, 11, 12] insert data transfers statically and use increasingly sophisticated analyses and transformations to avoid redundant transfers. However, the conservativeness inherent in a compiler-only approach cannot avoid all redundant transfers. Our approach of using the compiler to only insert coherence checks while utilizing runtime coherence state to eliminate redundant transfers allows us to avoid this fundamental limitation that plagues earlier work. Compiler conservativeness can only result in redundant coherence checks in our approach, not redundant transfers.

X10CUDA	E.g.	Usage description	X10CUDA+AMM
`makeRemote`	J_c	GPU copy of shared data	*not needed*
	E_c	GPU private data	`Rail.make`
`copyFrom`, `copyTo`		Manual transfers between CPU and GPU	*not needed*
Remote GPU Rail references	J_c	Arguments to kernel functions	Original (J)

Table 2. Differences between X10CUDA and X10CUDA+AMM language features for GPU programs. Examples are from Listing 1.

5. X10CUDA+AMM

Our X10CUDA+AMM compiler is based on the publicly available X10CUDA compiler [21]. Through it, we extend the X10CUDA compiler to insert coherence checks in CPU code and optimize their placement. These coherence checks then invoke our runtime coherence mechanism which maintains coherence status for all Rails and performs dynamic data transfers as required. The X10CUDA runtime uses the asynchronous CUDA API to overlap computation with communication, we plan to use this to perform ahead-of-time transfers to the GPU in the future.

5.1 X10CUDA+AMM Programs

Programs written for X10CUDA+AMM differ from those written for X10CUDA in several ways by not requiring explicit memory management. Table 2 compares X10CUDA and X10CUDA+AMM languages features used for GPU programming.

To summarize, X10CUDA+AMM programs do not to allocate GPU Rails and transfer them between the CPU and GPU manually. If a GPU kernel requires the use of private data, the programmer simply allocates the private data as a standard CPU-side Rail. Even the arguments passed to GPU kernels refer to the original CPU Rails treating GPU kernels as if they were standard CPU functions. This results in programs that look like Listing 2, i.e., smaller, simpler and more portable than their X10CUDA versions.

5.2 Compiler Analyses and Transformations

To determine which Rails must be kept coherent across devices, the compiler conservatively identifies the read and write sets of each GPU kernel using a standard def–use analysis. It then inserts before each call to a kernel, coherence checks for each Rails in the kernel's read and write set. At runtime, before the kernel is invoked, these checks make the GPU copies coherent by transferring them as necessary.

The procedure to insert coherence checks in CPU code relies on compiler analysis to determine which reads or writes in the source code are likely to access stale data. The checks are then placed just before these reads or writes. Since this placement might result in redundant checks executed at runtime, our compiler optimizes their placement. The analysis to insert these checks and the optimization of their placement are discussed in detail in the following sections.

5.2.1 Inserting Coherence Checks in CPU code

As previously discussed in Section 4.3, CPU code lacks the clearly defined boundaries of a GPU kernel where coherence checks can be inserted. The coherence mechanism must therefore check every read or write on the CPU to ensure that no stale data is read or written. We implement this check in the form of two methods, `check_read()` (Section 5.3.2) and `check_write()` (Section 5.3.3), that are invoked on a Rail object before a read and a write respectively. If the data in the Rail is stale, these methods trigger memory transfers and ensure that data available to subsequent

$$
\begin{aligned}
OUT_{WRITE}(ENTRY) &= \emptyset \\
IN_{WRITE}(s) &= \cap_{p,p\in pred(s)} OUT_{WRITE}(p) \\
OUT_{WRITE}(s) &= (IN_{WRITE}(s) \cup write_s) - kill_s \\
FIRST\_WRITTEN(s) &= OUT_{WRITE}(s) - IN_{WRITE}(s) \\
\\
OUT_{READ}(ENTRY) &= \emptyset \\
IN_{READ}(s) &= \cap_{p,p\in pred(s)} OUT_{READ}(p) \\
OUT_{READ}(s) &= (IN_{READ}(s) \cup read_s) - kill_s \\
FIRST\_READ(s) &= (OUT_{READ}(s) - IN_{READ}(s)) \\
&\quad -IN_{WRITE}(s)
\end{aligned}
$$

Figure 4. Data flow equations to determine placement of coherence checks

reads or writes is updated. They also update the Rail's coherence status to not stale, so that redundant transfers are prevented.

Our compiler could insert calls to these functions before every statement that read or wrote to a Rail, but that would be inefficient because of the overheads involved. Not all of these calls are necessary, though. A Rail's coherence status on the CPU can only change when a GPU kernel that reads or writes it is invoked. If it can be determined that a static reference (read or write) would always access coherent data the check can be eliminated. For example, if a static reference to a Rail is preceded by another reference to the same Rail and no GPU kernel invocations occur in between these two references, the second reference is guaranteed to always access coherent data, since the Rail would have been made coherent by the first reference[3].

Our analysis identifies such references so that the compiler can eliminate coherence checks whenever possible. Specifically, for every statement, our approach ultimately calculate two sets: $FIRST\_READ$ and $FIRST\_WRITTEN$ that contain Rails for which that statement is the first reader or writer respectively along some path from entry or the previous GPU kernel call. Our compiler then inserts `check_read()` and `check_write()` calls for these Rails before the statement. Note that there may be one first read or first write for a Rail after each GPU kernel call.

The data flow analysis to calculate $FIRST\_READ$ and $FIRST\_WRITTEN$ (Figure 4) begins by initializing three sets for each statement: the $read_s$ set which contains all Rails read in that statement, the $write_s$ set which contains all Rails written to in that statement, and the $kill_s$ set which contains Rails which may have gone stale during execution of that statement. The $kill_s$ set for a kernel invocation, for example, contains all the Rails read or written to by that kernel.

The dataflow equations in Figure 4 are iterated over the control-flow graph. This produces the set of Rails already read ($IN_{read}(s)$) and written ($IN_{write}(s)$) along all paths from ENTRY to statement s. Intuitively, these sets contain Rails that have reads or writes respectively along all paths from ENTRY to s. If R is a Rail in one of these sets, and if s reads or writes R, we do not need to invoke `check_read()` or `check_write()` on R because a prior read or write would have already done so – a coherent copy of R is available on all paths to s. It is, therefore, the difference of the OUT and IN sets for reads and writes that gives us the Rails first read or written to in this statement.

Once the dataflow analysis has converged, calls to `check_read()` and `check_write()` are inserted for each statement for Rails in that statement's $FIRST\_READ(s)$ and $FIRST\_WRITTEN(s)$ sets respectively.

[3] Note that the coherence is maintained at the granularity of the whole Rail.

```
var J:Rail[Float]! = Rail.make[Float](...);
val E_c = Rail.make[Float](...)
...
for (iter=0; iter< niter; iter++){
    J.CPU.check_read()
    for (...) {
      for(...) {
        // read J
      }
    }

    //Run GPU kernels
    E_c.GPU.check_write()
    J.GPU.check_read()
    srad_cuda_1(E_c, ..., J, ...);

    E_c.GPU.check_read()
    J.GPU.check_write()
    srad_cuda_2(E_c, ..., J, ...);
}
```

Listing 3. Listing 2 after insertion and of placement optimization of checks

5.2.2 Optimizing Placement of Coherence Checks

The analysis of the previous section inserts `check_read()` and `check_write()` as close to a read or write as possible. However, this can result in sub-optimal placements. Consider this example:

```
for(i = ...) {
   R.CPU.check_read();
   ... = R(i)
}
```

In this code, `R.CPU.check_read()` will check if R is coherent in every iteration of the loop. However, the loop in this example contains no calls to GPU kernels, and hence R cannot go stale during execution of this loop. If R was stale on entry to the loop, the first dynamic instance of `R.CPU.check_read` will result in a transfer, but the subsequent dynamic instances from the loop will be redundant.

We can hoist this call out of the loop, resulting in:

```
R.CPU.check_read();
for(i = ...) {
   ... = R(i)
}
```

We implement a pass in our compiler that moves calls to `check_write()` or `check_read()` for a Rail R out of a loop if the following conditions hold: (i) The reference to R is loop invariant, (ii) No statement in that loop causes R to possibly go stale on the CPU (directly or indirectly calls a GPU kernel).

This pass is repeated until no calls are moved. Listing 3 is the result of compiling Listing 2 by our compiler. Checks have been inserted at the appropriate points, and the check for J has been hoisted out of the doubly-nested reduction loop.

5.3 Runtime Support

The X10CUDA+AMM runtime is responsible for tracking the coherence state of a Rail during program execution. It also initiates data transfers dynamically across the CPU and GPU as required.

5.3.1 Modifications to the Rail type

In X10CUDA+AMM, the Rail type is extended to maintain state that is used by the coherence mechanism. Each Rail object contains a coherence status for each device (CPU, GPU) which marks it as "stale" or "not stale" on that device.

This new state also includes the locations of buffers allocated for this Rail on each device. Buffers on the GPU are only allocated the first time data is transferred to it.

We also add the `check_read()` and `check_write()` methods to the Rail type. These methods maintain the coherence status for the Rail, and also initiate transfers as necessary. The Rail is always coherent after these methods are invoked.

5.3.2 The check_read() method

The `check_read()` method (Algorithm 1) signals the coherence mechanism that the CPU or GPU will read the Rail's data in the code to follow.

This method updates the coherence status of the Rail on the device (CPU or GPU) by marking it not stale on that device, after updating the stale Rail by transferring from another non-stale copy if necessary.

In our implementation, we use a highly optimized version of this method to implement checks in CPU code to lower overheads.

Algorithm 1 Pseudocode for the Rail's `check_read()` method

```
if device == GPU then
    if GPU.data_ptr == NULL then
        GPU.data_ptr = cuMemAlloc(Rail.length)
    if GPU.stale then
        copy(destination = GPU.data_ptr, source = CPU.data_ptr)
        GPU.stale ← False
else
    if CPU.stale then
        copy(destination = CPU.data_ptr, source = GPU.data_ptr)
        CPU.stale ← False
```

5.3.3 The check_write() method

The `check_write()` method (Algorithm 2) signals the coherence mechanism that the CPU or GPU will write to the Rail.

The `check_write()` method first performs the equivalent of a `check_read()`, updating a stale Rail from a non-stale copy. The coherence status of the Rail on the device is then set to not stale while all other copies of the Rail are marked as stale.

Again, like `check_read()`, our implementation uses an optimized check to lower overheads. Also, in the case of some X10 library functions that completely overwrite a Rail, our implementation skips the call to `check_read()`, saving on the transfer of data that would be overwritten anyway.

Algorithm 2 Pseudocode for the Rail's `check_write()` method

```
if device == GPU then
    if GPU.data_ptr == NULL then
        GPU.data_ptr = cuMemAlloc(Rail.length)
    if GPU.stale then
        copy(destination = GPU.data_ptr, source = CPU.data_ptr)
        GPU.stale ← False
    CPU.stale ← True
else
    if CPU.stale then
        copy(destination = CPU.data_ptr, source = GPU.data_ptr)
        CPU.stale ← False
    GPU.stale ← True
```

6. CPU–GPU Communications Manager (CGCM) for X10CUDA

The CPU–GPU Communications Manager (CGCM) [8] is a system that automates memory allocations and transfers of data between the CPU and GPU. In that work, CGCM is coupled to an auto-parallelization system that produces GPU code from the programs written in C/C++. In order to compare this state-of-the-art system with X10CUDA+AMM, we port CGCM to X10CUDA, resulting in X10CUDA+CGCM.

Transfers in CGCM are handled by `map`, `unmap` and `release` calls placed into the code by the compiler. The compiler brackets each kernel invocation site with these calls for every Rail in the kernel's read and write set. Simplistically, a `map` and its corresponding `unmap` transfer data to and from the GPU, whereas `map` and `release` manipulate a runtime reference count to prevent some redundant transfers.

To decrease redundant transfers, the CGCM compiler performs *map promotion*, which hoists the above function calls out of loops and function bodies. Map promotion does not succeed if there are CPU references or modifications to the Rails inside the loop/function.

To improve the applicability of map promotion, CGCM performs *glue kernel identification* whereby CPU code that interferes with map promotion is converted to a single-threaded GPU function where possible. We perform map promotions and glue kernel identifications faithfully for all our benchmarks as well.

7. Evaluation

Our objective is to evaluate the AMM scheme and its impact on program execution time. We compare the performance of our AMM implementation against (i) X10CUDA implementation which uses programmer inserted (manual) memory management, and (ii) X10CUDA+CGCM in Sections 7.2–7.4. In Section 7.5, we contrast AMM against the compiler-only OpenMPC and the runtime-only ADSM schemes. In Section 7.6, we quantify the overheads of our implementation.

7.1 Methodology

The benchmarks used in the evaluation are taken from the Rodinia Benchmark suite (v1) [4]. The original benchmarks are available as C++ and CUDA programs and contain manually parallelized kernels and explicit hand-tuned memory allocations and transfers. We port these benchmarks to X10CUDA as faithfully as possible, maintaining the same memory allocations and transfers as in the C++ version. These programs are referred to as the X10CUDA (manual) versions.

X10CUDA does not support passing arrays of structures to the GPU, so we convert programs that use arrays of structures extensively (backprop, bfs, streamcluster) to use structures of arrays instead. The version of X10CUDA used in our work also does not support GPU textures or constant memory, so we modify the kmeans benchmark which uses these features to work without them.

The heartwall, leucocyte, cfd and mummergpu benchmarks are not included because they make extensive use of C/C++ native libraries, GPU SIMD types and structures as kernel arguments, none of which are supported by the base X10CUDA compiler.

We run hotspot for 360 iterations and srad for 100 iterations as described in the Rodinia paper [4].

We rewrite each X10CUDA version of the benchmark for X10CUDA+AMM using Table 2 as a guide. In this rewritten version, all data transfers are implicit. We convert GPU-only private Rails to CPU-side Rails.

To obtain the X10CUDA+CGCM versions, we take the implicit version of the benchmark produced for X10CUDA+AMM and perform Map Insertion, Map Promotion and Glue Kernel Identification manually. Map promotion hoists the map calls for Rails out of all loops in all programs except kmeans and streamcluster. In hotspot, the main loop limits map promotion because the Rodinia source code refers to Rails used by the GPU kernels in a loop-dependent fashion. In srad, a glue kernel for the CPU-side reduction improves the applicability of map promotion. No such opportunities are found in kmeans or streamcluster.

Lastly, the naive version of each benchmark uses the same source code as the X10CUDA+AMM but has transfers inserted

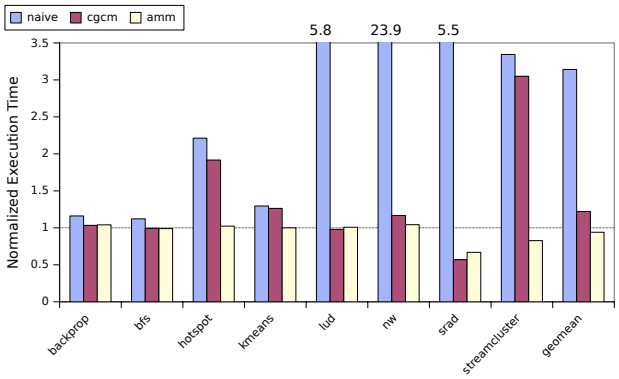

Figure 5. Normalized execution time of benchmarks using automatic memory management relative to the manual version. Lower is better.

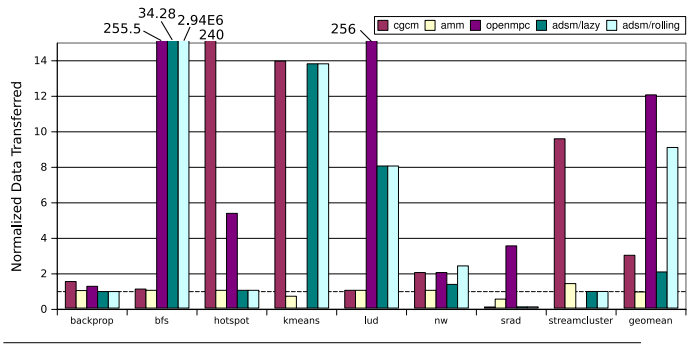

Figure 6. Size of data transferred by X10CUDA+AMM, X10CUDA+CGCM, Cetus+OpenMPC, ADSM versions normalized to data transferred by the manual version. Lower is better.

statically by the compiler. Note that the manual version of each benchmark acts as the baseline for all our experiments.

All of the resulting versions are compiled using the same base X10 compiler, which invokes gcc 4.4.5 and CUDA 2.3 compilers with -O3 optimizations. The benchmarks are run on a system equipped with an NVIDIA Tesla C1060 with 4GB RAM. The CPUs are two quad-core Intel Xeon E5405 2GHz with 16GB RAM. In all our experiments the CPU code and the GPU code are executed in a single X10 Place, which uses two CPU cores (the default) and a single GPU. Each program was run 10 times, and the runtime averaged across those ten runs.

We report normalized execution time of X10CUDA+AMM, X10CUDA+CGCM and X10CUDA+Naive relative to the runtime of the manual version of the program which has hand-tuned memory transfers as in the original Rodinia source code. Normalized execution times are aggregated using geometric means.

7.2 Performance of Individual Schemes

Figure 5 shows the normalized execution times of the implicit X10CUDA+Naive, X10CUDA+CGCM and X10CUDA+AMM versions over their manual version.

First, for reasons discussed in Section 3, the X10CUDA+Naive implementation of memory management performs very poorly for all programs, slowing down execution time by 3.14x (geomean).

Second, X10CUDA+AMM is able to perform as well as the manual version except for nw and backprop where it is within 4%. In two benchmarks, srad and streamcluster, AMM outperforms the manual version of the benchmarks by 34% and 18% respectively. This is due to the reduction in redundant transfers as explained later in Section 7.3.

While X10CUDA+CGCM achieves a higher performance in srad (improvement of 44% primarily due to the glue kernel optimization which reduces the overall data transferred) and comparable performance in bfs, lud and nw, it suffers significant slowdowns in hotspot, kmeans and streamcluster – 91%, 26% and 205% respectively – compared to the manual versions.

Overall, X10CUDA+CGCM is slower than manual by 22%. X10CUDA+AMM, on the other hand, is 6% faster than manual. X10CUDA+AMM is 1.29x faster than X10CUDA+CGCM and 3.33x faster than X10CUDA+Naive.

Next, we compare the amount of data transferred by these schemes to understand their speedups.

7.3 Data Transferred by X10CUDA+AMM

From Figure 6, other than in streamcluster, X10CUDA+AMM does not transfer more data than the equivalent manual version. Further, for three of the benchmarks – backprop, srad, kmeans, it actually transfers *less* data than the manual version.

Examination of the source code for backprop, srad, kmeans reveals they have redundant transfers in the original Rodinia source code that transfer non-stale data to the GPU. X10CUDA+AMM detects and eliminates these transfers. In srad's case, for example, as we noted in Section 4.4, the $J.copyTo$ inside the loop (Listing 1) is redundant after the first iteration.

The case of streamcluster is more complicated. The excess data transferred is not due to X10CUDA+AMM transferring non-stale or private data or even through eager data transfers. It turns out that streamcluster is the only benchmark that performs partial transfers of a Rail in the manual version. Since X10CUDA+AMM always transfers back an entire Rail, the automatic version of streamcluster transfers back more data (3.91x) for one Rail (`work_mem_h`), than the manual version.

However, there are other Rails in streamcluster that are transferred from the CPU to the GPU. These Rails (`p_assign`, `p_weight` and `p_cost`) are only modified occasionally by the CPU inside a randomized algorithm. The manual version of streamcluster transfers these Rails on every kernel invocation, but as X10CUDA+AMM tracks the coherence state of each Rail, they are only transferred if they have been modified by the CPU. In this case, X10CUDA+AMM transfers only 9.5GB instead of 11.5GB transferred by the manual version. Overall, X10CUDA+AMM transfers 1.37x more data than manual in streamcluster. However, the automatic version of streamcluster is 20% faster than the manual version. Overall X10CUDA+AMM transfers on average 0.9x the data transferred by manual.

7.4 Data Transferred by X10CUDA+CGCM

Figure 6 shows that for six of the eight programs, X10CUDA+CGCM transfers more data than the equivalent programmer-tuned manual version. The amount of data transferred is as high as 240x for hotspot, 9.5x for streamcluster and 13.9x for kmeans. In lud, the amount of data transferred under X10CUDA+CGCM is equal to that transferred under manual because the map promotion successfully hoists the map outside the main loop, and then outside the function. In srad, the glue kernel leads to all code being run on the GPU therefore transferring only 0.06x of the data transferred by manual.

To examine why X10CUDA+CGCM transfers more data for the remaining six programs we use the analysis of Section 3 to plot in Figure 7, a breakup of the data transferred by X10CUDA+CGCM. As described in Section 6, X10CUDA+CGCM does not track CPU reads or writes at all. This causes it to transfer non-stale data redundantly, prevents it from identifying private GPU-only data and also causes eager transfers. Note that GPU-only private data has no CPU

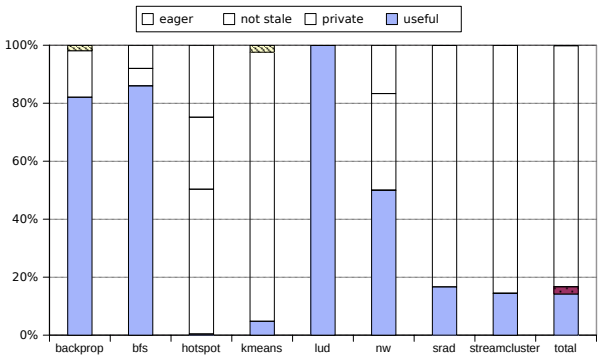

Figure 7. Breakup of data transfers performed by X10CUDA+CGCM

reads or writes, so map promotion does succeed in reducing the amount of private data transferred. However, X10CUDA+CGCM still ends up transferring GPU-only private data at least once.

Loops in which map promotion failed end up contributing to both non-stale redundant transfers and eager transfers. In kmeans and streamcluster, where opportunities to perform map promotion were not found, X10CUDA+CGCM performs equivalent to the naive scheme.

To summarize, X10CUDA+CGCM ends up transferring 2.9x more data (geomean) than manual and only 14% of the total data transferred by X10CUDA+CGCM turns out to be useful. Compared to this, all the data transferred by our X10CUDA+AMM is useful.

7.5 Comparison with Cetus+OpenMPC, ADSM and DyManD

To contrast CGCM and AMM with a compiler-only approach, we present data transfer information for benchmarks compiled by the OpenMPC compiler [11] which translates OpenMP programs to CUDA. It inserts data transfers statically by performing interprocedural analysis to identify GPU-resident and CPU-live variables. We were only able to compile six Rodinia benchmarks with Cetus 1.3. Figure 6 shows that the OpenMPC compiler transfers from 1.22x to 256x the data transferred by the manual versions for these six benchmarks, even when the benchmarks were compiled with the highest level of memory transfer optimizations as supported by OpenMPC. Overall, OpenMPC transfers 12x (geomean) more data.

We also evaluate ADSM [6] which is a fully runtime scheme, but requires the programmer to annotate shared data. It uses OS-level memory protection mechanisms to intercept reads and writes to this shared data. We use the libgmac 0.0.20 [7] implementation and evaluate both Lazy and Rolling schemes. We find that Lazy is faster and transfers less data as compared to the Rolling scheme. ADSM handles partial data transfers and tracks coherence information on the CPU so it is able to eliminate redundant transfers in srad and streamcluster. However, since it does not track GPU coherence information, it transfers non-stale data from the GPU, for example in kmeans. We observe that data structure initialization can confuse ADSM's Rolling scheme leading to repeated redundant transfers. In the extreme case exhibited by bfs, this transfers 2.94×10^6 times more data. Compared to manual, on average ADSM transfers 2x more data for Lazy, and 9x (1.4x without bfs) more data for Rolling.

These excess data transfers lead to loss of performance, as seen in Figure 8 which shows the performance of the ADSM schemes normalized to Rodinia CUDA code. ADSM is faster than manual only for srad, and is upto 2.2x slower for kmeans. Overall, ADSM Lazy is 32% (geomean) slower than the manual Rodinia code and Rolling is 4x slower (39% without bfs). Similarly, OpenMPC's

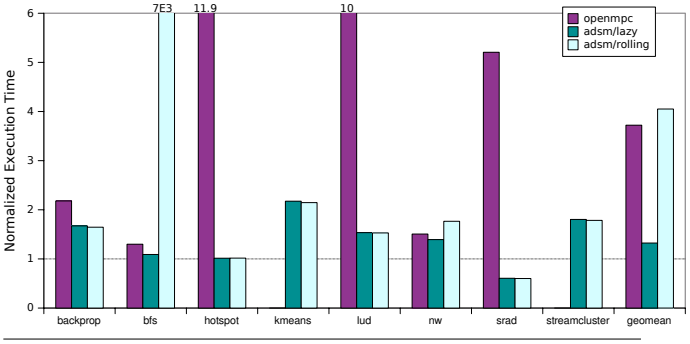

Figure 8. Normalized execution time for OpenMPC and ADSM relative to the Rodinia CUDA manual benchmarks

Benchmark	check_read()	check_write()	OoL
backprop	24	15	29
bfs	3	11	12
lud	9	2	9
nw	6	8	10
srad	4	3	6
hotspot	4	2	6
kmeans	13	11	19
streamcluster	53	37	44

Table 3. Number of coherence checks inserted and hoisted per benchmark. OoL refers to checks that have been placed outside-of-loops.

Benchmark	reads + writes	check_write	check_read
backprop	18,088,926	2,228,377	2,228,345
bfs	859,964	65	12
hotspot	1,835,384	5	3
kmeans	1,975,962,573	6487	6516
lud	65,537	1	1
nw	16,822,812	264	2
srad	14,221,329	2	103
streamcluster	4,733,235,158	21,228,272	26,905,978

Table 4. Number of dynamic `check_write()` and `check_read()` executions compared to number of dynamic reads and writes per benchmark

excess data transfers cause it to never be faster than manual for any program. Overall it is 3.72x (geomean) slower than manual.

DyManD [9] extends CGCM with a memory optimization framework similar to ADSM. However its state machine is considerably less detailed than that of ADSM – tracking only ownership but not coherence state. We therefore believe that its performance will be similar to that of ADSM.

7.6 Overheads of redundant check_read() and check_write()

Table 3 shows the number of `check_read()` and `check_write()` calls that are inserted by our compiler into each benchmark. The *OoL* entry in the table corresponds to the sum of `check_read()` and `check_write()` calls that end up outside loops after the optimization procedure described in Section 5.2.2. The calls that are not hoisted are in loops that contain GPU kernel calls and/or that have loop-dependent references to Rails. In addition, in our current implementation, we do not hoist checks that are inside conditional blocks of code, so as to prevent redundant transfers, even if those conditional blocks lie inside a loop.

Table 4 shows the number of actual dynamic calls that are executed during a run of each benchmark. The absolute number of dynamic `check_read()` and `check_write()` calls are low when

compared to the actual number of reads and writes. Only in backprop do they form a significant fraction (12.3%) of the number of reads and writes. This is because of a loop that iterates over the outermost index of a 2D Rail in the innermost loop, thus creating a loop dependence that prevents the hoisting of the `check_read()` and `check_write()` calls. For all other benchmarks, the fraction of checks to reads and writes is less than 0.01% with the exception of streamcluster for which the checks fraction is 1%.

From this we estimate the runtime overhead for execution of these checks. We measure the cycles taken by `check_read()` and `check_write()` using the `RDTSC` instruction [16]. On average, these methods take 8–9 cycles per call[4] when no data is transferred. The overhead in execution time is very low (the maximum of 2.46% is experienced by backprop) with the average overhead being around 0.35% of runtime.

8. Related Work

Software caching schemes have been used on the Cell accelerator [5] to communicate data between the CPU and the Cell's SPUs. However, unlike SPUs, GPUs cannot initiate memory transfers, so it is not possible to realize software caches on current GPUs.

The OpenMPC compiler translates OpenMP code to GPU kernels [11] and inserts transfers by using an interprocedural analysis to identify GPU-resident and CPU-live variables at compile-time. Our scheme identifies possibly stale reads or writes instead and inserts coherence checks, not transfers. OpenMPC is evaluated in Section 7.5.

Baskaran et al. develop an automatic polyhedral model-based framework [1] to insert data transfers statically. Their method is closely tied to their parallelization framework and does not involve any runtime coherence mechanism to avoid transfers of non-stale data. Our work can automate transfers for both manually-written kernels as well as automatically parallelized GPU code.

Saha et al. describe a programming model for C on heterogeneous x86 platforms [17] which provides a logical shared address space between the CPU and an accelerator modeled on the Larrabee [18]. Coherence is maintained by the CPU and the accelerator by intercepting reads and writes using the operating system's memory protection mechanisms. However, this requires the accelerator to support virtual memory and protection to support its coherence mechanisms, something current GPUs lack.

The ADSM/GMAC [6] system also provides shared pointers, but does so via an asymmetric distributed memory system that does not require the GPU to participate in coherence management. It is implemented as a layer over CUDA. Reads and writes on the CPU are intercepted using the operating system memory protection mechanisms. Shared objects are annotated by a programmer. Our work does not require operating system support or annotations.

Our work is closely related to the CGCM [8] system which provides a compiler and runtime to automatically manage memory for CUDA programs written in C++. DyManD [9] extends CGCM by adding a runtime ADSM-like memory optimization framework.

We have compared and evaluated our work against CGCM, ADSM and OpenMPC in this paper.

9. Conclusions

In this paper, we have presented the AMM system which is a hybrid compiler-assisted runtime coherence scheme to automatically manage all aspects of CPU–GPU communication including allocations and memory transfers. Our design was motivated by the observation that not all redundant transfers can be completely eliminated at compile time. Our system can be used both by a programmer writing GPU kernels manually and as part of a parallelizing compiler. It significantly eases the task of programming the GPU by eliminating a potential source of errors and performance problems.

We have incorporated AMM into the X10CUDA compiler and runtime, and evaluated it on a set of benchmark programs from the Rodinia suite. The AMM system achieves comparable performance to programmer-inserted manual memory management. In fact it achieves a speedup of 1.06x over manual by eliminating some redundant data transfers. Compared to a state-of-the-art memory management system, CGCM [8], our AMM system is faster by 1.29x. Moreover, compared to other existing runtime-only and compiler-only proposals, it also transfers 2.2x to 13.3x less data on average.

[4] Our implementation is very efficient. In an experiment where *all* reads and writes were subject to coherence checks, we observed noticeable slowdowns only in kmeans (19%) and streamcluster (8%).

Acknowledgments

The authors acknowledge partial funding from Microsoft Corporation towards this work.

References

[1] M. M. Baskaran, U. Bondhugula, et al. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP*, 2008.

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

[3] P. Charles, C. Grothoff, V. Saraswat, et al. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[4] S. Che, M. Boyer, J. Meng, et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[5] A. E. Eichenberger, K. O'Brien, et al. Optimizing compiler for the CELL processor. In *PACT*, 2005.

[6] I. Gelado, J. E. Stone, et al. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ASPLOS*, 2010.

[7] I. Gelado et al. GMAC: Global Memory for Accelerators (version 0.0.20). URL `http://code.google.com/p/adsm/`.

[8] T. B. Jablin, P. Prabhu, et al. Automatic CPU-GPU communication management and optimization. In *PLDI*, 2011.

[9] T. B. Jablin, J. A. Jablin, et al. Dynamically Managed Data for CPU-GPU architectures. In *CGO*, March 2012.

[10] Khronos. OpenCL: The open standard for parallel programming of heterogeneous systems. URL `http://www.khronos.org/opencl`.

[11] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC*, 2010.

[12] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP*, 2009.

[13] V. W. Lee, C. Kim, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, 2010.

[14] NVIDIA. CUDA: Compute Unified Device Architecture. URL `http://developer.nvidia.com/cuda`.

[15] NVIDIA. NVIDIA CUDA C Programming Guide version 4.0. 2011.

[16] G. Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. 2010.

[17] B. Saha, X. Zhou, et al. Programming model for a heterogeneous x86 platform. In *PLDI*, 2009.

[18] L. Seiler, D. Carmean, et al. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3), 2008.

[19] A. Sidelnik, B. L. Chamberlain, M. J. Garzaran, and D. Padua. Using the High Productivity Language Chapel to target GPGPU architectures. Technical report, UIUC Dept. of Computer Science, 2011.

[20] TOP500.org. The Top 500. URL `http://www.top500.org/`.

[21] x10 lang.org. X10 2.1 cuda. URL `http://docs.codehaus.org/display/XTENLANG/X10+2.1+CUDA`.