https://doi.org/10.22670/knom.2022.25.1.62

GPU 메모리의 접근 다양성을 가진 워크로드에 대한 데이터 지역성 분석 실험

김 지 은◆, 신 현 일*, 엄 현 상*, 김 윤 희°

Empirical experiments of profiled data locality for memory-divergent workloads on GPU

Jieun Kim◆, HyunIl Shin*, Hyeonsang Eom*, Yoonhee Kim°

요 약

GPU의 계산적 효율성이 알려짐에 따라 GPU를 컴퓨터 그래픽스에만 활용하는 것이 아닌 범용적인 응용에서 활용하는 GPGPU(General-Purpose computing on Graphics Processing Units) 수행이 HPC(High Performance Computing), DL(Deep Learning) 등 다양한 워크로드에서 수행된다. 이에 따라 GPU의 제한적인 memory를 효율적으로 이용하려는 연구가 활발하다. 특히 memory-divergent 워크로드의 경우 memory 사용패턴에 따라 성능이 크게 좌우될 수 있다. 본 논문에서는 memory-divergent 워크로드를 프로파일링하여, data locality를 분석하고 data locality를 객관화할 수 있도록 지수화하였다. 그리고 data locality 지수와 실제 캐시의 관계를 분석을 통해 data locality와 캐시 적중률 간에 상관관계가 존재하지만, 캐시의 제한적인 크기로 인해 접근하는 thread 수가 증가하면 그 상관관계가 약해짐을 확인하였다. 따라서 본 논문은 data locality 값을 공식화하고 그 값과 캐시간의 관계를 분석하였다.

Key Words : GPU, Data Locality, Cache, Parallel Thread Execution(PTX), static profiling, memory-divergent workloads

ABSTRACT

As the computational efficiency of GPUs is known, GPGPU performance, which is utilized in general-purpose applications, is performed on various workloads such as HPC and DL. Accordingly, research to efficiently use the limited memory of GPUs is active. In particular, in the case of a memory-divergent workload, performance may depend greatly on the memory usage pattern. In this paper, the memory-divergent workload is profiled to analyze data locality and to index it to objectify data locality. In addition, through analysis of the relationship between the data locality and the cache hit ratio, it was confirmed that there was a correlation between the data locality and the cache hit rate, but the correlation weakened as the number of threads approached increased due to the limited size of the cache. Therefore, this paper formulates the data locality value and analyzes the relationship between the value and the cache.

※이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2021R1A2C1003379).

◆ First Author : Seoul National University Department of Computer Science and Engineering, jieun0116@snu.ac.kr
° Corresponding Author: Sookmyung Women's University Department of Computer Science, yulan@sookmyung.ac.kr
* Seoul National University Department of Computer Science and Engineering, shuin3180@naver.com, hseom@cse.snu.ac.kr

논문번호 : KNOM2022-01-11, Received July 12, 2022; Revised July 28, 2022; Accepted August 9, 2022

Ⅰ. 서 론

GPU가 GPGPU(General-Purpose computing on Graphics Processing Units)로 많이 사용됨에 따라 HPC(High Performance Computing), DL(Deep Learning) 등에서 다양한 응용에서 GPU가 사용된다. 이렇듯 다양한 워크로드가 GPU에서 수행됨에 따라 data 접근 방식에 GPU의 수행방식과 적합하지 않는 워크로드도 존재한다. GPU는 많은 thread가 동시에 명령어를 수행하는 반면 캐시의 크기가 작기때문에 GPU의 자원을 효율적으로 사용하면서 워크로드를 수행하기 위해선 GPU 캐시의 data locality에 대한 고려가 필요하다.

GPU 캐시의 data locality이란 gpu의 명령어 수행방식인 SIMD(Single Instruction Multiple Data)로 인해 찾아볼 수 있는 다중 thread간 locality의 의미한다. GPU는 하나의 명령어에 대해 다수의 thread가 수행한다. 이때 다른 thread들이 서로 같은 data를 접근하는 경우가 생기는데 이러한 것을 GPU 캐시상에서의 data locality라고 한다. 최근 이러한 data locality을 분석하고[2,7-8,13] 이를 고려하여 GPU 스케줄링에 반영하는 연구가 많이 진행되어 왔다.[1,11-12]

특히 논문[7]은 gpu 워크로드 중 memory 접근이 coalescing하지 않은 워크로드를 분석하면서 캐시 크기의 한도에 따른 캐시 라인 사용량 분석 및 L1, L2 캐시 값과 워크로드의 locality값 간의 관계를 분석한다. memory 접근이 coalescing 하지 않은 워크로드는 memory divergent 워크로드라고 명명한다. 이러한 워크로드의 경우 한 warp가 memory 접근 관련 명령을 실행할 때 여러 번의 memory 트랜잭션이 필요하므로 비효율적인 memory 접근을 하게 된다. 따라서 이 논문은 이러한 워크로드를 대상으로 하여 locality 및 gpu 캐시 접근 양상을 분석한다. 이 논문에서는 locality가 발생하는 그룹의 단위는 warp와 thread block으로 나누었고, 각 그룹의 intra-locality와 inter-locality로 나누어 분석한다.

LocalityGuru[12]의 경우 thread block 수준의 data locality를 파악하는 방법을 설명한다. 논문[12]에서는 PTX(Parallel Thread Execution) 코드를 사용하여 워크로드의 locality를 분석한다. PTX 코드에서 memory 접근 명령어를 기준으로 삼아 각 thread 별로 memory에 접근하는 주소를 분석한다. 산출한 결과를 locality 지수로 추출하고 data locality를 평가한다.

본 논문에서는 memory-divergent 워크로드를 프로파일링하여 정적분석을 통해 data locality를 분석하고 data locality를 지수화하여 워크로드의 data locality 정도를 표준화하여 표현한다. 또한, 그 data locality 지수와 실제 실험에서의 캐시와의 관계를 분석한다.

실험은 NVIDIA A30 gpu에서 논문[7]이 말하는 memory-divergent 워크로드를 대상으로 논문[12]의 PTX 코드 기반 locality 분석을 통해 intra-, inter-locality와 gpu L1/L2 캐시간의 관련도 분석을 수행하였다. 그 결과 data locality와 캐시 이용에는 큰 상관관계가 존재하지만, 캐시의 제한적인 크기로 인해 접근하는 thread 수가 증가하면 그 상관관계가 약해짐을 확인하였다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 관련 연구에 대한 설명을 통해 locality와 캐시의 관계 및 locality 분석방식을 설명한다. 3장에서는 PTX 코드 기반으로 locality를 분석하는 자세한 방법론에 대한 것과 locality값을 지수화하는 방법에 대해 말한다. 4장에서는 실험 방법 및 결과에 대해서 설명하고 5장에서 결론을 통해 위 논문이 수행한 점과 향후 연구에 대해 말한다.

Ⅱ. 관련 연구

GPU를 수행하는 단위인 thread는 thread block으로 묶여서 SM(Streaming Multiprocessor)에 스케줄링이 되어 실행한다. Thread block은 grid 단위로 묶여 존재한다. 하나의 thread block에 있는 thread들은 32개의 thread로 구성된 warp로 묶여 같은 instruction이 동시에 수행된다. 이러한 GPU의 특성으로 인해 warp, thread block 사이에서 data locality가 나타난다. Data locality는 intra-locality과 inter-locality로 나눌 수 있다. Intra-locality는 그룹 내에서 발생하는 locality을 의미하며, thread block 내에 있는 thread 중 같은 memory 주소에 접근하는 thread가 많다면 그 thread block은 intra-locality이 높다고 할 수 있다. Inter-locality 경우 다른 그룹에 있는 thread이지만 접근하는 memory 주소가 겹칠 경우에 생기는 locality이다. 이러한 locality을 워크로드로부터 분석하여 워크로드의 특성을 파악하는 연구들이 진행되어 왔다.[1-2,6-7,10-12]

2.1 Memory-Divergent 워크로드에 대한 캐시에서의 data locality 분석

Sohan.Lal[7]은 data locality 및 data coalescing이 GPU 성능에 어떠한 영향을 미치는 지를 실험을 통해 알아본 논문이다. GPU는 높은 계산 능력을 가지고 있지만, CPU에 비해 캐시 크기가 작게 구성되어 있다. 이로 인해 data 이동에 있어서 병목현상이 발생하게 되는데, 특히 분산된 memory 접근은 이를 가중한다. Divergent memory란, 한 warp에 있는 thread들의 요청이 한 번의 memory transaction으로 완료되지 않는 것을 말하며, 이러한 것을 uncoalsced memory access라고 한다. 이러한 워크로드의 경우 특히 inter thread block locality를 활용하기 어렵게 하여 GPU 성능을 저해한다. 따라서 위 논문은 분산된 memory 접근이 GPU 성능에 얼마나 영향을 끼치는지를 분석하기 위해 정량적 분석을 수행한다. data locality의 경우 warp 수준과 thread block 수준으로 크게 구분지었고, 각 수준은 inter-locality와 intra-locality으로 나누어 다양하게 세분화한다.

표 1. locality 정의 [11]
Table 1. locality definition [11]

Locality category	definition
Intra-locality	Locality within Thread Blocks
Inter-locality	Locality between Thread Blocks

본 논문에서 대상으로 하는 locality는 표 1과 같다. 표 1은 PAVER[11]에서 정의한 intra-TB, inter-TB locality와 동일하며 본 논문에서는 이를 intra-locality와 inter-locality로 부른다. 위 논문은 실험을 GPGPU-sim[3]에서 수행하여 L1, L2 캐시 크기를 각각 16KB와 128KB로 설정하여 실험하는 것과 두 캐시 크기 모두를 제한하지 않는 상황을 설정하여 data 분산 및 캐시 크기의 한계로 인한 영향을 알아보고, 캐시 크기에 한계가 없을 경우, 캐시 라인에 올라간 data가 다른 thread에 의해 얼마나 사용되는지를 확인한다.

2.2 data locality 정적 분석

LocalityGuru[12]는 PTX 코드를 분석하여 thread block간의 memory 접근이 겹치는 정도를 locality로 정의하고, thread block간에 locality 정도를 분석한다. 먼저 PTX 코드는 일반적인 용도의 병렬 스레드 실행을 위한 가상 시스템과 ISA(Instruction Set Architecture)를 정의한다.[6] PTX 코드는 CUDA

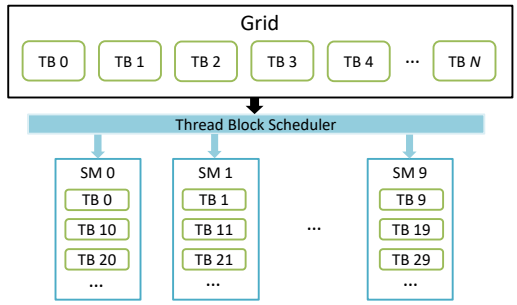


그림 1. Thread Block의 SM 내 스케줄링 예시[5]
Fig. 1. Scheduling example of Thread Block in SM [5]

코드를 실행하기 위해 컴파일할 때 생성되는 어셈블리 코드의 형식을 띈다. 따라서 사용자가 작성한 코드보다 정형성을 갖기 때문에 분석하기에 적합하다. 또한, 그림 1을 보면 Thread Block간의 locality가 어떻게 발생하는 지 알 수 있다. 그림 1.은 thread block이 SM(streaming multiprocessor)에 스케줄링 되는 방식을 보여준다. 하나의 SM에 여러 thread block이 배치될 수 있으며, 따라서 하나의 SM에 배치된 thread block들의 경우 SM 내에 있는 L1 캐시를 공유한다. 이로 인해 thread 간의 locality가 L1 캐시에도 영향을 끼칠 수 있다. 이러한 특성을 고려하여 위 논문은 thread block간 data 재사용 정도를 파악하여 thread block간의 inter-locality를 보여준다.

위 논문[12]은 SM간 inter-locality를 파악하는 것에 초점을 두었고, 이를 L1 혹은 L2 캐시와 연관지어 locality와 캐시의 상관관계에 대해서는 분석하지 않았다. 다만, 위 논문에서도 SM 내의 thread block, thread, warp간의 data 재사용률이 높다면 L1 캐시에 올라가 있는 값도 추방되기 전에 더 많이 활용될 것이라고 언급한 바 있다.

PAVER[11]는 위 논문의 결과인 locality map정element보를 기반으로 스케줄링하는 데에 사용하는 것으로 확장한 논문이다. PAVER[11]에서는 locality를 수치로 표현하기 위해 Sparsity Score(SpScore)와 Degree of Sharing(dos)를 제안하였다.

SpScore는 $\text{SpScore} = 1.0 - \frac{non\text{-}zero\ elements\ in\ matrix}{total\ elements\ in\ matrix}$ 로 표현되어 입력 matrix가 얼마나 sparse한 지를 표현하였다. Degree of sharing은 $degree\ of\ sharing = \frac{shared\ blocks}{shared\ TBs}$ 으로 표현되며 shared TBs는 thread block 중 하나라도 data

reference를 공유하는 thread block을 말한다. Shared blocks은 data reference의 공유가 존재하는 block의 개수를 의미한다.

Ⅲ. PTX 코드 기반 locality 분석

본 논문은 LocalityGuru[12]에서 제안한 PTX 코드 기반 locality 분석 방법론을 사용한다. LocalityGuru에선 inter thread block locality만 분석해내었지만, 본 논문에서는 intra thread block locality도 분석하여 하나의 thread block내에 있는 thread간의 data 재사용 정도를 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 2. Backprop PTX 코드 예시
Fig. 2. Backprop PTX code example

3.1 load global 기반 syntax tree 생성 방법

Locality map은 syntax tree를 기반으로 생성된다. Syntax tree는 PTX 코드를 해석하는 과정에서 만들어진다. PTX 코드는 여러 instruction의 나열로 구성되어 있고 그 예시는 그림 2와 같다. 이 PTX 코드를 트리로 표현하면 그림 3과 같이 표현되며, 트리를 표현하기 위한 자료구조는 그림 4와 같다. PTX instruction들은 아래와 같이 구성되어 있다.

operation.type dst, src(1-3)

수행하고자 하는 명령어인 operation, 수행결과를 저장할 destination, 그리고 명령어를 수행할 source로 구성된다. Source의 경우 operation 종류에 따라 1개에서 3개까지 있을 수 있다. Syntax tree는 PTX 코드 중 한 instruction을 정하고 그 instruction의 destination이 어떤 source로 어떤 연산을 통해 생성되는 지를 따라가며 생성한다. LocalityGuru[12]에서는 memory 접근의 locality을 파악하고자 했기 operation이 “ld.global”인 instruction들을 기준으로

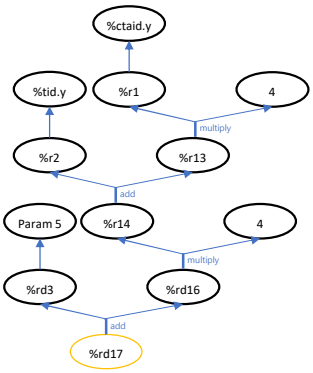


그림 3. Backprop Syntax tree
Fig. 3. Backprop Syntax tree

syntax tree를 생성한다. “ld.global” instruction은 PTX instruction 중 memory에 접근하여 읽어오는 명령어이기 때문에 PTX 코드에서 load global(ld.global) operation을 찾고 이 instruction의 source인 memory 접근 주소가 어떻게 생성되는지 PTX 코드를 추적하며 syntax tree를 구성한다. PTX 코드를 추적해가며 접근 주소를 찾는 과정은 kernel의 input parameter, input array 혹은 thread id, block id 등 고정 parameter를 만날 때까지 반복한다. 이렇게 추적이 완료된 syntax tree를 통해 thread가 접근하는 memory 주소에 대한 정보를 알아낼 수 있다.


```
"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 4. Backprop Syntax tree in json type
Fig. 4. Backprop Syntax tree in json type

그림 2는 Rodinia 벤치마크[4] 중 backprop 워크로드의 PTX instruction 중 일부를 가져온 것이다. 위 instruction 중 global memory에 접근하는

register인 %rd17이 추적의 기준이 되어 추적을 시작한다. %rd17을 찾았다면, 그 다음은 이 register가 destination인 "add.s64 %rd17, %rd3, %rd16" instruction을 찾는다. 이후 이 instruction의 source register인 %rd3과 %rd16에 대해 동일한 행동을 수행한다. 이러한 과정을 모든 load global instruction에 대해 수행하여 PTX 코드에 있는 모든 load global instruction을 추적하여 syntax tree를 생성한다.

3.2 syntax tree 기반 Locality map 생성 방법

이렇게 만들어진 syntax tree를 통해 실제 쓰레드가 ld.global instruction으로 접근하는 주소를 파악하기 위해선 syntax tree를 추적하여 접근 주소를 계산해야 한다. Syntax tree의 leaf node가 갖는 값은 tid, ctaid, ntid 값 혹은 사용자의 input array의 주소값이다. 따라서 위 parameter에 대해서는 실제 값을 대입하여 opcode에 맞는 연산을 수행함으로써 memory 접근 주소값을 알아낼 수 있다. 이때, leaf 노드가 ctaid.x, ctaid.y, tid.x tid.y라면 그 값은 고정값이 아닌 자신이 속한 thread block id, 그리고 자신의 thread id에 따라 달라는 변동 값이다. 그리고 그 범위는 ctaid의 경우 0부터 grid 크기-1까지고, tid의 경우 0부터 thread block 크기-1로 범위가 형성된다. 그림 3과 그림 4의 Backprop의 syntax tracing의 결과 예시에서 살펴보면, %r17을 추적했을 때 그 결과는

$$\mathrm{Param5} + ((\%\mathrm{tid.y} + (\%\mathrm{ctaid.y} \times 4)) \times 4)$$

가 된다. Rodinia[4]의 Backprop의 경우 tid.y와 ctaid.y의 범위 값과 param5의 input array의 시작 주소를 넣으면 thread가 접근하는 주소를 알 수 있다. 이 과정을 통해 thread parameter에 따라 접근하는 memory 주소의 위치와 그 memory 주소에 접근하는 thread들을 파악할 수 있다. 이를 활용하여 thread block간에 memory 접근 주소가 겹치는 정도를 표현한 것이 inter-locality map이고, thread block내에서 memory 접근 주소가 겹치는 정도를 표현한 것이 intra-locality map이다.

3.3 intra/inter TB locality 지수 추출

PAVER[11]에서는 locality map을 활용하여 thread block scheduling을 더 효율적으로 한다. 이때 locality 정도와 접근의 분산성을 정의하기 위해 locality map에서 얻은 정보를 지수화한다. 그 지수화는 SpSocre와 Degree of sharing(dos)이 있는데 SpScore는 다음과 같이 계산하며,

$$SpScore = 1 - \frac{non\ zero\ elements\ in\ input\ matrix}{total\ elements\ in\ input\ matrix}$$

Degree of sharing은 다음과 같이 수식화한다.

$$Degree\ of\ sharing = \frac{\#\ of\ TB\ with\ locality}{\#\ of\ total\ TB}$$

위의 수식은 SpScore의 경우 입력 data의 분산성을 표현하며, Degree of Sharing은 data 공유가 있는 thread block이 얼마나 높은 강도로 locality를 갖는지를 지수화한 것이다. 이는 워크로드의 locality 정도를 표현하는 데 좋은 방식이지만, memory 접근 명령어 대비 locality 정도를 표현한다고 보기 어려워 본 논문에서는 수정한 방식을 사용한다.

본 논문에서는 locality를 지수화하기 위해선 locality가 생기는 tread block 혹은 thread간의 조합으로 비교하며, data 접근 명령어인 ld.global instruction의 개수도 고려하여 일반화한다. 따라서 이를 반영하여 thread block intra-/inter-locality를 지수화한다. Inter-locality degree of sharing(dos)의 경우 전체 inter-locality map에서 locality 값을 thread block의 조합 수, ld.global instruction 수와 thread block의 크기의 곱으로 나누었으며, 아래의 수식으로 표현한다.

$$TH\_count = ntid.x \times ntid.y$$

$$th\_GM = \#\ of\ ld.global\ inst \times TH\_count$$

$$TB\_l = thread\ block\ that\ has\ locality$$

$$Inter\text{-}locality\ degree\ of\ sharing(dos) = \frac{\sum(\#\ of\ inter\text{-}locality)}{comb(TB\_l, 2) \times th\_GM}$$

이때 thread block의 크기인 ntid.x*ntid.y를 해주어 하나의 global load 명령에 대해서 두 thread block이 가질 수 있는 최대 locality 값인 thread block의 크기로 나누어 일반화한다. 또한, 실제 locality가 있는 TB 조합만 세어 캐시 라인에 이미 올라와 있는 주소의 locality가 반영될 수 있도록 한다. Intra-locality degree의 경우에도 inter-locality를 구하는 방식과 유사하며, locality를 구하는 범위가 thread block 내로 한정된다는 차이가 있다. Intra-locality degree of sharing(dos)는 아래와 같이 수식화한다..

$$TH\_l = thread\ \ that\ has\ bcat\quad y$$

$$Intra\ -bcaly\quad degree\quad of\ shari\ ng(dos) = \frac{\sum(\#\ of\ intra\quad -bcaly\quad)}{com\ b\ (TH\_l, 2)\ \times\ \#\ of\ gbbal\quad inst}$$

또한, 한 그룹 내에서 발생하는 locality의 빈도를 표현하기 위해 appearance frequency를 정의한다. Inter-locality appearance frequency는 전체 thread block의 수 대비 locality가 존재하는 thread block 이고, intra-locality appearance frequency는 한 thread block 내에 있는 thread 개수 대비 locality가 존재하는 thread의 개수이다. 이를 표현한 식은 다음과 같다.

$$Inter\text{-}locality\ appearance\ frequency = \frac{count(locality\ existing\ thread\ block)}{total\ thread\ block}$$

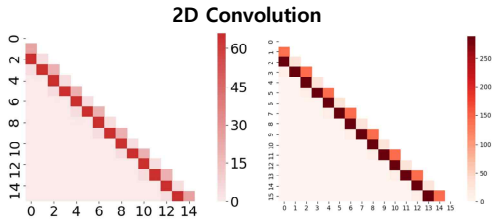
$$Intra\text{-}locality\ appearance\ frequency = \frac{count(locality\ existing\ thread)}{total\ thread\ per\ thread\ block}$$

Ⅳ. 실 험

본 실험은 data locality와 실제 캐시 값 간의 상관관계를 분석한다. 위 논문에서 LocalityGuru[12]를 참조하여 만든 PTX 분석기가 논문[12]과 유사하다고 볼 수 있는지를 판단하고, 이 PTX 기반 memory 접근 locality의 지수를 추출하고 그 분석이 실제 A30의 캐시 사용 패턴 간의 관계를 보인다.

4.1 LocalityGuru[12]의 locality map과 본 논문에서 생성한 locality map 비교

LocalityGuru[12]에서 생성한 locality map과 본 논문에서 구현한 locality map을 비교하였다. 그림 5는 localityGuru에서 분석한 polybench[10]의 2Dconvolutional을 분석한 localityGuru[12]의 locality map 결과와 본 논문에서 수행한 inter-locality map 결과이다. 두 그림을 보면 thread block간에 locality가 발생하는 패턴 양상이 유사하다는 것을 알 수 있다. 하지만 두 map간에 locality의 정도가 다르게 나타나는 차이를 보인다. 이 원인은 한 thread block에서 같은 행 또는 열이 모두 같은 memory 주소에 접근할 경우, 본 논문은 그 행



(a) **2D Convolution** locality map made by LocalityGuru[11] (b) **2D Convolution** locality map made by our paper

그림 5. LocalityGuru[12]의 locality map과 본 논문에서 생성한 locality map 비교
Fig 5. Comparison of locality map between LocalityGuru[12]'s and our's

또는 열의 개수만큼 세었지만, localityGuru[12]는 이를 한 번의 memory 접근으로 본 것으로 예상된다. 이러한 차이는 앞서 locality 값을 thread block의 크기와 instruction의 횟수로 나누어 상쇄함으로써 locality 값을 지수화하는 것으로 보완하였다.

4.2 워크로드 선정

Locality를 분석하고 이를 캐시 값과 비교할 워크로드는 논문[7]의 "L2 캐시 사이즈에 따른 워크로드 캐시 라인 사용 양상 실험" 결과를 참고하여 선정하였다. 논문[7]에서는 실험한 23개의 워크로드 중 캐시 크기를 한정했을 때와 무한정으로 했을 때를 비교하였다. 실험한 워크로드 중 수행 커널을 특

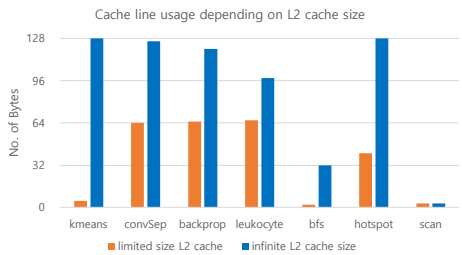


그림 6. L2 캐시 크기에 따른 캐시 라인 활용도[7]
Fig 6. Cache line usage depending on L2 cache size[7]

정할 수 없는 9개를 제외하고 캐시 크기에 따른 캐시라인 사용량에 따른 그룹을 나누어 그림 6의 총 6개 워크로드를 선정하였다. 12개의 워크로드 중 grid와 thread block 크기의 구성이 다양한 6개를 선정하여 워크로드의 특성 및 thread의 특성을 반영하여 실험하도록 워크로드로 선정하였다. L2 캐시 크기에 따른 캐시 라인 사용량의 차이가 큰 그룹으로는 kmeans, backprop, leuokcyte, convolutionSeparble(conSep), hotspot을 선정하였고, 차이가 적은 그룹으로는 bfs을 선정하였다. 그리고 둘 간의 차이가 거의 없는 워크로드로 scan을 선정

표 2.워크로드 특성
Table 2. workload feature

WORKLOAD	GRID x size	GRID y size	THREAD BLOCK x size	THREAD BLOCK y size	# of Thread	# of ld.global inst
kmeans	49	1	256	1	12544	5
convSep	1	32	16	4	2048	20
backprop	1	64	16	16	16384	20
leukocyte	36	1	320	1	11520	10
bfs	128	1	512	1	65536	14
hotspot	43	43	16	16	473344	1
scan	26	1	256	1	6656	2

하여 실험하였다. 각 워크로드의 grid 크기 및 thread block 크기는 표 2를 보면 알 수 있다.

4.3 실험 환경

실험은 NVIDIA의 A30 gpu를 사용하였다. A30의 memory 계층은 그림 7과 같이 구성된다. A30의 L1 캐시 크기는 192KB이고, 하나의 SM내에 하나씩 들어있다. 또한, 같은 SM 내에 스케줄링된 thread만 접근할 수 있는 캐시이다. A30의 L2 캐시는 24MB이며 워크로드의 모든 thread가 접근할 수 있다.

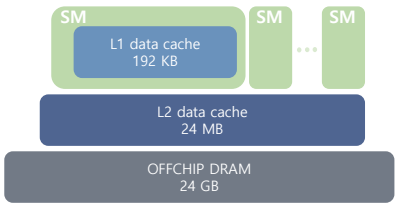


그림 7. A30 의 메모리 계층
Fig. 7. Memory hierarchy of A30

4.4 실험 결과

그림 8은 inter-locality map의 값을 바탕으로 계산한 inter-locality 지수값인 inter-locality degree of sharing(dos), inter-locality appearance frequency 값과 그 워크로드를 A30에서 수행했을 때 나오는 L2 캐시의 값이다. 위 그림을 봤을 때 scan 워크로드를 제외하고는 모두 60% 이상의 L2 캐시 hit을 보인다. 이는 논문[7]에서도 언급하고 있는 바와 같이 L2 캐시의 경우 워크로드를 수행하는 모든 thread가 접근할 수 있으므로, 그에 따라 한번 올라간 캐시 라인이 재사용될 확률이 L1 캐시보다 높기 때문에 나타나는 양상으로 볼 수 있다. 워크로드별로 살펴보면, kmeans와 convSep, hotspot의 경우 inter-locality dos에 비해 L2 캐시 hit ratio가 낮은 것을 볼 수 있다. 이는 이 워크로드가 갖는 locality 값이 캐시의 한정된 크기로 인해 반영이 다 되지

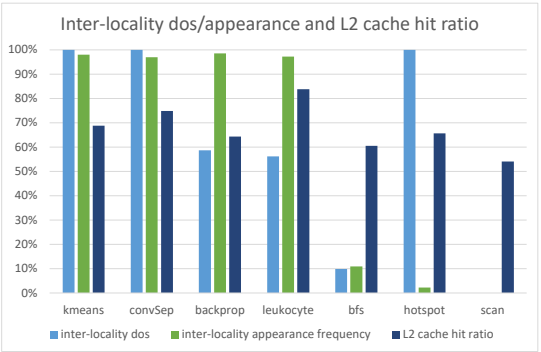


그림 8. inter-locality dos/appearance frequency와 L2 캐시 hit ratio
Fig 8. inter-locality dos/appearance frequency and L2 cache hit ratio

않아 나오는 결과이다. 이에 반에 다른 워크로드는 inter-locality dos가 실제 L2 캐시 hit ratio보다 낮은데, 이는 locality 외에 캐시에 영향을 주는 인자가 있을 것으로 예상할 수 있다. 영향을 주는 인자로는 coalescing한 접근을 들 수 있다. coalescing한 접근이 있을 경우, 같은 주소를 접근하는 것이 아니기 때문에 locality로 고려되지 않는다. 하지만 한 캐시 라인의 값을 공유할 수 있기 때문에 워크로드에 coalescing한 메모리 접근이 있다면 캐시 적중률에 긍정적인 영향을 줄 수 있다.

또한, backprop 같은 경우는 실제 L2 캐시와 값 차이가 5.65%로 오차가 낮은 것으로 볼 때, locality 정도가 캐시 적중에 높은 영향을 줌을 알 수 있다. 다른 워크로드의 경우 kmeans, convSep, leukocyte, hotspot은 대략 30% 정도의 오차가 있는데 이는 캐시 크기와 coalescing 그리고 thread의 개수가 영향을 끼치는 것으로 보인다. 그림 9는 워크로드의 L1 캐시와 intra-locality 관계를 표현한다. 그림 9를 보

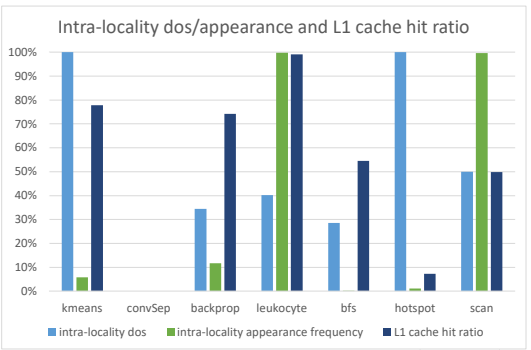


그림 9. intra-locality dos/appearance frequency와 L1 캐시 cache hit ratio
Fig 9. intra-locality dos/appearance frequency and L1 cache hit ratio

면 kmeans와 hotspot은 L2 캐시와 inter-locality

dos의 관계와 같이 intra-locality dos가 L1 캐시 hit ratio보다 높은 양상이 동일하게 보인다. 이는 캐시 크기에 비해 locality 값이 높은 워크로드의 특성이 반영됨을 알 수 있다. ConvSep의 경우 L1 캐시 hit ratio가 0%로 나오는데, 이는 convSep 워크로드가 하나의 SM 안에 있는 TB간 혹은 한 TB 내의 thread 간에는 intra-locality가 없다는 것을 알 수 있다. Srad 워크로드 같은 경우엔 inter-locality dos와 L2 캐시 값을 비교했을 땐, inter-locality dos 값이 0%로 나와 실제 L2 hit ratio인 54.01%와 차이를 보인다. 하지만 intra-locality dos 값과 L1 hit ratio와의 값을 비교했을 땐 intra-locality dos 50%와 L1 캐시 적중률 49.79%로 높은 유사도를 보였다.

또한, 캐시 hit ratio를 50%를 기준으로 높고 낮음을 판단할 때, 그림 8을 보면 L2 캐시의 경우 inter-locality appearance frequency가 100% 가까이 되는 kmeans, convSep, backprop, leukocyte 워크로드에 대해선 locality dos와 캐시 hit ratio가 같은 경향성을 보이며, 위 워크로드의 thread 개수는 총 12544, 2048, 16384, 11520으로 bfs와 hotspot에 비해 적은 thread 개수로 구성되어 있다. 하지만 워크로드의 thread 개수가 50000개가 넘어가는 bfs와 hotspot의 경우, dos와 hit ratio 간에 연관성이 낮다는 걸 알 수 있다. 따라서 워크로드의 데이터 locality 값과 캐시 hit ratio간의 더 정확한 연관성 분석을 위해선 머신의 캐시 크기에 대한 고려와 워크로드의 thread 개수, coalescing 정도 등 워크로드의 특성까지 고려하는 것이 필요하다.

V. 결 론

본 논문에서는 memory-divergent 워크로드를 프로파일링하여 정적분석을 통해 data locality를 분석하여 data locality의 객관적인 비교를 위해 지수를 정의하고, data locality 지수와 실제 실험에서의 캐시와의 관계를 분석한다. 그 결과 data locality와 캐시 이용에는 큰 상관관계가 존재하지만, 캐시의 제한적인 크기로 인해 접근하는 thread 수가 증가하면 그 상관관계가 약해짐을 확인하였다. 향후 연구로는 coalescing을 반영하는 locality 분석 지수의 개발이 필요하다. 또한, 시뮬레이션을 통해 실험 환경을 반영하는 방안을 연구하여 프로파일링을 보다 정확하게 할 수 있도록 한다.

References

[1] Li, A., Song, S.L., Liu, W., Liu, X., Kumar, A., Corporaal, H.: “Locality-aware CTA clustering for modern GPUs.” In: Proceedings of the international conference on architectural support for pro-gramming languages and operating systems, ASPLOS (2017)

[2] Li, C., Song, S.L., Dai, H., Sidelnik, A., Hari, S.K.S., Zhou, H.: “Locality-driven dynamic GPU cache bypassing” In: Proceedings of the 29th ACM on international conference on supercomputing, ICS (2015)

[3] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In IEEE Symposium on Performance Analysis of Systems and Software (ISPASS’09). IEEE, 163 – 174

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in Workload Characterization, 2009.

[5] https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

[6] https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[7] Sohan Lal, Bogaraju Sharatchandra Varma, Ben Juurlink, “A Quantitative Study of Locality in GPU Caches for Memory-Divergent Workloads”, International Journal of Parallel Programming, 50, pages189 – 216, April 2022

[8] Rhu, M., Sullivan, M., Leng, J., Erez, M.: “A locality-aware memory hierarchy for energy-efficient GPU architectures.”, In: Proceedings of the 46th annual IEEE/ACM international symposium on microarchitecture, MICRO (2013)

[9] Vijaykumar, N., Ebrahimi, E., Hsieh, K., Gibbons, P.B., Mutlu, O.: The locality descriptor: a holistic cross-layer abstraction to express data-locality in GPUs. In: Proceedings of the international sym- posium on computer architecture (ISCA) (2018)

[10] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," URL: http://www.cs.ucla.edu/pouchet/software/polybench, 2012.

[11] 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.

[12] D. Tripathy, Amirali Abdolrashidi, Quan Fan, Daniel Wong, and Manoranjan Satpathy. 2021 (To appear). LocalityGuru: A PTX Analyzer for Extracting Thread Block-level Locality in GPGPUs. Proceedings of the 15th IEEE/ACM International Conference on Networking, Architecture, and Storage (2021 (To appear)).

[13] Tang, X., Pattnaik, A., Kayiran, O., Jog, A., Kandemir, M.T., Das, C., "Quantifying data locality in dynamic parallelism in GPUs." Proc. ACM Meas. Anal. Comput, Syst (2018)

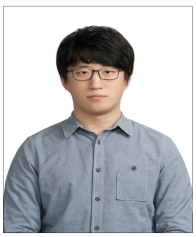
김 지 은 (Jieun Kim)

2020년 2월 : 숙명여자대학교 법학부, 컴퓨터과학전공 졸업(학사)

2021년 3월~현재 : 서울대학교 컴퓨터공학과 석사과정

<관심분야> GPU profiling, static profiling

신 현 일 (HyunIl Shin)



2017년 2월 : 서울대학교 컴퓨터공학과 학사 졸업

2017년 9월~현재 : 서울대학교 컴퓨터공학과 석박통합과정

<관심분야> GPU scheduling

엄 현 상 (Hyeonsan Eom)

1992년 : 서울대학교 계산통계학 졸업(학사)

1996년 : University of Maryland at College Park Computer Science 졸업(석사)

2003년: University of Maryland at College Park Computer Science 졸업(박사)

<관심분야> HPC 최적화, GPU scheduling, 분산시스템

김 윤 희 (Yoonhee Kim)

1991년 : 숙명여자대학교 전산학과 졸업(학사)

1996년 : Syracuse University 전산학과 졸업(석사)

2003년: Syracuse University 전산학과 졸업(박사)

1991년~ 1994년 한국전자통신연구원 연구원.

2000년~2001년 Rochester Institute of Technology 컴퓨터공학과 조교수.

2001년 ~ 2016년 숙명여자대학교 컴퓨터과학부 교수.

2017년 ~ 현재 숙명여자대학교 소프트웨어학부 교수.

<관심분야> 클라우드 컴퓨팅, 워크플로우 제어, HPC 클라우드, Accelerated Computing(GPUs)