Heterogeneous Data Race Exhibiting Programs

Akash Panda[1], Divyanshu Bhandari[2], Rishabh Ravindra Meshram[2], Shubham Sharma[2]

***Problem Statement*— Evolution of a tighter integration of CPUs and GPUs are emerging with features such as shared memories, coherence, and atomics to reduce the computing overheads and to increase the efficiency of the entire system. Programming languages allow programmers to exploit these architectures for productive collaboration between CPU and GPU threads.[2] In process of doing it, heterogeneous data races created a big problem in the whole scenario. We have come up with programs exhibiting heterogeneous data races, which can be used as a benchmark for heterogeneous data race detection algorithms.**

I. Introduction

A. Motivation

Heterogeneous system architecture consists of two components namely CPU and GPU, and they share the same memory address space called Unified Memory. Data resides in the memory space is read or manipulated by both the processor so we need a proper synchronization between them for the correct execution of the program.

In massively parallel program, we use different levels of shared memory to communicate between threads. Recent version of CUDA support global shared memory and scoped synchronization. Some of such instructions are __threadfence() , __threadfence_block() , atomicAdd() , atomicAdd_block(), etc. __synthreads() also act like a scope synchronization primitive, as it synchronizes within a block. Recently CUDA has also added __syncwarp() which is a warp synchronization primitive. Hence, it a possible to write a racey program that is composed of atomics, but wrongly scoped . [7]

If we are able to detect data race in heterogeneous system architecture we can improve the reliability of the programs in a collaborative environment. This requires tools that detect data races. In order to design such tools, we need to come up with a set of programs containing such kind of data races. By analysing these programs we can accordingly design algorithms/tools to detect data races.

B. Achievement

We took some massively parallel workloads (Page rank, Graph connectivity, Graph colouring and Coalesced Transpose) and implemented them using CUDA programming model. We also implemented work stealing, where each block used block scoped atomics while performing its own work, but would use system wide atomics while stealing some other block's work. This introduction lead to a data race. The same memory location being protected by two different scopes in two different cases.

[1]M.Tech(Research), Department of CSA IISc Bangalore, India akashpanda@iisc.ac.in

[2]M.Tech, Department of CSA IISc Bangalore, India {bdivyanshu,shubhamsharma,rishabhm}@iisc.ac.in

II. Background

A. GPU Hardware

GPU hardware differs from CPU hardware fundamentally.

Memory(global, constant, shared), Streaming Multiprocessors(SMs) and Streaming Processors(SPs) form the basic blocks of a GPU Hardware. An array of SMs, each of which has N cores make up the GPU. This forms the key aspect that allows the scaling of the processor. Addition of more SMs to the device would make the GPU process more tasks, only if we have enough parallelism in the task.

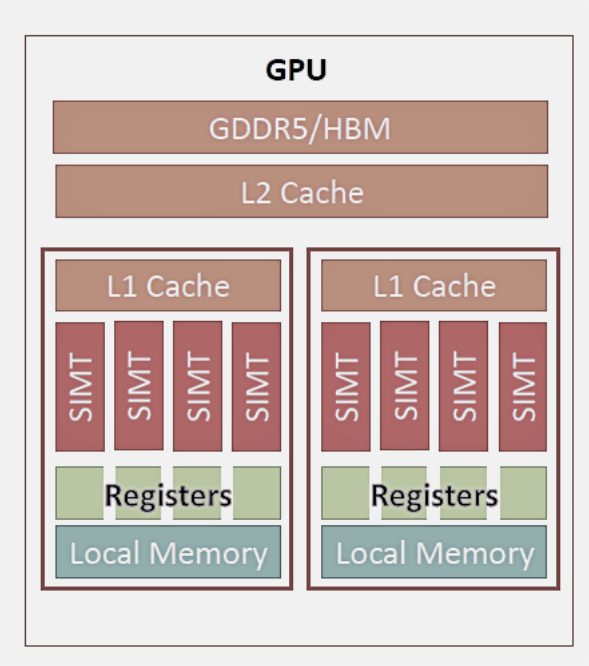

Fig. 1. HSA Cache Hierarchy

B. Hierarchy

GPUs are designed to run thousands of threads concurrently, hence are massively data parallel. CUDA programming model arranges the threads in hierarchy. Threads are grouped into warps. Multiple warps form a thread block. Multiple thread blocks form a kernel. Massively data parallel algorithms are being targeted using GPU programming model, where multiple threads work on the same code but different data.

C. CUDA Programming Model

Thousands of threads are executed massively parallel to achieve high throughput. The GPU programming model incorporates hierarchy to help programmers organize threads. Blocks are a collection of the threads. A block is a unit of execution on the Heterogeneous System Architecture(HSA) component. Compute unit performs execution. An HSA component can have one or more compute units. A block is partitioned into warps to match GPU's execution width. Warps execute on SIMD units. Each thread has a set of its registers and private memory. Threads within a block can be executed in an extended SIMD (single instruction, multiple data) style. That is, threads are gang scheduled in chunks called warps. [1] Figure 1 shows an example of HSA cache hierarchy as seen by the threads.

D. Heterogeneous Systems

HSA combines and exploits the capabilities and features of the CPUs and GPUs for the users who like to exploit systems more than the traditional usage scenario.

CPU/GPU interactions can be of following types:[5]

1) Pinned host memory - CPU memory that the GPU can directly access.
2) Command buffers - The buffers written by the CUDA driver nd read by GPU to control its execution.
3) CPU/GPU synchronization: how the GPU's progress is tracked by CPU.

Shared virtual memory(SVM), system-wide atomics and scoped atomics are the latest features of heterogeneous system architecture and programming models which is important for collaboration. Host and device processor share the same virtual address range via the **Shared Virtual Memory**. It improves performance in collaborative programs, as this would now not require memory transfer mechanisms. With the introduction of SVM, memory coherence became an important issue. [2] **System-wide atomics** provides support to a variety of CPU-GPU collaborative patterns by enabling fine grain synchronization across devices.[4] **Scoped atomics** provides support of a lower granularity of atomics, that the collaborative threads can agree upon. For example, if we can make sure that the communication is not required beyond the block, we can use a block synchronization primitive or a block scoped atomic operation.

E. Memory Model

Parent and child share the same global and constant memory storage, but have distinct local and shared memory.
Global memory : Parent kernel and child kernel have coherent access to global memory, but with weak consistency guarantee.
Constant memory : Constants cannot be modified, even between parent and child launches. Value of all __constant__ variables must be set by host before the launch.
Shared memory are private to a thread block. Behaviour is undefined when an object in these locations is being accessed outside of the scope within which it belongs to.
Local memory is private to the executing thread, and is not visible to threads outside the thread.
Table in Figure 2 shows the scope and lifetime of different variables.

	Scope	*Lifetime*
Register	1 thread	Thread
Local	1 thread	Thread
Shared	1 block	Block
Global	All threads + host	Host Allocation
Constant	All threads + host	Host Allocation

Fig. 2. Memory model

III. Related Works

Derek R. Hower *et al* presented Heterogeneous-race-free Memory Models, where they have embraced scoped synchronization with a new class of memory consistency model that add scoped synchronization to data-race free memory models like those of c++ and java.

Marc S. Orr *et all* introduced Remote Scope promotion. They have discussed about Static local sharing and dynamic local sharing. Static local sharing is that shared data is partitioned statically and each block has its own copy of data that they have to work on, whereas dynamic partitioning means, partitioning takes place while execution of threads. Each block picks up more work when it completes a set of work. In static partitioning, we can keep the shared data in block scope, whereas in dynamic partitioning we have to keep data in global scope. Work stealing is an example of static as well as dynamic partitioning. In work stealing, the data is partitioned statically. But if some block takes more time in completing its work and some other block has finished its work, it can pull work remaining in other block's queue.

Juan *et al* presented CHAI(Collaborative Heterogeneous Applications for Integrated-architectures), which is a suite of collaborative heterogeneous benchmarks that use to maximum advantage the heterogeneous architectures, and spread over a range of collaborative patterns. They concluded CHAI as much needed for evaluation of emerging heterogeneous systems.[2]

Matthew D. Sinclair *et al* presented "HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs", where they combined set of micro-benchmarks from various papers and came up with a set of GPU micro-benchmarks that uses various kinds of synchronization. [3]

We want to come up with a set of programs that lack various kinds of synchronization and would actually exhibit data race in them. Our work will focus on studying various data races in heterogeneous architecture programs and come up with programs actually having those. These set of programs would be helpful for algorithms detecting heterogeneous data races. In this way, our work differs from the above mentioned papers, which presented workloads for heterogeneous systems.

IV. Project Work

We used Nvidia CUDA Toolkit 9.2 and driver version Driver Version: 396.37 for this project. Tesla P40 GPU was used during the course of project. Our project was to write programs that would have heterogeneous data race in it arising because of usage of wrong scoped synchronization primitives and atomics. We started with writing simple program using system wide atomics. Removing them lead to a data race in the programs. We then started understanding CUDA programming model and what types of programs exploit the GPU architecture. We then understood the different programs being written in CHAI workload. The programs provided us with an idea of how to exploit massively parallel algorithms using CUDA programming model. We also understood the programs in the workload and learn about different synchronization primitives in GPUs. Then we identified some parallel workloads. We zeroed on to Page rank, Graph connectivity, Coalesced Transpose and Graph colouring problem. We implemented them using CUDA programming model.

A. Implemention on Different Workloads

Below there are four different problems which possesses high level of parallelism that we have used as workloads.

Page Rank:

Graph Coloring: Graph coloring problem deals with assigning colors to the vertices such that no two adjacent vertices get the same color. However, coloring an arbitrary graph is known to be NP-Hard problem. So, we need parallel algorithms to exploit utilities of multi-core hardware systems(GPUs). Each core can independently process a subtask and speedup the overall performance. In this parallel implementation, we divide the whole task of coloring the graph into smaller subtasks. Further, each block of threads is responsible to complete a subtask independently where each thread of the block process a vertex(coloring of the vertex) in parallel with other threads. Also, we implement work stealing that is if a block completes its subtask earlier then that block can steal the work of other block. This implementation is inspired by Pingfan Li et al. paper"High Performance Parallel Graph Coloring on GPGPUs".

Graph Connectivity: Graph Connectivity is well known problem which shows a given graph is connected or not. Above implementation of graph connectivity also calculated the distance of each node from a given node which makes easier to implement other problems on graphs(like Shortest Distance,Minimum Spanning Tree, Connected Components etc.). Implementation of traversal(search) algorithm is inspired by the Siddharth Srinivasa paper "Accelerating Large Graph Algorithms on the GPU 2007". Which helps our algorithm to traverse faster.

Coalesced Transpose: This technique to find transpose of a matrix make use of tiling and shared memory to avoid the large strides through global memory which gives a performance improvement over the naive method to find transpose of a matrix.The algorithm used is inspired by "Optimizing Matrix Transpose in CUDA by Greg Ruetsch and Paulius Micikevicius(January 2009)".

Input Format: All the above algorithm works on the same input file of 9th DIMACS Implementation Challenge. These algorithms can use any input file which is of format is as follows:
(Beginning of the file)
Nodes Edges Source_node

A0 B0
A1 B1
...

C0 D0
C1 D1
...
Each tuple (Ai, Bi) represents one node. Each tuple (Cj, Dj) represents one edge. Thus, the file contains the list of nodes, followed by the list of edges. Ai indicates the position where the edges of node i start in the list of edges. Bi means the number of edges of node i. Cj is the node where edge j terminates (i.e., the head of the edge).

B. Dividing the Global Data-Structure

We introduced a global queue as the shared data structure and then divided tasks into multiple queues for each block processing its own queue elements.

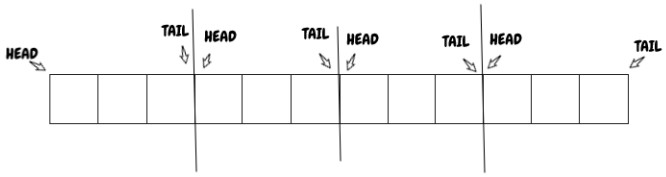

Fig. 3. **Global Queue Shared Among Blocks**

As shown in the fig 3 we divided queue into the multiple smaller queues which are accessed by blocks. each queue has its head and tail pointers which are implemented in the programs such that blocks can access respective queues for completion of their tasks.

C. Work Stealing Implementation

Then we went a step further to implement work stealing. Dynamic global sharing lead us to implement work stealing. Whenever a block finished its work, it would search for free work on others queues which has not being processed yet and would pick up those for processing. This process leads to **work stealing**. While implementing work stealing, we used two different scopes of atomics in two different cases. An atomic function performs a read-modify-write atomic operation in global or shared memory.[5] When a

block is working in its own queue, it would use block wise atomics(i.e. atomicAdd_block(), etc.), but once a block is stealing other's work, it would use device wise atomics(i.e. atomicAdd(), etc.). These two different usage of scopes lead to a data race when the same location being accessed in two different scopes by different threads in different blocks.

atomicAdd_block() implies that the instruction is atomic only with respect atomics from other threads in the same thread block.
atomicAdd() reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address.

V. Future work and conclusion

In this course project, we looked into different workloads for heterogeneous systems and then came up with a set of programs that have the heterogeneous data race in them.

These programs can be used as a set of inputs for Heterogeneous Data race detection algorithms.

The set of programs can be found at https://gitlab.com/akashpanda/e0243-project-heterogeneous-races.git

References

[1] "HSA Programmers Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writers Guide, and Ob-ject Format (BRIG) Version 1.0 Provisional," HSA Founda-tion, Spring 2013.

[2] J. Gmez-Luna, I. El Hajj, L.-W. Chang, V. Garcia-Flores, S. Garcia de Gonzalo, T.Jablin, A. J. Pea, W.-M. Hwu. Chai: Collaborative Heterogeneous Applications for Integrated-architectures. In Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2017.

[3] Matthew D. Sinclair, Johnathan Alsop, Sarita V. Adve. HeteroSync: A benchmark suite for fine-grained synchronization on tightly coupled GPUs. In proceedings of IEEE International Symposium on Workload Characterization (IISWC), 2017.

[4] W.-m. W. Hwu, Heterogeneous System Architecture: A New Compute Platform Infrastructure. Morgan Kaufman, 2015.

[5] "The Cuda Handbook - A Comprehensive Guide to GPU Programming", Nicholas Wilt, 2013.

[6] Marc S. Orr, Shuai Che, Ayse Yilmazer, Bradford M. Beckmann, Mark D. Hill, David A. Wood. Synchronization Using Remote-Scope Promotion . In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015.

[7] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, David A. Wood . Heterogeneous-race-free Memory Models . In Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS) 2014.