creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사 학위 논문

GPU를 위한 Feature Extraction 및 Matching Algorithm 병렬화

Parallelization of Feature Extraction and Matching Algorithm for GPU Architecture

2018년 2월

서울대학교 대학원

전기 컴퓨터 공학부

이 철 희

GPU를 위한 Feature Extraction 및 Matching Algorithm 병렬화

Parallelization of Feature Extraction and Matching Algorithm for GPU Architecture

지도교수 이혁재

이 논문을 공학박사 학위논문으로 제출함

2018년 2월

서울대학교 대학원

전기 컴퓨터 공학부

이 철 희

의 공학박사 학위 논문을 인준함

2018년 2월

위 원 장: 채수익 (인)

부위원장: 이혁재 (인)

위 원: 조남익 (인)

위 원: 오성회 (인)

위 원: 이규중 (인)

초 록

Feature matching은 image 사이에서 유사한 feature를 찾기 위하여 computer vision 분야에 많이 사용된다. Matching 알고리즘은 feature 사이의 geometry 정보를 사용하는가에 따라 first-order matching과 high-order graph matching 방법으로 분류된다. 일반적으로 사용되는 방법은 first-order matching으로, 각 feature의 주변 영역에 대한 정보를 기반으로 유사한 feature를 찾는다. 그러나, 이 방법은 feature 사이의 geometry 정보를 고려하지 않기 때문에 반복되는 패턴, 텍스쳐 또는 불분명한 주변 영역 등으로 인한 이미지 모호성에 민감하다는 단점이 있다. 반면, high-order graph matching 은 feature 사이의 geometry 정보를 기반으로 matching을 수행한다. 이 algorithm은 first-order matching에 비하여 반복되는 형태나 불분명한 영역에 더 강인하다. 그러나, high-order matching는 높은 연산 복잡도로 인하여 처리 속도가 매우 느리다. 본 논문에서는 GPU를 위한 high-order graph matching의 네 가지 병렬화 방법을 제시한다.

첫째, high-order graph matching의 symmetric 특징을 이용하여 GPU에서 적합하지 않은 memory 접근 pattern을 제거한다. High-order graph matching에서 병렬 처리를 어렵게 하는 것은 같은 memory 위치에서 여러 개의 thread들이 동시에 data를 update함으로써 발생되는 write collision이다. High-order graph matching에서는 같은 solution을 생성할 수 있는 여러 개의 objective function이 존재한다. 이 성질을 이용하여 제안하는 방법은 write collision을 일으키는 operation들을 같은 solution을 갖는 다른 operation들로 교체하여 write collision을 제거한다.

둘째, memory 사용량 감소를 위한 Approximate Spectral Matching algorithm을 적용하고, 이를 GPU에서 효율적으로 처리하기 위한 병렬처리 방법을 제시한다. High-order graph matching은 similarity 값을 저장하기 위하여 매우 큰

memory 사용량을 요구한다. Approximate Spectral Matching 은 feature 사이의 geometry 정보를 approximation함으로써, similarity의 개수를 감소시켰다. 그러나, 이 방법은 sparse tensor를 이용해야 하며, 이는 GPU에서 효율적으로 처리되기 어려운 memory 접근 pattern을 발생시킨다. 본 연구에서는 이 memory 접근 pattern이 나타나는 연산 과정을 graph matching algorithm에서 분리하고, 분리된 연산을 GPU에서 효율적으로 처리하기 위한 thread structure를 제시한다.

셋째, memory 사용량을 추가적으로 감소시키기 위한 approximation 방법을 제시한다. Approximate Spectral Matching을 사용할 때, 높은 matching 정확도를 위해서는 approximation 정도가 낮아야 한다. 그러나, approximation 정도가 낮을 때, memory 사용량은 크게 감소되지 못한다. Approximate Spectral Matching Algorithm이 reference feature set만 approximation한다는 고려하여, 본 연구에서는 query feature set도 함께 approximation하는 방법을 제시한다. 제안하는 방법을 사용함으로써, similarity 개수가 크게 감소되며, memory 사용량은 approximation 정도에 영향을 받지 않는다. 또한, shared memory를 이용하여 제안하는 approximation 방법을 GPU에서 효율적으로 처리하기 위한 방법을 제시한다.

넷째, 제안하는 approximation 방법의 GPU utilization 향상을 위한 tuple clustering 방법을 제안 한다. 제안하는 approximation 방법을 사용할 경우, thread-block 사이의 workload unbalance가 발생한다. 이는 GPU streaming multiprocessor의 workload를 달라지게 하며, streaming multiprocessor의 utilization 을 크게 하락 시킨다. Thread-block의 workload unbalance가 발생하는 원인은 각 thread-block이 접근하는 data structure가 sparse하며, 이러한 sparse data structure의 non-zero element 개수가 서로 다르기 때문이다. 본 연구에서는 tuple clustering 방법을 이용하여 sparse data structure들의 non-zero element 개수를 유사하게 만든다. 또한, score 기여도 기반 clustering 방법을 제시하여 matching 정확도 하락을 최소화한다.

Scale-invariant feature transform (SIFT)는 computer vision 분야에서 가장 널리

사용되는 feature이다. 이 feature는 크기, 회전 등 다양한 image 변화에 강인하다. 그러나, 이러한 높은 강인함을 위하여 SIFT의 연산 복잡도는 매우 높으며, 이를 실시간으로 처리하는 것은 어렵다. 본 연구에서는 SIFT 알고리즘을 GPU에서 효율적으로 처리되기 위한 방법을 제시한다. SIFT에서 scale-space 생성 과정을 분석하고, Gaussian filter 크기와 scale-space image 크기를 동시에 감소시키는 것이 feature의 강인함을 크게 하락시키지 않으면서 큰 속도 향상을 얻을 수 있음을 확인하였다. 이러한 관찰로부터, 실시간 처리를 위하여 SIFT 알고리즘을 변형하고 구현하였다. CPU와 GPU를 효율적으로 이용하는 최적화 방안을 제시하여, 추가적인 속도 향상을 달성하였다.

***Index Terms*— High-order graph matching, feature matching, GPU 최적화, 병렬처리, Scale-invariant feature transform, CUDA**

차 례

초 록 i

차 례 iv

그림 목차..... vii

표 목 차..... xii

제1장 서 론.....1

1.1 연구 배경 1

1.2 연구 내용 6

1.3 논문 구성 9

제2장 High-order Graph Matching Algorithm 기존 연구 10

2.1 Assignment matrix..... 10

2.2 Tuple similarity..... 12

2.3 Affinity tensor..... 14

2.4 Score function..... 15

2.5 Power iteration algorithm..... 17

제3장 High-Order Graph Matching 알고리즘의 GPU 수행을 위한 효율적인 병렬화 방법 19

3.1 GPU 병렬화를 위한 data access pattern 분석..... 19

3.2 GPU를 위한 제안하는 graph matching 알고리즘..... 23

3.3 실험 결과..... 30

제4장 Approximate Spectral Matching algorithm을 위한 GPU 병렬화.....49

4.1 Approximate Spectral Matching 알고리즘..... 49

4.2 Sparse tensor를 위한 기존 연구 적용..... 57

4.3 제안하는 sparse index tensor를 처리 방법..... 62

4.4 실험 결과..... 71

제5장 Memory 사용량 감소를 위한 approximation기반 base tensor sparsification.....76

5.1 Memory 사용량 감소를 위한 base tensor sampling..... 76

5.2 제안하는 base tensor sparsification..... 80

5.3 제안하는 approximation 방법의 GPU 구현..... 87

5.4 실험 결과..... 93

제6장 GPU에서 효율적인 index tensor 처리를 위한 tuple clustering.....98

6.1 Index tensor의 형태와 처리 시간과의 관계..... 98

6.2 효율적인 병렬화를 위한 tuple clustering..... 102

6.3 Score 기여도 기반 clustering 방법 105

6.4 실험 결과 110

제7장 SIFT에서 scale-space 변형 기반 complexity 감소 방법 117

7.1 SIFT overview 117

7.2 Matching 성능과 complexity 분석 121

7.3 제안하는 scale-space 생성 방법 128

7.4 Mobile GPU를 위한 최적화 134

7.5 실험 결과 139

제8장 결 론.....155

참 고 문 헌.....158

그림 목차

그림 2-1 Matching된 features 형태 11

그림 2-2 Assignment matrix X..... 12

그림 2-3 Graph 비교 예시..... 14

그림 2-4 Algorithm 1: Power iteration method..... 18

그림 3-1 The thread structure for the 3rd-order graph matching..... 20

그림 3-2 Tensor-vector multiplication에 대한 block-level memory access pattern..... 21

그림 3-3 다양한 조건에 따른 tensor-vector multiplication의 data access pattern ... 24

그림 3-4 $jv < iv < kv$ 조건을 이용한 power iteration algorithm 25

그림 3-5 Write collision을 제거한 power iteration algorithm..... 27

그림 3-6 Power iteration을 위한 제안하는 알고리즘 28

그림 3-7 제안하는 tensor-vector multiplication의 memory access pattern..... 29

그림 3-8 ANN 기반 first-order matching과 제안하는 high-order graph matching의 matching 정확도 비교..... 35

그림 3-9 tensor-vector multiplication를 수행할 때, 각 SM이 active 상태인 비율 38

그림 3-10 Feature 개수에 따른 처리 시간..... 39

그림 3-11 다양한 개수의 matching candidate에 따른 matching 결과 41

그림 3-12 제안하는 방법이 적용되었을 때 처리 시간..... 42

그림 3-13 matching 성능을 측정하기 위한 test image..... 44

그림 3-14 그림 3-13의 test image에서 matching candidate 개수에 따른 matching 정확도 변화 45

그림 3-15 그림 3-13의 test image에서 matching candidate 개수에 따른 correct match 개수 변화 46

그림 3-16 그림 3-13 (a) test image에 대한 matching 결과 47

그림 4-1 Approximate Spectral Matching algorithm의 $10°$ approximation 예 50

그림 4-2 Affinity tensor에서 같은 Ta에 속하는 두 tuple의 similarity 51

그림 4-3 Index tensor와 base tensor 52

그림 4-4 Base tensor 생성 과정 54

그림 4-5 Index tensor와 base tensor를 이용한 power iteration algorithm 55

그림 4-6 Index tensor과 base tensor을 이용한 GPU 기반 power iteration algorithm 56

그림 4-7 Sparse matrix의 효율적인 병렬화를 위한 data structure 57

그림 4-8 같은 GPU에서의 sparse matrix-vector multiplication 동작 58

그림 4-9 Sparse index tensor에 대한 기존 sparse data 처리 방법 적용 59

그림 4-10 $j1$, $k1$ index들을 이용한 V 접근 pattern 61

그림 4-11 $Ut, i1$ 63

그림 4-12 식 (4-2)에 thread들의 memory 접근 pattern 63

그림 4-13 Thread들이 하나의 $Un, i1$ element를 연산할 때의 memory 접근 pattern 65

그림 4-14 한 개의 thread가 하나의 $Ut, i1$ element를 연산할 때의 memory 접근 pattern 66

그림 4-15 그림 4-14의 첫 번째 순서 67

그림 4-16 제안하는 $Ut, i1$ 생성 과정 70

그림 4-17 House test set 73

그림 4-18 Transform 정도에 따른 matching accuracy 75

그림 5-1 feature 개수에 따른 memory 사용량 77

그림 5-2 base tensor memory 사용량 감소를 위한 기존 연구 78

그림 5-3 Sparsification 정도에 따른 matching 개수 변화 79

그림 5-4 Feature set 2에 대한 base tensor approximation 81

그림 5-5 Base tensor의 분리 82

그림 5-6 Base tensor들의 index tensor가 동일 83

그림 5-7 Base tensor approximation에 의한 affinity tensor 분리 84

그림 5-8 Index tensor와 index 개수 85

그림 5-9 Reference index tensor와 query index tensor 및 similarity matrix를 이용한 power iteration algorithm 86

그림 5-10 각 block에서 graph matching을 수행할 때, 접근하는 index tensor와 similarity matrix 88

그림 5-11 GPU에서 approximation base tensor의 처리 방법 90

그림 5-12 shared memory를 이용한 query index tensor 처리 방법 91

그림 5-13 제안하는 최적화 방법에 따른 memory 접근 pattern. 92

그림 5-14 Approximated base tensor의 matching 정확도 비교 97

그림 6-1 두 개의 SM이 접근하는 index tensor 99

그림 6-2 base tensor의 index 개수에 따른 처리 속도 100

그림 6-3 각 SM에서의 처리 시간과 GPU에서의 처리 시간 관계 100

그림 6-4 GPU에서 효율적으로 처리되기 위한 index tensor 101

그림 6-5 제안하는 tuple clustering 전략 103

그림 6-6 Reference index tensor를 위한 score 기여도 기반 clustering 방법. 104

그림 6-7 Graph matching이 완료되었을 때, matching된 모습 106

그림 6-8 하나의 reference index tensor에 대한 score 기여도 107

그림 6-9 Reference index tensor의 non-zero element 개수와 score 기여도의 관계 108

그림 6-10 Clustering 전과 후의 query index tensor를 이용한 memory 접근 pattern 109

그림 6-11 Reference index tensor 개수가 666개일 때, clustering 전과 후의 SM activity 비교 111

그림 6-12 Approximated tuples 개수가 171개일 때, clustering 전과 후의 SM activity 비교 113

그림 6-13 Tuple clustering에 따른 matching 정확도 비교 116

그림 7-1 SIFT 동작 120

그림 7-2 $\sigma init$ 변화에 따라 검출되는 feature들의 scale 변화 123

그림 7-3 Matching score와 feature 개수에 대한 $\sigma init$의 영향 124

그림 7-4 DoG space에서 scale 1에서의 scale selection 과정 127

그림 7-5 그림 7-3에서 low-scale 부분 127

그림 7-6 $\sigma init$에 따른 scale-selection 정확도를 측정하기 위한 실험 130

그림 7-7 Scale selection 정확도에 대한 $\sigma init$의 영향 131

그림 7-8 제안하는 Gaussian scale-space 생성 방법 132

그림 7-9 Octave 1의 첫 번째 scale을 위한 pre-octave generation 133

그림 7-10 Mobile CPU와 GPU에서 제안하는 algorithm의 구현 137
그림 7-11 하나의 octave에서 Gaussian scale-space의 병렬 생성 방법 138
그림 7-12 Test image sets 139
그림 7-13 SIFT algorithm들의 feature 개수 151
그림 7-14 Non-SIFT algorithm과의 feature 개수 비교 152
그림 7-15 다양한 algorithm의 matching score 153
그림 7-16 다양한 algorithm의 precision-recall curve 154

표 목 차

표 3-1 tensor-vector multiplication에서 thread 개수와 처리 시간 30

표 3-2 Tensor-vector multiplication의 profile 결과 32

표 3-3 Graph matching algorithm의 처리 시간 32

표 3-4 energy 효율 비교 34

표 3-5 ANN 기반 first-order matching과 high-order graph matching의 수행 시간 . 34

표 3-6 제안하는 방법과 reduction 방법에서 tensor-vector multiplication의 처리 시간 36

표 3-7 다른 high-order graph matching algorithm들의 수행 시간 42

표 4-1 10°도 approximation에서 tensor-matrix multiplication 과정의 처리 시간.. 72

표 4-2 5°도 approximation에서 tensor- matrix multiplication 과정의 처리 시간... 72

표 5-1표는 affinity tensor의 non-zero element 개수 93

표 5-2 Affinity tensor의 memory 사용량 비교 94

표 5-3 5°도 단위 approximation에서 feature 개수에 따른 처리 시간 비교 95

표 5-4 10°도 단위 approximation에서 feature 개수에 따른 처리 시간 비교..... 96

표 6-1 제안하는 clustering 방법의 적용 전과 후의 처리 시간 비교 114

표 7-1 SIFT에서 load instruction의 수행 횟수 122

표 7-2 Half-type 변경에 따른 처리 시간 변화 136

표 7-3 Original SIFT 대비 제안하는 SIFT의 data 접근 비율 140

표 7-4 다양한 구현에 따른 SIFT 처리 시간 142

표 7-5 Feature 검출 algorithm들의 평균적인 처리 시간 142

표 7-6 다양한 SIFT 구현에서 정확히 matching된 feature의 개수 147

제1장 서 론

1.1 연구 배경

Feature matching은 image에서 유사한 feature를 찾기 위하여 computer vision 분야에 많이 사용된다. Feature matching은 물체 인식 [1], shape matching [2], image retrieval [3], 3D reconstruction [4], optical flow estimation [5] 등의 응용에 널리 사용되고 있다. Feature matching의 정확도에 따라서 이를 사용하는 응용의 범위가 결정되기 때문에, 정확한 feature matching을 위한 연구들이 활발히 진행되고 있다. Matching 알고리즘은 feature 사이의 geometry 정보를 사용하는가에 따라 first-order matching과 high-order graph matching 방법으로 분류된다. 일반적으로 사용되는 방법은 first-order matching으로, 각 feature의 주변 영역에 대한 정보를 기반으로 유사한 feature를 찾는다. Feature 주변 영역의 정보를 표현하기 위하여 SIFT [1], SURF [6], BRIEF [7] descriptor가 많이 사용된다. 가장 유사한 feature는 Euclidean distance를 이용하여 검색된다. 이 방법은 feature 사이의 geometry 정보를 고려하지 않기 때문에 반복되는 패턴, 텍스쳐 또는 불분명한 주변 영역 등으로 인한 이미지 모호성에 민감하다는 단점이 있다. 이러한 문제를 해결하기 위하여 feature 위치들 간의 상관관계를 보고 matching을 수행하는 high-order graph matching 방법이 연구되고 있다[8-15]. 이 방법은 feature들의 위치를 graph의 node로, feature 사이의 geometry 정보를 graph의 edge로 활용하기 때문에 graph matching이라고 부른다. n차원 graph matching 방법에서는 n 개의 feature 사이의 geometry 정보를 기반으로 matching을 수행하게 된다.

First-order matching 방법과 비교하여 graph matching 방법의 단점은 연산 복잡도가 높다는 것이다. 두 이미지에서 추출된 feature의 개수가 각각 N_1과 N_2일

때, First-order matching의 complexity는 $O(N_1N_2)$ 이다. 반면, third-order graph matching의 complexity는 $O(N_1^3N_2^3)$으로 연산 복잡도가 더 높다. 이는 CPU에서 느린 처리 속도로 나타나게 된다. First-order matching 방법으로 사용한 nearest neighbor matching 방법은 수십 ms 이내의 처리시간이 소요되는 반면([16],[17]), high-order graph matching은 수 초 이상 소요되었다[10]. High-order graph matching의 높은 정확도에도 불구하고 이러한 높은 복잡도는 실시간 수행을 요구하는 다양한 응용 분야에 대한 적용에 걸림돌이 되고 있다.

High-order graph matching 알고리즘의 높은 복잡도의 원인은 feature 조합 간의 similarity를 저장해 놓는 공간인 affinity tensor의 크기가 매우 크기 때문이다. 따라서, affinity tensor의 크기를 감소시키기 위한 연구들이 진행되었다. Reference image에서 feature들을 임의로 sampling하여 matching 조합 개수를 감소시키는 방법이 시도되었다 ([11]-[13]). 추가적인 감소를 위하여, sampled feature들과 비교될 query image의 feature 개수를 k-nearest neighbor과 같은 algorithm을 사용하여 제한하였다. 그러나 실제 application에서 feature들은 feature extraction algorithm([1], [6], [16]-[19])으로부터 추출되며 이러한 feature의 개수들이 매우 많아, 이 방법은 일반적으로 사용되기 어렵다. Affinity tensor의 크기를 감소시키기 위하여 일반적으로 시도되는 방법은 first-order matching을 사전 단계에서 이용하는 것이다 ([9], [10]). Feature descriptor를 이용하여 first-order matching을 수행하고, 낮은 distance를 갖는 matching 조합들이 matching candidate가 된다. 그리고, high-order matching은 이 matching candidate에 대해서만 수행된다. 또 다른 방법으로, [10]은 approximated matching similarity를 이용하여 중복되는 matching을 제거하였다. 그러나, 이러한 연구에도 불구하고 여전히 실시간 동작을 위한 처리속도에는 미치지 못하고 있다.

복잡한 알고리즘들을 실시간으로 처리하기 위하여 GPU 활용 기술에 관한 연구가 최근 활발히 진행되고 있다. First-order matching을 위한 GPU 활용 기술은 많이 시도되어 왔으며 높은 처리 속도 향상을 달성하였다. SIFT descriptor를

활용한 first-order matching의 처리 속도를 GPU를 사용하여 25배 향상시키는 연구가 수행되었다 [20]. 또한, first-order matching 알고리즘 가운데 가장 많이 사용되는 kNN 알고리즘의 속도를 30배[21], kNN의 근사 알고리즘인 ANN 알고리즘을 10배 이상 속도를 향상시킨 연구가 수행되었다[22]. 하지만, high-order graph matching의 연산 구조는 first-order matching의 연산 구조와 완전히 다르다. 따라서, 기존의 연구 결과들을 high-order graph matching에 적용시키기는 어렵다.

Lowe[1]가 제안한 Scale-invariant feature transform (SIFT)는 가장 강인한 feature 검출 algorithm 중 하나로 알려져 있다. 이 feature는 다양한 image 변화 및 뒤틀림에도 높은 성능을 보여준다. Scale에 불변하기 위하여, SIFT는 input image로부터 scale space를 생성하고, 이 scale space에서 feature를 검출한다. 회전에 불변하기 위하여, SIFT는 각 feature의 dominant orientation을 추출하고, 이 orientation에 따라 feature descriptor를 생성한다. 그러나, 이러한 SIFT 강인함을 위하여 연산 복잡도가 매우 높아졌으며, 이는 SIFT가 실시간 application에서의 동작을 어렵게 한다.

SIFT feature 생성 과정의 속도를 향상시키기 위한 많은 연구가 진행되었다 [32]-[42]. 이러한 연구는 일반적으로 두 가지 방법으로 분류된다. 첫 번째 방법은 original algorithm을 변경하지 않고 효율적으로 구현하는 것이다. Huang et al.[32]는 segment-based operation을 이용한 SIFT hardware 가속기를 제안하였다. 결과적으로, 640x480 image에 대하여 30fps의 처리 속도가 달성되었다. GPU를 위한 SIFT algorithm의 병렬 구현도 연구되었다[33]-[35]. Heymann et al. [33]는 인접한 pixel들을 하나의 texture pixel로 packing함으로써 GPU vector operation을 효율적으로 이용하였다. 이 구현은 QuadroxFX 3400 GPU를 이용하여 20 fps의 처리 속도를 달성하였다. Sinha et al. [34]도 SIFT algorithm을 병렬화하여 NVIDIA 7800 GTX GPU에서 10fps의 처리 속도를 달성하였다. Wu et al. [35]은 NVIDIA 8800 GTX GPU를 사용하여 CUDA와 GLSL을 이용한 두 가지 SIFT 구현을 제시하였으며, 각각 27.1fps와 23.0fps의 처리 속도를 달성하였다.

두 번째 방법은 SIFT를 algorithmic level에서 변형한 것이다. Algorithm을 단순화하는 것은 matching 정확도를 감소시키지만, 적은 computing resource로 높은 속도 향상을 달성할 수 있다. Desktop GPU platform에서는 SIFT의 실시간 처리가 병렬화와 최적화 만으로 상대적으로 쉽게 달성되기 때문에, algorithmic-level 변형은 거의 시도되지 않았다. 반면, algorithm 단순화는 hardware 구현에서 많이 연구되었다. Yao et al. [36] 과 Jiang et al. [37] 연산 복잡도를 감소시키기 위하여 scale의 개수와 Gaussian filter의 standard deviation 값을 낮추었다. 이 구현들은 병렬화 정도를 향상시키기 위하여 descriptor 생성 과정도 변형시켰다. Yao et al.가 제안한 방법은 640x480 image에서 SIFT를 30fps로 처리하며, Jiang et al.이 제안한 방법은 512x512 image에서 150.38fps의 처리 속도를 보여준다. Chiu et al. [38]은 scale-space 생성 과정의 복잡도를 감소시키기 위하여 Gaussian filter를 integral image기반 box filter로 교체하였다. 이 방법은 1920x1080 image에 대하여 30fps의 처리 속도를 달성하였다.

Desktop GPU 는 높은 성능은 computing power 를 제공한다. 그러므로, SIFT 의 실시간 처리가 challenging 하지는 않다[33]-[35]. 최신 high-end GPU 의 clock 속도는 1 GHz 가 넘으며, 수 천개의 core 가 존재한다. 반면, mobile GPU 의 경우, 제한된 device 공간과 power 소모량으로 인하여 clock 속도는 600 MHz 정도이며 256 개의 core 정도만 존재한다. 그러므로, 고성능의 desktop GPU 를 위한 기존 방법들은 mobile GPU 를 위한 실시간 SIFT 처리 방법에 사용될 수 없다. Mobile GPU 를 위한 SIFT 구현은 [39], [40]에서 연구되었다. 두 연구 모두 Qualcomm Snapdragon S4 Pro APQ8064 processor 를 사용하였으며, 이 processor 는 1.5GHz quad-core CPU 와 Adreno 320 GPU 를 포함하고 있다. Rister et al. [39]은 CPU 와 GPU 사이의 heterogeneous computing 을 이용하여 feature detection 을 가속화하였으며, 320x280 image 에 대하여 10fps 의 처리 속도를 달성하였다. Wang et al. [40]은 SIFT 의 workload 를 분석하고 task 를 CPU 와 GPU 에 분배하여 320x256 image 에서 10fps 의 처리 속도를 달성하였다. 최신

smartphone 의 camera 해상도가 증가함에 따라, 두 구현 모두 high resolution image 에서 실시간 application 에 사용되기에는 적합하지 않다. 그러므로, mobile GPU 에서는 high resolution image 에서 실시간 처리를 달성하기 위하여 SIFT algorithm 을 변경할 필요가 있다.

1.2 연구 내용

본 연구에서는 GPU에서 high-order graph matching algorithm과 SIFT algorithm을 효율적으로 처리하기 위한 다섯 가지 병렬 처리 방법을 제안한다. 첫째로 GPU 구조에 적합한 graph matching algorithm 연산 처리 방법을 제안한다. Graph matching algorithm은 iteration 과정으로부터 최적의 solution을 찾아나간다. 이때 iteration 과정은 GPU에서 병렬화 될 수 없다. 따라서, 각 iteration에서 수행되는 연산 시간을 최소화함으로써 전체 수행시간을 감소시켜야 한다. 본 연구에서는 각 iteration에서 수행되는 연산 과정의 memory 접근 pattern을 분석한다. 이때 문제가 되는 것은 다수의 thread들이 동시에 동일한 memory 주소에 write 동작을 수행하는 write collision이다. Write collision은 병렬로 수행되어야 하는 thread들을 순차적으로 처리되게 함으로써 수행 시간을 크게 증가시킨다. 본 연구에서는 write collision을 유발하는 memory 접근 pattern을 제거하기 위하여 affinity tensor의 supersymmetric 특성을 이용한다. 동일한 결과를 갖는 연산 과정들을 조합하여 write collision이 발생하는 memory access pattern만을 제거한다.

둘째로 affinity tensor의 memory 사용량을 감소시키기 위하여 approximated matching similarity[10]를 차용하고, sparse tensor를 효율적으로 처리하는 방법을 제안한다. Affinity tensor는 두 feature set 사이의 similarity 값들은 저장하는 tensor로, memory 사용량이 매우 크다. Approximated matching similarity를 사용함으로써, affinity tensor는 두 종류의 tensor(index tensor와 base tensor)로 분리된다. 이 때 index tensor는 sparse tensor이며, GPU에서 효율적으로 처리되기 어렵다. 기존의 GPU를 위한 sparse matrix-vector multiplication연구를 sparse index tensor에 적용해 볼 수 있다. 그러나, 이 방법을 적용하였을 때, GPU에서 효율적으로 처리되기 어려운 memory 접근 pattern이 발생한다. 본 연구에서는 이 memory 접근 pattern이 나타나는 연산 과정을 graph matching algorithm에서 분리한다. 그리고 분리된 연산을 효율적으로 처리하기 위한 thread structure를 제시한다.

셋째, memory 사용량을 추가적으로 감소시키기 위한 approximation 방법을 제시한다. Approximate Spectral Matching을 사용할 때, 높은 matching 정확도를 위해서는 approximation 정도가 낮아야 한다. 그러나, approximation 정도가 낮을 때, memory 사용량은 크게 감소되지 못한다. Approximate Spectral Matching Algorithm이 reference feature set만 approximation한다는 고려하여, 본 연구에서는 query feature set도 함께 approximation하는 방법을 제시한다. 제안하는 방법을 사용함으로써, similarity 개수가 크게 감소되며, memory 사용량은 approximation 정도에 영향을 받지 않는다. 또한, shared memory를 이용하여 제안하는 approximation 방법을 GPU에서 효율적으로 처리하기 위한 방법을 제시한다.

넷째, 제안하는 approximation 방법의 효율적인 GPU 병렬화를 위한 tuple clustering 방법을 제안 한다. Approximation 정도에 따라 memory 사용량은 변하지 않게 되었지만, approximation 정도가 낮을수록 처리 시간은 증가하게 된다. 속도 향상을 위하여 approximation 정도를 높일 경우, matching 정확도가 하락하게 되는 문제점도 있다. 또한, 제안하는 approximation 방법에서는 thread-block 사이의 workload imbalance가 발생한다. 이는 GPU streaming multiprocessor의 workload를 달라지게 하며, streaming multiprocessor의 utilization을 크게 하락 시킨다. Thread-block의 workload unbalance가 발생하는 원인은 각 thread-block이 접근하는 data structure가 sparse하며, 이러한 sparse data structure의 non-zero element 개수가 서로 다르기 때문이다. 본 연구에서는 approximation 정도를 높이면서도 matching 정확도 하락 정도를 낮추는 tuple clustering 방법을 제시한다. 각 approximated tuple들의 score 기여도를 측정하고, score 기여도에 따라 approximation 정도를 다르게 한다. 이를 위하여 낮은 approximation에서 affinity tensor를 생성하고, score 기여도에 따라 clustering 동작을 수행한다. 이때, score 기여도는 high-order graph matching algorithm이 완료되었을 때 측정될 수 있는 값이다. 그러나, score 기여도는 approximated tuple들의 non-zero element 개수로 approximation 될 수 있다. 따라서, index tensor를 non-zero element 개수로

clustering함으로써, approximation에 의한 matching 정확도 하락을 막는 동시에 workload balance도 맞춘다.

다섯째, SIFT feature 생성 과정을 GPU에서 효율적으로 처리하는 방안을 제안한다. 이를 위하여 algorithmic-level 단순화와 GPU-specific 최적화 방법이 같이 시도 되었다. SIFT 생성 과정에서 처리해야 하는 data 양은 처리 속도를 결정하는 중요한 요소이다. 따라서, space-space 생성 과정의 특징을 분석하여 data 접근량을 감소시킨다. Data 접근량을 감소시키기 위한 첫 번째 방법은 Gaussian filter의 σ 값을 감소시킴으로써 Gaussian filter 크기를 감소시키는 것이다. 두 번째 방법은 scale-space 생성 과정에서 사용되는 image 크기를 감소시키는 것이다. 제안하는 SIFT는 GPU에서 동작하도록 OpenCL로 구현되었으며, 추가적인 속도 향상을 위하여 몇 가지 GPU-specific 최적화 방법과 함께 CPU와 GPU 사이의 이종 구조의 utilization을 향상시키기 위한 방법이 적용되었다.

1.3 논문 구성

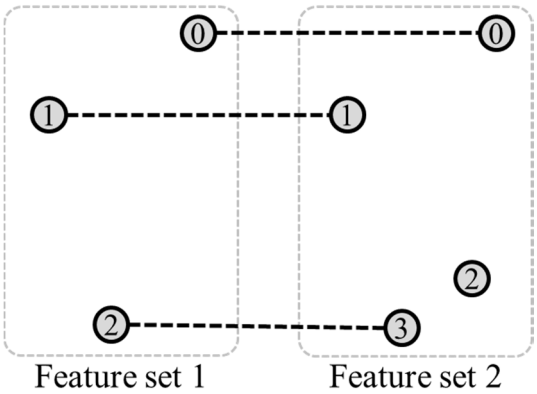
본 연구의 나머지 구성은 다음과 같이 구성된다. 2장에서는 high-order graph matching algorithm 기존 연구를 소개한다. 3장에서는 GPU 구조에 적합한 graph matching algorithm 연산 처리 방법에 대하여 자세히 설명한다. 또한 제안하는 연산 처리 방법에 대한 실험 결과를 확인한다. 4장에서는 affinity tensor의 memory 사용량을 감소시키기 위하여 approximated matching similarity를 이용한 방법[10]을 설명하고, sparse tensor를 효율적으로 처리하는 방법을 제시한다. 5장에서는 affinity tensor의 memory 사용량을 추가적으로 감소시키기 위한 approximation 방법을 제시한다. 6장은 제안하는 approximation 방법을 효율적으로 parallelization 하기 위한 tuple clustering 방법을 제안한다. 7장에서는 SIFT feature 생성 과정을 GPU에서 효율적으로 처리하는 방안을 제안한다. 마지막으로 본 연구에 대한 결론을 맺는다.

제2장 High-order Graph Matching Algorithm 기존 연구

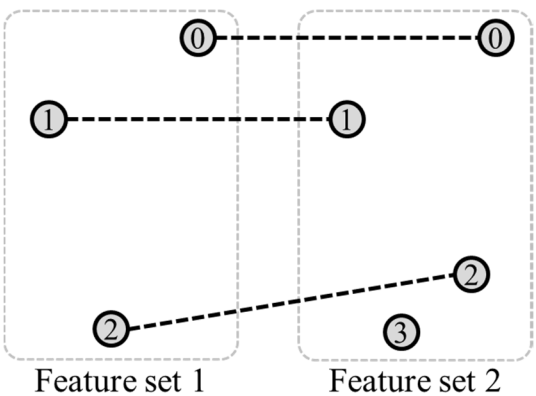
2.1 Assignment matrix

두 feature set 사이에서는 feature가 matching 될 수 있는 다양한 조합이 존재한다. 그림 2-1 (a)와 (b)는 feature set 1과 feature 2가 다르게 matching된 예를 보인다. 회색 circle들은 각 feature set을 구성하는 feature들을 나타내며, feature set 1에는 3개의 feature가, feature set 2에는 4개의 feature가 존재한다. Feature안의 숫자들은 feature를 가리키는 index를 의미한다. Feature set 사이의 점선들은 feature가 matching된 것을 나타낸다. 그림 2-1 (a)에서는 feature set 1의 feature 0,1,2가 각각 feature set 2의 feature 0,1,3에 matching이 되었다. 반면, 그림 2-1 (b)에서는 feature 0,1,2가 feature set 2의 feature 0,1,2에 matching이 되었다.

Graph matching algorithm에서는 그림 2-1과 같은 두 feature set 사이의 matching 정보를 표현하기 위하여 assignment matrix를 사용한다. 그림 2-2 (a)와 (b)의 matrix는 각각 그림 2-1 (a)와 (b)의 matching에 대하여 구성한 assignment matrix X를 나타낸다. 두 feature set을 구성하는 feature의 개수가 각각 N_1개와 N_2개라고 할 때, matching 조합은 N_1 by N_2 binary assignment matrix로 표현될 수 있으며, 이 예제에서 X는 3 by 4 matrix가 된다. 이 matrix의 element는 0과 1로만 표현되며, 그림 2-2 (a)의 $X(0,0) = 1$은 feature set 1의 1번째 feature와 feature set 2의 1 번째 feature가 matching이 되었다는 의미이다. 반면 그림 2-2 (a)의 $X(2,2) = 0$은 feature set 1의 3번째 feature와 feature set 2의 3번째 feature는 matching이 되지 않는다는 의미이다. X는 graph matching algorithm에서 구하고자 하는 목표이다.

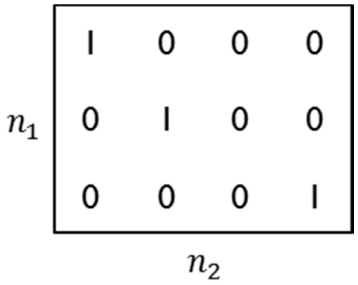


(a) Feature set 1의 features 0, 1, 2가 feature set 2의 feature 0, 1, 3과 match

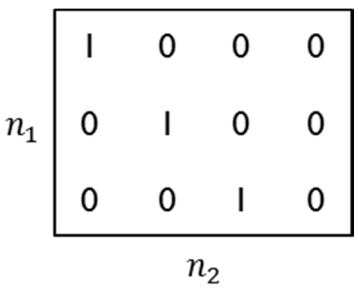


(a) Feature set 1의 features 0, 1, 2가 feature set 2의 feature 0, 1, 2와 match

그림 2-1 Matching된 features 형태



(a) 그림 2-1 (a)로부터 만들어진 Assignment matrix X



(b) 그림 2-1 (b)로부터 만들어진 Assignment matrix X

그림 2-2 Assignment matrix X

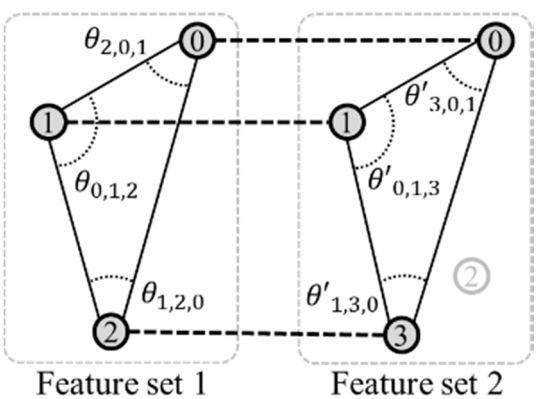
2.2 Tuple similarity

그림 2-1의 두 matching 중에서 어떤 것이 더 잘 된 matching인지 판단하기 위해서 graph matching algorithm이 이용하는 것은 feature 사이의 geometric information이다. Third-order graph matching에서 사용하는 geometric information은 세 feature가 구성하는 삼각형의 형태이다. 세 feature들이 이루는 삼각형을 tuple이라고 할 때, 각 feature set에서 선택된 두 tuple의 similarity를 비교한다. 그림 2-3에서는 그림 2-1의 (a)와 (b) 중에서 어떤 matching이 잘 된 matching인지를 판단하기 위해, 세 feature로 만들어지는 tuple의 similarity를 비교하고 있

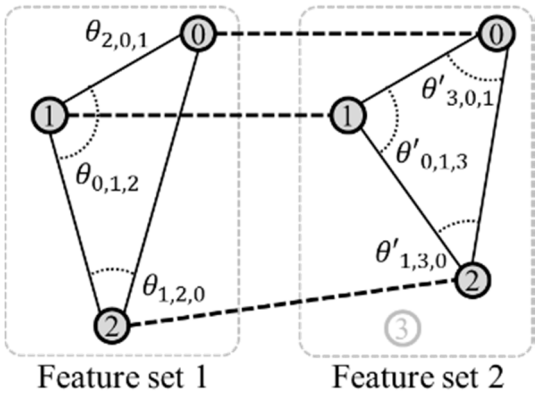
다. 그림에서와 같이 비교되는 두 tuple은 두 feature set이 어떻게 matching되었는가에 따라 다르다. 예를 들어, feature set 1의 (0,1,2) tuple에 대응되는 feature set 2의 tuple은 그림 2-3 (a)에서는 (0,1,3)이지만, 그림 2-3 (b)에서는 (0,1,2)이다. 두 tuple의 similarity 계산을 위하여 사용되는 것은 삼각형의 각도이다. Feature set 1과 2에서 선택한 tuple의 feature index가 각각 (i_1, j_1, k_1), (i_2, j_2, k_2)라고 할 때, 두 tuple의 similarity는 식 (2-1)과 같이 계산된다. Similarity 값이 클수록, 두 tuple은 더 유사한 것이다.

$$Simlarity((i,j,k),(i',j',k')) = exp[-2(|sin(\theta_{k,i,j}) - sin(\theta'_{k',i',j'})| + |sin(\theta_{i,j,k}) - sin(\theta'_{i',j',k'})| + |sin(\theta_{j,k,i}) - sin(\theta'_{j',k',i'})|)] \quad (2\text{-}1)$$

그림 2-3 (a)와 (b)에서는 두 tuple의 각도가 비교되는 예시를 보여준다. 그림 2-3 (a)에서 i_1, j_1, k_1과 i_2, j_2, k_2는 각각 0,1,2와 0,1,3 값을 가진다. Graph 1의 $\theta_{2,0,1}$, $\theta_{0,1,2}$, $\theta_{1,2,0}$는 30°, 40°, 110°이며, graph 2의 $\theta'_{3,0,1}$, $\theta'_{0,1,3}$, $\theta'_{1,3,0}$는 28°, 38°, 114°이므로, 식 (2-1)로부터 계산되는 similarity 값은 0.85이다. 반면, 그림 2-3 (b)에서 i_2, j_2, k_2는 0,1,2이다. $\theta'_{2,0,1}$, $\theta'_{0,1,2}$, $\theta'_{1,2,0}$는 각각 40°, 54°, 86°이며, 이 때의 similarity 값은 0.48이다. 그림 2-3 (a)의 similarity 값이 더 크므로, 그림 2-1 (a)의 경우가 더 잘 된 matching인 것이다.



(a) 그림 1-1 (a)에 대한 graph 비교



(b) 그림 1-1 (b)에 대한 graph 비교

그림 2-3 Graph 비교 예시

2.3 Affinity tensor

그림 2-1 (a)의 matching이 최적의 matching인지를 판단하기 위해서는 두 feature set에서 만들어질 수 있는 모든 matching 조합에 대한 tuple similarity 값과 비교해야 한다. Affinity tensor H는 feature set 1과 2 사이의 모든 tuple similarity 값들을 저장하고 있다. 3rd-order matching에서 H는 6 dimensional tensor이다. Feature set 1과 2에서 선택한 세 개의 feature에 대한 index를 (i_1, j_1, k_1), (i_2, j_2, k_2)라고 할 때, 두 tuple의 similarity 값은 $H((i_1, i_2), (j_1, j_2), (k_1, k_2))$로 표현된다. 예를 들어, $H((0,0),(1,1),(2,3))$은 그림 2-3 (a)에서 비교하는 두 tuple의 similarity 값이며, 이 값은 식 (2-1)로부터 계산된 0.85이다. 반면, $H((0,0),(1,1),(2,2))$은 그림 2-3 (b)에서 비교하는 두 tuple의 similarity 값이며, 이 값은 식 (2-1)로부터 계산된 0.48이다.

Graph matching algorithm에서는 match에 대한 index (i_1, i_2)를 $i_v = i_1 * N_2 + i_2$

로 vectorization하여 사용한다. 이에 따라, X 는 N_1N_2 크기의 vector로 vectorization되어 사용된다. 이 vector를 X_v라고 할 때, $X(i_1, i_2)$는 $X_v(i_1 * N_2 + i_2)$ 가 된다. H 또한 3 dimensional tensor로 변형되어 사용한다. 그림 2-4는 이 tensor를 나타내며, 이를 H_v라고 정하였다. 각 dimension의 size는 n_1n_2로 동일하며, 각 dimension에 대한 index를 i_v, j_v, k_v라고 할 때, i_v, j_v, k_v는 각각 H의 (i_1, i_2), (j_1, j_2), (k_1, k_2) 값들이 vectorization된 값을 갖고 있다. 다시 말하면, $i_v = i_1 * N_2 + i_2$, $j_v = j * N_2 + j_2$, $k_v = k_1 * N_2 + k_2$ 값을 갖는다. 따라서, $H_v(i_v, j_v, k_v)$는 $H(i_1, i_2)$, (j_1, j_2), (k_1, k_2)) 을 의미한다. H_v를 그림 2-1의 두 feature set에 대하여 구성한 것이라고 할 때, H_v의 한 dimension의 size는 12이며 총 element 수는 $12^3 = 1{,}728$이다. $H_v(0,5,11)$은 그림 2-3 (a)에서 비교하고 있는 두 tuple의 similarity에 해당한다. 왜냐하면, feature set 1의 (0, 1, 2) tuple과 feature set 2의 (0,1,3) tuple이 선택되었을 때, (i_v, j_v, k_v) 값은 각각 0*4+0=0, 1*4+1=5, 2*4+3=11 이기 때문이다. 마찬가지로, $H_v(0,5,10)$는 그림 2-3 (b)에서 비교하는 두 tuple의 similarity 값이다. Tuple 비교가 가능한 조건($i_v \neq j_v \neq k_v$)이 아닌 경우에는 식 (2-1)에 의한 similarity 값을 계산할 수 없다. 따라서, 이 경우 H_v 값은 0이다.

2.4 Score function

모든 tuple들이 matching되었을 때, 모든 matching된 tuple의 similarity 값의 합은 다음과 같다:

$$score(X_v) = \sum_{i_v, j_v, k_v} H_v(i_v, j_v, k_v) X_v(i_v) X_v(j_v) X_v(k_v) \qquad (2\text{-}2)$$

주어진 feature set에서 다양한 matching 조합이 존재하며, score(X_v)는 matching

조합 X_v에 영향을 받는다. 최적의 matching에서는 score에 더해지는 similarity 값이 최대가 됨으로써, score 값 또한 최대가 된다. 이것은 graph matching 문제가 식 (2-2)를 최대화 시키는 X_v를 찾는 optimization 문제로 표현될 수 있음을 의미한다.

그림 2-3 (a)의 matching을 나타내는 affinity tensor H_v의 element는 총 6개이다. 즉, $H((0,0),(1,1),(2,3))$, $H((0,0),(2,3),(1,1))$, $H((1,1),(0,0),(2,3))$, $H((1,1),(2,3),(0,0))$, $H((2,3),(0,0),(1,1))$, and $H((2,3),(1,1),(0,0))$이 모두 같은 matching을 나타내므로, 그 값은 동일하다: $H((0,0),(1,1),(2,3)) = H((0,0),(2,3),(1,1)) = H((1,1),(0,0),(2,3)) = H((1,1),(2,3),(0,0)) = H((2,3),(0,0),(1,1)) = H((2,3),(1,1),(0,0))$. Vector화된 형태에서는 다음과 같은 관계를 갖는다: $H_v\,(0,5,11)) = H_v(0,11,5) = H_v(5,0,11) = H_v(5,11,0) = H_v(11,0,5) = H_v(11,5,0)$. 따라서, affinity tensor H(or H_v)는 supersymmetric 한 성질이 있음을 알 수 있다.

이러한 supersymmetry 특징은 식 (2-2)의 score(X_v)가 같은 값을 6번 포함하고 있음을 의미한다. 만약 6개의 동일한 계산 과정 중에서 하나만 수행한다면, 연산 복잡도는 1/6으로 감소될 것이다. 따라서, score(X_v)를 최소화하는 것은 $i_v < j_v < k_v$ 조건에 대해서만 $H_v(i_v,j_v,k_v)X_v(i_v)X_v(j_v)X_v(k_v)$을 계산하고 더하는 score'($X_v$)를 최소화 하는 것과 같다.

$$score'(\mathrm{X}_v) = \sum_{i_v<j_v<k_v} H_v(i_v,j_v,k_v)X_v(i_v)X_v(j_v)X_v(k_v) \qquad (2\text{-}3)$$

2.5 Power iteration algorithm

Score function (2-3)을 최대로 만드는 X_v를 구하는 문제는 integer quadratic programming problem으로 polynomial time 알고리즘이 존재하지 않는 NP-hard 문제이다. 이에 따라 다양한 relaxation기반 접근 방법들이 시도 되었다 ([8]-[13]). 즉, X_v를 relaxation 시킨 후 해를 구하고, 다시 이를 X_v로 바꾸는 것이다. Relaxed assignment matrix에 대한 solution은 가장 큰 eigenvalue값을 갖는 H_v의 principal eigenvector을 구하는 것이다. 이를 계산하는 일반적인 방법은 그림 2-4. Algorithm 1의 power iteration method 이다 [23, 24]. Line 3~13은 power iteration method의 tensor-vector multiplication이 수행되는 과정을 보여준다. 이 과정은 i_v, j_v, k_v로 선택된 세 개의 match $V(i_v)$, $V(i_v)$, $V(i_v)$에 가중치를 준 their tuple similarity $H_v(i_v, j_v, k_v)$를 더하는 방법으로 수행된다. Their tuple similarity 값이 높을수록, V에 더 큰 값이 더해진다. 이 동작은 두 feature set의 모든 tuple 비교 조합에 대하여 수행된다. 모든 tuple 비교 조합은 하나의 unique tensor만 선택함으로써 가능하며, Algorithm 1의 line 3~6은 이를 선택하기 위한 조건이다. 전체 i_v, j_v, k_v 조합에 중에서, $i_v < j_v < k_v$ 경우만 선택하였다. 그리고 line 7~9와 같이 선택된 $H_v(i_v, j_v, k_v)$를 구성하는 3 개의 match에 값을 update하는 동작을 수행한다. 결과 vector V'는 normalization되며, 다시 tensor-vector multiplication의 input이 된다. V는 매 iteration 마다 식 (2-3)의 score를 증가시키는 방향으로 수렴해 간다. V가 충분히 수렴되었을 때, 높은 값이 나타나는 V의 element만 binarization시켜 X를 구성한다. 이 X는 최종 matching 조합이 된다.

1: Initialize V randomly

2: **Repeat**

3: **for** k_v = 0 to $N_1N_2 - 1$ **do**

4: **for** j_v = 0 to $N_1N_2 - 1$ **do**

5: **for** i_v = 0 to $N_1N_2 - 1$ **do**

6: **if** $i_v < j_v < k_v$

7: $V'(i_v) \leftarrow V'(i_v) + H_v(i_v, j_v, k_v)V(j_v)V(k_v)$

8: $V'(j_v) \leftarrow V'(j_v) + H_v(i_v, j_v, k_v)V(i_v)V(k_v)$

9: $V'(k_v) \leftarrow V'(k_v) + H_v(i_v, j_v, k_v)V(i_v)V(j_v)$

10: **end**

11: **end**

12: **end**

13: **end**

14: $V \leftarrow \frac{1}{||V||_2}$

15: **Until** convergence

그림 2-4 Algorithm 1: Power iteration method

제3장 High-Order Graph Matching 알고리즘의 GPU 수행을 위한 효율적인 병렬화 방법

3.1 GPU 병렬화를 위한 data access pattern 분석

본 구현에서는 하나의 thread가 하나의 tensor-vector multiplication (lines 6-10 in Algorithm 1)을 수행한다. 이를 위하여 모든 i_v,j_v,k_v 경우에 대하여 thread를 생성한 후, line 6의 조건에 해당하는 경우에만 GPU의 thread가 line 7~9의 과정을 수행하도록 하였다. 모든 i_v,j_v,k_v 경우의 수는 $(N_1N_2)^3$개이므로, power iteration 과정의 thread 개수는 $(N_1N_2)^3$가 된다.

그림 3-5는 thread structure와 block/thread ID 분배 과정을 보여준다. 왼쪽 그림은 thread structure를 나타내며, $(N_1N_2)^3$개의 work-item으로 구성되어 있다. 전체 work-item에 대하여 2차원 grid를 할당하고 하나의 block에 1차원 thread를 생성하였으며 다음과 같이 thread/block ID를 할당하였다: thread ID=i_v, block ID=(j_v, k_v). 따라서, $H_v(i_v,j_v,k_v)$는 block ID가 (j_v, k_v)이며 thread ID가 i_v인 thread가 접근하게 된다. 이로부터, 하나의 block을 구성하는 work-item들은 각각 다른 i_v 값을 가지며, j_v와 k_v는 같은 값을 가진다. j_v와 k_v는 block마다 서로 다르게 된다.

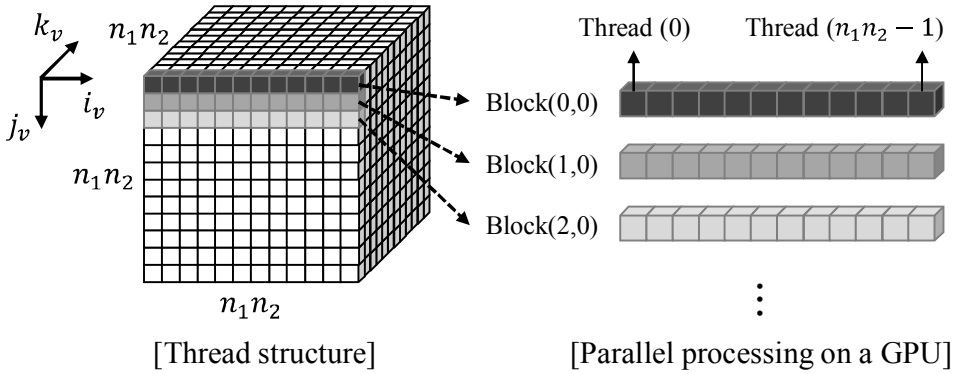
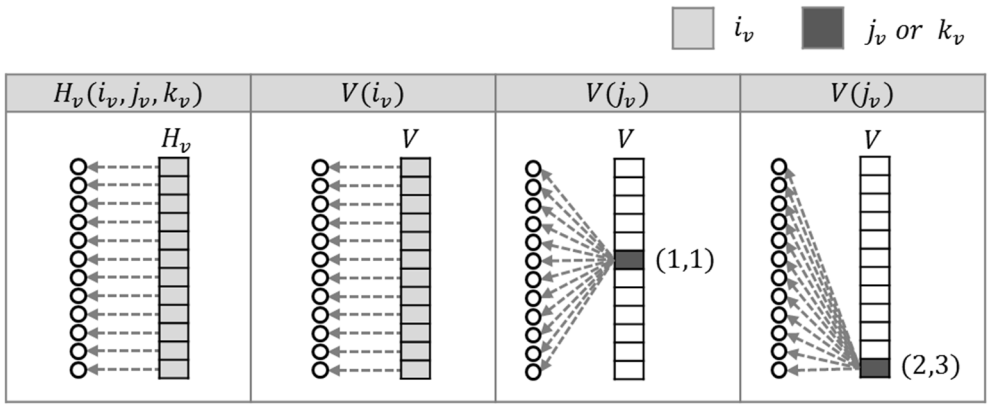


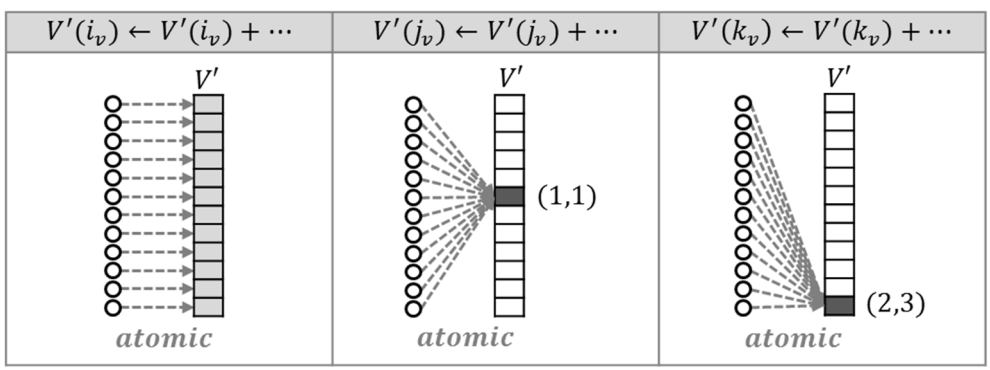
그림 3-1 The thread structure for the 3rd-order graph matching.

그림 3-2는 이러한 thread structure를 기반으로 tensor-vector multiplication 과정의 block-level memory access-pattern을 보여준다. 이 block은 그림 2-3 (a)의 $H_v(0,5,11)$를 이용하는 work-item을 포함하고 있다. 각 그림에서 원 모양은 GPU의 thread들이며, 점선 화살표는 각 thread들이 접근하는 data를 나타낸다. i_v, j_v, k_v들은 thread가 H_v, V, V'를 접근하는 주소가 된다. 그림 3-2 위의 그림에서 보인 바와 같이, i_v가 가리키는 주소에 접근하는 경우 회색을, j_v 또는 k_v가 가리키는 주소에 접근하는 경우 검은색으로 표시하였다. i_v ,j_v, k_v는 각각 scalar 값을 가지지만, 그림 3-2에서는 조금 더 쉬운 이해를 위하여, 이 값들을 (i,i'), (j,j'), (k,k')로 나타내었다. 하나의 block에서 block ID는 동일하므로, block을 구성하는 모든 thread들은 같은 값의 j_v와 k_v 값을 가지며, 각각 $(1,1)$과 $(2,3)$의 값을 갖게 된다. Block을 구성하는 thread들은 thread ID에 따라 다른 i_v 값을 가지며, 각각 0부터 11까지의 값을 갖게 된다. 이 block에서 접근하는 H_v 값은 $H_v(0,5,11)$부터 $H_v(11,5,11)$이다. 그림 3-2 (a)와 그림 3-2 (b)는 각각 read operation과 write operation을 나타낸다. Algorithm 1 line 7~9의 연산을 위해서는 memory로부터 $H_v(i_v,j_v,k_v)$와 $V(i)$, $V(j)$, $V(k)$를 read 해야 한다. 따라서, 그림 3-2 (a)와 같이 네 개의 read operation이 수행된다. 또한, line 7~9에서는

각각 $V'(i)$, $V'(j)$, $V'(k)$의 위치에 연산 결과를 저장하고 있다. 따라서, 그림 3-2 (b)와 같이 세 개의 write operation이 수행된다. Write 동작은 atomic operation 형태로 수행되는데, 그 이유는 memory 주소에 data를 update하는 동안 다른 thread가 해당 data를 변경하지 말아야 하기 때문이다.



(a) Read 동작



(b) Write 동작

그림 3-2 Tensor-vector multiplication에 대한 block-level memory access pattern

Block-level에서 tensor-vector multiplication의 read 동작은 두 가지 형태로 수행

된다. H_v에 해당하는 memory read는 연속된 주소의 data를 가져오는 형태로 수행된다. 또한, V에서 i_v가 가리키는 read 동작도 연속된 주소로부터 값을 읽어온다. 반면, V에서 j_v와 k_v가 가리키는 (1,1)과 (2,3)위치에서의 read 동작은 모든 thread가 동일한 위치에서 값을 읽어오는 broadcasting 형태로 수행된다. 이 두 종류의 memory access pattern은 모두 적은 수의 memory transaction으로 가져오기 때문에 GPU 구조에 적합하다. Block-level에서의 write 동작도 두 가지 형태로 수행된다. $V'(i)$에 대한 write 동작에서는 연속되고 인접한 주소에 write 동작을 수행한다. 이 동작에서는 thread 간의 write collisions이 발생하지 않는다. 반면, $V'(j)$와 $V'(k)$에 대한 write 과정은 모든 thread가 동일한 위치에 값을 update하고 있다. 이 경우 thread간의 write collision이 발생한다. 이는 block 내부의 thread들이 모두 같은 j_v, k_v 값을 갖고 있기 때문이다. 이 write collision은 block내의 thread들이 serialization되는 문제를 발생시키며, tensor-vector multiplication 과정의 처리속도를 증가시킨다.

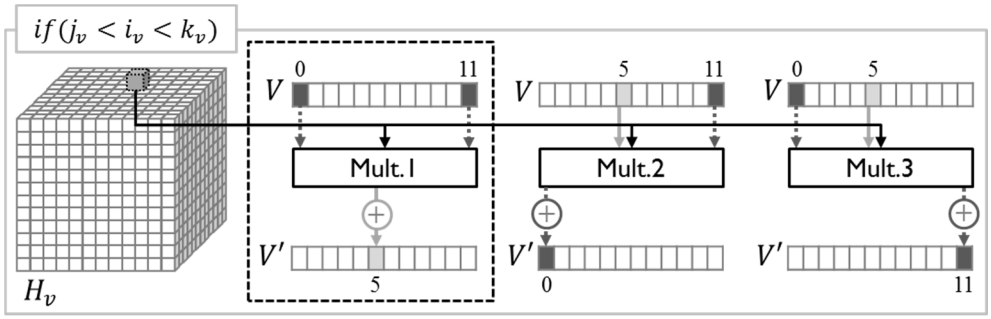
3.2 GPU를 위한 제안하는 graph matching 알고리즘

이 섹션에서는 그림 2-4의 power iteration algorithm에서 발생하는 write collision을 제거하기 위한 새로운 알고리즘을 제시한다. 그림 3-3은 그림 2-4의 line 7, 8, 9에서 수행되는 세 가지 tensor-vector multiplication의 data 접근 형태를 보여준다. 그림에서 Mult.1, Mult.2, Mult.3은 각각 line 7, 8, 9의 세 multiplication 과정을 나타낸다. Mult.2와 Mult.3 과정에서는 같은 j_v와 k_v 주소에 write하는 동작으로 인하여 write collision이 발생한다.

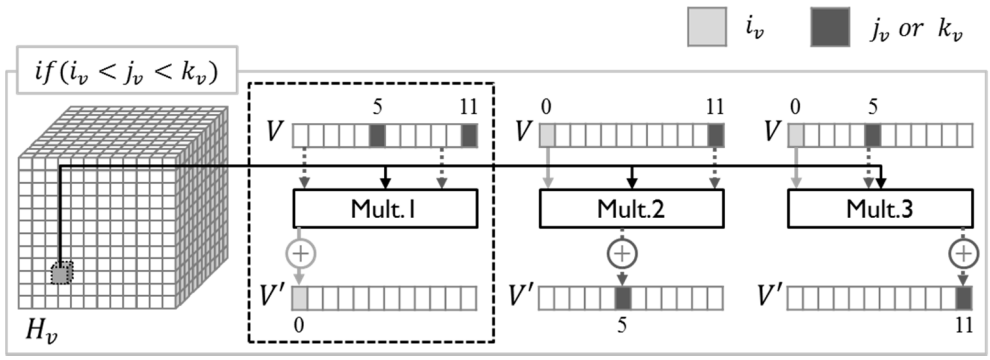
이 write collision은 affinity tensor의 supersymmetry 특성을 이용하여 제거될 수 있다. Affinity tensor의 supersymmetry 특성은 $score'(X_v)$ 을 다른 조건(e.g., $j_v < i_v < k_v$)으로 수행될 수 있게 하며, 이 새로운 score 함수는 아래와 같이 표현될 수 있다.

$$score''(X_v) = \sum_{j_v<i_v<k_v} H_v(i_v, j_v, k_v) X_v(i_v) X_v(j_v) X_v(k_v) \qquad (3\text{-}1)$$

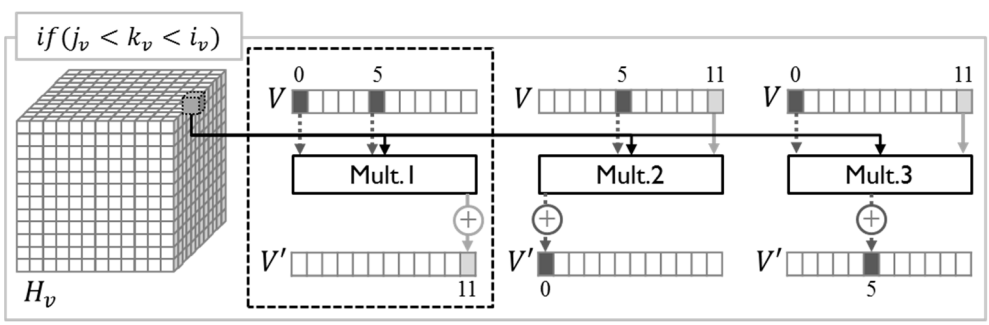
이 새로운 score 함수에 대하여 power iteration 알고리즘을 수행하였을 때 얻어지는 결과는 (2-3)의 $score'(X_v)$ 에 대한 power iteration 알고리즘 수행 결과와 같다. 이것은 그림 2-4의 power iteration 알고리즘이 그림 3-4와 같이 변경될 수 있음을 의미한다.



(a) $i_v < j_v < k_v$



(b) $j_v < i_v < k_v$



(c) $j_v < k_v < i_v$

그림 3-3 다양한 조건에 따른 tensor-vector multiplication의 data access pattern

1: Initialize V randomly
2: Repeat
3: **for** $k_v = 0$ to $N_1N_2 - 1$ **do**
4: **for** $j_v = 0$ to $N_1N_2 - 1$ **do**
5: **for** $i_v = 0$ to $N_1N_2 - 1$ **do**
6: **if** $j_v < i_v < k_v$
7: $V'(i_v) \leftarrow V'(i_v) + H_v(i_v, j_v, k_v)V(j_v)V(k_v)$
8: $V'(j_v) \leftarrow V'(j_v) + H_v(i_v, j_v, k_v)V(i_v)V(k_v)$
9: $V'(k_v) \leftarrow V'(k_v) + H_v(i_v, j_v, k_v)V(i_v)V(j_v)$
10: **end**
11: **end**
12: **end**
13: **end**
14: $V \leftarrow \frac{1}{||V||_2}$
15: Until convergence

그림 3-4 $j_v < i_v < k_v$ 조건을 이용한 power iteration algorithm

그림 3-4은 그림 3-3 (b)와 같이 새로운 data 접근 패턴을 발생시킨다. 마찬가지로, 그림 3-3 (c)와 같이, 다른 수행 조건 $j_v < k_v < i_v$를 이용하여, power iteration 알고리즘을 수행할 수 있다. 이 세 가지 알고리즘에 대하여, 모든 동작이 완료되었을 때 얻어지는 solution은 동일하다. 반면, 세 가지 그림에서도 볼 수 있듯이 세 알고리즘의 data 접근 패턴은 모두 다르다.

그림 3-3 (b)의 Mult.1은 다음 수식(Line 7 in 그림 3-4)을 나타낸다:

$$V'(i_v) \leftarrow V'(i_v) + H_v(i_v, j_v, k_v)V(j_v)V(k_v) \quad (3\text{-}2)$$

만약 그림 3-4에서 i_v와 j_v가 서로 바뀌게 되면, 수식은 다음과 같이 바뀐다:

$$V'(j_v) \leftarrow V'(j_v) + H_v(j_v, i_v, k_v)V(i_v)V(k_v) \quad (3\text{-}3)$$

Affinity tensor는 supersymmetric하므로, $H_v(j_v, i_v, k_v)$는 $H_v(i_v, j_v, k_v)$와 같으며, 식 (3-3)은 그림 2-4 알고리즘의 line 8의 수식과 일치한다. 이것은 그림 3-3 (a)의 Mult.2와 그림 3-3 (b)의 Mult.1이 같은 동작 임을 의미한다. 그러므로, 그림 3-3 (a)의 Mult.2는 그림 3-3 (b)의 Mult.1으로 교체될 수 있다. 그림 3-3 (a)의 Mult.2 대신에 그림 3-3 (b)의 Mult.1을 사용하는 장점은 그림 3-3 (b)의 Mult.1에서는 write collision이 발생하지 않는다는 것이다. 유사한 방식으로, 그림 3-3 (a)의 Mult.3은 그림 3-3 (c)의 Mult.1으로 교체될 수 있으며, 결과적으로 write collision이 제거된다. 요약하면, 그림 3-3 (a), (b), (c)에서 Mult.1 과정만 수행함으로써, write collision을 제거할 수 있다. 그림 3-5에 이 새로운 algorithm을 제시하였다.

…
6: **if** $i_v < j_v < k_v$
7: $V'(i_v) \leftarrow V'(i_v) + H_v(i_v, j_v, k_v)V'(j_v)V'(k_v)$
8: **end**
9: **if** $j_v < i_v < k_v$
10: $V'(i_v) \leftarrow V'(i_v) + H_v(i_v, j_v, k_v)V'(j_v)V'(k_v)$
11: **end**
12: **if** $j_v < k_v < i_v$
13: $V'(i_v) \leftarrow V'(i_v) + H_v(i_v, j_v, k_v)V'(j_v)V'(k_v)$
14: **end**
…

그림 3-5 Write collision을 제거한 power iteration algorithm

그림 3-5의 알고리즘에서 line 7, 10, 13의 세 수식은 같다. 그러므로, line 6부터 14는 다음과 같이 하나의 조건으로 합쳐질 수 있다.

$$\text{if}((i_v < j_v < k_v)||(j_v < i_v < k_v)||(j_v < k_v < i_v))$$
$$V'(i_v) \leftarrow V'(i_v) + H_v(i_v, j_v, k_v)V(j_v)V(k_v) \quad (3\text{-}4)$$

i_v, j_v, k_v이 분명하지 않을 때에 affinity tensor의 element 값이 0이 된다는 사실을 이용하여, 식 (3-4)의 조건은 아래와 같이 간단해진다.

$$\text{if}\,(j_v < k_v)$$
$$V'(i_v) \leftarrow V'(i_v) + H_v(i_v, j_v, k_v)V(j_v)V(k_v) \quad (3\text{-}5)$$

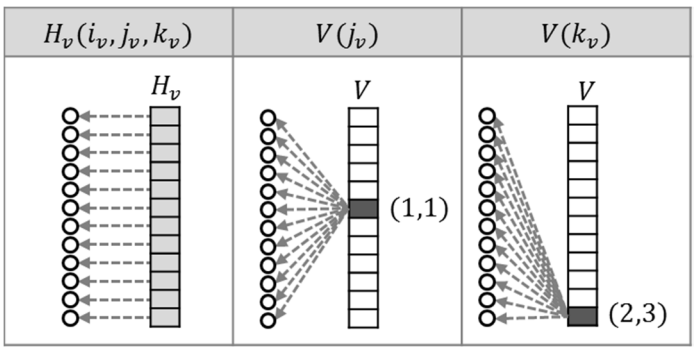
그러므로, power iteration algorithm의 최종 version은 그림 3-6과 같이 제시된다. 그림 3-6에서 line 6의 조건은 그림 2-4의 조건과 비교하여 relaxed되었다. 그러므로, 그림 3-6의 조건을 만족하는 수식의 개수는 그림 2-4의 경우보다 약 세 배 많다. 한편, 그림 2-4에서는 세 개의 수식이 수행되는 반면, 그림 3-6에서는 하나의 수식만이 실행된다. 그러므로, 수행되어야 하는 총 수식의 개수는 두 알고리즘에 대하여 거의 같다.

1: Initialize V randomly
2: **Repeat**
3: **for** k_v = 0 to $N_1N_2 - 1$ **do**
4: **for** j_v = 0 to $N_1N_2 - 1$ **do**
5: **for** i_v = 0 to $N_1N_2 - 1$ **do**
6: **if** $j_v < k_v$
7: $V'(i_v) \leftarrow V'(i_v) + H_v(i_v, j_v, k_v)V(j_v)V(k_v)$
8: **end**
9: **end**
10: **end**
11: **end**
12: $V \leftarrow \frac{1}{||V||_2}$
13: **Until** convergence

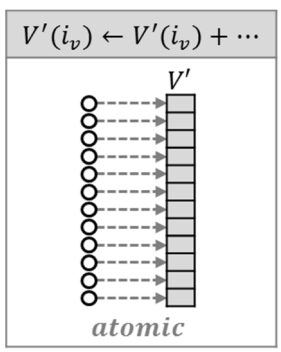
그림 3-6 Power iteration을 위한 제안하는 알고리즘

그림 3-7은 제안하는 tensor-vector multiplication의 block-level memory access pattern을 보여준다. H_v 에 대한 read 동작은 바뀌지 않으며, block 내부의 thread들은 H_v 의 연속적인 주소를 접근한다. V 에 대한 read 동작은 $V(j_v)$ 와

$V(k_v)$에 대해서만 수행되며, block 안의 모든 thread들은 같은 주소에서 data를 가져온다. Write 동작은 $V(i_v)$에 대해서만 수행되며, block 안의 thread들은 연속적인 주소를 접근한다. 이 memory access pattern에서 write collision은 발생하지 않는다.



(a) Read 동작



(b) Write 동작

그림 3-7 제안하는 tensor-vector multiplication의 memory access pattern

3.3 실험 결과

실험환경에서 사용한 CPU는 Intel Core i7-3770 3.40GHz이며, RAM은 16 GB, GPU는 Nvidia GTX 960이다. GPU기반 code는 Nvidia CUDA를 이용하여 구현되었으며, CPU 기반 code는 C++을 이용하여 구현되었다. GPU에서의 처리 시간은 Nvidia Nsight Visual Studio Edition을 이용하여 측정하였다. 실험에서 사용한 affinity tensor의 크기는 64MB이다. 표 3-1은 tensor-vector multiplication 과정의 처리 시간을 보여준다. 첫 번째 열은 algorithm의 종류를 나타내며, 두 번째는 total thread의 개수이다. 세 번째 열은 처리 시간을 나타낸다. 두 번째 행은 그림 3-3의 base algorithm을 나타낸다. 여섯 번째 행은 write collision을 제거한 제안하는 algorithm을 나타낸다. 세 번째 행부터 다섯 번째 행은 base algorithm에서 Mult.1, Mult.2, Mult.3 동작이 각각 수행되었을 때 처리 시간들을 보여준다.

표 3-1 tensor-vector multiplication에서 thread 개수와 처리 시간

Algorithm	#threads	처리 시간(ms)
Base	2,763,520	4.20
Base (only Mult.1)	2,763,520	0.45
Base (only Mult.2)	2,763,520	2.08
Base (only Mult.3)	2,763,520	2.37
제안 방법	8,388,608	1.34

Base algorithm의 tensor-vector multiplication 수행 시간은 4.20ms이다. Base implementation의 Mult. 1 과정에서는 write collision이 발생하지 않기 때문에, processing time이 0.45ms로 매우 적다. 반면, Mult.2와 Mult.3의 처리 시간은 각각 2.08ms와 2.37ms로 Mult.1 과정보다 약 5배 느리다. 이로부터, write collision으로 인한 속도 저하가 약 5배가 된다는 것을 확인할 수 있다. 여섯 번째 row

와 같이 affinity tensor의 supersymmetric 특성을 이용하여 write collision을 제거하였을 때, total thread 개수는 3배 증가한다. 그러나 processing time은 4.20ms 에서 1.34ms로, 3.4배 속도가 향상되었다. 이는 base algorithm의 Mult.1 동작이 3번 수행된 것과 처리 시간이 같다. 제안하는 algorithm의 동작은 base algorithm의 Mult.1 동작과 같기 때문에, 이러한 수치는 예상된 결과이다.

표 3-2는 base algorithm과 제안하는 algorithm의 GPU profiling 결과를 비교한 것이다. 첫 번째 열은 statistics를 나타내며, 두 번째와 세 번째 열은 base algorithm과 제안하는 algorithm에 대하여 측정된 값들이다. 두 algorithm에서 occupancy와 register per thread는 거의 같았다. 두 algorithm 모두 충분한 occupancy를 달성하였으며, 적은 개수의 register만 사용하였다. L2 transaction/request (global memory read)는 global memory read 횟수 대비 실제 transaction 횟수의 비율을 나타낸다. 이 값은 인접한 thread가 인접한 memory 주소를 접근할 때 감소된다. 제안하는 algorithm에서 L2 transaction/request (global memory read)는 2.3에서 2로 약간 감소되었다. Base algorithm에서는 V에 대한 read 동작이 broadcasting 형태뿐만 아니라 연속적인 주소에 대하여 병렬로 접근하는 형태도 수행되며, 이는 broadcasting 형태보다 조금 더 많은 transaction 횟수를 요구한다. 반면, 제안하는 algorithm에서 V에 대한 read 동작은 broadcasting 형태로만 수행된다. L2 transaction/request (atomic)은 write 동작이 atomic 형태로 수행될 때 global memory write 동작의 memory request 대비 실제 transaction 횟수의 비율을 나타낸다. 이 값은 접근하는 memory address가 더 분산되었거나, 여러 개의 thread가 같은 memory 주소에 data를 write 함으로써 thread 동작을 serialization 시킬 때 높아진다. 제안하는 algorithm의 L2 transaction/request (atomic)은 22.7에서 4.0으로 크게 감소되었다. 제안하는 방법에서는 write collision이 제거되었으며, 이는 memory transaction 횟수를 크게 감소시킨다.

표 3-2 Tensor-vector multiplication의 profile 결과

Statistics	Measured results	
	Base algorithm	제안하는 algorithm
Achieved occupancy	84.39%	77.87%
Register per thread	13	10
L2 transaction/request (Global memory read)	2.3	2.0
L2 transaction/request (atomic)	22.7	4.0

표 3-3 Graph matching algorithm의 처리 시간

알고리즘		처리 시간 (ms)			
		Tensor Construction	Each iteration		Total
			Tensor-vector Multiplication	Vector Normalization	
CPU	Base	439	83	1	2,119
	Proposed	1,253	102	1	3,293
GPU	Base	1.4	4.20	0.02	85.8
	Proposed	4.1	1.34	0.02	31.3

표 3-3은 high-order graph matching 전체 수행 시간을 보여준다. 첫 번째열은 algorithm들을 나타내며, 두 번째부터 다섯 번째 열은 high-order graph matching 각 step의 처리 시간을 보여준다. 마지막 열은 high-order graph matching의 전체 처리 시간을 보여준다. Tensor-vector multiplication과 vector normalization의 처리 시간은 power iteration에서 iteration이 한 번 수행될 때의 시간이다. 이 실험에서, power iteration의 iteration 횟수는 20회로 하였으며, 따라서 전체 처리 시간은 각 iteration에서의 처리 시간을 20배한 것에 tensor construction 시간을 더한

값과 거의 같다. 두 번째부터 세번째 행은 CPU에서의 처리 시간이며, 네 번째부터 다섯 번째 행은 GPU에서의 처리 시간을 나타낸다.

CPU에서 basic algorithm의 전체 수행 시간은 2,119ms이며, 이를 단순히 GPU에 implementation하는 것 만으로도 processing time이 85.8ms로 처리 시간이 약 25배 감소하였다. 제안하는 write collision 제거 방법을 적용하였을 때, CPU에서 처리 시간은 2,119ms에서 3,293ms로 증가하였다. 제안하는 방법에서 생성해야 하는 H의 element 개수가 세 배로 증가하였으며, 이는 tensor construction의 수행 시간을 크게 증가시킨다. 또한, tensor-vector multiplication의 수행 시간도 약간 증가하였다. 반면, GPU에서 제안하는 방법은 처리 시간을 85.8ms에서 31.3ms로 크게 감소시킨다. Tensor construction에서의 처리 시간도 약 3배 증가하지만, 전체 수행시간에서 차지하는 비중은 크지 않다. 반면, tensor-vector multiplication에서의 수행 시간은 크게 감소하였다. 제안하는 방법은 GPU hardware의 병렬 처리에 적합한 방법이기 때문에, CPU에서는 속도 향상이 되지 않으며 GPU에서만 속도 향상이 되는 것을 확인할 수 있다. 전반적으로 제안하는 방법은 CPU 에서의 base algorithm보다 68배 빠르며, GPU에서의 base algorithm보다 약 3배 빨랐다.

표 3-4는 high-order graph matching algorithm을 수행할 때, CPU와 GPU에서의 energy 효율을 비교한 것이다. Energy 측정을 위하여 Intel CPU에서는 Inter Power Gadget을 사용하였으며, GPU에서는 NVIDIA Visual Profiler를 사용하였다. 첫 번째 행은 구현들을 나타내며, 두 번째 행은 high-order graph matching algorithm에서 소모되는 평균 power 나타낸다. 세 번째 행은 algorithm이 완료되었을 때, 총 소비 energy를 나타낸다. 전반적으로 CPU 구현의 평균 power 소모량은 GPU 구현의 평균 power 소모량보다 높았으며, 제안하는 GPU 구현의 평균 power 소모량이 가장 높게 나타났다. 이는 제안하는 방법에서 affinity tensor를 접근하기 위하여 많은 양의 read operation이 수행되기 때문이다. 하지만, 제안하는 방법의 energy 소비량은 가장 낮게 나타났다. 이 이유는 제안하

는 방법에 의하여 수행 시간이 크게 감소되었기 때문이다.

표 3-4 energy 효율 비교

구현	CPU	GPU (base)	GPU (proposed)
Average Power (W)	25.7	39.9	50.6
Energy (kJ)	54.47	3.42	1.58

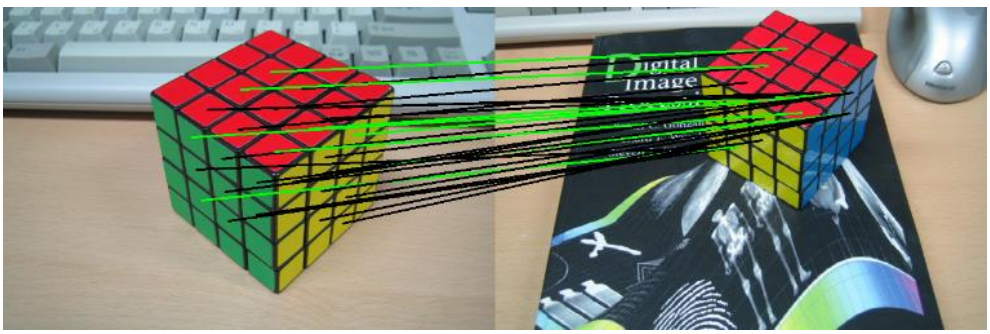
표 3-5는 128개의 matching candidates에 대하여 ANN algorithm과 제안하는 algorithm의 수행 시간을 비교한 것이다. ANN algorithm은 first-order matching에서 사용되며, 제안하는 방법은 high-order matching에 사용된다. 이 실험에서는 first-order matching위해서 SIFT descriptor의 Euclidean distance가 사용되었다. ANN 기반 first-order matching의 처리 속도는 high-order graph matching의 CPU 및 GPU 구현 보다 각각 3,211배, 119배 빠른 것으로 나타났다. 또한, 이 ANN 기반 first-order matching은 제안하는 GPU기반 high-order graph matching 구현 보다도 43배 빠른 처리 속도를 보였다. ANN 기반 first-order matching의 빠른 처리 속도는 algorithm의 time complexity가 $O(n_1 n_2)$이기 때문이다. 반면, high-order graph matching의 time complexity는 $O(n_1^3 n_2^3)$이다.

표 3-5 ANN 기반 first-order matching과 high-order graph matching의 수행 시간

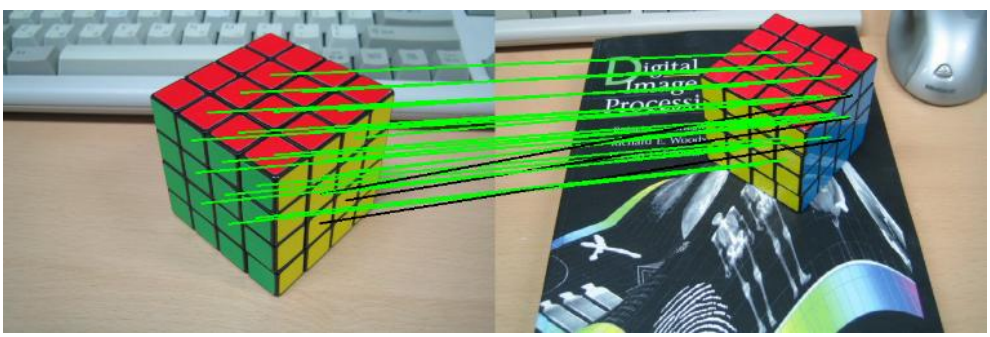
구현	처리 시간 (ms)
ANN을 이용한 first-order matching (CPU)	0.09
High-order graph matching (CPU)	289
High-order graph matching (GPU)	10.7
제안하는 high-order graph matching (GPU)	3.9

그림 3-8은 129개의 matching candidates에 대하여 ANN 기반 first-order

matching과 제안하는 high-order graph matching의 matching 정확도를 비교한 것이다. 녹색과 검은색 선들은 각각 correct match와 false match를 나타낸다. 그림 3-8 (a)는 ANN 기반 first-order matching의 결과를 보여주며, false match의 개수가 true match 개수보다 많은 것을 확인할 수 있다. 그림 3-8 (b)는 high-order graph matching의 결과를 보여주며, 대부분의 match가 correct match이다. 표 3-5와 그림 3-8은 ANN algorithm과 high-order matching에서 complexity와 matching 정확도 사이의 trade-off 관계를 잘 보여준다.



(a) ANN을 사용한 first-order matching 결과 (8 true matches, 21 false matches)



(b) Third-order matching 결과 (22 true matches, 3 false matches)

그림 3-8 ANN 기반 first-order matching과 제안하는 high-order graph matching의 matching 정확도 비교

Write collision 제거는 본 연구의 main contribution 으로, write collision 제거를 위한 다른 GPU 연구와 처리 시간을 비교하였다. Write collision 을 제거하기 위한 일반적인 방법은 shared memory 를 이용하여 reduction 을 수행하는 것이다[26]. 각 step 에서 thread 들은 shared memory 의 두 위치에서 data 를 읽어오고 더한다. 그리고, 더한 data 를 다시 shared memory 에 저장한다. Data 가 shared memory 에 저장될 때 마다 synchronization 동작이 필요하다. 최근에는 warp shuffle 을 이용하여 reduction 동작이 향상되었다. Warp shuffle 은 shared memory 를 이용하지 않고 thread 사이의 data 를 서로 전달 할 수 있게 해준다. 이를 이용하여 reduction 동작 사이의 synchronization 동작이 필요하지 않게 되었으며, 이는 처리 시간을 단축시킨다. 이 기능은 CUDA 3.0 이상을 지원하는 최신 NVIDIA GPU 에서 사용이 가능하다. OpenCL 에서는 OpenCL 2.1 [27] 이상을 지원하는 GPU 에서 사용이 가능하다.

표 3-6 제안하는 방법과 reduction 방법에서 tensor-vector multiplication의 처리 시간

구현	처리 시간 (ms)
제안하는 write collision 제거 방법	396
Shared memory를 이용한 reduction	815
Warp shuffle을 이용한 reduction	467

표 3-6은 제안하는 algorithm과 reduction 동작의 수행 시간을 비교한 것이다. 두 번째 행은 제안하는 algorithm을 나타내며, 세 번째와 네 번째 행은 각각 shared memory와 warp shuffle을 사용한 reduction 동작을 나타낸다. Shared memory를 이용한 reduction 동작은 가장 느린 처리 속도를 보였다. Reduction

동작에서 warp shuffle이 사용되었을 때, 처리 속도는 크게 향상되었다. 그러나, 제안하는 write collision 제거 방법의 처리 시간은 396ms로, warp shuffle을 이용한 reduction 방법 보다도 더 빠르다. 또한, 제안하는 algorithm은 CUDA 3.0이나 OpenCL 2.1 이상을 지원하는 최신 programming 환경을 요구하지 않는다. 따라서, 제안하는 방법은 mobile GPU를 포함한 대부분의 GPU에 적용될 수 있다.

제안하는 algorithm은 병렬화가 잘 되기 때문에, 병렬화를 위한 특별한 technique을 요구하지 않는다. 그림 3-6과 같이, line 7의 수식은 iteration 사이의 data dependence가 없기 때문에, 모든 loop iteration에 대하여 병렬로 수행될 수 있다. 다시 말해서, line 7의 수식이 GPU의 processing element들에 분배될 때, workload가 모든 processing element에 대해서 거의 균일하게 분배될 수 있다는 것이다. Workload balance를 망가뜨릴 수 있는 유일한 요소는 line 6의 if-condition ($j_v < k_v$)이다. 어떤 processing element들이 이 조건을 만족하지 않는다면, 이 조건문은 이러한 processing element들이 동작하지 못하게 한다. 다행히도, 이 if-condition은 가장 안쪽 loop의 iteration variable인 i_v에 대한 dependence가 없다. 대신에, 바깥 loop의 iteration variable인 j_v 와 k_v 에 대해서만 dependence를 갖는다. 제안하는 병렬화 방법에서, i_v는 thread ID를 결정하며, j_v와 k_v는 block ID를 결정한다. 그러므로, $j_v < k_v$ 조건은 block들의 수행을 결정한다. 그러나, GPU에서 streaming multiprocessor (SM)에 대한 block 할당은 dynamic하게 scheduling된다. 이 예제에서, dynamic scheduler는 $j_v < k_v$ 조건을 만족하는 block들만 선택해서 가능한 SM들에 할당한다. 그러므로, 모든 SM들은 효율적으로 utilization되며, workload는 dynamic scheduler에 의해서 균일 해진다.

이러한 workload 균일성은 실험으로부터 관찰된다. 그림 3-9는 제안하는 algorithm이 수행될 때, 각 SM이 active 상태인 비율(activity)을 보여준다. 이 data는 GPU profiler로부터 얻어진 것이다. 이 graph에서 보여지는 것처럼, SM 사이의 workload balance는 거의 완벽하게 달성되었다. 이 결과는 제안하는

algorithm이 workload balance를 위한 특별한 병렬화 technique을 요구하지 않으며 효율적으로 처리된다는 것을 보여준다.

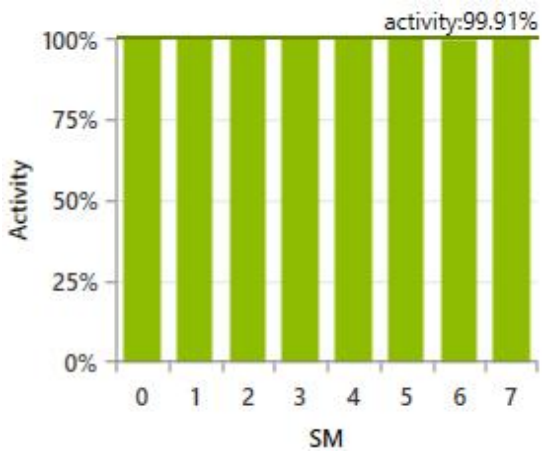
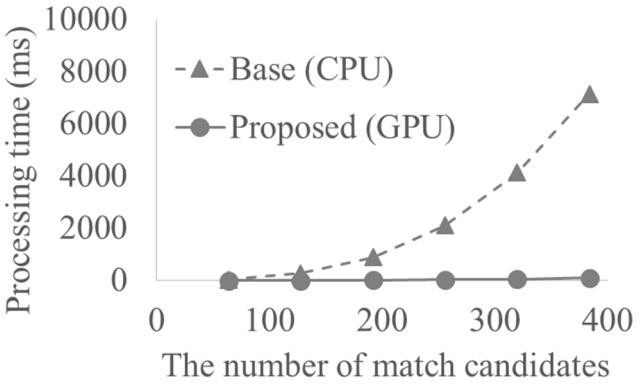


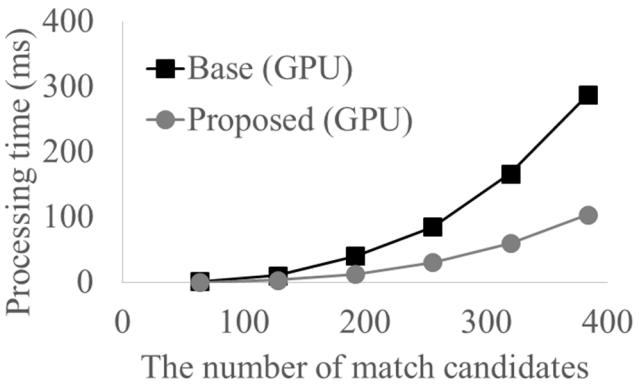
그림 3-9 tensor-vector multiplication를 수행할 때, 각 SM이 active 상태인 비율

그림 3-10은 graph matching에 사용된 feature 개수에 따른 처리 속도를 보여준다. 이 실험에서는 matching candidate의 개수를 변경시켜가며 처리 시간을 측정하였다. 그림 3-8 (a)는 CPU에서 동작하는 base algorithm과 GPU에서 동작하는 proposed algorithm을 비교하였다. 그림 3-8 (b)는 GPU에서 동작하는 base algorithm과 proposed algorithm을 비교하였다. 그림에서와 같이 CPU에서의 처리 속도는 GPU에서보다 매우 느린 것으로 나타났다. Matching candidate의 개수가 200개 일 때, CPU에서의 처리 시간은 1초를 초과하였다. Matching candidate의 개수가 400개 일 때에는 CPU에서의 처리 시간이 거의 8초였다. 반면, GPU에서 동작하는 알고리즘들의 처리 시간은 matching candidate의 개수가 400개 이하일 때 300ms를 넘지 않았다. Matching candidate의 개수가 400개 일 때,

proposed algorithm의 처리 시간은 104ms로, 거의 10 fps의 속도로 도작한다. 반면, GPU에서 base algorithm의 처리 시간은 290ms로서, 약 3.4 fps의 속도로 동작한다.



(a) CPU에서 base algorithm과 GPU에서 제안하는 algorithm의 처리 시간



(b) GPU에서 base algorithm과 제안하는 algorithm의 처리 시간

그림 3-10 Feature 개수에 따른 처리 시간

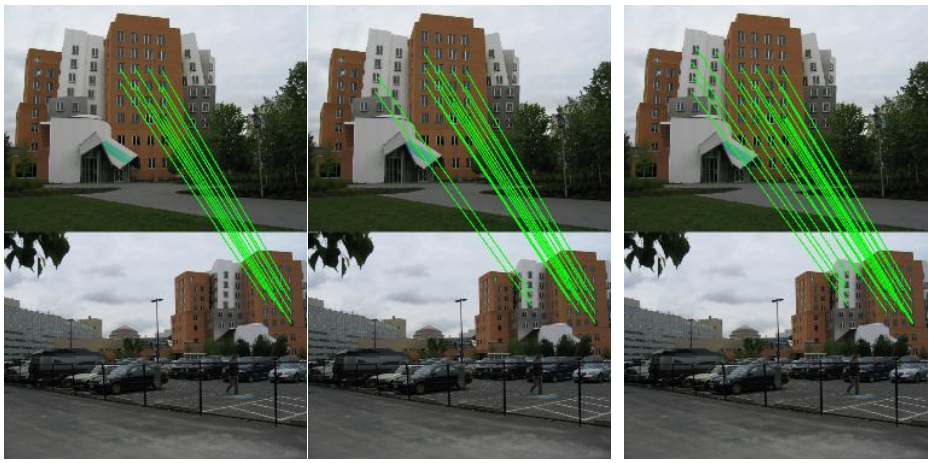
처리 시간은 실시간으로 처리될 수 있는 feature 개수를 결정한다. 일반적으

로, 30 fps의 처리 시간은 실시간 처리를 위한 요구 조건이다. 이러한 조건으로, 그림 3-8의 실험 결과는 제안하는 방법이 GPU에서 260개의 feature를 처리할 수 있음을 보여준다. 반면, base 알고리즘은 GPU와 CPU에서 동작하였을 때 각각 160개와 60개의 feature를 처리할 수 있다. Image matching을 위하여 많은 feature 개수가 사용될수록, matching accuracy는 증가하게 된다. 따라서, 제안하는 방법은 matching accuracy를 크게 향상시킨다. 그림 3-11는 [9]과 [25]에서 사용한 test image를 이용하여 feature 개수에 따른 matching 결과를 보여준다. MSER feature [19]이 각각의 image에서 추출되었으며, SIFT descriptor를 이용하여 matching candidate들을 생성하였다. 그리고, 서로 다른 개수의 matching candidate(64개, 160개, 256개)를 이용하여 high-order graph matching이 수행되었다. 그림 3-9의 결과는 high-order graph matching에 의해 생성된 matching 중에서 RANSAC algorithm[28]을 이용하여 outlier를 제거한 결과이다. 그림 3-11 (a)부터 (c), (d)부터 (f)는 같은 test image를 사용하였다. 그림 3-11 (a)와 (d)는 64개의 matching candidates를 사용할 때의 결과이다. 그림 3-11 (b)와 (e)는 160개의 matching candidates를 그림 3-11 (c)와 (f)는 256개의 matching candidates를 사용할 때 matching 결과이다. 그림에서와 같이 matching feature 개수가 증가할수록 matching accuracy가 증가하는 것을 확인할 수 있다.

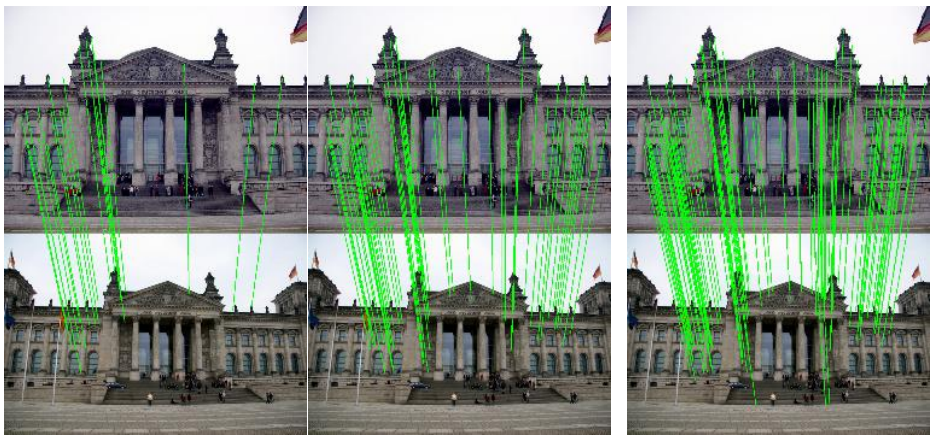
Base algorithm(TM)뿐만 아니라 기존 연구들(RRWHM [9], ASM [10], HGM [12], MCMC [14])의 수행 시간과 matching 정확도를 실험적으로 비교하였다. 모든 구현들은 저자들의 code를 기반으로 하고 있으며, MATLAB code들은 공평한 비교를 위하여 C++ code들로 변경되었다.

표 3-7은 다른 high-order graph matching algorithm들의 처리 시간을 비교한 것이다. 첫 번째부터 세 번째 열은 각각 high-order graph matching algorithm들과 구현 방법, 수행 시간을 나타낸다. 이 실험에서, matching candidate의 수는 256으로 하였다. CPU 구현에서 ASM의 처리 속도가 가장 빠른 것으로 나타났으며, MCMC의 처리 속도가 가장 느린 것으로 나타났다. TM(base algorithm)의

base GPU 구현과 제안하는 GPU 구현 모두 CPU에서 동작하는 algorithm들 보다 빠른 것으로 나타났다.



(a) 64 (b) 164 (c) 256



(d) 64 (e) 164 (f) 256

그림 3-11 다양한 개수의 matching candidate 에 따른 matching 결과

표 3-7 다른 high-order graph matching algorithm들의 수행 시간

Algorithm	구현 방법	처리 시간 (ms)
TM	CPU	2175.3
HGM	CPU	973.7
RRWHM	CPU	1518.7
ASM	CPU	741.1
MCMC	CPU	27642.2
TM	GPU	85.4
Proposed TM	GPU	30.9

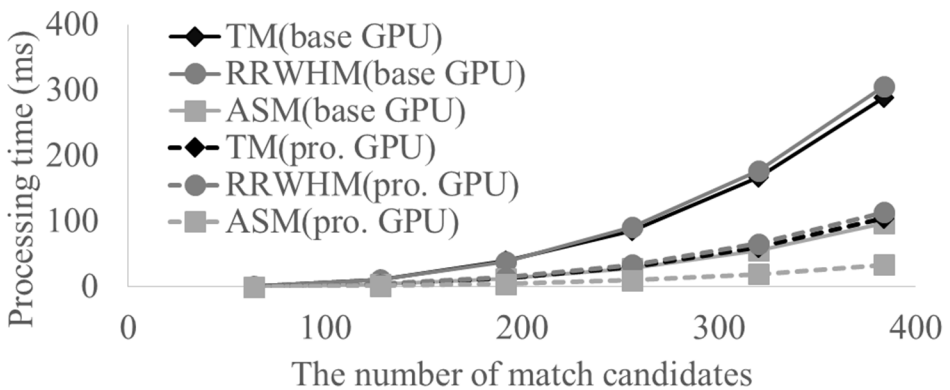


그림 3-12 제안하는 방법이 적용되었을 때 처리 시간

제안하는 algorithm은 supersymmetric tensor를 사용하고 power iteration method를 사용하는 모든 algorithm에 적용될 수 있다. High-order graph matching algorithm에서 사용되는 모든 affinity tensor는 supersymmetric 형태이다. 또한, power iteration method는 RRWHM, ASM, MCMC에 사용될 수 있다. RRWHM의 affinity-

preserving page rank 동작은 IQP 문제의 relaxed version이며, power iteration method를 이용하여 해결될 수 있다. ASM은 feature set 1의 tuple을 approximation하고 power iteration method를 적용하였다. MCMC는 기존의 graph matching algorithm에 Markov Chain Monte Carlo framework를 적용하였다. 다시 말하면, 이 framework를 사용하여 feature set 1에서 active feature들이 선택되며, 이 feature들에 대해서만 기존의 graph matching algorithm이 적용된다. 그러므로, 이러한 graph matching algorithm을 위해서 RRWHM이나 TM이 사용될 수 있으며, 제안하는 방법도 적용될 수 있다. 제안하는 algorithm의 적용 가능성을 보이기 위하여, 다른 high-order graph matching algorithm들을 제안하는 algorithm을 사용하여 GPU로 구현하였다. 그림 3-12는 TM과 다른 algorithm들(RRWHM과 ASM)에 대한 base GPU 구현과 제안하는 GPU 구현의 처리 시간을 보여준다. 가로축은 64부터 384개까지 변하는 matching candidate 개수를 나타내며, 세로축은 수행 시간을 나타낸다. 이 graph에서 실선과 점선은 각각 base 구현과 제안하는 구현을 나타낸다. 3개의 algorithm 중에서, RRWHM과 TM은 유사한 처리 속도를 보였으며, ASM의 처리 속도는 가장 빠른 것으로 나타났다. ASM에 대한 base 구현의 처리 속도는 TM과 RRWHM에 대한 제안하는 구현의 처리 속도와 유사했다. 그러나, ASM에 대한 제안하는 구현의 처리 속도는 ASM에 대한 base GPU 구현보다 약 3배 빠른 것으로 나타났다.

Base algorithm(TM)과 RRWHM, ASM, HGM, MCMC의 matching 정확도를 측정하기 위한 실험을 진행하였다. 그림 3-13은 이 실험에서 사용된 test set을 보여준다. 그림 3-13 (a)는 [9]에서 사용된 test image이며, 그림 3-13 (b)와 (c)는 [25]에서 사용된 test image이다. 여러 개의 matching candidate 수(64, 128, 192, 256개)에 대하여 실험을 진행하였으며, matching 성능을 비교하기 위하여 matching 정확도와 correct match 개수를 측정하였다.

(a) (b) (c)

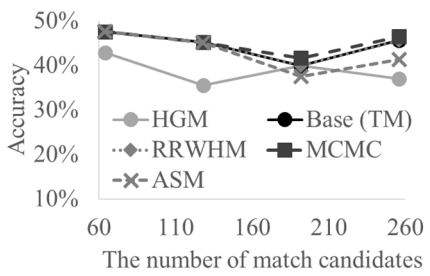
그림 3-13 matching 성능을 측정하기 위한 test image

Matching 정확도는 식 3-6 과 같이 전체 matching 개수 중에서 correct match 개수의 비율로 계산된다.

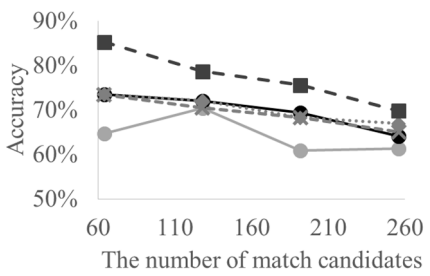
$$\frac{\#Correct\ matches}{\#Correct\ matches+\#false\ matches} \quad \text{(식 3-6)}$$

그림 3-14 (a)-(c)는 각각 그림 3-13 (a)-(c)의 test image에 대한 matching 정확도를 보여준다. 그림 3-14 (a)에서 matching candidate의 개수는 matching 정확도에 큰 영향을 주지 않았다. 그림 3-14 (b)에서는 matching candidate의 개수가 증가함에 따라 matching 정확도가 감소되었다. 그림 3-14 (c)에서는 algorithm마다 matching 정확도 변화가 다르게 나타났다. Matching 정확도에 있어서 이러한 다른 경향성은 matching candidate 개수에 따라 outlier의 개수가 다르기 때문이다. 그림 3-15는 matching candidate 개수가 달라질 때, correct match의 개수를 보여

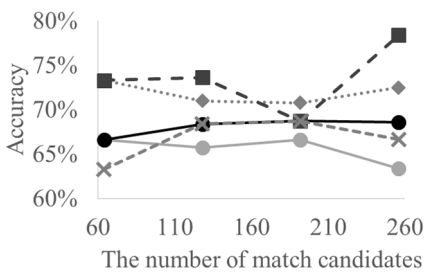
준다. 이 실험에서 다룬 모든 high-order graph matching algorithm에 대하여, matching candidate 개수가 증가함에 따라 correct match의 개수도 증가하였다. 일반적으로, 충분한 개수의 correct match 개수가 존재한다면, RANSAC과 같은 outlier 제거 algorithm을 사용하여 false match는 제거될 수 있다. 그림 3-16은 RANSAC algorithm을 사용하여 outlier가 제거되었을 때, 그림 3-14의 matching 결과를 보여준다. 그림 3-16 (a)-(e)와 (f)-(j)는 각각 64개와 256개의 matching candidate에 대한 matching 결과를 보여준다. 각 algorithm 오른쪽의 숫자는 matching candidate의 개수를 나타낸다. 그림에서 보인 것처럼, matching candidate 개수가 증가할수록, correct match 개수는 증가한다.



(a)

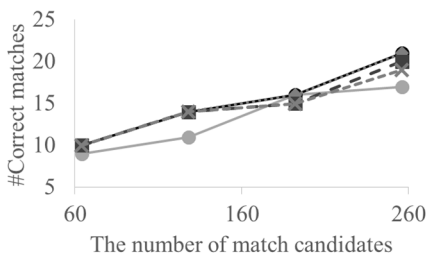


(b)

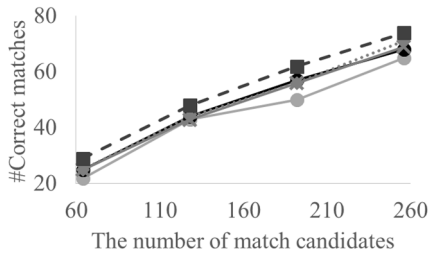


(c)

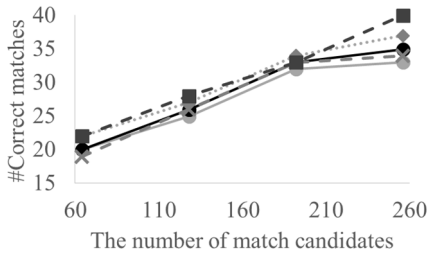
그림 3-14 그림 3-13의 test image에서 matching candidate 개수에 따른 matching 정확도 변화



(a)

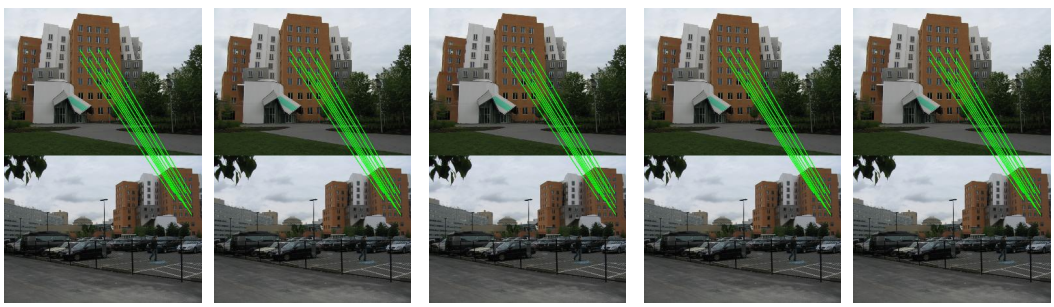


(b)

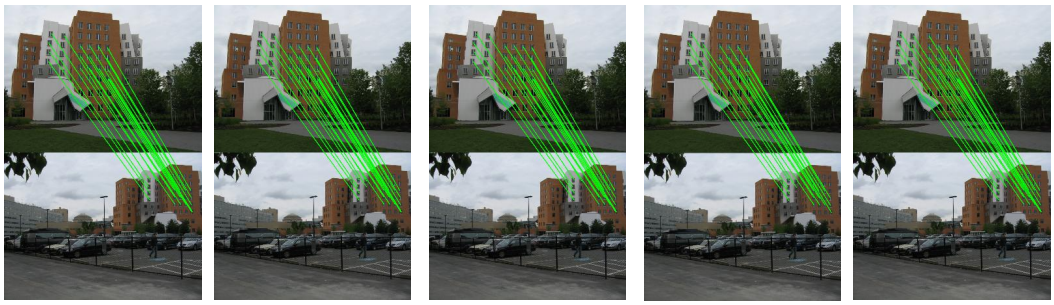


(c)

그림 3-15 그림 3-13의 test image에서 matching candidate 개수에 따른 correct match 개수 변화



(a) TM (b) ASM (c) RRWHM (d) HGM (e) MCMC



(f) TM(256) (g) ASM(256) (h)RRWHM(256) (i) HGM(256) (j) MCMC(256)

그림 3-16 그림 3-13 (a) test image에 대한 matching 결과

다른 architecture(many-core, multi-core, FPGA)에 대한 제안하는 방법의 효율성을 분석하였다. 제안하는 방법은 NVIDIA GPU나 다양한 mobile GPU 같은 SIMD (single Instruction Multiple Data) programming model을 사용하는 many-core architecture를 위하여 설계되었다. 제안하는 방법에서 제거한 write collision은 SIMD에서 빈번하게 발생하는 것이기 때문이다. SIMD program에서 같은 instruction들은 여러 개의 processing core에 의해 동시에 처리된다. 반면, multi-core architecture는 각각의 core가 각각 다른 instruction을 수행하는 MIMD (Multiple Instruction Multiple Data) programming model을 지원한다. 결과적으로, multi-core architecture에서 write collision은 거의 발생하지 않는다. Multi-core 병렬화에서는 제안하는 algorithm이 original graph matching algorithm보다 필연적으로 낫지는 않다는 것을 의미한다. 따라서, multi-core architecture에서는 original algorithm도 효율적으로 사용될 수 있다. FPGA 구현에서 제안하는 algorithm의 효율성은 FPGA에서 processing core들이 어떻게 설계되는가에 따라 달라진다. FPGA는 어떻게 programming되는지에 따라 다양한 종류의 architecture를 지원할 수 있기 때문이다. 만약 FPGA에서 processing core들이 many-core architecture에서의 core들과 유사하게 설계된다면, 이 processing core에서도 write collision

문제가 발생할 것이다. 이때, 제안하는 방법을 통하여 write collision이 제거될 수 있다.

제4장 Approximate Spectral Matching algorithm을 위한 GPU 병렬화

4.1 Approximate Spectral Matching 알고리즘

Feature set 1과 2의 feature 개수가 각각 N_1와 N_2개일 때, 각 feature set에서의 tuple 개수는 각각 N_1^3과 N_2^3개 이다. 이때, affinity tensor의 크기는 두 feature set의 총 similarity 개수로 두 feature set의 tuple 개수의 곱이 된다($N_1^3 N_2^3$). Feature의 개수가 128개일 때, affinity tensor를 저장하기 위하여 필요한 memory 크기는 16TB가 된다. 결국, feature의 개수가 많아지면 affinity tensor를 memory에 저장하는 것 조차도 불가능하게 된다. [10]은 Affinity tensor의 memory 사용량 감소를 위하여 feature set 1의 tuple들을 각도에 따라 approximation하는 approximate spectral matching algorithm을 제안하였다.

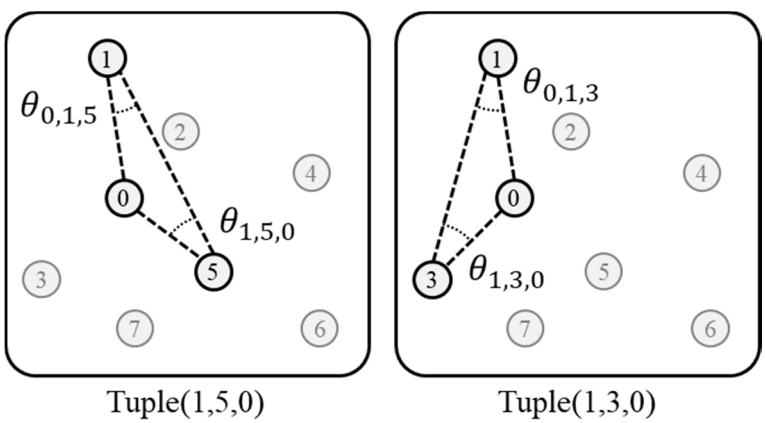
그림 4-1은 tuple이 approximation되는 예를 보여준다. $T(i_1, j_1, k_1)$를 feature set 1에서 feature i_1, j_1, k_1으로 구성되는 tuple이라고 하자. 각 tuple에서 $\theta_{i,j,k}$를 line j-i와 line j-k 사이의 각도를 나타낸다고 할 때, 그림 4-1 (a)는 tuple들의 각도가 10° 단위로 approximation되는 예를 보여준다. $\theta_{k,i,j}$ 와 $\theta_{i,j,k}$가 각각 0°~10°, 0°~10° 범위에 속하는 tuple들은 $\theta_{k,i,j}$ 와 $\theta_{i,j,k}$가 각각 5°인 tuple로 approximation된다. 반면, $\theta_{k,i,j}$ 와 $\theta_{i,j,k}$가 각각 10°~20°, 0°~10°인 tuple들은 $\theta_{k,i,j}$ 와 $\theta_{i,j,k}$가 각각 15°와 5°인 tuple로 approximation된다. 그림 4-1 (b)는 $T(1,5,0)$와 $T(1,3,0)$을 보여준다. $T(1,5,0)$에서 $\theta_{0,1,5}$와 $\theta_{1,5,0}$는 각각 21°와 25°이므로, 모두 25°로 approximation된다. $T(1,3,0)$에서 $\theta_{0,1,3}$와 $\theta_{1,3,0}$는 각각 27°와 29°이므로, 모두 25°로 approximation된다. 따라서, 두 tuple은 모두 같은

approximated tuple (그림 4-1 (a)의 T_a[37])에 속한다고 할 수 있다.

θ_{k_1,i_1,j_1}	θ_{i_1,j_1,k_1}
0 ~ 10	0 ~ 10
10 ~ 20	0 ~ 10
…	…
20 ~ 30	**20 ~ 30**
…	…
0 ~ 10	170 ~ 180

T_a	$\dot{\theta}_{k_1,i_1,j_1}$	$\dot{\theta}_{i_1,j_1,k_1}$
0	5	5
1	15	5
…	…	…
37	**25**	**25**
…	…	…
170	5	175

(a) Tuple approximation



(b) 같은 T_a로 approximation되는 tuple 예

그림 4-1 Approximate Spectral Matching algorithm의 **10°** approximation 예

Approximation에 의하여 affinity tensor에서 중복되는 similarity 값들이 나타나게 된다. 그리고 이러한 원리를 이용하여 affinity tensor의 크기는 감소될 수 있다. 그림 4-2는 affinity tensor에서 같은 approximated tuple들에 속하는 $T(1,3,0)$ 과

$T(1,5,0)$에 대한 similarity 값들을 보여준다. 여기서 두 feature set의 feature의 개수는 8개로 같다. $T'(i_2,j_2,k_2)$를 feature set 2에서 feature i_2, j_2, k_2으로 구성되는 tuple이라고 할 때, $T(i_1,j_1,k_1)$와 $T'(i_2,j_2,k_2)$의 similarity 값은 affinity tensor에서 $H((k_1,k_2),(j_1,j_2),(i_1,i_2))$에 해당한다. 그림에서 H는 3차원 tensor로 표현되었으며, (k_1,k_2), (k_1,k_2), (k_1,k_2)은 $k_1*N_2+k_2$, $j_1*N_2+j_2$, $i_1*N_2+i_2$을 나타낸다. $T(1,3,0)$과 feature set 2의 모든 tuple 사이의 similarity들은 $H((0,k_2),(3,j_2),(1,i_2))$에 저장된다. 그림은 이러한 similarity 값들이 affinity tensor에 저장되는 형태를 보여준다. i_2, i_2, i_2은 각각 0~N_2-1 값을 가진다. 따라서, $H((0,k_2),(3,j_2),(1,i_2))$는 작은 tensor로 표현된다. 그림에서 검은색에 가까울수록 similarity 값이 1에 가까운 것이며, 흰색에 가까울수록 0에 가까운 것이다. $T(1,5,0)$과 feature set 2의 모든 tuple 사이의 similarity들도 $H((0,k_2),(5,j_2),(1,i_2))$에 저장되며, 그림과 같이 작은 tensor로 표현된다. 이때, $T(1,3,0)$ 과 $T(1,5,0)$ 은 approximation에 의하여 같으므로, $H((0,k_2),(3,j_2),(1,i_2))$와 $H((0,k_2),(5,j_2),(1,i_2))$의 모든 element는 같은 값을 갖는다.

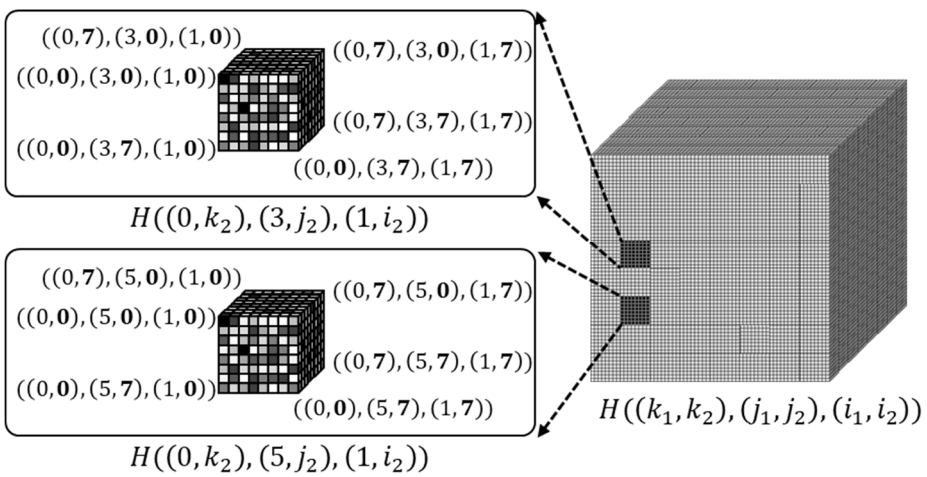


그림 4-2 Affinity tensor에서 같은 $\boldsymbol{T_a}$에 속하는 두 tuple의 similarity

$((0,k_2),(3,j_2),(1,i_2))$와 $H((0,k_2),(5,j_2),(1,i_2))$는 같은 값을 갖는 tensor이므로 이를 중복 저장할 필요가 없다. [10]에서는 이러한 tensor를 한 번만 저장하고 이를 base tensor라고 부른다. 그리고, affinity tensor에서 이 base tensor의 위치를 따로 저장한다. 그림 4-3은 base tensor와 이 base tensor의 위치를 나타내는 index tensor를 보여준다. Base tensor들의 위치는 Index tensor의 (k_1,j_1,i_1)위치에 1 값으로 표현된다. $T(1,3,0)$ 과 $T(1,5,0)$ 에 대한 base tensor의 위치는 index tensor에서 $(0,3,1)$, $(0,5,1)$ 위치에 1 값(검정색으로 표시)으로 저장된다. Index tensor에서 나머지 element들은 0 값(흰색으로 표시)으로 저장된다.

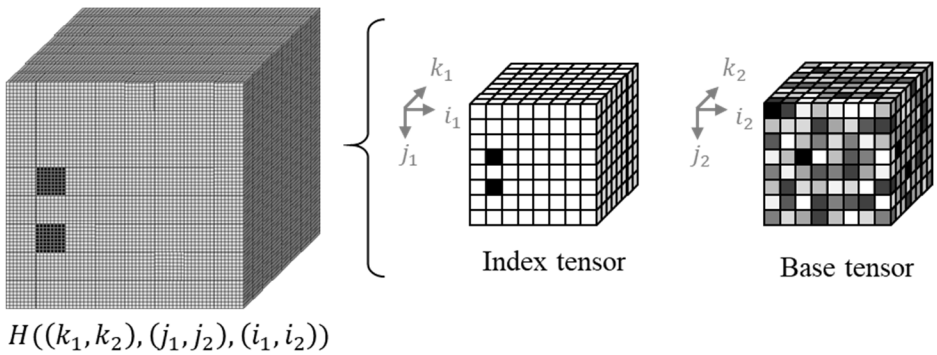


그림 4-3 Index tensor와 base tensor

이러한 index tensor와 base tensor는 approximated tuple($T_a[0]$~$T_a[N_T-1]$)마다 존재할 것이다. Approximated tuple 개수가 N_T개이고, approximated tuple $T_a[t]$마다 존재하는 index tensor와 base tensor를 I_t과 B_t이라고 하자. 그림 4-4는 10° 단위 approximation에서 H 가 index tensor와 base tensor로 decomposition된 형태를 보여준다. Approximation에 의하여 표현될 수 있는 각도의 개수를 N_a라고 할 때, approximated tuple 개수 N_T는 식 (4-1)로 결정된다.

$$N_T = N_a(N_a + 1)/2 \quad (4\text{-}1)$$

10° approximation에서 N_a는 18(180/10)이며, N_T은 171이다. 따라서, index tensor와 base tensor는 각각 171개씩 존재한다. I_t는 N_1-by-N_1-by-N_1 크기의 tensor이며, 그림에서 feature set 1의 feature 개수가 8개이므로, 각 index tensor 는 8-by-8-by-8 tensor 이다. I_t는 base tensor가 존재하는 위치에만 1 값을 가지며, 나머지는 0 값으로 구성되므로 sparse binary tensor라고 할 수 있다. B_t은 N_2-by-N_2-by-N_2 크기의 tensor이며, 그림에서 feature set 2의 feature 개수가 8개이므로, 각 base tensor 는 8-by-8-by-8 tensor 이다. B_t 의 모든 위치에는 tuple들의 similarity 값이 저장 되어있으므로, dense tensor라고 할 수 있다. 그림에서 64 x 64 x 64 크기의 full affinity tensor는 171개의 8 x 8 x 8 크기의 index tensor와 base tensor로 분리되었다. 이렇게 feature set 1의 tuple들을 approximation하게 되면, feature set 1에서 tuple 종류가 N_1^3개에서 approximated tuple 개수(N_T)로 감소된다. 그리고 두 feature set 사이의 총 similarity 개수는 $N_1^3 N_2^3$ 개에서 $N_T N_2^3$ 개로 감소된다. 10° approximation의 경우, 두 feature set 사이의 similarity 개수는 $512 \times 512 = 262{,}144$ 개에서 $171 \times 512 = 87{,}552$ 개로 감소된다. 이에 따라 affinity tensor를 저장하기 위한 memory 사용량도 감소된다.

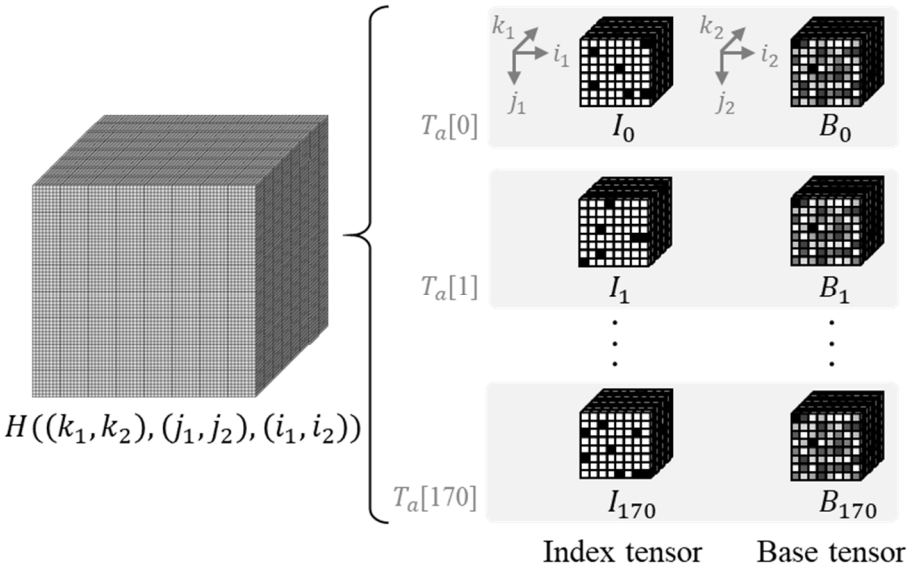


그림 4-4 Base tensor 생성 과정

그림 4-5는 index tensor I_t과 base tensor B_t를 이용한 power iteration algorithm을 보여준다. 식 2-4의 original power iteration algorithm과 비교하여 line 3에서 approximated tuple을 선택하는 loop가 추가되었다. 두 개의 tensor(I_t 과 B_t)와 matrix V의 multiplication 과정이 매 iteration 마다 수행된다.

1: Initialize V randomly

2: **Repeat**

3: **for** $t = 0$ to $N_T - 1$ **do**

4: **for** $k_1 = 0$ to $N_1 - 1$ **do**

5: **for** $j_1 = 0$ to $N_1 - 1$ **do**

6: **for** $i_1 = 0$ to $N_1 - 1$ **do**

7: **for** $k_2 = 0$ to $N_2 - 1$ **do**

8: **for** $j_2 = 0$ to $N_2 - 1$ **do**

9: **for** $i_2 = 0$ to $N_2 - 1$ **do**

10: $V'(i_1, i_2) \leftarrow V'(i_1, i_2) + I_t(k_1, j_1, i_1) B_t(k_2, j_2, i_2) V(j_1, j_2) V(k_1, k_2)$

11: **end**

12: **end**

13: **end**

14: **end**

15: **end**

16: **end**

17: **end**

18: $\mathrm{V} \leftarrow \frac{1}{||\mathrm{V}||_2} \mathrm{V}'$;

19: **Until** convergence

그림 4-5 Index tensor와 base tensor를 이용한 power iteration algorithm

그림 4-6은 그림 4-5의 power iteration algorithm이 GPU에서 처리되는 과정을 보여준다. 그림과 같이 power iteration algorithm은 N_T 개의 독립적인 연산 과정으로 나누어진다. 각 연산 과정에서는 $I_t \otimes B_t \otimes V \otimes V$ 을 계산하며, 결과 값은 V'에 저장된다. 각각의 연산 과정은 서로 독립적이므로, GPU의 streaming multiprocessor (SM)에서 개별적으로 처리될 수 있다. 하나의 SM은 여러 개의

GPU core로 구성된다. 그림에서 GPU core 개수는 128개이며, 이 core들은 각 $I_t \otimes B_t \otimes V \otimes V$ 연산을 병렬로 처리한다. 그러나, I_t 은 sparse tensor이므로, 이는 GPU에서 효율적으로 병렬처리 되기 어렵다. 따라서, sparse tensor를 효율적으로 처리하기 위한 방법이 필요하다.

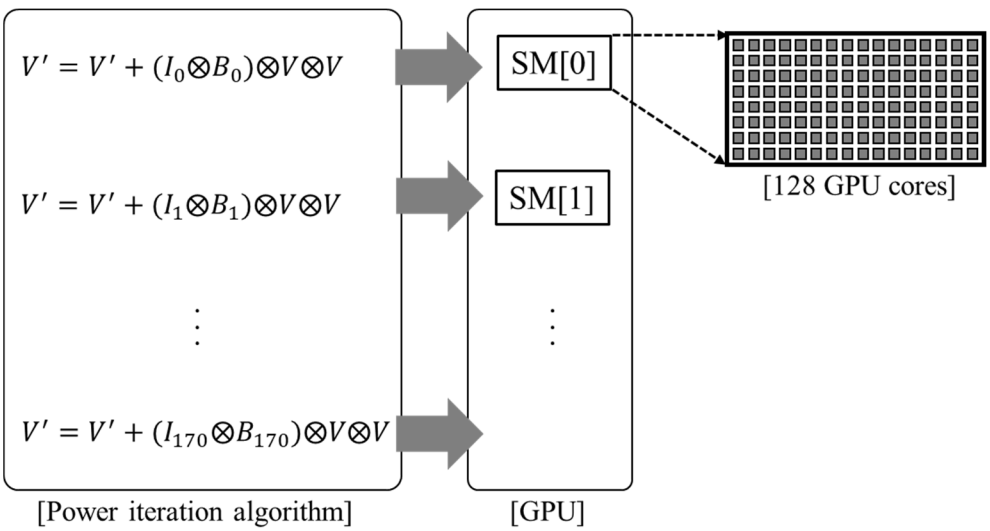
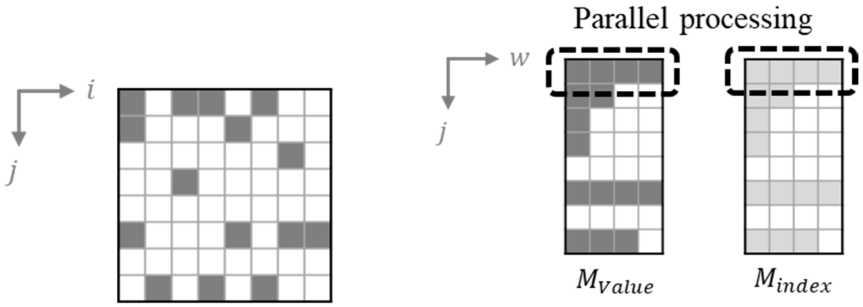


그림 4-6 Index tensor과 base tensor을 이용한 GPU 기반 power iteration algorithm

4.2 Sparse tensor를 위한 기존 연구 적용

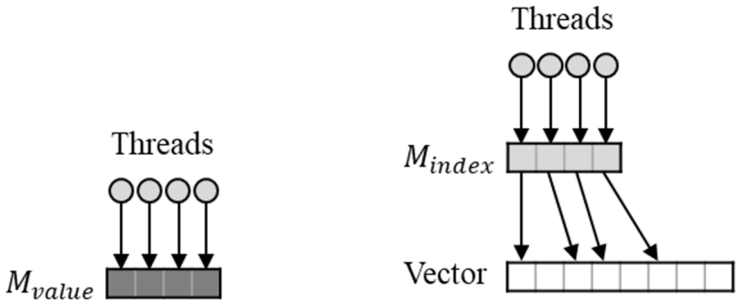
Sparse data를 효율적으로 처리하기 위한 기존 연구는 sparse matrix-vector multiplication에서 많이 수행되었다. Sparse matrix-vector multiplication을 처리하기 위하여 가장 일반적으로 사용되는 방법은 sparse matrix를 GPU 동작에 효율적인 data structure로 변경하는 것이다. 그림 4-7 (a)는 sparse matrix M을 나타내며, 검정색 부분은 non-zero element를, 흰색 부분은 zero element들을 나타낸다. 그림에서 i와 j는 각각 column address와 row address를 나타낸다. 그림 4-7 (b)는 M이 GPU에서 효율적으로 처리되도록 변경한 data structure를 나타낸다. 이 data structure는 두 matrix로 구성된다. 첫 번째 matrix는 M_{value}로, M의 non-zero element들을 j address에 따라 정렬한 것이다. 두 번째 matrix는 M_{index}로, non-zero element들의 i address를 저장하고 있다. 각 matrix의 가로 길이는 M의 non-zero element를 j address에 따라 정렬하였을 때, 가장 많은 non-zero element 개수로 결정된다. 기존 연구에서는 효율적인 parallelization을 위하여 같은 j address를 갖는 M_{value}와 M_{index}의 element들이 병렬로 처리된다.



(a) Sparse matrix M (b) Modified M

그림 4-7 Sparse matrix의 효율적인 병렬화를 위한 data structure

그림 4-8은 같은 j address를 갖는 M_{value}와 M_{index}의 element들을 이용하여 matrix-vector multiplication을 수행할 때, thread들의 memory 접근 pattern을 보여준다. Matrix-vector multiplication을 위해서는 M_{value}와 vector을 각각 read해야 한다. 그림 4-8 (a)는 여러 개의 thread들이 M_{value}의 element들을 read하는 형태를 보여준다. 인접한 thread들은 연속된 주소의 M_{value}에 접근하고 있다. 그림 4-8 (b)는 여러 개의 thread들이 vector element를 접근하는 모습을 보여준다. Thread들이 접근하는 vector의 주소는 M_{index}에 의해 결정된다. 따라서, thread들은 연속된 주소의 M_{index}를 read하고, 이 값을 이용하여 vector에 접근한다. 만약, M_{index}의 element 값들이 큰 차이가 나지 않는다면, coalescing 접근로 vector의 data를 read할 수 있다.

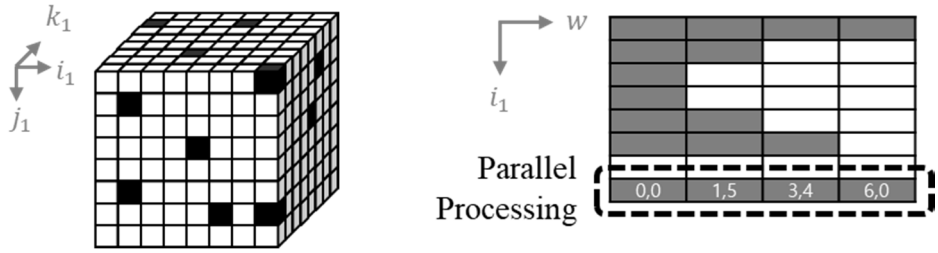


(a) Matrix 접근 (b) Vector 접근

그림 4-8 같은 GPU에서의 sparse matrix-vector multiplication 동작

본 논문에서는 sparse matrix-vector multiplication에서 사용한 기법을 sparse index tensor에 적용하였다. 그림 4-9는 index tensor에 그림 4-7의 방법을 적용한 것이다. 그림 4-9 (a)는 I_0을 나타내며, 흰색 부분은 값이 0인 element를, 검정색 부분은 값이 1인 element를 나타낸다. i_1, j_1, k_1은 각각 index tensor의 주소를 가리킨다. 이 tensor의 non-zero 값들은 1로 모두 동일하다. 따라서, I_0의 non-

zero 값은 따로 저장할 필요가 없으며, non-zero element의 address만 효율적으로 저장하면 된다. 그림 4-9 (b)는 I_0에서 non-zero element의 address가 GPU에서 효율적으로 처리되도록 변경한 것이다. 여기서 이 data structure를 $I_{D,0}$라고 하였다. I_0의 non-zero element들이 i_1 address에 따라 정렬되었으며, 하나의 element $I_{D,0}(i_1, w)$는 j_1와 k_1 address 값을 가지고 있다. 이 matrix의 가로 길이는 I_0의 non-zero element를 i address에 따라 정렬하였을 때, 가장 많은 non-zero element 개수로 결정된다. I_0에서 i_1이 7일 때, 1 값을 갖는 4개의 element가 존재하며, 각 element들의 주소는 다음과 같다: (7, 0, 0), (7, 1, 5), (7, 3, 4), (7, 6, 0). 이 같은 i_1 address를 갖는 element는 $I_{D,0}$에서 i_1이 7인 주소에 위치하게 되며, 각 element는 (0,0), (1,5), (3,4), (6,0)의 값을 가지고 있다. 그리고, sparse matrix-vector multiplication의 경우와 같이, 같은 i_1 address를 갖는 element들을 thread들이 병렬로 처리하도록 할 수 있다.



(a) $I_0(k_1, j_1, i_1)$ (b) $I_{D,0}(i_1, w)$

그림 4-9 Sparse index tensor에 대한 기존 sparse data 처리 방법 적용

그림 4-5의 power iteration algorithm에서 line 3~17의 tensor-matrix multiplication 과정은 식 (4-2)로 표현될 수 있으며, 이를 4-9 (b)의 data structure를 이용한 tensor-matrix multiplication 과정으로 표현하면, 식 (4-3)이 된다.

$$V' = \sum_{t=0}^{N_T-1} \sum_{k_1=0}^{N_1-1} \sum_{j_1=0}^{N_1-1} \sum_{i_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} \sum_{j_2=0}^{N_2-1} \sum_{i_2=0}^{N_2-1} V'(i_1, i_2) + I_t(k_1, j_1, i_1) B_t(k_2, j_2, i_2) V(j_1, j_2) V(k_1, k_2)$$

(4-2)

$$V' = \sum_{t=0}^{N_T-1} \sum_{i_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} \sum_{j_2=0}^{N_2-1} \sum_{i_2=0}^{N_2-1} \sum_{w=0}^{K-1} V'(i_1, i_2) + B_t(k_2, j_2, i_2)\, V\big(I_{D,t}(i_1, w)_j, j_2\big) V\big(I_{D,t}(i_1, w)_k, k_2\big)$$

(4-3)

식 (4-2)에서 $I_{D,t}$는 그림 4-9 (b)의 data structure를 나타내며, K는 $I_{D,n}$의 가로 길이를 나타낸다. $I_{D,t}(i_1, w)_j$과 $I_{D,t}(i_1, w)_k$는 각각 $I_{D,t}(i_1, w)$의 j_1 index와 k_1 index 값을 나타낸다. 이때, 가장 안쪽 loop variable w는 V와 B_t에 영향을 주지 않으므로, 다음 식으로 변경될 수 있다.

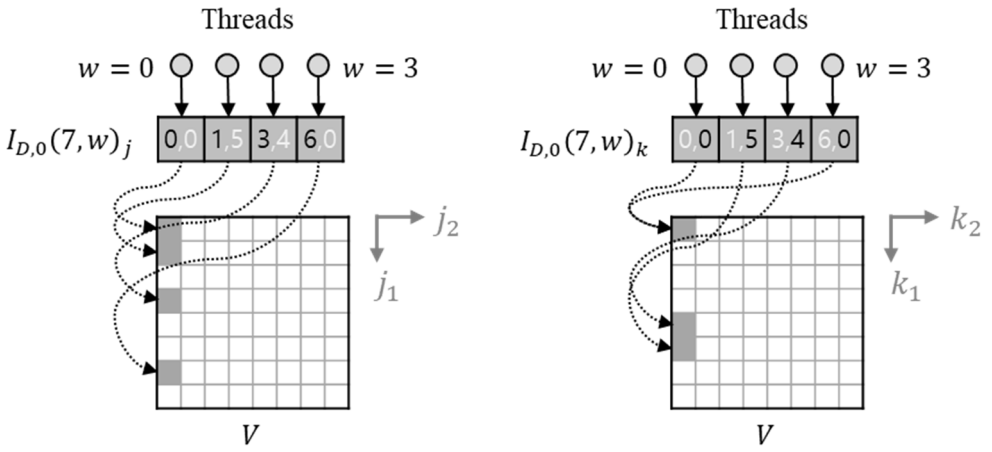
$$V' = \sum_{t=0}^{N_T-1} \sum_{i_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} \sum_{j_2=0}^{N_2-1} \sum_{i_2=0}^{N_2-1} V'(i_1, i_2) + B_t(k_2, j_2, i_2) \sum_{w=0}^{K-1} V\big(I_{D,t}(i_1, w)_j, j_2\big) V\big(I_{D,t}(i_1, w)_k, k_2\big)$$

(4-4)

여기서, j_1와 k_1 index에 의해 접근하는 data structure는 $V\big(I_{D,t}(i_1, w)_j, j_2\big)$와 $V\big(I_{D,t}(i_1, w)_k, k_2\big)$이다. 식 (4-4)는 GPU에서 수행될 때, 두 번의 동작으로 나누어질 수 있다. 첫 번째 동작은 가장 안쪽 loop ($\sum_{w=0}^{K-1}$)를 thread들이 병렬로 처리하는 과정이다. 두 번째 동작은 thread들이 첫 번째 연산 결과와 $B_t(k_2, j_2, i_2)$를 곱하여, 이 결과를 $V'(i_1, i_2)$에 더하는 과정이다. 두 번째 과정에서 인접한 thread들이 인접한 주소의 V와 B_t에 접근하도록, thread id=i_2관계가 되도록 할 수 있다. 이렇게 함으로써, thread들은 서로 다른 i_2 값을 가지며, 인접한 thread들은 인접한 memory 주소($V'(i_1, 0)$~$V'(i_1, N_2 - 1)$, $B_t(k_2, j_2, 0)$~$B_t(k_2, j_2, N_2 - 1)$)에 접근하게 된다. 반면, 첫 번째 연산에서 thread들은 thread id=w관계를 갖는

다. 이때, thread들은 random한 형태로 memory에 접근한다.

그림 4-10은 j_1, k_1 index들($I_{D,t}(i_1,w)_j$와 $I_{D,t}(i_1,w)_k$)을 이용하여 thread들이 V를 접근할 때, memory 접근 pattern을 보여준다. 그림 4-10 (a)에서는 thread들이 j_1 index들을 이용하여 V를 접근하고 있다. V의 세로축과 가로축은 각각 j_1과 j_2 값에 의하여 결정된다. 첫 번째 thread는 V에서 j_1이 0인 주소에 접근하며, 두번째, 세번째, 네번째 thread들은 각각 j_1이 1, 3, 6인 주소에 접근한다. Thread들은 각각 V의 다른 row address에 접근하고 있다. 만약, 인접한 thread들이 memory의 인접한 주소에 접근한다면, 한 번의 transaction으로 data를 가져올 수 있다. 그러나, 그림 4-10 (a)에서 thread들이 접근하는 data들은 인접한 주소에 존재하지 않는다. Thread들이 접근하는 data가 4개이며, 각 data의 주소는 큰 차이가 난다. 따라서, 4개의 thread들이 접근하는 data를 가져오기 위해서는 4번의 transaction이 수행되어야 한다. 그림 4-10 (b)에서도 마찬가지로, thread들이 접근하는 data들의 주소가 큰 차이가 난다. 다만, 여기서는 첫 번째와 네 번째 thread가 접근하는 주소가 같기 때문에, 3번의 transaction으로 4개의 thread가 필요한 data를 가져올 수 있다.



(a) $V\left(I_{D,t}(i_1,w)_j, j_2\right)$에 대한 접근 (b) $V\left(I_{D,t}(i_1,w)_k, k_2\right)$에 대한 접근

그림 4-10 $\boldsymbol{j_1}$, $\boldsymbol{k_1}$ index들을 이용한 $\boldsymbol{V}$ 접근 pattern

4.3 제안하는 sparse index tensor를 처리 방법

그림 4-10의 memory 접근 pattern을 발생시키지 않는 방법은 이 연산을 수행하지 않는 것이다. 그러나, 식 (4-3)에서 $\sum_{w=0}^{K-1} V\left(I_{D,t}(i_1,w)_j, j_2\right) V\left(I_{D,t}(i_1,w)_k, k_2\right)$을 먼저 연산해야지만, $B_t(k_2, j_2, i_2)$과의 multiplication 과정을 수행할 수 있다. 따라서, 제안하는 sparse index 병렬처리 방법은 가장 안쪽 loop ($\sum_{w=0}^{K-1}$)에 대한 연산 과정이 미리 계산되어 있다고 가정하였다. 본 논문에서 이 data structure를 U_{t,i_1}라고 하였다. 식 (4-5)는 $\sum_{w=0}^{K-1} V\left(I_{D,t}(i_1,w)_j, j_2\right) V\left(I_{D,t}(i_1,w)_k, k_2\right)$을 U_{t,i_1}로 변환한 것이다.

$$V' = \sum_{t=0}^{N_T-1} \sum_{i_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} \sum_{j_2=0}^{N_2-1} \sum_{i_2=0}^{N_2-1} V'(i_1, i_2) + B_t(k_2, j_2, i_2) U_{t,i_1}(j_2, k_2) \tag{4-5}$$

U_{t,i_1}는 GPU에서 효율적으로 접근되어야 하므로, j_1과 k_1에 의한 영향이 없어야 하며, t, i_2, j_2와 k_2값이 바뀔 때 마다 변경되어야 한다. 여기서, base tensor B_t은 i_2, j_2, k_2에 영향을 받으므로, U_{t,i_1}도 j_2, k_2만 영향을 받는 $N_T \times n_1$개의 matrix로 표현하였다. 그림 4-11은 U_{t,i_1}를 보여준다. 이 U_{t,i_1}는 그림과 같이 N_2-by-N_2의 matrix로 표현된다. 가로 방향은 j_2 address를 나타내며, 세로 방향은 k_2 address를 나타낸다. 각 element들은 여러 개의 j_1, k_1 index에 대한 $\sum_{w=0}^{K-1} V\left(I_{D,t}(i_1,w)_j, j_2\right) V\left(I_{D,t}(i_1,w)_k, k_2\right)$ 결과를 저장하고 있다.

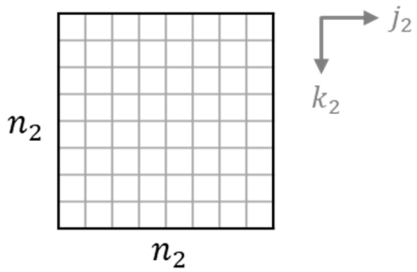
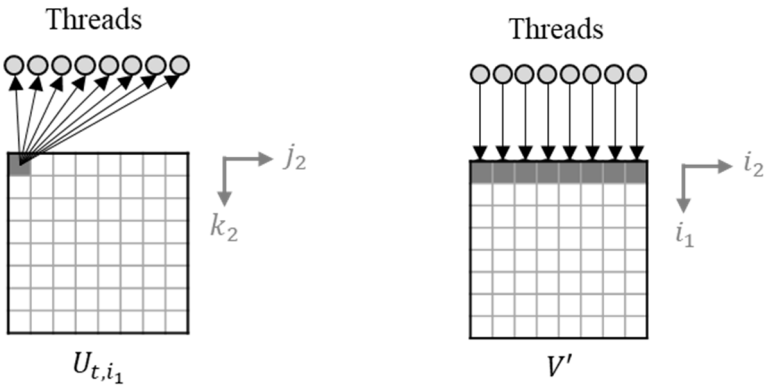


그림 4-11 U_{t,i_1}

U_{t,i_1}를 사용함으로써, (4-5)의 연산 과정은 GPU 구조에 적합하게 된다. 그림 4-12은 (4-5)에 대한 thread들의 U_{t,i_1}와 V에 대한 접근 pattern을 보여준다. 8개의 thread들은 $i_2 = 0$부터 $i_2 = 7$까지의 값을 가지며, 같은 j_2와 k_2 값을 가진다. 그림에서 j_2 와 k_2 은 모두 0 값이다. 그림 4-12 (a)는 (4-5)에서 thread들이 $U_{n,i_1}(k_2, j_2)$를 read하는 동작을 나타내며, 그림 4-12 (b)는 $B_t(k_2, j_2, i_2)U_{t,i_1}(j_2, k_2)$ 연산 결과를 thread들이 $V'(i_1, i_2)$ 에 저장하는 동작을 나타낸다. Thread들이 $U_{t,i_1}(k_2, j_2)$를 접근할 때에는 같은 주소의 값을 읽어온다. Thread들이 $V(i_1, i_2)$에 접근할 때에는 인접한 주소에 값을 저장한다. 이 모든 동작들은 GPU 구조에 적합하다.



(a) $U_{t,i_1}(k_2, j_2)$에 대한 접근 (b) $V'(i_1, i_2)$에 대한 접근

그림 4-12 식 (4-2)에 thread들의 memory 접근 pattern

식 (4-6)은 하나의 U_{t,i_1}를 element를 계산하기 위한 식을 나타낸다.

$$U_{t,i_1}(j_2,k_2)=\sum_{w=0}^{K-1}V(I_{D,t}(i_1,w)_j,j_2)V(I_{D,t}(i_1,w)_k,k_2) \qquad (4-6)$$

이를 GPU 에서 효율적으로 처리하는 것이 중요하다. 그림 4-13 은 여러 개의 thread 들이 하나의 U_{t,i_1} element 값을 계산할 때의 memory 접근 pattern 을 나타낸다. 세 개의 memory 점근 pattern 중에서 왼쪽은 U_{t,i_1}에 값을 저장하는 과정을 나타내며, 두 번째와 세 번째는 각각 $V(I_{D,t}(i_1,w)_j,j_2)$, $V(I_{D,t}(i_1,w)_k,k_2)$ 값을 read 하는 과정을 나타낸다. U_{t,i_1} 에 값을 저장하는 과정에서는 thread 들이 하나의 address 에 값을 update 할 때 발생하는 write collision 이 발생한다. 따라서, 이 과정은 한 번의 transaction 으로 수행되지 못하고, 네 번의 transaction 으로 수행된다. $V(I_{D,t}(i_1,w)_j,j_2)$ 와 $V(I_{D,t}(i_1,w)_k,k_2)$ 에 대한 read 과정은 그림 4-10 (a)와 (b)와 같이 각각 4 transaction 과 3 transaction 으로 처리된다. 결국, 하나의 U_{t,i_1} element 생성을 위한 read 와 write 과정 모두 GPU 에 효율적으로 처리되지 못하고 있다.

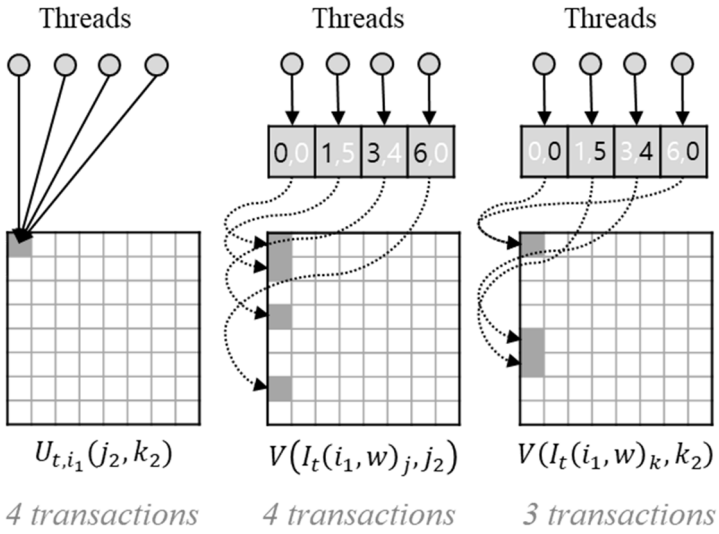


그림 4-13 Thread들이 하나의 $\boldsymbol{U_{n,i_1}}$ element를 연산할 때의 memory 접근 pattern

그림 4-13 에서는 transaction 횟수가 thread 개수에 비례한다. 이때, U_{t,i_1}에서 하나의 element 를 생성하는 과정을 하나의 thread 가 처리해도 transaction 수가 거의 같다. 그림 4-14 는 하나의 U_{t,i_1}를 생성하는 과정을 하나의 thread 가 처리하도록 한 것이다. Thread 아래의 숫자는 thread 가 data 를 접근하는 순서를 나타낸다. 1 번이 먼저 수행되며, 4 번이 가장 나중에 수행된다. 하나의 thread 가 data 를 한 번 접근할 때마다, 한 번의 transaction 이 필요하다. 따라서, U_{t,i_1}에 data 를 write 하는 과정과 V의 element 를 read 하는 과정 모두 각각 4 번의 transaction 으로 수행된다. 이 transaction 횟수는 그림 4-13 에서 thread 들이 V 에서 가져오는 개수와 transaction 횟수가 거의 유사하다. 따라서, 여러 thread 들이 하는 일을 하나의 thread 가 하도록 하더라도, transaction 수는 거의 차이가 없다.

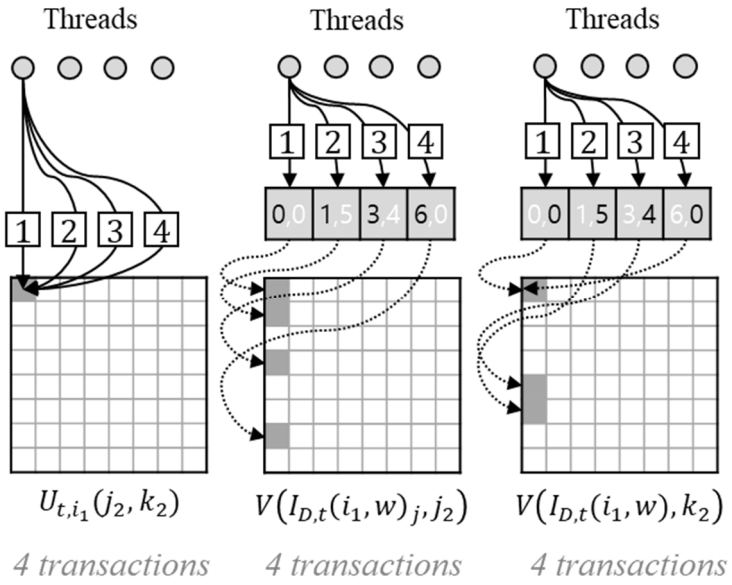


그림 4-14 한 개의 thread가 하나의 $\boldsymbol{U_{t,i_1}}$ element를 연산할 때의 memory 접근 pattern

그림 4-15 는 그림 4-14 에서 첫 번째 순서에 해당하는 부분만 표현한 것이다. 다음 순서에서 접근되는 부분들은 V에서 옅은 회색으로 표시하였다. 이때, 아무 동작도 하고 있지 않은 다른 thread 들이 V에 대한 row address 접근이 아닌 transaction 횟수를 증가시키지 않는 다른 일을 할 수 있다면, U_{t,i_1} 생성 과정의 효율을 증가시킬 수 있다.

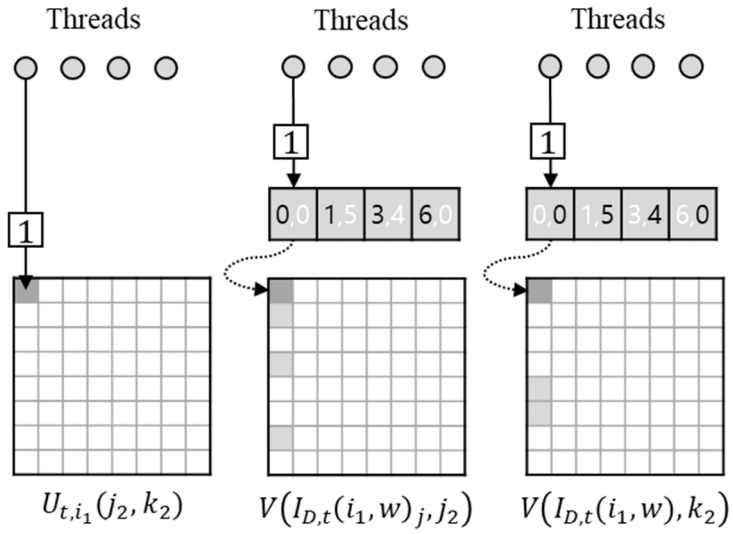
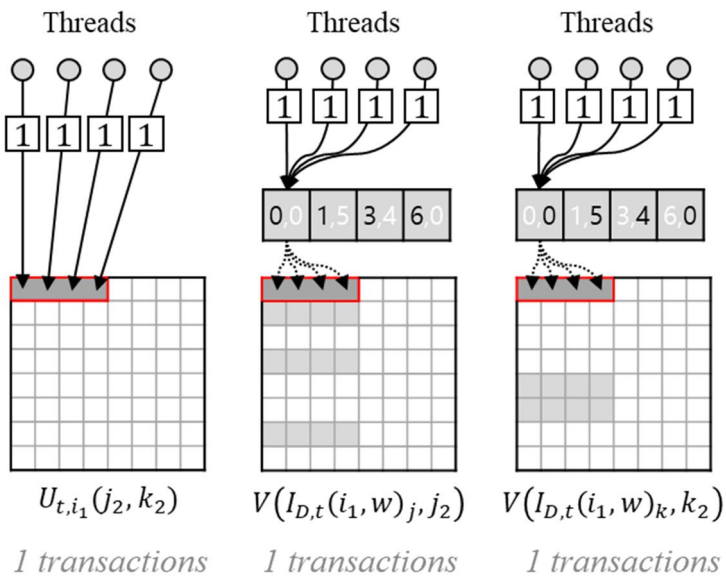


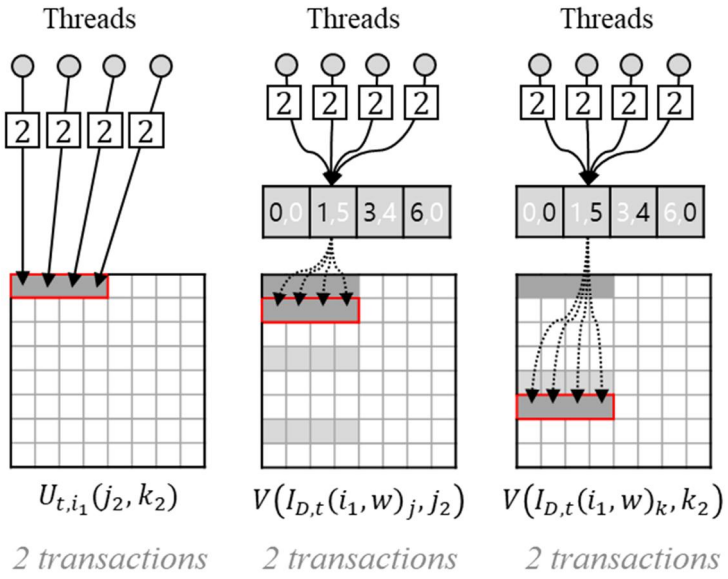
그림 4-15 그림 4-14의 첫 번째 순서

그림 4-16은 제안하는 U_{t,i_1} 생성 방법을 나타낸다. 그림 4-16 (a), (b), (c), (d)는 각각 첫 번째, 두 번째, 세 번째, 네 번째 순서를 나타낸다. 각 순서에서 첫 번째 thread들은 U_{t,i_1}의 첫 번째 element를 생성하고, 나머지 두 번째, 세 번째 네 번째 thread들은 각각 U_{t,i_1}의 두 번째, 세 번째, 네 번째 element를 생성한다. 각 순서에서 thread들이 접근하는 j_1, k_1 index 값을 같다. 이것이 가능한 이유는 모든 U_{t,i_1}를 생성하기 위하여 참조하는 index tensor $I_{D,i}$는 같기 때문이다. 이에 따라, thread들이 V를 read할 때와 U_{t,i_1}에 값을 저장할 때, 인접한 thread가 접근하는 주소가 인접하게 된다. 각 그림의 밑에는 누적 transaction 횟수가 적혀있다. 그림 4-16 (a)의 세 과정은 각각 한 번의 transaction으로 수행된다. 그림 4-16 (b)에서도 마찬가지로 한 번의 transaction으로 data를 read하거나 write 하게 되며, 누적 transaction 횟수는 각각 2가 된다. 각 thread들이 모든 동작을 완료하였을 때, 그림 4-16 (d)와 같이 모든 과정은 4번의 transaction으로 완료된다. 4-13의 과정의 11번의 transaction으로 하나의 U_{t,i_1} element를 생성한 것과 다르게, 제안하는 방법은

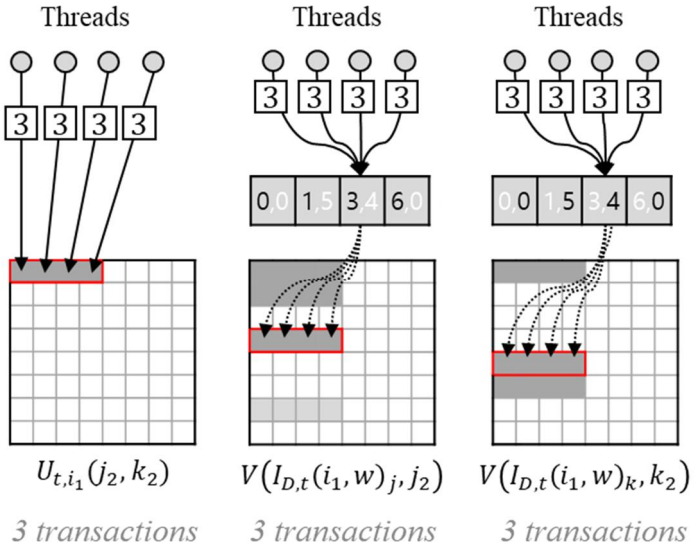
12번의 transaction으로 4 개의 U_{t,i_1} element를 생성하였다. 사용하는 thread의 개수가 많아질수록, 제안하는 방법에 의한 속도 향상이 더 커지게 된다.



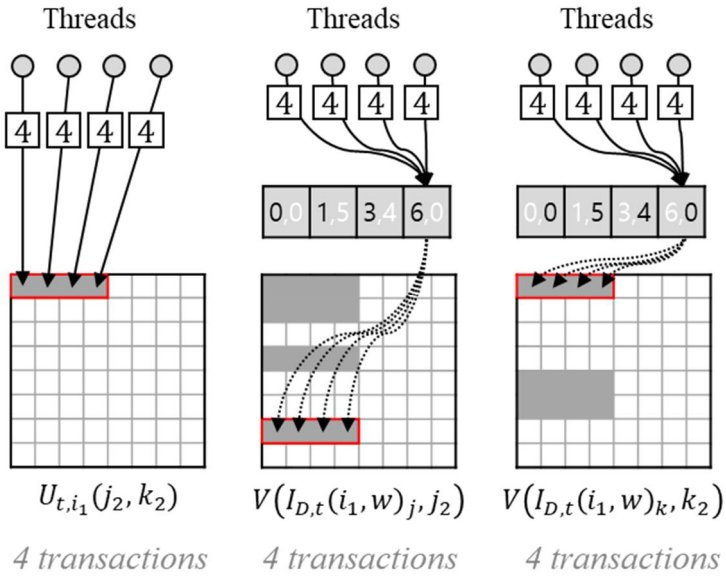
(a) 첫 번째 순서



(b) 두 번째 순서



(c) 세 번째 순서



(d) 네 번째 순서

그림 4-16 제안하는 $\boldsymbol{U_{t,i_1}}$ 생성 과정

4.4 실험 결과

표 4-1과 표 4-2는 GPU에서 original approximate spectral matching algorithm(ASM) [10]과 제안하는 방법이 적용된 ASM에서 tensor-matrix multiplication 과정의 처리 시간을 비교한 것이다. 표 4-1은 10° 도 approximation이 적용되었을 때의 처리 시간이며, 표 4-2는 5° 도 approximation에서의 처리 시간을 나타낸다. 10° 도 approximation에서 approximate tuple의 개수는 666개이며, 이에 따라 index tensor와 base tensor의 개수도 각각 666개이다. 5° 도 approximation에서 approximate tuple 개수는 171개이며, 이에 따라 index tensor와 base tensor의 개수도 각각 171개이다. 각 표에서 첫 번째 행은 feature 개수를 의미한다. 두 번째 행은 sparse index tensor를 GPU에서 효율적으로 처리하기 위한 방법을 적용하지 않았을 때의 처리 시간이다. 세 번째 행은 4.2장의 sparse 처리를 위한 기존 연구를 적용했을 때의 처리 시간이다. 네 번째 행은 제안하는 sparse index tensor 처리 방법을 적용하였을 때의 처리 시간이다.

표 4-1에서 sparse 처리 방법을 적용하지 않았을 때, 16개의 feature를 사용한 tensor-matrix multiplication 처리 시간은 1.2ms로 빠른 처리 시간을 보였다. 하지만, feature의 개수가 64개일 때에는 2,658ms로 2초가 넘는 처리 시간을 보였다. 64개의 feature를 사용할 때, 기존 sparse 처리 방법을 적용하면 약 2배의 처리 속도 향상을 보였다. 그러나, 여전히 1초가 넘는 처리 시간을 보인다. 반면, 제안하는 방법에서는 feature의 개수가 64개일때에 처리 시간이 120ms로, sparse 처리 방법을 적용하지 않았을 때보다 20배 이상, 기존 연구보다 10배 이상 빠른 처리 속도를 보였다. 표 4-2와 같이 5° approximation으로 바꾸었을 때, 제안하는 방법에서만 큰 처리 시간 변화가 있었다. 이는 기존 연구에서는 transaction 횟수가 feature 개수에 영향을 받는 반면, 제안하는 방법에서는 approximation 정도에 영향을 받기 때문이다.

구현 방법	처리 시간 (ms)		
	16 features	32 features	64 features
Sparse 처리 X	1.2	37.2	2,658
기존 연구	1.1	23.4	1,273
제안 방법	0.21	5.3	120

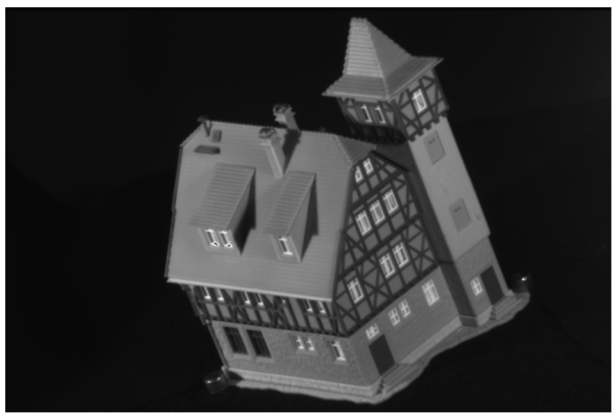
표 4-1 **10**°도 approximation에서 tensor-matrix multiplication 과정의 처리 시간

표 4-2 **5**°도 approximation에서 tensor- matrix multiplication 과정의 처리 시간

구현 방법	처리 시간 (ms)		
	16 features	32 features	64 features
Sparse 처리 X	1.8	34.2	2,996
기존 연구	1.3	24.9	1,006
제안 방법	0.27	8.1	267

그림 4-17 은 approximation 에 따른 matching 정확도 비교를 위하여 사용한 House test set 이다. 이 test set 은 transform 정도에 따라 house 의 기울기가 변하게 된다. 그림 4-17 (a)는 transform 정도가 0 인 reference image 를 나타낸다. 그림 4-17 (b)는 transform 정도가 100 인 test image 를 나타낸다. 이 test set 은 각 image 마다 30 개의 manually selected feature 를 제공한다. 따라서, 제안하는

방법에서는 이 feature 를 이용하여 approximation 정도에 따른 matching 정확도를 측정하였다.



(a) Transform = 0

(b) Transform = 100

그림 4-17 House test set

그림 4-18 은 approximation 에 따른 matching 정확도를 측정한 것이다. 가로축은 그림 4-17 의 House data set 의 transform 정도를 나타내며, 세로축은 correct match 개수를 나타낸다. 100% 정확히 matching 되었을 때 correct match 의 개수는 30 이다. 세 개의 approximation 정도에 대하여 성능을 측정하였다: 2° 단위 approximation, 5° 단위 approximation, 10° 단위 approximation. Feature set 1 의 tuple 을 10° 단위로 approximation 하였을 때, matching 성능은 매우 낮게 나타났다. Transform 정도가 가장 적을 때에도 30 개 중에서 20 개만 matching 되었으며, transform 이 더 커지면 matching 개수가 더 하락하였다. Feature set 1 의 tuple 을 5° 단위로 approximation 하였을 때에는 matching 성능이 크게 향상 되었다. Transform 정도가 작을 때에는 전부 정확하게 matching 되었다. Transform 정도가 가장 높을 때에도 30 개의 matching 중에서 25 개의 correct match 가 나타났다. Feature set 1 의 tuple 을 5° 단위로 approximation 하였을 때에는 더 높은 matching 정확도를 보였다. 2° 단위로 approximation 하였을 때, 정확도는 더 높아졌으나 5° 단위로 approximation 과 큰 차이가 없었다. 만약 제안하는 방법이 5° 단위로 approximation 된 조건에 적용되고 추출된 feature 개수가 64 개라면, 높은 matching 정확도로 sparse 처리를 적용하지 않은 방법보다 약 10 배, 기존 연구보다 4 배 빠르게 동작할 수 있다.

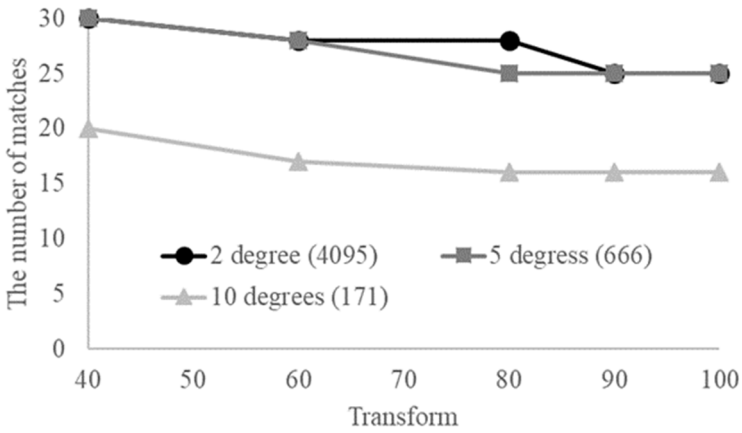


그림 4-18 Transform 정도에 따른 matching accuracy

제5장 Memory 사용량 감소를 위한 approximation기반 base tensor sparsification

5.1 Memory 사용량 감소를 위한 base tensor sampling

높은 matching 정확도를 얻기 위해서는 approximation 정도가 낮아야 한다. 그러나, approximation 정도가 낮으면 base tensor 의 개수가 많아지며, 이는 memory 사용량을 증가시킨다. 그림 5-1 은 approximation 정도에 따른 base tensor 의 memory 사용량을 보여준다. 가로축은 feature 개수이며, 세로축은 memory 사용량을 보여준다. 여기서 두 feature set 의 feature 개수는 같다고 가정하였다. Feature set 1 의 tuple 을 10° 단위로 approximation 하였을 때에는 memory 사용량이 크지 않을 것을 볼 수 있다. 그러나, feature set 1 의 tuple 을 2° 단위로 approximation 하였을 때에는 memory 사용량이 크게 증가하였다. 최신 GPU 의 memory 크기가 10GB 내외인 것을 감안할 때, 2° 단위 approximation 에서 feature 개수가 100 개만 넘더라도, GPU 에서 base tensor 를 저장하는 것은 불가능하게 된다.

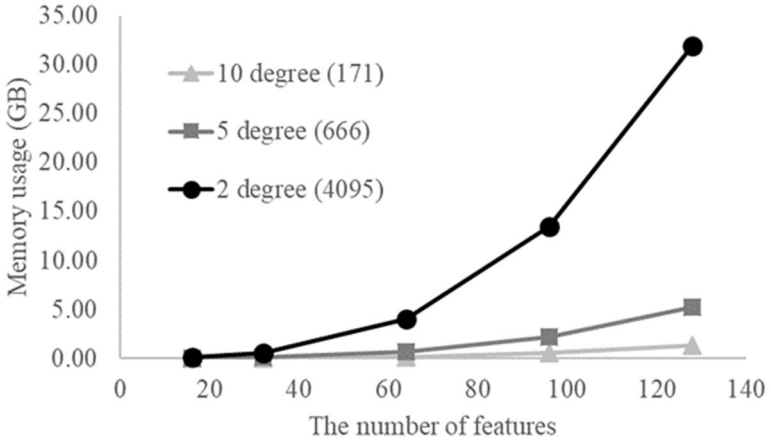


그림 5-1 feature 개수에 따른 memory 사용량

Feature set 1 에 N_1개의 feature 가 존재 할 때, 모든 index tensor 의 non-zero element 개수 총 합은 N_1^3이다. 반면, approximated tuple 개수가 N_T개이고, feature set 2 에 N_2개의 feature 가 존재할 때, 모든 base tensor 의 non-zero element 개수 총 합은 $N_T N_1^3$이다. 따라서, 전체 memory 사용량 중에서 index tensor 의 비중은 크지 않으며, 기존 연구[10]에서는 memory 사용량을 감소시키기 위하여 base tensor 를 sparse 하게 만들었다. 그림 5-2 는 base tensor sparsification 과정을 보여준다. 각 tensor 에서 similarity 값이 클 수록 검정색에 가까운 색으로 표시하였다. Sampling 이전의 base tensor 는 similarity 값이 작은 것부터 큰 것까지 다양한 값이 존재하는 dense tensor 이다. 이때 [10]에서는 base tensor 에서 similarity 값이 큰 element 만 남기고 나머지는 0 값으로 바꾸었다. Sparsification 후 base tensor 는 오른쪽 그림과 같이 similarity 값이 큰 element(검은색에 가까운 element)만 남으며, 0 값(흰색 element)이 많은 부분을 차지한다. 더 많은 element 를 0 으로 바꿀수록, base tensor 의 non-zero element 개수는 더 감소되며, 이는 memory 사용량을 감소시킨다.

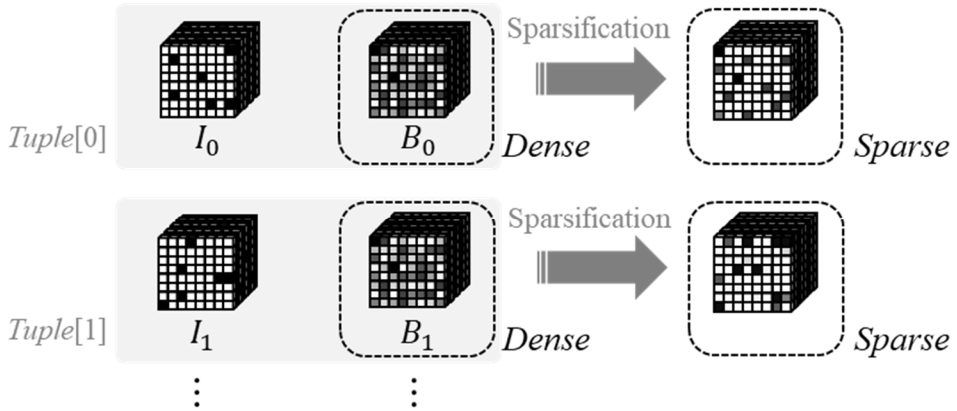


그림 5-2 base tensor memory 사용량 감소를 위한 기존 연구

Sparsification 비율은 matching 정확도를 변경시킬 것이다. Sparsification 정도에 따른 matching 정확도를 측정하기 위한 실험을 진행하였다. 실험을 위하여 사용한 test set 은 House test set 이다. Base tensor 의 sparsification 정도를 바꾸어가며(99%, 98%, 95%), 정확히 matching 된 비율을 측정하였다. 그림 5-3 은 sampling 정도에 따른 matching 정확도를 보여준다. 가로축은 image transform 정도를, 세로축은 correct match 개수를 나타낸다. 그래프에서 점선은 sparsification 을 하지 않았을 때의 matching 정확도를 나타낸다. Base tensor 에서 similarity 값이 상위 1%에 해당하는 element 만 남기고 나머지는 0 으로 하였을 때, matching 정확도 하락이 매우 큰 것을 알 수 있다. Base tensor 에서 similarity 값이 상위 5%에 해당하는 element 만 남기고 나머지를 0 으로 하였을 때, matching 정확도 하락이 나타나지 않았다. 이처럼, 성능 하락을 최소화하기 위한 base tensor sparsification 정도가 제한되어 있음을 알 수 있다. Non-zero element 하나를 저장하기 위해서는 non-zero value 와 non-zero element 의 base tensor 내에서의 주소(i_2, j_2, k_2)를 각각 저장해야 한다. 다시 말하면, non-zero

element 하나를 저장하기 위해서 4 개의 element 를 저장해야 한다. 결국, base tensor 를 95% sparsification 하였을 때, 기존 연구에서 감소될 수 있는 memory 사용량은 20 배가 아닌 5 배 정도이다.

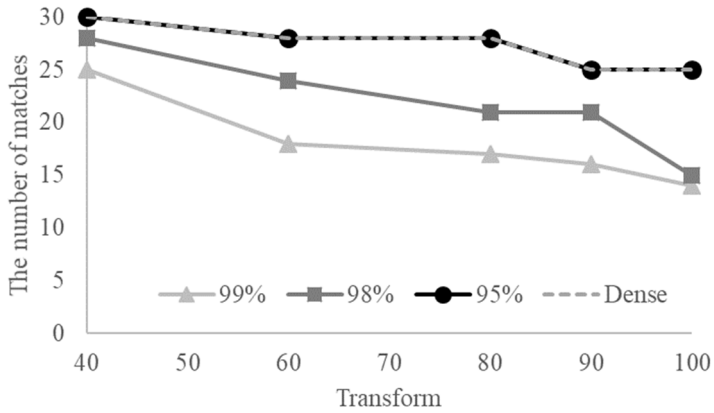


그림 5-3 Sparsification 정도에 따른 matching 개수 변화

5.2 제안하는 base tensor sparsification

Feature set 1에 N_1개의 feature가 있고, feature set 2에 N_2개의 feature가 존재할 때, feature set 1과 2에는 각각 N_1^3와 N_2^3개의 tuple이 존재한다. Affinity tensor의 크기는 두 feature set 사이에서 모든 tuple들의 similarity 개수이며, $N_1^3 N_2^3$이 된다. Approximate Spectral Matching algorithm[10]은 feature set 1의 tuple을 approximation하였다. Approximated tuple의 개수가 $N_{T,1}$이라고 할 때, feature set 1의 tuple 종류는 N_1^3 개에서 $N_{T,1}$ 개로 감소되며, 두 feature set 사이의 총 similarity 개수는 $N_1^3 N_2^3$에서 $N_{T,1} N_2^3$으로 감소된다. 만약, feature set 2의 tuple도 approximation하여 feature set 2의 tuple 개수를 $N_{T,2}$개로 감소시킬 수 있다면, 두 feature set 사이의 총 similarity 개수는 $N_{T,1} N_2^3$에서 $N_{T,1} N_{T,2}$로 감소될 것이다.

그림 5-4는 feature set 2의 tuple을 approximation할 때의 base tensor structure 변화를 보여준다. 왼쪽 그림은 feature set 1의 tuple만 approximation 했을 때, 생성되는 index tensor와 base tensor를 나타낸다. 이는 4-1장에서 설명된 Approximate Spectral Matching algorithm[10]이다. 그림은 feature set 1의 tuple들을 10° 단위로 approximation 시킨 것이며, 이에 따라 총 171개의 index tensor($I_0 \sim I_{170}$)와 base tensor($B_0 \sim B_{170}$)가 존재한다. 여기서 index tensor는 sparse tensor이며, base tensor는 similarity 값을 저장하고 있는 tensor이다. 이 방법에서 총 similarity 개수가 $N_{T,1} N_2^3$인 이유는 N_2-by-N_2-by-N_2 base tensor가 $N_{T,1}$개 존재하기 때문이다. 만약, feature set 2의 tuple들을 $N_{T,2}$개로 approximation 한다면, base tensor의 N_2^3개의 similarity들은 $N_{T,2}$개로 감소될 것이다. 그림은 feature set 2의 tuple들을 10° 단위로 approximation한 것이며, 감소된 B_t의 similarity들을 vector S_t로 표현하였다. 각 S_t은 $N_{T,2} = 171$개의 similarity 값을 가지며, 총 similarity 개수는 모든 S_t의 element 개수가 된다: $N_{T,1} N_{T,2} = 171 \times 171 = 29{,}241$. 그림에서 검은색은 similarity 값이 큰 것을 나타내며, 흰색은 similarity 값이 작은 것을 나타낸다.

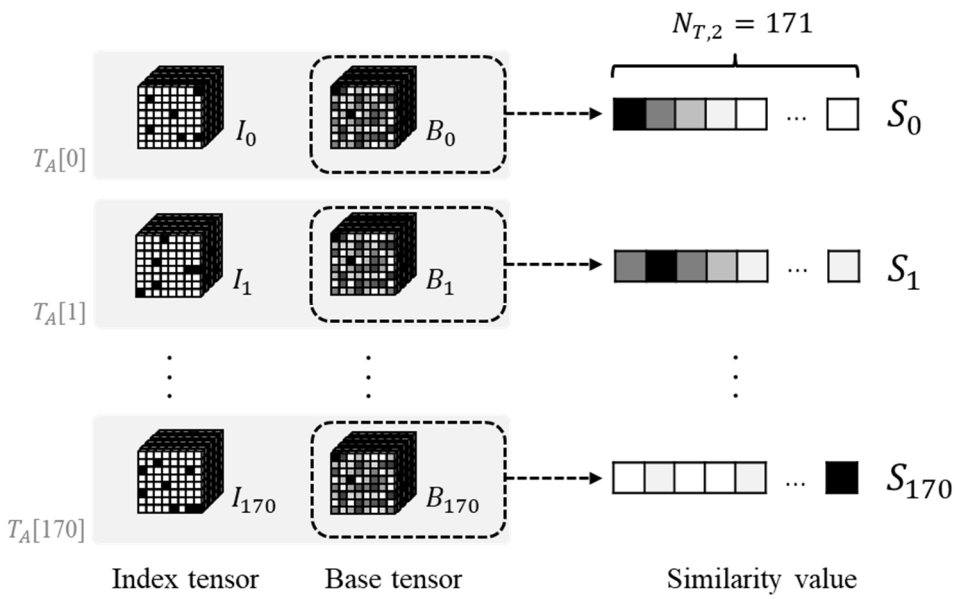


그림 5-4 Feature set 2에 대한 base tensor approximation

이때, feature set 2의 tuple들이 어떤 approximated tuple에 속하는지를 나타내는 query index tensor가 필요하다. 그림 5-5는 B_0가 feature set 2 tuple의 approximation에 의하여 171개의 similarity(S[0][0]~S[0][170])로 나누어 졌을 때, 각 similarity 값들을 approximated tuple($T_{A,2}[0]$~ $T_{A,2}[170]$)에 따라 나열하고, 그에 대한 index tensor($I_{q,(0,0)}$~$I_{q,(0,170)}$)를 보여준 것이다. Approximated tuple마다 하나의 similarity value와 index tensor가 존재한다. 하나의 index tensor는 N_2-by-N_2-by-N_2 tensor이며, $S[t_1][t_2]$에 대한 index tensor는 $I_{q,(t_1,t_2)}$로 표현된다. 만약, feature set 2에서 세 개의 feature i_2, j_2, k_2에 의하여 선택된 tuple $T(i_2, j_2, k_2)$가 $T_{A,2}[t_2]$로 approximation된다면, $I_{q,(t_1,t_2)}(k_2, j_2, i_2)$ 위치에 1 값이 저장된다. 이렇게 feature set 2의 모든 tuple의 approximation 정보가 $I_{q,(0,0)}$부터 $I_{q,(0,170)}$에 저장된다. 이에

따라, 각 B_{t_1}에서 feature set 2에 대한 index tensor의 non-zero element 개수는 N_2^3 개가 된다. Similarity 값을 $N_{T,2}$ 개로 감소시켰더라도, index tensor의 non-zero element가 N_2^3가 되어 base tensor의 memory 사용량을 오히려 증가시키게 되는 것이다.

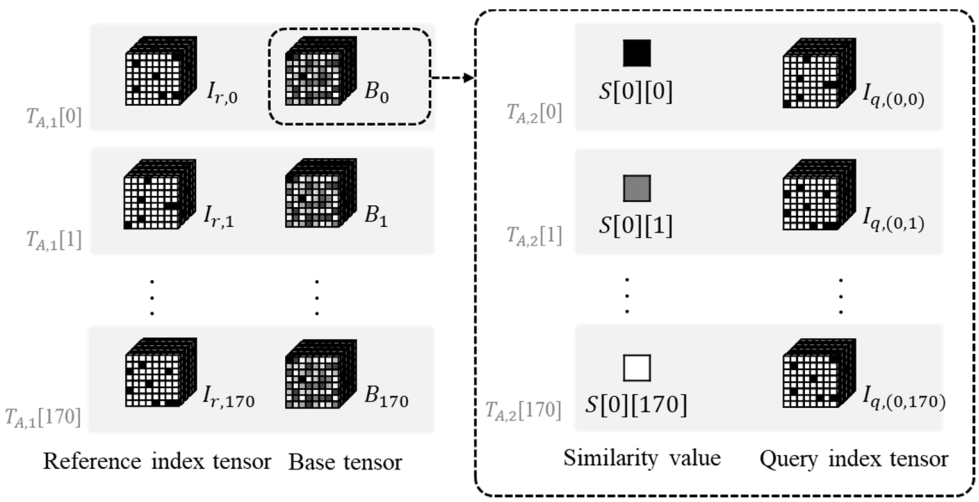


그림 5-5 Base tensor의 분리

다른 base tensor에 대한 query index tensor들을 살펴 볼 때, 서로 같은 query index tensor가 존재한다. 그림 5-6은 B_0와 B_1에 대한 similarity 값들과 query index tensor들을 보여준다. B_0에 대한 similarity 값들(S[0][0], S[0][1], …)은 B_1에 대한 similarity 값들(S[1][0], S[1][1], …)과 다르다. 그러나, B_0에 대한 index tensor와 B_1에 대한 index tensor들은 동일하다. 즉, 다음과 같은 관계를 갖는다: $I_{q,(0,0)} = I_{q,(1,0)}$, $I_{q,(0,1)} = I_{q,(1,1)}$, …. 이렇게 되는 이유는 index tensor들은 feature set 2 tuple들의 approximation 정보를 가지고 있으며, similarity 값들과는 무관하기 때문이다. 따라서, query index tensor들을 중복으로 저장할 필요가 없게 된다.

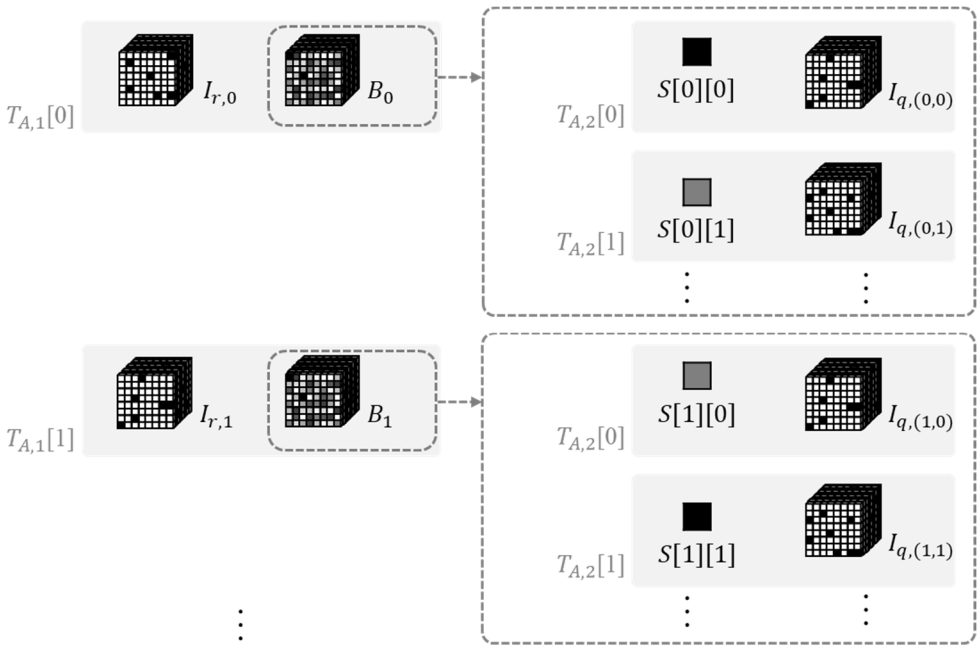


그림 5-6 Base tensor들의 index tensor가 동일

그림 5-5는 feature set 1과 2의 tuple들을 모두 approximation하였을 때, affinity tensor가 분리된 것을 보여준다. Affinity tensor는 크게 reference index tensor와 query index tensor, similarity matrix로 구성된다. 여기서, 두 feature set의 tuple들이 모두 10°단위로 approximation되었다고 가정하였다. 이에 따라, feature set 1에 대한 171개의 reference index tensor($I_{r,0}$~$I_{r,170}$)와 feature set 2에 대한 171개의 query index tensor(171개의 index tensor($I_{q,0}$~$I_{q,170}$)가 존재한다. Similarity matrix는 171-by-171 matrix이다. S[10][20]은 $I_{r,10}$에 속하는 feature set 1의 tuple들과 $I_{q,20}$에 속하는 feature set 2의 tuple들 사이의 similarity 값이며, 모두 동일한 값을 가진다.

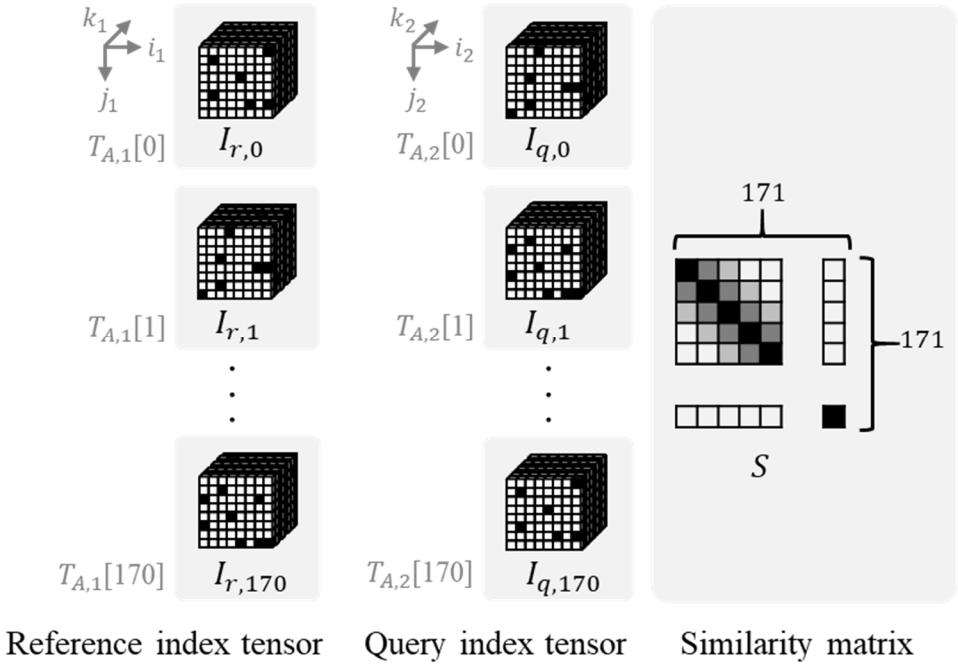


그림 5-7 Base tensor approximation에 의한 affinity tensor 분리

그림 5-6은 index tensor의 non-zero element 개수들을 보여준다. Reference index tensor은 feature set 1으로부터 생성되는 index tensor를 의미하며, query index tensor는 feature set 2로부터 생성되는 index tensor를 나타낸다. Reference index tensor의 전체 non-zero element 개수(1의 개수)는 feature set 1의 tuple 개수이며, N_1^3이다. 마찬가지로, query index tensor의 전체 non-zero element 개수도 N_2^3이다. 두 index tensor의 non-zero element 개수 합은 $N_1^3 + N_2^3$이다. 따라서, 두 index tensor에서 non-zero element만 저장 한다면, memory 사용량도 $N_1^3 + N_2^3$이 될 것이다. Approximation에 따라 각 reference index tensor와 query index tensor의 개수는 변하지만, 두 tensor의 총 non-zero element 개수는 변하지 않는다. 따라서, 제안하는 방법은 approximation에 관계 없이 affinity tensor의 memory 사용량이

동일하다.

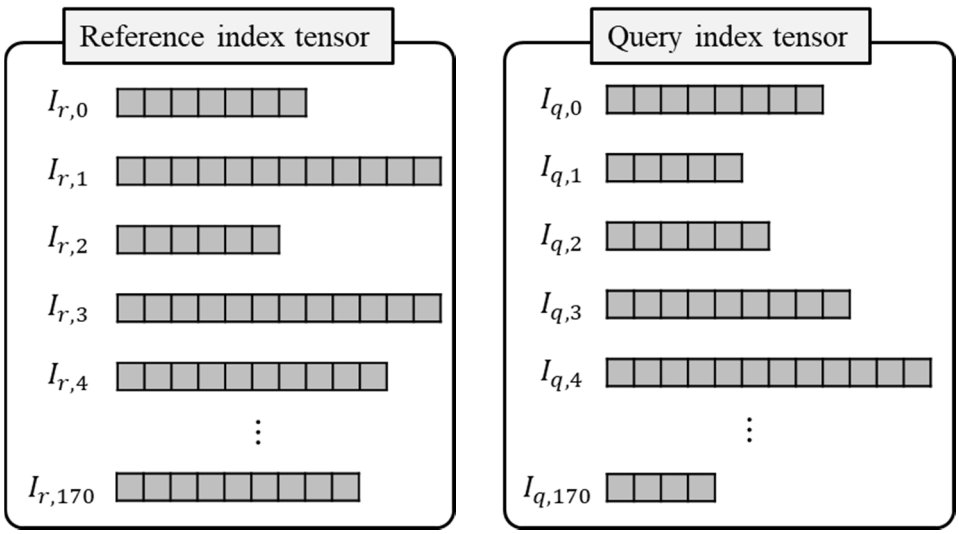


그림 5-8 Index tensor와 index 개수

그림 5-9는 Reference index tensor와 query index tensor, similarity matrix를 이용한 power iteration algorithm을 보여준다. Matrix V의 주소는 reference index tensor($I_{\mathrm{r,t_1}}$)와 query index tensor($I_{\mathrm{q,t_2}}$)에 의해 결정된다. 매 iteration 마다 matrix S와 V의 multiplication 과정이 수행된다.

1: Initialize V randomly

2: **Repeat**

3: **for** $t_1 = 0$ to $N_{T,1} - 1$ **do**

4: **for** $w_1 = 0$ to $K_1 - 1$ **do**

5: **for** $t_2 = 0$ to $N_{T,2} - 1$ **do**

6: **for** $w_2 = 0$ to $K_2 - 1$ **do**

7: $V'\left(I_{r,t_1}(w_1)_i, I_{q,t_2}(w_2)_i\right) \leftarrow V'\left(I_{r,t_1}(w_1)_i, I_{q,t_2}(w_2)_i\right) +$
$S(t_1, t_2)V(I_{r,t_1}(w_1)_j, I_{q,t_2}(w_2)_j)V(I_{r,t_1}(w_1)_k, I_{q,t_2}(w_2)_k)$

8: **end**

9: **end**

10: **end**

11: **end**

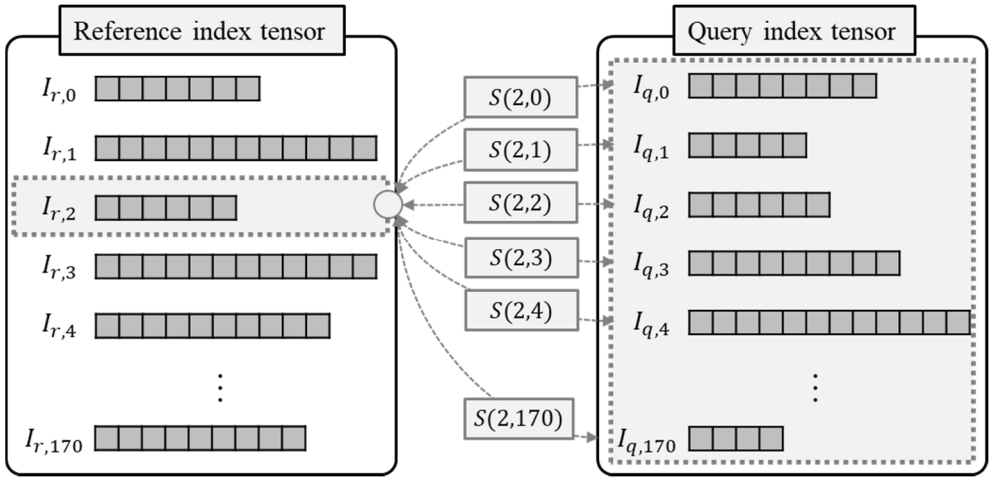
12: $\mathrm{V} \leftarrow \frac{1}{||\mathrm{V}||_2}\mathrm{V}'$;

13: **Until** convergence

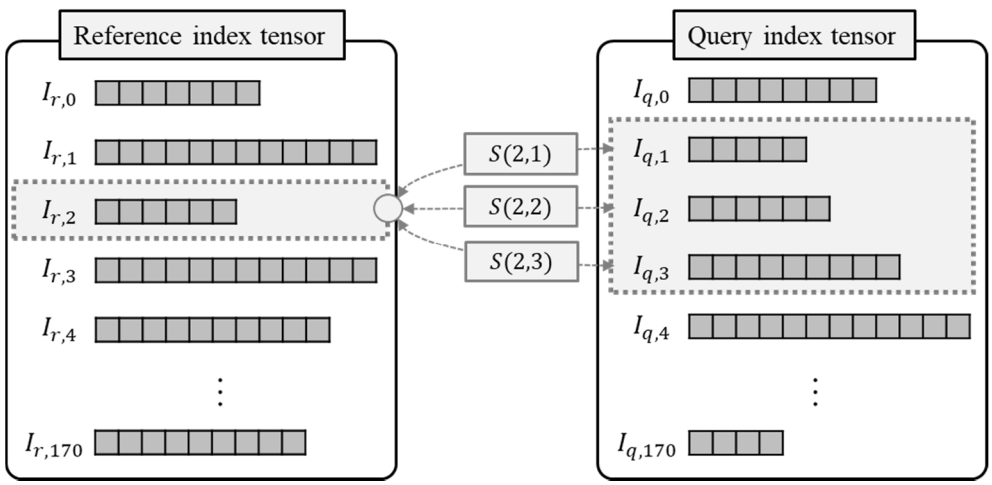
그림 5-9 Reference index tensor와 query index tensor 및 similarity matrix를 이용한 power iteration algorithm

5.3 제안하는 approximation 방법의 GPU 구현

기본적인 GPU 구현은 $N_{T,1}$개의 block을 생성하고, 하나의 block에서 그림 5-9의 line 4~10을 처리하도록 하였다. 그림 5-10는 하나의 block이 read해야하는 index tensor와 similarity tensor를 나타낸다. 그림 5-10 (a)는 $t_1 = 2$일 때, 하나의 block이 read하는 index tensor와 similarity tensor를 나타내다. 하나의 block은 $I_{r,2}$와 $I_{q,0}$~$I_{q,170}$의 index tensor를 read하며, $S(2,0)$~$S(2,170)$의 similarity matrix를 read한다. 그리고 이를 이용하여 그림 5-10 line 7의 연산을 병렬로 처리한다. 이 때, 171개의 similarity 값 중에서 많은 값들은 0에 가까운 값들을 가지며, 일부만 이용해도 matching 정확도 하락을 일으키지 않는다는 것을 그림 5-3에서 보였다. 그림 5-10 (b)는 일부의 similarity만 이용하였을 때, 하나의 block이 read해야하는 index tensor와 similarity matrix를 나타낸다. 3개의 similarity 값만 이용하여 graph matching을 수행한다고 할 때, read해야하는 index tensor는 크게 감소된다. 즉, 하나의 block은 총 4개의 index tensor($I_{r,2}$, $I_{q,1}$, $I_{q,2}$, $I_{q,3}$)와 3개의 similarity value(S(2,1), S(2,2), S(2,3))만 read하여 그림 5-9의 line 4~10을 수행하면 된다. 대신, 이 동작을 위하여 높은 similarity 값을 갖는 element를 matrix S에서 찾아야 한다. 그러나, similarity 개수가 171개일 경우, 171개의 thread들을 생성하여, 원하는 similarity 값들을 한 번에 찾을 수 있다.



(a)



(b)

그림 5-10 각 block에서 graph matching을 수행할 때, 접근하는 index tensor와 similarity matrix

Index tensor는 1과 0으로만 구성된 sparse tensor이다. 따라서, sparse index tensor를 GPU에서 효율적으로 처리하기 위한 방법이 필요하다. Graph matching을 수

행할 때 그림 5-9와 같이, $V'\left(I_{1,t_1}(w_1)_j, I_{2,t_2}(w_2)_j\right)$와 $V'\left(I_{1,t_1}(w_1)_k, I_{2,t_2}(w_2)_k\right)$를 read하여 연산을 수행하고, 연산 결과를 $V'\left(I_{1,t_1}(w_1)_i, I_{2,t_2}(w_2)_i\right)$에 저장한다. 여기서, I_{1,t_1}의 element는 V의 row address를 결정하며, I_{2,t_2}의 element은 V의 column address를 결정한다. 만약 thread들이 I_{1,t_1}의 element들을 병렬로 처리할 경우, V의 row address가 random하게 접근된다. 이는 GPU에서 효율적으로 처리되기 어려운 memory 접근 pattern이다. 반대로, thread들이 I_{2,t_2}의 element들을 병렬로 처리할 경우, V의 column address가 random하게 접근된다. 이는 GPU에서 shared memory를 이용하여 효율적으로 처리될 수 있다. 따라서, GPU에서 효율적인 처리를 위하여 I_{1,t_1}의 element의 element는 순차적으로 처리하고, I_{2,t_2}의 element는 병렬로 처리한다.

그림 5-11은 GPU에서 shared memory를 이용한 graph matching algorithm 처리 방법을 보여준다. 먼저, $I_{r,2}$ 하나의 element를 read하여, i_1, j_1, k_1을 결정한다. $I_{r,2}(w_1)_i$, $I_{r,2}(w_1)_j$, $I_{r,2}(w_1)_k$는 각각 하나의 $I_{r,2}(w_1)$ element의 i_1, j_1, k_1 값을 의미한다. 그림에서, w_1은 0일 때 경우이며, $I_{r,2}(w_1)_i$, $I_{r,2}(w_1)_j$, $I_{r,2}(w_1)_k$는 각각 3, 2, 5 값이다. 이로부터, 접근해야 하는 V의 row address들이 결정된다. 그리고 $V(i_1, 0)$ 부터 $V(i_1, N_2 - 1)$, $V(j_1, 0)$ 부터 $V(j_1, N_2 - 1)$, $V(k_1, 0)$ 부터 $V(k_1, N_2 - 1)$을 shared memory $V_{sh,i}$, $V_{sh,j}$, $V_{sh,k}$에 저장한다. 그림 5-12 (a), (b), (c)는 thread들이 각각 $I_{q,1}$, $I_{q,2}$, $I_{q,3}$을 이용하여 $V_{sh,i}$, $V_{sh,j}$, $V_{sh,k}$에 접근하는 형태를 보여준다. Thread들은 query index tensor의 i_2, j_2, k_2 index들을 각각 read하여 $V_{sh,i}$, $V_{sh,j}$, $V_{sh,k}$에 접근한다. 이때, column address 단위의 random 접근이 발생한다. 그러나, shared memory에서는 global memory와는 다르게, 인접한 thread가 인접한 shared memory의 address에 접근할 필요가 없다. 따라서, query index tensor의 element들에 의해 발생하는 random 접근은 효율적으로 처리된다. 또한, $V_{sh,i}$, $V_{sh,j}$, $V_{sh,k}$는 $I_{q,1}$, $I_{q,2}$, $I_{q,3}$를 처리 할 때, 계속 재사용된다. 따라서, shared memory를 이용함으로써 global memory 접근량도 감소시킬 수 있다. Thread들은

$V_{sh,j}$와 $V_{sh,k}$의 data를 read하여 연산을 수행하고 연산 결과를 $V_{sh,i}$에 저장한다. 최종적으로 $V_{sh,i}$를 $V(i_1, i_2)$에 저장하여 연산을 마무리한다.

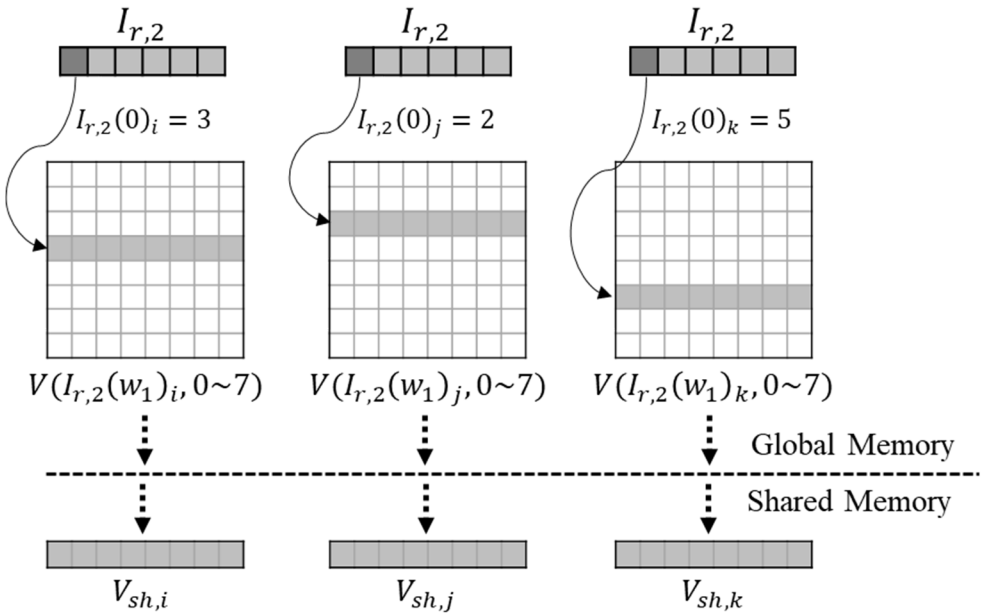
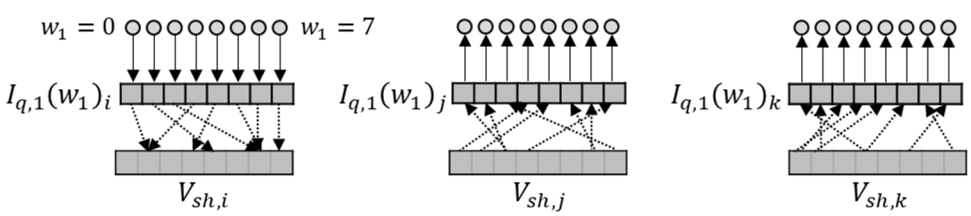
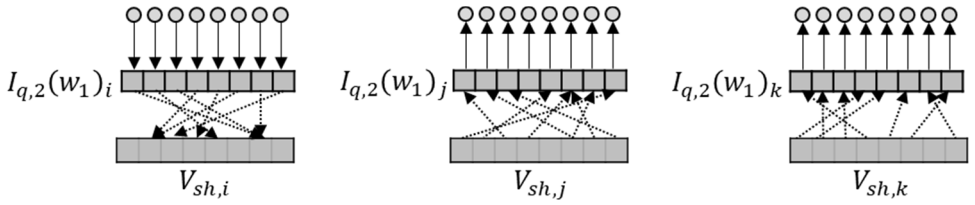


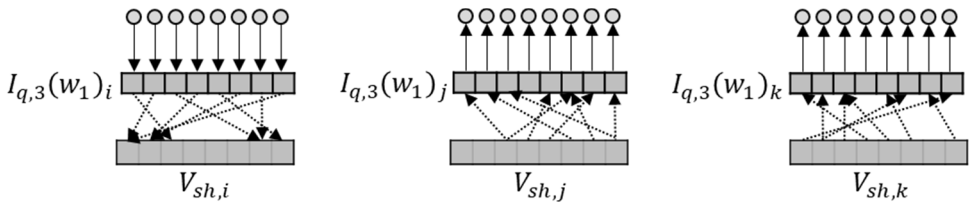
그림 5-11 GPU에서 approximation base tensor의 처리 방법



(a)



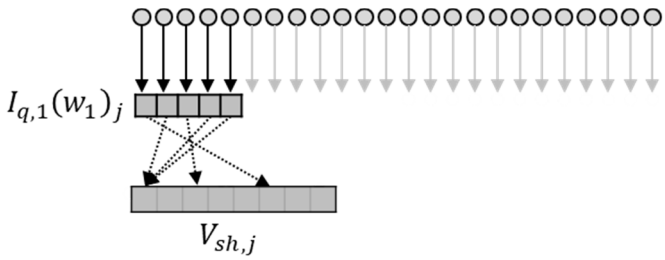
(b)



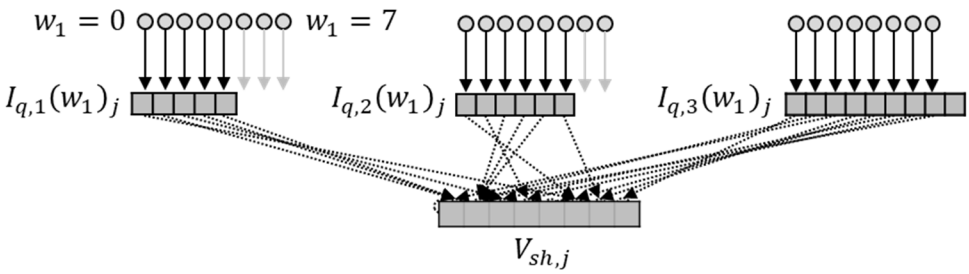
(c)

그림 5-12 shared memory를 이용한 query index tensor 처리 방법

하나의 block에서 생성될 수 있는 thread의 개수는 1,000개가 넘는다. 그러나, 각 query index tensor의 non-zero element 개수는 thread 개수보다 적은 경우가 많다. 그림 5-13 (a)는 $I_{q,1}$의 non-zero element 개수가 5개일 때, thread들의 memory 접근 pattern을 나타낸다. 많은 thread들 중에 5개의 thread만 연산을 수행하고, 나머지 thread들은 5개의 thread들의 연산이 끝날 때까지 기다리고 있거나 이용되지 못한다. 이는 thread들의 GPU의 utilization을 크게 하락 시킨다. 그림 5-13 (b)는 thread utilization을 향상시키기 위하여 $I_{q,1}$, $I_{q,2}$, $I_{q,3}$을 함께 처리한 것이다. 이를 위하여, thread들은 8개 단위로 나누어지고, 8개의 thread들은 각각 $I_{q,1}$, $I_{q,2}$, $I_{q,3}$을 맡아서 처리한다. 이 방법을 적용함으로써, 이용되는 thread가 많아지며, 이는 thread utilization을 크게 향상시킨다.



(a) 최적화 전의 memory 접근 pattern



(b) 최적화 후의 memory 접근 pattern

그림 5-13 제안하는 최적화 방법에 따른 memory 접근 pattern.

5.4 실험 결과

표 5-1은 graph matching algorithm에 따른 affinity tensor의 non-zero element를 개수를 나타낸다. 첫 번째 행은 여러 종류의 graph matching algorithm을 나타내며, 두 번째 행은 affinity tensor의 non-zero element 개수를 나타낸다. 두 번째 열은 2장의 high-order graph matching algorithm을 나타내며, 세 번째 열은 approximate spectral matching algorithm을 네 번째 열은 제안하는 approximation 방법을 나타낸다. Feature set 1과 feature set 2의 feature 개수가 N_1과 N_2일 때, original high-order graph matching algorithm에서 affinity tensor의 non-zero element 개수는 $N_1^3 N_2^3$이다. ASM에서 affinity tensor의 non-zero element 개수는 $N_1^3 + N_{T,1} N_2^3$로 변한다. 여기서, $N_{T,1}$은 feature set 1에서 approximated tuple 개수이며, approximation 정도에 따라 달라지는 값이다. 제안하는 approximation 방법에서 affinity tensor의 non-zero 개수는 $N_1^3 + N_2^3$이 된다. 제안하는 방법에서 non-zero element 개수는 approximate tuple 개수 $N_{T,1}$에 영향을 받지 않는다.

표 5-1표는 affinity tensor의 non-zero element 개수

Graph matching algorithm	High-order graph matching	Approximate Spectral Matching	제안하는 approximation
Affinity tensor의 non-zero 개수	$N_1^3 N_2^3$	$N_1^3 + N_{T,1} N_2^3$	$N_1^3 + N_2^3$

표 5-2는 feature 개수와 approximation 정도에 따라 affinity tensor의 memory 사용량을 비교한것이다. 첫 번째 행은 다양한 graph matching algorithm을 나타내며, 두 번째 행은 feature 개수가 64개이고 5°도 단위 approximation 조건을, 세

번째 행은 feature 개수가 64개이고 2°도 단위 approximation 조건을, 네 번째 행은 feature 개수가 128개이고, 2°도 단위 approximation 조건을 나타낸다. 두 번째 열은 2장의 high-order graph matching algorithm을 나타내며, 세 번째 열은 approximate spectral matching(ASM) algorithm을, 네 번째 열은 제안하는 approximation 방법을 나타낸다. Feature 개수가 64개일 때, 5° 도 단위 approximation에서 original high-order graph matching의 memory 사용량은 256GB로 매우 크다. 이때, ASM을 사용하게 되면 memory 사용량은 669MB로 크게 감소된다. 제안하는 방법의 memory 사용량은 6MB로, ASM보다도 약 100배 작다. Approximation 정도를 5°도 단위에서 2°도 단위로 낮추었을 때, ASM의 memory 사용량은 약 4GB로 크게 증가하였다. 반면, 제안하는 approximation 방법의 memory 사용량은 6MB로 변하지 않는다. Feature 개수를 128개로 증가시켰을 때, original high-order graph matching algorithm의 memory 사용량은 4TB로 매우 크게 증가하였다. 또한, ASM의 memory 사용량도 32GB로, 최신 GPU들의 memory 크기가 10GB 수준인 것을 감안할 때, 이 algorithm 역시 GPU에서 구현되기 어렵다. 반면, 제안하는 approximation 방법의 memory 사용량은 48MB로 일반 GPU 뿐만 아니라 mobile GPU에서도 구현될 수 있는 수준이다.

표 5-2 Affinity tensor의 memory 사용량 비교

Graph matching algorithm	High-order graph matching	ASM	제안하는 approximation
Feature 개수: 64 Approximation: 5°	256GB	669MB	6MB
Feature 개수: 64 Approximation: 2°	256GB	4GB	6MB
Feature 개수: 128 Approximation: 2°	16TB	32GB	48MB

표 5-3은 5°도 단위 approximation에서 feature 개수에 따른 처리 시간을 비교한 것이다. 첫 번째 행은 feature 개수를, 두 번째 행은 4장의 제안하는 ASM 구현 방법을, 세 번째 행은 5장에서 제안하는 approximation 방법의 GPU 구현을 나타낸다. 이 실험에서 feature set 1과 feature set 2의 feature 개수는 동일하다. 첫 번째 행에서 feature 개수가 32라는 것의 의미는 feature set 1과 feature set 2의 feature 개수가 각각 32개라는 것이다. 32개의 feature를 사용하였을 때, 제안하는 approximation 방법에 따른 처리 속도 향상은 1.4배 정도이다. 하지만, 64개와 96개의 feature를 사용하였을 때, 제안하는 방법에 따른 속도 향상은 약 4배로 증가하였다.

표 5-3 **5**°도 단위 approximation에서 feature 개수에 따른 처리 시간 비교

구현 방법	처리 시간 (ms)		
	32 features	64 features	96 features
제안하는 ASM 구현	8.1	267	1,814
제안하는 approximation 방법	5.6	67	492

표 5-4는 10°도 단위 approximation에서 feature 개수에 따른 처리 시간을 비교한 것이다. 사용한 algorithm은 표 5-3과 동일하다. 제안하는 approximation 방법의 처리 속도는 ASM 구현보다 약 1.2배~2배 빠른 것으로 나타났다. 전반적으로 5°도 단위 approximation에 비하여 처리 속도 향상이 적게 나타났다. 이러한 현상이 발생하는 원인은 제안하는 approximation 방법에서 thread-block workload가 균일하지 않기 때문이다. Thread-block들은 GPU의 streaming multiprocessor(SM)에 할당되어 처리되는데, 만약 block의 개수가 충분히 생성된

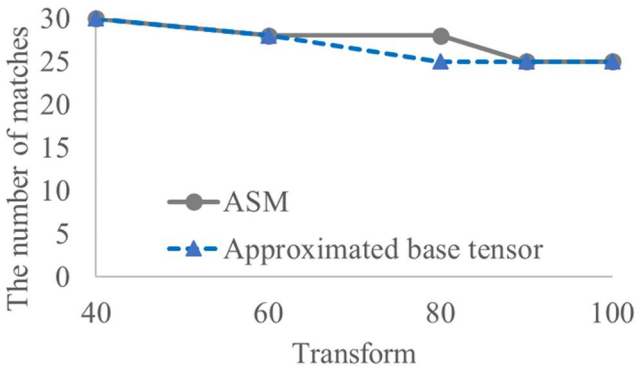
다면 dynamic scheduling에 의하여 SM의 workload를 균일하게 맞출 수 있다. 하지만, block의 개수가 적을 경우, SM의 workload를 균일하게 하는데 한계가 있다. Approximation 정도가 높을수록 생성되는 thread-block의 개수는 감소하므로 5°도 단위 approximation보다 10°도 단위 approximation에서 workload 균일성이 하락한다. 그리고 이는 표 5-4의 결과와 같이 속도 향상 정도가 크지 않게 되는 것이다. 다음 6장에서는 이러한 workload imbalance를 분석하여 workload 균일성을 위한 clustering 방법을 제안한다.

표 5-4 10°도 단위 approximation에서 feature 개수에 따른 처리 시간 비교

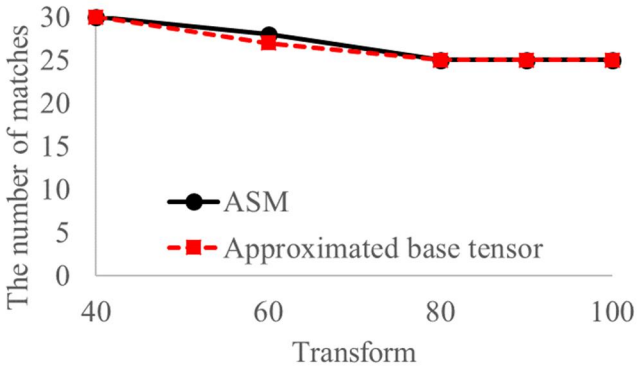
구현 방법	처리 시간 (ms)		
	32 features	64 features	96 features
제안하는 ASM 구현	5.3	120	747
제안하는 approximation 방법	3.8	66	590

그림 5-10은 approximated base tensor와 ASM의 matching 정확도를 비교한 것이다. 제안하는 approximation 방법은 feature set 2도 approximation하기 때문에, feature set 1만 approximation하는 ASM 방법보다 matching 정확도가 하락한다. 실험에서 제안하는 approximation 방법은 feature set 1과 feature set 2를 동일하게 approximation하였다. 사용한 test image는 House test set이며, 그림에서 가로축은 test image의 transform 정도를, 세로축은 correct match 개수를 나타낸다. 그림 5-10 (a)는 2° 단위 approximation에서 ASM과 제안하는 approximation 방법을 비교한 것이다. Transform 정도가 80일 때를 제외하고는 제안하는 방법과 ASM의 matching 정확도가 같았다. Transform 정도가 80일때에만 ASM의 matching 정확

도가 제안하는 방법보다 더 높았다. 그림 5-10 (b)는 5° 단위 approximation에서 ASM과 제안하는 approximation 방법을 비교한 것이다. Transform 정도가 60일 때를 제외하고는 제안하는 방법과 ASM의 matching 정확도가 같았다. Transform 정도가 60일 때에도 matching 정확도 차이가 거의 없었다. 결론적으로, 제안하는 방법은 matching 정확도를 크게 하락시키지 않으면서도, memory 사용량을 획기적으로 감소시켰다.



(a) 2° 단위 approximation



(b) 5° 단위 approximation

그림 5-14 Approximated base tensor의 matching 정확도 비교

제6장 GPU에서 효율적인 index tensor 처리를 위한 tuple clustering

6.1 Index tensor의 형태와 처리 시간과의 관계

그림 5-10의 과정은 하나의 block이 접근하는 index tensor를 나타내며, GPU에서 이러한 block들은 streaming multiprocessor(SM)에서 처리된다. 그림 6-1은 서로 다른 두 개의 SM이 참조하는 index tensor를 나타낸다. SM1은 $I_{r,1}$과 $I_{q,0}$, $I_{q,1}$, $I_{q,2}$를 접근하고, SM4는 $I_{r,4}$과 $I_{q,3}$, $I_{q,4}$, $I_{q,5}$를 접근한다고 가정하였다. 그림에서 reference index tensor와 query index tensor 모두 10° 단위로 approximation 하였으며, 이에 따라 index tensor의 개수는 각각 171개이다. 그림 6-1에서 SM들이 참조하는 index tensor들의 non-zero element 개수는 SM의 처리 시간에 영향을 준다.

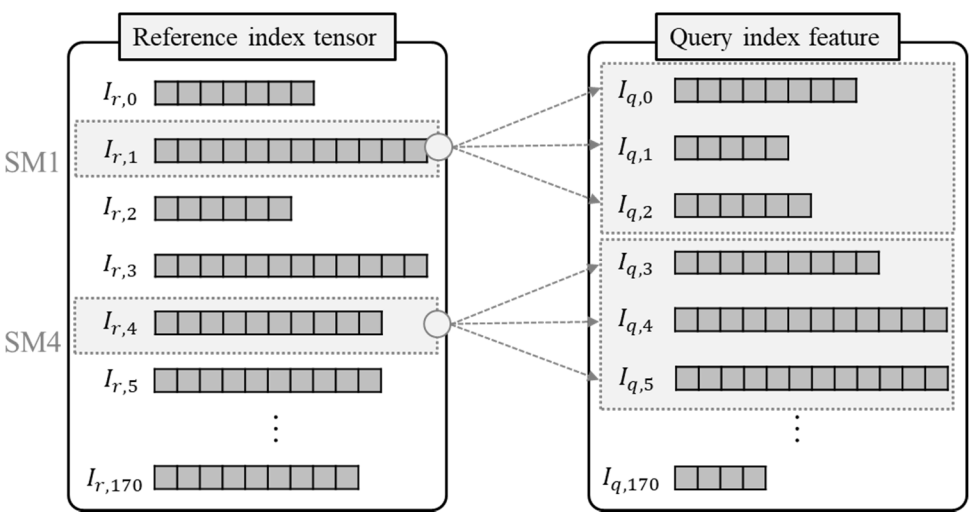
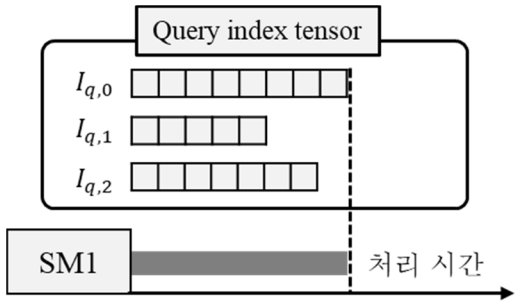
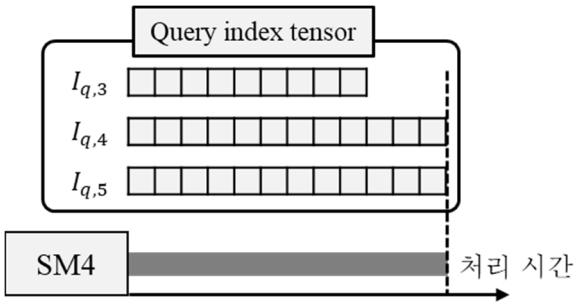


그림 6-1 두 개의 SM이 접근하는 index tensor

그림 6-2는 하나의 SM이 참조하는 index tensor의 non-zero element 개수와 처리 시간의 관계를 보여준다. 그림 6-2 (a)는 그림 6-1의 SM1에 대한 처리 시간을 보여주며, 그림 6-2 (b)는 그림 6-2의 SM4에 대한 처리 시간을 보여준다. 그림 5-9의 구현으로부터, 그림 6-2 (a)의 $I_{q,0}$, $I_{q,1}$, $I_{q,2}$은 병렬로 처리된다. 이때, SM1의 처리 시간은 $I_{q,0}$, $I_{q,1}$, $I_{q,2}$ 중에서 non-zero element 개수가 가장 많은 index tensor에 영향을 받는다. $I_{q,0}$의 non-zero element 개수가 가장 많기 때문에, SM1에서의 처리 시간은 $I_{q,0}$의 non-zero element 개수에 의해 결정된다. 반면, 그림 6-2 (b)에서는 SM4가 $I_{q,3}$, $I_{q,4}$, $I_{q,5}$을 병렬로 처리하며, $I_{q,4}$와 $I_{q,5}$의 non-zero element 개수가 가장 많다. 따라서, SM4에서의 처리 시간은 $I_{q,4}$와 $I_{q,5}$의 non-zero element 개수에 의해 결정된다. 결국, index tensor들의 non-zero element 개수들은 SM들의 처리 시간을 다르게 한다.



(a)



(b)

그림 6-2 base tensor의 index 개수에 따른 처리 속도

그림 6-3은 각 SM에서의 수행 시간과 GPU에서의 수행 시간 관계를 보여준다. GPU에서의 처리 시간은 마지막으로 종료되는 SM의 처리 시간과 동일하다. 만약, 그림 6-3과 같이 SM의 처리 시간이 균일하지 않다면, 빠르게 종료되는 SM들은 idle 상태로 다른 SM의 동작이 종료될 때까지 기다려야 한다. 따라서, SM의 utilization이 최대가 되도록 각 SM에게 workload를 동일하게 분배하는 것이 중요하다.

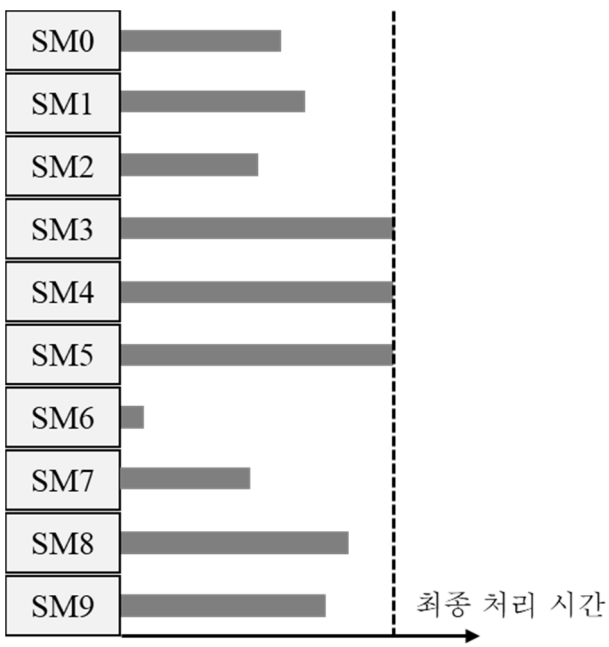


그림 6-3 각 SM에서의 처리 시간과 GPU에서의 처리 시간 관계

그림 6-4는 GPU에서 효율적으로 처리되기 위한 이상적인 index tensor의 형태를 보여준다. Workload balance를 위하여 index tensor들의 non-zero element 개수

들은 균일 해야 한다. 그림에서 reference index tensor의 개수(100개)가 query index tensor 개수(1,000개)보다 더 많다. 이는 각 reference index tensor는 개별적인 thread-block에서 처리되는데, 이 thread-block 개수가 감소될 수록, 처리 시간이 비례하여 감소되기 때문이다.

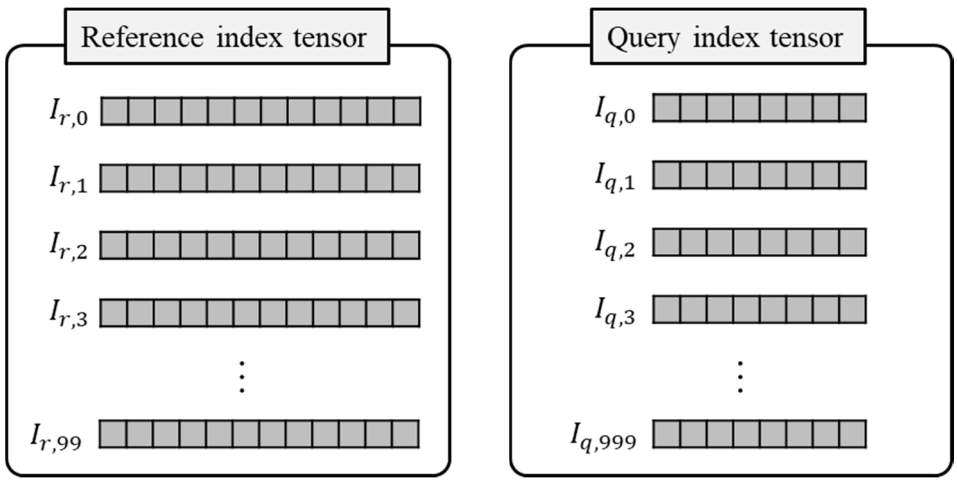
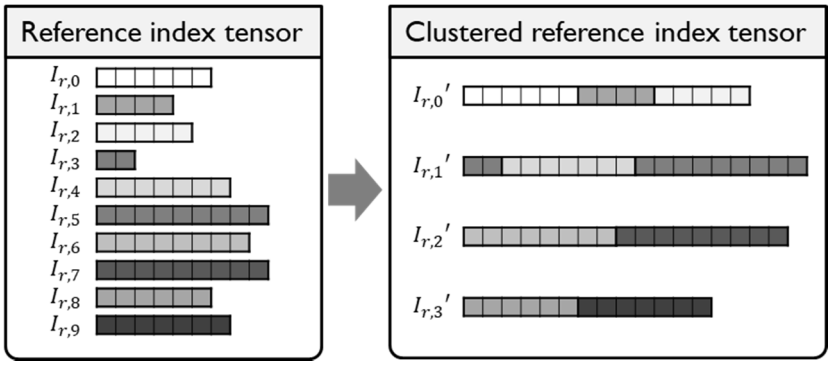


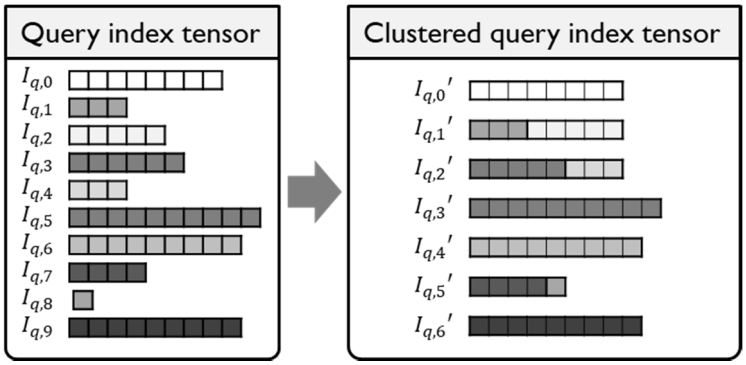
그림 6-4 GPU에서 효율적으로 처리되기 위한 index tensor

6.2 효율적인 병렬화를 위한 tuple clustering

그림 6-4와 같은 이상적인 index tensor를 생성하기 위하여, 본 연구에서는 tuple clustering 방법을 제안한다. 그림 6-5는 제안하는 clustering 기본 전략을 보여준다. 제안하는 clustering 전략은 low approximation에서 많은 개수의 index tensor를 생성한 후에, tuple clustering을 통하여 원하는 개수의 index tensor를 만드는 것이다. Low approximation에서는 왼쪽 그림과 같이 index tensor의 개수가 많으며, index tensor들의 non-zero element 개수가 균일하지 않다. 그림 6-5 (a)는 reference index tensor를 위한 clustering 전략을 나타내며, 그림 6-5 (b)는 query index tensor를 위한 clustering 전략을 나타낸다. Reference index tensor에서는 index tensor의 개수를 감소시키는 것이 중요하다. 따라서, 그림과 같이 많은 개수의 reference index tensor($I_{r,0}$, $I_{r,1}$, …, $I_{r,9}$)에서 clustering 과정을 통하여 적은 개수의 clustered reference index tensor ($I_{r,0}'$, $I_{r,1}'$, $I_{r,2}'$, $I_{r,3}'$)를 생성한다. Query index tensor에서는 index tensor들의 non-zero element 개수 균일성이 더 중요하다. 따라서, 그림과 같이 많은 개수의 query index tensor ($I_{r,0}$, $I_{r,1}$, …, $I_{r,9}$)에서 clustering 과정을 통하여 non-zero element 개수가 균일한 clustered query index tensor ($I_{q,0}'$, $I_{q,1}'$, …, $I_{q,6}'$)를 생성한다. 이 경우, clustered query index tensor의 개수는 clustered reference index tensor보다 더 많다.



(a) Reference index tensor를 위한 clustering 전략



(b) Query index tensor를 위한 clustering 전략

그림 6-5 제안하는 tuple clustering 전략

Query index tensor를 위한 clustering의 경우, 단순히 각 index tensor의 non-zero element 개수를 기반으로 clustering하면 된다. 그러나, reference index tensor의 경우, 같은 방법으로 clustering을 수행하면 matching 정확도가 하락 할 수 있다. High-order graph matching에서는 matching이 잘 되었는지를 판단하기 위하여 score(식 2-2)를 측정한다. 이때, reference index tensor마다 전체 score에 기여하는 정도가 다를 것이다. 즉, 어떤 reference index tensor는 score 기여도가 클 것이고, 어떤 reference index tensor는 score 기여도가 낮을 것이다. 각 reference index tensor I_{r,t_1}에 해당하는 approximated tuple $T_{a,1}[t_1]$이 존재한다. 만약, score 기여도가 큰 approximated tuple이 더 approximation 된다면, 전체 score가 낮아질 가능성이 크다. 반대로, score 기여도가 작은 approximated tuple이 더 approximation 되더라도, 전체 score가 낮아질 가능성은 적다. 따라서, 본 연구에서는 matching 성능 하락을 최소화 하기 위하여 approximated tuple들의 score 기여도를 측정하고, 이를 기반으로 clustering을 수행한다.

그림 6-6은 reference index tensor를 위한 score 기여도 기반 clustering 방법을 보여준다. 그림에서 왼쪽 부분은 clustering 전의 index tensor이며, 오른쪽 부분

은 clustered index tensor들을 나타낸다. 각 index tensor마다 tuple 형태와 score 기여도를 나열하였다. Clustering 이전의 index tensor에서 approximate tuple들은 형태가 유사한 순서대로 나열되어 있다. 그리고 각 tuple 마다 score 기여도가 옆에 숫자로 나타나있다. 제안하는 clustering 방법은 첫 번째 index tensor부터 score 기여도 값을 차례대로 더하면서, 더해진 score 기여도 값이 특정 threshold 이상인지 확인한다. Score 기여도 합이 특정 threshold 이상일 경우, 현재까지의 tuple들을 이용하여 하나의 clustered tuple을 생성한다. 그림 6-6은 threshold가 24일 때의 예를 보여준다. 첫 번째부터 세 번째 tuple의 score 기여도 합이 24이므로, 이 세개의 tuple은 하나의 clustered tuple이 된다. 4번째 tuple은 score가 24이므로, 이 하나의 tuple은 그대로 clustered tuple이 된다. 이러한 방식으로, 모든 index tensor에 대해서 clustering 과정을 수행한다. Clustering 과정이 완료되었을 때, clustered reference index tensor의 score 기여도는 유사해진다.

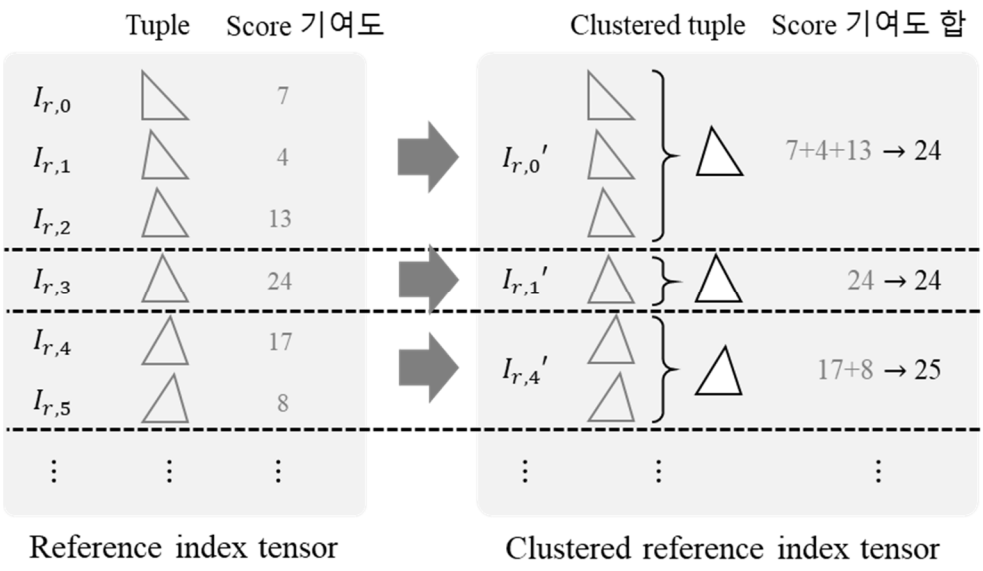


그림 6-6 Reference index tensor를 위한 score 기여도 기반 clustering 방법.

6.3 Score 기여도 기반 clustering 방법

본 연구에서는 어떤 reference index tensor I_{r,t_1}에 대한 score 기여도를 (6-1)과 같이 정의하였다.

$$Score(I_{r,t_1}) = \sum_{k_1=0}^{N_1-1}\sum_{j_1=0}^{N_1-1}\sum_{k_1=0}^{N_1-1} I_{r,t_1}(i_1,j_1,k_1) \sum_{k_2=0}^{N_2-1}\sum_{j_2=0}^{N_2-1}\sum_{i_2=0}^{N_2-1} H(k_1,k_2,j_1,j_2,k_1,k_2)X(i_1,i_2)X(j_1,j_2)X(k_1,k_2) \quad (6\text{-}1)$$

여기서 X는 graph matching algorithm이 완료 되었을 때 계산되는 값이다. 결국, reference index tensor의 score 기여도는 graph matching을 수행한 후에 측정될 수 있는 것이다. 또한, score 기여도 측정은 낮은 approximation에서 수행되어야 하기 때문에, 연산량이 매우 많다. 따라서, 본 논문에서는 reference index tensor의 score 기여도를 approximation 할 수 있는 방법을 제시한다. 식 (6-1)에서 하나의 non-zero $I_{r,t_1}(i_1,j_1,k_1)$ 값에 대하여 $X(i_1,i_2)X(j_1,j_2)X(k_1,k_2)$이 1이 되는 (i_2,j_2,k_2) 경우는 한 가지뿐이다. 그림 6-7은 graph matching이 완료되었을 때, feature들이 matching되어 있는 모습을 보여준다. 두 feature set 사이의 점선은 matching된 feature들을 나타낸다. 그림 6-7에서 feature set 1의 feature 1, 0, 2로 이루어진 tuple을 I_{r,t_1}에 속한 tuple이라고 가정하자. 그리고 식 (6-1)의 X는 그림 6-7의 matching 정보를 가지고 있다고 가정하자. 그렇다면 이 X는 (0,0), (1,1), (2,2), (3,3), (4,7) 위치에서만 1 값을 가지며, 나머지부분은 0이다. I_1, j_1, k_1 값이 각각 1,2,0일 때, $X(i_1,i_2)X(j_1,j_2)X(k_1,k_2)$가 1이 되는 조건은 i_2, j_2, k_2가 각각 1, 2, 0일 때 뿐이다.

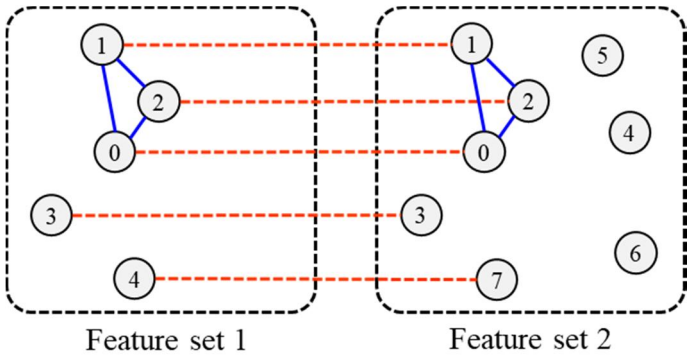
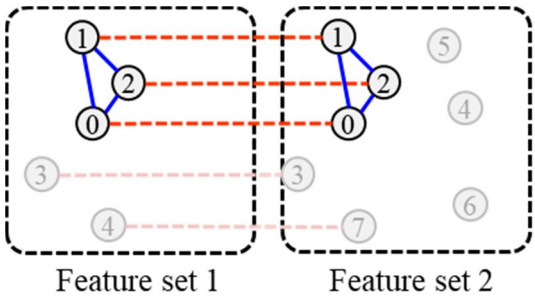
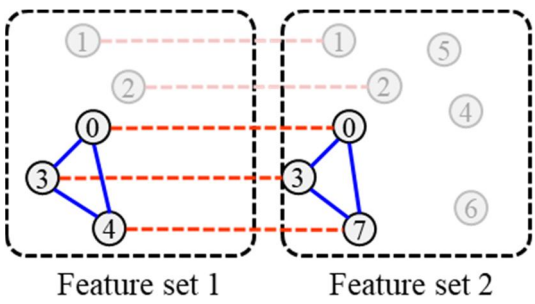


그림 6-7 Graph matching이 완료되었을 때, matching된 모습

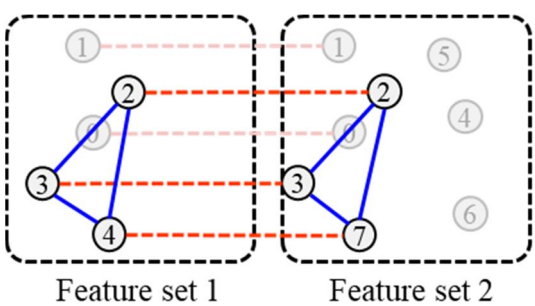
$X(i_1, i_2)X(j_1, j_2)X(k_1, k_2)$ 이 1값을 가질 때, 더해지는 similarity 값 $H(k_1, k_2, j_1, j_2, i_1, i_2)$는 1에 가까운 값을 가진다. 왜냐하면, graph matching이 완료되었을 때, matching이 정확히 되었다면, 더해지는 similarity도 1에 가까울 것이기 때문이다. 그림 6-7에서 feature set 1의 feature 1, 0, 2로 만들어지는 tuple과 이에 대응되는 feature set 2의 feature 1, 0, 2로 만들어지는 tuple의 모양은 서로 유사하다. 따라서, 이 둘의 similarity 값을 클 것이다. 그림 6-8은 어떤 reference index tensor의 non-zero 개수가 3개일 때, 계산되는 3개의 similarity를 나타낸다. 먼저, feature set 1의 세 개 tuple들은 같은 reference index tensor에 속하기 때문에 tuple의 형태가 유사하다. 그림 6-8 (a), (b), (c)로부터 더해지는 similarity 값을 각각 similairty1, similarity2, similarity3이라고 할 때, 어떤 reference index tensor I_{r,t_1}의 score 기여도는 similarity1+similarrity2+similarity3이 된다. 이때, 그림 6-8 (a), (b), (c)에서 대응되는 두 tuple들의 형태가 유사하기 때문에 각 similarity 값을 1로 가정할 수 있다. 그렇다면, I_{r,t_1}의 score 기여도는 3이라고 볼 수 있으며, 이 값은 I_{r,t_1}의 non-zero element 개수와 일치한다.



(a) Similarity 1



(b) Similarity 2



(c) Similarity 3

그림 6-8 하나의 reference index tensor에 대한 score 기여도

결국, reference index tensor I_{r,t_1}의 score 기여도는 I_{r,t_1}의 non-zero element 개수로 approximation 할 수 있다. 그리고 모든 index tensor의 score 기여도도 그림 6-9와 같이 해당 index tensor의 index 개수로 approximation 할 수 있다. Reference index tensor의 non-zero element 개수는 tensor를 생성하는 과정에서 정해지는 값이므로, graph matching을 수행 전에 알 수 있는 정보이다. 결론적으로, reference index tensor와 query index tensor 모두 non-zero element 개수를 기반으로 clustering함으로써, 원하는 clustered index tensor를 얻을 수 있다.

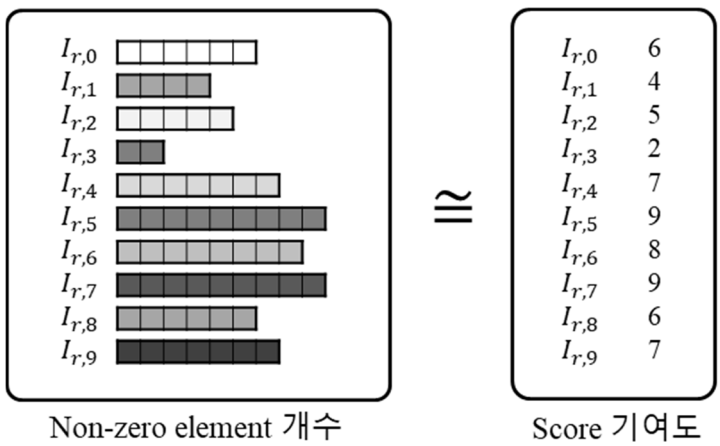
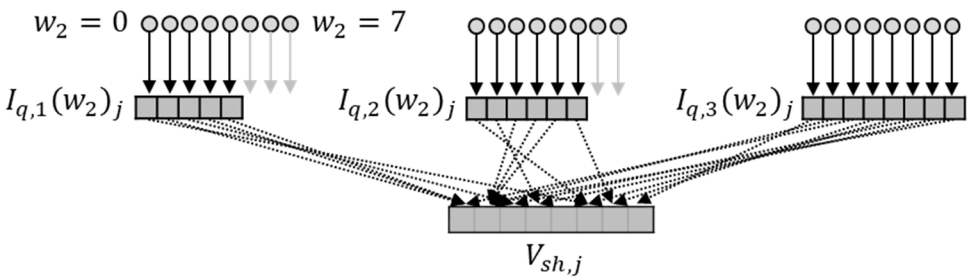


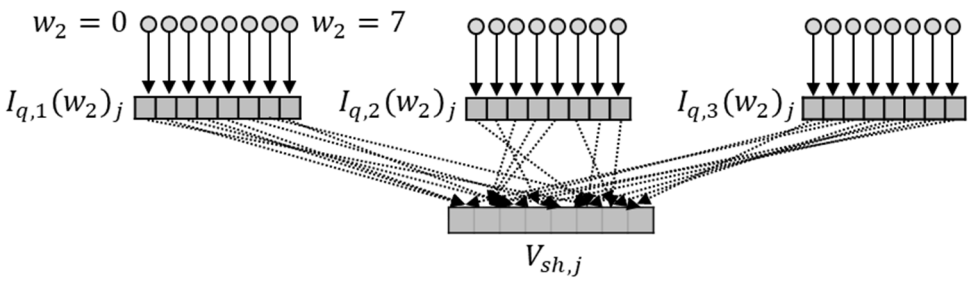
그림 6-9 Reference index tensor의 non-zero element 개수와 score 기여도의 관계

그림 6-10은 clustering 전과 후의 GPU thread들의 memory 접근 pattern을 보여준다. 그림 6-10 (a)는 clustering 전의 memory 접근 pattern이며, 그림 6-10 (b)는 clustering 후의 memory 접근 pattern이다. Clustering 수행 전에는 thread들이 접근하는 query index tensor의 non-zero element 개수가 다르다. 이에 따라, 24개의 thread 중에서 5개의 thread는 아무 일도 하지 않는다. Clustering 수행 후에

는 thread들이 접근하는 query index tensor의 non-zero element 개수가 같다. 이에 따라, 24개의 thread들이 모두 동작을 수행한다.



(a) Clustering 전

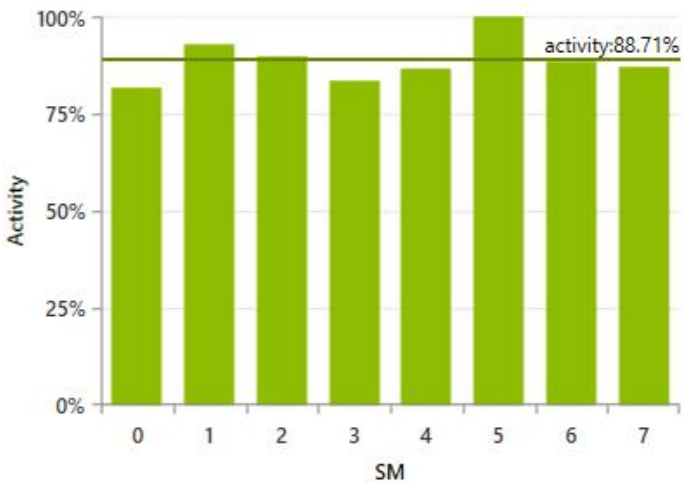


(b) Clustering 후

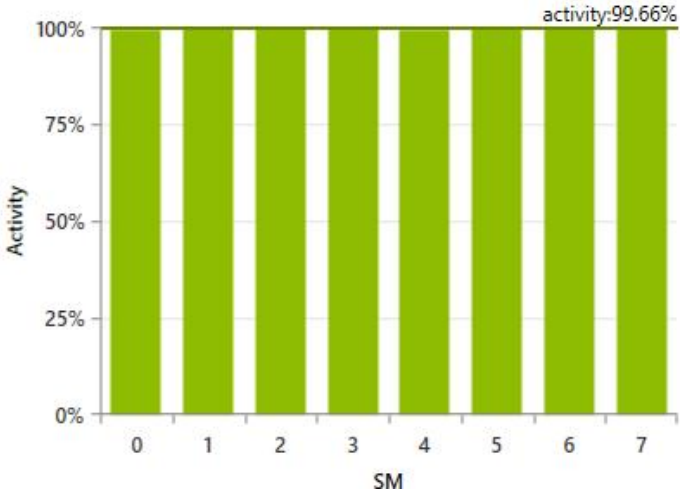
그림 6-10 Clustering 전과 후의 query index tensor를 이용한 memory 접근 pattern

6.4 실험 결과

그림 6-11은 reference index tensor 개수가 666개 일 때, clustering 전과 후의 SM utilization들을 비교한 것이다. 실험을 위하여 사용한 GPU는 NVIDIA GTX 960이며, 이 GPU는 8개의 SM으로 구성되어있다. SM utilization 측정을 위하여 NVIDIA Visual Profiler를 이용하였다. 그림 6-11 (a)는 clustering 전의 SM utilization 결과를 나타내며, 그림 6-11 (b)는 clustering 방법이 적용되었을 때 SM utilization 결과를 나타낸다. Graph에서 가로축은 SM들을 가리키는 번호를 나타내며, 세로축은 utilization 정도를 나타낸다. 그림 6-11 (a)와 같이 clustering 방법이 적용되지 않았을 때에는 SM 간의 utilization 정도가 차이가 났다. 그리고 평균적인 SM utilization은 88.71%인 것으로 나타났다. 반면, 그림 6-11 (b)와 같이 clustering 방법이 적용되었을 때, SM 간의 utilization 차이가 거의 없었다. 평균적인 SM utilization은 99.66%로 100%에 가까웠다.



(a)

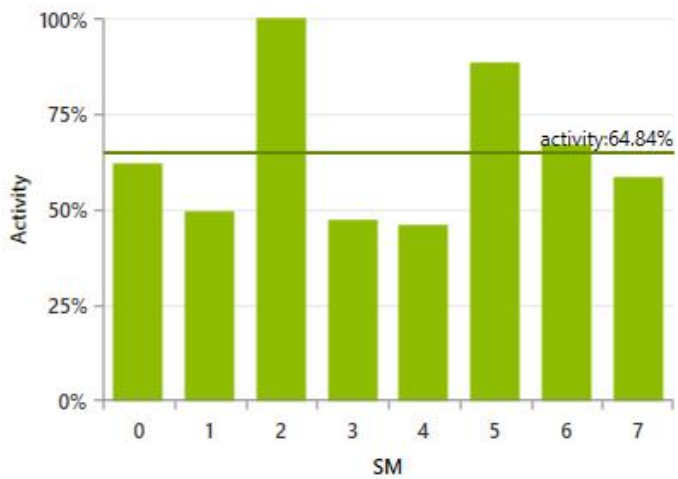


(b)

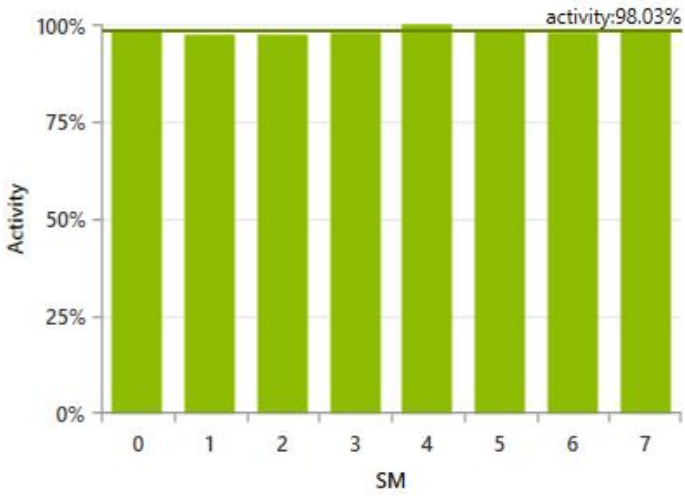
그림 6-11 Reference index tensor 개수가 666개일 때, clustering 전과 후의 SM activity 비교

그림 6-12는 reference index tensor 개수가 더 적을 때(171개), clustering 전과 후의 SM utilization들을 비교한 것이다. 그림 6-12 (a)는 clustering 전의 SM utilization 결과를 나타내며, 그림 6-12 (b)는 clustering 방법이 적용되었을 때 SM utilization 결과를 나타낸다. Reference index tensor 개수가 더 적을 때에는 SM 간의 utilization 차이가 더 크게 나타났다. 8개의 SM 중에서 3개의 SM은 50%의 utilization도 달성하지 못하였다. 평균적인 SM utilization은 64.84%로 나타났다. 반면, clustering 방법을 적용하였을 때에는 reference index tensor 개수가 적을 때에도 매우 높은 SM utilization을 보여주었다. 대부분의 SM utilization이 100%를 달성하였으며, 평균적인 utilization도 98.03%였다. Reference index tensor 개수에 따라 utilization이 차이 나는 이유는, reference index tensor 개수가 적으면 thread block 개수도 감소하기 때문이다. Thread block 개수가 적을 때, thread

block들의 workload가 균일하지 않으면, SM의 workload 차이가 많이 나게 된다. 반면, thread block 개수가 많을 때에는 workload가 균일하지 않더라도, SM의 workload 차이가 적어진다. 충분한 개수의 block을 생성하는 것은 SM의 utilization을 위하여 중요하다. 하지만, 충분한 개수의 block을 생성할 수 없을 때에는 block의 workload를 균일하게 만드는 것이 중요하다. 제안하는 방법은 block들의 workload를 균일하게 만들었기 때문에, block의 개수가 적더라도 SM의 workload차이가 균일하다.



(a)



(b)

그림 6-12 Approximated tuples 개수가 171개일 때, clustering 전과 후의 SM activity 비교

SM utilization 차이는 처리 속도의 차이로 나타난다. 표 6-1은 제안하는 clustering 방법을 적용하기 전과 적용 후의 처리 속도를 비교한 것이다. 두 번째 행은 reference index tensor 개수가 666개일 때를 보여주며, 세 번째 행은 reference index tensor 개수가 171개를 보여준다. Reference index tensor 개수가 666개 일 때에는 clustering 방법을 적용하기 전과 후의 처리 시간이 큰 차이가 없었다. 이는 clustering 전에도 GPU SM들의 utilization이 충분히 확보되었기 때문이다. 반면, reference index tensor 개수가 171개일 때, 제안하는 clustering 방법을 적용한 처리 시간은 clustering 방법을 적용하지 않았을 때보다 약 6배 향상되었다. Clustering 방법을 적용하기 전에는 SM들의 workload 차이가 많이 나며, 이는 GPU에서 처리 속도 저하로 나타난다. 반면, 제안하는 clustering 방법을 적용하였을 때에는 충분한 SM utilization이 확보되었기 때문에 처리 속도 저하

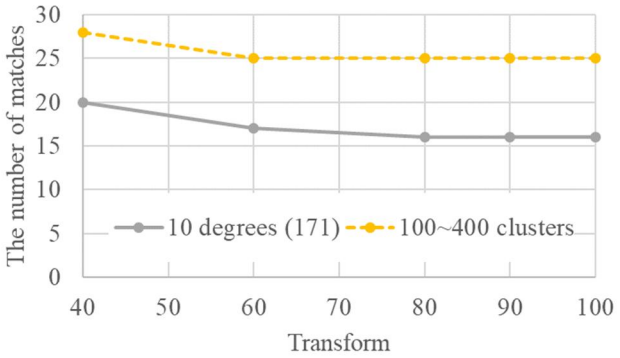
가 나타나지 않는다. 이는 666개의 reference index tensor를 사용했을 때 보다도 약 5배 빠른 속도를 보였다.

표 6-1 제안하는 clustering 방법의 적용 전과 후의 처리 시간 비교

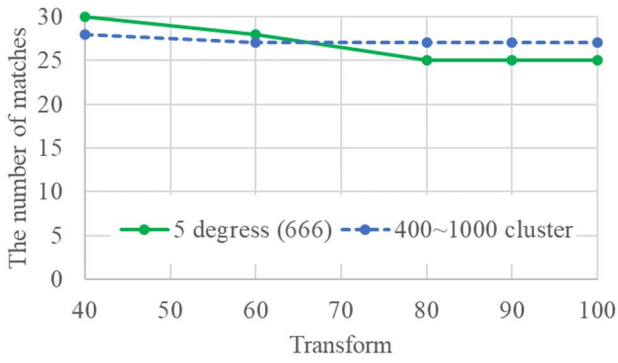
Approximation 조건	처리 시간 (ms)	
	Clustering 적용 전	Clustering 적용 후
5° approximation (666 approximated tuples)	492	464
10° approximation (171 approximated tuples)	590	93

그림 6-13은 clustering 전과 후의 matching 정확도를 측정한 것이다. Matching 정확도 측정을 위하여 House test image를 사용하였다. 그림에서 가로축은 transform 정도를 나타내며, 세로축은 correct match 개수를 나타낸다. ASM의 approximation 정도를 다르게 하여(10°, 5°, 2°) matching 정확도를 측정하였으며, 이 때 reference index tensor의 개수는 각각 171, 666, 4095개이다. ASM과의 비교를 위하여, 제안하는 approximation 방법을 사용하였을 때 clustered reference index tensor 개수를 ASM의 index tensor와 유사하게 하여 matching 정확도를 비교하였다. 그림 6-13 (a)는 ASM에서 10° 단위 approximation 조건일 때(reference index tensor 개수가 171개)와 clustered index tensor의 개수를 비슷하게 했을 때 matching 정확도를 비교한 것이다. ASM에서 10° 단위로 tuple들을 approximation하였을 때, matching 정확도 하락이 매우 크게 나타났다. 반면 제안하는 clustering 방법을 적용하였을 때에는 transform 정도가 100일 때에도, 30

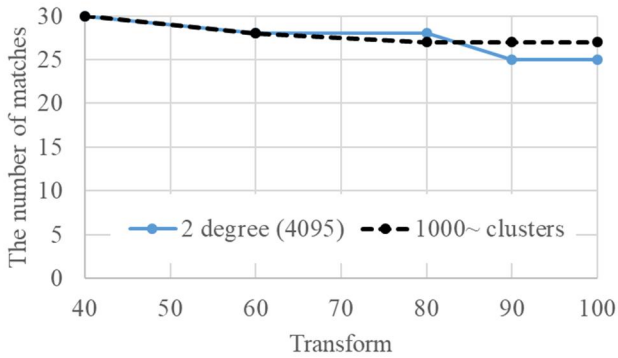
개중 25개의 correct match가 발견되었다. 그림 6-13 (b)는 ASM에서 5° 단위 approximation 조건(reference index tensor 개수가 666개)에서 clustered index tensor의 개수를 비슷하게 했을 때 matching 정확도를 비교한 것이다. Transform 정도가 작을 때에는 제안하는 clustering 방법의 matching 정확도가 약간 낮게 나타났다. 그러나, transform 정도가 강해질수록, 제안하는 방법의 matching 정확도는 거의 하락하지 않는 반면 ASM의 matching 정확도는 하락하였다. 그림 6-13 (c)는 ASM에서 2° 단위 approximation 조건(reference index tensor 개수가 4,095개)와 clustered index tensor의 개수를 비슷하게 했을 때 matching 정확도를 비교한 것이다. Transform 정도가 낮을 때에는 matching 정확도가 비슷하였으며, transform 정도가 커질 때에는 제안하는 방법의 matching 정확도가 더 높게 나타났다. 제안하는 방법에서 clustered index tensor 개수가 1,000개 이상일 때에 matching 정확도가 모두 동일한 것으로 나타났다.



(a) 10° 단위 approximation



(b) 5° 단위 approximation



(c) 2° 단위 approximation

그림 6-13 Tuple clustering에 따른 matching 정확도 비교

제7장 SIFT에서 scale-space 변형 기반 complexity 감소 방법

7.1 SIFT overview

SIFT algorithm은 네 개의 주요 과정으로 구성된다: Scale-space construction, feature detection, orientation assignment, descriptor generation. Scale space는 점차적으로 smoothing되는 Gaussian image으로 구성된다. Gaussian image $L(x, y, \sigma)$는 식 (7-1)과 같이 source image $I(x, y)$와 Gaussian filter $G(x, y, \sigma)$를 convolution하여 생성된다. 여기서, x와 y는 source image에서의 pixel 좌표를 나타내며, σ는 표준편차 값이다.

$$L(x, y, \sigma) = I(x, y) * G(x, y, \sigma) \tag{7-1}$$

$L(x, y, \sigma)$에서 σ 값이 커질수록, scale-space에서 이 Gaussian image의 scale도 높아진다. 그림 7-1 (a)와 같이, SIFT의 scale-space는 여러 개의 octave로 구성된 pyramid를 형성한다. 각 octave는 점차적으로 smoothing되는 여섯 개의 Gaussian image로 구성된다. 다시 말하면, 하나의 octave에는 서로 다른 6개의 scale이 존재한다. Octave의 층이 올라갈수록 Gaussian image의 가로, 세로 길이는 2배씩 작아지는 반면, 각 Gaussian image의 σ 값은 두 배씩 커진다. 하나의 octave에서 Gaussian image의 σ 값은 $2^{1/3}$배씩 차이가 난다. 따라서, octave o에서 scale s의 Gaussian image는 $L(\mathrm{x}, \mathrm{y}, 2^{o+s/3} \cdot \sigma_{init})$로 표현된다. 여기서, $L(x, y, \sigma_{init})$는 source image와 $G(x, y, \sigma_{init})$를 convolution함으로써 얻어지는 octave 0에서 가

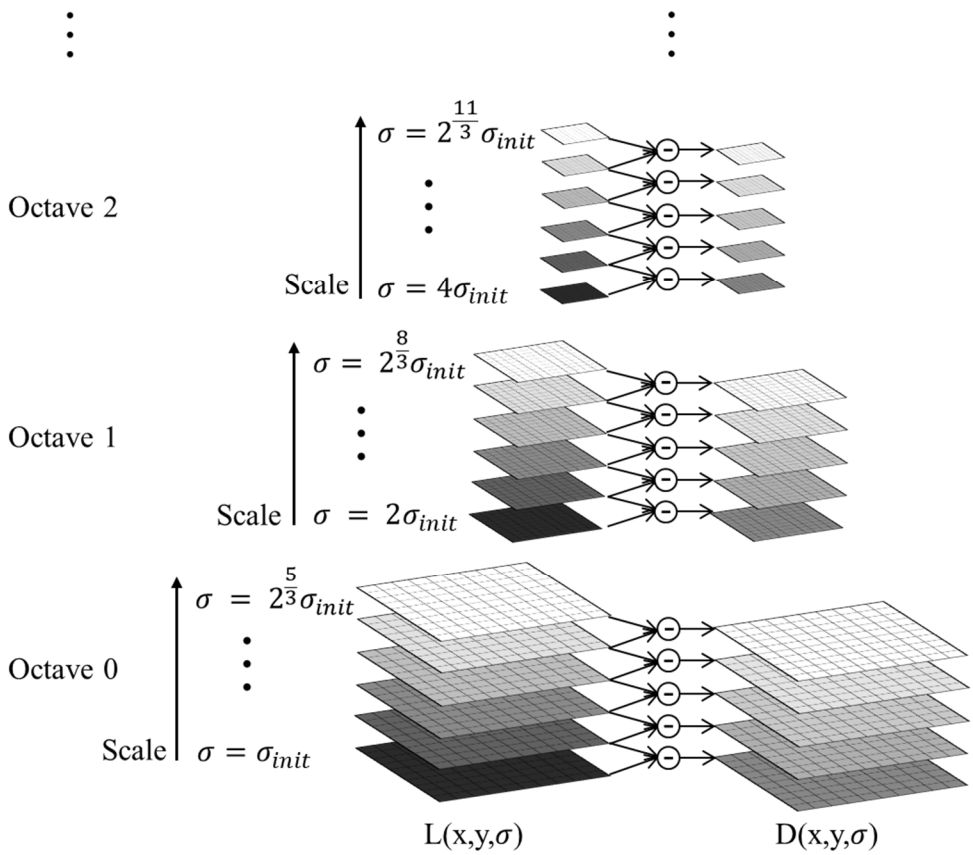
장 낮은 scale을 나타낸다. Octave 0의 나머지 다섯 scale들은 가장 낮은 scale image로부터 생성된다. Octave 1의 가장 낮은 scale은 octave 0의 네 번째 Gaussian image $L(x, y, 2\sigma_{init})$를 downscaling하여 생성된다. Octave 1의 나머지 Gaussian image들은 같은 방식으로 생성된다. 나머지 octave들에 대해서도 이러한 동작이 수행된다. Gaussian scale-space가 생성되면, 식 (7-2)와 같이 인접한 scale의 image 차로 difference-of-Gaussian (DoG) image를 생성한다. DoG space는 다음 과정의 feature detection 과정에서 사용되며, Gaussian scale-space는 orientation assignment와 descriptor generation 과정에서 사용된다.

$$D(x, y, \sigma) = L\left(x, y, 2^{1/3}\sigma\right) - L(x, y, \sigma) \tag{7-2}$$

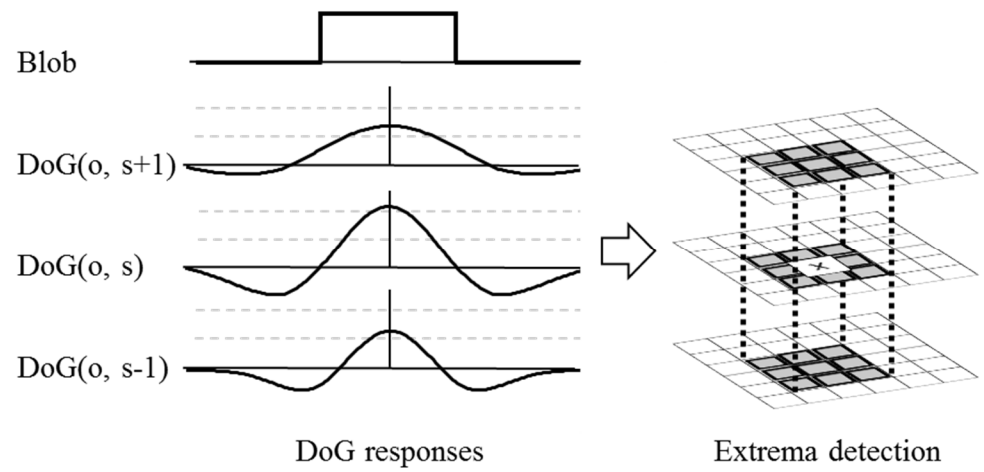
Feature detection 과정에서는 그림 7-1 (b)와 같이 DoG space를 이용하여 blob을 추출한다 [43]. DoG에서 극 값은 blob의 중앙에서 나타난다. Blob의 반지름이 DoG의 σ 값과 특정 constant의 곱과 일치하게 되면, DoG 값은 최대가 된다 [44]. DoG space에서 maximum DoG response를 갖는 blob을 검출하기 위해서, 극 값들이 검출된다. 오른쪽 그림에서 X로 표시된 현재 pixel은 현재 image의 인접한 8개의 pixel 및 s+1과 s-1 scale image에서 인접한 9개의 pixel들과 비교된다. 극 값은 feature 후보 군이 된다. 안정된 feature를 검출하기 위하여 추가적인 동작이 수행된다. 불안정한 후보 군들은 제거되며, 정수 단위의 pixel 좌표와 scale들은 실수 단위의 좌표와 scale로 refine된다. 만약 feature 후보 군이 특정 contrast threshold보다 큰 DoG pixel 값을 갖고 있다면, 이 후보 군은 SIFT feature가 된다.

그림 7-1 (c)는 octave 0의 scale s에서 feature에 대한 orientation assignment와 descriptor generation 과정을 보여준다. 두 동작 모두 patch 영역에 대하여

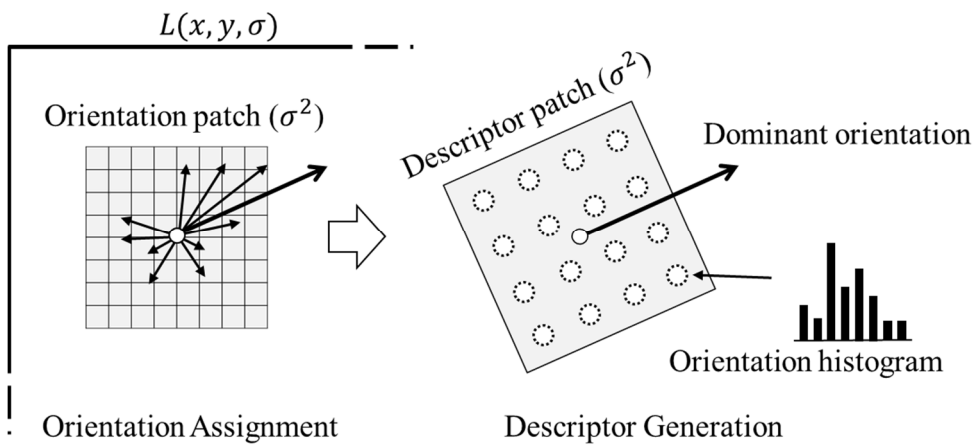
orientation histogram을 생성한다. 이 patch 영역의 중심은 feature의 위치이며, patch의 크기는 σ의 제곱에 비례한다. Orientation histogram은 각 patch 영역에 대하여 생성되며, 가장 큰 orientation이 dominant orientation으로 결정된다. Descriptor patch는 dominant orientation 방향으로 회전되며, 하얀색 동그라미로 표시된 위치에서 orientation histogram들이 생성된다. 이 histogram은 feature descriptor가 된다.



(a) Scale-space 생성 과정



(b) Feature 검출 과정



(c) Orientation assignment와 descriptor generation 과정

그림 7-1 SIFT 동작

7.2 Matching 성능과 complexity 분석

이 장에서는 SIFT를 위한 data 접근량을 분석한다. 이러한 결과는 표 7-1에 요약되어 있다. 이 분석에서 image 해상도의 크기는 1280x720이며, 각 image마다 1,500개의 feature가 검출되었다고 가정하였다. 표 7-1의 첫 번째 열은 SIFT의 main function들을 나타내며, 두 번째 열은 data 접근량을 나타낸다. 여기서 data 접근량은 각 함수에서 load instruction의 실행 횟수를 나타낸다. 표 7-1에서 마지막 열은 load instruction 개수에 영향을 주는 요소들을 나타낸다. 여기서 각 load instruction 마다 memory에서 4 byte가 load 된다고 가정하였다. Scale space는 $\sigma_{init} = 1.6$ 조건으로 생성되었으며, 하나의 Gaussian image를 생성하기 위하여 1차원 Gaussian filter가 가로 방향과 세로 방향으로 각각 사용된다. 2차원 Gaussian filtering은 1차원의 세로 방향과 가로 방향의 filtering 동작으로 분리될 수 있다. 하나의 2차원 Gaussian filter를 사용하는 대신 두 개의 1차원 Gaussian filter를 사용함으로써, 연산량을 크게 감소시킬 수 있다[39]. 현재 scale의 Gaussian image는 이전 scale의 Gaussian image를 이용하여 만들어진다. 이러한 설정은 matching 성능과 연산량 사이의 trade-off를 고려할 때 합리적이라고 할 수 있다[1]. 표 7-1의 분석 결과는 Gaussian scale-space 생성 과정이 가장 많은 data 접근량을 요구한다는 것을 보여주고 있다. 이는 전체 data 접근량의 반 이상을 차지한다. 그리고, 총 data 접근량은 source image 크기의 약 361배이다.

표 7-1 SIFT에서 load instruction의 수행 횟수

Function	load instruction count (in millions)	Determining Factor
Gaussian scale space construction	192.04	Image size, σ_{init}
DoG generation	12.1	Image size
Feature detection*	97.98	Image size
Orientation Assignment	1.8	σ_{init}, # of features
Descriptor generation	29.3	σ_{init}, # of features
Total	333.22	

* 이 값은 feature detection에서 load instruction의 최대 수행 횟수를 나타낸다. 평균 값은 이 값보다는 훨씬 낮을 것이다.

표 7-1의 세번째 열과 같이, Gaussian scale-space 생성 과정의 data 접근량은 image 크기와 σ_{init} 값에 비례한다. 따라서, σ_{init} 값을 감소시킴으로써, data 접근량을 감소시킬 수 있다. Scale-space 생성 과정 동안, Gaussian filter의 크기는 σ_{init}에 비례한다. 또한, 그림 7-1 (c)와 같이, orientation assignment와 descriptor generation 과정의 patch size도 σ_{init}의 제곱에 비례한다. 그러므로, σ_{init}를 감소시키는 것은 SIFT의 연산 복잡도를 크게 감소시킬 수 있다. 반면, σ_{init}의 변화는 인식률과 feature 개수에 영향을 준다. 그림 7-2는 σ_{init} 변화에 따라 달라지는 matching score와 feature 개수를 보여준다. 각 graph의 결과는 [45]에서 사용된 모든 test image에서의 평균 결과이다. Matching score는 두 image에서 검출된 feature 중에서 정확하게 matching된 feature의 비율을 나타낸다. σ_{init}이 감소할수록, matching score는 하락하며, feature의 개수는 증가하는 것을 알 수 있다.

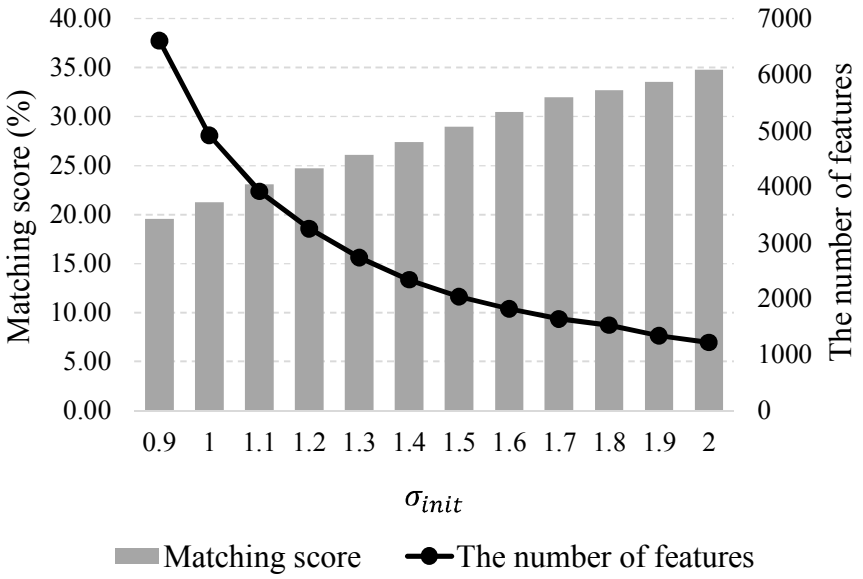


그림 7-2 $\boldsymbol{\sigma_{init}}$ 변화에 따라 검출되는 feature들의 scale 변화

σ_{init} 의 변화는 feature가 검출되는 scale에 영향을 준다. 그리고 이는 matching score와 검출되는 feature 개수에 영향을 주는 것이다. 그림 7-3은 σ_{init} 변화에 따른 feature들의 scale 변화를 보여준다. 가로축은 σ_{init}을 나타내며, 세로축은 feature가 검출되는 scale을 나타낸다. 여기서, feature의 scale이란 feature가 검출되는 DoG image의 σ 값을 나타낸다. 그림 7-1 (a)의 하나의 octave에 속하는 다섯 개의 DoG 중에서 가장 높은 scale과 낮을 scale을 갖는 DoG는 feature detection 과정에서 비교 용도로만 사용된다. 그러므로, 각 octave에서 세 개의 가능한 scale이 존재하며, 이는 그림에서 $s = 1,2,3$으로 표현되었다.

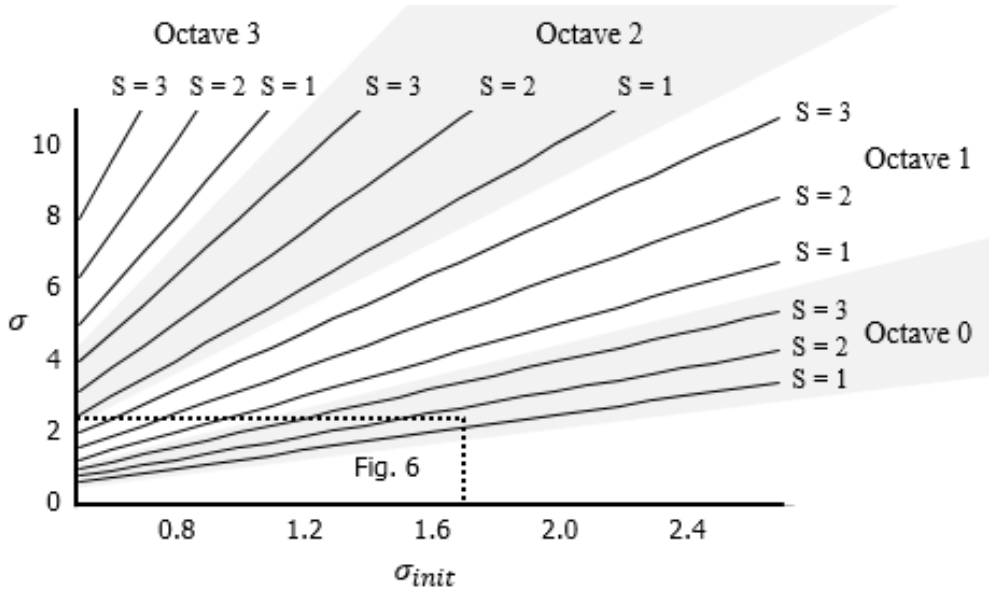


그림 7-3 Matching score와 feature 개수에 대한 $\boldsymbol{\sigma_{init}}$의 영향

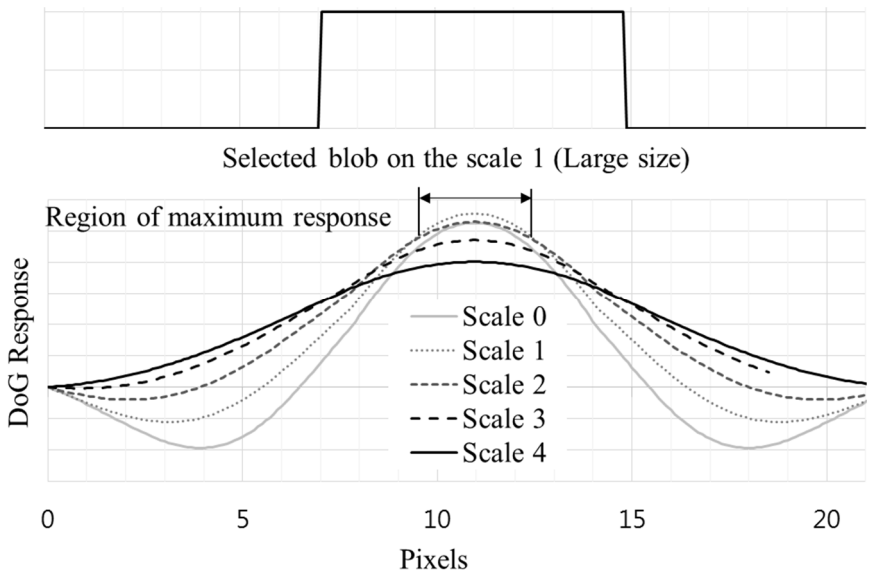
그림 7-3과같이, σ_{init} 값이 감소될수록, feature의 scale은 낮아지는 경향을 보인다. Low-scale의 DoG는 high-scale의 DoG보다 더 많은 high frequency 요소들을 가지고 있다. 이는 높은 σ 값을 갖는 Gaussian filter는 high-frequency 요소들을 더 많이 제거하기 때문이다. High frequency 요소들을 갖고 있는 DoG에서는 feature들이 image의 세부적인 부분에서 추출된다. 이 경우, image의 작은 부분들이 인지될 수 있다. 그러나, 이러한 세부적인 부분들은 이미지에서 다른 세부적인 부분들과 쉽게 구별되기 어려우며, DoG는 noise 정보를 포함하고 있을 수 있다. 결과적으로, feature의 신뢰도가 떨어지며, 이는 matching score의 하락으로 나타나게 된다. 게다가, 이러한 세부적인 부분에서는 많은 개수의 feature가 검출된다. 많은 개수의 feature들은 descriptor generation 과정의 처리 시간을 증가시키게 된다. 이 것은 σ_{init} 감소에 의해 scale-space 생성과정에서 얻어지는 속도 향상을 상쇄시킨다. 반대로, 큰 σ_{init} 값은 검출되는 feature의 개수를

감소시킨다. 이것은 low-scale feature들이 검출되지 않기 때문이다. 대신, 신뢰도가 높은 feature들만 검출되므로 matching score는 증가하게 된다. 그러나, image의 세부적인 부분들은 인식이 안 될 수 있다.

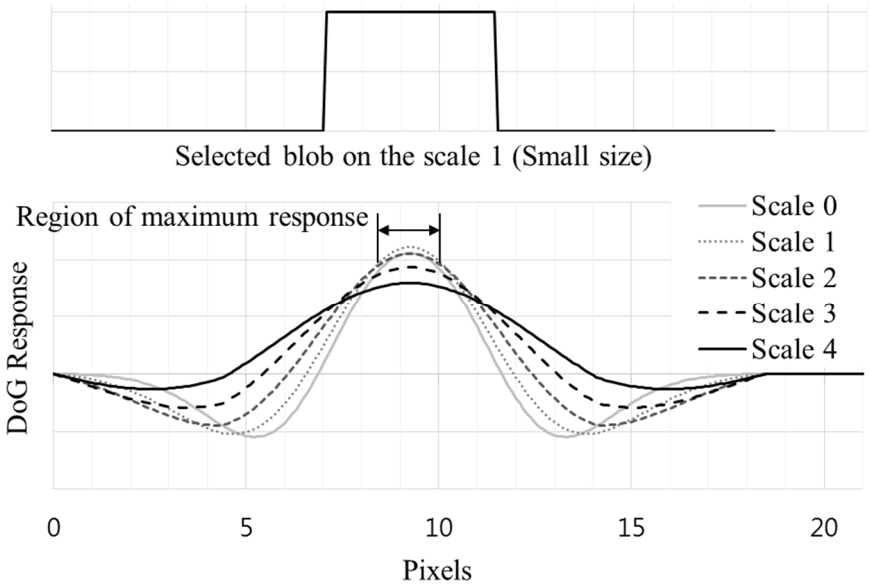
Matching score를 변화시키는 또 다른 요소로서, 낮은 σ_{init} 값은 feature가 검출될 때 scale-selection 과정의 정확도를 하락 시킨다. 그림 7-4 (a)와 (b)는 σ_{init} 값이 변화할 때, scale 1에서 선택되는 blob의 크기를 보여준다. 각 그림의 위 부분은 scale 1에서 선택되는 blob의 형태를 보여준다. 그림의 아래부분에서 가로축은 pixel 위치를 나타내며, 세로축은 DoG의 response 값을 나타낸다. 그림 7-4 (a)와 (b)는 각각 σ_{init} 값이 큰 경우와 작은 경우를 나타낸다. σ_{init}이 작아지면, 검출되는 blob의 크기가 작아지며, 이에 따라 region of maximum response도 작아진다. 만약 σ_{init}이 크게 감소되면, region of maximum response는 1 pixel 이하로 감소된다. 이 경우, 최대 값을 갖는 DoG의 scale이 일관되지 못할 수 있다. Octave 숫자가 증가할수록 image 크기가 감소되는 것을 감안할 때, 높은 octave에서도 유사한 문제가 발생한다. 정확하지 못한 scale selection은 scale 변화에 의한 matching 성능을 하락 시킨다.

Scale 변화는 feature의 성능에 영향을 주는 주요 요소로서, feature scale을 original SIFT의 scale과 동일하도록 보장하는 것이 중요하다. σ_{init} 값이 감소되었을 때, low scale의 feature 개수 증가는 matching 성능을 하락시킬 수 있다. 그림 7-5는 그림 7-3에서 점선 부분으로 표시된 low-scale 부분을 크게 확대한 것이다. Original SIFT의 feature scale(σ_{init} = 1.6)들은 동그라미로 표시하였다. 만약 σ_{init} 값이 1.6에서 어떤 k로 감소된다면, 그림 7-5에서 네모로 표시된 것처럼 feature의 scale들은 낮아진다. 1.6에서 k로의 σ_{init} 값 변화는 $\sigma_{init} = 1.6$일 때의 모든 scale보다 낮은 두 scale들을 생성한다. 만약 두 개의 추가된 low scale에서 feature가 추출되지 않는다면, $\sigma_{init} = k$일 때의 feature의 scale들은 original SIFT의 feature scale들과 거의 동일하다. 마찬가지로, 그림 7-5에서 $\sigma_{init} = k'$일 때 feature의 scale들을 삼각형으로 표시하였다. $\sigma_{init} = 1.6$일 때와 비교하여 세 개

의 low scale이 생성되었다. 이 새로 생성된 세 개의 scale들은 모두 octave 0에 속한다. 만약 octave 0에 해당하는 scale-space가 생성되지 않는다면, 검출되는 feature들은 original SIFT의 feature들과 동일한 scale에 위치한다. 요약하자면, σ_{init} 값이 감소되었을 때, 만약 새롭게 생성되는 low scale들이 octave 0에 속하고 octave 0에 해당하는 scale-space가 생성되지 않도록 σ_{init} 값을 선택된다면 (k'), feature의 scale들은 original SIFT의 feature scale과 동일하게 된다.



(a) 큰 σ_{init}



(b) 작은 σ_{init}

그림 7-4 DoG space에서 scale 1에서의 scale selection 과정

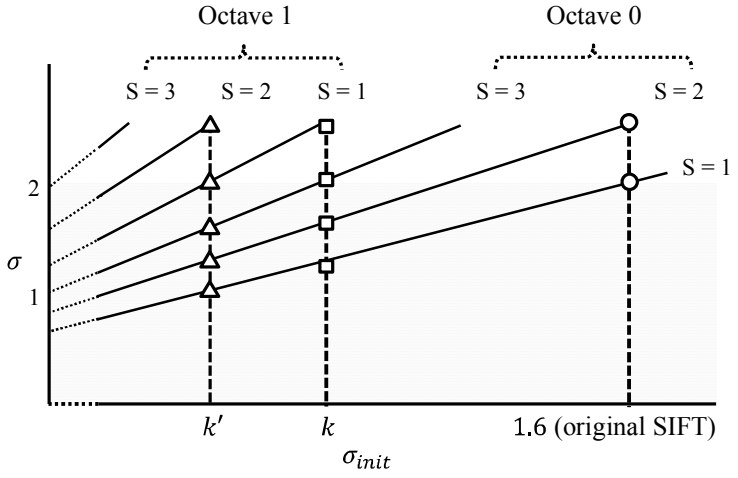


그림 7-5 그림 7-3에서 low-scale 부분

7.3 제안하는 scale-space 생성 방법

Scale s와 octave o의 Gaussian image로 만들어지는scale space는 (7-3)과 같이 표현될 수 있다.

$$\left\{L\left(x, y, 2^{o+\frac{s}{3}} \cdot \sigma_{init}\right) \middle| s = 0,1, \ldots,5, o = 0,1, \ldots, N_o - 1\right\} \quad (7\text{-}3)$$

여기서, o의 범위는 0부터 $N_0 - 1$이며, s의 범위는 0부터 5이다. (7-3)과 같은 scale을 유지하면서 octave 0를 포함하지 않는 새로운 scale-space를 생성하기 위해서는 o가 o-1로 변경되어야 하며, feature detection 과정이 octave 1부터 수행되어야 한다. 식 (7-4)는 octave 0를 생성하지 않으면서 (7-3)과 동일한 scale을 갖는 scale-space를 나타낸다.

$$\left\{L\left(x, y, 2^{o-1+\frac{s}{3}} \cdot \sigma_{init}\right) \middle| s = 0,1, \ldots,5, o = 1,2, \ldots, N_o\right\} \quad (7\text{-}4)$$

(7-4)에서, $2^{o-1+\frac{s}{3}}$에 2를 곱하고 σ_{init}을 2로 나눔으로써 동일한 scale-space (7-5)가 얻어진다.

$$\left\{L\left(x, y, 2^{o+\frac{s}{3}} \cdot \frac{\sigma_{init}}{2}\right) \middle| s = 0,1, \ldots,5, o = 1,2, \ldots, N_o\right\} \quad (7\text{-}5)$$

식 (7-5)는 Scale space가 octave 1부터 생성되고 σ_{init}이 반으로 감소된다면 (7-

3) 의 scale space와 동일한 scale-space를 얻을 수 있음을 의미한다. 제안하는 방법에서 $\frac{\sigma_{init}}{2}$ 값을 표현하기 위하여 σ_{init_half}라는 새로운 기호를 사용하였다. 식 (7-5)는 다음과 같이 변경된다.

$$\left\{L\left(x, y, 2^{o+\frac{s}{3}} \cdot \sigma_{init_{half}}\right) \middle| s = 0,1, \ldots,5, o = 1,2, \ldots, N_o\right\} \tag{7-6}$$

감소되는 σ_{init} 값에 의하여, scale-selection 과정의 정확도가 감소될 수 있다. 그림 7-6은 σ_{init}의 변화에 따른 scale-selection 정확도를 측정하기 위한 실험을 보여준다. 실험 결과, σ_{init} 값이 증가할수록, scale-selection 정확도가 향상되는 것으로 나타났다. 또한, σ_{init} 값이 약 3.0정도로 증가되었을 때, 정확도 향상 정도가 saturation되는 것을 확인하였다. 따라서, 3.0의 σ_{init} 값은 3.0 이하의 다양한 σ_{init} 값에서 scale-selection 정확도를 측정하기 위한 reference 값으로 사용되었다. 특정한 blob에 대하여 정확한 scale이 결정 되었는지를 판단하기 위하여, w x h 크기의 original image는 k배 scaled-up 되었다. 3.0의 σ_{init} 값은 정확한 scale-space를 생성하기에 충분한 값이므로, 이 scaled image의 σ_{init} 값은 3.0으로 설정되었다. Original image에 대하여 t로 표시된 σ_{init} 값을 변경하고, original image와 scaled image에서 검출된 feature들을 비교하였다. 이를 위하여, 각각 (x, y, σ)와 (x', y', σ')로 표시된 두 image의 scale-space 좌표를 비교하였다. x와 y 좌표를 비교하기 위하여, $\left|x - \frac{x'}{k}\right|$와 $\left|y - \frac{y'}{k}\right|$ 값을 계산하여 이 값들이 ε 보다 작은 지를 확인하였다. 이 실험에서 ε 값은 1 pixel으로 하였다. Scaled image에서는 σ_{init} 값도 커지므로, σ 비교를 위해서는 $s = s'$ 와 $o = o'$의 관계를 갖는지를 확인하였다. 두 image의 σ_{init} 값 비율은 3.0/t이며, 이 값은 k와 일치한다.

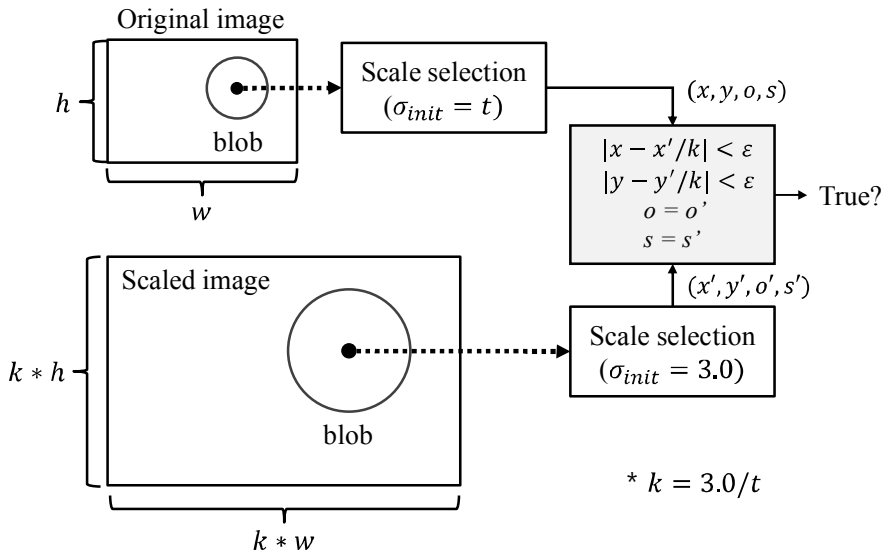


그림 7-6 σ_{init}에 따른 scale-selection 정확도를 측정하기 위한 실험

각 σ_{init} 값에 대하여, 실험에 의한 결과가 true로 나오는 비율을 측정하였다. 그림 7-7은 이 실험의 결과를 보여준다. 가로축은 σ_{init} 값을 나타내며, 세로축은 scale-selection의 정확도를 보여준다. σ_{init} 값이 작아졌을 대, scale-selection의 전반적인 정확도는 하락하였다. Original SIFT의 scale-space와 동일한 scale space를 생성하기 위하여 σ_{init} 값을 0.8로 하였을 때, scale-selection 정확도가 크게 하락하였다. Scale-selection 정확도 하락을 최소화하면서 original SIFT와 비슷한 scale-space를 얻기 위해서는 σ_{init} 값을 적당히 증가시켜야 한다. 그림 7-7에서 σ_{init}가 1.1보다 클 때에는 scale-selection 정확도가 서서히 하락한다. σ_{init} 값이 1.0으로 작아지는 순간부터, scale-selection 정확도 하락이 큰 것으로 나타났다. 이러한 관찰로부터, 제안하는 방법에서는 σ_{init_half} 값을 1.1로 하였다. 제안하는 방법의 Gaussian filter의 크기는 σ_{init} 값이 1.1일 때와 같으며, scale-space의

scale은 σ_{init} 값이 2.2일 때와 같다. 이 값은 original SIFT의 σ_{init} 값(1.6) 보다 더 큰 값이다.

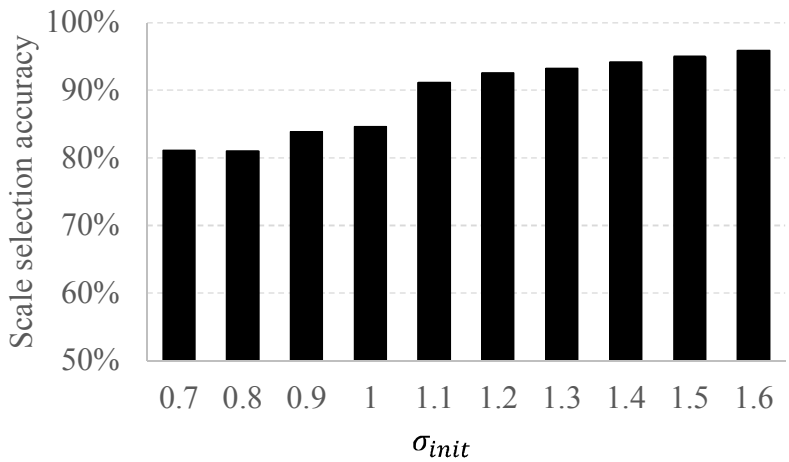


그림 7-7 Scale selection 정확도에 대한 $\boldsymbol{\sigma_{init}}$의 영향

그림 7-8은 octave 0가 필요하지 않는 제안하는 scale-space 생성 방법을 자세히 보여준다. 그러나, octave 1의 가장 낮은 scale은 octave 0의 네 번째 scale을 down-scaling하여 생성된다. 따라서, pre-octave generation이라는 과정이 필요하며, 이 과정에서는 input image를 $G(x, y, 2 * \sigma_{init_hafl})$와 convolution하여 (6-6)에서 o가 0이고, s가 3인 $L(x, y, 2 * \sigma_{init_half})$를 생성한다. Octave 1의 첫 번째 scale이 $L(x, y, 2 * \sigma_{init_half})$로부터 만들어지면, octave 1의 나머지 scale들은 이 첫 번째 scale의 Gaussian image를 이용하여 만들어진다. Octave 2의 첫 번째 scale은 octave 1의 네 번째 scale을 down-scaling하여 생성하며, octave 2의 나머지 scale들은 octave 1과 동일한 방법으로 생성된다. 나머지 octave들도 같은 방식으로 생성된다.

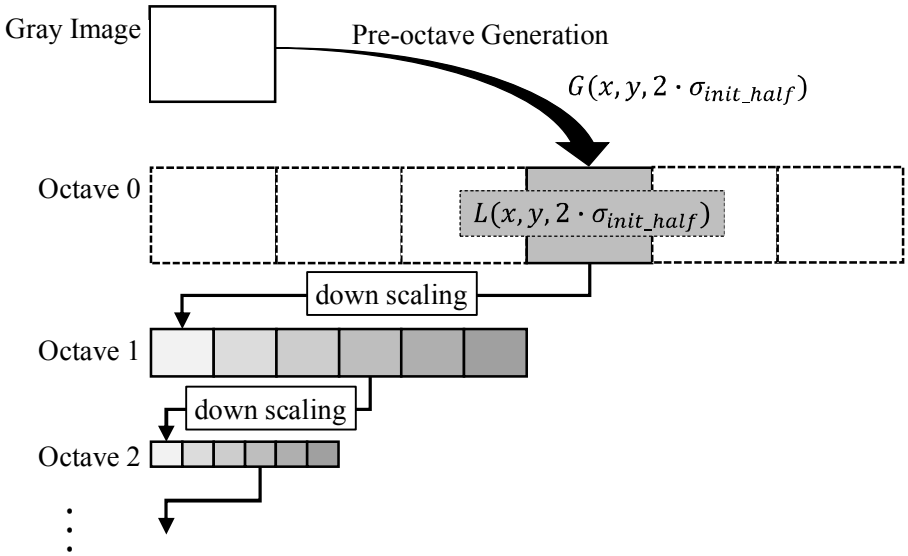


그림 7-8 제안하는 Gaussian scale-space 생성 방법

Scale-space 생성 과정의 연산 복잡도를 추가적으로 감소시키기 위하여, pre-octave generation과 octave 0에서의 downscaling 방법을 결합하였다. Pre-octave generation 과정은 original image 크기에 수행되기 대문에, 많은 양의 연산을 요구한다. Pre-octave generation에서 생성되는 Gaussian image가 feature 검출에는 사용되지 않고 octave 1의 첫 번째 scale을 위해서 sub-sample된다는 것을 고려할 때, octave 0의 Gaussian image의 모든 pixel을 생성할 필요가 없다. 그러므로, pre-octave-generation 과정에서 Gaussian filtering 동작은 octave 1에서 사용될 pixel들에 대해서만 수행되면 된다. 이러한 결합된 동작은 그림 7-9에 설명되어 있다. 가로 방향의 Gaussian filter는 input image의 모든 pixel에 적용된다. 세로 방향의 Gaussian filter는 가로 방향으로 filtering된 image에서 짝수 위치의 pixel들에 대해서만 적용된다. 제안하는 방법은 세로 방향의 Gaussian filtering 동작에 대하여 연산량을 1/4로 감소시켰다.

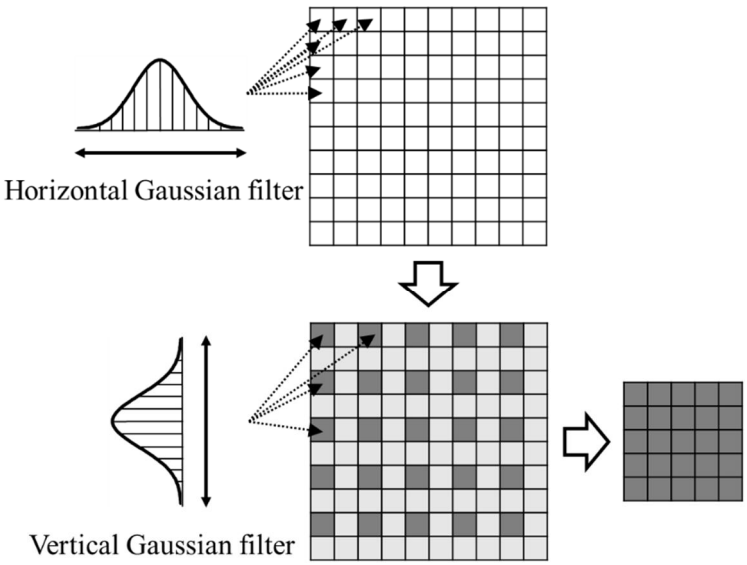


그림 7-9 Octave 1의 첫 번째 scale을 위한 pre-octave generation

σ_{init}과 image 크기는 Gaussian scale-space 생성 과정의 연산 복잡도를 결정하는 두 가지 주요 요소이다. 이어지는 과정들도 이 두 가지 요소에 영향을 받는다. 제안하는 방법의 주요 장점은 matching 성능을 하락시키지 않고 σ_{init}과 image 크기를 동시에 감소시키는 것이다. 제안하는 방법에 의한 연산 복잡도 감소를 다음과 같이 요약하였다. 첫째, scale-space 생성 과정에서 요구되는 data 양이 크게 감소된다. 이는 octave 0 생성 과정의 대부분 연산을 수행하지 않으며, Gaussian filter 크기 또한 감소되었기 때문이다. 둘째, scale-space 생성 이후 이어지는 과정에서 처리해야하는 image의 크기도 크게 감소된다. 이는 scale-space가 octave 1부터 시작하기 때문이다. 셋째, 감소된 σ_{init} 값은 orientation assignment와 descriptor generation 과정의 patch 크기를 감소시킨다. 결과적으로, 제안하는 방법은 scale-space 생성 과정뿐만 아니라 SIFT 동작 전반적인 처리 속도를 크게 향상시킨다.

7.4 Mobile GPU를 위한 최적화

그림 7-10은 mobile CPU와 GPU에 최적화된 제안하는 SIFT 구현을 보여준다. 연산 과정은 CPU와 GPU 사이의 data 전송을 최소화 하는 방향으로 분리되었다. SIFT 대부분의 함수들은 GPU에 의해 수행된다. Input image는 GPU에서 처리되도록 block 모양의 workgroup으로 나뉜다. 이 input image는 GPU에서 세 과정으로 처리된다: Gaussian scale-space construction, Difference of Gaussian generation, Feature detection. 검출된 feature들은 CPU에 전달되고 CPU에 의해서 serialization된다. Serialized feature들은 GPU로 다시 전달된다. Orientation assignment과 descriptor generation 과정이 serialized feature에 대하여 수행된다. Histogram이 생성될 때, global memory 접근을 감소시키기 위하여 local memory가 사용되었다. Orientation assignment와 descriptor generation 과정에서 중복되는 연산을 피하기 위하여 필요한 gradient data는 Gaussian scale-space로부터 미리 계산된다.

Mobile GPU를 위하여 세 가지 최적화 방법이 적용되었다. 첫째로, memory bandwidth를 감소시키기 위하여 texture의 data type을 변경하였다. 그림 7-10의 제안하는 구현에서 texture는 italic으로 표현하였다. Image를 저장하기 위하여 일반적으로 사용되는 single-precision floating-point data를 half-precision floating point로 바꾸었다. 이 type은 OpenCL에서 지원되는 16-bit floating-point format이다. Half-precision의 data 크기는 single-precision의 반이므로, 이 type 변환은 data 크기를 크게 감소시킨다. Gaussian scale-space 생성 과정에서 type conversion에 의한 정확도 하락을 최소화하기 위하여, type 변환은 세로 방향의 Gaussian filter에 의한 결과 image에만 적용되었다. GPU에서의 나머지 동작들은 모두 single-precision float type으로 수행되었다. 가로 방향의 Gaussian filter 결과도 single-precision floating-point 형태로 저장된다. Type 변환이 각 sub-image set에 대하여 적용되었을 때, matching score의 평균적인 변화를 측정하기 위한 실험

을 진행하였다. 평균적으로, type 변환에 의한 matching score 감소는 0.2%였다. 표 7-2는 type 변환에 영향을 받는 함수들의 수행 시간 변화를 보여준다. 첫 번째 열은 함수 이름들이며, 두 번째와 세 번째 열은 single-precision과 half-precision이 각각 적용되었을 때의 수행 시간을 보여준다. 네 번째 열은 single-precision을 사용했을 때의 수행 시간 대비 half-precision을 사용했을 때의 수행 시간이 감소되는 비율을 보여준다. 마지막 열에서 보인 것처럼, half-precision을 사용하였을 때, 수행 시간은 10.63%에서 19.53%가 감소되었다.

둘째로, 기존 연구의 방법[41]을 적용하여 Gaussian scale-space를 병렬로 생성함으로써 dependency를 제거한다. 통상적인 scale-space 생성 방법에서는 Gaussian image들이 이전 scale image를 이용하여 반복적으로 생성된다. 이 통상적인 방법은 Gaussian filter 크기를 감소시킨다. 그러나, 이 방법은 반복되는 동작에서 발생하는 연산 dependence는 병렬 처리의 효율을 하락 시킨다. 제안하는 구현에서는 모든 scale의 Gaussian image를 생성하기 위하여 가장 낮은 scale image만 이용한다. 그림 7-11은 하나의 octave에서 Gaussian scale-space를 생성하기 위한 과정을 보여준다. 검은색과 하얀색 화살표는 각각 세로 방향과 가로 방향의 Gaussian filter를 나타낸다. 화살표 위의 숫자는 Gaussian filtering 순서를 나타낸다. 세로 방향의 filtering 동작이 먼저 수행되며, 세로 방향 filter의 결과 image를 이용하여 가로 방향의 filtering 동작이 수행된다. 세로 방향의 filtering은 모든 data에 병렬로 수행되므로, 하나의 kernel이 세로 방향의 모든 filtering 동작을 수행한다. 같은 방법으로, 다른 하나의 kernel은 가로 방향의 모든 filtering 동작을 수행한다. Scale 1부터 5까지를 생성하기 위하여 scale 0만 사용함으로써, Gaussian filter의 크기는 증가하게 되며, 이는 연산량을 증가시킨다. 많은 memory read를 감소시키기 위하여, vertical filtering을 위하여 memory에서 읽어오는 data를 모든 다섯 scale에 대하여 공유한다. 제안하는 방법에 의하여 연산 복잡도가 증가함에도 불구하고, kernel 사이의 dependence 감소로 처리 속도가 향상되었다.

셋째로, CPU와 GPU사이의 이종 병렬 처리는 전반적인 연산 시간을 감소시킨다. Feature serialization은 CPU에서 병렬처리 없이 수행되기 때문에 상대적으로 처리시간이 길다. 그러나, 제안하는 구현에서 serialization 시간은 GPU에서의 동작으로 가려진다. 각 octave에서 세 scale의 feature들이 존재할 때, feature들은 scale에따라 group화 된다. CPU에서 첫 번째 scale feature에 대한 serialization이 수행되는 동안, GPU에서는 gradient calculation 동작이 수행된다. 두 동작이 CPU와 GPU에서 모두 종료되면, serialized feature들은 CPU에서 GPU로 전달된다. 이 전달된 feature들에 대하여 GPU에서 orientation assignment와 descriptor generation 과정이 수행되는 동안, CPU에서는 두 번째 scale feature들에 대한 serialization이 수행된다. 두 동작이 모두 종료되면, serialized feature들은 CPU에서 GPU로 전달된다. 이어지는 과정들도 그림 7-10과 같이 유사한 방식으로 수행된다.

표 7-2 Half-type 변경에 따른 처리 시간 변화

Operations	Float-type (in ms)	Half-type (in ms)	Reduction Ratio
Pre-octave generation	2.84	2.38	16.20 %
Gaussian scale space construction	5.64	5.01	11.25 %
DoG generation	1.69	1.36	19.53 %
Feature Detection	8.17	6.86	16.03 %
Gradient generation	5.09	4.15	18.47 %
Orientation Assignment	4.15	3.71	10.60 %
Descriptor Generation	13.06	11.67	10.64 %

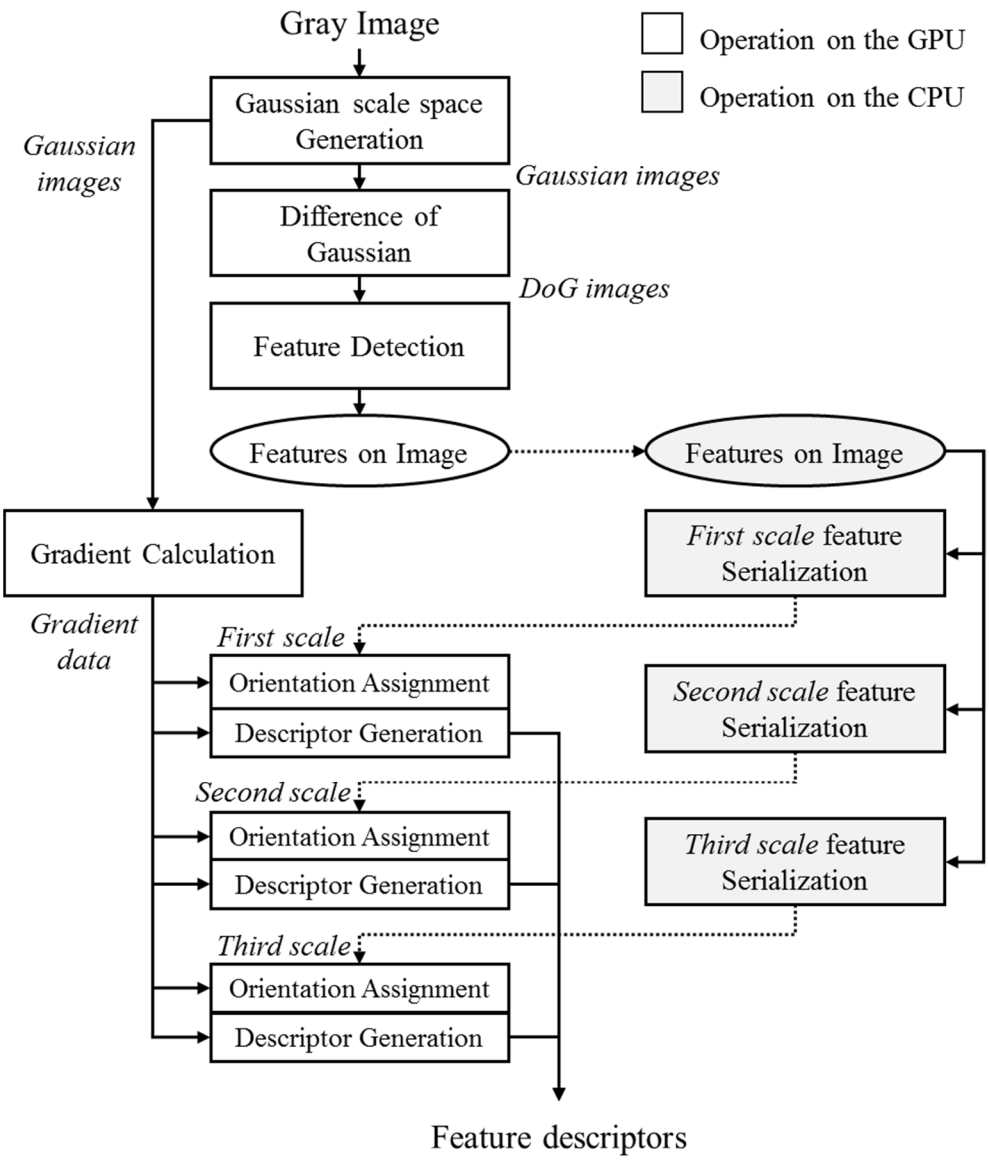


그림 7-10 Mobile CPU와 GPU에서 제안하는 algorithm의 구현

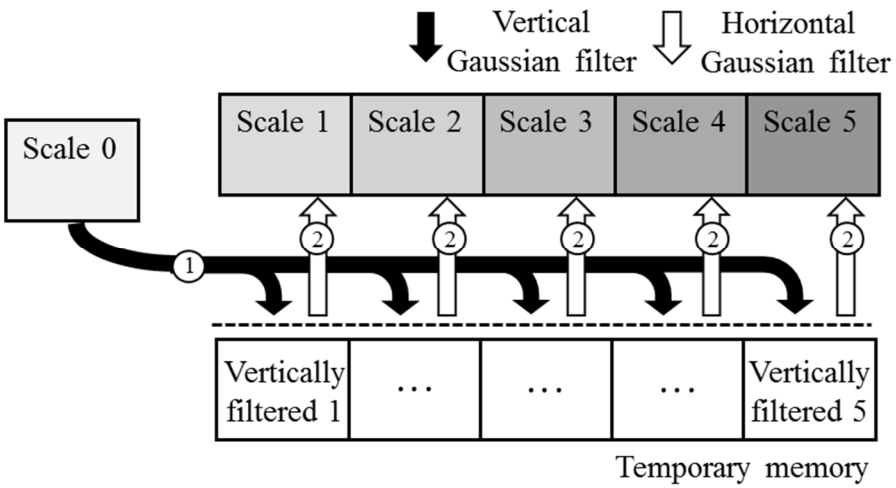
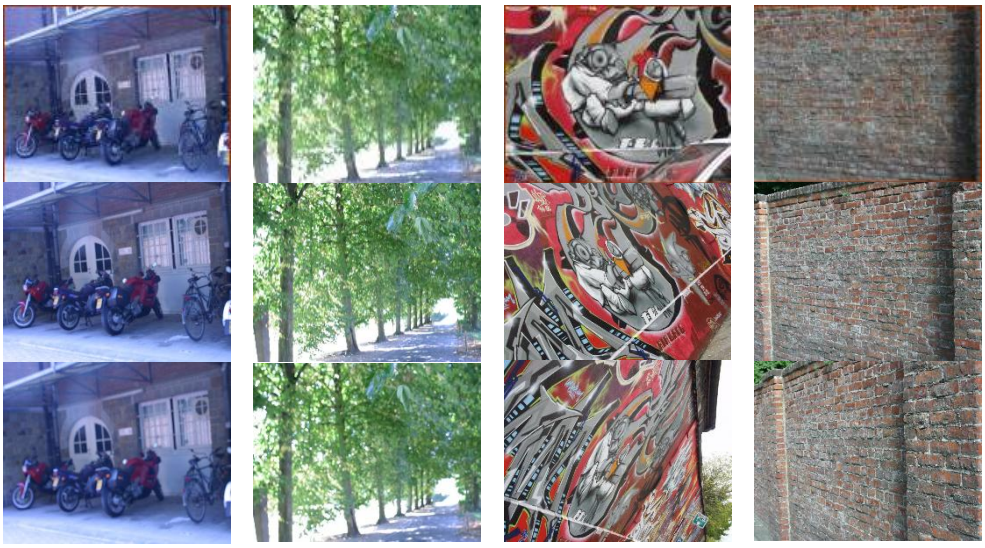


그림 7-11 하나의 octave에서 Gaussian scale-space의 병렬 생성 방법

7.5 실험 결과



(a) Bikes (b) Trees (c) Graf (d) Wall

(e) Bark (f) Boat (g) Leuven (h) UBC

그림 7-12 Test image sets

제안하는 SIFT는 Adreno 420 GPU기반 Qualcomm Snapdragon 805 processor를

사용하는 Galaxy S5 LTE-A에 구현되었다. SIFT는 C++ 언어를 사용하여 구현되었으며, Java Development Kit (JDK, version r9c)와 OpenCL 1.2 library를 사용하여 compile되었다. SIFT의 기본적인 framework는 Hess SIFT[46]를 기반으로 하고 있다. 기존 연구에서 사용되었던 image doubling과 multiple orientation 기법은 본 구현에 적용되지 않았다. 그림 7-12는 (a)부터 (h)까지 다양한 transform 정도를 가지는 8개 sub-image set으로 구성되는 test image set을 보여준다. 각 sub-image set은 다양한 transform 정도를 갖는 여섯 개의 image로 구성되며, 그림 7-12에서는 reference와 세 번째, 다섯 번째 transformed image만 보였다. 그림 7-12 (a)와 (b)는 blur에 대한 변화를, 그림 7-12 (c)와 (d)는 viewpoint에 대한 변화를, 그림 7-12 (e)와 (f)는 scale과 rotation에 대한 변화를, 그림 7-12 (g)는 밝기에 대한 변화를, 그림 7-12 (h)는 JPEG 압축에 따른 변화를 포함하고 있다.

표 7-3 Original SIFT 대비 제안하는 SIFT의 data 접근 비율

Function	Determining factors			Total
	Image size	σ_{init}	Number of features	
Gaussian scale space construction	0.30	0.69 + α	-	0.26
DoG generation	0.25	-	-	0.25
Feature detection	0.25	-	-	0.25
Gradient Generation	0.25	-	-	0.25
Orientation Assignment	-	0.47	0.54	0.25

표 7-3은 original SIFT 대비 제안하는 SIFT의 data 접근 비율을 보여준다.

표의 숫자들은 original SIFT 대비 제안하는 SIFT의 load instruction 수행 횟수의 비율을 나타낸다. 첫 번째 열은 SIFT 동작들을 나타내며, 두 번째부터 네 번째 열은 세 가지 결정 요소(image 크기, σ_{init} , feature 개수)의 변화에 따라 감소되는 data 접근 비율을 나타낸다. 마지막 열은 세가지 결정 요소를 종합한 각 함수에 대한 최종 data 접근 비율을 나타낸다. 두 번째 열에서, octave 0를 건너뛰면서 처리해야하는 image 크기가 감소되었다. 이것은 DoG generation과 feature detection, gradient generation 과정의 data 접근량을 1/4로 감소시킨다. 반면, Gaussian scale-space 생성 과정의 data 접근량은 0.30으로 감소된다. 이는 Gaussian scale-space 생성 과정에서 pre-octave generation 과정이 추가적으로 포함되기 때문이다. 세 번째 열에서 Gaussian scale-space 생성 과정의 data 접근량은 0.69(1.1/1.6)으로 감소되며, orientation assignment와 descriptor generation 과정의 data 접근량은 0.47(1.1/1.6*1.1/1.6)으로 감소된다. Pre-octave generation 과정을 고려하였을 때, 추가적인 data 접근량(α로 표시)가 필요하다. 이 값은 약 0.19이며, 마지막 열의 최종 data 접근량에 포함된다. 네번째 열에서 제안하는 SIFT의 feature 개수는 0.54로 감소된다. 이 값은 그림 7-12의 image set에서 평균적으로 측정된 것이다. 이에 따라, orientation assignment와 descriptor generation 과정에서의 data 접근량은 0.54로 감소된다. 최종 비율은 두 번째부터 네 번째 열의 비율 값의 곱으로 계산된다. Gaussian scale-space 생성 과정의 data 접근량은 0.26으로 감소되며, DoG generation부터 gradient generation까지의 과정은 0.25로 감소된다. Orientation assignment와 descriptor generation 과정의 data 접근량은 0.25로 감소된다.

표 7-4 다양한 구현에 따른 SIFT 처리 시간

Function	Processing Time (ms)			
	CPU	GPU	+ SSR	+ OpenCL optimization
Packing	-	0.71	0.72	0.72
Pre-octave generation	-	-	2.84	2.38
Gaussian scale space construction	1579.98	24.58	5.64	3.47
DoG generation	53.89	6.49	1.69	1.36
Feature detection	312.24	28.24	8.17	6.86
Feature transfer	-	10.28	1.06	0.62
Feature Serialization	-	20.66	3.56*	-
Gradient Generation	563.63	19.56	5.09	-
Orientation Assignment	177.66	9.61	4.15	-
Descriptor generation	1,368.89	39.94	13.06	19.93
Total	4,056.29	160.07	45.98	35.34

* Time processed in CPU

표 7-5 Feature 검출 algorithm들의 평균적인 처리 시간

Algorithms	Detector/image (ms)	Descriptor/feature (μs)
OR_SIFT	1946.11	695
S4O2_SIFT	754.37	228
DS_SIFT	460.11	655
SURF	737.43	523
BRISK	58.33	37
ORB	53.96	85
SSR_SIFT	465.50	321
SSR_SIFT (GPU)	15.41	12

표 7-4는 optimization 방법에 따른 처리시간을 비교한 것이다. 처리 시간은 그림 7-12의 image set으로 측정되었으며, image의 해상도는 1280x720로 모두 변경되었다. 첫 번째 열은 SIFT의 함수들을 나타내며, 두 번째부터 다섯 번째 열은 평균적인 처리 시간을 나타낸다. GPU 구현에서 이전 연구[39]에서 제시된 optimization 방법들이 추가적인 속도 향상을 위하여 적용되었다. 적용된 기법들은 on-the-fly-code 생성, efficient packing이다. 이전 연구의 contribution 중에서 workload partitioning은 GPU 수행에 적합하지 않기 때문에 적용되지 않았다. 네 번째 열의 제안하는 scale-space 단순화 방법(SSR)은 세 번째 열의 GPU 구현에 적용된 것이다. 다섯 번째 열의 결과는 네 번째 열의 구현에서 OpenCL optimization 방법이 추가적으로 적용된 것이다. CPU에서 동작하는 SIFT의 수행 시간은 4,056.29ms이며, 평균적으로 3,035 feature가 검출되었다. SIFT가 GPU에서 수행될 때, 전반적인 처리 시간은 160.07ms로 감소되었다. 제안하는 SSR 방법이 적용되었을 때, 처리 시간은 45.98ms로 감소되었으며, 평균적으로 1,621 feature가 검출되었다. OpenCL 최적화 방법이 추가적으로 적용되었을 때, 전반적인 처리 시간은 35.34ms로 감소되었다. 기존 연구와 SSR_SIFT의 처리 속도는 세 번째와 다섯 번째 열을 비교함으로서 알 수 있다. 표 7-4에서와 같이, 제안하는 SIFT의 처리 속도는 기존의 GPU 구현보다 4.53배 빠르다.

Mobile device에서 동작하는 다양한 feature 검출 algorithm의 속도 측정을 위하여 추가적인 실험들이 진행되었다. 네 가지 다른 SIFT 구현과 세 가지의 non-SIFT feature 검출 알고리즘과 비교를 진행하였다: Original SIFT, Jiang et al.[37]가 제시한 scale-space 단순화 방법, original SIFT를 down-sampled image에서 수행한 것, 제안하는 방법, SURF[6], BRISK[16], ORB[17]. LP_SIFT[38]은 실험 parameter들이 제시되어 있지 않아 실험에서 비교하지 않았다. Algorithmic-level에서 Jiang et al이 제시한 방법은 SIFT parameter의 변경과 SIFT descriptor의 변경이다. 본 논문의 주요 내용은 scale-space 생성

과정 이므로, Jiang et al.의 두 가지 contribution 중에서 scale-space parameter의 변경만 비교를 위하여 구현되었다. 또한, Jiang et al.의 SIFT descriptor 변경은 task-level 병렬화를 위한 방법이기 때문에, GPU에 적용되기에는 적합하지 않다. Jiang의 SIFT 구현에서 octave의 개수는 2개이며, 각 octave마다 scale의 개수는 4개이다. 각 octave에서 Gaussian image의 σ 값들은 각각 1.1, 1.3, 1.6, 2.0이다. Contrast threshold 값은 0.03으로 setting되어있다. Down-sampled image을 이용한 SIFT의 경우, input image의 가로, 세로 크기가 반으로 감소되었다. SURF, BRISK, ORB의 경우, OpenCV library의 algorithm들이 사용되었다.

표 7-5는 mobile CPU에서 동작하는 다양한 feature 추출 알고리즘들의 처리 시간을 보여준다. Mobile GPU에서 동작하는 SSR_SIFT의 처리 시간도 표의 가장 아래 행에 제시하였다. OR_SIFT는 original SIFT를 나타내며, S4O2_SIFT는 Jiang et al.이 제시한 방법을, DS_SIFT는 image down-sampling 방법을, SSR_SIFT는 제안하는 scale-space 감소 방법을 나타낸다. 두 번째 열은 feature를 검출하는데 걸리는 시간을 나타내며, 세 번째 열은 각각의 feature에 대하여 descriptor를 생성하는데 걸리는 시간을 나타낸다. OR_SIFT의 처리 시간은 SURF보다는 느리지만, SSR_SIFT의 처리 시간은 OR_SIFT 뿐만 아니라 SURF보다도 빨랐다. 반면, SSR_SIFT는 BRISK와 ORB보다는 느렸다. SSR_SIFT가 GPU에서 동작할 때, 제일 아래 행과 같이 처리시간이 크게 감소되었다. LP_SIFT와 SSR_SIFT의 연산량을 비교하였을 때, SSR_SIFT의 Gaussian scale-space 생성 과정은 LP_SIFT의 Gaussian scale-space 생성 과정보다 연산량이 3.25배 더 많았다. 하지만, SSR_SIFT에서 DoG, feature detection, gradient generation 과정의 연산량은 LP_SIFT의 30% 수준이었으며, 나머지 동작에서도 68%~76% 수준이었다. SSR_SIFT의 matching 정확도는 LP_SIFT보다 28% 더 높았다.

Mobile GPU에서 다른 SIFT 단순화 방법의 구현 적합도는 지금부터 보일 것이다. Jiang et al.[37]이 제시한 SIFT 연산 방법은 S4O2_SIFT의 단순화된

parameter를 사용하며 효율적인 hardware 구현을 위해서 task-level 병렬화를 달성하였다. 그러나, task-level 병렬화는 hardware 구현에서 효율적인 방법이며, GPU 연산에서는 효율적이지 않다. 이는 GPU가 data-level 병렬화에 최적화되어있기 때문이다. 그러므로, Jiang et al.[37]이 제시한 방법은 GPU 연산에 적합하지 않다. LP_SIFT[38]와 original SIFT의 차이는 scale-space 생성 방법이다. LP_SIFT는 scale-space를 생성하기 위하여 SURF와 유사한 integral image를 사용한다. 그러나, integral image의 생성 과정은 image의 왼쪽 위부터 오른쪽 아래 pixel들을 반복해서 더하는 과정이 필요하며, 이는 GPU의 병렬 처리를 어렵게 한다.

Matching 성능을 평가하기 위하여, 네 가지 실험이 수행되었다. 첫째는 feature의 절대 scale을 나열하고, 각 scale 마다 정확히 matching의 개수를 계산하였다. 이 실험으로부터, original SIFT와 제안하는 SIFT의 scale-space 유사도를 측정할 수 있다. 둘째로, feature 개수를 측정하였다. 셋째로, image의 transform 정도에 따라 matching score를 측정하였다. Matching score는 검출된 feature 중에서 정확히 matching된 feature의 비율로 정의된다. 넷째로, precision과 recall을 측정하였다. Precision과 recall은 다음과 같이 정의된다:

$$precision = \frac{\#correct\ matches}{\#correct\ matches + \#false\ matches} \tag{9-7}$$

$$Recall = \frac{\#correct\ matches}{\#correspondences} \tag{9-8}$$

표 7-6은 네 가지 SIFT 구현의 scale-space 유사도를 보여준다. Tree (blur), Bark (scale), Wall (view), Leuven (illumination) image set을 사용하여 blur, scale, view

및 밝기 변화에 대한 강인함을 측정하였다. 각 image set에서 첫 번째와 네 번째 image가 matching되었을 때, 네 번째 image에 대한 correct match의 개수를 scale에 따라 보였다. 첫 번째 열은 σ의 변화에 따른 absolute scale을 나타낸다. 각 행은 σ 의 범위를 나타낸다. Scale이 증가할 때 마다, σ 값은 $2^{1/3}$ 배 증가한다. DS_SIFT의 경우, original SIFT의 관점에서 feature의 scale을 측정하기 어렵다. 그러므로 이 표에서는 descriptor patch 크기가 σ 제곱에 비례한다는 전제에 따라 DS_SIFT의 scale들이 descriptor patch의 크기에 따라 결정된다. Down-sampled image에서 검출된 feature들의 patch의 가로, 세로 길이가 두 배가 되었으며, 커진 patch 크기에 해당하는 scale이 해당 feature의 scale로 사용되었다. OR_SIFT에서는 $\sigma = 2.02$ 인 scale부터 feature들이 존재한다. S4O2_SIFT에서는 σ 값이 1.3과 2.6인 scale에서만 feature들이 검출된다. DS_SIFT에서는 $\sigma = 4.03$ 인 scale부터 feature들이 존재하며, SSR_SIFT에서는 $\sigma = 2.77$인 scale부터 feature들이 검출된다.

일반적으로, scale이 감소되면 feature의 개수가 증가한다. 그리고 이것은 matching되는 feature 개수를 증가시킨다. 두 번째 열의 blur 변화에서는 matching된 feature 중에서 high-scale feature의 비율이 높게 나타났다. 세 번째 열의 scale 변화에서는 feature들이 대부분 low-scale에서 matching되었다. View와 밝기 변화의 경우, Wall image set에서는 low-scale feature가 중요하였으며, Leuven image set에서는 high-scale feature가 중요하였다. S4O2_SIFT는 high scale에서 feature를 검출하지 않기 때문에, 두 번째와 세 번째 열에서도 볼 수 있듯이 high-scale feature들이 중요한 image set에서는 matching 개수가 적었다. DS_SIFT에서는 OR_SIFT에서 하위 세 개의 scale에 해당하는 feature들이 추출되지 않았다. Scale 변화 정도가 커지거나 low-scale feature가 중요해질수록 matching 개수는 크게 감소되었다. 따라서, image의 작은 부분에 대한 인식률은 크게 하락된다. SSR_SIFT에서는 OR_SIFT에서 가장 낮은 scale에 해당하는 feature만 추출되지 않는다. 따라서, SSR_SIFT는 낮은 scale에서 가장 적은

matching 개수 감소를 보인다. 그러므로, SSR_SIFT는 OR_SIFT와 가장 유사하다고 할 수 있다.

표 7-6 다양한 SIFT 구현에서 정확히 matching된 feature의 개수

Scale	Blur (Tree)				Scale (Bark)				View (Wall)				Illumination (Leuven)			
	OR	S4O2	DS	SSR	OR	S4O2	DS	SSR	OR	S4O2	DS	SSR	OR	S4O2	DS	SSR
1.14 - 1.43	-	0	-	-	-	0	-	-	-	68	-	-	-	39	-	-
1.43 - 1.81	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1.81 - 2.28	28	-	-	-	62	-	-	-	96	-	-	-	72	-	-	-
2.28 - 2.87	43	55	-	31	45	0	-	31	61	56	-	45	63	18	-	48
2.87 - 3.62	33	-	-	21	22	-	-	21	29	-	-	21	51	-	-	35
3.62 - 4.56	21	-	15	28	13	-	10	6	17	-	14	14	26	-	20	32
4.56 - 5.74	19	-	20	21	4	-	4	3	13	-	7	6	18	-	22	15
5.74 - 7.23	18	-	22	5	0	-	3	2	4	-	8	3	14	-	8	10
7.23 - 9.11	10	-	8	8	0	-	2	0	3	-	1	4	5	-	10	5
9.11 - 11.48	7	-	7	2	0	-	1	0	3	-	4	2	11	-	11	9
11.48 - 14.46	5	-	5	5	0	-	0	0	1	-	1	0	5	-	4	3
14.46 - 18.22	-	-	4	3	-	-	0	0	-	-	0	0	-	-	1	4
18.22 - 22.96	-	-	3	-	-	-	0	-	-	-	0	-	-	-	2	-
22.96 - 28.93	-	-	3	-	-	-	0	-	-	-	0	-	-	-	1	-
Total	184	55	87	124	146	0	20	63	277	124	35	95	265	57	79	161

그림 7-13은 다양한 SIFT 구현에 대하여 8개의 image set에서 feature 개수를 보여준다. 각 image set은 여섯 개의 image로 구성되며, 가로축은 transform 정도를 나타낸다. SSR_SIFT와 DS_SIFT의 contrast threshold는 original SIFT와 같다. S4O2_SIFT에서 이 threshold는 0.03으로 설정하였다. S4O2_SIFT에서는 0.03의 높은 contrast threshold가 낮은 scale에서 feature가 추출되지 못하게 한다. 그림 7-13과 같이 낮은 contrast image에서는 feature가 거의 추출되지 않는다. 마찬가지로, blurred image에서도 feature의 개수가 크게 감소된다. Blurring 정도가 증가할수록 low-scale feature가 검출되지 않기 때문이다. 전반적으로, image 특징에 따라서 feature의 개수가 크게 변한다. DS_SIFT에서는 OR_SIFT와 비교하여 feature 개수가 75% 감소되었다. 추출되지 않은 feature는 OR_SIFT의 가장 낮은 세 개의 scale에 속하며, 이는 image의 scale이 변화할 때 matched feature의 개수를 감소시킨다. SSR_SIFT에서는 feature 개수가 OR_SIFT 대비 46% 감소되었다. 이는 OR_SIFT의 가장 낮은 scale에서 feature가 추출되지 않았기 때문이다. 다른 방법과 비교하여, feature 개수 감소가 크지 않으며, image 변형에 관계없이 충분한 개수의 feature가 추출되었다.

Non-SIFT algorithm과의 feature 개수도 비교하였다. 공평한 비교를 위한 이러한 algorithm에서 추출되는 평균 feature 개수가 SSR_SIFT의 경우와 유사해지도록 feature detection threshold를 조절하였다. 그림 7-14 (a), (b)와 같이, SURF는 low-contrast image에서 많은 개수의 feature를 추출하지 않는다. 게다가, blurring 정도가 증가할수록, feature의 개수가 급격히 감소된다. BRISK와 ORB의 feature 개수는 image에 따라 변화가 크다. SSR_SIFT의 경우, 추출되는 feature 개수가 image 특징에 크게 영향을 받지 않았다. Leuven image set에서만 적은 개수의 feature가 추출되었는데, 이 test image에서는 그림 7-14와 같이 모든 algorithm들에서 추출되는 feature 개수가 적다.

Matching score와 precision-recall을 측정하기 위하여, 모든 algorithm에 대한 threshold 값을 조정하여 유사한 feature 개수가 추출되도록 하였다. 일반적으로,

DS_SIFT에서는 image 크기가 작기 때문에, 가장 적은 수의 feature가 추출된다. 그러므로, 다른 algorithm의 threshold를 조절하여 DS_SIFT와 feature 개수가 유사해지도록 하였다. 결과적으로, 대부분의 image에 대해서 약 500개의 feature가 추출되었다. 그림 7-15는 8개의 image set에서 matching score를 보여준다. 가로축은 transform 정도를 나타내며, 세로축은 matching score를 나타낸다.

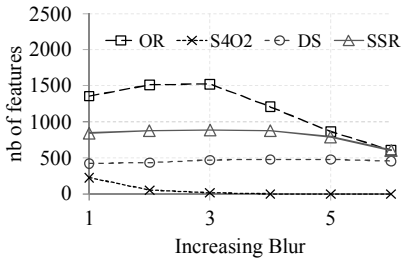
SSR_SIFT의 전반적인 matching score는 OR_SIFT를 제외하고 다른 algorithm보다 대체로 높았다. Blur 변화에 대해서 SSR_SIFT의 matching score는 OR_SIFT의 matching score보다 높았다. 이는 SSR_SIFT에서 feature의 scale이 더 높으며, 이는 feature의 reliability를 증가시키기 때문이다. 그러나, scale이나 view 변화에 대해서는 SSR_SIFT의 matching score가 OR_SIFT의 matching score보다 약간 낮았다. 이는 OR_SIFT에서 scale selection이 더 정확하기 때문이다. DS_SIFT의 matching score는 그림 7-15 (a), (b), (e)에 대해서만 SSR_SIFT보다 높았다. Blur 변화의 경우 high-scale feature의 비율이 증가하는데, DS_SIFT의 matching score는 high-scale feature가 추출되었을 때만 높았다. S4O2_SIFT의 전반적인 matching score는 SSR_SIFT보다 낮았다. 특히, image의 변화 정도가 강해질수록, matching score는 더 낮아졌다.

SSR_SIFT는 그림 7-15 (a) 결과를 제외하고 SURF보다 더 높은 matching score를 보여주었다. 그림 7-15 (a)에서 SURF의 matching score는 SSR_SIFT보다 약 15% 높았다. 그림 7-15 (b), (c), (f), (g)에서 SSR_SIFT의 matching score는 SURF보다 5% 높았다. 나머지 image에서는 SSR_SIFT와 SURF의 matching score가 유사하였다. ORB, BRISK와 비교하여 SSR_SIFT는 그림 7-15 (a) 결과를 제외하고 비슷하거나 더 높은 matching score를 보여주었다.

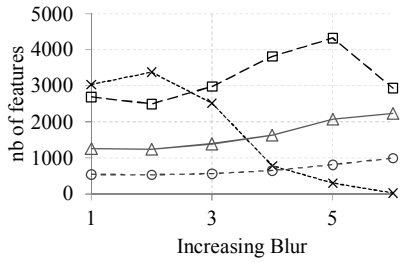
그림 7-16은 각 sub-image set에서 첫 번째와 세 번째 image가 matching되었을 때, precision-recall graph를 보여준다. 가로축은 precision 정도를 나타내며, 세로축은 recall 정도를 나타낸다. Recall-precision curve가 왼쪽 위로 치우칠수록, 해당 algorithm의 성능이 더 좋은 것이다. SSR_SIFT의 matching 정확도는

OR_SIFT보다 약간 낮거나 유사하였다. 이는 original SIFT의 scale selection이 더 정확하기 때문이다. S4O2_SIFT와 비교할 때, SSR_SIFT는 그림 7-16 (b), (d)의 경우를 제외하고 더 좋은 precision-recall curve를 보여준다. 그림 7-16 (e), (f)에서 보인 것과 같이 S4O2_SIFT의 matching 정확도는 scale 변화에서 특히 낮게 나타났다. 이는 S4O2_SIFT의 scale 개수가 매우 적기 때문이다. 그림 7-16 (a), (b), (g)와 같이 DS_SIFT는 blur와 light 변화에서 SSR_SIFT보다 더 좋은 precision-recall curve를 보여주었다. 이는 DS_SIFT에서 feature의 scale이 높으며, 높은 scale에서는 down-sampling에 의한 image 저하 정도가 상대적으로 낮기 때문이다. 그러나, 다른 sub-image set에서는 high scale의 중요도가 감소된다. 그림 7-16 (e)의 경우, high precision에서는 DS_SIFT의 recall이 SSR_SIFT보다 낮았다. 그림 7-16 (c)와 (h)에서는 DS_SIFT의 matching 정확도가 크게 하락하였다.

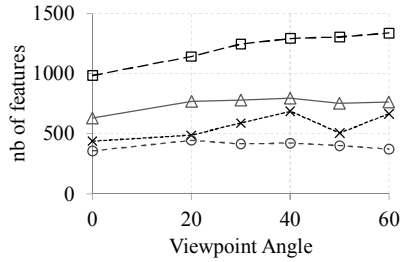
그림 7-16 (c), (e), (h)에서 SSR_SIFT의 recall은 SURF보다 더 높게 나타났다. 그림 7-16 (a), (d), (f)에서 SURF는 high precision에서 낮은 recall을 보여주었다. 그림 7-16 (g)와 같이 SURF의 precision-recall curve는 illumination 변화에 대해서만 더 높게 나타났다. BRISK, ORB와 SSR_SIFT를 비교할 때, SSR_SIFT는 그림 7-16 (c), (e), (f)에서 더 높은 precision-recall을 보여주었다. 그림 7-16 (a), (d)의 결과와 같이, BRISK는 high precision에서 낮은 recall을 보여준다. 그림 7-16 (b)는 BRISK가 SSR_SIFT보다 precision-recall이 높은 유일한 경우이다. ORB와 비교하였을 때, SSR_SIFT는 그림 7-16 (a)를 제외하고 더 높은 precision-recall 결과를 보여주었다.



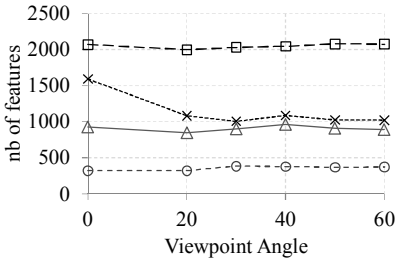
(a) Bikes



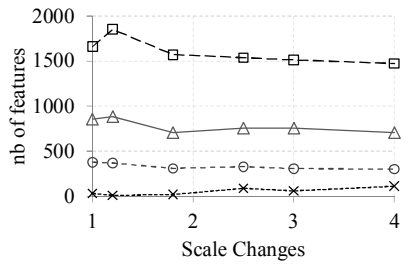
(b) Trees



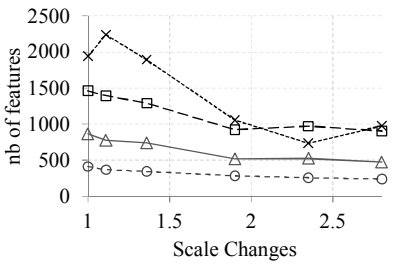
(c) Graf



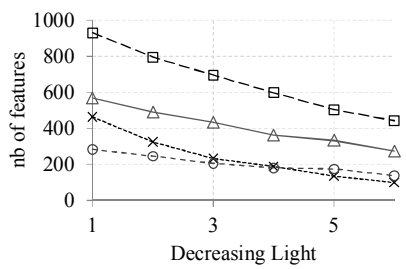
(d) Wall



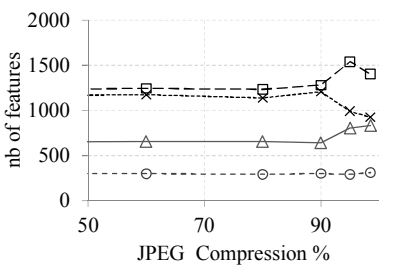
(e) Bark



(f) Boat

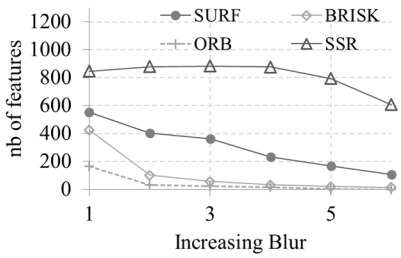


(g) Leuven

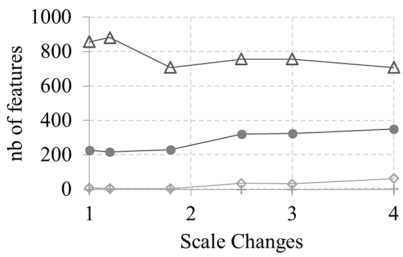


(h) UBC

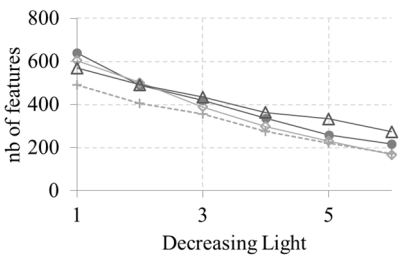
그림 7-13 SIFT algorithm들의 feature 개수



(a) Bikes

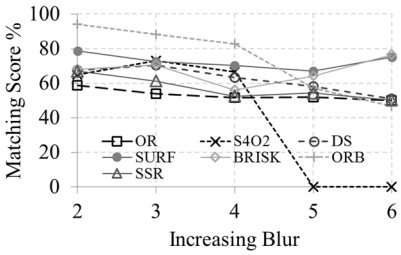


(b) Bark

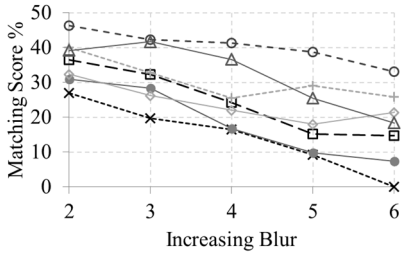


(a) Leuven

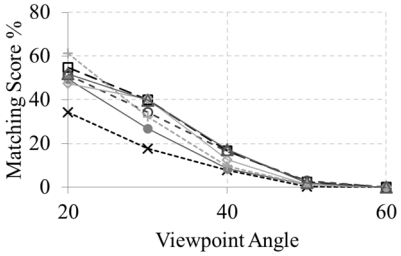
그림 7-14 Non-SIFT algorithm과의 feature 개수 비교



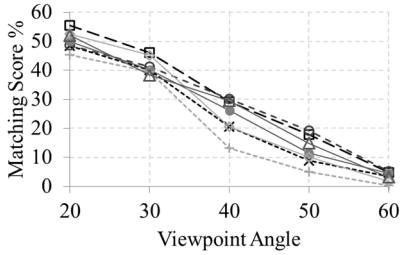
(a) Bikes



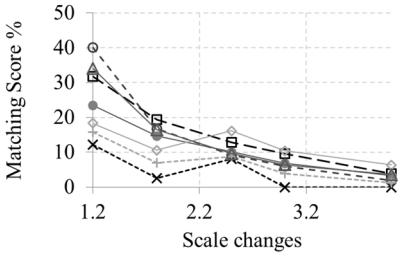
(b) Trees



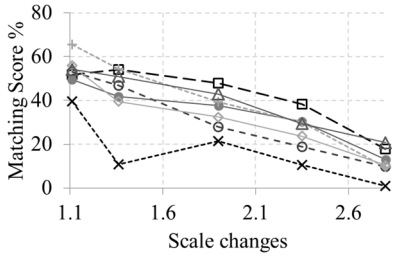
(c) Graf



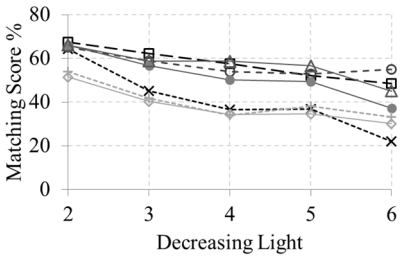
(d) Wall



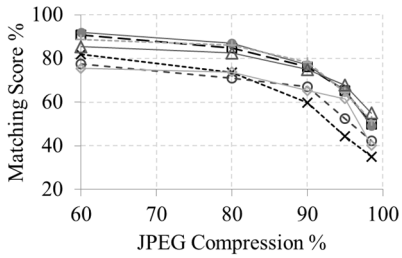
(e) Bark



(f) Boat

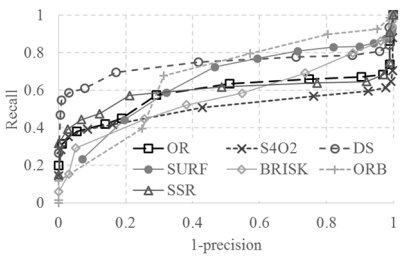


(g) Leuven

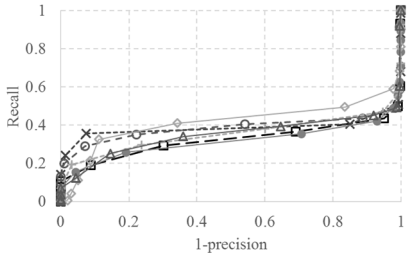


(h) UBC

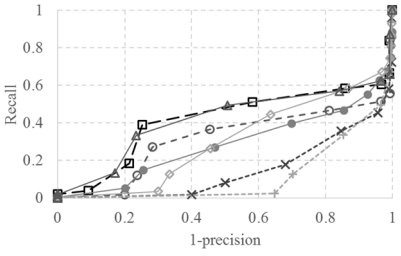
그림 7-15 다양한 algorithm의 matching score



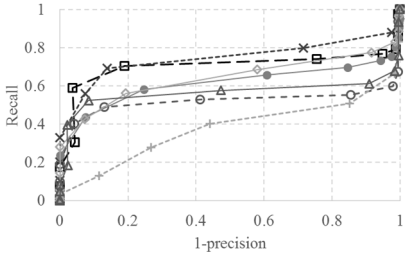
(a) Bikes



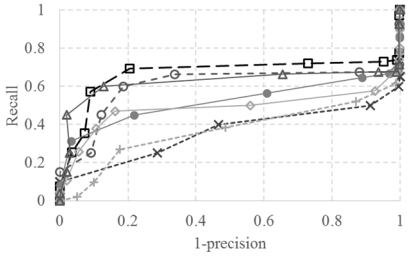
(b) Trees



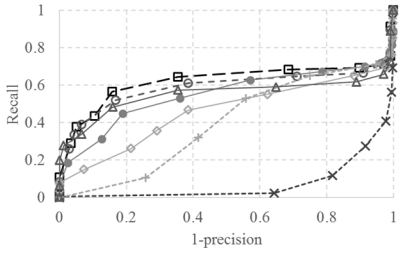
(c) Graf



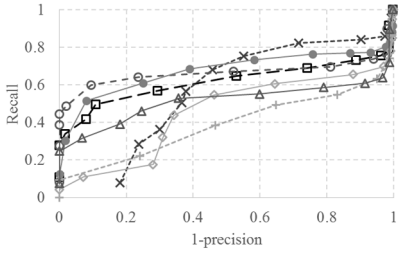
(d) Wall



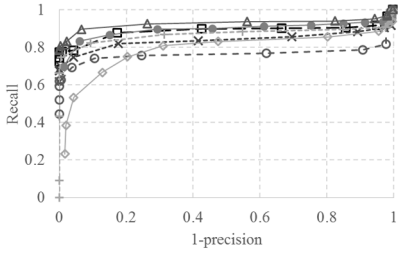
(e) Bark



(f) Boat



(g) Leuven



(h) UBC

그림 7-16 다양한 algorithm의 precision-recall curve

.

제8장 결 론

본 논문은 feature matching algorithm의 효율적인 GPU 병렬화를 위한 네 가지 방법을 제시하였으며, feature extraction algorithm의 효율적인 병렬화를 위한 한 가지 방법을 제시하였다.

첫째, high-order graph matching의 symmetric 특징을 이용하여 GPU에서 적합하지 않은 memory 접근 pattern을 제거하였다. High-order graph matching algorithm은 전반적인 처리 속도를 하락시키는 memory collision 문제를 겪는다. 본 연구에서 이 memory collision은 affinity tensor의 supersymmetric 성질을 이용하여 제거되었으며, memory collision 제거로 약 3배의 속도 향상을 얻었다. 최신 parallelizing solution들과 비교하였을 때에도, 제안하는 방법은 shared memory에서 reduction을 이용한 방법보다 20% 빨랐으며, warp shuffle을 이용한 reduction 방법에서도 20% 빨랐다.

둘째, memory 사용량 감소를 위한 Approximate Spectral Matching algorithm을 적용하고, 이를 GPU에서 효율적으로 처리하기 위한 병렬처리 방법을 제시하였다. Approximate Spectral Matching 은 feature 사이의 geometry 정보를 approximation함으로써, similarity의 개수를 감소시켰다. 그러나, 이 방법은 sparse tensor를 이용해야 하며, 이는 GPU에서 효율적으로 처리되기 어려운 memory 접근 pattern을 발생시킨다. 본 연구에서는 이 memory 접근 pattern이 나타나는 연산 과정을 graph matching algorithm에서 분리하고, 분리된 연산을 GPU에서 효율적으로 처리하기 위한 thread structure를 제시하였다. 제안하는 방법을 사용하여 최대 20배의 속도 향상을 얻었으며, 기존 sparse data 처리 연구보다도 최대 10배의 속도 향상을 얻었다.

셋째, memory 사용량을 추가적으로 감소시키기 위한 approximation 방법을

제시하였다. 기존의 Approximate Spectral Matching Algorithm이 reference feature set만 approximation한다는 고려하여, 본 연구에서는 query feature set도 함께 approximation하는 방법을 제시하였다. 제안하는 방법을 사용함으로써, similarity 개수가 크게 감소되며 memory 사용량이 100배 이상 감소되었다. 또한, shared memory를 이용하여 제안하는 approximation 방법을 GPU에서 효율적으로 처리하기 위한 방법을 제시하였으며, 최대 4배에 가까운 처리 속도 향상을 보였다.

넷째, 제안하는 approximation 방법의 GPU utilization 향상을 위한 tuple clustering 방법을 제안하였다. 본 연구에서는 tuple clustering 방법을 이용하여 sparse data structure들의 non-zero element 개수를 유사하게 만든다. 이로써 thread-block 사이의 workload가 균일 해진다. 또한, score 기여도 기반 clustering 방법을 제시하여 matching 정확도 하락을 최소화하였다. 제안하는 방법을 사용하여 유사한 matching 정확도에서 최대 6배에 가까운 처리 속도 향상을 보였다.

다섯째, 본 논문은 SIFT 생성 과정에서 GPU에서 data 접근량을 감소시키는 방안을 제안하였다. Scale-space 생성 과정에서 image 크기뿐만 아니라 Gaussian filter size의 standard deviation 값을 감소시킴으로써 연산 복잡도가 크게 감소되었다. 제안하는 SIFT는 mobile GPU에서 구현되었으며, HD image에서 28.30fps의 처리속도를 보였다. 부정확한 scale selection을 피하기 위하여, 제안하는 방법은 scale space에서 scale을 더 높이게 되었다. 그럼에도 불구하고, 단순한 down-scaling 방법과 비교하여 더 많은 scale을 처리하면서도 유사한 처리 속도를 보여주었다.

빠른 처리 속도로 인하여 제안하는 방법은 CPU 구현 보다 훨씬 많은 feature를 처리할 수 있게 되었다. 이것은 image의 더 많은 부분들에서 correct match들이 발견될 수 있게 하며, high-order graph matching algorithm을 사용하는 application들의 안정성을 향상시킨다. 제안하는 방법들은 algorithmic-level에서

구현된 논문으로 CUDA나 OpenCL의 특정한 version에서만 지원되는 함수들도 요구하지 않는다. 그러므로, 제안하는 방법은 SIMD-type many-core architecture의 다양한 GPU machine에 적용될 수 있다.

참 고 문 헌

[1] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *Int. J. Comput. Vis.*, vol. 60, no. 2, pp.91-110, 2004.

[2] Y. Zheng, and D. Doermann, “Robust point matching for nonrigid shapes by preserving local neighborhood structures,” *IEEE Trans. Pattern. Anal. Mach. Intell.*, vol. 28, no. 4, pp. 643-649, 2006.

[3] C. Schmid, and R. Mohr, “Local grayvalue invariants for image retrieval,” *IEEE Trans. Pattern. Anal. Mach. Intell.*, vol. 19, no. 5, pp. 530-534, 1997.

[4] J. Cheng, C. Leng, J. Wu, H. Cui, and H. Lu, “Fast and accurate image matching with cascade hasing for 3D reconstruction.” *In Proc. IEEE Conf. Comput. Vis. Pattern. Anal*., pp. 1-8, 2014.

[5] L. Xu, J. Jia, and Y. Matsushita, “Motion detail preserving optical flow estimation,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 9, pp. 1744-1757, 2011.

[6] H. Bay, T. Tuytelaars, and L. V. Gool, "SURF: Speeded up Robust Features," *in Proc. European Conf. Comput. Vis.*, 2006, pp. 404-417.

[7] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: Binary robust independent elementary features," *in Proc. European Conf. Comput. Vis.,* pp. 778-792, 2010.

[8] O. Duchenne, F. bach, K. In-So, and J. Ponce, “Tensor-based high-order graph matching ” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 12, pp. 2383-2395, 2011.

[9] J. Lee, M. Cho, and K. M. Lee, “Hyper-graph matching via reweighted random walks,” *IEEE Int. Conf. Comput. Vis. Pattern. Recog*., pp. 1633-1640, 2011.

[10] S. Park, S.-K. Park, and M. Hebert. “Fast and scalable approximate spectral matching for higher order graph matching,” *IEEE Trans. Pattern. Anal. Mach. Intell.,* vol. 36, no. 3, pp. 479-492, 2014.

[11] M. Chertok, and Y. keller, “Efficient high order matching,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 12, pp. 2205-2215, 2010.

[12] R. Zass, and A. Shashua, “Probabilistic graph and hypergraph matching,” *IEEE Int. Conf. Comput. Vis. Pattern. Recog*., 2008.

[13] Z.-Q. Cheng, Y. Chen, R. R. Martin, Y.-K. Lai, and A. Wang, “SuperMatching: feature matching using supersymmetric geometric constraints,” *IEEE Trans. Visual. Comput. graphics*, vol. 19, no. 11, pp. 1885-1894, 2013.

[14] R. Zhang; W. Wang, "Second- and High-order Graph Matching for Correspondence Problems," *IEEE Trans. Circuits Syst. Video Technol.*, vol.PP, no.99, pp.1-1.

[15] X. Shi, H. Ling, W. Hu, J. Xing, Y. Zhang, "Tensor Power Iteration for Multi-Graph Matching,” IEEE. Conf. Comput. Vis. Pattern. Recog., pp. 5062-5070, 2016.

[16] Chli, and R. Siegwart, “BRISK: Binary robust invariant scalable keypoints,” *in Proc. IEEE Int. Conf. Comput. Vis.*, 2011, pp. 2548–2555.

[17] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, “ORB: An efficient alternative to SIFT or SURF,” in Proc. *IEEE Int. Conf. Comput. Vis*., 2011, pp. 2564–2571.

[18] E. Rosten, R. Porter, and T. Drummond, "Faster and better: A machine learning approach to corner detection," *IEEE Trans. Pattern. Anal. Mach. Intell*., vol. 32, no. 1, pp. 105-119, Jan., 2010.

[19] J.Matas, O. Chu,. M. Urban, and T. Pajdla, “Robust wide-baseline stereo from maximally stable extremal regions,” *Image and vision computing*, vol. 22, no. 10, pp. 761-767, 2004.

[20] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, “K-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching,” *IEEE Int. Conf. Image. Process.*, pp. 3757-3760, 2010.

[21] Q. Kuang, and L. Zhao, "A practical GPU based kNN algorithm," *in Proc. Int. computer Sci. Computational Technol.*, pp. 151-155, 2009.

[22] A. Chariot, R. Keriven, "GPU-boosted online image matching," *Int. Conf. Pattern. Recog.*, pp. 1-4, 2008.

[23] G.H. Golub and C.F.V. Loan, *Matrix Computations*, Johns Hopkins Univ. Press, 1996.

[24] L. De Lathauwer, B. De Moor, and J. Vandewalle, "On the best rank-1 and rank-(r1,r2,. . .,rn) approximation of higher-order tensors," *SIAM J. Matrix Anal. Appl.,* 21(4):1324–1342, 2000.

[25] J. Heinly, E. Dunn, and J. M. Frahm, "Comparative evaluation of binary features," *in Proc. European Conf. Comput. Vis.,* pp. 759-773, 2012.

[26] J. Luitjens, Faster Parallel Reductions on Kepler, http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler, Feb., 2014.

[27] L. Howes, The OpenCL specification (Version 2.1, Document Revision 23), https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf, Nov., 2015.

[28] O. Chum and J. Matas, "Matching with PROSAC-progressive sample consensus," *IEEE Computer Society Conf. Comput. Vis. Pattern Anal*., June, 2005.

[29] J. R. Rice, R. F. Boisvert, "Solving elliptic problems using ELLPACK," Springer Verlag, 1984

[30] M. Brown, D. G. Lowe, "Automatic panoramic image stitching using invariant features," Int. J. Comput. Vis., vol. 74, no. 1, pp 59-73, 2007 .

[31] S. Battiato, G. Gallo, G. Puglisi, S. Scellato, "SIFT features tracking for video stabilization," in Proc. ICIAP, 2007, pp. 825-830.

[32] F.-H. Huang, S.-Y. Huang, J.-W. Ker, and Y.-C. Chen, "High-performance SIFT hardware accelerator for real-time image feature extraction," IEEE Trans Circuits Syst. Video Technol., vol. 22, no. 3, pp. 340-351, 2012.

[33] S. Heymann, K. Muller, A. Smolic., B. Frohlich., T. Wiegand., “SIFT implementation and optimization for general-purpose GPU,” in Proc. WSCG, 2007, pp. 317-322.

[34] S. N. Shnha, J.-M. Frahm, “Feature tracking and matching in video using programmable graphics hardware,” Mach. Vis. Appl., vol. 22, no. 1, pp. 207-217, 2011.

[35] C. Wu, SiftGPU : A GPU implementation of scale invariant feature transform (SIFT), 2007, [Online]. Available: http:/cs.unc.edu/~ccwu/siftgpu/.

[36] L. Yao, H. Feng, Y. Zhu, Z. Jiang, D. Zhao, W. Feng, “An architecture of optimized SIFT feature detection for an FPGA implementation of an image matcher,” in Proc. Int. Conf. Field-Programmable Technol,, 2009, pp. 30-37.

[37] J. Jiang, X. Li, and G. Zhang, “SIFT hardware implementation for real-time image feature extraction,” IEEE Trans. Circuits Syst. Video Technol., vol. 24, no. 7, pp. 1209-1220, 2014.

[38] L.-C. Chiu, T.-S. Chang, J.-Y. Chen, and N. Y.-C. Chang, “Fast SIFT design for real-time visual feature extraction,” IEEE Trans. Image Process., vol. 22, no. 8, pp. 3158-3166, 2013.

[39] B. Rister, G. Wang, M. Wu, “A fast and efficient SIFT detector using the mobile GPU,” in Proc. ICASSP, 2013, pp. 2674-2678.

[40] G. Wang, B. Rister, and J. R. Cavallaro, “Workload analysis and efficient OpenCL-based Implementation of SIFT algorithm on a smartphone,” in Proc. GlobalSIP, 2013, pp. 759-762.

[41] E. S. Kim and H.-J. Lee, “A novel hardware design for SIFT generation with reduced memory requirement,” J. Semicond. Technol. Sci., vol. 13, no. 2, pp. 157–169, Apr. 2013.

[42] J. Yum, C. Lee, J.-S. Kim, and H.-J. Lee, “A Novel Hardware Architecture with Reduced Internal Memory for Real-time Extraction of SIFT in an HD Video,” IEEE

Trans. Circuits Syst. Video Technol.

[43] T. Tuytelaars and K. Mikolajczyk, "Local invariant feature detectors: a survey," Foundations and Trends in Computer Graphcis and Vision, vol. 3, no. 3, pp. 177-280, 2008.

[44] K. Mikolajczyk and C. shmid, "Scale & affine invariant interest point detectors," Int. J. Comput. Vis., vol. 60, no. 1, pp. 63–86, 2004.

[45] K. Mikolajczyk and C. Shmid, "A performance evaluation of local descriptors," IEEE Trans. Pattern Anal. Mach. Intell., vol. 27, no. 10, pp. 1615-1630, 2005.

[46] R. Hess. SIFT Feature Detector (Source Code). [Online]. Available: http://blogs.oregonstate.edu/hess/code/sift/

Abstract

Parallelization of Feature Extraction and Matching Algorithm for GPU Architecture

이 철 희(Chulhee Lee)

전기 컴퓨터 공학부(Electrical and Computer Engineering)

The Graduate School

Seoul National University

Feature matching is widely used in the field of computer vision for finding similar features in images. Feature matching algorithms are classified as first-order matching and high-order graph matching based on whether or not geometric information among features is used in matching operations. A commonly used method is first-order matching, which identifies similar features based on information around the features. First-order matching is disadvantageous in that it is sensitive to image ambiguity such as repeated patterns, textures, or unclear areas because it does not consider the geometric information among features. On the other hand, high-order graph matching is a feature-matching algorithm that uses geometric information among features. This algorithm is more robust to repetitive patterns or unclear areas than first-order matching that uses only feature descriptors. However, the processing speed of high-order matching is very slow because of its high computational complexity. This paper presents four parallelization methods for

high-order matching for GPU execution.

First, the memory access pattern that are not suitable for GPU execution is removed by using the symmetric property of the high-order graph matching algorithm. The obstacle for parallelization is the write collision caused by multiple threads that must simultaneously update the data at the same memory location. In high-order matching, multiple formulations of the objective function can generate the same solution. By taking advantage of this property, the proposed algorithm replaces the operation causing write collision with another operation eliminating the collision while generating the same solution.

Second, Approximate Spectral Matching algorithm is applied to reduce memory usage and the efficient parallelization method is proposed for a GPU. High-order graph matching requires very large amount of memory to store similarity values. Approximate Spectral Matching reduces the number of similarity values by approximating the geometry information between features. However, this method uses a sparse tensor, which results in a memory access pattern that is difficult to be efficiently processed on a GPU. In this study, the operation that generates this memory access pattern is separated from the matching algorithm. And a thread structure for the separated operation is presented for efficient processing on a GPU.

Third, an additional approximation method is proposed to further reduce memory usage. When Approximate Spectral Matching algorithm is used, degree of approximation should be low for high matching accuracy. However, when the degree of approximation is low, the number of similarity values are not reduced significantly. Considering that the Approximate Spectral Matching algorithm only approximates the reference feature set, the query feature is also approximated in the proposed method. This approximation greatly reduces the number of similarity values, and the memory usage is not affected by the degree of approximation. In addition, efficient parallelization of the proposed approximation method is presented using a shared memory.

Fourth, tuple clustering is proposed to improve GPU utilization of the proposed approximation method. When the proposed approximation method is used, workload

imbalance occurs between thread-blocks. This causes imbalance in the workload of GPU streaming multiprocessors and greatly degrades utilization of GPU hardware. The reason for the workload imbalance among thread-blocks is that data structures accessed by each thread-block are sparse and the number of non-zero elements in the sparse data structures are different. In this study, the number of non-zero elements of sparse data structures is made similar by using tuple clustering method. In addition, a clustering method based on score contribution is proposed to minimize drop in matching accuracy.

Scale-invariant feature transform (SIFT) is one of the most widely used local features for computer vision. This feature is invariant to a variety of image transforms and distortions such as image scaling and rotation. However, computational requirements for this level of robustness of SIFT are complex, which makes it difficult to run SIFT in real-time. This paper proposes an efficient scheme to optimize the SIFT algorithm for a GPU. It analyzes the conventional scale-space construction step in the SIFT generation, finding that reducing the size of the Gaussian filter and the scale-space image leads to a significant speed-up with only a slight degradation of the quality of the features. Based on this observation, the SIFT algorithm is modified and implemented for real-time execution. Additional optimization techniques are employed for a further speed-up by efficiently utilizing both the CPU and the GPU.

Keywords : High-order graph matching, feature matching, GPU optimization, parallel processing, Scale-invariant feature transform, CUDA

Student Number : 2012-20843