공학석사학위논문

정적 프로파일링을 통한 GPU 워크로드의 데이터 지역성 및 L1 캐시 분석 연구

A Study on Data Locality and L1 Cache Analysis of GPU Workload Using Static Profiling

2023년 2월

서울대학교 대학원

컴퓨터 공학부

김 지 은

정적 프로파일링을 통한 GPU 워크로드의 데이터 지역성 및 L1 캐시 분석 연구

A Study on Data Locality and L1 Cache Analysis of GPU Workload Using Static Profiling

지도교수 엄현상

이 논문을 공학석사 학위논문으로 제출함

2022년 11월

서울대학교 대학원

컴퓨터 공학부

김지은

김지은의 공학석사 학위논문을 인준함

2023년 1월

위 원 장	염 헌 영	(인)
부위원장	엄 현 상	(인)
위 원	전 병 곤	(인)

초록

GPU는 계산적 효율성이 알려짐에 따라 컴퓨터 그래픽스에만 활용되고 있는 것뿐만 아니라 고성능 컴퓨팅, 머신러닝 등 다양한 워크로드에서도 사용되고 있다. 이에 따라 GPU의 한정된 메모리 자원을 효율적으로 이용하려는 연구가 활발히 진행되고 있다. 특히 GPU 메모리 계층 구조에서 크기가 작지만 접근 속도가 빠른 L1 데이터 캐시를 효율적으로 활용하기 위해선 워크로드의 데이터 지역성을 알고 활용하는 것이 매우 중요하다. 따라서 본 논문은 워크로드의 데이터 지역성을 분석하고 객관화하기위한 척도를 제안한다. 이를 위해 PTX 코드를 기반으로 한 정적 프로파일링을 수행하여 지역성을 나타내는 지표를 정의하고 실제 다양한 GPU 워크로드들을 분석한다. 이러한 분석을 통해 본 연구가 제시하는 지역성 척도가 실제 실행 시 캐시 활용도와 유의미한 연관성이 있음을 확인하였다.

주요어: GPU profiling, Data Locality, Coalescing, PTX code, L1 cache

학번: 2021-26019

목차

초록 **i**

목차 **ii**

그림 목차 **iv**

제 1 장 서 론 **1**

1.1 GPU에서의 coalesced memory 접근 3

1.2 구성 7

제 2 장 관 련 연 구 **8**

2.1 데이터 지역성 정적 분석 9

제 3 장 본 론 **11**

3.1 연구의 내용 11

3.1.1 PTX 코드를 통한 데이터 지역성 분석 12

3.1.2 Load global 기반 syntax tree 생성 방법 13

3.1.3 Syntax tree 기반 지역성 그래프 생성 방법 15

3.1.4 PTX 코드를 통한 워프 수준 coalescing 접근 분석 16

제 4 장 실 험 **19**

4.1 실험 환경 19

4.2 L1 캐시와 coalescing의 관계 20

4.3 Degree of coalescing 그래프를 통한 분석 24

제 5 장 결 론 **29**

5.1 결과 분석 29

Abstract **34**

그림 목차

그림 1.1 Thread Block의 SM 내 스케줄링 예시 2

그림 1.2 Thread Block의 SM 내 스케줄링 예시 2

그림 1.3 Warp memory access type [1] 4

그림 2.1 Thread Block의 SM 내 스케줄링 예시 8

그림 3.1 Backprop PTX 코드 예시 11

그림 3.2 Backprop syntax tree 12

그림 3.3 Backprop Syntax tree in json type 13

그림 3.4 coalescing 그래프 생성 과정 16

그림 4.1 NVIDIA A30 spec 19

그림 4.2 workloads 20

그림 4.3 Relation between degree of coalescing and L1 hit rate . . . 21

그림 4.4 Coalescing Graph Graph in simple workload 24

그림 4.5 Degree of coalescing Graph in real workload 25

그림 4.6 Degree of coalescing, sector expectation and sector/request 26

그림 4.7 Sector access expectation과 L1 sector/request 27

제 1 장 서 론

GPU가 GPGPU로 많이 사용됨에 따라 고성능 컴퓨팅, 머신 러닝, 빅데이터 분석 등에서 다양한 응용 분야에서 GPU가 사용된다. 다양한 워크로드가 GPU에서 수행되고 활용하는 데이터의 크기가 커짐에 따라 GPU가 수행하는 워크로드의 메모리 접근 효율성을 고려하게 된다. GPU를 효율적으로 사용하기 위해서는 GPU의 병렬 연산의 특성과 메모리 계층을 알고 수행하는 것이 필요하다. GPU는 다수의 스레드가 하나의 수행 단위로 묶여 동시에 같은 명령어를 수행한다. 이러한 GPU 수행 특성으로 인해 스레드의 묶음이 동시에 메모리 접근 명령어를 수행할 때 메모리 계층별로 데이터 지역성이 발생한다. 이때 발생하는 데이터 지역성을 분석하는 것을 통해 워크로드의 데이터 접근 특성을 분석할 수 있고, 실제 메모리 계층에서의 접근 효율성을 예상할 수 있다

GPU에서 발생할 수 있는 데이터 지역성은 GPU 수행 방식에 따라 세분화할 수 있다. GPU는 워크로드를 실행할 때 수행할 스레드의 묶음을 지정한다. 스레드의 묶음에는 계층이 존재하는데, 가장 큰 단위는 그리드이다. 그리드를 구성하는 것이 스레드 블록이며, 스레드 블록은 스레드로 구성된다. 그리드의 크기 및 스레드의

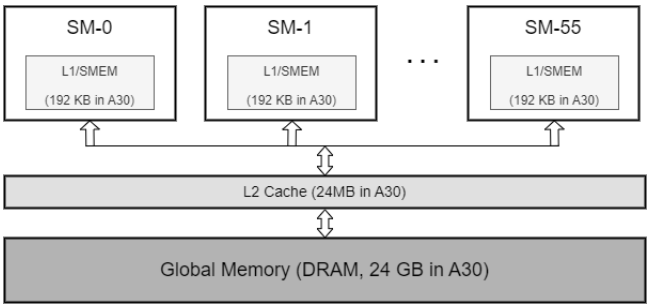


그림 1.1 Thread Block의 SM 내 스케줄링 예시

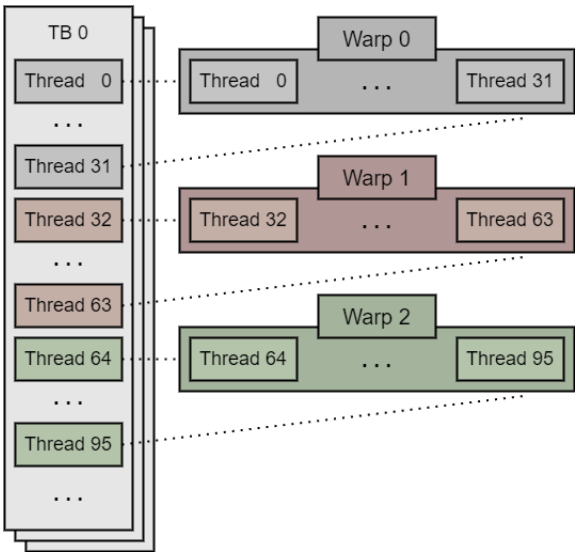


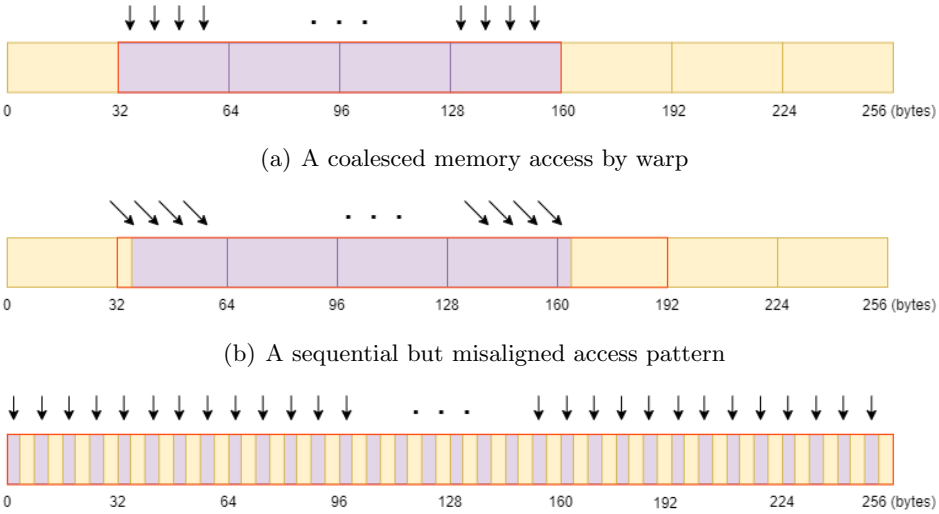
그림 1.2 Thread Block의 SM 내 스케줄링 예시

크기는 사용자가 지정하여 수행할 수 있지만, 실제 수행에 있어 스레드 블록 내의 스레드들은 다시 32개의 스레드로 나뉘어 워프(wrap)로 분할되며, warp는 같은 명령어를 동시에 수행한다. 이처럼 GPU는 다중 스레드를 수행하면서, 32개의 스레드 묶음 단위도 명령어 동시 수행을 하는 것으로 인해 다양한 데이터 지역성이

발생한다. 먼저 크게는 그리드 내 스레드 블록간의 데이터 지역성이 발생할 수 있다. 그림 2.1에서 보면 여러 스레드 블록들은 SM(streaming multiprocessor)에 스케줄링된다. 이때 그림 1.1에서 알 수 있듯이 SM은 GPU의 L2 캐시를 공유한다. 따라서 SM간에 겹치는 데이터 접근이 많이 발생할 경우 SM간 데이터 지역성이 높아지게 되고, 이는 L2 캐시 접근 효율성에 영향을 준다. L1 캐시의 경우 SM 내부에 존재하며 SM 내부에 있는 스레드만 접근이 가능하다. 그림 1.2에 나타나듯이 SM 내부에는 여러 스레드 블록이 존재하며, 각 스레드 블록은 다시 워프단위로 나뉘게 된다. SM 내부의 데이터 지역성은 스레드 블록간의 데이터 지역성과 워프 내의 데이터 지역성에 영향을 받을 수 있다.

1.1 GPU에서의 coalesced memory 접근

이와 같이 스레드를 일정 단위로 묶어 실행하는 GPU 아키텍처를 잘 사용하기 위한 고려사항 중 하나는 글로벌 메모리 접근의 coalescing이다. 하나의 워프가 글로벌 메모리 접근 명령어를 수행할 때, 워프 내의 스레드들이 접근하는 메모리 주소의 양상에 따라 글로벌 메모리 접근성의 효율성이 결정된다. 이러한 양상을 구분할 수 있는 기준 중 하나는 워프 내의 스레드들이 접근하는 메모리 주소가



(a) A coalesced memory access by warp

(b) A sequential but misaligned access pattern

(c) A stride access pattern

그림 1.3 Warp memory access type [1]

연속된 접근인지를 확인하는 것이다. 데이터 타입의 크기가 4 바이트일 때 한 워프가 접근하는 최적의 접근 바이트는 128 바이트이다. 하나의 섹터는 32 바이트로 구성되며 접근 섹터 개수가 작을수록 캐시 라인 내에서 공통된 접근이 발생함을 나타낸다. 접근 섹터 및 바이트의 숫자가 높을수록 분산되고 비효율적인 메모리 액세스를 의미할 수 있다. [2] 그림 1.3를 보면 1.3(a)은 워프 내의 스레드가 연속된 접근을 하는 양상을 보인다. 이 경우 워프가 글로벌 메모리 접근을 완료하기 위해선 4개의 섹터를 한번에 가져오는 이동으로 워프 내 모든 스레드가 수행을 완료할 수 있다. 이에 반해 1.3(b)의 경우 연속된 접근을 하지만 그 범위가 섹터의 단위인 32 바이트에 맞지 않아 5개의 sector에 접근해야 하는 단점이 있다. 또한 1.3(c)

의 경우에도 워프의 스레드들이 메모리 접근을 일정 간격으로 건너뛰며 접근을 하는데 이때 다수의 섹터를 접근하는 오버헤드가 발생한다. 이 외에도 스레드들이 메모리를 128 바이트 단위로 접근하지 않아 오버헤드가 생길 경우 coalescing하지 않은 접근이라 한다.

Localityguru [3]에서는 PTX 코드 기반 정적 프로파일링을 수행하였다. [3]에서는 스레드 블록간의 데이터 재사용성을 분석하여 어떠한 스레드 블록을 같은 SM 내에 스케줄링하면 좋을 지를 판단하고자 하였다. 또한 데이터 지역성을 분석하고 [4–8] 이를 고려한 GPU 스케줄링에 반영하는 연구가 많이 진행되어 오고 있다 [3, 9]. 특히 LocalityGuru [3]의 경우 PTX 코드를 활용한 정적 프로파일링을 통해 상호 스레드 블록 간의 데이터 지역성을 파악하는 방법을 소개했다. 이를 통해 스레드 블록 간에 겹치는 데이터 접근 주소를 파악하고, 높은 데이터 지역성을 가지는 스레드블록 조합을 찾아낼 수 있는 스레드 블록 지역성 그래프를 제안했다.

하지만 스레드 블록 수준의 데이터 재사용성 분석에는 실제 캐시 사용성과 연관 짓기에는 한계가 있다. L1 캐시의 경우 하나의 SM 내의 스레드만 접근할 수 있다. 하지만 스레드 블록 수준의 데이터 재사용성을 보는 것은 GPU의 스레드들이 글로벌 메모리에서 데이터를 가져오는 특성을 반영하지 못한 데이터 지역성

분석이다. GPU에선 워프 단위로 같은 명령어를 수행하게 되는데, 글로벌 메모리 접근의 경우 워프 내의 각 스레드가 접근하는 데이터 위치들의 분포 양상이 성능에 큰 영향을 끼치게 된다. 이는 데이터 coalescing으로 표현할 수 있으며 워프 내 스레드들이 접근하는 데이터가 연속되어 있는 지와 얼마나 밀집되어 있는 지가 성능에 영향을 줄 있다. 따라서 본 논문은 이러한 워프 수준의 데이터 지역성 분석을 통해 실제 캐시 사용 양상을 예측하고자 한다.

본 논문에서는 PTX 코드를 활용한 정적 프로파일링을 통해 워크로드의 메모리 접근 양상을 분석하고, 데이터 지역성과 coalescing의 정도를 분석한다. 이를 통해 데이터 지역성과 coalescing을 수치화 하고 위 수치들과 L1 캐시 간의 연관성을 확인한다. 실험은 NVIDIA A30 GPU을 사용하였고 rodinia [10]와 polybench [11] 워크로드를 대상으로 논문 [3]의 PTX 코드 기반 정적 프로파일링을 통해 coalescing degree와 GPU L1 캐시, 데이터 접근 명령어에 필요한 sector/request 개수간의 관련도 분석을 수행하였다.

그 결과, coalescing 척도의 값이 70%이상인 6개의 워크로드 중 6개가 L1 캐시 hit rate 60%이상의 값을 보이며 coalescing의 척도가 캐시 hit rate과 상관관계가 있음을 보였다. 또한 프로파일링한 수치로 예측한 sector 접근 예상 값이 실제

sectors/request와 높은 연관성을 보임을 확인했다. 이를 통해 워프 내의 스레드의 데이터 지역성이 L1 캐시 hit rate과 sectors/request에 영향을 주는 것을 정적 프로파일링을 통해 알 수 있었다.

1.2 구성

본 논문은 다음과 같은 구성을 갖는다.

- **제 2 장**은 GPU에서 워크로드 내의 지역성을 분석하는 관련 연구와 coalesced access에 대한 설명을 한다.
- **제 3 장**에서는 PTX 코드 기반으로 데이터 지역성을 분석하는 자세한 방법론과 PTX를 기반으로 coalescing을 분석한 방법에 대해 설명한다.
- **제 4 장**은 실험 방법 및 결과에 대해서 설명한다.
- **제 5 장**은 결론으로 데이터 지역성, coalescing, 캐시 간의 연관성에 대해 정리한다.

제 2 장 관 련 연 구

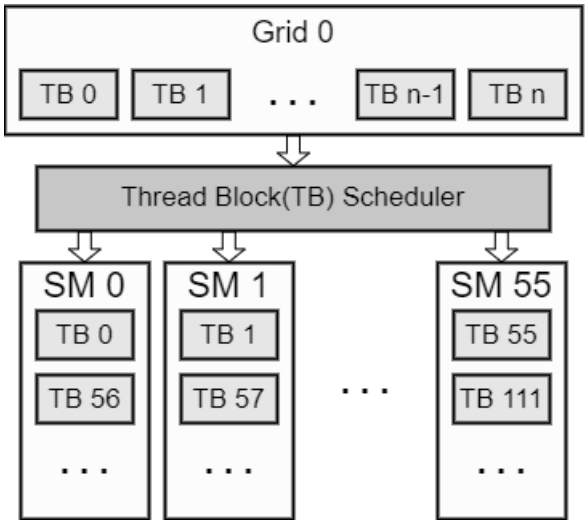


그림 2.1 Thread Block의 SM 내 스케줄링 예시

그림 2.1는 GPU를 수행하는 단위인 스레드가 스레드 블록으로 묶여 SM에 스케줄링이 되는 모습을 표현한다. 스레드 블록은 그리드 단위로 묶여 존재한다. 하나의 스레드 블록에 있는 스레드들은 32개의 스레드로 구성된 워프로 같은 명령어가 동시에 수행된다. 이러한 GPU의 특성으로 인해 스레드, 워프, 스레드 블록 사이에서 데이터 지역성이 나타난다. 데이터 지역성은 intra-locality과 inter-locality로 나눌 수 있다. Intra-locality는 스레드 블록내에서 발생하는 locality을

의미하며, 스레드 블록 내에 있는 스레드 중 같은 메모리 주소에 접근하는 스레드가 많다면 그 스레드 블록은 intra-locality이 높다고 할 수 있다. Inter-locality 경우 다른 스레드 블록에 있는 스레드이지만 접근하는 메모리 주소가 겹칠 경우에 생기는 지역성이다. Inter-locality가 높다면 모든 스레드가 접근 가능한 L2 캐시 수준에 영향을 끼칠 수 있다. 이러한 지역성을 워크로드로부터 분석하여 워크로드의 특성을 파악하는 연구들이 진행되어 왔다. [3–7, 9].

2.1 데이터 지역성 정적 분석

LocalityGuru [3]는 PTX 코드를 분석하여 스레드 블록간의 메모리 접근이 겹치는 정도를 locality로 정의하고, 스레드 블록간에 지역성 정도를 분석한다. 그림 2.1을 보면 GPU는 하나의 SM(streaming multiprocessor)에 여러 스레드 블록이 배치되는 것을 알 수 있다. 따라서 하나의 SM에 배치된 스레드 블록들의 경우 SM 내에 있는 L1 캐시를 공유하게 되고 이로 인해 스레드 간의 locality가 L1 캐시에도 영향을 끼칠 수 있다. 이러한 특성을 고려하여 위 논문은 스레드 블록간 데이터 재사용 정도를 파악한 스레드 블록간의 inter-locality를 분석한다. 하지만 위 논문는 SM간 inter-locality를 파악하는 것에 초점을 두어 지역성을 분석하였지만,

이를 실제 메모리 계층의 L1 혹은 L2 캐시와 연관지어 분석하지 않았다. 다만, 위 논문에서도 SM 내의 스레드 블록, 스레드, 워프간의 데이터 재사용률이 높다면 L1 캐시에 올라가 있는 값도 추방되기 전에 더 많이 활용될 것이라고 언급한 바 있다. PAVER [9]는 위 논문의 결과인 지역성 그래프 정보를 스케줄링에 사용하는 것으로 확장한 논문이다. PAVER [9]에서는 locality를 SpSocre와 Degree of Sharing으로 지수화하여 inter-locality 정도를 표현한다. 그 지수화는 SpSocre와 Degree of sharing이 있는데SpScore는 다음과 같이 계산하며,

SpScore = 1- (non-zero elements in input matrix /total elements in input matrix)

Degree of sharing은 다음과 같이 수식화한다.

Degree of sharing = number of TB with locality / number of total TB

위의 수식은 SpScore의 경우 입력 데이터의 분산성을 표현하며, Degree of Sharing은 데이터 공유가 있는 스레드 블록이 얼마나 높은 강도로 지역성을 갖는지를 지수화한 것이다. 위 논문은 이러한 데이터 지역성 정의를 통해 스레드 블록 스케줄링을 하고자 하였다.

제 3 장 본 론

3.1 연구의 내용

본 연구에서는 PTX 코드를 기반으로 데이터 지역성과 coalescing 정도를 분석하는 하여 캐시 사용 양상과의 연관성을 분석한다.

```
bpnn_adjust_weights_cuda(
ld.param.u64 %rd4, [param_0];
ld.param.u32 %r2, [param_1];
ld.param.u64 %rd5, [param_2];
ld.param.u64 %rd6, [param_4];
ld.param.u64 %rd7, [param_5];
cvta.to.global.u64 %rd1, %rd6;
shl.b32 %r3, %r2, 4;
add.s32 %r4, %r3, 16;
mov.u32 %r5, %ctaid.y;
add.s32 %r6, %r2, 1;
mov.u32 %r7, %tid.y;
mov.u32 %r1, %tid.x;
...
cvta.to.global.u64 %rd3, %rd7;

...
add.s32 %r14, %r1, 1;
...
mul.wide.s32 %rd16, %r14, 4;
add.s64 %rd17, %rd3, %rd16;
ld.global.f32 %f11, [%rd17];
```

그림 3.1 Backprop PTX 코드 예시

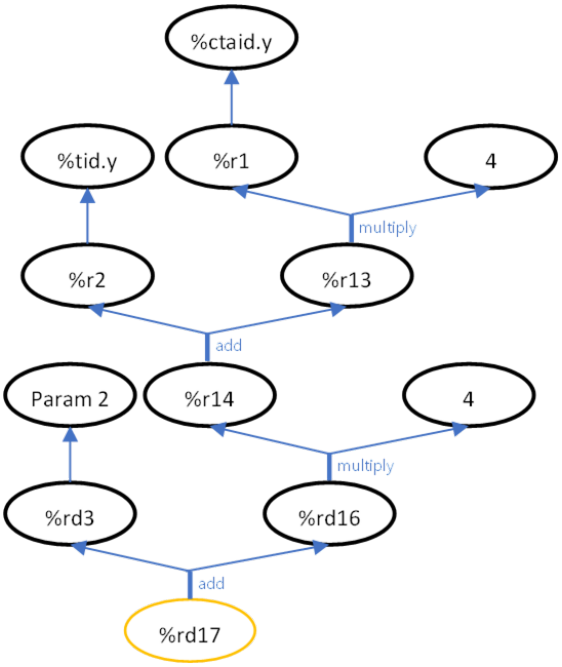


그림 3.2 Backprop syntax tree

3.1.1 PTX 코드를 통한 데이터 지역성 분석

본 논문은 LocalityGuru [3]에서 제안한 PTX 코드 기반 지역성 분석 방법론을 기반으로 하여 정적 프로파일링을 수행한다. 논문 [3]에선 스레드 블록 스케줄링을 위한 스레드 블록간의 inter-locality 지역성를 중점으로 분석하였다. 하지만 본 논문에서는 지역성 분석 범위를 워프 수준의 coalescing으로 좁혀, PTX 코드 기반 워프 내의 스레드간 coalescing 정도를 분석한다.

```
"ld.global.f32 %f11, [%rd17];": [
        {   "reg": "%rd17",
            "opcode": "add.s64",
            "parent": -1
        },
        {   "reg": "%rd3",
            "opcode": "cvta.to.global.u64",
            "parent": 0
        },
        {   "reg": "%rd16",
            "opcode": "mul.wide.s32",
            "parent": 0
        },
            ...
        {   "reg": "[param_5]",
            "opcode": "",
            "parent": 3
        },
            ...
        {   "reg": "%tid.x",
            "opcode": "",
            "parent": 7
        }
    ],
```

그림 3.3 Backprop Syntax tree in json type

3.1.2 Load global 기반 syntax tree 생성 방법

지역성 그래프는 syntax tree를 기반으로 생성된다. Syntax tree는 ptx 코드를 해석하는 과정에서 만들어진다. PTX 코드는 여러 명령어의 나열로 구성되어 있고 그 예시는 그림 3.1과 같다. 이 PTX 코드를 트리로 표현하면 그림 3.2과 같이 표현되며, 트리를 표현하기 위한 자료구조는 그림 3.3와 같다. PTX 명령어들은 아래와 같이 구성되어 있다.

operation.type dst, src[1-3]

수행하고자 하는 명령어인 operation, 수행결과를 저장할 destination, 그리고 명령어를 수행할 source로 구성되어 있다. Source의 경우 operation 종류에 따라 1개에서 3개까지 있을 수 있다. Syntax tree는 PTX 코드 중 한 명령어을 정하고 그 명령어의 destination이 어떤 source로 어떻게 생성되는 지를 따라가며 생성한다. 논문 [3]에서는 메모리 접근의 지역성을 파악하고자 했기 operation이 "ld.global"인 명령어들을 기준으로 syntax tree를 생성한다. "ld.global" 명령는 PTX 명령 중 메모리에 접근하여 읽어오는 명령어이기 때문에 PTX 코드에서 load global(ld.global) operation을 찾고 이 명령어의 메모리 접근 주소가 어떻게 생성되는지 PTX 코드를 추적하며 syntax tree를 구성해간다. PTX 코드를 추적해가며 접근 주소를 찾는 과정은 kernel의 입력 파라미터, 입력 행렬 혹은 스레드 id, 스레드 블록 id 등 고정 파라미터를 만날 때까지 반복한다. 이렇게 추적이 완료된 syntax tree를 통해 스레드가 접근하는 메모리 주소에 대한 정보를 알아낼 수 있다. 그림 3.1는 Rodinia 벤치마크 [10] 중 backprop 워크로드의 PTX 명령 중 일부를 가져온 것이다. 위 명령 중 글로벌 메모리에 접근하는 레지스터인 rd17이 추적의 기준이 되어 추적을 시작한다. rd17을 찾았다면, 그 다음은 이 레지스터가 목적지인 "add.s64 rd17, rd3, rd16" 명령을 찾는다. 이후 이 명령의 source

레지스터인 rd3과 rd16에 대해 동일한 행동을 수행한다. 이러한 과정을 모든 load 글로벌 명령에 대해 수행하여 PTX 코드에 있는 모든 load global 명령을 추적하여 syntax tree를 생성한다.

3.1.3 Syntax tree 기반 지역성 그래프 생성 방법

이렇게 만들어진 syntax tree를 사용하여 실제 ld.global 명령으로 접근하는 주소를 파악하기 위해선 syntax tree를 리프 노드에서부터 추적하여 스레드가 접근하는 주소를 계산하는 것이 필요하다. 리프 노드의 레지스터 값은 스레드 ID(tid), 스레드 블록 ID(ctaid), 스레드 개수(ntid) 값 혹은 사용자의 입력 행렬의 주소값이다. 따라서 고정되어있는 값이거나 사용자가 지정해주는 파라미터에 대해서는 실제 값을 대입하여 연산에 맞는 계산을 수행하면 메모리 접근 주소값을 알아낼 수 있다. 이때, 리프 노드가 ctaid.x, ctaid.y, tid.x tid.y라면 그 값은 고정값이 아닌 자신이 속한 스레드 블록의 id, 그리고 자신의 스레드 id에 따라 달라는 변동값이다. 그리고 그 범위는 ctaid의 경우 0부터 그리드 크기-1까지고, tid의 경우 0부터 스레드 블록 크기-1로 범위가 형성된다. 그림 3.1과 그림 3.3의 backprop의 syntax tracing의 결과예시에서 살펴보면, r17을 추적했을 때 그 결과는

add(mul(add(tid.x, 1), 4) , param5)

가 된다. Rodinia의 Backprop의 경우 입력 크기가 1024일 때 tid.x의 범위는 0에서 15이고, 이 값과 param5의 입력 행렬의 시작 주소를 넣으면 스레드가 접근하는 주소를 알 수 있다. 이 과정을 통해 스레드 파라미터에 따라 접근하는 메모리 주소와 그 메모리 주소에 다른 스레드도 접근하는지를 파악할 수 있다.

3.1.4 PTX 코드를 통한 워프 수준 coalescing 접근 분석

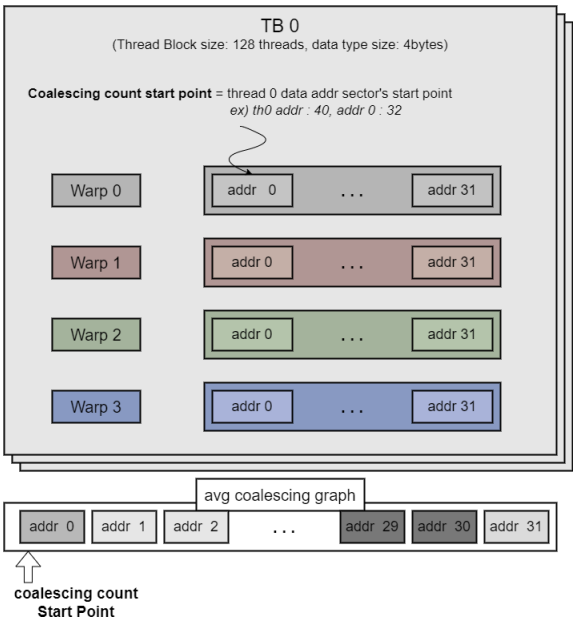


그림 3.4 coalescing 그래프 생성 과정

PTX 코드를 통해 스레드가 접근하는 글로벌 메모리 접근 주소를 알아낸 뒤 이를 토대로 워프내의 스레드가 접근하는 메모리 주소의 지역성을 분석한다. 워프 수준에서의 지역성을 분석하는 것은 워프내에 존재하는 32개 스레드의 지역성을

분석하는 것이다. 이 32개의 스레드의 지역성을 분석하기 위해선 이들의 지역을 정의하는 것이 필요하다. 워프 내의 coalescing을 판단하는 지역을 coalescing 범위라고 정의한다. Coalescing 범위는 워프의 첫번째 스레드가 접근하는 섹터의 시작 위치부터 128 바이트 이내의 곳까지이다. 섹터란 32 바이트 묶음을 의미한다. 본 연구에서는 데이터 타입 크기에 따라 접근 범위가 달라지기 때문에 데이터 타입의 크기를 4 바이트로 고정하여 coalescing을 분석하였다. Coalescing 범위는 그림 3.4에서도 설명하듯 coalescing 그래프의 시작점은 워프의 첫번째 스레드가 접근하는 주소가 아닌 그 주소가 포함된 섹터의 시작점을 말한다. 이와 같이 정의한 이유는 GPU의 캐시는 sectored 캐시로, 데이터 접근이 필요할 때 섹터 단위로 글로벌 메모리에서 데이터를 읽어올때 해당 데이터만 읽어오는 것이 아닌 해당 데이터가 포함된 전체 섹터를 읽어오게 된다. 따라서 접근 데이터의 섹터의 시작점이 coalescing graph를 생성하기 위한 시작점이 되고, 섹터의 시작점은 접근 주소를 32로 나눈 값의 몫을 32로 곱한 값을 의미한다.

이 시작점을 기준으로 해당 워프 내에 있는 모든 스레드들이 그 시작점에서 32개의 기준점에 대해 해당 워프 내에 있는 모든 스레드들이 그 시작점에서 128 바이트 이내에 있는 곳에만 접근하는 스레드의 개수를 세어 coalescing 그래프을

생성한다.

Degree of coalescing = sum(# of coalescing) / (line_cnt *# of global inst)

이후 전체 스레드 갯수를 32개로 나눈 수와 동일한 line_갯수와 global 메모리에 접근하는 명령어의 개수로 나누어 degree of coalescing을 생성하고 위 정보를 그래프로 표현한다. Coalescing 그래프는 워프의 첫번째 스레드가 접근하는 섹터의 시작점을 기준으로 32개의 데이터 내에 스레드 접근이 어떻게 되는지를 보여주는 그래프이다. 이 그래프를 통해 워프의 coalescing 정도와 섹터를 몇개 접근하게 될 지 예측할 수 있다. 데이터 타입의 크기가 4바이트일 경우 coalescing 그래프가 표현하는 섹터의 개수는 4개이다. 이에 기반하여 워프가 접근하는 섹터 개수를 예측할 수 있다. 8개의 데이터 묶음을 하나의 섹터로 하여 그 sector에 접근하는 스레드가 존재한다면 그 섹터는 스레드가 접근한 곳으로 판단한다.

제 4 장 실 험

4.1 실험 환경

NVIDIA A30 Spec	
L1 cache size	192 KB
L2 cache size	24 MB
Memory	24 GB
Compute capability	8.0

그림 4.1 NVIDIA A30 spec

실험은 NVIDIA의 A30 gpu를 사용하였다. A30의 memory 계층은 그림 7.과 같이 구성된다. A30의 L1 캐시 크기는 192KB이고, 하나의 SM내에 하나씩 들어 있어 같은 SM 내에 스케줄링된 스레드만 접근할 수 있는 캐시이다. A30의 L2 캐시는 24MB이며 워크로드의 모든 스레드가 접근할 수 있다. 한번에 접근하는 데이터 크기는 cuda compute capability에 따라 달라지게 되는데, 본 머신의 경우 8.0 버전으로 32 바이트의 segment로 묶어 글로벌 메모리에 접근하게된다. 데이터 타입의 크기가 4 바이트일 경우 글로벌 메모리 load 요청에 대한 이상적인 섹터

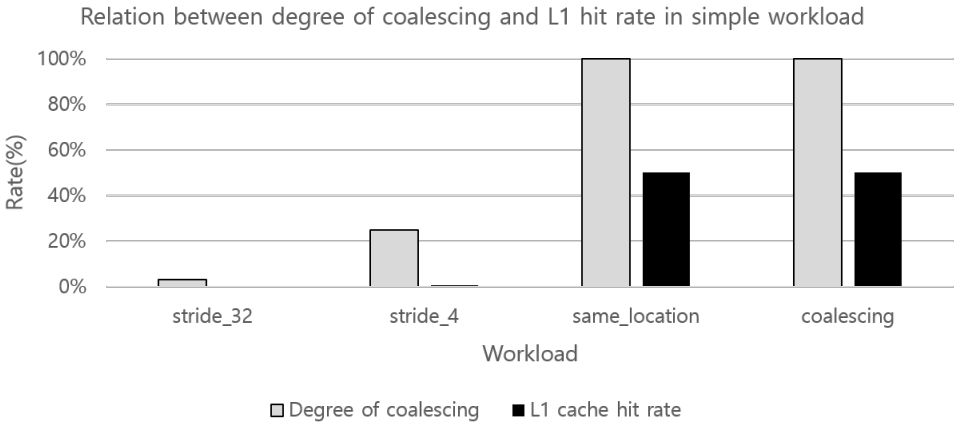
접근 횟수는 4회이다. [1] 같은 데이터에 많이 접근하는 양상이 있다면 이 횟수는 줄어들 수 있으며 최악의 경우 워프내의 모든 스레드가 다른 섹터를 접근한다면 하나의 글로벌 메모리 요청에 대해 32개의 섹터에 접근하게 되는 상황이 발생할 수 있다. [1]

4.2 L1 캐시와 coalescing의 관계

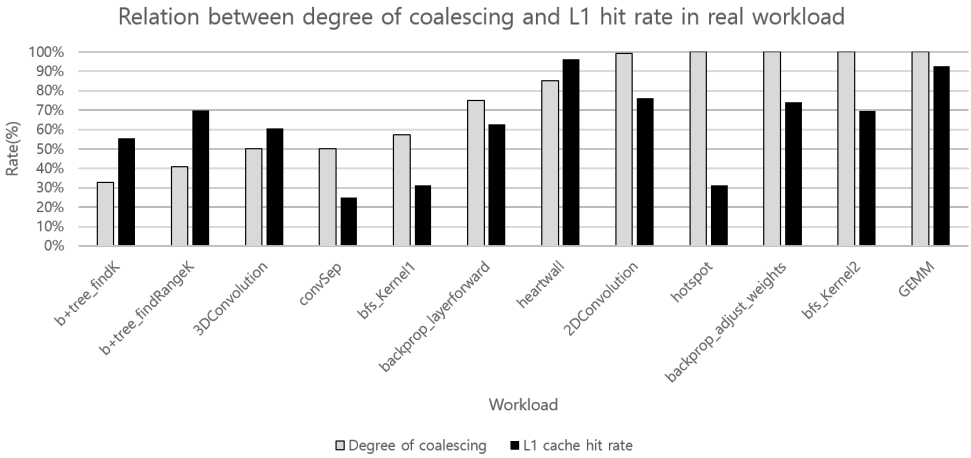
application_name	grid_x	grid_y	thread_x	thread_y
stride-32	32	1	64	1
stride-4	32	1	64	1
same-location	32	1	64	1
coalescing	32	1	64	1
b+tree_findK	60	1	256	1
b+tree_findRageK	60	1	256	1
3DConvolution	4	16	16	4
convSep	1	32	16	4
bfs_kernel1	128	1	512	1
backprop_forward	1	64	16	16
heartwall	51	1	64	1
2DConvolution	2	8	32	8
backprop_adjust_weights	1	64	16	16
bfs_kernel2	128	1	512	1
GEMM	2	8	32	8

그림 4.2 workloads

실험을 위해 사용한 워크로드는 4.2와 같다. 그림 4.3(a)에서 수행한 base 워



(a) Relation between degree of coalescing and L1 hit rate in base workload



(b) Relation between degree of coalescing and L1 hit rate in real workload

그림 4.3 Relation between degree of coalescing and L1 hit rate

크로드는 지역성과 coalescing의 양상을 단적으로 보여주기 위한 워크로드들이다. Stride 워크로드의 경우 한 워프의 스레드들이 n만큼 건너뛰며 데이터를 접근하는 형태로 데이터 재사용 및 coalescing이 적게 일어나도록 하는 워크로드이다. 따라서

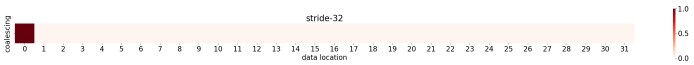
stride_32는 하나의 워프를 구성하는 32개의 스레드가 한번의 메모리 요청로 가져올 수있는 128 바이트 중 공유하는 것이 없는 워크로드이다. Stride_4의 경우 4개의 데이터씩 떨어져가며 주소를 접근하는 것으로 한번의 메모리 요청으로 가져온 128 바이트 중 8개의 데이터를 사용하게 된다. 따라서 stride_4가 stride_32에 비해 더 높은 degree of coalescing값을 가지며 NVIDIA에서 제공하는 nsight compute 프로파일러로 분석했을 때, L1 캐시 hit rate이 stride_4에서 더 높게 나타났다. 이를 통해 같은 데이터를 접근하는 것뿐만 아니라 coalescing도 L1 캐시에 영향을 줌을 알 수 있다. 이에 반해 same_location과 coalescing 워크로드는 데이터 재사용 및 coalescing 정도가 높은 워크로드이다. Same_location의 경우 한 워프 내의 모든 스레드가 한 데이터 주소에 접근하도록 한 워크로드이다. 따라서 이 워크로드의 경우 degree of coalescing이 100%인 것을 볼 수 있다. Coalescing 워크로드 같은 경우에는 한 워프내의 32개 스레드가 각각 자신의 스레드 id에 맞게 접근하게한 워크로드이다. 따라서 이 워크로드는 한 워프 내의 스레드가 모두 다른 데이터 주소에 접근하지만 그 범위가 128 바이트 이내에 있기 때문에 degree of coalescing과 L1 캐시 hit rate이 same_location과 동일한 값을 가진다. 이 base 워크로드를 통해 degree of coalescing이 스레드의 메모리 접근 주소의 양상을 반영하고 있음을

알 수 있다. 하지만 또한 degree of coalescing이 100%라고 해서 캐시 hit rate도 100%가 되는 것은 아니란 걸 알 수 있다. 캐시 hit rate엔 coalescing의 정도뿐만 아니라 스레드 블록 간의 데이터 재사용, 데이터 접근 요청 횟수 등에 의해 변동될 수 있기 때문에 degree of coalescing이 캐시 hit rate과 같은 값을 갖기는 어렵다.

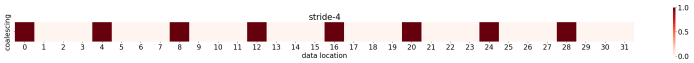
그림 4.3(b)은 rodinia [10] 워크로드의 degree of coalescing과 L1 캐시 hit rate 값이다. 이 그림을 보면, degree of coalescing이 70% 이상일 경우 L1 캐시 hit rate이 60% 이상인 것을 볼 수있다. 이를 통해 degree of coalescing이 높은 워크로드가 L1 캐시 hit rate도 높은 축에 속한다는 것을 알 수 있다. B+tree의 경우 findK 함수와 findRageK 함수가 존재한다. 이 두 함수 모두 degree of coalescing이 32.66%, 40.63%로 낮은 축에 속한다. 하지만 그에 반에 cache hit은 55.56%,69.73%으로 degree of coalescing에 비해 높은 값을 보인다. 이는 데이터 밀집 정도로 인한 것으로 보인다. 그림 4.7과 같이 findK 함수의 경우 L1 sector / request가 2.14이고 findRangeK의 경우 2.08로 optimal 섹터 접근 개수인 4개보다 적은 섹터를 접근하는 것을 알 수 있다. 이는 4.5(a)와 4.5(b)를 봤을 때 앞의 2개의 섹터에 접근이 집중되어 있는 것으로 분석된다. 이에 따라 실제 섹터 접근 개수도 4개 이하로 나타남을 알 수 있다. 따라서 특정 데이터에 많이 접근하는 b+tree의

특성으로 인해 낮은 degree of coalescing 임에도 상대적으로 높은 캐시 hit rate 이 나온다고 분석된다. 또한 같은 application의 다른 두개의 함수끼리 비교해봤을 때, b+tree, backprop, bfs 모두 degree of coalescing이 높은 함수가 캐시 hit rate 도 높은 것으로 나오는 것을 볼 수 있다.

4.3 Degree of coalescing 그래프를 통한 분석

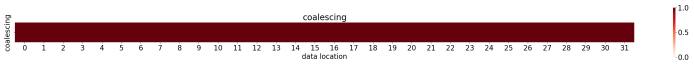


(a) Coalescing Graph - stride_32



(b) Coalescing Graph - stride_4

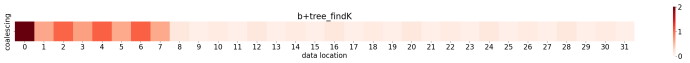
(c) Coalescing Graph - same_location



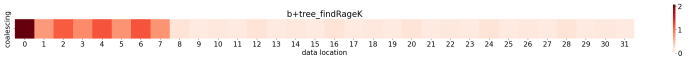
(d) Coalescing Graph - coalescing

그림 4.4 Coalescing Graph Graph in simple workload

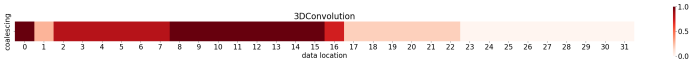
그림 4.5은 Degree of coalescing이 어떻게 형성되는 지와 한 워프 내에서의 데이터 접근 밀집도를 예측해볼 수 있다. NVIDIA profiling guide에서도 언급하듯 한 워프 내의 스레드들이 같은 data address에 접근하는 것은 sector/request를 최적의 개수인 4 이하로 낮출 수 있다고 한다. 그림 4.5와 4.6를 보면 실행한 워크로드



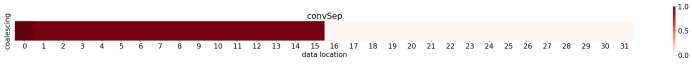
(a) Coalescing Graph - b+tree_findK



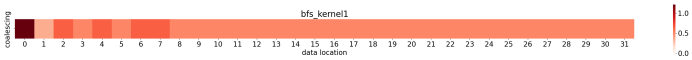
(b) Coalescing Graph - b+tree_findRangeK



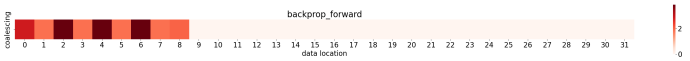
(c) Coalescing Graph - 3DConvolution



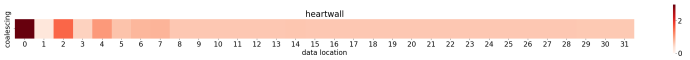
(d) Coalescing Graph - convSep



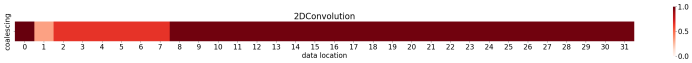
(e) Coalescing Graph - bfs_Kernel1



(f) Coalescing Graph - backprop_forward

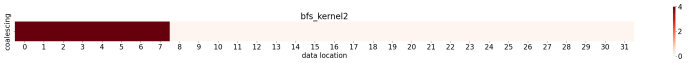


(g) Coalescing Graph - heartwall

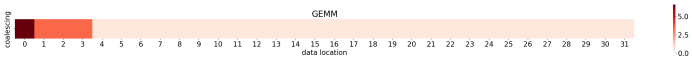


(h) Coalescing Graph - 2DConvolution

(i) Coalescing Graph - backprop_adjust_weights



(j) Coalescing Graph - bfs_kernel2



(k) Coalescing Graph - GEMM

그림 4.5 Degree of coalescing Graph in real workload

Kernel Name	Degree of Coalescing	Sector Access per warp	Sector Access expectation	L1 sectors per request
stride-32	3.13%	1	32	32
stride-4	25.00%	4	16	16
same-location	100.00%	1	1	1
coalescing	100.00%	4	4	4
b+tree_findK	31.95%	4	12.51	2.15
b+tree_findRageK	39.51%	4	10.12	2.21
3DConvolution	50.00%	3	6	4.62
convSep	48.83%	2	4.09	4
bfs_kernel1	53.57%	4	7.47	4.91
backprop_forward	75.00%	2	2.67	2.89
heartwall	74.27%	4	5.39	3.33
2DConvolution	90.01%	4	4.44	4.33
backprop_adjust_weights	100.00%	3	3	3.21
bfs_kernel2	100.00%	1	1	1
GEMM	100.00%	4	4	2.51

그림 4.6 Degree of coalescing, sector expectation and sector/request

중 워프 내 데이터 접근 밀집도가 높게 나타나는 same_location, b+tree_findK 그리고 b+tree_findRangeK, bfs_kernel2 등을 보면 이 워크로드들의 sector/request가 4보다 이하인 것을 알 수 있다. 이는 data가 같은 곳에 접근하다 보니 request 대비 접근해야하는 섹터의 수가 줄어든 것을 알 수 있다. Same_location 워크로드의 경우 모든 스레드가 한 곳에 접근하기 때문에 접근하는 섹터 또한 1개만 접근하면 된다. B+tree_findK와 findRangeK의 경우 각각 10.45개와 13개의 스레드가 첫번째 스레드가 읽은 데이터를 동일하게 접근하였다. 하지만 이를 제외한 나머지 이 시작 데이터로부터 128 바이트 넘는 주소에 대한 request를 하였다. 하지만 4보다 낮은 sector/request로 미루어 보아, 나머지 스레드 또한 그 안에서 coalescing이 있었을 것으로 예상할 수 있다.

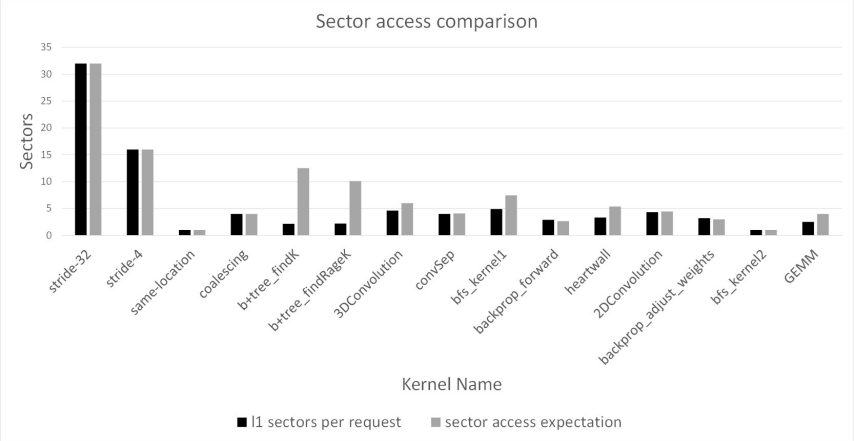


그림 4.7 Sector access expectation과 L1 sector/request

앞서 분석한 Degree of coalescing과 coalescing 그래프에서 파악할 수 있는 워프 내 섹터 접근 개수(sector access per warp)를 기반으로 섹터 접근 개수를 예측한 sector access expectation을 추출하였다. 섹터 접근 예측값을 파악하기 위해선, 128 바이트 범위 내 섹터 접근 개수 x 1/Degree of coalescing으로 계산한다. 이는 워프의 coalescing 범위에서 접근하는 섹터 접근 개수와 이러한 128 바이트 범위가 몇개 필요한지를 degree of coalescing의 역으로 계산하여 알아내는 값이다. 이를 토대로 하나의 warp에서 접근하게 될 섹터의 개수를 예측한다. 물론 워크로드 내의 여러 글로벌 메모리 접근 명령이 존재하고 각 글로벌 접근별로 coalescing 및 밀집도가 다를 수 있기 때문에 실제 L1 sector per request와는 차이는 보인다. 하지만 베이스 워크로드의 경우 동일한 형태의 글로벌 메

모리 접근을 하기때문에 예측 섹터 접근 개수와 실제 NVIDIA nsight compute 로 프로파일링한 L1 sectors per request 결과가 동일하게 나타났다. 실제 워크로드들 중에서도 convSep, backprop_forward, 2DConvolution, backprop_forward, bfs_kernel2는 예상 섹터 접근 수치와 실제 L1 sectors per request 값간의 오차가 1 섹터 미만으로 나타났다. 또한 4 섹터를 초과하는 워크로드인 3DConvolution과 bfs_kernel1,에 대해서는 그 접근 섹터의 개수가 4를 초과한다는 사실이 일치하는 프로파일링을 하였다. 이러한 결과를 통해 degree of coalescing은 L1 cache hit의 결과와 연관성이 있음을 알 수 있고, degree of coalescing과 coalescing 그래프로 파악한 워프 내 섹터 접근 개수를 통해 실제 L1 sector/request와 유의미한 관계가 있음을 보였다.

제 5 장 결 론

5.1 결과 분석

본 논문은 GPU 워크로드의 데이터 지역성을 PTX 코드 분석을 통해 파악한다. 스레드 블록 수준의 데이터 지역성 분석은 데이터 지역성을 잘 표현하기는 하나, 실제 L1 캐시의 성능과는 연관성이 약했다. 이러한 한계를 극복하고자 데이터 지역성을 분석하는 수준을 스레드 블록에서 워프로 더 좁혀서 분석하였다. 워프의 데이터 지역성에서 주요하게 고려할 것은 coalescing 정도와 데이터 접근의 밀집도이다. 본 논문은 PTX 코드를 통해 스레드가 접근하는 주소를 파악할 수 있다는 점을 통해 워프 내에 스레드들의 coalescing 정도를 파악하고 coalescing 범위 내에서 데이터 접근이 얼마나 밀집되어 있는 지를 분석하였다. 이러한 분석을 각각 degree of coalescing과 sector access expactation로 정의하였다. 수행한 11개의 rodinia [10] 워크로드에서 degree of coalescing의 값이 70%이상인 6개의 워크로드 중 6개가 L1 캐시 hit rate 60%이상의 값을 보였다. 이를 통해 degree of coalescing 값이 높을 때 L1 캐시 hit rate도 높은 경향을 보임을 확인하였다. 또한 degree of

coalescing과 sector access per warp를 통해 알아낸 sector expectation이 실제 sectors/request와 높은 연관성을 보임을 확인했다. 이를 통해 워프 내의 스레드가 coalescing한 접근을 하는 지와 한 데이터 위치에 여러 스레드가 접근하는 양상이 있을 경우 캐시 hit과 sectors/request에 영향이 가는 것을 정적 프로파일링을 통해 알 수 있었다. 이러한 프로파일링은 정의한 수치뿐만 아니라 coalescing 그래프로 표현하여 시각화하였다. 향후 연구로는 위와 같이 분석한 지역성을 토대로 데이터 접근 양상이 다른 워크로드를 co-scheduling하는 스케쥴러로 확장될 수 있다.

참고문헌

[1] "BEST Practices Guide :: CUDA Toolkit Documentation." https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#device-memory-spaces.

[2] "nsight compute profiling tool." https://docs.nvidia.com/nsight-compute/ProfilingGuide/, 2022.

[3] D. Tripathy, A. Abdolrashidi, Q. Fan, D. Wong, and M. Satpathy, "Localityguru: A ptx analyzer for extracting thread block-level locality in gpgpus," in *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–8, IEEE, 2021.

[4] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware cta clustering for modern gpus," 2017.

[5] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic gpu cache bypassing," 2015.

[6] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient gpu architectures," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 86–98, IEEE, 2013.

[7] S. Lal, B. S. Varma, and B. Juurlink, "A quantitative study of locality in gpu caches for memory-divergent workloads," *International Journal of Parallel Programming*, vol. 50, no. 2, pp. 189–216, 2022.

[8] X. Tang, A. Pattnaik, O. Kayiran, A. Jog, M. T. Kandemir, and C. Das, "Quantifying data locality in dynamic parallelism in gpus," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 3, pp. 1–24, 2018.

[9] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, pp. 1–26, 2021.

[10] 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 *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44–54, Ieee, 2009.

[11] "Polybench: The polyhedral benchmark suite." `http://www.cs.ucla.edu/pouchet/software/polybench`, 2012.

[12] "Cuda refresher: The cuda programming model." `https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/`.

[13] "Cuda toolkit document." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[14] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 829–842, IEEE, 2018.

Abstract

As computational efficiency is known, GPUs are not only used for computer graphics, but also for various workloads such as high-performance computing and machine learning. Accordingly, research to efficiently use limited memory resources of GPUs is being actively conducted. In particular, it is very important to know and utilize the data locality of the workload in order to efficiently use the small but fast-access L1 data cache to handle memory bottleneck problems. Therefore, this paper proposes a scale for analyzing and objectifying workload-specific data locality. To this end, various workloads used by actual GPUs were analyzed through static profiling based on PTX code. As a result of the static analysis, it was confirmed that there was a relationship between the degree of coalescing, coalescing graph, and cache utilization at the actual execution.

Keywords: GPU profiling, Locality, Coalescing, PTX code, L1 cache

Student Number: 2021-26019