JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 18, No. 3, pp. 613-619, June. 2017

GPU 하드웨어 아키텍처 기반 sub-warp 단위 병렬 프리픽스(prefix) 연산의 정확한 구현

박 태 정
덕성여자대학교 디지털미디어학과

Correct Implementation of Sub-warp Parallel Prefix Operations based on GPU Hardware Architecture

Taejung Park
Department of Digital Media, Duksung Women's University, Samyangro 144gil 33, Dobong-gu Seoul 01369, Korea

[요 약]

본 논문에서는 대규모 데이터를 길이가 32 미만인 로컬 세그먼트 단위로 구분하고 이 로컬 세그먼트 내에서 정확한 GPU 병렬 프리픽스(prefix) 연산 결과를 출력하는 CUDA (Compute Unified Device Architecture) 코드를 제시한다. 이미 Mark Harris와 Michael Garland가 이러한 목적을 수행하기 위한 CUDA 코드를 이미 발표한 바 있으나 본 논문에서는 로컬 세그먼트의 길이가 32 미만일 때 기존 코드의 결과가 정확하지 않다는 사실을 살펴 보고 그 원인을 논의한 후, 정확한 결과를 출력하는 코드를 제안한다. 본 논문에서 다루는 로컬 세그먼트 단위의 병렬 프리픽스 연산은 최인접 요소 탐색(k-nearest neighbor search) 등은 물론 다양한 대규모 병렬 처리 알고리즘을 구성하는 기본 연산으로 활용 가능하다.

[Abstract]

This paper presents a CUDA (Compute Unified Device Architecture) code to achieve correct GPU parallel segmented prefix operation results with less than 32 segment length for large data arrays. Mark Harris and Michael Garland had published CUDA code to address the tasks. This paper shows that their code does not generate correct results when the local segment length is less than 32, discusses the cause of the problem, and presents a CUDA code that generates correct results. The segmented parallel prefix operation presented in this paper can be applied as a building block to various large parallel processing algorithms including the k-nearest neighbor search problems.

색인어 : CUDA, GPGPU, 병렬 프리픽스 연산, 세그먼트 적용 익스클루시브 스캔

Key word : CUDA, GPGPU, Parallel prefix operation, Segmented exclusive scan

http://dx.doi.org/10.9728/dcs.2017.18.3.613

Received 09 June 2017; **Revised** 16 June 2017
Accepted 25 June 2017

***Corresponding Author; Taejung Park**

Tel: +82-02-901-8339
E-mail: tjpark@duksung.ac.kr

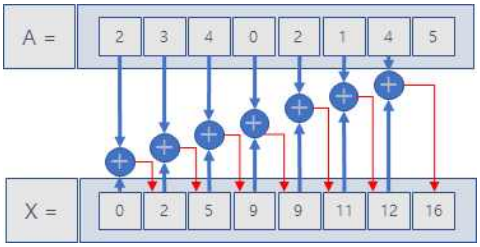

그림 1. 익스클루시브 스캔 연산 과정 (덧셈)
Fig. 1. Exclusive scan operation (addition)

Ⅰ. 서 론

GPGPU를 위한 병렬 처리에서 익스클루시브 스캔(exclusive scan) 등으로 표현되기도 하는 병렬 프리픽스(prefix) 연산[1]은 여러 연산에서 사용되는 중요한 연산 방식이다. 특히 대규모 연산을 짧은 시간에 수행한 후에 메모리 대역폭 한계로 인한 시간 지연이 발생하는 구조의 병렬 연산 아키텍처에서 계산한 결과는 주로 배열(array) 형태로 전송된다. 이 때 계산한 각 개별 배열 요소의 위치를 계산하기 위해 보통 프리픽스 (특히 exclusive sum) 연산이 필요하다.

일반적인 대규모 병렬 연산에서는 블록 단위 이상(inter-block)에서 병렬 연산이 중요하다고 보는 경우가 많으나 로컬 범위에서 가장 가까운 주변 탐색(k-nearest search) 등을 수행하는 대규모 병렬 처리에서 보다 작은 로컬 범위에서 최대, 최소 연산이 필요한 경우 GPU 하드웨어의 물리적 병렬 최소 실행 단위로 정의되는 warp 크기 이하의 범위 (k < 32)에서의 병렬 프리픽스 연산이 필요하다.

이러한 중요성으로 인해 NVIDIA 사가 개발한 Fermi 아키텍처[2] 이후 하드웨어 레벨에서 보다 효율적인 병렬 작업을 수행할 수 있도록 warp 단위 명령이 등장하였으며 [3]에서 NVIDIA 사의 Mark Harris와 Michael Garland는 이러한 명령을 이용한 병렬 프리픽스 연산 루틴을 소개했다. 그러나 이 소스 코드는 warp 크기 이하(sub-warp) 세그먼트를 설정하고 프리픽스를 계산할 경우 결과가 정확하지 않다.

본 논문에서는 병렬 프리픽스 연산을 살펴 보고 Mark & Michael의 코드를 분석한 후 이 분석을 토대로 이 오류를 수정한 정확한 코드를 제시한다.

Ⅱ. 병렬 프리픽스(Prefix) 연산 개요

2-1 일반적인 배경 지식

1) 프리픽스 연산

n개의 원소를 가지는 집합 A와 임의의 이진 연관 연산(binary associative operation) ⊙에 대해 스캔(scan) 또는 병렬 prefix 연산은 다음과 같이 정의된다[3].

$$A = [a_0, a_1, a_2, a_3, \ldots a_n]$$

$$S = inclusvie\ scan(\odot, A)$$
$$= [a_0, a_0 \odot a_1, a_0 \odot a_1 \odot a_2, \ \ldots, a_0 \odot a_1 \ldots \odot a_n]$$

이 때 집합 S를 정의하는 프리픽스(prefix) 연산은 인클루시브 스캔(inclusive scan)이라고 한다. 이와 유사하게 초기값 p를 별도로 고려하는 스캔 연산을 정의할 수 있는데 이 스캔 연산을 익스클루시브 스캔(exclusive scan)이라고 하며 다음과 같이 정의된다.

$$X = exclusive\ scan(\odot, p, A)$$
$$= [p, p \odot a_0, p \odot a_0 \odot a_1, \ldots, p \odot a_0 \odot a_1 \ldots \odot a_n]$$

예를 들어 A = [2, 3, 4, 0, 2, 1, 4, 5]이고 초기값 p = 0일 경우

익스클루시브 스캔의 결과는 [0, 2, 5, 9, 9, 11, 12, 16]이다(그림 1). 이 익스클루시브 스캔은 GPGPU 환경에서 중요한 대규모 병렬 연산을 실행한 후 출력 또는 중간 배열에 각 스레드가 연산 결과값을 저장하기 위한 배열 색인값을 계산하기 위한 용도로 널리 활용된다. 따라서 효율적인 병렬 프리픽스 연산 알고리즘이 GPGPU 초창기부터 연구되어 왔으며 NVIDIA CUDA 구현 시 메모리 대역폭, 메모리 종류, 액세스 패턴에 대한 고려와 루프 풀기(loop unrolling) 등이 병렬 처리 속도에 어떠한 영향을 주는지 설명하는 예제 코드로 제시되기도 했다[4].

좀더 구체적인 병렬 처리 과정에서 병렬 프리픽스 연산이 가지는 의의는 이후에 좀 더 자세하게 논의한다.

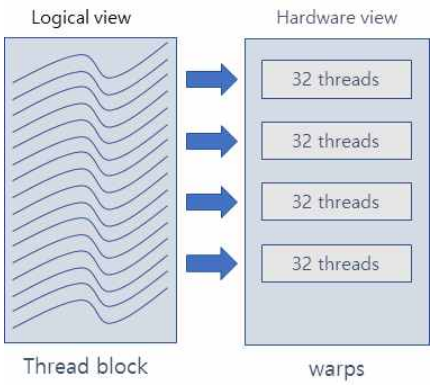

그림 2. CUDA의 논리적 개요와 하드웨어 개요
Fig. 2. Logical view and hardware view of CUDA

2) CUDA 구조

CUDA 프로그래밍 환경은 병렬 프로그래밍을 수행하기 위해 다층적인 레벨에서 다양한 개념으로 바라 볼 수 있다. 그림 2에서는 이러한 개념들 중 하나로 일반적인 프로그래밍을 위한 논리적 개념(logical view)과 이 논리적인 개념이 실제 GPU 하드웨어에서 작동되는 하드웨어 개념(hardware view)을 제시하고 있다.

CUDA 병렬 코딩 레벨에서는 코드 실행의 최소 단위인 스레드(thread)의 집합인 스레드 블록(thread block, 그림 1에서 왼쪽)을 정의하고 해결할 병렬 문제의 크기에 맞춰 이 스레드 블록들을 구성한다. 이 스레드 블록은 GPU 내에서 레지스터와 공유 메모리(shared memory)를 공유하며 Streaming Multiprocessor(SM)에서 '이론적으로' 동시에 실행된다. 그리고 이 스레드 블록들은 다시 그리드(grid)라고 하는 상위 집합을 구성한다[5].

코딩 레벨에서 이렇게 정의된 스레드 블록은 실제로 GPU 하드웨어에서 32개의 스레드들을 동시에 실행할 수 있는 멀티프로세서들에 할당된다(그림 1의 오른쪽). 이 때 하드웨어에서 동시에 실행할 수 있는 32개의 스레드들은 warp이라는 단위로 정의한다. 즉, 물리적으로 동시에 실행되는 하드웨어 최소 단위는 warp 단위(즉, 32개 스레드들이 동시에 실행)이다. 이에 비해 앞서 한 스레드 블록 내에서의 스레드들은 '이론적'으로 동시에 실행된다는 의미는 실제 GPU의 Streaming Multiprocessor의 하드웨어 성능에 따라 물리적으로 동시에 실행 가능한 스레드의 개수가 한정되며 따라서 이보다 더 많은 스레드를 포함하는 한 스레드 블록은 사실 상 순차적으로 내부 스레드들을 실행하지만, 이 스레드 블록 내부에서 스레드들이 공유하는 공유 메모리와 레지스터의 수명 주기가 스레드 블록 단위로 유지되며 스레드 실행의 독립성과 상호 작용성이 보장된다는 의미이다.

데이터 크기가 큰 병렬 처리 문제에서는 전역 범위에서 연산이 필요한 경우(예를 들어 전체 데이터 중 최소값, 최대값을 구하거나 전체 데이터를 정렬하는 경우)가 대부분이지만, 최근 병렬 처리 연구에서는 Morton code[6]를 이용하는 shifted sort[7] 등에서 로컬 범위(예를 들어 32개 세그먼트 이내)에서의 병렬 처리가 중요한 경우를 흔히 볼 수 있다. 전역 병렬 연산의 경우 전역 메모리(global memory) 대역폭이 병목 지점으로 작용하기 때문에 warp 단위의 연산 효율성 보다는 메모리 액세스 패턴과 메모리 캐시의 운영에 중점을 두게 되지만 warp 단위 또는 sub-warp 단위의 로컬 세그먼트 병렬 처리에서는 GPU의 Streaming Multiprocessor 코어의 작동 특성 측면에 집중해서 효율성을 개선하는 접근 방법을 취하게 된다. 다시 말해 물리적 병렬 처리 단위인 warp 내에서의 연산에서의 효율성 개선이 더 중요하다.

이러한 중요성에 기초해서, 본 논문에서는 warp 내(intra-warp 단위)에서의 병렬 연산의 효율성 개선에 초점을 맞추어 논의한다.

3) GPGPU에서의 병렬 프리픽스 연산의 의의

GPGPU와 같은 대규모 병렬 처리(massive parallel processing) 연산 환경에서는 일반적으로 병렬 프로세서의 처리 속도가 메모리 읽기/쓰기 속도보다 높기 때문에 대규모 연산 자체는 매우 짧은 시간에 병렬로 처리하지만 결과를 다시 메모리에 쓰기 위해서 상대적으로 많은 시간이 소요된다. 따라서 GPGPU를 위한 GPU 아키텍처와 병렬 처리 알고리즘은 다층적인 프로세서/메모리 아키텍처 각각에서 병렬 프로세서들이 한 연산을 끝나고 그 연산 결과가 메모리에 완전히 저장될 때까지의 시간을 다시 연산에 활용하기 위한 방향으로 발전해 왔다(예. warp scheduler의 스케줄링 방식[8]).

이러한 병렬 연산 구조에서는 보통 큰 크기의 배열(array)로 표현되는 대규모 데이터의 병렬 처리를 매우 짧은 시간에 처리하는 것도 중요하지만 사용자에게 필요한 정보를 정리해서 메모리에 저장할 수 있는 방법도 중요하다. 예를 들어 2^{20}개의 스레드를 이용해서 복잡한 연산을 수행하고 나서 그 결과를 2^{20}개 크기의 배열에 저장했을 때 이차적으로 이 배열의 요소들 중 최대, 최소를 구하거나 정렬을 수행해서 사용자가 인식할 수 있도록 알려야 하는 경우가 대부분이다. 또한 GPGPU의 특성에 적합한 정렬 방식인 래딕스 정렬(radix sort)에서도 중간 과정에서 프리픽스 연산(exclusive sum)을 이용한다[9].

2-2 Fermi 아키텍처에서의 병렬 Prefix 연산 최적화

1) GPU 하드웨어 측면의 지원

일반적인 병렬 처리에서 성능을 결정하는 병목 지점들 하나는 스레드들 사이의 데이터 전달과 그러한 전달을 위한 동기화 부분이다. GPGPU 병렬 처리에서는 스레드들 사이의 데이터 전달은 메모리(공유 메모리 및 전역 메모리)를 매개로 수행되는데 앞서 논의한 대로 메모리 대역폭이 병렬 처리의 성능을 결정하는 주요한 병목 지점들 중 하나로 작용한다는 사실은 다른 관점에서 일반적인 병렬 처리에서의 문제와 동일한 문제라고 볼 수 있다.

NVIDIA사는 GPGPU를 위해 자사의 GPU 아키텍처를 하드웨어적으로 개선해 나가면서 이러한 스레드들 간의 통신 및 동기화 문제를 하드웨어 차원에서 보다 빠르게 수행할 수 있도록 지원하고 있다. 대표적인 사례가 Fermi 아키텍처를 적용하면서 물리적으로 동시에 스레드 실행이 가능한 warp 단위(즉 32개 스레드)에서 공유 메모리나 전역 메모리를 거치지 않고 스레드들이 warp 내부에서 특정 데이터의 연산 결과나 현재 상태를 단 몇 클럭만에 얻을 수 있는 명령어들을 제공한다.

표 1에서는 compute capability 2.0에서부터 하드웨어 차원에서 지원하는 몇 가지 유용한 함수들을 소개한다.

`__popc(uint x)` 함수는 스레드 단위로 실행되며 32비트 `uint (unsigned int)` 변수 내에 1로 설정된 비트 개수(즉, 0~32 사이)를 반환한다. `__clz(uint x)` 함수도 스레드 단위로 수행되며 32비트 `uint (unsigned int)` 변수를 비트 단위로 나열했을 때 MSB(most significant bit)에서부터 1이 처음 나타날때까지 0의 개수를 반환한다. 이 두 함수는 Fermi 아키텍처에서부터 하드웨어 지원을 통해 머신 인스트럭션 1개만으로 실행된다.

표 1. Compute capability 2.0에서부터 제공되는 유용한 함수[10]
Table. 1. Some useful operations provided since compute capability 2.0 [10]

primitives (compute capability 2.0)	description
`int __popc(uint x)`	**"Population Count":** Count the number of bits that are set to 1 in a 32 bit integer **Returns :** Returns a value between 0 and 32 inclusive representing the number of set bits. **Description:** Count the number of bits that are set to 1 in x
`int __clz(uint x)`	**"Count Leading Zeros":** Return the number of consecutive high-order zero bits in a 32 bit integer. **Returns:** Returns a value between 0 and 32 inclusive representing the number of zero bits. **Description:** Count the number of consecutive leading zero bits, starting at the most significant bit (bit 31) of x.
`uint __ballot(int predi)`	Evaluate predicate for all active threads of the warp and return an integer whose Nth bit is set if and only if predicate evaluates to non-zero for the Nth thread of the warp and the Nth thread is active.

`__ballot` 함수는 하드웨어에서 동시에 실행되는 warp 단위, 즉 32개 스레드에서 각각의 스레드가 가진 변수 `predi` ("predicate")가 0이 아닌 값일 경우 해당 스레드의 색인(thread index, 0~31)에 해당하는 비트를 1로 설정한 32비트 `unsigned int` 값을 반환하고 그 값은 warp 내의 모든 스레드가 즉시 공유한다. 그림 3에서 `__ballot` 함수의 실행 예제를 제시한다(이 예제에서는 Thread ID 17부터 31까지 모두 Predicate의 값을 0으로 가정).

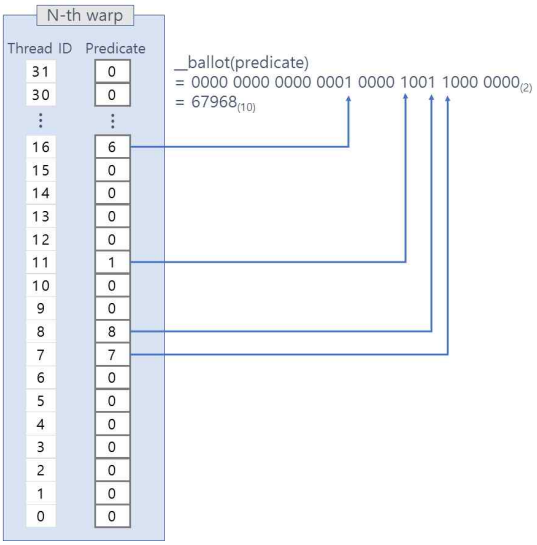

그림 3. __ballot 함수
Fig. 3. __ballot function

표 2. Mark Harris와 Michael Garland의 warp 내부 분할 이진 prefix sum 코드
Table. 2. Original intra-warp segmented binary prefix sum code by Mark Harris and Michael Garland

```
__device__ unsigned int lanemask()
{
  const unsigned int lane
        = threadIdx.x & (WARP_SIZE -1); //Ⓐ
  return (1 << (lane)) - 1;              //Ⓑ
}

__device__ unsigned int warp_segscan(bool p,
                                     unsigned int hd)
{
  const unsigned int mask = lanemask();

  hd = (hd | 1) & mask;

  unsigned int above = __clz(hd) + 1;
  unsigned int segmask = ~((~0U) >> above);

// Perform the scan

  unsigned int b = __ballot(p);
  return __popc(b & masklt & segmask);
}
```

2) Mark Harris와 Michael Garland의 구현 코드

Mark Harris와 Michael Garland는 직전에 논의하였던 Fermi 아키텍처의 새로운 하드웨어 기반 명령들을 이용해서 보다 효율적인 warp 내(intra-warp) prefix sum 코드를 [3]에서 소개하였다(표 2). 이 CUDA 코드에서 소개된 `warp_segscan` 함수는 warp 단위로 실행될 뿐만 아니라 32비트 unsigned int 타입의 인자 hd에 설정된 비트 패턴을 통해 warp 미만 단위, 즉, sub-warp 단위에서의 세그먼트(세그먼트 길이는 32 개 미만의 배열)별로 병렬 prefix sum 연산을 수행할 수 있다. 이러한 sub-warp 단위에서의 prefix sum 연산은 로컬 범위의 정렬이 필요한 shifted sort[7] 같은 애플리케이션에 필수적이며 GPGPU 구조에 이상적인 radix sort [9]에서 로컬 세그먼트 단위의 병렬 정렬 알고리즘에 바로 사용할 수 있다.

`lanemask()` 함수는 thread block 내에 있는 스레드들의 ID (threadIdx.x)로부터 현재 warp 내에서의 색인(lane, 0~31)을 계산한 후(표 2 Ⓐ) LSB(least significant bit)에서부터 현재 lane 번호까지 1로 설정된 32비트 unsigned int 값을 반환한다(표 2 Ⓑ). 이 코드에서 CUDA 상수 WARP_SIZE (표 2 Ⓐ)는 32로 설정되는 시스템 상수이다.

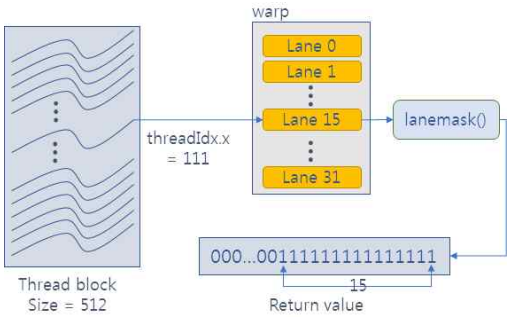

그림 4. lanemask() 함수
Fig. 4. lanemask() function

그림 4에서는 thread block 크기가 512 (즉, 이 블록 내에는 스레드가 512개 존재)일 때 threadIdx.x = 111인 thread가 warp 내부에서 lane 번호(15)를 어떻게 할당 받는지 제시하고 있다.

특히 표 2 Ⓐ의 bitwise AND 연산은 모듈러(modulus, %) 연산, 즉, `lane = threadIdx.x % WARP_SIZE`과 동일하지만 CUDA에서 상대적으로 좀 더 많은 시간이 걸리는 모듈러 연산 % 수행을 피하기 위해 적용되었다.

표 2의 `warp_segscan` 함수는 unsigned int 타입 변수인 hd를 통해 길이가 32 미만인 세그먼트 분할 패턴을 정하고 warp 내부에서 bool 타입 변수 p에 따라 익스클루시브 스캔(덧셈)을 수행한 결과를 반환하도록 의도되었다.

Ⅲ. 오류를 수정한 병렬 Prefix 연산 알고리즘

II 장에서는 CUDA의 일반적인 구조에서 warp 단위 개념, 병렬 prefix 연산의 의의을 살펴 보고 Fermi 아키텍처 이후에 도입된 하드웨어적인 개선을 활용해서 Mark Harris와 Michael Garland가 제안한 보다 효율적인 sub-warp 세그먼트 단위 병렬 prefix sum 알고리즘을 소개했다. 그러나 표 2에서 제시된 알고리즘은 인자 hd의 비트 패턴을 통해 정의된 sub-warp 단위의 세그먼트별 prefix sum을 정확하게 실행하지 않는 문제를 발견했다. 따라서 III 장에서는 Mark Harris와 Michael Garland의 알고리즘의 작동 결과를 통해 이 방법이 내포한 오류를 살펴보고 그 해결책을 제시한다.

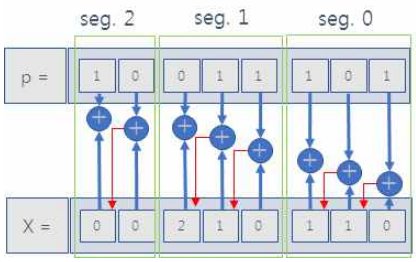

그림 5. 세그먼트 (길이 = 3) 적용 exclusive sum의 정확한 결과
Fig. 5. Exact result from segmented (length = 3) exclusive sum

segment pattaern	0	1	0	0	1	0	0	1
segment no.	2		1			0		
warp lane	7	6	5	4	3	2	1	0
p	1	0	0	1	1	1	0	1
hd = hd|1	0100 1001	0100 1001	0100 1001	0100 1001	0100 1001	0100 1001	0100 1001	0100 1001
mask	0111 1111	0011 1111	0001 1111	0000 1111	0000 0111	0000 0011	0000 0001	0000 0000
hd= hd&mask	0100 1001	0000 1001	0000 1001	0000 1001	0000 0001	0000 0001	0000 0001	0000 0000
above	26	29	29	29	32	32	32	33
(~0u)>>above	0011 1111	0000 0111	0000 0111	0000 0111	0000 0000	0000 0000	0000 0000	0000 000
segmask	1100 0000	1111 1000	1111 1000	1111 1000	1111 1111	1111 1111	1111 1111	1111 1111
b=__ballot(p)	1001 1101	1001 1101	1001 1101	1001 1101	1001 1101	1001 1101	1001 1101	1001 1101
b&mask&segmask	0000 0000	0001 1000	0001 1000	0000 1000	0000 0101	0000 0001	0000 0001	0000 0000
__popc(b & mask & segmask)	0	2	2	1	2	1	1	0

그림 6. Mark Harris와 Michael Garland의 sub-warp exclusive sum의 결과. 세그먼트 크기 3.
Fig. 6. Result table of Mark Harris and Michael Garland's sub-warp exclusvie sum with size of segment = 3.

3-1 기존 알고리즘의 오류

1) 구현 결과

그림 6에서는 8개 스레드에 대해 p = [1, 0, 0, 1, 1, 1, 0, 1]을 길이 3인 세그먼트로 나누고(hd = $01001000_{(2)}$) 표 2에 제시한 Mark Harris와 Michael Garland의 exclusive sum 연산을 수행한 결과를 제시한다. 각 행은 소스 코드가 실행되는 각 단계에서 변수값의 변화를 스레드별로 제시한다. 또한 이 예제의 정확한 계산 결과는 그림 5에서 확인할 수 있다.

그림 5와 그림 6을 비교해 보면 Mark Harris와 Michael Garland의 결과(x= [0, 2, 2, 1, 2, 1, 1, 0])가 정확한 결과(x = [0, 0, 2, 1, 0, 1, 1, 0])와 다름을 알 수 있다. 따라서 원래 저자들이 제시한 코드(표2)는 저자들의 주장과는 달리 sub-warp 레벨에서는 올바로 작동하지 않는다.

segment pattaern	0	1	0	0	1	0	0	1
segment no.	2		1			0		
warp lane	7	6	5	4	3	2	1	0
p	1	0	0	1	1	1	0	1
hd = hd|1	0100 1001	0100 1001	0100 1001	0100 1001	0100 1001	0100 1001	0100 1001	0100 1001
maskle	1111 1111	0111 1111	0011 1111	0001 1111	0000 1111	0000 0111	0000 0011	0000 0001
hd= hd&maskle	0100 1001	0100 1001	0000 1001	0000 1001	0000 1001	0000 0001	0000 0001	0000 0001
masklt	0111 1111	0011 1111	0001 1111	0000 1111	0000 0111	0000 0011	0000 0001	0000 0000
above	26	26	29	29	29	32	32	32
(~0u)>>above	0011 1111	0011 1111	0000 0111	0000 0111	0000 0111	0000 0000	0000 0000	0000 0000
segmask	1100 0000	1100 0000	1111 1000	1111 1000	1111 1000	1111 1111	1111 1111	1111 1111
b=__ballot(p)	1001 1101	1001 1101	1001 1101	1001 1101	1001 1101	1001 1101	1001 1101	1001 1101
b&mask&segmask	0000 0000	0000 0000	0001 1000	0000 1000	0000 0000	0000 0001	0000 0001	0000 0000
__popc(b & masklt & segmask)	0	0	2	1	0	1	1	0

그림 7. 본 논문에서 제시하는 수정한 코드(표 3)의 실행 결과
Fig. 7. Result table by corrected source code (Table 3)

3-2 오류 수정 방안

1) 오류의 원인 분석

앞서 논의한 대로 Mark Harris와 Michael Garland의 exclusive sum 연산은 sub-warp 단위에서는 올바로 작동하지 않는다. 가장 큰 이유 중 하나는 표 2에서 제시된 mask 변수에 계산되는 비트 패턴이 각 세그먼트들을 완전히 커버하지 않고 가장 왼쪽 비트가 항상 0이 되는 문제가 있다는 점이다.

문제를 해결하기 위해 제안하는 소스 코드에서는 커버 범위

가 다른 두 개의 mask를 계산하고 각각 masklt와 maskle로 지정했다. 이렇게 수정한 소스 코드는 표 3에서 제시한다.

그림 7은 앞서 논의한 예제와 동일한 문제에 대해 표 3을 실행시킨 결과이다. 이 결과는 그림 5의 정확한 세그먼트 적용 sub-warp exclusive sum 결과와 일치한다.

표 3. 본 논문에서 제안하는 오류 없는 warp 내부 분할 이진 prefix sum 코드

Table. 3. Corrected Intra-Warp segmented binary prefix sum code

```
__device__ unsigned int lanemask_lt()
{
  const unsigned int lane
        = threadIdx.x & (WARP_SIZE -1);
  return (1 << (lane)) - 1;
}
__device__ unsigned int lanemask_le()
{
  const unsigned int lane
           = threadIdx.x & (WARP_SIZE -1);
  return (1 << (lane+1)) - 1;
}
__device__ unsigned int warp_segscan(bool p,
                              unsigned int hd)
{
  const unsigned int masklt = lanemask_lt();
  const unsigned int maskle = lanemask_le();
  hd = (hd | 1) & maskle;
  unsigned int above = __clz(hd) + 1;
  unsigned int segmask = ~((~0U) >> above);

  unsigned int b = __ballot(p);
  return __popc(b & masklt & segmask);
}
```

3-2 결과 분석

표 3을 통해 제안하는 정확한 세그먼트 적용 sub-warp exclusive sum을 테스트하기 위해서 다양한 데이터 크기에 대해 실행 시간을 테스트하였다. 실행 시간 측정 방식은 초기 실행 시 오버헤드가 적용되는 GPU의 특성 상 정확한 시간 측정을 위해 [11]에서 제시한 warm up 커널 실행 후 측정 방식을 적용했다. 테스트 방식은 데이터 크기에 해당하는 배열의 모든 원소를 1로 설정한 후 각 스레드별로 warp_segcan 함수에서 bool 타입 인자 p에 값을 전달한 후 병렬 처리를 수행하도록 하였다.

테스트를 수행한 하드웨어 환경은 NVIDIA 사의 GTX Titan X (메모리 12GB)를 장착한 Intel i7 CPU(3.60GHz), RAM 16GB 시스템이며 Windows 7(64bit), CUDA 8.0을 적용하였다. 또한 실행 코드는 Visual Studio 2012 버전에서 CUDA 코드와 C++ 코드 모두 64비트 release 모드로 컴파일을 수행하였다.

표 4에서는 측정 결과를 제시한다. 이 실행 시간에는 GPU 상에서 warp_segcan 함수 실행뿐만 아니라 입력 배열 값에 1을 병렬 처리로 쓰는 루틴까지 포함되었다. 블록 크기(즉, 한 블록 내 스레드 개수)는 512로 지정했고 그리드(grid)는 1개만 사용했다. 입력 배열은 unsigned int 타입으로 선언했고 그 크기는 2^{21}(=2,097,152)부터 시작해서 2배씩 증가시키면서 2^{28}(=268,435,456)까지 테스트했다.

실행 결과, 세그먼트를 구분하는 hd 패턴은 전체 실행 속도에 영향이 없는 것으로 확인되었다. 표 4의 결과를 보면 2^{22}부터 데이터 크기가 2배씩 증가함에 따라 실행 시간도 약 2배씩 증가하는 사실을 확인할 수 있다. 따라서 제안하는 알고리즘은 입력 데이터 개수에 따라 선형적으로 실행 시간이 증가한다고 할 수 있다.

표 4. 데이터 크기에 대한 실행 시간

Table. 4. Execution time for various data sizes

data size	time (ms)
2^{21}	3
2^{22}	3
2^{23}	7
2^{24}	13
2^{25}	26
2^{26}	51
2^{27}	94
2^{28}	185

Ⅳ. 결 론

최근 shifted sort 등과 같은 새로운 병렬 처리 알고리즘들이 다양한 문제에 적용되면서 대규모 데이터를 32개 이하의 작은 로컬 세그먼트로 나누고 이 세그먼트 단위에서 프리픽스 연산 결과를 이용하는 알고리즘이 필요하게 되었다. Mark Harris와 Michael Garland는 [3]를 통해 NVIDIA사에서 Fermi 아키텍처부터 적용되기 시작한 하드웨어 기반 명령을 사용함으로써 병렬 프리픽스 연산의 효율을 크게 높일 수 있음을 증명했고 로컬 세그먼트로 나눈 후 이 단위에서 프리픽스 연산을 수행하는 소스 코드를 공개한 바 있다.

그러나 본 논문에서는 Mark Harris와 Michael Garland의 코드는 로컬 세그먼트 크기가 32 미만인 경우 올바른 결과를 출력하지 않음을 보였고 그 원인을 분석한 후 정확한 연산을 수행하는 코드를 제시하였다.

참고문헌

[1] wikipedia. Available: https://en.wikipedia.org/wiki/Prefix_sum
[2] Fermi architecture white paper. Available: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf
[3] M. Harris, M. Garland, and W. Hwu (editor-in-chiefs), GPU Computing Gems Jade Edition, 1st ed. Morgan Kaufmann Pub., ch. 3, pp. 29-38, 2011.
[4] Parallel Prefix Sum on the GPU (Scan). Available: http://www.umiacs.umd.edu/~ramani/cmsc828e_gpusci/ScanTalk.pdf
[5] CUDA C Programming guide. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4jURrxaId
[6] T. Park, “Analysis of Morton Code Conversion for 32 Bit IEEE 754 Floating Point Variables”, The Journal of Digital Contents Society, Vol. 17, No. 3, pp. 165-172, June 2016.
[7] S. Li, L. Simons, J. B. Pakaravoor, F. Abbasinejad, J. D. Owens, and N. Amenta, “kANN on the GPU with shifted sorting,”. In Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics (EGGH-HPG'12), Switzerland, pp. 39-47, 2012.
[8] J. Cheng, M. Grossman, and T. McKercher, Professional CUDA C Programming, 1st ed. Wrox, pp. 90-93, 2014.
[9] Mark Harris, GPU Gems 3, ch. 39. “Parallel Prefix Sum (Scan) with CUDA”. Available: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html
[10] CUDA Toolkit documentation. Available: http://docs.nvidia.com/cuda/
[11] J. Cheng, M. Grossman, and T. McKercher, Professional CUDA C Programming, 1st ed. Wrox, pp. 84-87, 2014.

박태정(Taejung Park)

1997년 : 서울대 전기공학부 (공학사)
1999년 : 서울대 전기공학부 대학원 (공학 석사, 반도체 물리 전공)
2006년 : 서울대 전기컴퓨터공학부 대학원 (공학박사, 컴퓨터 그래픽스 전공)
2006년～2013년: 고려대학교 연구교수
2013년～현재 : 덕성여자대학교 디지털미디어학과 조교수
※관심분야 : 컴퓨터그래픽스, 병렬처리, 게임 물리, 수치해석, 3차원 모델링