JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 26 No. 9, pp. 1-12, September 2021
https://doi.org/10.9708/jksci.2021.26.09.001

A Tool for On-the-fly Repairing of Atomicity Violation in GPU Program Execution

Keonpyo Lee*, Seongjin Lee*, Yong-Kee Jun**

*Student, Dept. of AI Convergence Engineering, Gyeongsang National University, Jinju, Korea
*Professor, Dept. of AI Convergence Engineering, Gyeongsang National University, Jinju, Korea
**Professor, Dept. of Aerospace Software Engineering, Gyeongsang National University, Jinju, Korea

[Abstract]

In this paper, we propose a tool called ARCAV (Atomatic Recovery of CUDA Atomicity violation) to automatically repair atomicity violations in GPU (Graphics Processing Unit) program. ARCAV monitors information of every barrier and memory to make actual memory writes occur at the end of the barrier region or to make the program execute barrier region again. Existing methods do not repair atomicity violations but only detect the atomicity violations in GPU programs because GPU programs generally do not support lock and sleep instructions which are necessary for repairing the atomicity violations. Proposed ARCAV is designed for GPU execution model. ARCAV detects and repairs four patterns of atomicity violations which represent real-world cases. Moreover, ARCAV is independent of memory hierarchy and thread configuration. Our experiments show that the performance of ARCAV is stable regardless of the number of threads or blocks. The overhead of ARCAV is evaluated using four real-world kernels, and its slowdown is 2.1x, in average, of native execution time.

▸ **Key words**: Concurrent program, GPU program, Concurrency error, Atomicity violation, On-the-fly repairing

[요 약]

본 논문은 GPU 프로그램의 메모리의 상태 및 접근사건과 배리어 위치 정보를 감시하고, 실제 메모리 쓰기를 배리어 영역 종료 직전에 발생시키거나 배리어 영역을 재수행시켜 원자성 위배를 수행 중에 수리하는 도구인 ARCAV (Atomatic Recovery of CUDA Atomicity violation)를 제시한다. 기존의 연구들은 Lock과 Sleep 명령어를 사용하여 원자성 위배를 진단 및 수리하도록 구현되었기 때문에 지원되는 명령어와 동기화 기법이 CPU (Central Processing Unit) 프로그램과 다른 GPU (Graphics Processing Unit) 프로그램에는 적용될 수 없었고, GPU 프로그램에서는 원자성 위배의 탐지에 대한 연구만 수행되었다. 제안하는 ARCAV는 GPU 프로그램의 실행모델에서 수행될 수 있도록 설계되어 스레드 구성과 메모리 계층에 무관하게 실세계에서 발생한 원자성 위배를 대표하는 네 가지 패턴의 원자성 위배를 실시간으로 탐지하고 수리할 수 있다. 실험 결과 동시에 실행되는 스레드 개수와 구성에 무관하게 일정한 오버헤드를 보였다. 원자성 위배를 프로그램 수행 중에 실시간으로 탐지하고 수리하기 위해 소요되는 오버헤드는 네 개의 실세계 GPU 커널에서 실험되었고, 원본 프로그램 대비 평균 2.1배의 수행시간으로 동작하였다.

▸ **주제어**: 병행 프로그램, GPU 프로그램, 동시성 오류, 원자성 위배, 자율수리

• First Author: Keonpyo Lee, Corresponding Author: Yong-Kee Jun
*Keonpyo Lee (gnurvy2@gnu.ac.kr), Dept. of AI Convergence Engineering, Gyeongsang National University
*Seongjin Lee (insight@gnu.ac.kr), Dept. of AI Convergence Engineering, Gyeongsang National University
**Yong-Kee Jun (jun@gnu.ac.kr), Dept. of Aerospace Software Engineering, Gyeongsang National University
• Received: 2021. 08. 13, Revised: 2021. 09. 17, Accepted: 2021. 09. 17.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

원자성 위배(Atomicity Violation)는 병행프로그램에서 발생하는 동시성 오류의 일종으로, 원자적으로 수행되어야 하는 구간이 다른 스레드에 의해 위배되는 현상이다[1-2]. 동시성 오류들은 비결정적인 결과를 발생시켜 프로그램이 실패하게 만들 수 있지만, 복잡한 스레드 인터리빙으로 인해 탐지가 어려워 개발단계에서 모든 오류를 탐지하고 수리하는 것이 불가능하고, 탐지된 오류를 올바르게 수정하는 것도 어렵다[1-2]. 원자성 위배의 자율수리 기술은 개발 과정에서 제거되지 않고 프로그램에 내재되어있는 원자성 위배를 수행 중에 탐지하고 이를 수리하여 프로그램이 실패하지 않도록 한다[3-8].

기존의 연구들은 전통적인 CPU 프로그램의 원자성 위배를 수리하도록 개발되었다[3-8]. GPU 프로그램은 스레드 스케줄링, 메모리 구조, 동기화 기법이 CPU 프로그램과는 다르기 때문에 기존의 기법으로 원자성 위배를 올바르게 수리할 수 없다. 기존 기법들은 GPU 프로그램에서 지원되지 않는 Lock과 Sleep 명령이 필수적이거나 체크포인트(Checkpoint)와 롤백(Rollback)이 사용된다. lock과 sleep 명령을 직접 구현하여 사용하면 교착상태나 많은 시간 오버헤드를 발생시킬 수 있다[9]. 체크포인트와 롤백은 개별 스레드의 체크포인트 및 롤백이 불가능하고, 큰 오버헤드가 발생하기 때문에 많은 스레드와 공유메모리 접근을 사용하는 GPU 프로그램에는 적용하기 어렵다.

본 논문은 배리어의 위치와 공유메모리 상태 및 접근을 감시하여 두 배리어 사이의 영역에서 원자성 위배가 발생하였을 때 잘못된 결과 값이 메모리에 반영되지 않도록 하는 도구인 ARCAV를 제시한다. CARR은 영역 내 메모리 접근을 사본메모리로 우회시키고, 오류 발생을 탐지하여 실제 메모리에 반영하거나, 영역 내 연산을 재수행 시킨다. CARAR은 Lock과 Sleep 명령을 사용하지 않아 GPU 프로그램에 적용할 수 있다. 원자성 위배가 발생할 수 있는 네 가지 접근사건 조합의 합성 프로그램과 실제 프로그램 네 개 모두 수리할 수 있었다. 시간 오버헤드는 실제 프로그램들에서 탐지와 수리에 평균 2.1배로 측정되었다.

본문의 구성은 다음과 같다. 2, 3, 4장에서는 각각 연구배경, 관련연구, 문제점을 설명한다. 5, 6장에서는 제안하는 기법의 구조와 동작 예시와 구현방법을 설명한다. 7장에서는 실험 및 결과분석을 정리하고, 마지막 5장에서는 본 논문의 결론과 향후 연구되어야 할 과제를 제시한다.

II. Preliminaries

이 장에서는 연구를 이해하기 위해 필요한 사전지식으로 GPU 프로그램, 원자성 위배와 자율수리, 기존 기법 소개 및 해결하고자 하는 문제점에 대한 내용을 설명한다.

2.1. GPU Program

GPU 프로그램은 높은 데이터 병렬화 성능을 달성하기 위해 사용하는 병행 프로그램이다[10]. 이를 위해 GPU는 CPU보다 적은 수의 명령어를 처리할 수 있는 수 백에서 수 천개의 코어로 이루어져 있고, 이들은 Streaming Multiprocessors (SMs)에 의해 그룹화되어 관리된다. GPU의 메모리는 독립적으로 운용되며, 많은 메모리 접근을 위해 넓은 대역폭의 메모리 버스를 지니고 있다[10].

GPU 프로그램은 SIMT(Single Instruction Multiple Thread)이라 불리는 실행모델로 수행된다[10-11]. SIMT 실행 모델에서 스레드들은 Fig. 1(a)와 같이 그리드(Grid), 블록(Block), 워프(Warp) 단위로 그룹화 된다. 워프는 SM에 의해 직접 관리되는 스레드들로, 프로그램 카운터를 공유하여 동시에 스케줄링되는 최소 단위이다. 블록은 워프의 집합으로 프로그래머가 지정한 개수의 스레드로 구성된다. 그리드는 커널(Kernel)이라 불리는 GPU 프로그램을 수행하기 위해 할당된 모든 스레드의 집합을 의미한다.

CPU 병행 프로그램과는 다르게 SIMT 실행 모델에서의 공유메모리는 Fig. 1(a)처럼 계층 구조를 이룬다. shared memory는 같은 블록 내의 스레드가 공유하는 메모리로 블록마다 할당되고, global memory는 커널 내의 모든 스레드가 공유할 수 있다. 동기화는 블록 단위의 배리어, 간단한 원자적 연산을 가능하게 하는 Atomic Functions, 그리고 Nvidia의 Pascal 및 Volta 아키텍처부터 지원되는 Cooperative Groups를 통한 그리드 및 Warp 단위의 동기화가 사용된다[11].

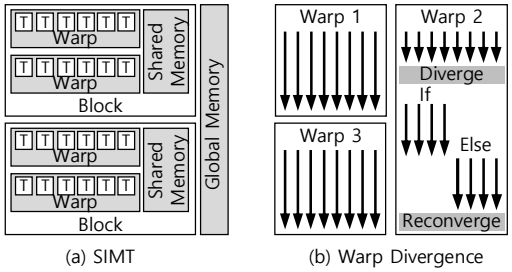

Fig. 1. SIMT Execution Model

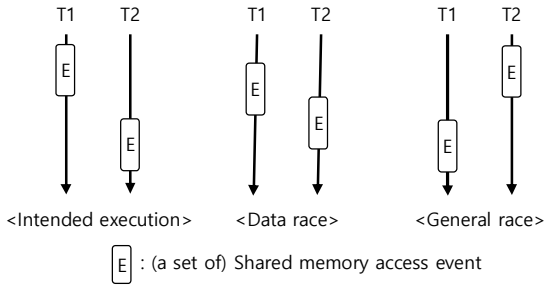

Fig. 2. Race Conditions

2.2. Atomicity Violation

원자성 위배는 병행프로그램에서 발생하는 동시성 오류(Concurrency Error)인 경합 조건(Race Condition)의 일종이다[1-2]. Fig. 2와 같이 경합 조건은 같은 공유메모리에 원자적으로 접근되어야 하는 접근사건들 사이에 다른 스레드의 접근사건이 수행되어 데이터의 연속성이 위배되는 Data Race, 스레드들의 공유메모리 접근사건 혹은 그 집합이 프로그래머의 의도와 다른 순서로 실행되는 General Race로 구분된다[1-2]. 원자성 위배는 경합 조건 중 프로그램 수행 중에 프로그래머의 의도에 부합하지 않는 결과를 발생시킨 Data Race를 의미한다.

GPU에서 발생하는 원자성 위배는 워프 단위 스케줄링에 의해 Inter-Warp와 Intra-Warp 유형으로 나뉜다[12]. Inter-Warp 원자성 위배는 서로 다른 워프에 속한 스레드들이 발생시키는 위배이다. Intra-Warp 원자성 위배는 Warp 단위로 스케줄링 되는 쓰기 명령어로 인해 발생하거나 Fig. 1(b)와 같이 Warp 내의 스레드들이 조건문에 의해 분기되었을 때 이를 순차적으로 수행하도록 하는 Warp Divergence에 의해 발생한다. Table 1은 CUDA (Compute Unified Device Architecture) 프로그램의 간단한 원자성 위배 예시이다.커널을 수행하는 각 스레드들은 공유변수 a[n]에 읽기 및 쓰기 연산을 수행한다. 하지만 공유메모리 배열 a의 인덱스가 스레드 ID를 나타내는 threadIdx.x를 32로 나눈 나머지가 되기 때문에 32개 이상의 스레드가 수행될 때 같은 공유메모리 주소에 두 개 이상의 스레드가 접근하게 되어 의도하지 않은 값인 1을 반환하게 될 수 있다.

2.3. On-the-fly Repairing of Atomicity Violation

원자성 위배를 탐지하기 위한 기법들이 개발되었지만 모든 결함을 탐지하는 것은 불가능하다[3-8]. 기존의 탐지 기법들은 많은 거짓 알람(False Alarm)을 발생시키거나 디버깅에 많은 시간과 비용이 소모되며, 잘못된 패치로 인해 오류가 남아있거나 다른 오류를 유발할 수 있다[1, 2, 13, 14]. 이러한 이유로 개발 과정에서 제거되지 못한 원자성 위배가 존재 할 수 있으며, 원자성 위배의 자율수리 기술은 개발 과정에서 제거되지 않은 잠재적인 원자성 위배들로 인한 프로그램 실패를 방지할 수 있다. 원자성 위배의 자율수리는 프로그램 수행 중(On-The-Fly)에 스레드 인터리빙을 제한하거나 변경하여 원자성 위배가 발생할 확률을 감소킨다[3-8].

Table 1. Example of Atomicity Violation

```
1. __global__ void increment(int *shared)
2. {
3.         int i = threadIdx.x % 32;
4.         shared[i]++;
5. }
```

원자성 위배의 자율수리는 진단과 수리의 시점에 따라 Forward Recovery와 Backward Recovery 방식으로 분류할 수 있다[15]. Forward Recovery 방식은 원자성 위배가 예상될 시 스레드의 수행을 지연(Stall)시켜 다른 스레드 접근 이후에 수행하도록 한다. Backward Recovery는 Checkpoint와 Rollback를 통해 원자성 위배가 발생 시 이전으로 프로그램 흐름을 되돌린다. 하지만 Backward Recovery는 Checkpoint와 Rollback을 커널 내에서 사용할 수 없거나 개별 스레드에 사용할 수 없어 GPU 프로그램에 구현이 불가능하다[16]. Forward Recovery 방식은 탐지할 수 있는 오류의 종류가 Backward Recovery에 비해 적지만, 낮은 오버헤드로 수리가 가능하다[15]. 많은 스레드와 공유메모리 접근을 사용하는 프로그램의 경우 오버헤드가 스레드와 공유메모리 접근 수 에 비례하여 증가하기 때문에 Forward Recovery 방식이 유리하다.

III. Related Work

Forward Recovery 방식의 원자성 위배 자율수리 기법은 수리 방식에 따라 Region Treatment와 Access Treatment 방식으로 분류될 수 있다. Region Treatment는 원자 영역을 식별 또는 설정하고 원자 영역 단위로 진단 및 수리하는 방식으로 Krena 등의 연구[3], Lucia 등의 연구[4], Yu 등의 연구[5], Zhu 등의 연구[6]의 예가 있다. Access Treatment 방식은 원자 영역을 식별하지 않고 개별 접근사건 단위로 진단 및 수리되는 방식으로 Yu 등의 연구[7]와 Zhang 등의 연구[8]가 해당된다.

Krena 등의 연구[3]는 수행 전에 각 공유메모리에 사용되는 Lock을 식별하고, 수행 중에 블록 혹은 엑세스에

Lock/Unlock이나 Sleep을 삽입하여 수리한다. 하지만 Lock/Unlock을 사용하는 경우 원본 프로그램에 명시적 Lock이 존재해야 하고 데드락 등의 새로운 문제를 발생시킬 소지가 있으며, Sleep을 사용하는 경우 수리가 실패할 수 있다. 시간 오버헤드는 논문에서 실험된 결과에 따르면 4개의 프로세서와 15개의 스레드를 가지는 프로그램을 대상으로 하였을 때 3.75배의 수행 시간이 소모된다.

Lucia 등의 연구[4]는 프로그램 수행 전에 코드를 일정한 크기로 분리하고, 다른 스레드에서 수행되는 조각들이 같은 공유메모리에 접근하고 있을 시 스레드 동작을 지연시킨다. 이 기법은 분리한 코드영역의 크기에 따라 원자성 위배의 자율수리 확률과 오버헤드가 증가한다. 이 연구에서는 9개의 실제 프로그램에 기법을 적용하여 원본 프로그램의 원자 영역이 코드 조각으로 구분된 암시적인 원자 영역에 포함되어 원자성 위배가 방지되는 비율을 확인한 결과 99.8%의 원자성 위배가 방지되었다. 하지만 이 연구는 시간 오버헤드에 대해서는 분석하지 않았다.

Yu 등의 연구[5]는 프로그램의 반복적인 시험 수행으로 얻어진 안전한 인터리빙과 공유메모리 접근정보를 기반으로 프로그램에 트랜잭션을 할당하고, 같은 공유메모리에 접근하는 트랜잭션들이 동시에 실행되지 않도록 한다. 하지만 트랜잭션 영역을 할당하기 위해 가능한 많은 인터리빙이 오류 없이 수행되도록 시험이 설계되야 한다. 이 연구는 14개의 실제 프로그램을 대상으로 자율수리를 시험하였고 그 중 11개의 오류를 수리할 수 있었다. 시간 오버헤드는 0.6%로 낮았지만, 대규모 스레드와 공유변수를 사용하는 프로그램을 대상으로는 실험하지 않았다.

Zhu 등의 연구[6]은 슬라이딩 윈도우 기법을 기반으로 원자성 위배를 탐지 및 수리하는 기법인 TolerSW를 제시한다. TolerSW는 최근 실행된 메모리 명령을 기록하는 윈도우를 각 스레드에 할당하고 다른 스레드와 비교하여 원자성 위배를 탐지한 후, 늦게 시작된 스레드를 지연시키는 방법으로 수리한다. 이 연구는 16코어 CPU에서 10개의 벤치마크 프로그램을 대상으로 실험하여 평균 2% 미만의 오버헤드만이 발생하였으나, TolerSW의 멀티코어 프로세서에 하드웨어적 구현을 추가해야 하는 단점이 있다.

Access Treatment에 해당하는 Yu 등의 연구[7]와 Zhang 등의 연구[8]는 프로그램의 모든 공유변수 접근에 ID를 부여하고, 반복 시험 수행을 통해 안전한 인터리빙을 기록한다. 이를 기반으로 프로그램 수행 시 접근사건 발생 직전에 미리 수행되어야 하는 접근사건의 발생 여부를 확인하고, 위반 시 스레드를 지연시킨다. 두 연구의 가장 큰 차이점은 Yu 등의 연구[7]는 코드 라인 단위로 접근사건을 인식하지만 Zhang 등의 연구[8]는 모든 동적 접근 명령어를 인식할 수 있다는 것이다. 두 연구 모두 트랜잭션을 생성하는 Yu 등의 연구[5]과 마찬가지로 많은 시험 수행이 필요하다. Zhang 등의 연구[8]의 경우 실제 프로그램에서 발생한 동시성 오류 사례 35건을 대상으로 자율수리를 시험하였고, 모든 사례를 수리하였다. 일반적인 프로그램에서 1% 미만의 적은 시간 오버헤드 발생을 장점이 있지만, 매트릭스 연산으로 공유메모리 액세스가 많은 프로그램에서는 1300% 이상의 시간 오버헤드가 발생한다.

IV. Issues on Repairing Atomicity Violations in GPU Program

기존 연구들을 GPU 프로그램에 적용하는 것은 두가지 어려움이 있다. 첫 번째로 GPU 프로그램에는 구현에 필수적인 Lock과 Sleep 명령어가 지원되지 않는다[11]. 두 번째 문제로 명령어들을 직접 구현하여 사용할 수 있지만 Lock은 Unlock 동작 시점의 보장 불가, 직렬화 오버헤드, Intra-Warp 데드락과 같은 오동작을, Sleep은 무한정 대기를 발생시킬 수 있다[9].

Fig. 3은 기존 기법들이 Lock과 Sleep의 사용하는 방법을 요약한 그림이다. Fig. 3(a)는 기존 기법의 진단 단계로, 각 스레드들의 공유메모리 접근정보를 기록하고 비교하는 동작을 수행할 때 Lock을 사용하여 원자성 위배를 방지한다. 수리 단계에서는 Fig. 3(b), (c)와 같이 Lock 이나 Sleep를 사용하여 접근사건의 발생 시점을 지연시킨다. 하지만 GPU 프로그램의 SIMT 실행모델은 컨텍스트 스위칭을 발생시키는 Sleep과 스레드들을 직렬화 시킬 수 있는 Lock에 해당하는 명령어를 지원하지 않는다.

GPU 프로그램에서 Lock은 Atomic Function을 사용하여 관행적으로 구현될 수 있다. Table 2의 코드는 Lock 구현의 예시이다. 원자적으로 수행되는 지속적인 Compare and Swap 명령으로 Lock을 획득하고, 임계영

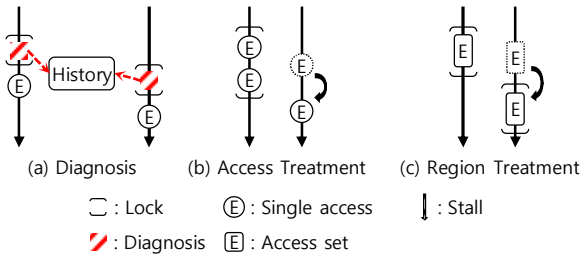

Fig. 3. Usage of Lock and Sleep

Table 2. Example of Lock Implementation

```
1. while (0 != (atomicCAS(mutex, 0, 1))) { }
2. /*critical section*/
3. atomicExch(mutex, 0);
```

역 수행 후 Exchange 명령을 통해 반환한다. 이렇게 구현한 Lock을 사용하면 세가지 문제가 발생할 수 있다. 첫 번째로 GPU 프로그램에서는 프로그래머가 임의로 스레드의 스케줄러 선점을 발생시킬 수 없기 때문에 Lock이 선점되었을 때 Unlock의 동작 시점을 보장할 수 없다. 두 번째 문제는 접근사건마다 동작하는 Lock으로 인해 CPU 프로그램보다 많은 스레드가 직렬화되어 오버헤드를 발생시킬 수 있다. SIMT 실행 모델에서는 리드 내의 모든 스레드가 같은 코드를 실행하고, 워프 내의 모든 스레드가 동시에 메모리에 접근하기 때문에 더 많은 스레드가 직렬화될 수 있다. 마지막으로 같은 워프 내에서 두 개의 Lock이 삽입되어 교차적으로 사용되는 경우 두 스레드가 서로 획득한 Lock의 반환을 대기하는 Intra-Warp 데드락이 발생할 수 있다.

GPU 프로그램에서 Sleep의 관행적 구현은 클럭 사이클을 Busy Waiting으로 감시하는 것이다. 하지만 이 방법은 Livelock을 발생시킬 수 있다. 특히 Intra-Warp 원자성 위배가 발생하였을 때 Warp Divergence에 의해 아직 실행되지 않은 분기의 접근사건 종료를 현재 실행중인 분기에서 무한정 대기할 수 있다. 따라서 GPU 프로그램의 원자성 위배를 자율수리하기 위해서는 Lock과 Sleep에 의지하지 않는 새로운 기법이 필요하다.

V. Design of The Proposed Scheme

본 논문은 GPU 프로그램에서 발생하는 원자성 위배를 자율적으로 수리할 수 있는 도구인 ARCAV (Atomatic Recovery of CUDA Atomicity violation)를 제시한다. ARCAV가 동작하는 방법을 요약하면 메모리 접근을 사본메모리로 우회하여 원자 영역 종료 시점에서 원자성 위배를 진단하고, 원자 영역의 재수행으로 수리하는 것이다. 원자 영역은 배리어로 구분되는 구간으로, 원자 영역의 종료 직전에 원자성 위배의 발생 여부에 따라 원본메모리로 사본 값이 반영되거나, 재수행된다. 이 장에서는 제시된 기법의 동작 흐름과 내부 구조를 설명한다.

5.1. Execution Flow

Fig. 4는 서로 다른 블록에 속한 스레드 2개로 수행되는 간단한 프로그램과 ARCAV를 적용한 프로그램의 수행의

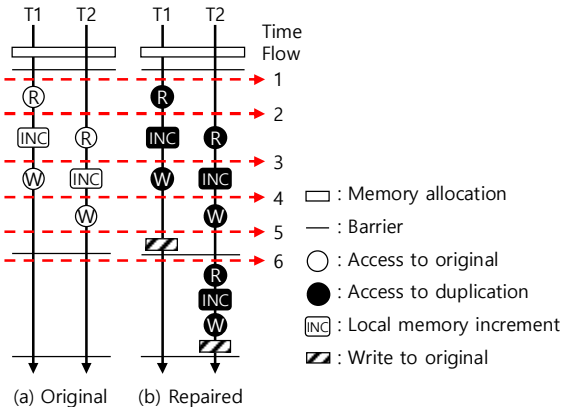

Fig. 4. Example of Repariring Atomicity Violation

비교를 도식화한 것이다. Fig. 4(a)는 원본 프로그램으로 블록 수준 배리어인 __syncthreads()로 감싸진 영역에서 공유메모리에 증가 연산을 수행한다. 이때 두 스레드는 각기 다른 블록에 속해 있기 때문에 스레드 1의 원자성이 스레드 2에 의해 깨어질 수 있다.

Fig. 4(b)는 자율수리가 적용된 수행으로, 프로그램 시작시점에 할당된 메모리의 정보를 입력받아 크기가 확장된다. 공유메모리는 같은 크기만큼 추가 할당되어 사본메모리로 사용되고, 공유메모리의 배열 크기만큼 접근역사를 기록하기 위한 공간이 추가된다. 스레드 지역 메모리는 각 스레드 별로 사본메모리가 생성된다.

Fig. 4(b)에서 배리어 영역이 시작된 시점 1 이후의 메모리 접근은 입력받은 메모리 접근정보를 기반으로 사본메모리에의 접근으로 대체된다. 검은 원으로 표현된 공유메모리 접근은 수행 직전에 해당 공유메모리 영역의 접근역사를 조회하여 다른 스레드에서 쓰기사건을 일으켰는지 확인하고, 아무런 쓰기가 발생하지 않았을 경우 자신의 스레드 ID를 접근역사에 기록한다. 시점 3의 스레드 1 쓰기사건은 메모리 영역에 미리 수행한 스레드가 없기 때문에 사본메모리에 값을 저장하고 접근역사에 ID를 기록한다. 시점 4의 스레드 2 쓰기사건은 접근역사 조회 시 스레드 1의 ID가 확인되기 때문에 원자성 위배가 발생함을 보고하고 사본메모리에는 값을 반영하지 못한다.

이후 배리어 영역이 종료되는 시점 5에서는 Barrier 수행 후 사본메모리를 원본메모리에 반영하며, 오류가 예상되는 스레드 2는 흐름을 영역의 시작 시점으로 변경하고, 접근역사를 초기화한다. 이후 스레드 1에 의해 변화한 원본메모리 및 사본메모리의 값을 기반으로 같은 연산을 수행한다. 이를 통해 원자성 위배로 인한 잘못된 값이 공유 및 지역 메모리에 반영되지 않는다.

ARCAV에 의한 배리어 영역 재수행의 회수는 원자성 위배에 관련된 스레드의 개수로 결정된다. 원자성 위배가

발생한 스레드가 여러 개일 경우 재수행될 때마다 차례대로 메모리에 반영되며, 직렬화되어 동작한 것과 같은 결과를 발생시킨다. 같은 배리어 영역을 수행하는 스레드들은 마지막 원자성 위배가 수리될 때까지 대기하게 된다. 하지만 최적화를 위해 의도적으로 모든 스레드가 같은 값을 메모리에 쓰는 경우, 불필요한 수리로 인한 오버헤드를 방지하기 위해 재수행하지 않는다.

5.2. Design

ARCAV의 전체적인 설계는 Fig. 5에 나타내었다. 제안된 기법은 3가지 모듈과 두 타입의 데이터로 구성된다. Memory Allocation 모듈은 프로그램이 메모리를 할당할 때 사본메모리와 접근역사를 생성하는 모듈이다. Diagnosis 모듈은 메모리 엑세스를 우회하고 접근역사를 확인하여 원자성 위배 오류의 발생 여부를 진단하는 모듈이다. 마지막으로 Treatment 모듈은 실제 메모리에 데이터를 반영시키고 오류 발생 시 재수행 시키는 모듈이다. Shared Data는 각 스레드에서 공유되는 데이터이며, Thread Local Data는 스레드별로 유지되는 데이터를 담는다.

5.2.1. Memory Allocation

Memory Allocation 모듈은 지역 및 공유메모리 정보를 토대로 사본 및 접근역사를 생성한다. 공유메모리의 할당은 Fig. 6과 같이 원본메모리와 동일한 크기 및 값을 가지는 사본메모리와, 공유메모리의 배열 요소마다 4byte에 해당하는 크기의 접근역사를 생성한다. 공유메모리의 타입은 원본과 같다. Memory Allocation에 의해 새롭게 할당되는 공유메모리의 크기는 다음과 같다.

original size * 2 + element size * 4 byte

지역 메모리의 사본은 원본과 완전히 동일한 크기와 타입으로 할당한다. 이렇게 생성된 사본 공유메모리와 접근역사는 Shared Data, 사본 지역메모리는 Thread Local Data에 저장된다.

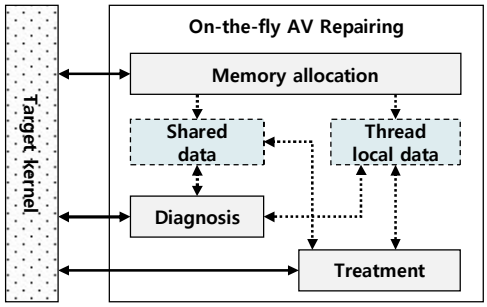

Fig. 5. Overall Design

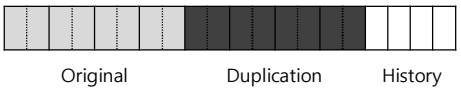

Fig. 6. Allocated Memory

5.2.2. Diagnosis

Diagnosis 모듈은 배리어 영역 내의 메모리 접근을 사본메모리로의 접근으로 변경시킨다. 지역 메모리에 대한 접근사건의 경우 단순히 주소 값을 사본의 값으로 변경하지만, 공유메모리의 경우는 Algorithm 1에 의해 접근할 메모리 주소가 변경된다. Algorithm 1의 addr은 접근할 메모리의 주소, tid는 스레드의 ID를 나타내고, 접근역사인 hist로의 접근은 Atomic Function을 통해 접근된다.

DiagnoseRead는 공유메모리 읽기사건을 대체하여 적용된다. 다른 스레드가 쓰기사건을 발생시킨 주소에 접근 시 원본메모리 값을 반환하며, 다른 스레드가 쓰기 이전에 값을 읽은 상태와 같은 결과를 낸다. 읽기사건만으로는 위배가 발생하지 않으므로 접근역사를 수정하지 않는다.

DiagnoseWrite는 쓰기사건에 적용되는 규칙으로, 가장 먼저 접근한 스레드만 ID를 역사에 기록하고 사본에 값을 쓰도록 한다. 다른 스레드가 이미 쓰기를 발생시킨 공간에 접근하는 경우 쓰기사건은 무시된다. 하지만 쓰기사건을 무시하기 위해 분기되는 경우 Warp Divergence에 의한 성능 저하가 발생할 수 있기 때문에 임시로 생성된 공간에 쓰기사건을 발생시키는 것으로 분기의 실행을 방지한다.

같은 워프의 두 스레드가 두 메모리 공간에 교차적으로 접근하였을 경우 2,5 절에서 설명한 Intra-Warp 데드락을 발생시킬 수 있다. 이를 방지하기 위하여 접근역사와 스레드 ID를 비교할 때 CheckIntrawarp가 수행된다. 두 번째 메모리 접근사건에서 두 스레드가 서로에 의해 원자성 위배 경고를 발생시키는 것을 방지하기 위해, 가장 낮은 tid를 가지는 스레드만 쓰기사건을 발생시키고, 나머지 스레드는 자신이 수정한 접근역사를 초기화시킨다.

5.2.3. Treatment

Treatment 모듈은 수리 영역을 식별하고, 영역 종료 시 사본메모리 값을 결과에 반영하거나 영역을 재수행 시킨다. 수리 영역의 식별은 배리어 동기화 명령의 위치를 식별하여 두 개의 영역 사이로 지정된다. 하지만 __syncthreads() 배리어 명령은 분기내에서 사용될 시 정상적인 동작을 보장하지 않으므로, 코드블록 내에 삽입할 시 의도하지 않은 결과를 발생시킬 수 있다. 따라서 블록 내 모든 스레드가 수행하는 범위로 수리 영역을 지정하기

Algorithm 1. Pseudocode of Diagnosis

```
Global Variable:
  original[addr]: original shared memory
  dupl[addr]: duplicated shared memory
  hist[addr]: access history

function DiagnoseRead (addr, tid)
  if hist[addr] = tid then
    return dupl[addr]
  else
    return original[addr]
  end if
end function

function DiagnoseWrite(addr, tid, value)
  if hist[addr] = tid or -1 then
    dupl[addr] := value
    hist[addr] := tid
  else if hist[addr] % 32 = tid % 32 then
    CheckIntrawarp(addr, tid)
  else
    alert := true
  end if
end function

function CheckIntrawarp (addr, tid, value)
  if tid > hist[addr] then
    {x|x = tid∩ x∈hist} :=−1
  end if
  DiagnoseWrite(addr, tid, value)
end function
```

위해 Table 3과 같은 추가 식별 포인트를 사용한다.

수리 동작은 영역 내의 코드를 수행하고 영역의 끝에 도달하면 다시 한번 배리어를 사용하여 모든 스레드가 영역 수행을 종료하는 것을 대기한다. 이후 사본 공유메모리의 연산 결과를 원본 공유메모리에 반영한다. 이때 경보가 발생한 스레드의 연산 결과는 원본메모리에 반영하지 않는다. 메모리값 반영이 종료되면 경보가 발생한 스레드는 흐름을 시작 배리어 직전으로 되돌려 영역을 재수행하며, 정상 스레드는 영역내 코드 수행을 생략하여 재수행하는 스레드들의 영역 종료를 대기한다.

수리 영역은 블록 영역과 그리드 영역으로 나뉘며, 재수행 방법이 다르게 적용된다. 블록 영역은 블록 내 동기화인 __syncthread()로 식별되고 shared memory에 대한 수리가 적용된다. 그리드 영역은 Host에서 실행된 커널 전체를 영역으로 식별하고, global memory에 대한 수리가 적용된다. Global 영역은 커널 내 동기화가 불가능하므로, Host에서 메모리 반영이 수행되고, 경보가 발생된 블록이 영역을 재수행하도록 커널을 다시 수행시킨다.

Table 3. Additional Point for Treatment Region

Condition	Additional Point
__global__ function	start/end of Function
if/loop/function code block which contains __syncthreads()	before/after/start/end of code block

VI. Implementation of The Proposed Scheme

이 장에서는 먼저 제안된 ARCAV의 구현과 원본 프로그램에 제안하는 도구를 적용하는 과정을 설명한다. 구현은 Nvidia의 GPU 및 CUDA 프로그래밍 언어로 작성된 프로그램을 대상으로 구현되었다. ARCAV이 구현된 환경은 CUDA Toolkit 9.2, LLVM 5.0 컴파일러, Ubuntu 16.04.4 (kernel 4.4)이다. CUDA Toolkit 9.2 버전은 LLVM 5.0을 기반으로 하는 Nvcc 컴파일러가 내장되어 있으며 NVVM IR 1.5 버전이 사용된다. ARCAV의 동작은 Nvidia Compute Capability 6.0 이상의 GPU와 리눅스 환경에서 동작하도록 구현하였으며, 이하의 버전에서는 동작을 보장하지 않는다.

ARCAV는 수리할 프로그램에 적용하기 위한 방법으로 Fig. 7과 같이 CUDA 프로그램의 컴파일러인 Nvcc를 확장하여 프로그램의 컴파일단계에서 사용한다. 확장된 Nvcc는 자율수리에 필요한 기능을 프로그램 컴파일 시 삽입하고, 수행 중에 작동시킨다. Nvcc의 확장은 LD_PRELOAD를 사용하여 중간코드를 획득하고, 이를 LLVM IR을 활용하여 정적으로 코드를 삽입한다.

Nvcc 컴파일러는 입력된 코드를 전처리한 후 cicc를 호출하여 이를 GPU 중간코드인 ptx로 변경한다. 소스코드가 ptx로 변경되는 과정은 오픈소스 컴파일러인 LLVM Bitcode를 기반으로하는 NVVM IR이라는 중간코드로 변경하여 수행되는데 이를 통해 ptx로 변하기 직전의 NVVM IR을 획득하면 이를 LLVM으로 수정할 수 있다.

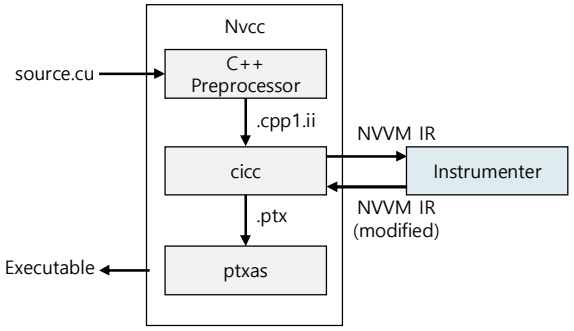

Fig. 7. Instrumentation Overview

이를 위해 LD_PRELOAD를 통해 cicc의 ptx 출력함수를 대신 수행할 동적 라이브러리와, Nvcc의 cicc 호출에 LD_PRELOAD를 더해줄 동적 라이브러리를 구현하였다.

ARCAV의 구현에는 오버헤드를 최소화하기 위해 5장에서 설명된 diagnosis 기법에 추가적인 최적화 기법 2가지가 도입되었다. 첫 번째 최적화 기법은 유해하지 않은 경합 조건으로 인한 원자성 위배의 진단을 방지하기 위해 사본메모리에 미리 쓰여진 값과 쓰려는 값이 같을 경우 경보를 발생시키지 않는 것이다. 두 번째 최적화 기법은 소스코드 분석 과정에서 읽기사건만이 존재하는 공유변수로의 접근과, 쓰기사건이 하나이고 상수 값을 쓰는 공유변수로의 접근을 진단 대상에서 제외시키는 것이다.

기법이 올바르게 구현되었음을 검증하기 위해 Table 1의 increment 커널에 ARCAV를 적용하여 원본 커널 및 atomicAdd를 사용한 버전과 a 배열의 총합을 비교하였다. 시험은 64개의 스레드로 실행하여 스레드 0~31과 스레드 32~63이 같은 메모리 주소에서 원자성 위배를 발생시킨다. 시험의 결과는 Table. 4와 같이 원자성 위배가 발생하지 않았음을 확인하였다.

VII. Experimentation and Analysis

이 장에서는 GPU 프로그램의 원자성 위배를 수행 중에 자율수리할 수 있음을 보이기 위한 실험과 그 결과를 분석한다. 7.1절에서는 수리 가능한 원자성 위배의 패턴을 검증하기 위한 실험, 7.2절에서는 실제 GPU 프로그램에서의 탐지 및 수리 여부와 시간 오버헤드를 기존 탐지기법들과 비교한 실험, 그리고 7.3절에서는 메모리 계층과 수리 여부 및 스레드 구성에 따른 시간 오버헤드를 측정하기 위한 실험을 수행하였다. 실험에 사용된 환경은 Intel i7 3700 CPU, 16GB RAM, 768개의 CUDA 코어를 가지는 Nvidia Geforce 1050 그래픽 카드를 활용하였으며, CUDA Toolkit 9.2, LLVM 5.0 컴파일러로 컴파일되고 Ubuntu 16.04.4 (kernel 4.4)에서 실행하였다.

7.1. Repair Coverage of ARCAV

ARCAV가 수리할 수 있는 원자성 위배의 유형을 확인하기 위한 실험의 테스트 케이스는 원자성 위배가 발생하는 패턴들을 분석한 연구들을 참조하였다. Lu 등의 연구결과[2]에 의하면 deadlock이 아닌 동시성 버그는 1개의 공유변수에서 발생하는 경우가 약 66%이며 약 96%의 버그가 2개의 스레드에서 발생한다. 관련된 접근사건의 개수는 2, 3, 4 개인 경우가 각각 약 30%, 46%, 16%에 해당한다. Zhang 등의 연구결과[8]는 원자성 위배의 발생을 직렬화가 불가능한 5가지의 인터리빙 패턴으로 분류한다.

수리범위 실험은 기존 연구에서 분류된 5가지 원자성 위배 패턴 중 RRW|RRW 패턴을 제외한 4가지의 패턴 RR|W, WR|W, WW|R, RW|RW과 2.5절에서 설명된 교차 메모리 접근으로 인한 데드락을 확인하기 위한 W1W2|W2W1 패턴을 포함한 5가지 패턴을 포함한 합성 프로그램으로 수행되었다. 각 합성프로그램은 두가지 계층의 공유메모리 위치에서 발생하는 원자성 위배와, Intra/Inter-Warp 원자성 위배를 발생시켜 수리가 가능한지 확인하였다. 실험은 global memory와 shared memory Memory에서 원자성 위배가 발생하도록 두 번씩 수행되었다. Table 5는 실험의 결과로 원본과 수리가 적용된 프로그램의 결과값과 재수행이 발생하였는지 여부를 기록한 것이다.

실험 결과는 ARCAV가 5가지 패턴에 대해 원자성 위배를 공유메모리 계층에 상관없이 수행 중에 자율적으로 수리할 수 있음을 나타낸다. 또한 원자성 위배를 메모리 계층 및 워프에 관계없이 수리가 가능하다. 개별 패턴에 대한 분석으로는 RR|W 패턴과 WW|R 패턴은 읽기사건 우회 기능의 특성상 수리에 재수행이 필요하지 않으며, W1W2|W2W1 패턴의 수리에 새로운 오류가 발생하지 않았다.

Table 4. Test Result of Repairing increment kernel

	Original	Repaired	Atomic Function
Result	32	64	64

7.2. Repairing AV in Real-world GPU Kernels

실제 GPU 프로그램에 ARCAV를 적용하여 원자성 위배를 수리할 수 있음을 확인하기 위해 CUDA Toolkit 7.5의 예제 코드와 Rodinia 3.1[17] 벤치마크 모음에서 원자성 위배가 발생하는 4개의 커널을 선정하였다. Table 6은 선정된 커널의 세부 내용을 나타낸 것으로, 첫 번째 열과 두 번째 열은 프로그램 이름과 출처, 세 번째 열은 코드의 라

Table 5. Repair Coverage of ARCAV

Pattern	Occurance	Expected	Original	Repaired	Re-executed
RR|W	X	128	64	128	X
WR|W	X	128	64	128	O
WW|R	X	128	64	128	X
RW|RW	X	128	64	128	O
W1W2|W2W1	X	128	0	128	O

인 수, 네 번째 열은 블록 및 스레드의 구성, 마지막은 원자성 위배가 발생한 메모리 계층을 나타낸다. 이 실험에서는 선정된 프로그램들에 ARCAV를 적용하여 원자성 위배의 수리 여부와 오버헤드를 측정하였다.

7.2.1. Memory Hierarchy

Table 7과 Fig. 8은 본 논문에서 제안된 ARCAV와 기존 탐지기법의 원자성 위배의 탐지 여부와 오버헤드를 나타낸다. ARCAV는 4가지 커널에서 발생한 모든 원자성 위배를 탐지하고, 수행 중에 수리할 수 있었다. 하지만 Nvcc 컴파일러에서 기본으로 제공하는 memcheck[11]는 shared memory 계층만을 탐지하는 기법이기 때문에 global memory 계층에서 발생하는 동시성 오류를 탐지할 수 없었다. BARRACUDA[18]와 CURD[19] 탐지기법은 논문에서 제시한 탐지 결과를 참조하였으며, 4가지 커널 모두에서 원자성 위배를 탐지할 수 있었다.

7.2.2. Overhead

Fig. 8은 원본 프로그램과의 수행 시간을 비교한 결과이다. 이중 BARRACUDA와 CURD는 각 논문에서의 원본 코드 및 racecheck와의 수행 시간 비교 데이터를 가져와 표시하였으나 BARRACUDA의 scalarprod와 paticle filter의 시간 오버헤드에 관한 정보가 없었다. ARCAV를 적용하였을 때 소요되는 시간은 탐지만 수행할 경우(D) 평균 1.98배, 탐지와 수리를 모두 수행할 경우(D+R) 평균 약 2.1배의 수행 시간이 소요되었다. 가장 적은 오버헤드를 발생시킨 dxtc는 프로그램 중 공유메모리 액세스의 대부분이 shared memory로의 접근을 사용하였고, 대부분이 global memory는 읽기로만 수행되어 6장에서 설명된 최적화 기법으로 인해 모니터링 대상에서 제외되었다. bfs와

Table 6. Specification of GPU kernels

Program	Source	LoC	Block /Threads	Hierarchy
dxtc	CUDA 7.5	820	16384/64	Shared
scalarProd	CUDA 7.5	271	128/256	Shared
bfs	Rodinia 3.1	349	1945/512	Global
particlef_float	Rodinia 3.1	870	2/512	Global

Table 7. Detection and Repairing Result

Program	Detection			ARCAV	
	race check	BARRA CUDA	CURD	Detection	Detection +Repairing
dxtc	O	O	O	O	O
scalarProd	O	-	O	O	O
bfs	X	O	O	O	O
particlef_float	X	-	O	O	O

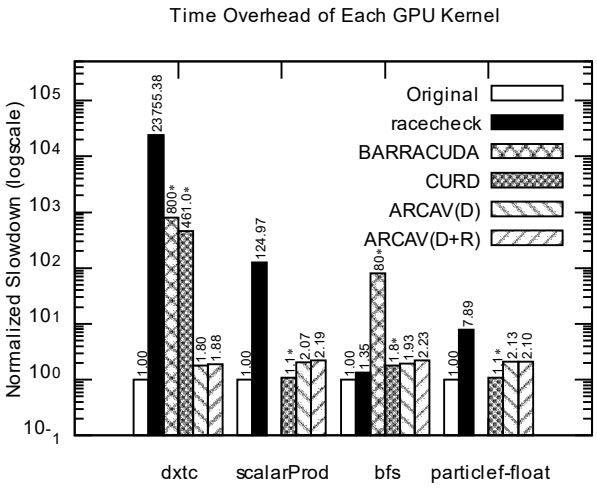

Fig. 8. Comparison of Time Overhead

particlef_float 프로그램은 공유메모리 접근의 대부분이 global memory로의 접근을 수행하였기 때문에 dxtc보다 높은 오버헤드가 발생하였다. scalarProd는 global memory 접근이 대부분 읽기사건이기 때문에 감시 대상에서 제외되고 shared memory 위주로 접근하지만, 반복문 내부의 짧은 범위에 배리어 동기화가 존재하여 잦은 수리 영역 설정으로 인한 오버헤드가 발생하였다.

7.3. Repairing Robustness

ARCAV가 발생시키는 시간 오버헤드의 분석을 위해 메모리 계층 및 수리 여부에 의한 오버헤드와 스레드 구성에 의한 오버헤드 분석 실험을 하였다. 각 실험에 사용된 합성 프로그램은 오버헤드 변화를 쉽게 파악할 수 있도록 메모리 접근 수를 증가시켜 시간 오버헤드가 실제 GPU 프로그램보다 높게 나타나도록 구현하였다. 합성 프로그램은 Table 1에 나타낸 커널 코드를 10만 회 반복하는 것으로, 초기화 및 반복문 구성을 제외한 모든 코드 라인에서 공유메모리에 접근한다. 그리고 실험 변수 설정 시 접근되는 공유메모리 계층, 스레드 구성, 그리고 원자성 위배가 발생하는 스레드는 커널 실행시점에 구성하도록 설계하였다. 두 실험 모두 전체 커널 수행과정 중 공유메모리 접근 명령의 비율이 실제 프로그램보다 높기 때문에 원본 프로그램과의 수행시간 보다는 각 케이스 별로 발생한 시간 오버헤드 간의 비교를 중점으로 분석하였다.

7.3.1. Memory Hierarchy

ARCAV는 공유메모리의 계층에 따라 다른 수리 영역과 자율수리 방법이 사용되며, shared memory와 global memory는 다른 접근속도를 가진다. Table 8은 접근 메모리 계층에 따른 시간 오버헤드를 실험한 결과이다.

Table 8. Slowdown by Thread, Memory, Atomicity Violation

Block/Thread	Memory	Detection	Detection +Repairing
2/32	Shared	1.79x	3.49x
	Global	3.65x	7.29x
32/32	Shared	1.83x	3.48x
	Global	3.90x	7.47x

Table 8의 세 번째 열은 원자성 위배가 발생하지 않아 탐지만이 수행되었을 경우의 수행 시간의 저하를 나타낸다. global memory에 접근 시에는 발생하는 탐지 오버헤드는 shared memory에 접근할 때보다 약 2배 높게 측정되었으며, 블록 개수를 증가시켜 전체 스레드 개수가 증가한 것에 영향을 받지 않았다.

Table 8의 네 번째 열은 원자성 위배가 발생하여 수리동작이 적용되었을 경우의 수행 속도 저하를 측정한 결과이다. 수리동작은 모든 스레드에서 원자성 위배가 발생하여 재수행이 동작하도록 지정하여 최대 오버헤드를 확인하였다. 수리가 적용될 경우 탐지만 수행했을 경우보다 2배의 수행 시간이 소요되었다. 스레드 구성에 의한 비율은 탐지만 수행하였을 경우와 마찬가지로 오차범위 내의 일정한 수치를 보이며, 수리 시 모든 스레드의 절반이 재수행되는 최악의 경우에도 탐지의 약 1.9배에 해당하는 수행 시간만이 소요되었다. global memory 접근 시 추가적으로 shared memory의 약 2.14배의 오버헤드가 발생하였다.

7.3.2. Thread Configuration

원자성 위배를 수리할 프로그램의 스레드 개수와 block 구성의 변화에 대한 수행 시간의 강건성을 확인하기 위해 스레드 구성에 대한 실험을 수행하였다. 실험은 Global 및 shared memory에 접근하는 합성 프로그램을 블록 내 스레드 개수를 128개부터 최대 설정 가능 개수인 1,024개까지 증가시키고, 블록 개수가 1개일 경우와 4일 경우로 설정하여 오버헤드의 변화를 확인하였다.

Fig. 9는 실험 결과를 정리한 것으로 x축은 블록 내 스레드 개수, y축은 원본 프로그램 대비 속도 저하의 배율을 의미한다. 자율수리가 적용되는 대상 프로그램이 shared memory에 대한 접근을 수행할 때는 블록당 스레드 개수가 128개에서 1024개로 증가함에 따라 약 20%의 오버헤드가 증가하였다. 하지만 블록 내 스레드 개수를 고정 시키고 블록 개수를 증가시켰을 경우에는 오버헤드가 증가하지 않았다. 이는 블록의 개수가 증가하더라도 shared memory에서 발생하는 원자성 위배를 수리하기 위해 사용되는 블록 내 배리어가 적용되는 스레드 개수가 같기 때문이다.

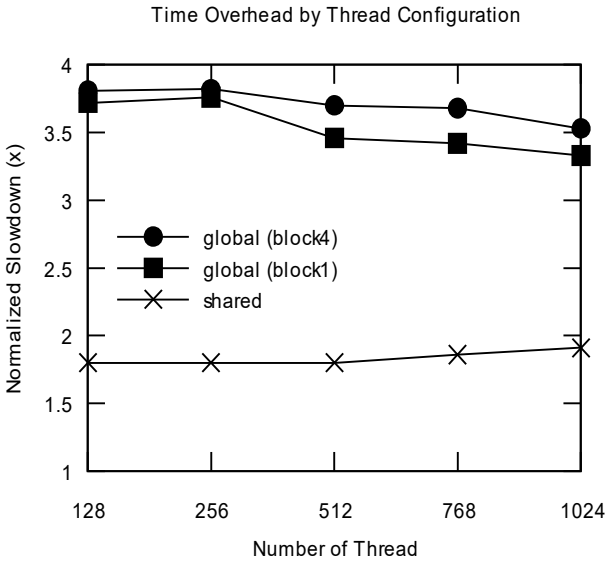

Fig. 9. Time Overhead by Thread Configuration

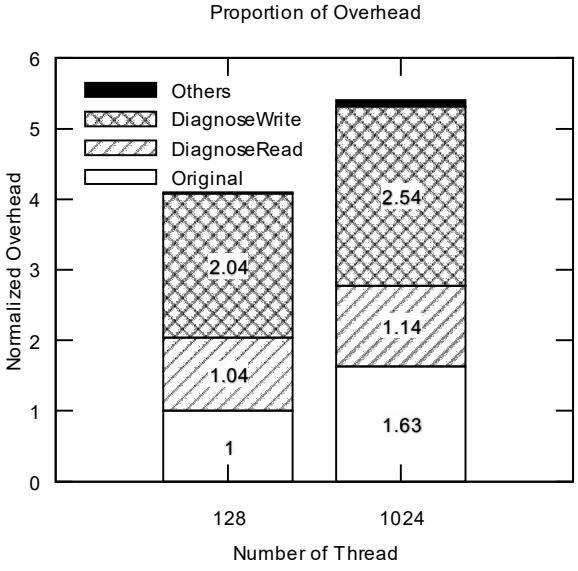

Fig. 10. Overhead Ratio

global memory에 접근하는 경우에는 블록당 스레드 수가 같고 블록 개수를 1개에서 4배로 증가시켰을 때 평균 약 6%의 오버헤드가 증가하였다. 블록 내 스레드 수를 증가시켰을 경우 128개에서 512개 까지는 오버헤드의 비율이 일정하게 유지되었으나, 768개에서 1,024개일 경우 오히려 오버헤드가 감소하였다. 이러한 현상의 원인은 원본 프로그램이 수리를 위한 기법보다 스레드 수에 의한 오버헤드 증가량이 크기 때문이다. Fig. 10은 ARCAV가 수행시간 중 기법이 차지하는 비중을 원본 코드의 수행시간으로 일반화하여 도식화한 것이다. 스레드를 128개에서 1024개로 증가시킬 경우 원본 코드에 의한 수행시간은 63% 증가하지만, 수리 기법에 의한 수행시간은 22%만 증가한다.

VIII. Conclusion

원자성 위배는 병행프로그램에서 발생하는 탐지 및 디버그가 어려운 오류로, 개발 과정에서 탐지하지 못한 결함이 남아 있을 수 있다. 따라서 원자성 위배를 프로그램 수행 중에 탐지하고 자율수리하는 것은 중요하다. 하지만 기존의 원자성 위배 자율수리 연구들은 CPU 프로그램을 대상으로 개발되어 GPU 프로그램에서 사용될 수 없었다. 본 논문은 GPU 프로그램에서 발생할 수 있는 원자성 위배를 프로그램 수행 중에 자율적으로 수리하는 도구인 ARCAV를 제시하였으며, 대표적인 5가지 패턴의 원자성 위배를 가지는 합성 프로그램과 실세계에서 사용되는 네 가지 GPU 프로그램을 수리할 수 있었다. 제안하는 기법의 오버헤드는 네 개의 실제 프로그램에서 원자성 위배를 탐지하고 수리하였을 때 평균 2.1배의 속도 저하가 발생하였고, 스레드의 개수와 구성에 의한 오버헤드가 거의 발생하지 않았다. 또한 원자성 위배의 탐지만 수행하는 기법들이 일부 프로그램에서 보이는 급격한 오버헤드 증가가 없었다. 향후 과제로 오버헤드의 발생 원인을 더 자세히 분석하고, 기법의 최적화를 통한 오버헤드의 감소를 수행할 예정이다.

ACKNOWLEDGEMENT

This research was supported by the Ministry of Trade, Industry and Energy (20005378) and the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2019R1G1A1100455).

REFERENCES

[1] R.H. Netzer and B.P. Miller, "What are race conditions?: Some issues and formalizations," ACM Letters on Programming Languages and Systems, Vol. 1, pp. 74-88, March 1992. DOI: 10.1145/130616.130623

[2] S. Lu, S. Park, E. Seo and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," ACM SIGOPS Operating Systems Review, Vol. 42, pp. 329-339, March 2008. DOI: 10.1145/1346281.1346323

[3] B. Krena, Z. Letko, R. Tzoref, S. Ur and T. Vojnar, "Healing data races on-the-fly," in Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging, pp. 54-64, July 2007. DOI: 10.1145/1273647.1273658

[4] B. Lucia, J. Devietti, K. Strauss and L. Ceze, "Atom-aid: Detecting and surviving atomicity violations," ACM SIGARCH Computer Architecture News, Vol. 36, pp. 277-288, July 2008. DOI: 10.1109/ISCA.2008.4

[5] J. Yu and S. Narayanasamy, "Tolerating concurrency bugs using transactions as lifeguards," in Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 263-274, Dec. 2010. DOI: 10.1109/MICRO.2010.56

[6] S. Zhu, Z. Chen, and G. Sun. "Tuning lock-based multicore program based on sliding windows to tolerate data race," The Journal of Supercomputing, Vol. 75, No. 12, pp. 7872-7894, June 2019. DOI: 10.1007/s11227-019-02921-7

[7] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," ACM SIGARCH Computer Architecture News, Vol. 37, pp. 325-336, June 2009. DOI: 10.1145/1555815.1555796

[8] M. Zhang, Y. Wu, S. Lu, S. Qi, J. Ren and W. Zheng, "A lightweight system for detecting and tolerating concurrency bugs," IEEE Trans.Software Eng., pp. 899-917, Oct. 2016. DOI: 10.1109/TSE.2016.2531666

[9] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao and D. Qian, "Software transactional memory for gpu architectures," in Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 1-10, Feb. 2014. DOI: 10.1145/2581122.2544139

[10] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro, Vol. 28, pp. 39-55, May 2008. DOI: 10.1109/MM.2008.31

[11] C. Nvidia, "CUDA C Programming Guide, version 9.1," NVIDIA Corp, 2018.

[12] P. Li, X. Hu, D. Chen, J. Brock, H. Luo, E.Z. Zhang and C. Ding, "LD: Low-overhead GPU race detection without access monitoring," ACM Transactions on Architecture and Code Optimization, Vol. 14, pp. 9, 2017.

[13] Z. Gu, E.T. Barr, D.J. Hamilton and Z. Su, "Has the bug really been fixed?" in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pp. 55-64, March 2010. DOI: 10.1145/3046678

[14] P. Godefroid and N. Nagappan, "Concurrency at Microsoft: An exploratory survey," in CAV Workshop on Exploiting Concurrency Efficiently and Correctly, May. 2008.

[15] L.L. Pullum, “Software fault tolerance techniques and implementation” Artech House, 2001, .

[16] B. Pourghassemi and A. Chandramowlishwaran, "CudaCR: an in-kernel application-level checkpoint/restart scheme for CUDA-enabled GPUs," in 2017 IEEE International Conference on Cluster Computing, pp. 725-732, Sep. 2017. DOI: 10.1109/CLUSTER.2017.100

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing," In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pp. 44-54, Oct. 2009. DOI: 10.1109/IISWC.2009.5306797

[18] A. Eizenberg, Y. Peng, T. Pigli, W. Mansky and J. Devietti, "BARRACUDA: binary-level analysis of runtime RAces in CUDA programs," in ACM SIGPLAN Notices, pp. 126-140, June 2017. DOI: 10.1145/3062341.3062342

[19] Y. Peng, V. Grover and J. Devietti, "CURD: a dynamic CUDA race detector," ACM SIGPLAN Notices, Vol. 53, pp. 390-403, April 2018. DOI: 10.1145/3296979.3192368

Authors

Keonpyo Lee received the M.S. degrees in Informatics from Gyeongsang National University (GNU) in 2018. He is currently a Ph.D. Student in the Department of AI Convergence Engineering, Gyeongsang National University (GNU). His research interests include airborne software and dependable software.

Seongjin Lee is currently an Associate Professor at Department of AI Convergence Engineering, Gyeongsang National University, South Gyeongsang Province, Korea. He received B.S., M.S., degree in Department of Electronics and Computer Engineering, Hanyang University, Seoul Korea in 2006 and 2008, respectively. He received his Ph.D in Computer Engineering in the same university in 2015. Before joining Gyeongsang National University in 2017, he worked as PostDoc and Assistant Research Professor at Hanyang University. His research interests include system performance, Avionics software, and AI.

Yong-Kee Jun received his BS degree in Computer Engineering from Kyungpook National University, and the MS and PhD degrees in Computer Science from Seoul National University. He is now a full professor in Department of Aerospace Software Engineering, Gyeongsang National University (GNU). He was the founding director of GNU Embedded Software Center for Avionics (GESCA), one of the national Information Technology Research Centers (ITRCs) of South Korea. His professional interests include dependable software, embedded software, and airborne software. Prof. Jun is a member of Association for Computing Machinery (ACM).