Interactive Program Debugging and Optimization for Directive-Based, Efficient GPU Computing

Seyong Lee[†], Dong Li[†], and Jeffrey S. Vetter[†] [⋆]

[†]Oak Ridge National Laboratory, [⋆]Georgia Institute of Technology
{lees2, lid1}@ornl.gov, {vetter}@computer.org

***Abstract*—Directive-based GPU programming models are gaining momentum, since they transparently relieve programmers from dealing with complexity of low-level GPU programming, which often reflects the underlying architecture. However, too much abstraction in directive models puts a significant burden on programmers for debugging applications and tuning performance. In this paper, we propose a directive-based, interactive program debugging and optimization system. This system enables intuitive and synergistic interaction among programmers, compilers, and runtimes for more productive and efficient GPU computing. We have designed and implemented a series of prototype tools within our new open source compiler framework, called Open Accelerator Research Compiler (OpenARC); OpenARC supports the full feature set of OpenACC V1.0. Our evaluation on twelve OpenACC benchmarks demonstrates that our prototype debugging and optimization system can detect a variety of translation errors. Additionally, the optimization provided by our prototype minimizes memory transfers, when compared to a fully manual memory management scheme.**

***Keywords*-interactive debugging; performance optimization; directive programming; GPU; OpenACC; OpenARC**

I. Introduction

GPU computing is now a mainstream computing scheme in high performance computing (e.g., Titan [1], TSUBAME2 [2], Tiahne-1A [3], and Keeneland [4]). GPU-like architectures have been highlighted as possible building blocks for future Exascale systems [5]. A key factor driving the popularity of GPUs is their capability to deliver cost-effective and energy-effective performance. However, these strengths come at the cost of programmability: programming GPUs requires expertise and can be complicated and error-prone.

Recently, several directive-based GPU programming models have been proposed to improve the productivity of GPU computing [6], [7], [8], [9], [10], [11], [12]. To understand these emerging GPU programming models in terms of programmability and performance, our previous work [13] thoroughly investigated them from multiple perspectives, and identified their limitations. Among those limitations, debuggability is one of the most immediate and pressing issues for directive-based GPU programming models. On one hand, these programming models hide multiple stages of complex optimizations and translations from programmers to improve productivity. On the other hand, the opaque nature of these programming models impose great challenges for users to debug and verify the program. For example, these models raise the abstraction of GPU memory hierarchy and provide unified memory management across architectures; this new level of abstraction complicates debugging for memory-related problems. In addition, these models employ a thread/task parallelism model different from the low-level programming models (e.g., CUDA [14]), which complicates debugging for concurrency. Although the GPU community has a number of tools [15], [16], [17], [18], [19] to debug, profile, and analyze GPU codes, they are limited to applications written in the low-level programming models. When applied to directive-based programming models, they lack the proper capabilities to attribute specific output GPU translated code, performance, and errors with their respective directive-annotated input programs. Hence, they do not give straightforward views for programmers to diagnose logic errors and performance problems at the directive level.

Another significant challenge for directive-based programming models is performance optimization, and, in particular, optimizing CPU-GPU data movement. In most of today's accelerator based systems, the address spaces for GPU and CPU are separate: to communicate data between the two, applications suffer a relatively high penalty from data transfer cost. This cost is especially high in discrete GPUs because of communication cost across the PCI-e bus. Although architectures that fuse GPUs and CPUs into a single chip are gaining in popularity [20], and many examples exist in the consumer market, they still require precise data orchestration and coordination between the CPU and the GPU for efficient performance. Directive-based programming models expose CPU-GPU memory management explicitly to programmers, forcing them to dictate the CPU-GPU data movement and coordination. Hence, minimizing data movement with correctness guarantees is left to the programmers, which can be difficult to realize, given the opaqueness of the translation from the high-level model to the low-level model. Although recent efforts [21], [9], [22], [23], [24] proposed automatic CPU-GPU data transfer schemes, they are limited to avoiding redundant transfers, and some of them cannot work with directive-based programming models.

1530-2075 2014
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/IPDPS.2014.57

IEEE computer society

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.

A. Contributions

To address the above problems, we propose and implement an interactive program debugging and optimization system for productive and efficient GPU computing using directive programming models. Our work is the first effort toward improving traceability and debuggability for directive-based GPU programming. Our major contributions are summarized as follows:

- We propose a directive-based, interactive GPU program debugging technique, which provides an intuitive and synergistic environment to provide better interaction among programmers, compilers, and runtimes than existing debugging tools [15], [18], [19].
- Our technique is able to verify correctness in the GPU-kernel translation and the CPU-GPU memory-transfer-code generation, solving the aforementioned debuggability problem.
- We present a combined compile-time/runtime method that interacts with the programmer to identify redundant CPU-GPU memory transfers, such that the programmer can optimize memory transfers iteratively, through the directives, and based on the suggestions offered by the proposed system. This intuitive user-interaction minimizes redundant data transfers, which is not available in the previous software coherence mechanisms [21], [22], [23] due to correctness constraints.

The rest of this paper is organized as follows. §II explains the background on the directive-based GPU programming and reviews related work. §III presents the proposed directive-based, interactive program debugging and optimization techniques. Evaluation and conclusions are presented in §IV and §V, respectively.

II. Background and Related Work

A. Directive-Based GPU Programming

Directive-based GPU programming models consist of three components: the compiler directives, library routines, and environment variables. With the directive-based programming models, GPU programs are written by augmenting sequential programs with a set of directives that describe important program properties, such as parallelism types to execute loops, and data scope and sharing rules to manage data and synchronize with CPU. At compile time, the directive compilers translate the annotated programs into low-level GPU programs by performing all the complex transformations, such as GPU-kernel-code generation and CPU-GPU memory-transfer-code generation.

Our work is based on OpenACC directives. OpenACC [11] is the first directive-based GPU programming standard portable across devices and compiler vendors. In an OpenACC program, a *compute region* specifies a code region to be executed on GPU, and the directives manage parallelism and guide how loops in the compute region should be executed; a *data region* sets a data boundary for GPU memory allocation, and the directives manage data and control memory transfers between CPU and GPU.

```
#pragma acc data create(q,w,...) ...
{ ... //Some compute regions
  for( it=1; it<= NITER; it++ )
    for( cgit=1; cgit <= cgitmax; cgit++ ) {
      ... //Some compute regions
      #pragma acc kernels loop gang worker
      for( j=1; j<= lastcol-firstcol+1; j++)
      {   q[j] = w[j];    }
      ... //Some compute regions
    } ... //Some compute regions
}
```

Listing 1: Code excerpt from NAS Parallel Benchmark CG ported to OpenACC. The code is manually optimized for memory management

The major benefit of using directive-based GPU programming models is that they obviate the need for dealing with complexity of low-level GPU programming languages, such as CUDA [14] and OpenCL [25], and they also hide most of complex optimization details specific to the underlying GPU architectures. These high-level abstractions allow programmers to focus on their algorithms, hence improving their productivity and portability. In addition, programmers can easily provide performance-critical information to the compilers, enabling various compile-time/runtime optimizations to better utilize the underlying system resources.

However, the high-level abstraction bestowed by directive models puts a significant burden on programmers in terms of debugging and performance optimization, as we shall show in the next section.

B. Motivation for Improving GPU Program Debugging

The high-level programming and automatic transformation in directive models bring several challenges for debugging GPU programs, especially when combined with implicit compiler optimizations. For example, automatic privatization and reduction variable recognition are two important compiler techniques used to parallelize loops [26]. Due to their importance, the existing GPU-directive compilers support these features to some extent. However, if the directive compilers miss privatization/reduction transformation opportunities, and the programmers do not explicitly specify data sharing rules for each compute region, the resulting GPU program can suffer from a race condition due to an incorrect data dependency. Identifying this concurrency problem is very difficult, and demands deep knowledge of both GPU programs and underlying compilers.

Moreover, evaluations with existing GPU-directive compilers [13] reveal that some compiler implementations do not always follow directives inserted by users, or even worse, those implementations can generate incorrect output

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.

codes, when the directives provided by programmers conflict with internal compiler analyses. These unexpected behaviors add additional challenges to the already complex debugging problems.

To improve debuggability for the directive-based GPU programming models, we must have a systematic approach that exposes more information to programmers for debugging purposes while maintaining high-level abstractions. First, we need more intuitive tools to delineate the incorrect code regions. Second, we need traceability mechanisms to attribute errors and performance issues back to input directive programs.

To address the above issues, we develop a directive-based, GPU kernel debugging mechanism that detects erroneous GPU kernels through step-by-step comparison between translated GPU kernels and corresponding input compute regions, described in §III-A.

C. Motivation for Improving GPU Memory Management

Currently, GPU programming models assume separate address spaces for CPU and GPU. To share data between CPU and GPU, many programs use explicit data transfers between CPU and GPU memories. GPU directives provide a rich set of *data clauses* to control memory transfers. These clauses rely on programmers to orchestrate memory management, which is another source for debugging and performance issues.

Listing 1 shows an excerpt from NAS Parallel Benchmark CG that we ported to OpenACC. The code is fully tuned for GPU memory allocation and CPU-GPU memory transfers. Within the code, the *data* directive is used to specify that GPU memory should be allocated for variable q and w for the duration of the following code region. In addition, each compute region is annotated with the *kernels* directive to tell the compiler that the compute region should be transformed into *GPU kernels* to be executed on GPU. In this example, there is no CPU-GPU data transfer for variables q and w, because these variables do not appear in any explicit/implicit memory-transfer clauses (e.g., *copy* and *update host*). We do not need to transfer data between CPU and GPU for these variables, because they are accessed only by GPU (i.e., *private GPU-only data*). To identify private GPU-only data to avoid redundant data transfers, the whole program should be examined to check whether there is any CPU code accessing the data. Considering possible aliasing issues, this process is very complex and error-prone. Moreover, if a variable is shared between CPU and GPU, locating the proper points to trigger CPU-GPU memory transfers is also non-trivial. For example, if a compute region resides in a loop, as shown in Listing 1, conservative data transfers may lead to redundant transfers, while inappropriately deferred transfers can cause incorrect outputs.

To guarantee program correctness, a naive memory management scheme is to copy all the data accessed in a compute region from CPU to GPU right before the corresponding GPU kernel call and copy back from GPU to CPU right after the kernel call. This scheme is the default OpenACC memory management scheme for variables that have no explicitly specified data transfer. As shown in Figure 1, however, the naive scheme can cause excessive (redundant) data transfers, which become a major performance bottleneck.

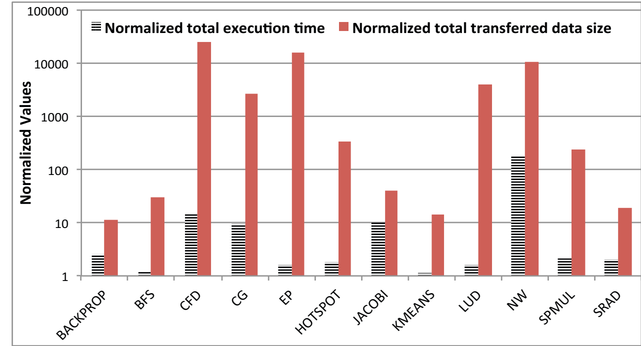

Figure 1: The execution time and transferred data size with OpenACC default memory management scheme. The values are normalized to those for fully optimized OpenACC code.

To address the above CPU-GPU memory transfer issue, several solutions have been proposed, including compiler-only solutions [9], runtime-only solutions [21], and hybrid compile-time/runtime solutions [24], [22], [23]. The state-of-the-art technique [23] uses a compiler-assisted, runtime coherence scheme that initiates necessary memory transfers at runtime, based on the CPU-GPU coherence status. This method can eliminate the following types of redundant transfers: (i) transfers of non-stale data, (ii) eager transfers of data, and (iii) transfers of private GPU-only data. The same software coherence solution can be applied to directive-based GPU programming models with minor modification. However, some redundant/incorrect transfers still exist because of the limitation in this solution. First, the runtime scheme in the solution does not handle redundant or incorrect transfers caused by user-inserted data clauses; hence, it cannot be leveraged for debugging purposes. Second, the coherence scheme in the solution updates all the first-written variables. This method can cause unnecessary data transfers, because some values may be *dead*: they may be no longer used, or may be overwritten before read. However, detecting dead variables is known to be a very complex and difficult task for both compilers and runtimes. For example, in Listing 1, the variable q is write-only in a compute region. However, we cannot make a conclusion that q is dead at the entrance to the compute region, because q is partially written. To automatically check the deadness of the variable q, we need a powerful compile-time array-section-analysis technique to handle all possible aliasing issues for coherence analysis, or demand runtime inspection for all data accesses, none of which is feasible.

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.

To address the above problems, we propose a directive-based, interactive memory-transfer verification and optimization technique, described in §III-B.

Table I compares our work with the existing ones. In general, our work complements the existing ones by providing high-level debugging and optimization capability for directive-based GPU programs.

III. Directive-Based, Interactive Program Debugging and Optimization

This section presents the interactive program debugging and optimization scheme. The proposed scheme consists of two parts: *GPU-kernel verification* (§III-A) to locate problematic kernels, and *memory transfer verification and optimization* (§III-B) to detect incorrect/missing/redundant memory transfers.

A. GPU Kernel Verification

Even though debugging in the directive-based GPU programming is very difficult, directive models provide at least one good reference for debugging: *the original sequential program*. If we can compare execution outputs of the input sequential CPU program to those of the translated GPU program in a lockstep manner, we can pinpoint the code regions requiring debugging. For various reasons (e.g., high overhead and too complex to trace all threads executing GPU kernels at finer granularity), a lockstep debugging strategy is impractical. Hence, we propose to compare these results at the granularity of a GPU kernel.

With the directive-based GPU models, the manual kernel verification is non-trivial. First, GPU memory is hidden by directive models, and thus programmers can not directly access GPU data. To compare the outputs of GPU kernels with those of corresponding code regions in the input sequential program, we must tweak the code to leverage CPU memory to make the comparison. Second, each kernel verification may require customized memory transfer patterns, which complicates debugging. Third, naive comparisons do not work due to inconsistent floating-point precision between CPU and GPU. In particular, CPU and GPU may use different numbers of bits to store floating-point numbers, and non-deterministic computation order enforced by parallel GPU thread executions can result in different outputs for the same computation.

We address the above kernel debugging problems through a user-assisted, automatic kernel verification mechanism. With the proposed mechanism, a user can specify which specific kernels to verify by adding directives or using environment variables (e.g., *"verificationOptions=complement=0,kernels=main_kernel0"* informs the compiler to verify a specific kernel, *main_kernel0*.). At the heart of the proposed mechanism, we introduce two techniques, called *automatic memory-transfer demotion* and *user-configurable result comparison*.

```
1  { ... //Some compute regions
2    for( it=1; it<= NITER; it++ )
3      for( cgit=1; cgit <= cgitmax; cgit++ ) {
4        ... //Some compute regions
5        #pragma acc kernels loop async(1) \
6        gang worker copy(q) copyin(w)
7        for( j=1; j<= lastcol-firstcol+1; j++)
8        {    q[j] = w[j];    }
9        //Sequential CPU version will be added.
10       #pragma acc wait(1)
11       //Result comparison codes will be added.
12       ... //Some compute regions
13     } ... //Some compute regions
14 }
```

Listing 2: Modified version of Listing 1 after applying memory-transfer demotion, but before result-comparison transformation.

The memory-transfer demotion refines memory transfer patterns such that all data accessed by the target GPU kernel are copied from CPU to GPU right before the kernel invocation, and all data modified by the kernel are copied back to a temporary space on the CPU after the kernel finished. This is implemented by (i) moving (*demoting*) any related data clauses in enclosing *data* regions to the target compute region (changing original data management policy), and (ii) adjusting transfer types for each data as necessary (e.g., putting the data in a *copyin* clause if the data are *read-only*; otherwise, in a *copy* clause).The memory-transfer demotion rules out program errors caused by missing memory transfers, and avoids unnecessary result comparison. With this method, the memory transfers as well as the kernel execution are converted to asynchronous ones to allow maximal overlapping with sequential CPU execution. Although this method has impacts on communication behaviors between CPU and GPU, it does not change the behavior of the target kernel. The main goal of the kernel verification is to verify that the translated GPU kernel behaves correctly with respect to the input sequential CPU code, so enforcing a strict execution order between CPU and GPU is not necessary. Moreover, all directives and runtime calls unrelated to the target kernel are removed, such that the unrelated compute regions are sequentially executed on CPU. This method allows the target kernel to always use data generated by the original sequential program, hence it avoids error propagation from the previous code regions. Listing 2 shows the codes after memory-transfer demotion is applied to Listing 1. In particular, data clauses in the enclosing *data* directive are moved to the target compute region, and additional directives are added for asynchronous kernel execution.

After memory-transfer demotion, the compiler makes another pass (i.e., result-comparison transformation) to compare the outputs of CPU and GPU. The compiler generates new versions of the code and some harness code to make

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.

Table I: Comparison of debugging (DG) and optimization (OP) tools

DG and OP tools	High-level DG and OP for directive-based programming	Data transfer optimization	User interaction	Configurability of DG and OP	fine-grained and detailed profiling
GPU PerfStudio [15] and Visual Profiler [17]	No	No	Limited	Limited	Yes
TotalView [18] and DDT [19]	Limited	No	Limited	No	Yes
[22], [23], [24]	No	Yes	No	Limited	No
This paper	Yes	Yes	Rich	Rich	No

the comparison (line 9 and 11 in Listing 2). If the output difference is bigger than the allowed error margin, then an error will be reported to the user. To handle various computation precision mismatch between CPU and GPU, the transformation allows users to configure the error margin according to the program characteristics and the underlying architectures. The user can also decide when to make the result comparison with user-defined conditions. For example, *"minValueToCheck=1e-32"* enforces that result is compared only if its value is bigger than a specified threshold (*1e-32*).

By comparing execution results of the reference sequential program and the translated GPU program at runtime at a kernel granularity, we can easily identify problematic kernels caused by various reasons, such as incorrect user annotations or incorrect compiler translation. However, the proposed kernel verification mechanism cannot detect errors caused by incorrect memory transfers. Hence, we further propose a memory transfer verification scheme, explained in the following section.

B. Memory Transfer Verification and Optimization

The memory-transfer verification and optimization scheme uses runtime coherence checking as an offline profiling tool to detect incorrect, missing, or redundant memory transfers. To accurately capture variable access information, each variable of interest (e.g., the variable shared between CPU and GPU and accessed within a compute region) is associated with one of three coherence statuses ("notstale", "maystale", "stale") on both CPU and GPU. In the current implementation, we track coherence status at the granularity of entire array or memory region allocated by a *malloc* call. Tracking memory accesses at this granularity is a common practice in GPU programming [27], [23]. Although CPU-GPU memory coherence at finer granularities can further reduce the overall size of memory transfers, too frequent status updates at finer granularity suffer from high data transfer latency between CPU and GPU. Therefore, using data coherence at a coarse granularity and efficiently utilizing GPU memories is a preferred approach to minimize both frequency and data size of CPU-GPU memory transfers. The coarse-grained coherence is also preferable for tracking purpose, since a fine grained tracking can take significant execution time, resulting in bad user experiences for debugging. In addition, because of aliasing issues, tracking at finer granularities is very vulnerable to losing tracking accuracy.

The coherence state is maintained by the runtime. In particular, all variables of interest start out as *not-stale* on CPU and GPU until the first write. If a variable is modified on either a CPU or GPU device, the state on the other device is set to *stale*. The state of a stale variable is reset to *not-stale* if the up-to-date value is copied back to the stale variable through memory transfers, or if the stale variable is overwritten locally. The runtime coherence state is used to detect missing memory transfers. In particular, if the state of a variable to be accessed by the local device is stale, this means that the other remote device has modified the variable, requiring a memory transfer from the remote device to the local device before the local access (i.e., *missing transfer*). The coherence state is also used to detect incorrect and redundant transfers. In particular, if the state of a variable for a memory transfer at the source is stale, then the memory transfer is incorrect, because the outdated value is copied (i.e., *incorrect transfer*); if the state of a variable for a memory transfer at the target is not-stale, then the memory transfer is redundant, because the target already has the up-to-date value (i.e., *redundant transfer*).

The above runtime checking mechanisms cannot detect certain redundant transfers. Particularly, when a memory transfer occurs, if the target is *dead* (refer to §II-C), then the memory transfer is redundant, because the copied value will not be used any more. As explained in §II-C, detecting dead variables accurately at either compile-time or runtime alone is known to be very challenging. Therefore, we propose an alternative, hybrid approach that combines the best efforts of the compiler and the runtime. This approach asks the compiler to specify those presumably dead variables (*may-dead*) and verified dead variables (*must-dead*) based on a static analysis. Then, the runtime checks which memory transfers actually involve may-dead/must-dead variables and reports them to programmers. It is up to the programmers to decide which memory transfers are redundant and safe to delete. The programmers' decisions are saved back into the input directive program for production run. These steps are repeated until no redundant or incorrect transfer is found. This hybrid approach makes a best effort to guarantee program correctness by conservatively detecting dead vari-

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.

ables, while providing sufficient hints for users to optimize performance.

Algorithm 1 May-Dead/May-Live Variable Analysis

$OUT_{Live}(EXIT) \Leftarrow \emptyset$
$OUT_{Dead}(EXIT) \Leftarrow \emptyset$
$OUT_{Live}(n) \Leftarrow \bigcup_{s,s \in succ(n)} IN_{Live}(s)$
$OUT_{Dead}(n) \Leftarrow \bigcap_{s,s \in succ(n)} IN_{Dead}(s)$
$IN_{Live}(n) \Leftarrow OUT_{Live}(n) - KILL(n) - DEF(n) + USE(n)$
$IN_{Dead}(n) \Leftarrow OUT_{Dead}(n) - KILL(n) + DEF(n) - USE(n)$

Algorithm 1 shows the data flow analysis to identify may-dead and must-dead variables. The algorithm also introduces *may-live* variables (i.e., variables that are accessed later but read before written) to facilitate analysis. The analysis begins by initializing may-dead and may-live sets at the program exit ($OUT_{Dead}(EXIT)$ *and* $OUT_{Live}(EXIT)$). From the exits to the program entry, the analysis checks *written* variables (*DEF(n)*) and *read* variables (*USE(n)*) in each statement (*n*). The *KILL(n)* set refers to variables that have gone stale during execution of the statement *n*. If a variable is written-first in all of the following execution paths, it is added to may-dead set; if the variable is read in some of the following execution paths, it is added to may-live set. If a variable is neither may-dead nor may-live, the variable will not be accessed any more in the following execution paths (i.e., adding to must-dead). We perform the above analysis twice, one for CPU variables and the other for GPU variables. Then, the may-dead and the must-dead variables in CPU and GPU are passed to the runtime. The runtime sets the state of must-dead variables to "not-stale" and sets the state of may-dead variables to "may-stale". At runtime, memory transfers occurred to those variables with the not-stale state are reported as *redundant*, while memory transfers occurred to those variables with the may-stale state are reported as *may-redundant*.

To enforce the above runtime memory-transfer-checking mechanism, the compiler should insert various runtime-check calls, detailed as follows:

- Each read/write access should be preceded by runtime checks, particularly *check_read()* and *check_write()*. They detect any missing or *may-missing* memory transfers; the may-missing refers to the case where a target variable is stale, but it is written before read. This case requires a memory transfer only if written data do not fully overlap with data that are read later.
- Each write access, which changes the variable state at the remote device to stale, should be also followed by *reset_status()*, if the corresponding variable at the remote device is may-dead or must-dead at the current statement; *reset_status()* changes the variable state at the remote device to may-stale or not-stale to detect redundant transfer.
- *reset_status()* call should be added for many other scenarios. These scenarios change variable states, including a memory deallocation call and a GPU kernel call containing reduction. The memory deallocation sets the variable state to stale; the GPU kernel call containing reduction sets the state of the reduction variable on GPU to stale, if the reduction is performed in a way that only CPU has the final reduction result.
- Each memory transfer should be followed by *set_status()* to check any incorrect/redundant/may-redundant transfers.

To implement the above runtime checks, we could naively insert the checks for each read/write access. However, this implementation is inefficient and results in large runtime overhead. We introduce a number of techniques to reduce the overhead. These techniques are based on a simple fact that static and successive accesses to a variable on CPU (GPU) without any interfering GPU (CPU) kernel calls will not change its coherence state on CPU (GPU). These techniques are detailed as follows:

- Coherence checking (*read_check()* and *write_check()*) for GPU data is only necessary at the kernel boundary, because no CPU execution can change variable states on GPU during the kernel execution.
- The *read_check()* call for CPU data needs to be inserted only for the first-read accesses along some path from program entry or from each GPU kernel call; a similar rule is applied to *write_check()*.
- The *reset_status()* call to change the status of may-dead or must-dead GPU variables needs to be inserted only to the last-CPU-write accesses along some path from program entry or from each GPU kernel call. For may-dead or must-dead CPU variables, the *reset_status()* call needs to be inserted only at the kernel boundary (right after a GPU kernel call).

The above optimizations require the compiler to locate first-read, first-write, and last-write accesses. To find first-read/write accesses, we use an algorithm similar to the one in [23]. To find last-write accesses, we use a similar algorithm (Algorithm 2), but performing backward, all-path data flow analysis along some path from program exits or from the next kernel calls.

Algorithm 2 Last-Write Analysis

$OUT_{Write}(EXIT) \Leftarrow \emptyset$
$OUT_{Write}(n) \Leftarrow \bigcap_{s,s \in succ(n)} IN_{Write}(s)$
$IN_{Write}(n) \Leftarrow OUT_{Write}(n) + DEF(n) - KILL(n)$
$LAST_{Write}(n) \Leftarrow IN_{Write}(n) - OUT_{Write}(n)$

In addition to the above optimizations, if the first-CPU-read/write accesses reside in a loop and the loop does not

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.

contain any GPU kernel calls, the corresponding runtime checks can be moved before the loop. This method reduces additional profiling overhead. Furthermore, unlike the previous runtime coherence scheme [23], which optimizes the placement of coherence checks only for the CPU side, our scheme optimizes the placement for both CPU and GPU. Optimizing GPU-coherence-check placement allows us to detect additional redundant transfers, which was not possible in the previous schemes. We use Listing 3 as an example to further explain this opportunity. In Listing 3, GPU-write checks are inserted before each kernel call (line 3 and 6). However, *set_status()* in line 8 can not detect redundant memory transfer in line 8 because *check_write()* in line 6 changes the CPU state of *b* to stale at each iteration. If the write checks in line 3 and 6 are moved before the enclosing loop (line 1), *set_status()* in line 8 can detect redundant memory transfer in line 8. We generalize the above example and conclude that *write_check()* call for a GPU kernel can be moved before the enclosing loop if the two conditions hold: (i) the enclosing loop does not contains CPU codes accessing the target variable, and (ii) no memory transfer call for the variable exists before the *write_check()* call within the loop.

Figure 2 generally depicts our system, where user-compiler-runtime interactions are iterated as necessary. Like most of user-interactive debugging tools (e.g., gdb), our tool may result in bad user experience when applying to a GPU program with potentially long runtime. However, the user can always use smaller input problem with our tool to debug and verify programs. In summary, our memory-transfer verification and optimization mechanism can detect various types of incorrect/missing/redundant memory transfers. In combination with the kernel verification method in §III-A, our interactive profiling techniques allow programmers to easily pinpoint various errors and data transfer redundancy residing in the input GPU-directive programs. For debugging and optimization problems, these techniques make the problems manageable for directive-based GPU programming.

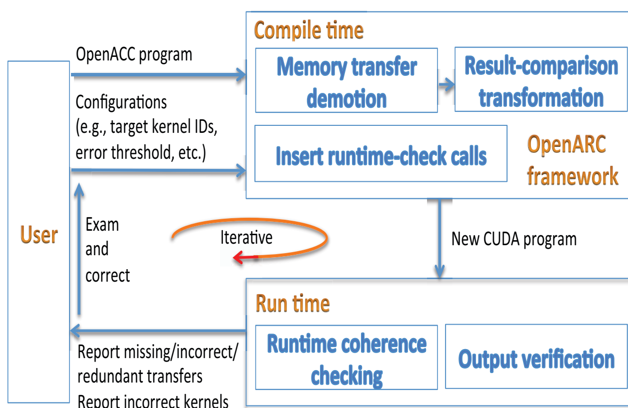

Figure 2: Interactive debugging/optimization system overview

```
1  ...
2  for( k=0; k<ITER; k++ ) {
3    check_write(a, GPU); check_read(b, GPU);
4    main_kernel0(a,b); //R/O(b), W/O(a)
5    reset_status(a, CPU, notstale);
6    check_write(b, GPU); check_read(a, GPU);
7    main_kernel1(a,b); //R/O(a), W/O(b)
8    memcpyout(b); set_status(b, CPU, notstale);
9  }
10 ...
11 check_read(b, CPU); ... //b is read here.
```

Listing 3: GPU code excerpt from JACOBI kernel with partially optimized memory transfers

C. Application Knowledge-Guided Debugging and Optimization

The above debug and optimization methods are general, and can be applied to any GPU kernel. However, our tools provide flexibility to customize the above methods based on application knowledge. Using debugging and optimization guided by application knowledge has a couple of benefits; (1) it avoids false positive debugging (e.g., the states of the target GPU kernel are different from those of the reference CPU execution, but the difference is acceptable); (2) it can accelerate debug and optimization by enforcing program invariance-based automatic bug detection, instead of relying on frequent interactions with users.

To implement (1), we introduce a set of directives to allow users to bound the values of the variables in the target GPU kernel. When the values of variables are different from the reference CPU execution, if they fall within the user-specified bound, the tools will ignore the difference and do not report them to the users. To implement (2), we introduce a debug assertion API. The users can implement the API by performing customized bug detection (e.g., establishing checksums and verifying them). The API will be inserted at the end of the kernel call to enable automatic error detection.

IV. Evaluation

In this section, we evaluate the coverage of the kernel verification scheme and memory-transfer verification/optimization scheme, and measure their overhead. Our work is based on an open source compiler framework called OpenARC [28], which provides extensible environment, where various performance optimizations and traceability mechanisms can be built for better debuggability and performance for challenging GPU programming.

A. Methodology

For evaluation, we selected twelve OpenACC programs from various application domains. These OpenACC programs include two kernel benchmark (*JACOBI* and *SPMUL*), two NAS Parallel Benchmarks (*EP* and *CG*), and eight Rodinia Benchmarks [27] (*BACKPROP*, *BFS*, *CFD*, *SRAD*,

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.

HOTSPOT, *KMEANS*, *LUD*, and *NW*). These programs are translated to output CUDA programs by OpenARC and then compiled using GCC 4.4.6 and NVCC 5.0 using *-O3* option. We executed compiled programs on a platform with Intel Xeon X5660 host CPUs and a NVIDIA Tesla M2090 GPU, using the largest available inputs that fit into the GPU memory. To get reliable results, we ran experiments multiple times (from 10 to 200 times, depending on the runtime variation). The reported results are the average values.

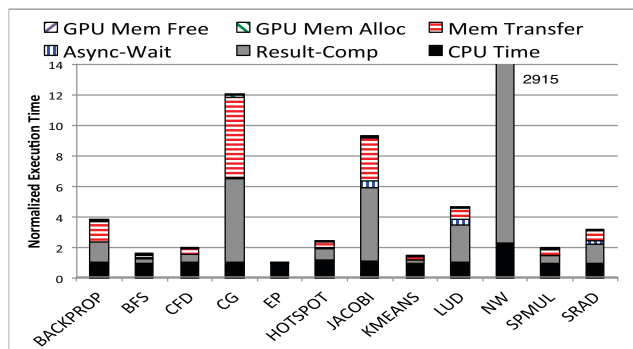

Figure 3: Breakup of execution time for kernel verification tests. The execution times are normalized to those of sequential CPU executions.

Table II: Summary for the kernel verification tests to detect race conditions caused by missing privatization or incorrect reduction recognition

Description	Count
Number of tested kernels	46
Number of kernels containing private data	16
Number of kernels containing reduction	4
Number of kernels incurring active errors	4
Number of kernels incurring latent errors	16

B. Kernel Verification

Figure 3 shows the breakdown of execution times when verifying all kernels in input programs. Within the figure, *Result-Comp* refers to the time to compare GPU kernel results against reference CPU results, *Mem Transfer* refers to memory transfers time between CPU and GPU, *Async-Wait* refers to the time that the host CPU has to wait to finish asynchronous GPU kernel executions and memory transfers, and *CPU Time* refers to the time spent for the host CPU work. The breakdown indicates that *Result-Comp* and *Mem Transfer* constitute most of the overhead. This is because each kernel verification always uses reference CPU data to avoid error contamination from previous GPU computation, and all data written by the kernels are examined to detect any errors potentially affecting program outputs.

Table II summarizes kernel verification tests to find kernel errors caused by race conditions. Those errors may occur if the GPU directive compiler fail to detect missing private variables or reduction variables. For those tests, private/reduction clauses are manually removed from the directive programs, and the directive compiler is configured to disable any automatic privatization or reduction recognition. In the table, active errors refer to the cases where race conditions actively alter program output, while latent errors do not change program behaviors. The proposed verification scheme successfully detected all active errors, but none of latent errors was detected. Further investigation found that the latent errors are related to private data. Even if a private variable is incorrectly translated as shared, the resulting race condition may not affect the program outputs if the compiler caches the intermediate values of the falsely shared variable; the race condition occurs when the value finally dumps back to the variable; however, the variable is no longer used after value dumping, hence incurring no visible errors. This shows that our verification scheme can effectively detect errors that directly affect kernel outputs. Theoretically, the proposed scheme can detect various active errors. In practice, the inconsistent floating-point precision between CPU and GPU can make error detection non-trivial and demand limited user involvement to pre-define error conditions.

```
– Copying b from device to host in update0
  (enclosing loop index = 1) is redundant.
– Copying b from device to host in update0
  (enclosing loop index = 2) is redundant.
...//repeated until the last k–loop iteration
```

Listing 4: Sample debugging messages for JACOBI kernel in Listing 3. *update0* refers to *memcpyout(b)* in line 8 in Listing 3.

C. Memory Transfer Verification and Interactive Optimization

To evaluate the performance of our memory transfer verification/optimization scheme, we first created unoptimized versions for tested programs. Then, the programs were iteratively optimized to find optimal transfer patterns according to our tool's suggestions. At each verification iteration, the tool provides three types of suggestions: (i) information on redundant memory transfers, (ii) error messages on missing/incorrect transfers, and (iii) warnings of may-redundant/may-missed transfers. Listing 4 shows sample debugging output that the tool generates for the *JACOBI* kernel in Listing 3, which says memory transfer of variable *b* from the device to the host in line 8 in Listing 3 is redundant except for the first iteration of the enclosing k-loop (line 2), indicating that the memory transfer can be deferred until the k-loop finishes (line 10). Then, user can modify data clauses in the input program according to the suggestions and rerun the program (iterate the verification steps as necessary).

Among the three types of suggestions, warning messages require special attention, since programmers have to verify correctness of the suggested changes. The may-redundant/

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.

may-missed transfers occur because of uncertainty in may-dead variables. In most cases, verifying deadness is relatively easy, since our tool informs which code section should be checked. If the whole data of the variable is written at that code, then the programmer can know the variable is guaranteed to be dead. In some cases, like NAS benchmark *CG* in Listing 1, where data are partially written, however, deciding deadness may become non-trivial. This challenge arises from the fact that the programmer has to check all the execution paths following the tool-suggested point to check whether written parts cover all the following reads.

Table III: Memory-transfer-verification performance. In the table, *# incorrect iterations* refers to the number of iterations where the tool suggests incorrectly, and *# uncaught redundancy* refers to the number of redundant memory transfers that the tool can not detect, compared to the manually optimized versions.

Benchmark	# total iterations	# incorrect iterations	# uncaught redundancy
BACKPROP	3	1	0
BFS	3	0	0
CFD	4	0	1
CG	2	0	0
EP	2	0	0
HOTSPOT	2	0	0
JACOBI	3	0	0
KMEANS	2	0	0
LUD	4	3	0
NW	2	0	0
SPMUL	3	0	0
SRAD	2	0	0

Table III summarizes the performance of the proposed scheme; in most cases, we successfully figure out optimal memory transfer patterns with up to four iterative verification steps. However, the table also indicates that the memory transfer verification scheme may falsely suggest in some cases (e.g., non-zero incorrect iterations in *BACKPROP* and *LUD*). Those cases occur when the compiler cannot resolve the relationship between (may-)aliased pointers. If the user follows the false suggestions, the resulting programs may be corrupted. However, the next verification step (i.e., kernel verification) is able to detect new errors caused by the previous incorrect suggestions and indicate the previous incorrect suggestions. (The memory transfer and kernel verification schemes complement each other.) Therefore, the user is still able to find optimal memory transfer patterns, even though intermediate wrong suggestions may unnecessarily prolong the iteration steps (e.g., *LUD*). The undetected redundant transfer in *CFD* is because current implementation locally optimizes the memory-transfer-checking mechanism proposed in §III-B.

Figure 4 shows the memory transfer verification overhead normalized to the execution times without the verification. The results indicate that the proposed verification scheme incurs negligible runtime overhead in most cases. The negative overheads mainly come from memory transfer time variation on PCI-e bus. The large performance variance in the PCI-e bus in combination with short execution time on GPU results in the negative overhead for those kernels.

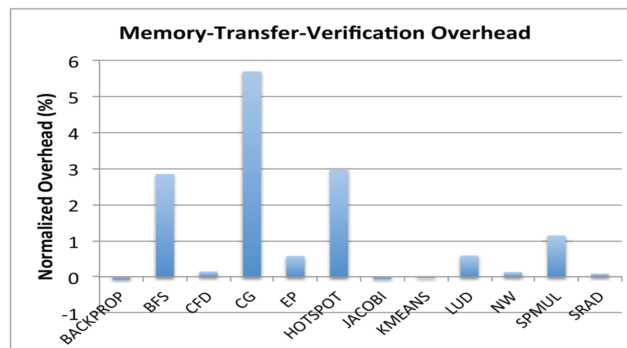

Figure 4: Memory-transfer-verification overhead normalized to no verification versions. The negative overheads are due to runtime variation in those cases with very short execution times.

V. Conclusions

We have presented novel techniques to debug and optimize programs for directive-based GPU programming. Our work is motivated by the challenges commonly seen in debugging and optimizing the directive-based GPU programs: even though those programming models provide very high-level abstraction, multiple stages of complex translations hidden by the directive compilers shift significant debugging and optimization burden to GPU program developers. The proposed, iterative profiling scheme offers an intuitive and synergistic environment, where programmers, compilers, and runtimes can interact with each other in an abstract manner to achieve productive GPU computing. We have implemented the proposed schemes on top of a new open source OpenACC compiler, called Open Accelerator Research Compiler (OpenARC). Evaluation using twelve OpenACC programs from various application domains demonstrates that our user-controlled kernel verification can detect all active errors affecting program outputs. In addition, our memory-transfer verification and optimization, combined with the kernel verification method, allows users to find optimal memory transfer patterns iteratively.

References

[1] A. Bland *et al.*, “Titan: 20-petaflop cray XK6 at oak ridge national laboratory,” in *Contemporary High Performance Computing: From Petascale Toward Exascale*, 1st ed., ser. CRC Computational Science Series, J. S. Vetter, Ed. Boca Raton: Taylor and Francis, 2013, vol. 1, p. 900.

[2] S. Matsuoka *et al.*, “TSUBAME2.0: The first petascale supercomputer in japan and the greenest production in the world,” in *Contemporary High Performance Computing: From Petascale Toward Exascale*, 1st ed.,

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.

ser. CRC Computational Science Series, J. S. Vetter, Ed. Boca Raton: Taylor and Francis, 2013, vol. 1, p. 900.

[3] X. Liao, Y. Lu, and M. Xie, “Tianhe-1A supercomputer: System and application,” in *Contemporary High Performance Computing: From Petascale Toward Exascale*, 1st ed., ser. CRC Computational Science Series, J. S. Vetter, Ed. Boca Raton: Taylor and Francis, 2013, vol. 1, p. 900.

[4] J. S. Vetter *et al.*, “Keeneland: Bringing heterogeneous GPU computing to the computational science community,” *IEEE Computing in Science and Engineering*, vol. 13, no. 5, pp. 90–95, 2011.

[5] P. Kogge *et al.*, “Exascale computing study: Technology challenges in achieving Exascale systems,” DARPA Information Processing Techniques Office, Tech. Rep., 2008.

[6] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski, “OpenMP for Accelerators.” in *IWOMP'11*, 2011, pp. 108–121.

[7] T. D. Han and T. S. Abdelrahman, “hiCUDA: High-level GPGPU programming,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, 2011.

[8] HMPP, “OpenHMPP directive-based programming model for hybrid computing,” [Online]. Available: http://www.caps-entreprise.com/openhmpp-directives/, (accessed Jan. 29, 2014).

[9] S. Lee and R. Eigenmann, “OpenMPC: Extended OpenMP programming and tuning for GPUs,” in *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*. IEEE press, 2010.

[10] A. Leung *et al.*, “A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction,” in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. ACM, 2010, pp. 51–61.

[11] OpenACC, “OpenACC: Directives for Accelerators,” [Online]. Available: http://www.openacc-standard.org, (accessed Jan. 29, 2014).

[12] PGI_Accelerator, “The Portland Group, PGI Fortran and C Accelarator Programming Model,” [Online]. Available: http://www.pgroup.com/resources/accel.htm, (accessed Jan. 29, 2014).

[13] S. Lee and J. S. Vetter, “Early evaluation of directive-based GPU programming models for productive Exascale computing,” in *the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE press, 2012.

[14] NVIDIA, “CUDA,” [Online]. Available: https://developer.nvidia.com/category/zone/cuda-zone, (accessed Jan. 29, 2014).

[15] AMD, “GPU PerfStudio 2,” [Online]. Available: http://developer.amd.com/tools-and-sdks/graphics-development/gpu-perfstudio-2/, (accessed Jan. 29, 2014).

[16] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam, “An experimental approach to performance measurement of heterogeneous parallel applications using CUDA,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. ACM, 2010, pp. 127–136.

[17] NVIDIA, “NVIDIA Visual Profiler,” [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler, (accessed Jan. 29, 2014).

[18] RogueWave, “TotalView graphical debugger,” [Online]. Available: http://www.roguewave.com/products/totalview.aspx, (accessed Jan. 29, 2014).

[19] Allinea, “Distributed Debugging Tool (DDT),” [Online]. Available: http://www.allinea.com/products/ddt/, (accessed Jan. 29, 2014).

[20] K. Spafford *et al.*, “The tradeoffs of fused memory hierarchies in heterogeneous architectures,” in *ACM Computing Frontiers (CF)*, Cagliari, Italy, 2012.

[21] I. Gelado *et al.*, “An asymmetric distributed shared memory model for heterogeneous parallel systems,” in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10. ACM, 2010, pp. 347–358.

[22] T. B. Jablin *et al.*, “Automatic CPU-GPU communication management and optimization,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. ACM, 2011, pp. 142–151.

[23] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, “Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12. ACM, 2012, pp. 33–42.

[24] T. B. Jablin *et al.*, “Dynamically managed data for CPU-GPU architectures,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. ACM, 2012, pp. 165–174.

[25] OpenCL, “OpenCL,” [Online]. Available: http://www.khronos.org/opencl/, (accessed Jan. 29, 2014).

[26] C. Dave *et al.*, “Cetus: A source-to-source compiler infrastructure for multicores,” *IEEE Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[27] S. Che *et al.*, “Rodinia: A benchmark suite for heterogeneous computing,” in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[28] “OpenARC: Open Accelerator Research Compiler,” [Online]. Available: http://ft.ornl.gov/research/openarc, (accessed Jan. 29, 2014).

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on September 03,2020 at 01:55:48 UTC from IEEE Xplore. Restrictions apply.