

[CSE 4152] 고급 소프트웨어 실습 I

『CUDA 프로그래밍의 기초 2』

GPU 메모리의 계층 구조 및 특성

담당교수: 컴퓨터공학과 임인성 (AS-905, 02-705-8493, ihm@sogang.ac.kr)

담당조교: 김영욱 (AS-914, 02-711-5278, kimyu7@sogang.ac.kr)

최신 NVIDIA GPU의 성능

GPU	GeForce GTX 980 (Maxwell)	GeForce GTX 1080 (Pascal)
SMs	16	20
CUDA Cores	2048	2560
Base Clock	1126 MHz	1607 MHz
GPU Boost Clock	1216 MHz	1733 MHz
GFLOPs	4981 ¹	8873 ¹
Texture Units	128	160
Texel fill-rate	155.6 Gigatexels/sec	277.3 Gigatexels/sec
Memory Clock (Data Rate)	7,000 MHz	10,000 MHz
Memory Bandwidth	224 GB/sec	320 GB/sec
ROPs	64	64
L2 Cache Size	2048 KB	2048 KB
TDP	165 Watts	180 Watts
Transistors	5.2 billion	7.2 billion
Die Size	398 mm ²	314 mm ²
Manufacturing Process	28 nm	16 nm

GeForce GTX 1080 (Pascal)와 GeForce GTX 980 (Maxwell)의 비교
(출처: NVIDIA GeForce GTX 1080 Whitepaper (2016))

1 GPU 메모리 계층 구조에 대한 이해

1.1 Streaming Multiprocessor의 구조 예

다음은 NVIDIA GeForce GTX 1080 Whitepaper(2016)에서 발췌한 NVIDIA사의 최신 GPU의 구조인 Pascal architecture에 관한 설명이다.



그림 1: GP104 SM diagram (출처: NVIDIA GeForce GTX 1080 Whitepaper (2016))

Pascal GPUs are composed of different configurations of Graphics Processing Clusters (GPCs), Streaming Multiprocessors (SMs), and memory controllers. The GeForce GTX 1080 and its GP104 GPU consist of four GPCs, twenty Pascal Streaming Multiprocessors, and eight memory controllers. In the GeForce GTX 1080, each GPC ships with a dedicated raster engine and five SMs. Each SM contains 128 CUDA cores, 64 KB of register file capacity, a 96 KB shared memory unit, 48 KB of total L1 cache storage, and eight texture units. The SM is a highly parallel multiprocessor that schedules warps (groups of 32 threads) to CUDA cores and other execution units within the SM. The SM is one of the most important hardware units within the GPU; almost all operations flow through the SM at some point in the rendering pipeline. With 20 SMs, the GeForce GTX 1080 ships with a total of 2560 CUDA cores and

160 texture units. The GeForce GTX 1080 features eight 32-bit memory controllers (256-bit total). Tied to each 32-bit memory controller are eight ROP units and 256 KB of L2 cache. The full GP104 chip used in GTX 1080 ships with a total of 64 ROPs and 2048 KB of L2 cache.

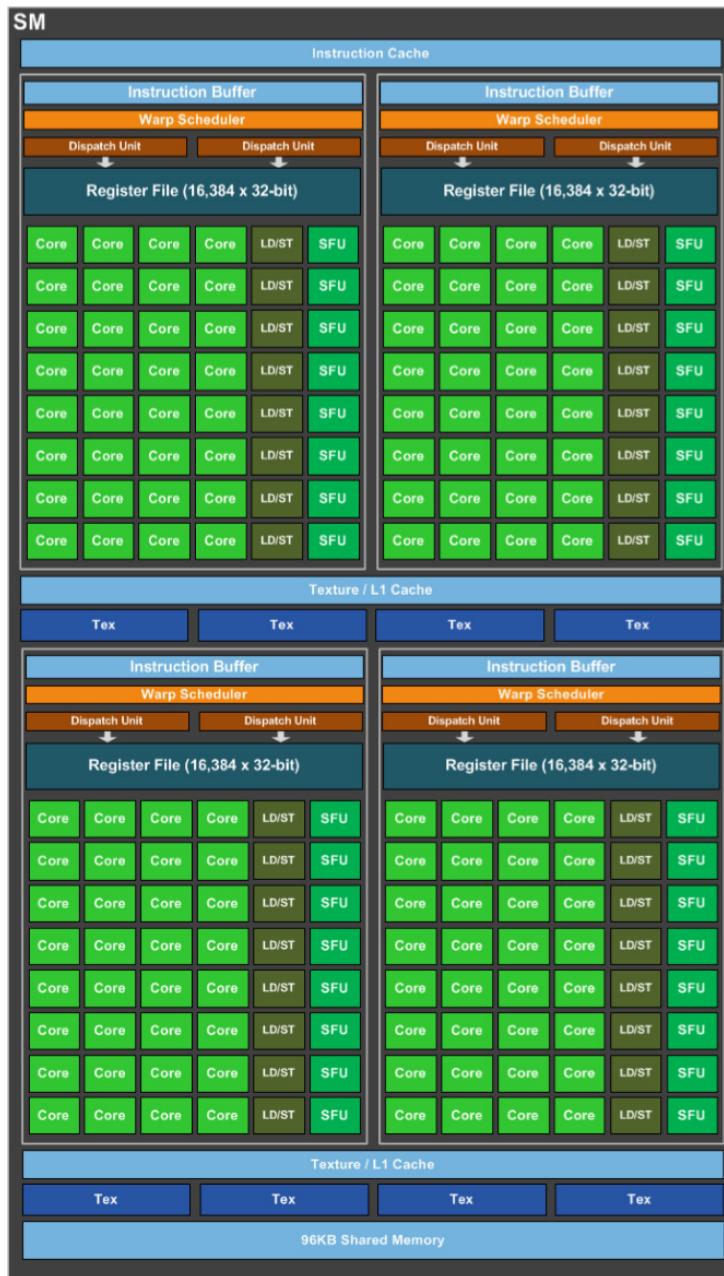


그림 2: GP104 SM diagram(출처: NVIDIA GeForce GTX 1080 Whitepaper (2016))

그림 1과 2를 보면서, register file, shared memory/L1 cache, L2 cache, 그리고 off-chip memory와 global memory에 대한 연결 통로인 memory controller에 대하여 자세히 살펴보자.

1.2 CUDA 프로그래밍 모델

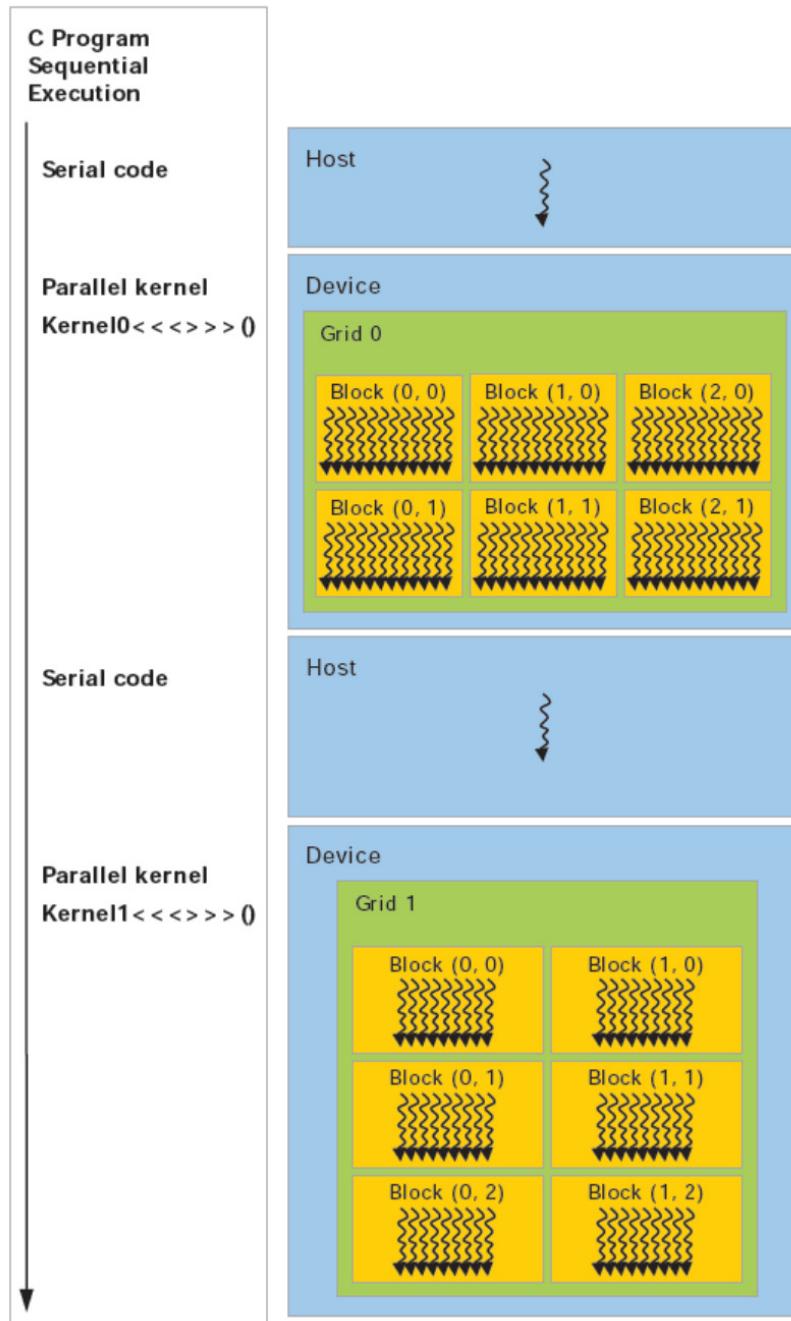


그림 3: CUDA 프로그래밍 모델(출처: CUDA C PROGRAMMING GUIDE(PG-02829-001_v8.0, September 2016))

일반적으로 CUDA 기반의 응용 프로그램은 순차적인 C 프로그램과 병렬적인 CUDA 커널 프로그램이 혼재하여 수행이 된다(그림 3 참조). C 코드는 CPU에서 수행이 되고 CUDA 커널 프로그램은 GPU에서 수행이 되는데, CUDA 프로그래밍 시스템에서는 각각의 프로세

서를 호스트(host)와 디바이스(device)라 부른다. 또한 전체 프로그램은 CPU, 즉 호스트에서 구동이 되므로 호스트 응용(host application)이라 부른다.

호스트와 디바이스는 자신만의 메모리 공간을 유지하는데, 각각을 호스트 메모리(host memory)와 디바이스 메모리(device memory)라 한다. 한편 두 메모리 공간 사이에 데이터를 주고 받는 작업이 필요한데, 이러한 작업은 다음 절에서 살펴볼 디바이스 메모리 공간에 할당되는 global memory, constant memory, 그리고 texture memory를 통하여 수행하게 된다. 즉 호스트 쪽에서 CUDA runtime 함수들을 사용하여 디바이스 메모리 공간에 상기 부류의 메모리들을 적절히 allocation/deallocation을 하면서 또한 필요한 데이터를 상호간에 전송도록 할 수 있다. 다음은 두 기기 간의 메모리 전송의 예를 보여주고 있다.

```

cudaMalloc(&d_pAOS, size);

cudaMemcpy(d_pAOS, p_AOS, size, cudaMemcpyHostToDevice);

dim3 dimBlock(BLOCK_WIDTH, BLOCK_HEIGHT);

dim3 dimGrid(ARRAY_2D_WIDTH /dimBlock.x, ARRAY_2D_HEIGHT/dimBlock.y);

TransformAOSKernel <<< dimGrid, dimBlock >>>(d_pAOS, m);

cudaMemcpy(p_AOS, d_pAOS, size, cudaMemcpyDeviceToHost);

cudaFree(d_pAOS);

```

1.3 CUDA 메모리 계층 구조

CUDA 커널 프로그램이 수행이 되는 각 쓰레드는 다음과 같이 계층적으로 구성이 되는 몇 개의 부류의 메모리 공간에 접근할 수 있다(그림 4 참조).

A. Private local memory: Register and local memory

- Scope: the thread to which the register space is allocated
- Lifetime: the thread
- Latency: fastest

B. Shared memory

- Scope: all threads of the block
- Lifetime: the block

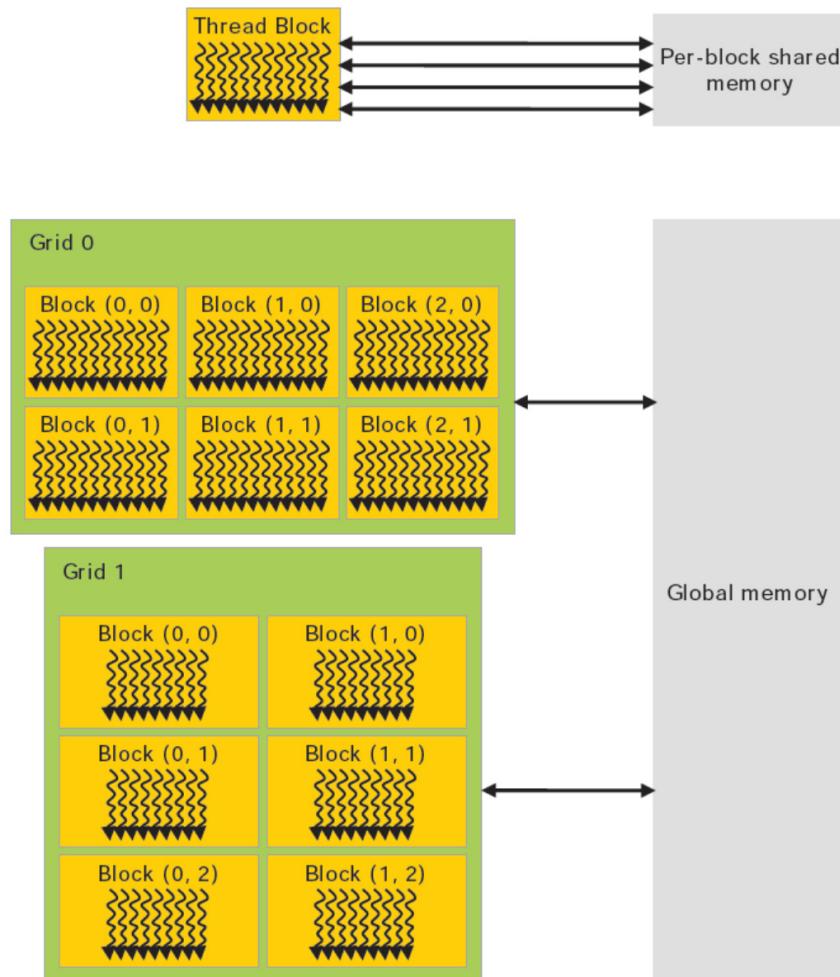


그림 4: CUDA 메모리 계층 구조(출처: CUDA C PROGRAMMING GUIDE(PG-02829-001_v8.0 — September 2016))

- Latency: a few dozens of cycles

C. Global memory

- Scope: all threads within the host application
- Lifetime: the host application
- Latency: a few hundred cycles (without caching)

D. Constant memory

- Scope: all threads within the host application
- Lifetime: the host application
- Latency: fast when all threads in a warp access the same location

E. Texture memory

- Scope: all threads within the host application
- Lifetime: the host application
- Latency: fast when all threads in a warp access the physically adjacent location

위 메모리들 중 register와 shared memory는 on-chip memory로서 상대적으로 상당히 빠른 접근 속도를 제공한다. 반면에 나머지 메모리들은 off-chip 메모리들로서 특히 global memory를 접근하는데 있어 수백 사이클 정도의 상당한 지연(latency)이 발생한다. 따라서 최근의 GPU들은 global memory를 좀 더 빠르게 접근하기 위하여 L1 캐시와 L2 캐시를 제공하고 있다(그림 5 참조). 이렇듯 CUDA 커널 프로그램 작성 시 사용할 수 있는 메모리의 종류와 그에 대한 성격이 각자 다른데, 효율적인 GPU 프로그램을 작성하기 위해서는 효과적인 메모리 사용이 필수적이라 할 수가 있다.

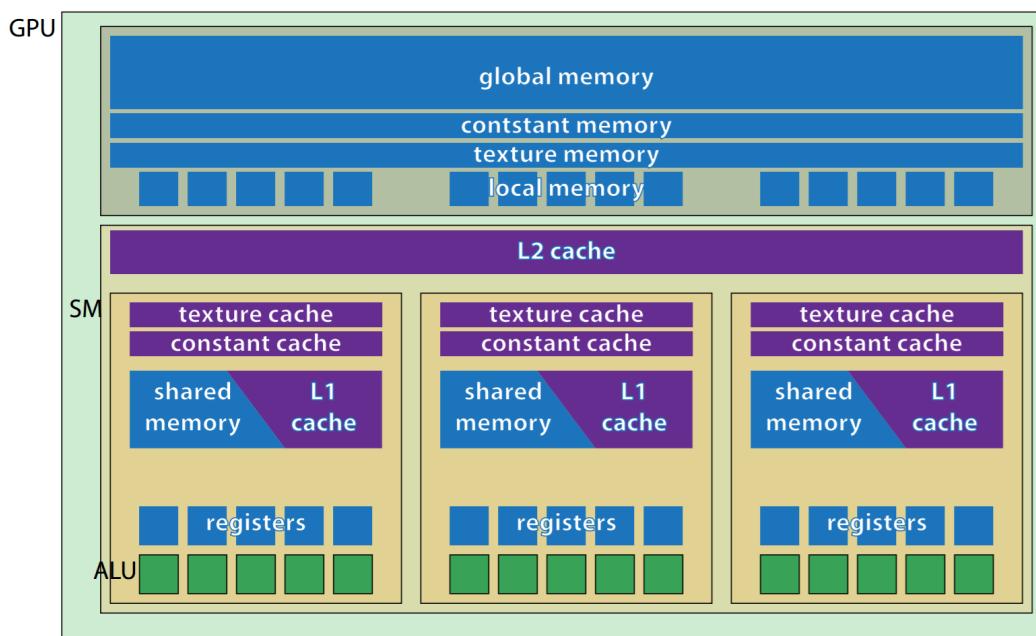


그림 5: CUDA 메모리 계층 구조 (출처: CUDA Memory Hierarchy (P. Danilewski, October 2012))

2 Global memory 접근 시의 memory coalescing에 대한 이해

2.1 CUDA 시스템 상에서의 memory coalescing

이전 장의 자료에서 설명한 바와 같이 CUDA 커널 코드는 한 CUDA 블럭 내의 쓰레드들을 32개의 쓰레드로 구성되는 와프(warp) 단위로 나누어 병렬적으로 수행이 된다. 따라서 (divergence가 발생하지 않았을 경우) 커널 프로그램에서 global memory에 있는 데이터를 접근하려 할 때, 예를 들어, $a[threadIdx.x] = b[threadIdx.x] + 3;$ 과 같은 문장을 수행 시키려할 경우, 한 와프 내의 32개의 쓰레드들로부터 먼저 (각각 $b[threadIdx.x]$ 에 대한) 총 32개의 메모리 접근 시도가 동시에 발생한다. 이때 CUDA 시스템에서는 메모리를 순차적으로 32번 접근하여 한 와프에서 필요로 하는 데이터를 가져오는 것이 아니라, 가급적 그 러한 메모리 접근 시도를 합쳐서(coalescing) 적은 회수의 메모리 접근(가능하다면 한 번의 메모리 트랜잭션을 통하여)으로 효율을 증대시키려 한다. 이러한 과정을 memory coalescing 또는 coalesced memory access라 한다.

그런데 global memory 접근 시 항상 memory coalescing이 잘 되는 것이 아니라 주어진 조건을 만족을 시켜야 한다. 현재의 GPU의 경우 32 바이트, 64 바이트, 그리고 128 바이트 크기의 메모리 트랜잭션을 제공하는데, 당연히 각 메모리 트랜잭션은 해당하는 바이트 크기에 정렬(alignment)되어 진행이 된다(예를 들어, 128 바이트 메모리 트랜잭션은 첫 시작 주소가 128의 배수인 지점부터 128바이트를 접근할 수 있음). 한 와프가 global memory 접근 명령을 수행할 때, CUDA 시스템은 각 쓰레드로부터의 메모리 접근들을 가급적 적은 회수의 메모리 트랜잭션으로 합쳐 결과적으로 메모리 접근의 효율을 높이려 한다.

예를 들어, 그림 6은 한 와프의 32개 쓰레드들이 각각 4바이트 크기의 변수를 접근하려 할 때 32개의 변수 영역이 128바이트의 정렬된 범위 내에 존재하는 경우인데, 이 경우 32 개의 메모리 접근 시도가 한 개의 128 바이트 트랜잭션으로 합쳐서 처리가 된다(캐싱이 되는 경우). 반면에 다음 그림 7은 한 와프의 쓰레드들이 연속한 128 바이트 영역을 접근하는 것은 동일하나, 시작 지점이 128 바이트 정렬 지점에서 네 바이트 떨어진 지점에 존재하기 때문에 두 번의 128 바이트 트랜잭션으로 나누어져(캐싱이 되는 경우) 메모리 접근이 일어난다. 따라서 결과적으로 메모리 접근의 효율이 반으로 떨어지게 된다.

그래도 이 두 경우는 양호한 경우인데, 만약 한 와프의 32개 쓰레드 각각이 접근하는 메모리 영역이 지역적으로 멀리 떨어져 있을 때, 최악의 경우 32번의 메모리 트랜잭션이 일어나고 따라서 메모리 접근 효율이 매우 떨어지게 된다. CUDA 프로그램 작성 시 가장 신경을 써야 할 부분 중의 하나는 off-chip 메모리인 global memory 접근의 효율을 가급적 높이도록 코딩을 하는 것인데, 실제로 global memory 접근에 대한 최적화 노력은 종종 커널 프로그램 수행 시간에 상당한 영향을 미치게 된다. 따라서 메모리 접근 시 아래와 같은 특성에 대하여 잘 생각해보기 바란다.

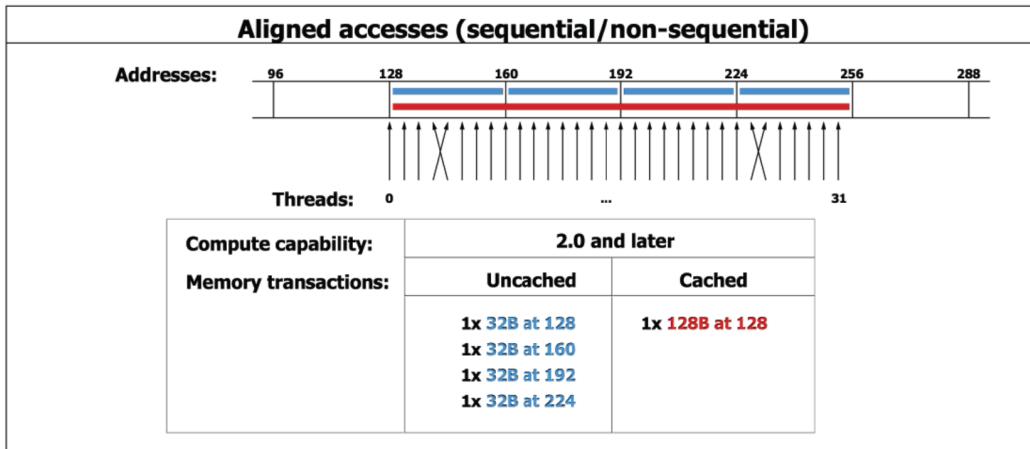


그림 6: Global memory coalescing의 예 1(출처: CUDA C PROGRAMMING GUIDE (PG-02829-001_v8.0 — September 2016))

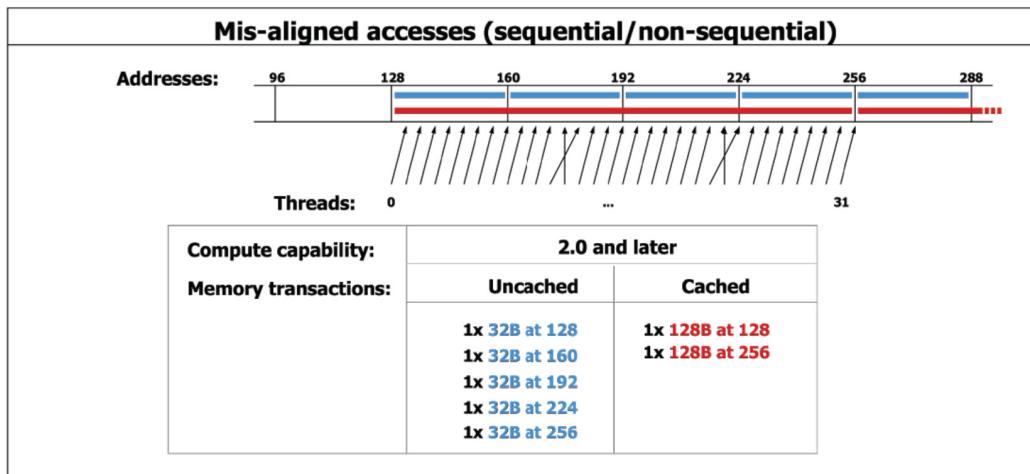


그림 7: Global memory coalescing의 예 2(출처: CUDA C PROGRAMMING GUIDE (PG-02829-001_v8.0 — September 2016))

- Sequential versus non-sequential memory access
- Coherent versus sparse memory access
- Aligned versus misaligned memory access

2.2 예제: 4차원 벡터를 원소로 가지는 배열의 처리 문제

이 절에서는 CUDA 시스템 상에서 global memory 접근 시 memory coalescing 과정이 성능에 어떠한 영향을 미치는지에 대하여 예제 문제를 통하여 살펴보자.

C/C++ 기반의 CPU 코드

다음과 같은 C 프로그램을 살펴보자.

```
#define N_POINTS 65536

typedef struct {
    float x; float y; float z; float w;
} POINT_ELEMENT;

typedef struct {
    float x[N_POINTS]; float y[N_POINTS];
    float z[N_POINTS]; float w[N_POINTS];
} POINTS_SOA;
    :

void transform_points_AOS(POINT_ELEMENT *p_AOS, int n_points, int m)
{
    int i, j;
    float tmp;

    for (i = 0; i < n_points; i++) {
        for (j = 2; j <= m; j++) {
            tmp = 1.0f / (float) j;

            p_AOS[i].x += tmp*p_AOS[i].x; p_AOS[i].y += tmp*p_AOS[i].y;
            p_AOS[i].z += tmp*p_AOS[i].z; p_AOS[i].w += tmp*p_AOS[i].w;
        }
    }
}

void transform_points_SOA(POINTS_SOA *p_SOA, int n_points, int m) {
    int i, j;
    float tmp;

    for (i = 0; i < n_points; i++) {
```

```

    for (j = 2; j <= m; j++) {
        tmp = 1.0f / (float) j;

        p_SOA->x[i] += tmp*p_SOA->x[i]; p_SOA->y[i] += tmp*p_SOA->y[i];
        p_SOA->z[i] += tmp*p_SOA->z[i]; p_SOA->w[i] += tmp*p_SOA->w[i];
    }
}

void main(void) {
    float compute_time;
    int n_points, cutoff;
    POINT_ELEMENT Points_AOS[N_POINTS];
    POINTS_SOA Points_SOA;

    n_points = N_POINTS;
    cutoff = 10000;

    generate_point_data(Points_AOS, &Points_SOA, n_points);
    transform_points_AOS(Points_AOS, n_points, cutoff);
    transform_points_SOA(&Points_SOA, n_points, cutoff);
    :
}

```

여기서 두 함수 `transform_points_SOA()`와 `transform_points_SOA()`는 각각 `x`, `y`, `z`, 그리고 `w` 등 네 개의 필드로 구성된 4차원 벡터 배열을 받아 들여 각 벡터에 대하여 어떤 연산을 수행하고 있다. 코드를 살펴보면 쉽게 알 수 있듯이, 이 두 함수가 계산해주는 연산은 동일한데, 다만 4차원 벡터들을 메모리에 저장하는 방식에 차이가 있을 뿐이다. `main()` 함수에서 선언한 두 변수 `Points_AOS`와 `Points_SOA`에 대한 타입 `POINT_ELEMENT *`과 `POINTS_SOA`를 잘 살펴보면, 왜 전자를 `array of structure(AOS)` 방식, 그리고 후자를 `structure of array(SOA)` 방식을 사용하여 데이터를 표현하였다고 하는지를 이해할 수가 있다.

사용자 입장에서 각 데이터 표현 방식에 기반을 둔 위의 두 함수는 동일한 연산을 수행해준다. 하지만 이 두 함수가 컴퓨터 내부에서 수행이 될 때 프로세서가 메모리를 접근하는 방식에는 상당한 차이가 있다. 특히 프로세서의 캐쉬의 활용도 입장에서는 상당한 차이를

발생시킬 수가 있다. 실제로 3.4 GHz Intel Core i7 CPU 상에서 이 두 함수의 수행 시간을 측정해본 결과 적지 않은 시간 차이를 확인할 수 있었는데(그림 8과 9 참조), 그러한 원인에 대해서는 코드 최적화 주제와 관련한 실습에서 살펴보았다.

```
C:\> C:\WINDOWS\system32\cmd.exe
////////// First round //////////
*** Array of structure: Time taken = 2884.042ms
*** Structure of array: Time taken = 7944.014ms
--- AOS.10.x = 3.384695e+005, SOA.10.x = 3.384695e+005
--- AOS.20.y = 4.422133e+005, SOA.20.y = 4.422133e+005

////////// Second round //////////
*** Array of structure: Time taken = 2910.719ms
*** Structure of array: Time taken = 7943.208ms
--- AOS.10.x = 2.118933e+005, SOA.10.x = 2.118933e+005
--- AOS.20.y = 3.873257e+005, SOA.20.y = 3.873257e+005

////////// Third round //////////
*** Array of structure: Time taken = 2911.286ms
*** Structure of array: Time taken = 8029.067ms
--- AOS.10.x = 7.266562e+004, SOA.10.x = 7.266562e+004
--- AOS.20.y = 4.770042e+005, SOA.20.y = 4.770042e+005

////////// Fourth round //////////
*** Array of structure: Time taken = 2937.673ms
*** Structure of array: Time taken = 7999.249ms
--- AOS.10.x = 4.334865e+005, SOA.10.x = 4.334865e+005
--- AOS.20.y = 6.668751e+004, SOA.20.y = 6.668751e+004
계속하려면 아무 키나 누르십시오 . . .
```

그림 8: CPU 상에서의 AOS 방식과 SOA 방식의 비교 1(a) ($N_POINTS = 65,536$)

또한 위의 프로그램에서와는 달리 SOA 방식에서 벡터 데이터를 다음과 같이 정의하고,

```
typedef struct {
    float *x; float *y; float *z; float *w;
} POINTS_SOA;
```

동적으로 메모리 공간을 할당한 경우의 시간 측정 결과가 그림 10에 도시되어 있는데, 앞에서처럼 두 방법의 수행 시간에 큰 차이가 나지 않음을 관찰할 수 있다. 이는 두 경우 SOA 배열 데이터에 대한 메모리가 할당이 되는 영역이 다르고 따라서 그에 따라 캐싱의 효과가 다르게 나타나는 것으로 추정되는데, 여하한 경우 AOS 방법이 SOA 방법보다 최소한 더 빠

```
C:\WINDOWS\system32\cmd.exe

////////// First round //////////
*** Array of structure: Time taken = 11507.114ms
*** Structure of array: Time taken = 13778.265ms
--- AOS.10.x = 7.943489e+004, SOA.10.x = 7.943489e+004
--- AOS.20.y = 3.009642e+004, SOA.20.y = 3.009642e+004

////////// Second round //////////
*** Array of structure: Time taken = 11669.289ms
*** Structure of array: Time taken = 13795.101ms
--- AOS.10.x = 2.631198e+005, SOA.10.x = 2.631198e+005
--- AOS.20.y = 1.430537e+005, SOA.20.y = 1.430537e+005

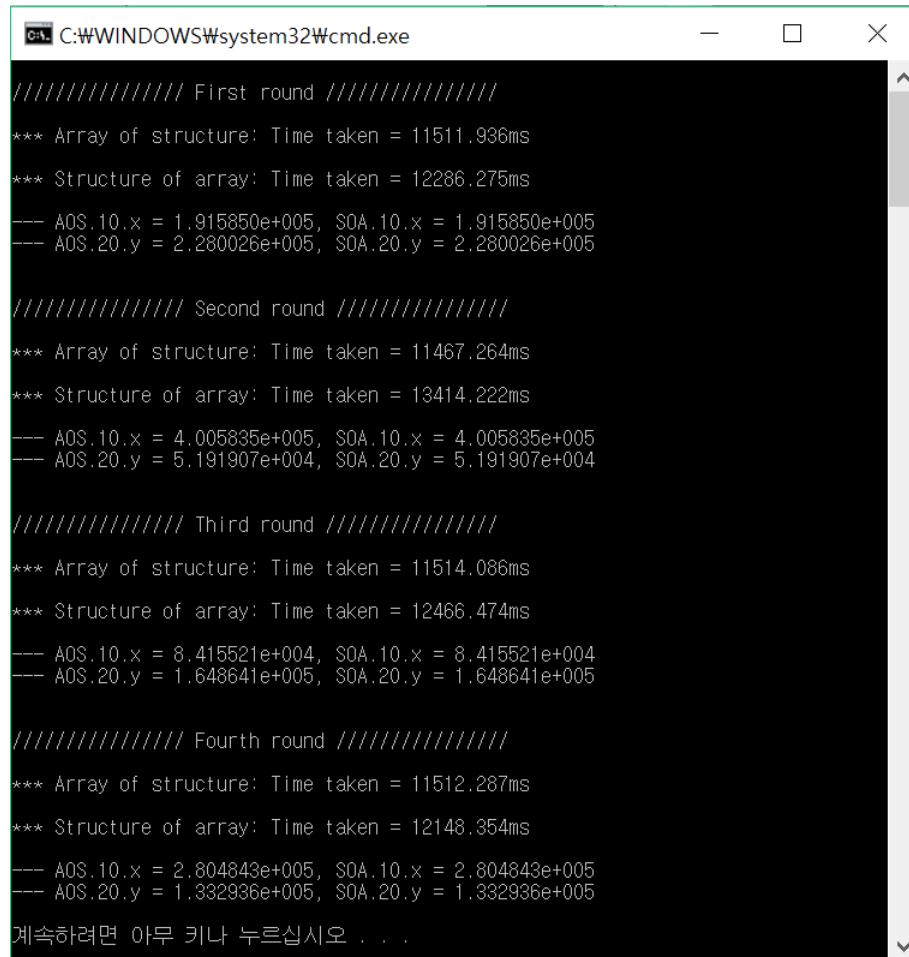
////////// Third round //////////
*** Array of structure: Time taken = 11473.809ms
*** Structure of array: Time taken = 13912.089ms
--- AOS.10.x = 4.466946e+005, SOA.10.x = 4.466946e+005
--- AOS.20.y = 2.559951e+005, SOA.20.y = 2.559951e+005

////////// Fourth round //////////
*** Array of structure: Time taken = 11546.033ms
*** Structure of array: Time taken = 14478.086ms
--- AOS.10.x = 1.176846e+005, SOA.10.x = 1.176846e+005
--- AOS.20.y = 1.344581e+004, SOA.20.y = 1.344581e+004

계속하려면 아무 키나 누르십시오 . . .
```

그림 9: CPU 상에서의 AOS 방식과 SOA 방법의 비교 1(b)(N_POINTS = 262,144)

음을 알 수가 있다(이 문제의 경우 데이터 접근 방식이 코드 최적화 관련 실습에서 다룬 2 차원 배열 접근의 예만큼 메모리 접근 속도에 크게 영향을 미치지 않음을 이해해보자).



```
/////////// First round ///////////
*** Array of structure: Time taken = 11511.936ms
*** Structure of array: Time taken = 12286.275ms
--- AOS.10.x = 1.915850e+005, SOA.10.x = 1.915850e+005
--- AOS.20.y = 2.280026e+005, SOA.20.y = 2.280026e+005

/////////// Second round ///////////
*** Array of structure: Time taken = 11467.264ms
*** Structure of array: Time taken = 13414.222ms
--- AOS.10.x = 4.005835e+005, SOA.10.x = 4.005835e+005
--- AOS.20.y = 5.191907e+004, SOA.20.y = 5.191907e+004

/////////// Third round ///////////
*** Array of structure: Time taken = 11514.086ms
*** Structure of array: Time taken = 12466.474ms
--- AOS.10.x = 8.415521e+004, SOA.10.x = 8.415521e+004
--- AOS.20.y = 1.648641e+005, SOA.20.y = 1.648641e+005

/////////// Fourth round ///////////
*** Array of structure: Time taken = 11512.287ms
*** Structure of array: Time taken = 12148.354ms
--- AOS.10.x = 2.804843e+005, SOA.10.x = 2.804843e+005
--- AOS.20.y = 1.332936e+005, SOA.20.y = 1.332936e+005

계속하려면 아무 키나 누르십시오 . . .
```

그림 10: CPU 상에서의 AOS 방식과 SOA 방법의 비교 ($N_POINTS = 262,144$)

CUDA 기반의 GPU 코드

이제 앞 절의 프로그램을 다시 살펴보면 이 두 함수가 수행하는 작업에는 GPU의 병렬처리 구조에 적합한 병렬성이 존재함을 할 수 있다. 각 벡터(POINT)마다 CUDA 쓰레드를 할당하여 병렬적으로 가속을 해주는 CUDA 기반의 GPU 코드의 예는 다음과 같다.

```
#define ELEM_PER_POINT (1 << 5)
#define N_ELEMS (1 << 22)
#define N_POINTS (N_ELEMS / ELEM_PER_POINT)

#define ARRAY_2D_WIDTH 1024
#define ARRAY_2D_HEIGHT (N_POINTS/ARRAY_2D_WIDTH)
#define BLOCK_WIDTH 128
#define BLOCK_HEIGHT 8

typedef struct {
    float elem[ELEM_PER_POINT];
} POINT_ELEMENT; // for AOS

typedef struct { // Array spaces are allocated dynamically.
    float *elem[ELEM_PER_POINT];
} POINTS_SOA;

__global__ void TransformAOSKernel(POINT_ELEMENT *A, int m) {
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    int id = gridDim.x*blockDim.x*row + col;

    for (int j = 2; j <= m; j++) {
        float tmp = 1.0f / (float) j;
        for (int i = 0; i < ELEM_PER_POINT; i++) {
            A[id].elem[i] += tmp*A[id].elem[i];
        }
    }
}
```

```
--global__ void TransformSOAKernel(POINTS_SOA A, int m) {  
    int row = blockIdx.y*blockDim.y + threadIdx.y;  
    int col = blockIdx.x*blockDim.x + threadIdx.x;  
    int id = gridDim.x*blockDim.x*row + col;  
  
    for (int i = 0; i < ELEM_PER_POINT; i++) {  
        for (int j = 2; j <= m; j++) {  
            float tmp = 1.0f / (float) j;  
            A.elem[i][id] += tmp*A.elem[i][id];  
        }  
    }  
}  
  
void transform_points_AOS(POINT_ELEMENT *p_AOS, int n_points, int m)  
{  
    POINT_ELEMENT *d_pAOS;  
    size_t size = N_POINTS * sizeof(POINT_ELEMENT);  
  
    cudaMalloc(&d_pAOS, size);  
    cudaMemcpy(d_pAOS, p_AOS, size, cudaMemcpyHostToDevice);  
  
    dim3 dimBlock(BLOCK_WIDTH, BLOCK_HEIGHT);  
    dim3 dimGrid(ARRAY_2D_WIDTH /dimBlock.x, ARRAY_2D_HEIGHT/dimBlock.y);  
    TransformAOSKernel <<< dimGrid, dimBlock >>>(d_pAOS, m);  
  
    cudaMemcpy(p_AOS, d_pAOS, size, cudaMemcpyDeviceToHost);  
    cudaFree(d_pAOS);  
}  
  
void transform_points_SOA(POINTS_SOA p_SOA, int n_points, int m) {  
    POINTS_SOA d_pSOA;  
    size_t size = N_POINTS * sizeof(float);
```

```

    cudaMalloc(&d_pSOA.x, size);
    cudaMemcpy(d_pSOA.x, p_SOAs.x, size, cudaMemcpyHostToDevice);
    :
    cudaMalloc(&d_pSOA.w, size);
    cudaMemcpy(d_pSOA.w, p_SOAs.w, size, cudaMemcpyHostToDevice);

    dim3 dimBlock(BLOCK_WIDTH, BLOCK_HEIGHT);
    dim3 dimGrid(ARRAY_2D_WIDTH/dimBlock.x, ARRAY_2D_HEIGHT/dimBlock.y);
    TransformSOAKernel <<< dimGrid, dimBlock >>> (d_pSOA, m);

    cudaMemcpy(p_SOAs.x, d_pSOA.x, size, cudaMemcpyDeviceToHost);
    :
    cudaMemcpy(p_SOAs.w, d_pSOA.w, size, cudaMemcpyDeviceToHost);

    cudaFree(p_SOAs.x);
    :
    cudaFree(p_SOAs.w);
}

```

CPU의 경우에서처럼 두 데이터 표현 방식에 따른 해당 함수의 수행 시간을 NVIDIA GeForce GTX 690 GPU에서 측정한 결과가 표 1에 주어져 있다. 이 실험에서는 벡터 당 4개가 아니라, 좀 더 보편적인 상황인 ELEM_PER_POINT 개의 필드를 가지는 경우를 고려하였다. 다시 한 번 강조하면 CUDA 쓰레드들은 인접한 32개씩 와프로 묶여 와프 단위로 병렬적으로 수행이 된다. 이때 위 두 방식의 함수가 실제로 수행 될 때, 한 와프의 32개의 쓰레드들이 global memory의 어떤 지점을 동시에 접근하는지 생각해보면, (i) ELEM_PER_POINT가 커질수록 AOS 방식을 사용할 때 SOA 방식을 사용할 경우보다 메모리 접근 국소성 (locality)이 매우 떨어지게 되고(따라서 비효율적인 global memory 접근으로 인하여 성능이 나빠지고), 또한 (ii) 그럼에도 불구하고 ELEM_PER_POINT가 비교적 작을 때는 캐싱 효과로 인하여 그러한 문제가 보완이 되어 (코드 상 원래 더 효율적인) AOS 방식이 성능이 더 좋음을 확인할 수 있다.

한 가지 언급할 점은, 위의 두 번째 `TransformSOAKernel(*)` 커널 함수 내에서 `float tmp = 1.0f / (float) j;` 문장을 반복적으로 수행하는 것은 매우 비효율적이다라는 점이다. 수행 속도를 향상시키려면 해당하는 모든 자연수에 대해 CPU에서 이 값을 한 번만 계산하여 디바이스 메모리의 constant memory 영역에 배열 형태로 올린 후,

	ELEM_PER_POINT				
	8	16	32	64	128
AOS	11.17	8.057	9.037	3,846	4,232
SOA	91.52	72.50	66.27	63.07	72.75

표 1: AOS 구조와 SOA 구조의 차이에 의한 CUDA 커널 프로그램의 성능 차이(단위: ms).

```

__constant__ float constantBuffer[1000];
:

void generate_constant_data(IN int m) {
    float *p_constant;

    // Allocate memory for p_constant of size sizeof(float)*m.
    // Compute p_constant[i-1] = 1.0f/(float) i for i = 2, 3, ..., m.
    cudaMemcpyToSymbol(constantBuffer, p_constant, sizeof(float)*m);
    free(p_constant);
}

```

해당 커널 프로그램에서 그 값을 사용하면 수행 속도가 눈에 띄게 향상 됨을 알 수 있다(constant memory의 경우 한 와프 내의 32개의 쓰레드들이 동일한 constant memory 영역을 접근할 때 매우 효율적임을 명심할 것).

```

__global__ void TransformSOAwithConstantMemKernel(INOUT POINTS_SOA A,
IN int m) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int id = gridDim.x*blockDim.x*row + col;
    int i, j;

    for ( i = 0; i < ELEM_PER_POINT; ++i ) {
        for ( j = 2; j <= m; j++ ) {
            A.elem[ i ][ id ] += constantBuffer[j-1]*A.elem[ i ][ id ];
        }
    }
}

```

}

또한 커널 프로그램 작성 시 global memory에 비해 접근 속도가 매우 빠른(그리고 한 블럭 내의 쓰레드들이 공유할 수 있는) shared memory를 효과적으로 사용할 경우, 첫 번째 커널 프로그램 TransformAOSKernel(*)의 속도로 상당히 향상시킬 수 있는데, 이에 대해서는 실습 시간에 알아보기로 하자(그림 11 참조).

The screenshot shows a command-line interface (cmd.exe) window on a Windows system. The window title is 'C:\WINDOWS\system32\cmd.exe'. The text output is as follows:

```
ELEMENT PER POINT = 64

///////////////////First round///////////////////
*** Array of structure: GPU Time taken = 3839.806ms
*** Structure of array: GPU Time taken = 62.832ms
--- AOS.10.x = 4.682053e+004, SOA.10.x = 4.682053e+004
--- AOS.20.y = 4.853669e+004, SOA.20.y = 4.853669e+004

///////////////////Second round/////////////////
*** Array of structure: GPU Time taken = 3819.481ms
*** Structure of array: GPU Time taken = 62.837ms
--- AOS.10.x = 2.343355e+007, SOA.10.x = 2.343355e+007
--- AOS.20.y = 2.429241e+007, SOA.20.y = 2.429241e+007

///////////////////Constant first round/////////////////
*** Array of structure: GPU Time taken = 3850.137ms
*** Structure of array: GPU Time taken = 17.726ms
--- AOS.10.x = 1.172840e+010, SOA.10.x = 1.172840e+010
--- AOS.20.y = 1.215847e+010, SOA.20.y = 1.215847e+010

///////////////////Constant second round/////////////////
*** Array of structure: GPU Time taken = 3848.445ms
*** Structure of array: GPU Time taken = 17.734ms
--- AOS.10.x = 5.870035e+012, SOA.10.x = 5.870035e+012
--- AOS.20.y = 6.085245e+012, SOA.20.y = 6.085245e+012

///////////////////Constant/Shared first round/////////////////
*** Array of structure: GPU Time taken = 919.464ms
*** Structure of array: GPU Time taken = 16.219ms
--- AOS.10.x = 2.937984e+015, SOA.10.x = 2.937984e+015
--- AOS.20.y = 3.045678e+015, SOA.20.y = 3.045678e+015

///////////////////Constant/Shared second round/////////////////
*** Array of structure: GPU Time taken = 919.466ms
*** Structure of array: GPU Time taken = 16.216ms
--- AOS.10.x = 1.470460e+018, SOA.10.x = 1.470460e+018
--- AOS.20.y = 1.524350e+018, SOA.20.y = 1.524350e+018

계속하려면 아무 키나 누르십시오 . . . ■
```

그림 11: Constant memory와 shared memory의 사용을 통한 성능 향상 결과

3 Shared Memory 접근 시의 Bank Conflict의 이해

앞에서 설명한 바와 같이 shared memory는 on-chip memory이므로 global memory나 local memory보다 훨씬 높은 대역폭과 낮은 지연을 제공한다. 특히 shared memory는 한 CUDA 블럭 내의 모든 쓰레드들이 공유할 수 있는 영역으로서, 데이터의 공유 외에도 서로간의 통신을 할 수 있는 기능을 제공하기 때문에 매우 유용한 부류의 메모리라 할 수 있다. 하지만 역시 32개의 쓰레드로 구성되는 와프에서 shared memory를 어떠한 방식으로 접근하는 가에 따라 메모리 접근 속도가 달라지기 때문에 주의를 해야 한다. 이러한 상황을 파악하기 위해 저는 shared memory 접근 시의 bank conflict에 대해 이해를 해보자.

Shared memory는 효율을 위하여 뱅크(bank)라고 부르는 동일한 크기의 모듈로 구성이 되어 있는데, 만약 여러 개의 쓰레드가 각각 서로 다른 뱅크의 영역을 접근한다면 이러한 접근은 동시에 이루어질 수가 있다. 반면에 어떤 두 쓰레드가 동일한 뱅크의 영역을 접근하여 하면 뱅크 충돌(bank conflict)이 발생하게 된다. 이 경우 (특별한 조건을 만족시키지 않을 경우) 두 개의 순차적인 메모리 접근이 발생하고 따라서 속도가 느려지게 된다. 한 와프의 32개 쓰레드로부터 32개의 shared memory 접근이 발생할 때, CUDA 시스템은 뱅크 충돌이 없는 메모리 접근들을 묶어 가급적 적은 회수의 순차적인 메모리 접근을 통하여 효율을 높이려 한다. 물론 프로그래머 입장에서는 항상 뱅크 충돌이 없도록 코딩을 할 수는 없으나 가급적 그러한 상황을 피하도록 커널 프로그램을 작성하는 것이 필요하다고 할 수 있다. 그림 12 과 13을 보면서 뱅크 충돌에 대하여 이해하여 보자(실습 및 숙제를 통하여 shared memory 의 bank conflict 감소를 통한 커널 프로그램의 속도 향상에 대하여 연습해볼 예정임).

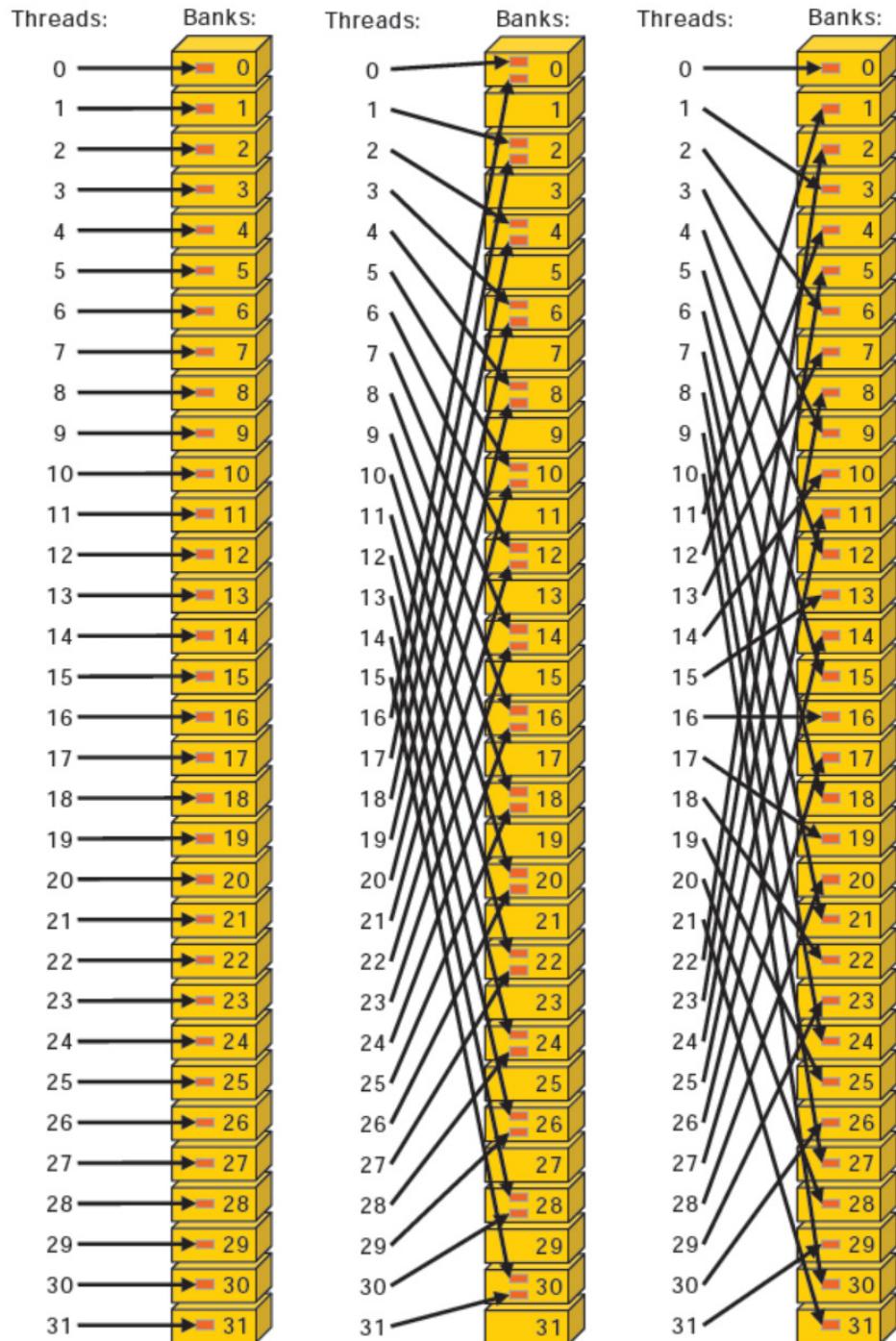


그림 12: Shared memory bank conflict의 예 1: 왼쪽/오른쪽은 뱅크 충돌 없음. 가운데: 두 개의 뱅크 충돌(출처: CUDA C PROGRAMMING GUIDE (PG-02829-001_v8.0 — September 2016))

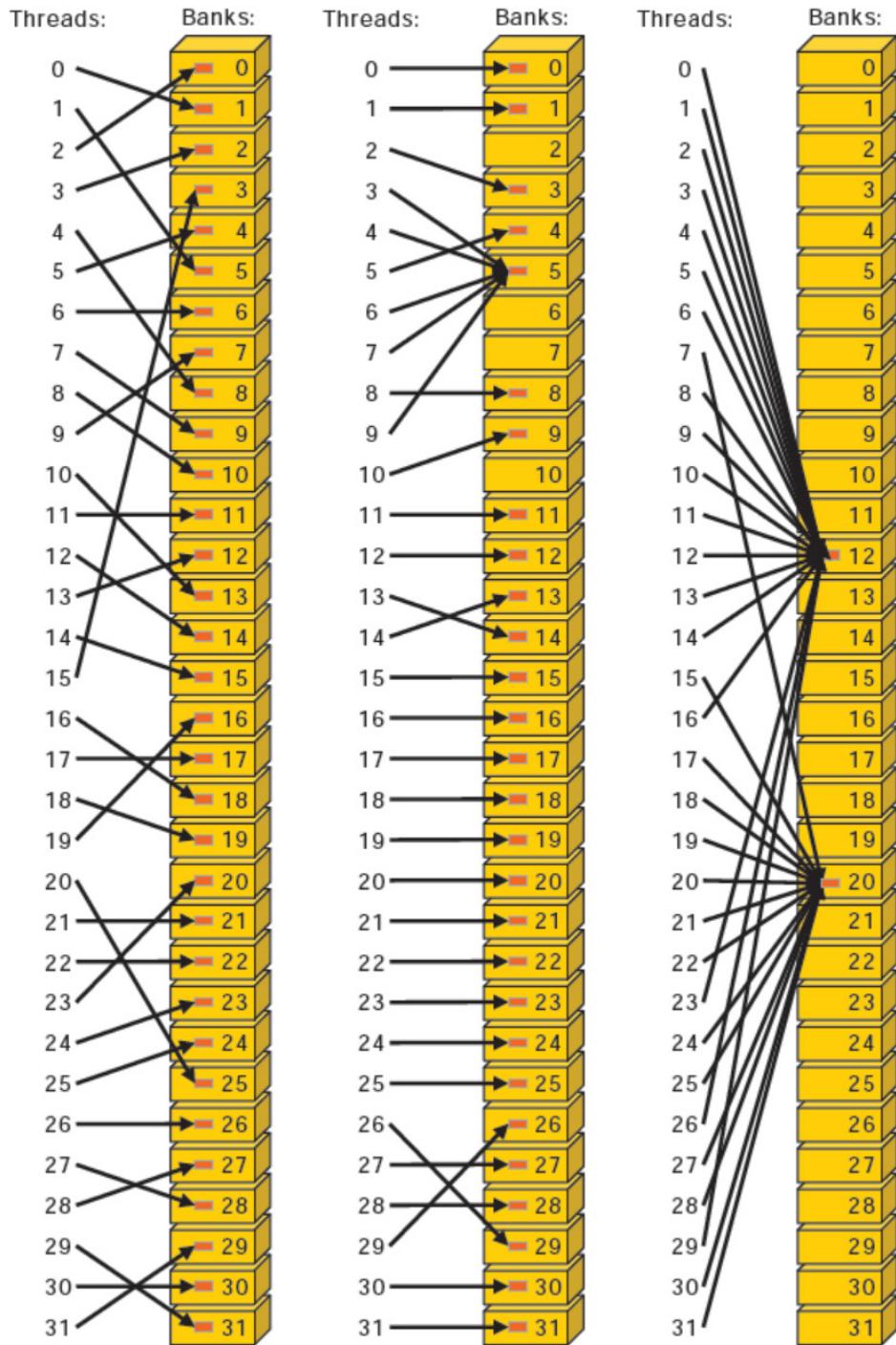


그림 13: Shared memory bank conflict의 예 2: 뱅크 충돌 없음(출처: CUDA C PROGRAMMING GUIDE (PG-02829-001_v8.0 — September 2016))

4 실습 문제

실습 문제 1

이번 실습 문제에서 해결해야 할 문제는 다음과 같다. (이번 실습을 위하여 지난 주 숙제 3번에 대하여 자신이 작성한 코드를 가지고 와도 무방함.)

-100과 100 사이의 값을 가지는 n 개의 정수로 구성된 수열 $X = \{x_i\}$ ($i = 0, 1, \dots, n - 1$)을 고려 하자. 이제 주어진 양의 정수 n_f 에 대하여 이 수열의 각 원소 a_i 를 중심으로 왼쪽과 오른쪽 각 방향으로 n_f 개의 원소들과 자신의 합 s_i 를 구하려 한다. 즉 또 다른 수열 $S = \{s_i\}$ ($i = 0, 1, \dots, n - 1$)의 값을 다음과 같이 $(2n_f + 1)$ 개의 원소의 값으로 표현할 수 있는데,

$$S_i = \sum_{k=i-n_f}^{i+n_f} x_k,$$

만약 왼쪽이나 오른쪽에 충분한 개수의 원소가 없을 경우 합이 가능한 원소들까지만 더한다. (예를 들어, $n_f = 5$ 이고 $i = 2$ 라면 $s_2 = \sum_{k=-3}^7 x_i$ 가 되는데, i 가 0보다 작거나 n 보다 같거나 클 경우 x_i 는 0값을 가진다고 생각하면 된다.)

이제 주어진 입력 데이터 파일에서 수열 $X = \{x_i\}$ 를 읽어들여 수열 $S = \{s_i\}$ 를 계산한 후 출력 파일에 저장해주는 문제를 CPU와 GPU 기반의 코드를 각각 작성하여 그 성능을 비교하여 보자.

- (i) 먼저 for 문장을 사용하여 순차적으로 n 번의 동일한 계산을 반복적으로 수행하는 C/C++ 함수(**CPU-Code**)를 작성한 후 CPU 수행 시간을 측정하라. 시간 측정 방법은 이전 실습 시간에 사용한 방법을 사용하고, 가급적 정확한 시간 측정을 위하여 여러 번 반복적으로 함수를 수행시킨 후 평균값을 취하라. (이때 데이터 입출력 시간은 제외하고 순수한 수열 계산 시간만 측정할 것.)
- (ii) 다음 이 문제를 해결해주는 CUDA 커널 프로그램(**GPU-Code-NO-SM**)을 작성한 후 가급적 정확하게 GPU 수행 시간을 측정하라. 이 프로그램은 shared memory 비사용 버전임.
- (iii) 다음 위의 코드를 CUDA의 shared memory 기능을 사용하여 성능을 향상 시켜주는 프로그램(**GPU-Code-SM**)을 작성하라.

- (iv) 두 프로그램 **GPU-Code-NO-SM**와 **GPU-Code-SM**의 성능 차이를 분석하기 위하여, 주어진 쓰레드 블럭 크기에 대하여 중복성을 의미하는 n_f 값을 변화시켜가면서, 두 버전의 시간을 측정한 후 왜 그러한 결과/차이가 산출되었는지 분석하라.

주의 1 이번 실습에서 입력 파일의 이름은 `Cuda_LAB1_input.bin`이고 출력 파일의 이름은 `Cuda_LAB1_output.bin`으로 한다.

주의 2 입출력 파일의 데이터는 모두 binary 형식으로 저장되며, 각 파일의 첫 4 바이트에는 원소의 개수 n 이, 그리고 다음 4 바이트에는 덧셈의 범위를 지정하는 n_f 인자가 각각 `int` 타입으로 저장된다. 이후 연달아 $4 \cdot n$ 바이트에는 각 파일에 해당하는 n 개의 원소값이 역시 `int` 타입으로 저장된다.

주의 3 CUDA 코드의 수행 시간을 측정할 때는 데이터 이동 시간은 제외한 순수한 커널 수행 시간만 측정하라.

주의 4 자신의 PC 환경이 허용하는 범위 내에서 최대한 큰 n 값을 찾은 후, n_f 값을 적절히 바꾸어 가면서 실험을 진행하라. n 은 최소 2^{24} 보다 큰 2의 제곱 수이어야 하며, n_f 인자는 다음과 같은 값을 고려할 것.

$$n_f = 1, 4, 16, 64, 256, 1024.$$

주의 5 CUDA 커널 작성 시 1차원 또는 2차원 그리드를 사용하며, 쓰레드 블럭의 크기를 적절히 바꾸어 가면서 수행 시간이 어떻게 변화하는지 실험을 통하여 확인하라. 특히 주어진 블럭 크기에 대하여 n_f 값을 증가시켜가면서 실험을 한 후 GPU 수행 시간이 어떻게 증가하는지 관찰하라.

주의 6 당연히 여러분의 프로그램을 정확한 결과를 생성해야 하며, 이는 조교가 자신의 방법으로 확인할 예정임.

실습 문제 2

Shared memory를 사용한 행렬-벡터 곱셈 계산의 병렬 가속

조교가 실습 시간에 설명.

5 숙제

제출 마감: ?월 ?일 오후 ?시 정각

제출물 및 방법: 조교가 실습 시간에 공지

이차원 영상에 대한 Gaussian filter 적용 문제

이차원 영상을 읽어 들인 후 주어진 크기의 Gaussian filter를 적용한 영상을 출력해주는 프로그램을 작성하라. 특히 아래에 기술한 세 가지 형태의 프로그램을 작성한 후 다양한 쓰레드 블럭 크기에 따른 수행 시간을 비교하라.

- CPU 코드
- Shared memory 비사용 GPU 코드
- Shared memory 사용 GPU 코드

자세한 내용은 조교가 실습 시간에 설명.