Automated Dynamic Analysis of CUDA Programs

Michael Boyer
University of Virginia
Dept. of Computer Science
Charlottesville, VA 22904
boyer@cs.virginia.edu

Kevin Skadron [*]
University of Virginia
Dept. of Computer Science
Charlottesville, VA 22904
skadron@cs.virginia.edu

Westley Weimer
University of Virginia
Dept. of Computer Science
Charlottesville, VA 22904
weimer@cs.virginia.edu

ABSTRACT

Recent increases in the programmability and performance of GPUs have led to a surge of interest in utilizing them for general-purpose computations. Tools such as NVIDIA's CUDA allow programmers to use a C-like language to code algorithms for execution on the GPU. Unfortunately, parallel programs are prone to subtle correctness and performance bugs, and CUDA tool support for solving these remains a work in progress.

As a first step towards addressing these problems, we present an automated analysis technique for finding two specific classes of bugs in CUDA programs: race conditions, which impact program correctness, and shared memory bank conflicts, which impact program performance. Our technique automatically instruments a program in two ways: to keep track of the memory locations accessed by different threads, and to use this data to determine whether bugs exist in the program. The instrumented source code can be run directly in CUDA's device emulation mode, and any potential errors discovered will be automatically reported to the user. This automated analysis can help programmers find and solve subtle bugs in programs that are too complex to analyze manually. Although these issues are explored in the context of CUDA programs, similar issues will arise in any sufficiently "manycore" architecture.

1. INTRODUCTION

The microprocessor industry has recently shifted from maximizing single-core performance to integrating multiple cores. Graphics Processing Units (GPUs) are notable because they contain many processing elements–up to 128 [12, p.73]–a level of multicore integration that is often referred to as "manycore." GPUs present a level of concurrency today that cannot be found in any other consumer platform. Although GPUs have been designed primarily for efficient execution of 3D rendering applications, demand for ever greater programmability by graphics programmers has led GPUs to become very general-purpose architectures, with fully featured instruction sets and rich memory hierarchies. Tools such as NVIDIA's CUDA have further simplified the process of developing general-purpose GPU applications.

[*]This work was conducted while Kevin Skadron was on sabbatical with NVIDIA Research.

Unfortunately, writing massively multi-threaded GPU programs is still difficult due to the risk of data races and the much greater likelihood and severity of resource contention. While a subtle synchronization bug in a CPU program with two threads may have an extremely low probability of occurring during execution, that same bug in a parallel program with thousands of threads will have a much higher probability of occurring. At the same time, testing and debugging tools that are common for desktop applications are often unavailable in a massively parallel environment.

Finding and solving a bug in a parallel program can also be extremely difficult for the same reasons. Most programmers can easily reason about a sequentially executing program and potentially even about a multi-threaded program with a small number of threads. However, as the number of threads increases to hundreds or even thousands, reasoning effectively about programs becomes much more challenging. Therefore, automated analyses are needed to aid programmers in finding and fixing bugs in their parallel programs.

We propose adapting software instrumentation techniques to CUDA programs. Traditional approaches (e.g., [4, 10, 11, 14]) are not directly applicable for various reasons: CUDA programs use barriers instead of locks, are massively multithreaded, have space and memory limitations, and are not written in strongly-typed high-level languages. We present an automated approach to instrumenting CUDA programs to detect and report certain classes of bugs at runtime. Programmers can use this approach to find correctness and performance bugs in code that may be too complex to analyze manually. As a proof of concept, we implemented our algorithm in a prototype tool that automatically detects race conditions and inefficient memory access patterns in CUDA programs. These two specific analyses were chosen because we consider them to be representative examples of two general classes of problems, synchronization errors and memory contention, that will occur more frequently as core counts and thread counts increase.

Even in situations where testing a program against a known correct output is possible, and thus the detection of race conditions is likely, our tool still provides a substantial benefit by pinpointing the exact variables involved. In situations where testing is more difficult, as is the case with memory access pattern problems that impact execution time but not correctness, our approach highlights optimization opportunities that programmers may not have been aware of. Our tool can help find bugs regardless of how obviously they manifest themselves in the program's behavior.

The rest of this paper is organized as follows. Section 2 motivates our interest in general-purpose computation on GPUs. Section 3 provides an overview of the CUDA system. Section 4 describes our automated approach to finding race conditions and Section 5 extends this approach to discover inefficient memory access patterns. Section 6 demonstrates the results of using the tool to analyze a real application. Section 7 discusses related work. Section 8 concludes and presents the future directions of this research.

2. GRAPHICS PROCESSING UNITS

Over the past few years, the performance of GPUs has been improving at a much faster rate than the performance of CPUs. For example, in early 2003, the most advanced GPU and CPU from NVIDIA and Intel, respectively, offered approximately the same peak performance. Four years later, NVIDIA's most advanced GPU provided six times the peak performance of Intel's most advanced CPU [12, p.1].

GPUs have been able to provide such rapid performance growth relative to CPUs by replicating simple processing elements (PEs), targeting throughput rather than single-thread performance, and devoting much less die area to caches and control logic. To further improve area efficiency, groups of PEs are harnessed together under SIMD control, amortizing the area overhead of the instruction store and control logic. Instead of large caches, GPUs cope with memory latency using massive multi-threading, supporting thousands of hardware thread contexts. Even if some threads are waiting to receive data from main memory, there will most likely be many other threads that can be executed while they wait.

2.1 General-Purpose Computation on GPUs

The tremendous growth in GPU performance and flexibility has led to an increased interest in performing general-purpose computation on GPUs (GPGPU) [7]. Early GPGPU programmers wrote programs using graphics APIs. This had the benefit of exposing some powerful GPU-specific hardware, but incurred the programming and execution overhead of mapping a non-graphics computation onto the graphics API and execution stack.

The two largest discrete GPU vendors, ATI and NVIDIA, recently released software tools designed to simplify the development of GPGPU applications. In 2006, ATI released Close-to-the-Metal (CTM) [15], which, as its name implies, is a relatively low-level interface for GPU programming that bypasses the graphics API. NVIDIA took a different approach with its tool, Compute Unified Device Architecture (CUDA) [12]. CUDA, supported on all NVIDIA GeForce, Quadro, and Tesla products based on the Tesla hardware architecture, allows programmers to develop general-purpose applications for the GPU using the C programming language, with some extensions described in Section 3.1. To date, CUDA has enjoyed more widespread use, and this work focuses specifically on programs developed using CUDA.

3. CUDA

NVIDIA's CUDA is a freely available language standard and development toolkit that simplifies the process of writing general-purpose programs for recent NVIDIA GPUs. CUDA is a software layer that supposes certain hardware abstractions. These are described in turn.

3.1 CUDA Software

CUDA consists of a runtime library and an extended version of C. The main abstractions on which CUDA is based are the notion of a *kernel* function, which is a single routine that is invoked concurrently across many thread instances; a software controlled scratchpad (which CUDA calls the "shared memory") in each SIMD core; and barrier synchronization.

CUDA presents a virtual machine consisting of an arbitrary number of *streaming multiprocessors* (SMs), which appear as 32-wide SIMD cores with a total of up to 512 thread contexts (organized into *warps* of 32 threads each). Kernels are invoked on a 2D *grid* that is divided into as many as 64K 3D *thread blocks*. Each thread block is mapped in its entirety and executes to completion on an arbitrary SM. Warps are multiplexed onto the SIMD hardware on a cycle-by-cycle granularity according to their execution readiness. Execution is most efficient if all threads in a warp execute in lockstep; divergence is handled with a branch stack and masking.

Each SM has a small, fast, software-controlled shared memory through which threads in a thread block may communicate. Threads may also read and write to much slower global memory, but thread blocks may be scheduled in any order onto the SMs and thread blocks run to completion before releasing their resources. This means that thread blocks should not try to communicate with each other except across kernel calls, as deadlock may occur otherwise.

A kernel runs to completion before a subsequent kernel may start, providing a form of global barrier. Within a thread block, arbitrary communication through the shared memory is allowed, but scheduling of warps is arbitrary and data races must therefore be prevented by executing an intra-thread-block barrier. Note that shared memory is private to each thread block; even thread blocks that happen to map to the same SM cannot access the shared memory of previous or co-resident thread blocks. Inter-thread-block communication must therefore occur across kernel calls.

CUDA's extensions to the C programming language are fairly minor. Each function declaration can include a *function type qualifier* that specifies whether the function will execute on the CPU or the GPU, and if it is a GPU function, whether it is callable from the CPU. Also, each variable declaration in a GPU function can include a *variable type qualifier*, which specifies where in the memory hierarchy the variable will be stored. These type qualifiers are similar to, for example, the local and global qualifiers for pointers in distributed memory systems (e.g., [6]). Finally, any call to a GPU function from within a CPU function must include an *execution configuration*, which specifies the grid and thread-block configuration and the amount of shared memory to allocate in each SM. Finally, kernels have special thread-identification variables automatically defined to allow threads to differentiate themselves and work on separate parts of a data set.

3.2 CUDA Hardware

There is nothing in the CUDA specification that prevents compilation of CUDA programs for non-GPU platforms, but NVIDIA so far only supports CUDA for its Tesla architecture, which encompasses the GeForce 8-series and recent Quadro GPUs and the Tesla GPU Computing product line. The flagship products provide 16 SMs, each consisting of 8 PEs. Once a kernel is launched, a hardware scheduler assigns each thread block to an SM with sufficient spare capacity to hold the entire thread block. If multiple (small) thread blocks fit onto a single SM, they will execute concurrently but cannot communicate or even be aware of the existence of their co-resident thread blocks.

Each thread is completely independent, scalar, and may execute arbitrary code and access arbitrary addresses. Memory access is more efficient if warps access contiguous data, and the benefits of SIMD execution are only realized if threads stay in lockstep. Since thread blocks of a single kernel call usually behave equivalently, we assume for simplicity that all programs we analyze contain a single thread block.

4. CORRECTNESS ANALYSIS

Compared to earlier GPGPU approaches, CUDA greatly simplifies the process of developing general-purpose programs for the GPU. CUDA allows the user to write nearly standard C code for a single thread, and execute that code concurrently across a multidimensional domain. Still, transitioning from developing single-threaded programs to developing multi-threaded programs introduces new classes of potential programming errors–this is true for any platform. Since the number of threads in a CUDA program can be so much larger than in a CPU program, these errors are potentially more likely to manifest themselves in CUDA. Thus, programmers need to pay special attention to ensuring the correctness of CUDA programs. Here we focus on race conditions, a specific type of correctness bug.

4.1 Race Conditions

CUDA programs, like any parallel program, are susceptible to *race conditions*, in which arbitrary behavior results when multiple unsynchronized threads write to and possibly read from the same memory location. Figure 1 shows a simple CUDA function that contains a race condition. Each thread executing this function first determines its own thread identifier as well as the total number of threads, then writes its identifier to shared memory, and finally reads the value written by its neighboring thread and writes it to global memory. A race condition exists because the value output by each thread in line eight depends on the order in which it is executed with respect to its neighboring thread. Since there is no explicit synchronization performed by this function, the behavior of the program is non-deterministic.

Explicit synchronization is needed to restore determinism. In CUDA, synchronization is accomplished using the function *syncthreads*, which acts as a barrier or memory fence. When a thread calls this function, it must wait for all other threads within its thread block to call the function before it can proceed to its next instruction. In the example discussed above, adding a call to syncthreads between lines seven and eight would prevent the race condition and remove the non-determinism.

```
1  extern __shared__ int s[];
2
3  __global__ void kernel(int *out) {
4    int id = threadIdx.x;
5    int nt = blockDim.x;
6
7    s[id] = id;
8    out[id] = s[(id + 1) % nt];
9  }
```

Figure 1: A simple CUDA program containing a race condition on lines seven and eight. Multiple threads execute the code in `kernel()` simultaneously.

A naïve approach to preventing race conditions is to add a call to syncthreads after every instruction that accesses memory. There are two significant problems with this approach. First, adding numerous calls to syncthreads can significantly degrade performance. Second, adding a call to syncthreads can potentially cause deadlock. Specifically, a program will deadlock if a syncthreads call is added inside of a conditional block that is not executed by all threads.

4.2 Automated Race Condition Detection

For simple programs, such as the one in Figure 1, race conditions can be detected by manually analyzing the source code. For more complex programs, this manual analysis quickly becomes infeasible. We have developed a tool that instruments CUDA source code to automatically detect race conditions. The tool operates as follows:

1. The program source code is parsed and converted to an intermediate representation that retains knowledge of CUDA-specific built-in declarations and type qualifiers. Our prototype uses the CIL framework [9].

2. The intermediate representation is transformed and instrumented. These transformations are only applied to functions declared as `__global__` or `__device__`, indicating that the functions will run on the GPU, and to accesses to variables declared as `__shared__`, indicating that the variables are stored in shared memory and accessible by all threads within a block.

3. The instrumented representation is converted back to CUDA's dialect of C, with CUDA-specific type qualifiers and annotations preserved.

The code added to check for race conditions contains declarations for two global bookkeeping arrays, which keep track of the number of read and write accesses made by all threads to each location in shared memory since the last synchronization point. It also contains two bookkeeping arrays local to each thread, which keep track of the accesses made only by that thread. After each instruction that accesses shared memory, code is added to update the bookkeeping arrays and then check for race conditions. In order for a given thread t to know that a race condition exists at a given shared memory location i, the following three conditions must be true: location i must have been both read from and written to[1]; at least one of the accesses to i must have

[1]Note that we are only checking for read-after-write (RAW) and write-after-read (WAR) hazards. Future implementations will check for write-after-write (WAW) hazards as well.

come from a thread other than t; and at least one of the accesses to i must have come from t. Any thread that detects a race condition sets a global flag; after synchronizing, one of the threads checks the flag and generates the appropriate output.

The amount of shared memory allocated to a function is specified when the function is called, not when it is defined. Since our tool is only guaranteed to have access to the definition of the instrumented function, we require that the user manually specify the amount of shared memory allocated. The instrumented code can be compiled and run directly. For our prototype tool, the code can only be run in device emulation mode and not on the actual GPU device since the instrumented code reports errors using the `printf` function, which cannot be called from the GPU. Section 6 discusses the performance overhead due to this emulation requirement.

4.3 Analysis Example

In the example function shown in Figure 1 earlier, we noted that a race condition existed at the expression on line eight. Running the function through the automated instrumentation tool and executing the resulting code using CUDA's device emulation mode generates the following output:

```
// RAW hazard at expression:
#line 8
  out[id] = s[(id + 1) % nt];
```

In other words, the automated instrumentation tool successfully detects and reports the race condition. Adding a call to syncthreads between lines seven and eight and re-running the automated analysis does not report a race condition, as we would expect. A more substantial example of the race condition analysis is presented in Section 6.

5. PERFORMANCE ANALYSIS

Parallelizing an application only makes sense if it significantly improves some other important aspect of the application, such as its performance. As we have seen earlier, maximizing the performance of a CUDA program generally requires the use of shared memory. Thus, it is important for programmers to understand the factors that may impact the performance of shared memory.

5.1 Bank Conflicts

Shared memory is located on-chip, making much faster to access than global memory, especially since global memory is not cached. In fact, accessing shared memory can be as fast as accessing a register, depending upon the memory access pattern of the threads. Physically, the shared memory is divided into 16 banks, with successive words in memory residing in successive banks. If all 16 threads in each half-warp access different banks, then the memory accesses will in fact be as fast as register access. If any threads access the same bank, then those accesses are serialized, decreasing the aggregate throughput of the program [12, p.56]. As the numbers of cores in many-core processors continue to increase, memory structures will need to be increasingly banked for scalability. Thus, we expect this type of analysis to be general enough to apply to many future architectures.

To demonstrate the performance impact of bank conflicts, we developed the simple CUDA program shown in Figure 2.

```
extern __shared__ int mem[];

__global__ void k(int *out, int iters) {
  int min, stride, max, i, j;
  int id = threadIdx.x;
  int num_banks = 16;
  char cause_bank_conflicts = 0;

  if (cause_bank_conflicts) {
    min = id * num_banks;
    stride = 1;
    max = (id + 1) * num_banks;
  } else {
    min = id;
    stride = num_banks;
    max = (stride * (num_banks - 1))
      + min + 1;
  }

  for (j = min; j < max; j += stride)
    mem[j] = 0;
  for (i = 0; i < iters; i++)
    for (j = min; j < max; j += stride)
      mem[j]++;
  for (j = min; j < max; j += stride)
    out[j] = mem[j];
}
```

Figure 2: CUDA program used to measure the impact of bank conflicts.

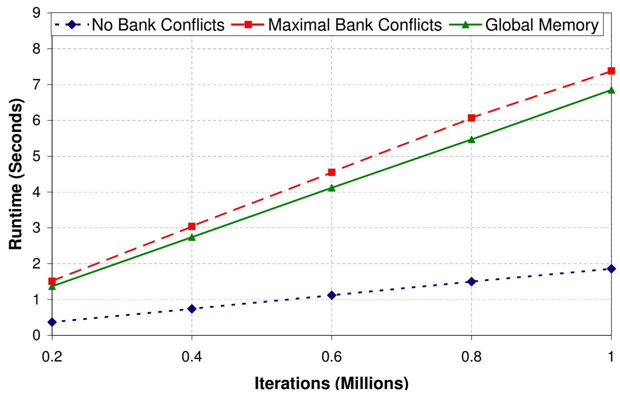

Figure 3: Performance impact of bank conflicts in the CUDA program shown in Figure 2.

Each thread in the program executes for a certain number of iterations. During each iteration, the threads access shared memory in a strided fashion. Depending on the value of the stride, this program can exhibit no bank conflicts or the maximum number of bank conflicts. Figure 3 shows the performance of the program in these two cases as a function of the number of iterations executed. The performance with maximal bank conflicts is approximately four times worse than the performance with no bank conflicts. In fact, the performance with maximal bank conflicts is worse than the same program rewritten to use the slow, off-chip global memory. Clearly, bank conflicts can have a significant impact on performance and must be taken into consideration when optimizing CUDA programs.

5.2 Automated Bank Conflict Detection

For simple programs, programmers may be able to manually analyze the memory accesses to detect bank conflicts.

However, as with the correctness analysis, this manual bank conflict detection quickly becomes infeasible as the complexity of the program increases. To address this problem, we have extended the tool presented earlier to also instrument CUDA programs to automatically measure bank conflicts. The instrumentation process is similar to the process described in Section 4.2; the only difference is in the specific instrumentation code added in the second step.

For this analysis, a declaration is added for a global array to store the addresses accessed by each thread. After every shared memory access, code is added for each thread to update its entry in the array. After synchronizing, one thread uses the global array to compute the memory bank accessed by each thread and determine the existence and severity of bank conflicts. This information is then reported to the user.

5.3 Analysis Example

We can use this automated instrumentation to analyze the example function shown earlier in Figure 2. In this specific case, we would expect the output of the instrumented program to depend on the value of `cause_bank_conflicts` specified in line seven. With the flag set to false, the instrumented program generates the following output for the first iteration of the first for loop:

```
// No bank conflicts at expression:
#line 21
  mem[j] = 0;
```

With the flag set to one, the instrumented program generates the following output:

```
// Bank conflicts at expression:
#line 21
  mem[j] = 0;
// Bank:       0  1  2  3  4  5  6  7  8 ...
// Accesses: 16
```

The automated instrumentation correctly determines that, with the flag set to true, this program exhibits the maximum number of bank conflicts during each iteration. This detailed output is potentially much more useful than a binary output indicating only whether or not bank conflicts occurred, since the performance impact depends on the severity of the conflicts. In addition, the tool output pinpoints the instruction causing the bank conflicts. Automatically providing programmers with this analysis allows them to easily determine whether they can improve the performance of their programs by adjusting shared memory access patterns.

6. RESULTS

We used the tool described in the previous sections to automatically analyze a real application, `scan` [3], which is included in the CUDA Standard Developer Kit. `Scan` implements the all-prefix-sums operation [2], which is a well-known building block for parallel computations, and is over 400 lines of CUDA code. The program uses explicit synchronization to avoid race conditions and defines a specific macro for avoiding bank conflicts. Thus, we can selectively remove synchronization statements in the program and observe if the tool successfully detects the race conditions that arise, and we can also enable the bank conflict avoidance macro and observe if the tool successfully detects the resulting lack of bank conflicts.

Code Version	Execution Environment	Average Runtime	Slowdown
Original	GPU	0.4 ms	
Original	Emulation	27 ms	
Instrumented (race conditions)	Emulation	324 ms	12.0x
Instrumented (bank conflicts)	Emulation	71 ms	2.6x

Table 1: Performance impact of emulation and instrumentation. Slowdown is relative to the performance of the original application in emulation mode.

6.1 Correctness Analysis

We analyzed `scan` using the race condition analysis presented in Section 4. The original version of `scan` is properly synchronized using three syncthreads calls. Analyzing the unmodified version, the instrumented application correctly reported no race conditions. We then generated three modified versions of `scan`, each of which had one of the three calls to syncthreads removed. Analyzing these modified versions of `scan`, each instrumented application correctly reported race conditions after the removed synchronization point, but not beyond the next synchronization point.

6.2 Performance Analysis

We also analyzed `scan` using the bank conflict analysis presented in Section 5. The original version of `scan` is designed to have a small but non-zero number of bank conflicts. Analyzing the unmodified version, the instrumented application correctly reported relatively mild bank conflicts (two threads accessing the same bank) for about one third of the memory accesses. Enabling the macro for avoiding all bank conflicts had an unexpected result: the number of statement with bank conflicts doubled and the severity of bank conflicts increased significantly, with some statements causing 16-way bank conflicts. This counter-intuitive result was confirmed with extensive manual analysis of the program's behavior. This example shows the utility of this analysis: with relatively little user effort we were able to determine that a program which we assumed exhibited efficient memory access patterns in fact has significant room for improvement in its memory performance.

6.3 Performance Impact of Instrumentation

As noted earlier, the instrumented source code produced by our prototype tool must be run in CUDA's device emulation mode rather than on the actual GPU hardware. Table 1 shows the performance of the original application, executing natively and in emulation mode, and the performance of the two instrumented versions of the application, both executing in emulation mode. As expected, emulation mode is significantly slower than native execution. When debugging a CUDA application, we expect that most CUDA programmers will run their applications in emulation mode for the improved debugging capabilities, such as the ability to call `printf`. Thus, when computing the performance overhead of the instrumentation code, we compare the runtime of the instrumented code against the original application running in emulation mode. We can see that the race condition detection degrades performance by a factor of 12, while the bank conflict detection degrades performance by a much more modest factor of 2.6. The performance degradation of these analyses is small enough to make them feasible for larger applications. Popular memory-debugging tools such as Purify can degrade performance by a factor of 40 or more [10].

7. RELATED WORK

A number of dynamic analyses exist that instrument programs to detect concurrency errors. The Eraser tool is a popular example: it transforms programs to explicitly track the set of locks held on each access to a shared variable [14]. Any variable not consistently protected by a lock is subject to race conditions. Unfortunately, CUDA programs are not amenable to such an analysis because the syncthreads memory barrier used for concurrency control is not a lock.

Our instrumentation approach is similar to work by Mellor-Crummey [8] and the LRPD test [13] in that we declare shadow arrays to track memory references. However, the work by Mellor-Crummey is specific to programs utilizing fork-join parallelism and the LRPD test speculatively parallelizes `doall` loops in Fortran programs; CUDA programs employ neither fork-join parallelism nor `doall` loops.

The CCured [10], Valgrind [11] and Purify [4] tools all instrument programs to verify memory safety and resource usage. They all incur run-time overhead on the instrumented program, and Valgrind and Purify are typically more expensive to use than our transformation. Most importantly, however, they address the orthogonal problem of memory safety. CUDA programs would certainly benefit from such tools, although in some cases the tools cannot be applied directly. It is not clear, for example, how to effectively extend CCured's bounds checking to concurrent programs.

Abadi et al. have proposed static type systems for race detection in Java programs [1]. However, they focus on particular idioms for synchronization not present in CUDA programs, they require non-trivial programmer annotations, and take advantage of Java's type safety. Other recent approaches work well on C programs but still require explicit locks (e.g., [5]). Our approach requires almost no annotations and works on barrier-synchronized, weakly-typed, C-style CUDA programs.

8. CONCLUSIONS

For future work, we intend to address some of the limitations of the current prototype. We also plan to add support for the detection of additional types of bugs. For example, this approach could be extended to detect other correctness bugs, such as out-of-bounds array accesses, or other performance bugs, such as inefficient global memory coalescing [12, p.50].

As the performance of manycore processors continues to increase rapidly, programmer interest in harnessing their computational power for general-purpose computations will intensify. Tools such as CUDA simplify the previously arcane process of developing these GPGPU applications. However, it is still easy for programmers to make mistakes that compromise the correctness or performance of their applications, and it can still be difficult to find and solve those mistakes.

We have presented a technique for automatically instrumenting CUDA programs to find and report, with essentially no programmer input, race conditions and inefficient shared memory access patterns. This automated analysis increases the likelihood that programmers will find and solve these specific problems, helping them increase both the correctness and efficiency of their CUDA programs.

9. ACKNOWLEDGMENTS

This work was supported in part by NSF grant IIS-0612049. We would like to thank the anonymous reviewers for their helpful comments.

10. REFERENCES

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[2] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Computer Science, Carnegie Mellon University, 1990.

[3] M. Harris. Parallel prefix sum (scan) with CUDA. Technical report, NVIDIA Corporation, 2007.

[4] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter Usenix Conference*, 1992.

[5] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *Computer Aided Verification*, 2007.

[6] B. Liblit and A. Aiken. Type systems for distributed data structures. In *Principles of Programming Languages*, 2000.

[7] D. Luebke and G. Humphreys. How GPUs Work. *IEEE Computer*, 40(2):96–100, 2007.

[8] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, 1991.

[9] G. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, 2002.

[10] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Principles of Programming Languages*, 2002.

[11] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, 2007.

[12] NVIDIA. CUDA Programming Guide. Technical report, NVIDIA Corporation, 2007. Version 1.1.

[13] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[15] M. Segal and M. Peercy. A performance-oriented data parallel virtual machine for GPUs. In *SIGGRAPH*, 2006.