# Performance Degradation Analysis of GPU Kernels

Jinpeng Lv[1], Guodong Li[2], Alan Humphrey[2] and Ganesh Gopalakrishnan[2]

[1] Electrical & Computer Engineering, University of Utah, Salt Lake City, UT
[2]School of Computing, University of Utah, Salt Lake City, UT

## 1 Motivation

Hardware accelerators (currently Graphical Processing Units or GPUs) are an important component in many existing high-performance computing solutions [5]. Their growth in variety and usage is expected to skyrocket [1] due to many reasons. First, GPUs offer impressive energy efficiencies [3]. Second, when properly programmed, they yield impressive speedups by allowing programmers to model their computation around many fine-grained threads whose focus can be rapidly switched during memory stalls. Unfortunately, arranging for high memory access efficiency requires developed computational thinking to properly decompose a problem domain to gain this efficiency. Our work currently addresses the needs of the CUDA [5] approach to programming GPUs. Two important classes of such rules are *bank conflict avoidance* rules that pertain to CUDA shared memory and *coalesced access* rules that pertain to global memory. The former requires programmers to generate memory addresses from consecutive threads that fall within separate *shared memory* banks. The latter requires programmers to generate memory addresses that permit coalesced fetches from the global memory. In previous work [6], we had, to some extent addressed the former through SMT-based methods. Several other efforts also address bank conflicts [7, 8, 4]. In this work, we address the latter requirement—detecting when coalesced access rules are being violated.

**Table 1.** Performance Comparison between Bank Conflicts and Memory Coalescing Violations:100000 iterations of: Read an int from shared memory; Write back to global memory

| #Threads | 1024 | 512 | 256 | 128 | 64 | 32 | 16 |
|---|---|---|---|---|---|---|---|
| B.K. | 0.035587 | 0.035589 | 0.017809 | 0.010054 | 0.008529 | 0.008532 | 0.008304 |
| M.C.V. | 0.417909 | 0.417933 | 0.208980 | 0.104515 | 0.052281 | 0.026188 | 0.013126 |

**Table 2.** Execution times in seconds with and without Memory Coalescing: 10000 iterations of: Read an int from global memory; Increment; Write back to global memory

| #Threads | $2^{24}$ | $2^{20}$ | $2^{16}$ | 1024 | 256 | 32 | 16 |
|---|---|---|---|---|---|---|---|
| Coalescing(s) | 14.210834 | 0.890957 | 0.055718 | 0.001061 | 0.000605 | 0.000580 | 0.000583 |
| No-Coalescing(s) | 128.674240 | 8.349093 | 0.520744 | 0.007005 | 0.001779 | 0.000994 | 0.000602 |

The motivation for detecting memory coalescing violations is provided by Table 1 and Table 2. These results demonstrate through experimental runs on actual CUDA hardware, that the efficiency of CUDA programs can be significantly influenced by performance flaws. Table 1 shows memory coalescing violations can take upto 10 times the runtime of bank conflicts, which leads to the conclusion that memory coalescing violations have more effects than bank conflicts on performance. Besides, Table 2 shows hand-written kernels that follow/violate the coalescing rules can differ by an order of magnitude in performance. Although these are small examples, they capture how performance degradations such as non-coalesced and bank-conflicting accesses may get strewn across one's code, with each occurrence giving rise to the same degree of performance degradation. Since GPU are *streaming architectures*, they are memory bandwidth-bound for all but the most trivial of problems. This immediately tells us that the additive effect of each performance degradation will severely degrade the whole kernel. Besides, another reason we utilize these small examples is to eliminate the effects of other performance flaws as much as possible and therefore we are able to measure the influences accurately. Our contribution is to ferret out each such performance degrading occurrence not easily discerned by conventional tools or by code inspection.

This paper analyzes different cases of memory coalescing violations and derives an approach to detect such performance flaws. Our approach has been implemented in our tool PUG [6]—a symbolic verifier for CUDA kernels based on Satisfiability Modulo Theories (SMT [9]).

## 2 Details of Bank Conflict and Coalescing

To provide context for our problem, we briefly introduce some basic concepts of GPUs and CUDA C. More details can be obtained through [2].

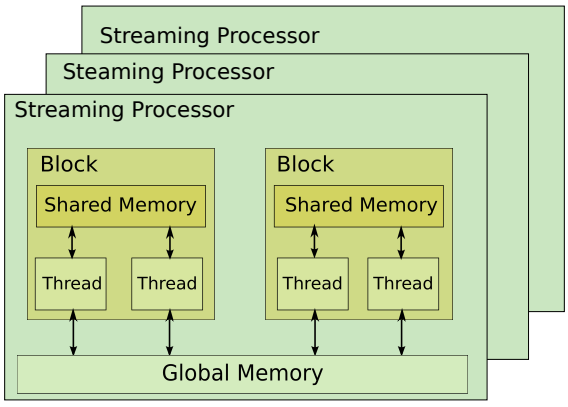


**Fig. 1.** CUDA Memory Model

GPUs consist of a number of streaming multiprocessors (**SM**), with each running up to thousands of parallel threads concurrently. These threads are

organized in **block** granularity and each block is further divided into **warps** which each consist of 32 consecutive threads. Correspondingly, a half-warp consists of 16 consecutive threads. Each warp of threads are co-scheduled on a SM and then execute the same instruction in a given clock cycle (SIMD fashion). **__syncthreads()** guarantees that every thread in the block has completed instructions prior to __syncthreads() before the hardware will execute the next instruction on any thread. **Global memory** is the memory allotted to the entire GPU. All threads can read and write to global memory. **Shared memory** is a fast work-space for threads within a block. It is worth mentioning that the shared memory is at least 100 times faster than global memory. The CUDA memory access model is shown in Fig. 1.

Under memory coalescing, **global memory** accesses by a half-warp of threads are combined into a single, wide memory access. This results in a bandwidth improvement of up to 16 times per half-warp—a figure that easily adds up. Given that each global memory access takes about $400-600$ clock cycles, the difference is between $6400-9600$ clock cycles without coalescing down to $400-600$ clock cycles with coalescing.

To ensure memory coalescing, the memory request for a half-warp must satisfy the following conditions[1]:

- Type Consistency Rule: the size of the words accessed by the threads must be 4, 8, or 16 bytes.
- Sequential Access Rule: threads must access the words in sequence: The $k^{th}$ thread in the half-warp must access the $k^{th}$ word.

Note that not all threads are required to participate. If the half-warp does not meet these requirements, 16 separate memory transactions are issued.

## 3 Detection of Memory Coalescing Violations

Memory coalescing violations are either *type consistency* or *sequential access* violations. A type consistency violation occurs when the size of the words accessed is not 4, 8, or 16 bytes (not one of *int, int2, int4* or their equivalents). For example, a kernel prototype declared as

```
__global__ void violation(int3 * data, int n, int iter)
```

results in a coalescing violation. Type checking, when feasible, achieves this check. Sequential access violations occur when the $k^{th}$ thread **in the half-warp** does not access the $k^{th}$ word in global memory, as in

```
1 __global__ void violation(int * data, int n, int iter)
2 { int idx = threadIdx.x;
3     if(idx%2==0)
4       for (int i = 0; i < iter; ++i)
5         data[idx+1] = data[idx] + 1;
```

[1] These conditions are required by GPUs with Compute Capability 1.0 and 1.1

```
6     else
7       for (int i = 0; i < iter; ++i)
8         data[n-idx] = 1;
9   data[idx] = data[idx+16];  }
```

At Line 5, thread *idx* reads *data* at *idx* and then writes *data* at *idx+1*. The *read* access strictly follows the sequential access rule. However, the *write* access breaks the sequential access rule in that thread *idx* writes *data* at *idx+1*. A similar violation occurs at Line 8. It is interesting to consider whether memory accesses at Line 9 violates the sequential access rule. Since memory $data[idx + 16, \cdots, idx + 31]$ accessed by *data* by a half-warp of threads is still in an aligned 64-byte memory segment, such an access is still a sequential access. As a result, there is no sequential access violation for both *read* and *write* accesses at Line 9. Based on the above observations coupled with the fact that memory coalescing is organized for every 16 threads, sequential access violation can be detected by checking this assertion across the entire kernel:

$$Index\ of\ array\ \%16\ ==\ thread\ ID\ \%16 \tag{1}$$

Our approach to detect memory coalescing violations is as follows:

- Detect Type Consistency Violations
- Detect Sequential Access Violation:
  - Each access to global memory is encoded as an *unsat* problem, also considering the path conditions leading to the access. Essentiall we check for $Index_{array}$ %16 $\neq$ threadID %16.
  - We then obtain a disjunction of all access encodings: $\vee(array_i \pmod{16} \neq threadId \pmod{16})$
  - If *unsat*, all memory accesses are coalesced; else we obtain a violation witness.

Our procedures for encoding CUDA C to SMT are described in detail in [6] and the workflow of PUG is illustrated in Fig. 2.

## 4 Experiment

Our proposed approach has been implemented in PUG [6]. To evaluate the effectiveness and efficiency of our approach, experiments involving different cases of violations (or non-violations) have been conducted, as shown in the following kernel functions.

```
void __global__ kernel1 (int *data)
{
 int idx = threadIdx.x+1;
 data[idx] = data[idx] + 16;
}
//return sat, violation: offset of index is 1
```

```
void __global__ kernel2 (int *data)
{
 int idx = threadIdx.x;
 data[idx] = data[idx+16] + 16;
}
// return unsat, no violation:
// idx+16 % 16=idx
```

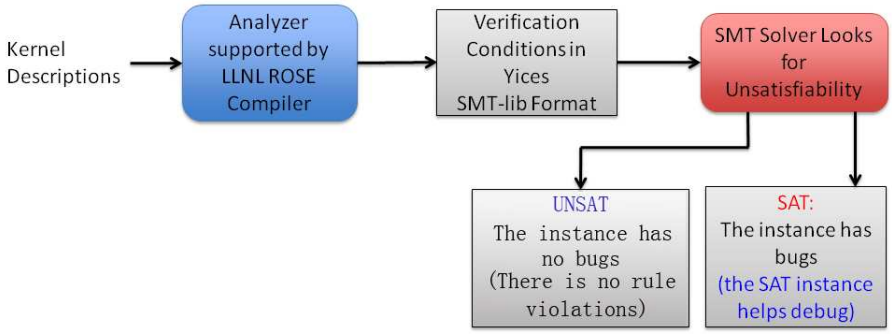


**Fig. 2.** The Architecture of PUG

```
void __global__ kernel3 (int *data)
{
 int idx = threadIdx.x;
 data[n-idx] += 16;
}
//return sat,violation:
//∃idx, n-idx % 16 ≠ threadIdx.x
```

```
void __global__ kernel4 (int *data)
{
 int idx = threadIdx.x+1;
 data[idx-1] += 16;
}
// return unsat, no violation
// idx-1=threadIdx.x+1-1=threadIdx.x
```

In addition to the above case-by-case violations (non-violations), complicated kernel functions (e.g. containing multiple violation cases, as the one shown in kernel 2) are also run by PUG. PUG catches all these violations (or non-violations) efficiently with a negligible runtime: $< 0.1$. The efficiency is derived from the fact that we only need to consider indices of *array* involved. All other information in the kernel function is discarded, which significantly reduces computations performed by the SMT solvers.

## 5 Conclusion and Future Work

The paper proposed an approach to detect memory coalescing violations and experimental results illustrate the efficiency and effectiveness of our proposed method. With the corporation of our proposed method, PUG now can detect race conditions, bank conflicts and memory coalescing violations efficiently.

Since there are not much work focusing on functional equivalence checking for CUDA kernels (concurrent programs), an attractive direction to evolve PUG is to incorporate functional equivalence checking whose major work is to check the equivalence between CUDA kernels and optimized CUDA kernels.

## References


1. Borkar, S., Chien, A.A.: The future of microprocessors. Communications of the ACM 54(5), 67–77 (2011)
2. Cuda programming guide version 3.0.
3. Dally, W.: (2010), keynote at SC 2010
4. Eddy Z. Zhang, Yunlian Jiang, Z.G., Tian, K., Shen, X.: On-the-fly elimination of dynamic irregularities for gpu computing. In: the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (2011)
5. Kirk, D.B., mei W. Hwu, W.: Programming Massively Parallel Processors. Morgan Kauffman (2010)
6. Li, G., Gopalakrishnan, G.: Scalable smt-based verification of gpu kernel functions. In: FSE in ACM SIGSOFT (2010)
7. Mai Zheng, Vignesh T. Ravi, F.Q., Agrawal, G.: Grace: A low-overhead mechanism for detecting data races in gpu programs. In: ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP) (2011)
8. Michael Boyer, K.S., Weimer, W.: Automated dynamic analysis of cuda programs. In: Third Workshop on Software Tools for MultiCore Systems (2008)
9. Satisfiability Modulo Theories Competition (SMT-COMP). `http://www.smtcomp.org/2009`