공학박사 학위논문

영상의 로컬 피처를 실시간으로 추출하기 위한 하드웨어 구조

A Novel Hardware Architecture for Real Time Extraction of Local Features

2016 년 8 월

서울대학교 대학원

전기 · 정보공학부

염 주 혁

영상의 로컬 피처를 실시간으로 추출하기 위한 하드웨어 구조

지도 교수 이 혁 재

이 논문을 공학박사 학위논문으로 제출함
2016 년 8 월

서울대학교 대학원
전기 · 정보공학부
염 주 혁

의 공학박사 학위논문을 인준함
2016 년 8 월

위 원 장 채 수 익 (인)

부위원장 이 혁 재 (인)

위 원 최 기 영 (인)

위 원 조 남 익 (인)

위 원 김 진 성 (인)

초 록

컴퓨팅 성능의 비약적인 발전과 보급은 컴퓨터 기술의 적용 분야를 데스크탑에서 스마트폰, 스마트 TV, 승용차 등에 이르기까지 폭넓은 범위로 넓히는 결과를 야기했다. 변화된 환경에서 대중은 기존에 없었던 좀 더 혁신적인 기능을 받아들일 준비가 되었고, 이에 부합하기 위해 Computer vision 기술은 점차 상용화의 길을 걷게 되었다. 물체 인식 및 추적, 3D reconstruction 등 폭넓게 응용될 수 있는 computer vision 기술들은 서로 다른 영상 사이에서 동일한 pixel을 찾는 image matching 기술을 필요로 한다. 관련 연구들 중 영상의 크기가 변하거나 회전하여도 안정적으로 matching이 가능한 Scale-Invariant Feature Transform (SIFT) 알고리즘이 제안되었고, 이후 카메라의 viewpoint 변화에도 강인한 Affine Invariant Extension of SIFT (ASIFT) 알고리즘이 제안되었다. SIFT 및 ASIFT 알고리즘은 image matching의 안정성이 높은 반면 많은 연산량을 요구한다. 이를 실시간 처리하기 위해 specifically designed hardware을 이용한 연산 가속 연구가 진행되고 있다.

본 논문에서는 실시간 (30frames/sec)으로 동작 가능한 ASIFT 하드웨어 구조를 제안한다. 이를 위해 첫번째로 SIFT feature를 실시간으로 연산 할 수 있는 SIFT 하드웨어 구조를 제안한다. SIFT 알고리즘은 널리 사용되는 만큼 많은 수의 가속 하드웨어 구조가 연구되었다. 대부분의 기존 SIFT 하드웨어는 실시간성을 충족시키지만, 이를 위해 과도하게 많은 내부 메모리를 사용하여 하드웨어 비용을 크게 증가시켰다. 이 이슈 사항으로 인해 내부 메모리와 외부 메모리를 혼용하는 새로운 SIFT 하드웨어 구조가 제안되었다. 이 경우 빈번한

외부 메모리 사용은 외부 메모리 latency로 인한 동작 속도 저하 문제를 일으킨다. 본 논문은 이 문제를 해결하기 위해 외부 메모리에서 읽어온 데이터 재사용 방안과, 외부 메모리에 저장하는 데이터에 대한 down-sampling 및 less significant bits 제거를 통한 데이터량 감소 방안을 제안한다. 제안하는 SIFT 하드웨어는 Gaussian image를 외부 메모리에 저장하며, 이 경우 descriptor 생성을 위해 local-patch를 읽어오는데 많은 외부 메모리 접근이 발생한다. 이를 저감하기 위해, 서로 다른 local-patch 상의 중복 데이터를 재사용하는 방안과 이를 위한 하드웨어 구조를 제시한다. 또한 Gaussian image의 데이터량 자체를 줄이기 위해 down-sampling 및 less significant bits 제거 방안을 이용한다. 이때 SIFT 알고리즘의 정확도 감소를 최소화하였다. 결과적으로 본 논문은 기존 state-of-the-art SIFT 하드웨어의 10.93% 크기의 내부 메모리만 사용하며, 3300개의 key-point에 대해 30 frames/sec (fps)의 속도로 동작 가능하다.

ASIFT 알고리즘 연산을 고속으로 수행하기 위해서는 SIFT 하드웨어에 affine transform된 영상을 제공하는 affine transform 하드웨어가 delay 없이 데이터를 제공할 수 있어야 한다. 하지만 일반적인 affine transform 연산 방식을 이용할 경우 affine transform 하드웨어는 외부 메모리에서 원본 영상을 읽어 올 때 불연속적인 주소로 접근하게 된다. 이는 외부 메모리 latency를 발생시키며 affine transform module이 충분한 데이터를 SIFT 하드웨어에 공급해주지 못하는 문제를 야기한다. 이 문제를 해결하기 위하여 본 논문은 SIFT feature의 rotation-invariant한 특성을 이용하여, affine transform 연산 방식을 변경하였다. 이 방식은 ASIFT 알고리즘이 취하는 모든 affine transform 연산을 수행할 때 연속된 외부 메모리 주소로 입력 영상을 접근할 수 있게 해준다. 이를 통해 불필요한 외부 메모리 latency가 크게 감소된다. 제안된 affine transform

연산 방식은 원본 영상을 scaling한 뒤 skewing하는 연산 과정을 거친다. 본 논문은 이 과정에서 scaling된 영상 데이터를 서로 다른 affine transform 연산에 재활용하는 방법을 제안한다. 이는 scaling 연산량을 감소시킬 뿐 만 아니라 외부 메모리 접근량도 감소시킨다. 제안된 방안들로 인한 affine transform 하드웨어의 속도 향상은 SIFT 하드웨어에 대기 없이 데이터를 공급할 수 있게 해주고, 최종적으로 utilization 향상을 통한 ASIFT 하드웨어의 동작 속도 향상에 기여한다. 결과적으로 본 논문에서 제안하는 ASIFT 하드웨어는 높은 utilization으로 동작이 가능하며, 이로 인해 2,500개의 key-point가 검출되는 영상에 대하여 30fps의 동작 속도로 ASIFT 알고리즘을 수행할 수 있다.

주요어: SIFT, ASIFT, Hardware accelerator, Bandwidth optimization, Data reuse, Affine transform hardware

학 번: 2012-30218

목 차

표 목차

그림 목차

제 1 장 서 론

1.1 연구 배경

컴퓨터를 이용한 영상 처리 (Computer vision) 기술은 관련 알고리즘에 대한 지속적인 연구와 컴퓨터의 성능 향상 및 보급에 힘입어 얼굴 인식, 차량 추적과 같은 기능이 일상 생활에 응용되는 수준에 이르렀다. 이러한 변화는 일반인들에게 computer vision 기술에 대한 관심과 기대감을 심어주었고, 이를 충족시키기 위해 좀 더 많은 computer vision 기능들이 상용화를 위해 연구 개발되고 있다.

Computer vision 기술들 중 영상의 영역 적 특징 (Local feature) 을 이용한 영상 정합 (Image matching) 기술은 물체 인식 (Object recognition), 움직임 추적 (Motion estimation), 3 차원 정보 복원 (3D reconstruction) 등 다양한 영상 처리 어플리케이션에서 사용될 수 있는 필수적인 기술이다. 영상의 local feature 란 이웃한 영역과 명도나 색상 등의 특성에 있어서 독특하게 구별될 수 있는 좁은 영역 또는 점을 의미한다 [1]. 예를 들어 물체 경계선의 모서리, 사람 얼굴의 점, 도형의 꼭지점 등이 이에 해당한다. 이는 key-point 라고 부르기도 한다.

Local feature 를 이용한 image matching 은 1) key-point 검출, 2) descriptor vector 계산, 3) feature matching 으로 구분된다 [2]. 동일한 장면이 촬영된 2 장의 image 를 matching 할 때, 우선 각각의 image 에서 key-point 를 추출한다. 하나의 key-point 는 다른 key-point 들과 구별되기 위해 key-point 주변의 일부 영상 (local-patch)를 이용해 descriptor 를 계산한다.

동일한 물체가 두 image 에서 촬영되었다면, 이 물체에서 검출된 key-point 의 local-patch 는 유사할 가능성이 크고, descriptor 역시 유사할 것이다. 이 점을 이용해 두 image 에서 descriptor 가 동일한 두 key-point 쌍을 찾는 feature matching 연산을 수행한다. Matching 된 key-point 쌍은 두 image 에서 동일한 위치의 key-point 의 관계 (correspondence) 정보를 제공한다.

Key-point 는 두 image 사이에서 물체의 크기가 달라지고 (scaling), 회전하거나 (rotation), 카메라의 시점이 움직이거나 (viewpoint change), 영상에서 blurring 이 발생하거나, 밝기가 변하는 (brightness change) 등의 영상 변환 (image transformation)이 발생하여도 안정적으로 동일한 위치에서 검출될 수 있다면 이를 안정적인 key-point 라고 한다. 그리고 좀 더 안정적인 key-point 를 검출하기 위하여 다양한 연구들이 진행되었다 [3-8]. 그 중에서 Lowe 에 의해 제안된 Scale-Invariant Feature Transform (SIFT) 알고리즘은 지금까지 연구된 local feature 들 중 가장 널리 사용된 알고리즘이다 [3]. 왜냐하면 SIFT feature 는 image 가 scaling 이 변하거나 rotation 되는 상황에서도 안정적으로 feature matching 이 가능하기 때문이다. 이를 scale, rotation invariant 하다고 표현한다. 이러한 장점 덕분에 SIFT feature 는 다양한 computer vision application 에서 사용되었으며 [9-11], 변형된 SIFT 알고리즘도 다수 제안되었다 [12-14].

SIFT 알고리즘은 대부분의 image transformation 에 강인하지만 viewpoint change 에는 취약하다. 이런 약점을 보완하기 위하여 Morel *et al.* 은 affine invariant extension of SIFT (ASIFT)를 제안하였다 [15]. ASIFT 알고리즘은 고려 가능한 입력 영상 카메라의 모든 viewpoint 변화를 입력 영상에 다양한 형태의 affine distortion 을 적용하는 방식으로 시뮬레이션 한다. 그 다음,

모든 시뮬레이션 영상에서 SIFT feature 를 추출한다. 이를 통해 ASIFT 알고리즘은 fully affine invariant 한 특성을 획득하게 된다.

SIFT 알고리즘은 안정성이 높은 local feature 생성할 수 있지만, 복잡한 연산이 요구되며 넓은 메모리 공간을 빈번히 사용하기 때문에 이 알고리즘을 실시간으로 처리하기 어렵다. 이 문제를 해결하기 위하여 general-purpose graphic processing unit (GPGPU)를 이용한 연산 가속 연구 [16-21]와 specific designed hardware 를 이용한 연산 가속 연구 [22-29]가 진행되었다. 두가지 접근법 중 hardware 기반 가속 연구들은 GPGPU 기반 연구에 비해 더 빠른 동작 속도와 적은 파워 소모가 가능하였다. 특히 Huang *et al.* 은 효율적인 SIFT 전용 하드웨어 설계를 통해 SIFT 알고리즘의 성능 저하 없이 VGA (640x480) 크기의 영상을 실시간 처리 (33ms/frame)하는 하드웨어를 제안하였다 [26]. 하지만 제안된 하드웨어는 과도한 내부 메모리를 사용하는 문제점을 갖고 있다.

ASIFT 알고리즘은 복잡한 SIFT 알고리즘을 다수의 시뮬레이션 영상에 반복적으로 적용하므로 SIFT 알고리즘보다 월등히 많은 연산을 수행한다. 따라서 ASIFT 알고리즘을 실시간으로 이용하기 위해서는 연산을 고속화 할 필요가 있다. Codreanu *et al.* 은 GPGPU 를 이용하여 ASIFT 알고리즘을 가속화 한 GPU-ASIFT 를 제안하였다 [30]. GPU-ASIFT 는 VGA 크기의 영상을 110ms/frame 의 속도로 처리하였다. 이는 고성능 PC 용 single-core CPU 를 이용한 것에 비해 40 배 빠른 동작 속도지만, 여전히 실시간으로 처리한다고 하기 부족한 속도이며, 고가의 dual-GPU card 를 이용하여 과도하게 비싼 하드웨어 리소스를 사용한 한계가 있다. 반면 SIFT 알고리즘과 같은 전용 하드웨어를 이용한 연산 가속화 연구를 아직까지 발표되지 않았다.

1.2 연구 내용

본 논문에서는 SIFT 및 ASIFT 알고리즘과 하드웨어 구조에 대한 최적화 연구를 통해 실시간 동작이 가능한 합리적 비용의 ASIFT 하드웨어를 제안한다. ASIFT 하드웨어는 크게 나누어 SIFT 연산 하드웨어와 affine transform 하드웨어로, 각각의 하드웨어 대한 고속화 연구가 진행되었다. 이는 아래와 같다.

첫째, 외부 메모리 bandwidth 요구량을 최적화한 새로운 SIFT 하드웨어 구조를 제안한다. 기존에 제시된 대부분의 SIFT 하드웨어는 실시간 연산은 가능하지만, 이를 위해 과도하게 많은 내부 메모리를 사용하여 하드웨어의 비용을 증가시켰다 [22-26]. 반면 Kim *et al.* 에 의해 제안된 SIFT 하드웨어는 내부 메모리의 일부를 외부 메모리로 대체하여 하드웨어 비용을 낮춘 새로운 SIFT hardware 구조를 제안하였다 [27-29]. 하지만 이는 외부 메모리 사용량을 크게 증가시켰고, 결과적으로 외부 메모리 latency 로 인한 속도 저하를 야기하였다. 이 문제를 해결하기 위하여 본 논문은 외부 메모리에서 읽어온 데이터 재사용 방안과, 외부 메모리에 저장하는 데이터에 대한 down-sampling 과 less significant bits 제거를 통한 데이터량 감소 방안을 제안한다.

제안된 SIFT 하드웨어는 SIFT 알고리즘이 사용하는 데이터 메모리 중에서 가장 큰 용량을 필요로 하는 Gaussian image buffer 를 외부 메모리에 위치시켰다. 이 경우, Gaussian image buffer 에서 descriptor 생성을 위한 local-patch 를 읽어오는데 많은 외부 메모리 접근이 발생한다. 이때 서로 다른 local-patch 사이에는 중복된 데이터가 존재한다. 본 논문은 앞선 local-patch 를 위해 읽어온 데이터를 다음 local-patch 연산에 재활용한다. 재활용될 수 있는 데이터의 양을 늘리기 위해 외부 메모리에는 scale 1 인 Gaussian

image 만 저장하고, 필요시 추가적인 Gaussian filtering 으로 나머지 scale 의 Gaussian image pixel 을 생성한다. 또한 연속한 local-patch 사이에 중복 데이터가 증가하도록 key-point 처리 순서를 변경하는 방식을 제안한다. 추가적으로, 외부 메모리에 저장되는 Gaussian image 의 데이터 크기 자체를 줄이기 위하여 Gaussian image 를 down-sampling 하는 방안을 제안한다. Gaussian image 는 이미 고주파 성분이 제거된 영상이므로 추가적인 anti-aliasing filter 없이 이 동작이 가능하다. 또한 Gaussian image pixel 의 fraction-bit 를 제거하여도 정확도에 큰 문제가 없다는 사실을 확인하고, fraction-bit 를 제거하였다.

둘째, 일반적인 affine transform 연산 방식을 이용하는 하드웨어는 외부 메모리에서 원본 영상을 읽어 올 때, 불연속적인 주소로 외부 메모리를 접근하게 된다. 이런 외부 메모리 접근은 빈번하게 외부 메모리 latency 를 발생시키며 결과적으로 ASIFT 하드웨어의 동작 속도를 저하시킨다. 이 문제를 해결하기 위하여 본 논문은 SIFT feature 의 rotation-invariant 한 특성을 이용하여, affine transform 연산 방식을 변경하였다. 이 방식은 ASIFT 알고리즘이 취하는 모든 affine transform 연산을 수행할 때 연속된 외부 메모리 주소로 입력 영상을 접근할 수 있게 해준다. 이를 통해 불필요한 외부 메모리 latency 가 크게 감소된다. 제안된 affine transform 연산 방식은 원본 영상을 scaling 한 뒤 skewing 하는 연산 과정을 거친다. 본 논문은 이 과정에서 scaling 된 영상 데이터를 서로 다른 affine transform 연산에 재활용하는 방법을 제안한다. 이는 scaling 연산량을 감소시킬 뿐 만 아니라 외부 메모리 접근량도 감소시킨다. 제안된 방안들로 인한 affine transform 하드웨어의 속도 향상은 SIFT

하드웨어에 대기 없이 데이터를 공급할 수 있게 해주고, 최종적으로 utilization 향상을 통한 ASIFT 하드웨어의 동작 속도 향상에 기여한다.

1.3 논문 구성

본 논문은 구성은 다음과 같다. 2장에서는 본 논문의 연구의 기초가 되는 SIFT 알고리즘과 ASIFT 알고리즘에 대해 설명하고, 두 알고리즘에 대한 기존 가속화 연구에 대해 소개한다. 3장에서는 SIFT 하드웨어를 위한 외부 메모리 bandwidth 최적화 방안과 이에 대한 하드웨어 구현, 그리고 제안된 방안에 따른 성능 향상을 기존 연구 내용과 비교를 통해 평가한다. 4장에서는 ASIFT 알고리즘을 위한 affine transform 연산 방식과 이를 적용한 ASIFT 하드웨어를 제안한다. 제안된 하드웨어가 일반적인 transform 방식을 이용하는 하드웨어와 비교해 어떤 장점을 가질 수 있는지 실험을 통해 제시한다. 5장에서는 본 논문에서 제안한 ASIFT 하드웨어에 대한 실험 결과를 제시하고, 6장에서 결론을 맺음으로 본 논문을 마무리한다.

제 2 장 이전 연구 소개 및 문제 제시

2 장에서는 본 논문의 연구 주제인 SIFT 알고리즘과 ASIFT 알고리즘에 대한 설명과, 알고리즘의 연산 속도를 향상하기 위한 기존 연구들을 소개한다. 또한 기존에 연구된 SIFT 및 ASIFT 구현들의 문제점을 제시하여 본 연구가 진행된 이유를 설명한다.

2.1 SIFT 알고리즘 및 연산 가속화 연구

2.1.1 Scale-Invariant Feature Transform (SIFT)

SIFT 알고리즘은 동일한 장면이 촬영된 서로 다른 평면 영상이 카메라의 위치가 멀어지거나 카메라의 종류가 바뀌는 등의 변화로 인해 영상에 맺힌 장면의 scaling이 변하더라도, 두 영상에서 동일한 점들을 찾아 연결하므로 두 영상의 correspondence를 알려줄 수 있는 기능을 제공한다 [3]. 이와 같은 기능을 제공하기 위해 SIFT 알고리즘은 그림 2.1과 같은 연산 과정을 수행한다. 이 알고리즘은 크게 1) key-point generation 단계와 2) descriptor generation 단계로 나뉜다. 첫째로 key-point는 source image에서 물체 경계선의 모서리나 볼록 튀어나온 점과 같이 일반적인 pixel들에 비해 독특한 특징을 가지고 있는 point를 의미한다. Key-point를 검출하기 위해 source

image에 대한 Gaussian pyramid generation 단계와 이 pyramid를 이용해 key-point detection을 수행하는 하는 단계가 수행된다. 둘째로 검출된 key-point들을 구분하기 위해 descriptor를 계산한다. 검출된 key-point의 주변 일부분의 영상인 local-patch의 내부 pixel들의 gradient 정보를 histogram화 한 데이터가 descriptor이다. 이것을 계산하기 위해 local-patch 내부에서 gradient magnitude & orientation (GMO) generation을 수행하고, 이를 이용해 descriptor generation이 진행된다. 각 단계의 자세한 설명은 다음 절에서 이어진다.

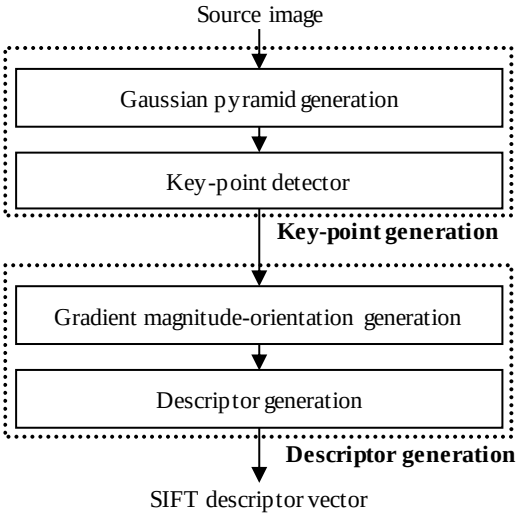


그림 2.1 SIFT 알고리즘의 연산 흐름도

2.1.1.1 Gaussian pyramid generation

SIFT 알고리즘은 Gaussian pyramid를 이용하여 key-point를 검출한다. Gaussian pyramid는 서로 다른 조건으로 Gaussian filtering된 image들로 구성된다. 기본적인 Gaussian filtering 연산은 식 (2.1)을 이용해 수행된다. source image $I(x,y)$에 Gaussian kernel $G(x,y,\sigma_i)$를 이용하여 convolution 연산을 수행한다. Filtering 결과 image인 Gaussian image는 $L(x,y,i)$라고 표현한다.

$$L(x,y,i) = G(x,y,\sigma_i) * I(x,y) \tag{2.1}$$

Gaussian kernel은 식 (2.2)와 같다. 여기서 σ_i는 Gaussian sigma를 의미하며, σ_i이 클수록 kernel의 크기도 커진다. index i는 scale index라고 부르고, Gaussian pyramid를 생성하는데 사용되는 다양한 크기의 Gaussian kernel과 Gaussian image를 구분하는데 사용된다.

$$G(x,y,\sigma_i) = \frac{1}{2\pi\sigma_i^2} e^{-\frac{x^2+y^2}{2\sigma_i^2}} \tag{2.2}$$

σ_i는 식 (2.3)과 같이 구성된다. σ_0는 Gaussian pyramid를 계산하는데 사용되는 Gaussian kernel 중 가장 작은 kernel의 sigma 값을 의미한다. S는 scale의 개수를 의미하고, Gaussian pyramid 하나의 octave 내에서는 총 $S+3$개의 Gaussian image가 존재하게 된다. 이에 따라 scale index i의 범위는 0에서 S+2까지다. 일반적으로 S와 σ_0는 Lowe가 제안한 값, 3과 1.6으로

정해진다.

$$\sigma_i = \sigma_0 \cdot 2^{\frac{i}{S}} \tag{2.3}$$

Gaussian pyramid는 식 (2.1) ~ (2.3)을 이용하여 구성되며, 이는 그림 2.2와 같다. 본 논문에서는 그림 2.2와 같이 3개의 octave를 구성한다. 첫번째 octave에는 scale index 0~5에 해당하는 Gaussian image로 구성된다. 네번째로 생성된 Gaussian image 인 $L_0(x,y,3)$ 은 x, y축으로 각각 1/2 sampling된 다음, octave 1를 위한 새로운 Gaussian image인 $L_1(x,y,0)$ 로 할당된다. $\sigma_3 = 2\sigma_0$ 이므로, $L_0(x,y,3)$ 를 1/2 sampling한 $L_1(x,y,0)$ 는 σ_0 로 Gaussian filtering된 image다. 이 image를 이용하여 octave 1과 동일한 방식으로 octave 2를 구성한다. 또한 동일한 연산 방식으로 octave 2를 구성한다.

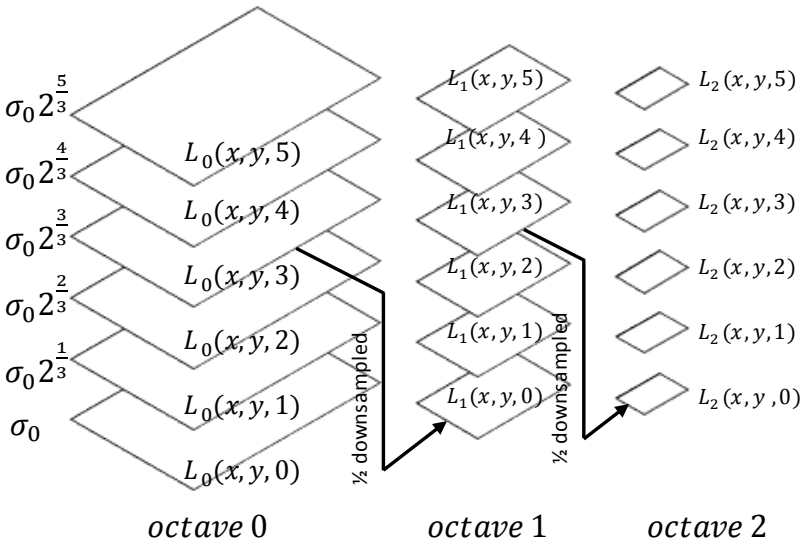


그림 2.2 Gaussian pyramid의 구성 (S = 3, Oct = 3)

2.1.1.2 Key-point detection

Gaussian pyramid를 구성한 다음, SIFT 알고리즘은 이웃한 scale index의 Gaussian image의 차 영상인 Difference of Gaussians (DoG) image를 식 (2.4)를 이용하여 생성한다.

$$D_i(x,y) = L(x,y,i+1) - L(x,y,i) \qquad (2.4)$$

하나의 octave 내에는 모두 6장의 Gaussian image가 있으므로, 한 octave에서 계산되는 DoG image의 수는 5장이다. 그 다음, 한 장의 DoG image와 이웃한 scale index의 DoG image 2장을 이용하여 DoG 값의 local maximum 또는 minimum을 검출하여 key-point 후보군을 선택한다.

Octave 0에 대한 DoG image 생성과 DoG extremum 결정 방법은 그림 2.3와 같다. Octave 0에 존재하는 6장의 Gaussian image를 이용하여 모두 5장의 DoG image $D_i(x,y)$는 계산할 수 있다. $D_i(x,y)$에서 key-point 후보를 검출하기 위해 이웃한 scale index의 $D_{i-1}(x,y)$와 $D_{i+1}(x,y)$를 참조한다. 만일 $D_i(x,y)$의 중심에 위치한 DoG pixel이 이웃한 8개의 DoG pixel들과, 상하에 위치한 이웃한 scale index의 $D_{i-1}(x,y)$ 와 $D_{i+1}(x,y)$ 의 동일 좌표와 이웃한 좌표에 위치한 18개의 DoG pixel 대비 최대이거나 최소 값일 경우 중심에 위치한 DoG pixel은 key-point 후보로 선정된다. 이 DoG extremum에는 검출된 위치의 scale, octave 정보가 할당된다. DoG extremum은 상하 scale index의 DoG image가 존재해야 구할 수 있으므로, $D_1(x,y)$, $D_2(x,y)$, $D_3(x,y)$에서만 key-point가 검출된다.

검출된 모든 key-point 후보는 location refinement, Low contrast

rejection, 그리고 Edge response elimination 단계를 거쳐서 최종적인 key-point인지 판별된다. Location refinement의 key-point의 stability를 check하는 단계이고, Low contrast rejection은 DoG Extremum 중 특정 threshold보다 낮은 상대적으로 평평한 DoG image 영역을 검출된 후보를 제거하는 단계이다. 마지막 Edge response elimination은 DoG Extremum 중 corner와 달리 다른 pixel과 구별하기 쉽지 않은 edge를 제거하는 단계이다.

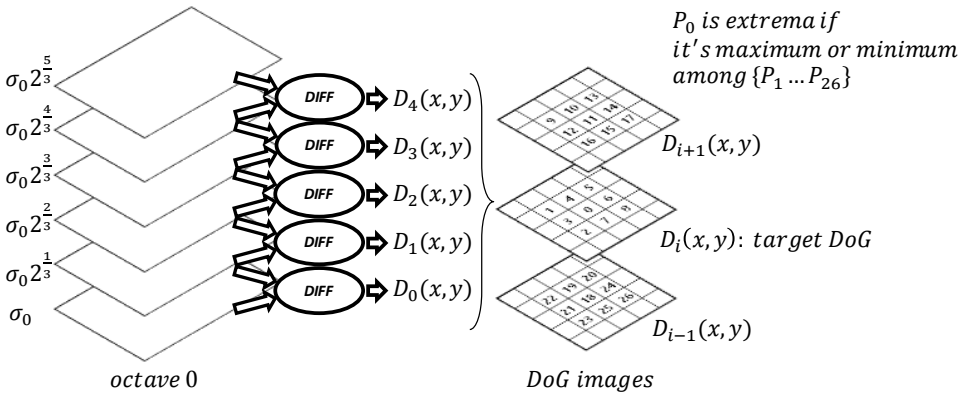


그림 2.3 DoG image 계산과 DoG extremum 결정 방법

2.1.1.3 Gradient magnitude-orientation generation

Key-point가 검출되고 나면, key-point를 구분하는데 사용되는 descriptor를 계산한다. SIFT descriptor는 영상의 gradient 값을 이용하여 계산되기 때문에, Gaussian pyramid에서 key-point가 검출된 $L(x,y,1)$, $L(x,y,2)$, $L(x,y,3)$에 대한 gradient magnitude와 orientation (GMO)를 계산한다. Gaussian image에 대한 GMO를 계산하기 위해서, 우선 Gaussian image pixel의 상하좌우에 위치한 이웃한 pixel과의 차 값이 gradient dx, dy를 계산한다. 이 계산은 식 (2.5)을 이용하여 계산된다.

$$\begin{aligned} \mathrm{dx} &= \mathrm{L_i}(x+1,y) - L_i(x-1,y) \\ \mathrm{dy} &= \mathrm{L_i}(x,y+1) - L_i(x,y-1) \end{aligned} \tag{2.5}$$

계산된 gradient dx, dy는 식 (2.6)를 이용하여 최종적으로 GMO가 계산된다.

$$\begin{aligned} \mathrm{mag(x,y)} &= \sqrt{dx^2 + dy^2} \\ \mathrm{ori(x,y)} &= \tan^{-1}(\frac{dy}{dx}) \end{aligned} \tag{2.6}$$

Descriptor를 생성하는데 사용되는 데이터는 검출된 key-point의 local-patch 내부에 존재하는 GMO 데이터다. 이때 local-patch 내부 GMO 데이터는 key-point가 검출된 Gaussian image에서 계산된 데이터를 가져온다. Local-patch의 width (W_{patch})는 key-point가 검출된 Gaussian image의 sigma 값에 비례하며, 이는 식 (2.7)와 같이 결정된다.

$$W_{patch} = 2Round\left(\frac{15}{\sqrt{2}}\sigma_i\right) + 1 \qquad (2.7)$$

SIFT descriptor는 rotation-invariant한 특성을 얻기 위하여 local-patch에 포함된 pixel들의 GMO 데이터를 이용해 계산된 dominant orientation만큼 local-patch를 회전시킨다. Dominant orientation의 계산 방법은 다음과 같다. 360°범위의 gradient orientation을 10°단위로 나눈 총 36-bin의 gradient histogram에 gradient magnitude 값을 축적한다. 이렇게 구해진 histogram의 최대값의 bin이 dominant orientation으로 결정된다. 만일 dominant orientation의 magnitude의 80% 이상인 또 다른 bin이 존재하면 이 bin도 또 다른 dominant orientation으로 결정된다. 본 논문에서 제시한 SIFT 하드웨어는 single dominant orientation만 사용한다. 왜냐하면 multiple dominant orientation을 사용할 경우, 생성해야 하는 descriptor 수가 증가하는 반면 matching performance는 큰 차이가 발생하지 않기 때문이다.

2.1.1.4 Descriptor generation

SIFT descriptor를 생성하는 과정은 3가지 단계로 이뤄진다. 먼저 계산된 dominant orientation으로 local-patch를 회전시킨다. 이 동작을 rotation normalization이라고 하고, 이를 통해 SIFT descriptor는 rotation-invariant한 특성을 갖게 된다. 그 다음 이를 5x5의 sub-block으로 나눈다. 각각의 sub-block들의 교차점에 위치한 16개의 지점을 중심으로 8-bin gradient histogram을 구성한다. 예를 들어 그림 2.4의 A pixel은 이웃한 hist[0][0], hist[0][1], hist[1][0], and hist[1][1]의 gradient histogram에 영향을 미친다. 이때 A pixel와 a, b, c, d와의 거리가 가까울수록 높은 가중치로 gradient histogram에 gradient amplitude가 누적된다. 이와 같은 연산을 Local-patch 내에 존재하는 모든 pixel에 대하여 적용한다. 그러면 총 4x4x8, 128개의 gradient magnitude로 구성된 descriptor vector가 결정된다. 최종적으로 descriptor vector는 이 vector의 norm을 이용하여 normalization된다. 이 과정을 통해 SIFT descriptor는 brightness에 강인한 특성을 갖게 된다.

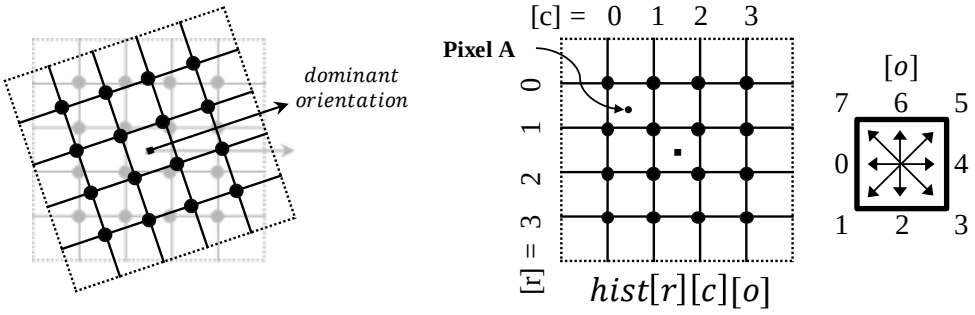


그림 2.4 Descriptor 생성 과정에 대한 설명

2.1.2 기존 SIFT 연산 가속화 연구 및 문제점

SIFT 알고리즘은 많은 연산량을 요구하기 때문에 실시간 처리가 어렵다. 이러한 한계를 극복하고 SIFT의 실시간 처리를 위해 general-purpose graphic processing unit (GPGPU)를 이용하여 SIFT 연산을 가속하는 연구와 specific designed hardware를 이용하여 가속하는 연구가 진행되었다. 2가지 접근법에 따른 SIFT 연산 가속 연구는 아래와 같다.

SIFT 알고리즘의 동작 속도 향상을 위하여 GPGPU를 활용한 연구가 다수 공개 되었다 [16-18]. Heymann *et al.* 가 제시한 GPGPU 기반 SIFT 연산 가속 기술은 일반적인 VGA 영상에 대하여 20frames/sec (fps)의 연산 속도를 보인다. 20fps 연산 속도는 실시간에 근접했다고 평가할 수 있다. 그러나, 과도한 하드웨어 리소스와 파워를 소모하기 때문에, SIFT 알고리즘이 활발히 활용될 수 있는 portable 기기에서 이용하기에 적합하지 않다. 이 단점을 극복하기 위하여 Mobile CPU 와 GPU를 이용하여 SIFT 연산을 가속화 하는 연구가 이뤄졌다 [19-21]. 이 연구들에서는 Mobile 기기에 최적화된 SIFT 알고리즘을 제시하였으나, 가장 빠른 동작 속도를 보이는 Rister *et al.* 의 SIFT 알고리즘 역시 320x280 크기의 영상에 대하여 9.9fps의 느린 동작 속도를 보여준다 [19]. 요약하면, GPU를 활용한 SIFT 알고리즘은 실생활에서 고속으로 활용하기에 부족한 동작 속도를 보인다고 평가할 수 있다. 반면, SIFT 알고리즘에 최적화된 하드웨어는 대부분 실시간 처리가 가능한 동작 속도를 보인다 [22-29]. Bonato *et al.* 에 의해 제시된 SIFT 하드웨어는 QVGA (320x240) 영상에 대하여 30fps의 속도로 동작 가능하다 [22]. Yao *et al.*, Mizuno *et al.*, Huang *et al.*, Qasaimeh *et al.* 그리고 Kim *et al.*이 제시한 하드웨어는 VGA (640x480) 영상에 대하여 실시간 처리가 가능한다 [23-24], [26-29]. 또한 Chiu *et al.* 의

하드웨어는 1080p (1920x1080) 크기의 영상에 대하여 실시간 처리가 가능하다 [25]. 이와 같은 연구 결과들을 통해 SIFT 연산의 실시간 처리를 위해서는 specifically designed hardware를 이용하는 접근법이 효과적이라는 것을 알 수 있다.

하드웨어로 구현된 SIFT 알고리즘은 software로 구현된 SIFT 알고리즘의 robust한 특성을 그대로 가지고 있어야 실제 application에 이용될 수 있다. 하지만 기존에 제안된 SIFT hardware는 SIFT 연산을 hardware에 적합한 방식으로 변형하거나 단순화하였다. 이러한 변경 사항은 제안된 SIFT hardware의 matching 정확도에 좋지 않은 영향을 미쳤다. Bonato *et al.* 은 descriptor 생성 시 key-point의 scale을 고려하지 않는다 [22]. Yao *et al.* 는 연산 되는 scale 수와 octave 수를 줄여 연산 복잡도를 줄였고 descriptor의 차원 수를 감소시켰다 [23]. Chiu *et al.* 의 하드웨어는 Gaussian filter의 coefficient를 크게 단순화시켰다. Kim *et al.* 은 descriptor vector 생성 시 descriptor normalization을 수행하지 않았다 [28]. 이와 같은 SIFT 연산의 단순화는 hardware 구현의 편의성을 높이고 연산 속도 향상을 이끌어 냈지만 SIFT 알고리즘의 matching 정확도를 낮추었다. 반면 Mizuno *et al.* 가 제안한 하드웨어는 SIFT 알고리즘을 크게 변경하지 않아 software로 구현된 SIFT와 유사한 성능을 보여준다 [24]. 하지만 이 하드웨어는 regions of interest (RoI) 영역을 한정하여 처리하기 때문에 실제로 처리 가능한 해상도가 낮다. Huang *et al.* 은 역시 software로 구현된 SIFT 알고리즘과 거의 동일한 정확도를 보이는 하드웨어를 제안하였다 [26]. 또한 이 하드웨어는 VGA 크기의 영상에 대하여 30fps로 동작이 가능하여, 기존에 제안된 SIFT hardware 중 가장 합리적인 hardware architecture를 제안하였다.

Huang *et al.* 이 제시한 SIFT 하드웨어는 동작 속도와 정확도 측면에서 바람직한 결과를 보여주었지만, 과도하게 많은 내부 메모리를 사용하는 문제점을 지니고 있다. Huang *et al.* 의 하드웨어는 SIFT 연산을 위해 필요한 모든 메모리 공간을 내부 메모리를 이용하여 구성하였다. 따라서 640x480 크기의 영상을 처리하기 위하여 5.729Mbits의 과도한 내부 메모리를 이용한다. 이와 같이 넓은 내부 메모리를 사용할 경우 hardware cost가 크게 증가한다. 그러므로 practical SIFT 하드웨어를 구성하기 위해서는 내부 메모리 크기를 최소화 하기 위한 방안이 필요하다.

2.2 ASIFT 알고리즘 및 연산 가속화 연구

2.2.1 affine invariant extension of SIFT (ASIFT)

SIFT 알고리즘은 scale, rotation invariant 하다는 장점을 지닌 반면 viewpoint 변환에 취약하다. Morel *et al.* 은 이 문제점을 보완하기 위하여 affine−invariant extension of SIFT (ASIFT) 알고리즘을 제안하였다. ASIFT 알고리즘은 영상의 affine 변환에 강인한 특성을 확보하기 위하여 평면 영상의 정면에서부터 카메라의 viewpoint 변화에 따라 다르게 나타나는 다양한 affine 영상 왜곡을 시뮬레이션한다. 그 다음 시뮬레이션 된 영상에서 SIFT feature를 추출한다.

그림 2.5는 source image와 camera pose 의 관계를 보여준다. Camera pose는 source image u 의 중앙 지점 o 을 중심으로 하는 반구 좌표계를 이용하여 표현된다. Camera pose가 source image의 정면에서부터 변화할 때 이로 인한 영상 왜곡을 시뮬레이션 한 affine transformed image는 latitude (θ), longitude (φ), zoom parameter (λ), spin parameter (ψ)를 가지고 표현된다. 이는 식 (2.8)과 같다. R_φ는 영상을 회전시키는 rotation matrix이고, $T_{1,1/t}$는 영상의 크기를 변화시키는 scaling matrix이다. ASIFT 알고리즘에서는 zoom parameter와 spin parameter는 각각 1과 0으로 고정된다.

$$A = T_{1,1/t}R_\varphi = \begin{bmatrix} 1 & 0 \\ 0 & 1/t \end{bmatrix} \begin{bmatrix} cos\varphi & -sin\varphi \\ sin\varphi & cos\varphi \end{bmatrix}, t = 1/cos\theta \qquad (2.8)$$

Affine transform 이용하여 시뮬레이션한 영상은 식 (2.9)과 같다. p는

affine transformed image의 위의 pixel 좌표이고, source image 상에서 이 pixel과 동일한 점의 좌표는 $q=(q_x, q_y)$이다. W와 H는 각각 source image u의 width와 height를 의미한다.

$$p = Aq = T_{1,1/t}R_{\varphi}q, \qquad 0 \le q_x \le W \; and \; 0 \le q_y \le H \tag{2.9}$$

그림 2.6은 source image에 대한 affine transform 과정을 단계별로 보여준다. (a)는 source image를 보여주며, 이는 R_{φ}에 의해 (b)와 같이 φ만큼 회전된다. 회전된 영상은 $T_{1,1/t}$에 의해 y축 방향으로 $1/t$만큼 scaling된다. Affine transform이 완료된 결과 영상은 그림 2.6의 (c)와 같다.

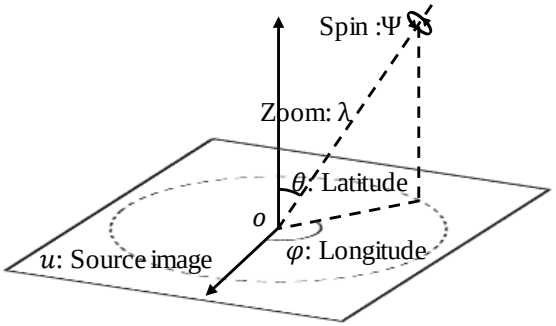
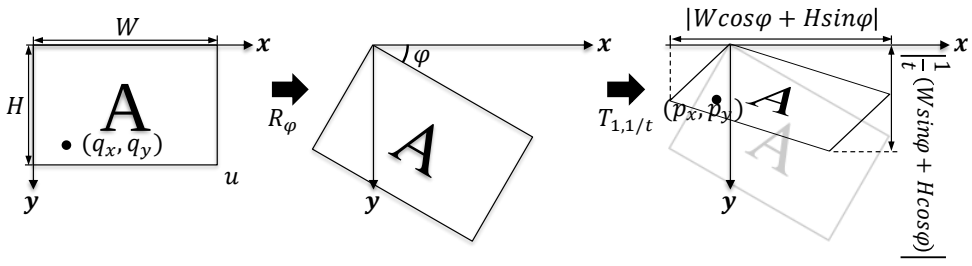


그림 2.5 Source image와 camera pose 사이의 관계 설명



(a) (b) (c)

그림 2.6 Source image에 대한 affine transform 연산의 단계별 과정

ASIFT 알고리즘은 fully affine-invariant하기 위해 미리 정의된 latitude (θ)와 longitude (φ)를 이용하여 affine transform을 수행한다. Latitude를 표현하는 또 다른 parameter인 tilt ($t = 1/cos\theta$)의 범위는 $[t_{min}, t_{max}] = [1, 4\sqrt{2}]$ 이고, longitude의 범위는 $[\varphi_{min}, \varphi_{max}] = [0^\circ, 180^\circ]$ 이다. Tilt와 longitude의 sampling step은 각각 $\Delta t = \sqrt{2}$ 와 $\Delta\varphi = 72^\circ/t$ 이다. 제시된 latitude와 longitude의 범위와 sampling step을 만족하는 viewpoint는 모두 42가지이고, 이는 표 2.1에서 보여준다. 표 2.1에서 각각의 viewpoint 위에 표시된 괄호 안의 숫자는 viewpoint를 구별하기 위한 viewpoint index이다. Viewpoint index i 에 대하여, 이 viewpoint에 해당하는 affine transform matrix는 A_i 로 표현한다. 그리고 A_i 를 이용하여 수행되는 ASIFT 연산은 $ASIFT(i)$ 라고 기술한다. 제시된 viewpoint에 해당하는 시뮬레이션된 영상이 생성되면, 이 영상에서 SIFT feature가 추출하고, $ASIFT(i)$가 완료된다.

표 2.1 Original ASIFT의 affine transform parameter

Latitude (Degree)	Longitude (Degree)														
0	(0) **0**														
45	(1) **0**	(2) **51**	(3) **102**	(4) **153**											
60	(5) **0**	(6) **36**	(7) **72**	(8) **108**	(9) **144**										
69	(10) **0**	(11) **25**	(12) **51**	(13) **76**	(14) **102**	(15) **127**	(16) **153**								
76	(17) **0**	(18) **18**	(19) **36**	(20) **54**	(21) **72**	(22) **90**	(23) **108**	(24) **126**	(25) **144**	(26) **162**					
80	(27) **0**	(28) **13**	(29) **25**	(30) **38**	(31) **51**	(32) **64**	(33) **76**	(34) **89**	(35) **102**	(36) **115**	(37) **127**	(38) **140**	(39) **153**	(40) **165**	(41) **178**

2.2.2 ASIFT 연산 가속화 연구

ASIFT 알고리즘은 source image를 다양한 viewpoint에서 바라본 시뮬레이션된 image를 다수 생성한다. 이 과정에서 SIFT feature를 추출해야 하는 입력 데이터량이 14배 가까이 증가한다. 즉, ASIFT 알고리즘의 연산량 요구량은 SIFT 알고리즘의 14배 정도로 과도하게 크다. 이러한 ASIFT 알고리즘은 고속으로 처리하기 위하여 Codreanu *et al.* 는 GPGPU를 이용해 연산 속도를 향상시킨 ASIFT 연구 내용을 발표하였다 [30]. 이를 GPU-ASIFT라고 부른다. GPU-ASIFT 는 VGA 크기의 source image를 1초에 9장 처리하는 동작 속도를 보여준다. 이것은 일반적인 desk-top PC용 고성능 CPU인 Intel Core i7-2600K를 이용하는 ASIFT software에 비해 40배 빠른 동작 속도다. 비록 GPU-ASIFT의 동작속도가 인상적인 속도 향상을 이뤘지만, 실시간 처리가 가능하다고 말하기에는 충분하지 못하다. 또한, GPU-ASIFT가 이정도의 속도 향상을 이룬 것은 NVIDIA GTX690 dual-GPGPU card라는 고가의 하드웨어 비용을 투입한 영향도 크다.

SIFT 알고리즘의 연산 가속 연구의 예를 보면 GPGPU를 이용한 가속 연구보다 specifically designed hardware를 이용한 접근법이 합리적인 하드웨어 비용을 투입하면서 실시간 연산이 가능하다는 점을 보여주었다. 그러나, ASIFT 연산 가속 연구에 있어서 ASIFT 알고리즘에 최적화된 하드웨어는 아직까지 발표된 것이 없다.

2.3 실시간 ASIFT 하드웨어 구현을 위한 연구 방향

본 논문은 실시간으로 ASIFT 알고리즘을 처리하는 하드웨어 구조를 제안한다. ASIFT 하드웨어는 크게 다양한 viewpoint에서 바라본 affine transform된 image를 만드는 Affine transform module과, 시뮬레이션된 image에서 SIFT feature를 추출하는 SIFT generation module로 구성된다. SIFT generation module은 2.1.2절에서 정리한 것과 같이 실시간 동작이 가능한 다양한 하드웨어 구조가 이미 제안되어 있다. 그 중 Huang *et al.* 이 제안한 SIFT 하드웨어는 동작 속도와 SIFT feature의 정확도 면에서 가장 좋은 결과를 보여주었다. 하지만 연산에 사용하는 모든 메모리 공간을 내부 메모리를 이용하여 구현하였기 때문에, 하드웨어 비용 측면에서 합리적이지 못하다. 본 논문에서는 Kim *et al.* 이 제안한 접근법과 같이 SIFT 하드웨어의 일부 메모리 공간을 외부 메모리에 위치시켜 내부 메모리 사용량을 최적화하는 접근법을 취한다. 이 방법이 하드웨어의 동작 속도를 저해하지 못하게 하기 위하여, 본 논문에서는 외부 메모리 bandwidth를 최소화하는 방안을 모색한다. 외부 메모리는 내부 메모리와 달리 요청된 데이터를 여러 cycle 뒤에 제공한다. 이러한 외부 메모리 latency는 SIFT 하드웨어의 동작 속도를 저하시킬 수 있다. 이 문제점을 제거하기 위해, 본 논문은 외부 메모리 접근량을 분석하고 이를 줄이는 방안을 연구하여 외부 메모리 interface가 전체 연산의 bottle-neck이 되지 않게 한다. 이는 하드웨어의 동작 속도를 저하시키지 않으면서 하드웨어 비용을 줄이는 결과를 이끌 수 있다. 이에 관하여 연구는 3장에서 자세히 소개한다.

ASIFT 하드웨어의 동작 속도를 최대화한다는 측면에서 affine transform module은 SIFT generation module의 utilization을 떨어뜨리지 않기 위해

SIFT generation module이 affine transform된 image를 요청할 때 즉각적으로 제공할 수 있어야 한다. 이 점에서 Affine transform 연산은 복잡하지 않기 때문에 연산에 소모되는 cycle은 문제가 되지 않는다. 반면 affine transform 과정에서 image를 rotation시키는 단계는 외부 메모리에 저장된 source image를 raster order scan과 같이 가로 방향으로 연속적으로 읽어오지 못한다는 것은 외부 메모리 latency로 인한 속도 저하를 문제가 된다. 불연속적인 외부 메모리 접근을 없애기 위하여 본 논문은 새로운 affine transform 연산 방식을 연구한다. SIFT feature는 rotation-invariant하다는 특성을 가진다. 이 특징을 이용하면 affine transform의 rotation 단계를 수행하지 않게 만들 수 있다. 이에 관한 자세한 연구는 4장에서 소개한다.

제 3 장 외부 메모리 bandwidth를 저감한 실시간 SIFT 하드웨어 구조

3장에서는 ASIFT 알고리즘의 가장 복잡한 연산 단계인 SIFT feature 생성 과정을 실시간으로 수행할 수 있는 하드웨어 구조를 제안한다. 합리적인 하드웨어 비용의 SIFT 하드웨어를 설계하기 위해, SIFT 연산의 중간 데이터 값을 저장하는 buffer를 외부 메모리에 위치시키는 접근법을 취한다. 이로 인한 외부 메모리 latency로 동작 속도가 감소하는 문제를 해결하기 위해, 외부 메모리 접근을 감소시키는 3가지 방안과 이를 적용한 SIFT 하드웨어 구조를 제안한다. 그리고 SIFT 하드웨어의 외부 메모리 접근 감소량과 동작 속도 분석을 통해 제안하는 방안의 효과를 실험적으로 제시한다.

3.1 외부 메모리에 저장될 SIFT 연산의 중간 데이터 고찰

SIFT 하드웨어의 내부 메모리 크기를 감소시키기 위해, 가장 크기가 큰 data buffer를 외부 메모리로 대체한다. SIFT feature를 추출하는 과정에서 생성한 데이터 중 많은 메모리 공간이 필요한 데이터는 Gradient magnitude & orientation (GMO) 데이터 또는 Gaussian image다. 그림 3.1이 보여주는 Huang *et al.* 이 제시한 하드웨어 구조를 보면, VGA 크기의 영상을 처리하기 위해 2.778Mbits의 큰 GMO buffer를 사용하는 것을 볼 수 있다. 이에 비해

Gaussian image buffer는 상대적으로 작은 138.2Kbits를 차지한다. 반면 Mizuno *et al.* 이 제시한 또다른 SIFT 하드웨어는 Gaussian image 전체를 저장하기 위해 큰 Gaussian image buffer를 사용하고, 대신 GMO buffer는 하나의 local-patch의 GMO 데이터만 저장할 수 있는 크기로 구성하였다 [24]. 만일 Mizuno *et al.* 의 buffer 구조를 Huang *et al.*이 제시한 SIFT 하드웨어 구조에 접목한다면 GMO buffer는 123.79Kbits로 감소되는 반면, Gaussian image buffer는 2.657Mbits로 크기가 증가한다. 요약하면, SIFT 하드웨어가 사용하는 내부 메모리의 일부를 외부 메모리로 대체하여 하드웨어 비용을 감소시킬 때, 내부 메모리 크기의 감소량 측면에서는 Gaussian image buffer를 선택하든, GMO buffer를 선택하든 거의 동일한 결과를 얻게 된다.

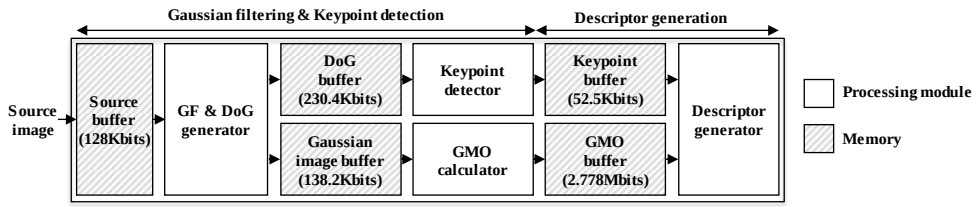


그림 3.1 Huang *et al.* 이 제시한 SIFT 하드웨어 구조 [26]

하드웨어의 내부 메모리를 외부 메모리로 대체할 때, 고려해야 하는 또 다른 조건은 외부 메모리 latency로 인산 하드웨어 동작 속도의 감소다. Gaussian image나 GMO data는 SIFT 연산에서 핵심적인 데이터이므로 빈번하게 메모리에 읽고, 쓰여진다. 외부 메모리 bandwidth가 증가하면 latency 역시 증가하게 되어 하드웨어의 응답 대기 시간이 길어진다. 이를 고려하기 위해,

Gaussian image 또는 GMO 데이터를 외부 메모리에 저장하였을 때 발생하는 외부 메모리 접근량을 분석한다.

표 3.1은 Gaussian image buffer를 외부 메모리에 위치시켰을 경우, SIFT 하드웨어가 외부 메모리에 얼마나 증가하는지 동작의 종류별로 보여준다. Source image load는 원본 image와 다음 octave를 위한 down-sampling된 image를 읽어오는 외부 메모리 접근을 보여주며, 한 프레임을 처리하는데 12.9MBits를 접근한다. Next octave source write는 다음 octave 연산을 위해 외부 메모리에 down-sampling된 image를 저장하는 접근을 의미한다. Gaussian image write는 생성된 Gaussian image를 descriptor 생성 시, 다시 사용하기 위해 외부 메모리에 저장하는 동작을 의미하며, 87.09Mbits의 접근량이 발생한다. External Gaussian image buffer에 저장된 Gaussian image는 key-point가 검출되었을 때, descriptor를 생성하기 위해서 다시 읽혀지며, 247.56Mbits로 가장 많은 외부 메모리 접근량을 차지한다. 이것은 3000개의 key-point가 검출된 상황을 가정한 수치다. 마지막으로 생성된 descriptor를 외부 메모리에 저장하는데 3.07Mbits의 외부 메모리 접근이 발생한다.

표 3.2는 GMO buffer를 외부 메모리에 위치시켰을 때 발생하는 외부 메모리 접근량을 보여준다. Source image load, next octave source write, descriptor write에 의한 외부 메모리 접근량은 표 3.1과 동일하다. GMO write과 read를 위한 접근량은 각각 94.35Mbits와 250.51Mbits다. 이것은 3000개의 key-point가 검출된 상황을 가정한 수치다. 종합해 보면, Gaussian image buffer를 외부 메모리에 위치시켰을 경우 이로 인한 외부 메모리 bandwidth 증가량은 334.65Mbits이고, GMO buffer를 외부 메모리로 옮겼을

때는 344.86Mbits로 두 경우 모두 거의 동일하다. 다시 말해, Gaussian image와 GMO 데이터 중 어떤 데이터를 외부 메모리에 저장하든 굉장히 많은 bandwidth 증가 문제가 발생하며, 그 양이 거의 동일하다는 것을 확인할 수 있다.

Gaussian image buffer를 외부 메모리에 위치하였을 때, SIFT 하드웨어를 실시간으로 동작시키기 위해 요구되는 bandwidth는 10.587Gbps다. 일반적인 SoC 칩에서 이용 가능한 16 data bits를 제공하는 SDRAM@133MHz, DDR2@400MHz의 bandwidth는 각각 2,128Mbps와 12.8Gbps다. 따라서 SDRAM@133MHz를 이용하는 SoC 칩에 병합된 SIFT 하드웨어는 실시간 동작이 어려울 만큼의 bandwidth 요구량을 보인다. DDR2@400Mhz를 이용할 경우 비록 SIFT 하드웨어는 실시간 동작이 가능하다 할지라도 83%의 bandwidth를 SIFT 하드웨어가 독점하므로 SoC 칩이 정상 운영되기 힘들다. 이런 문제를 해결하기 위해서는 SIFT 하드웨어의 bandwidth 요구량을 감소시키기 위한 연구가 필요하다.

본 논문은 Gaussian image buffer와 GMO buffer 중에서 외부 메모리 bandwidth를 최적화 하는데 적합한 Gaussian image buffer를 외부 메모리로 대체한다. Gaussian image는 GMO 데이터에 비해 외부 메모리 bandwidth를 줄이는데 적합한 3가지 특성을 가지고 있다. 첫 번째는 높은 scale의 Gaussian image는 낮은 scale의 Gaussian image를 이용하여 생성할 수 있다는 특성이다. 이것은 2개의 이웃한 key-point의 local-patch의 공통 데이터를 재사용할 때, 재사용 가능한 데이터량을 늘리는데 효과적으로 이용된다. 두 번째로, Gaussian image는 low-pass filtering된 image 이므로 별도의 anti-aliasing filtering 없이 down-sampling 및 interpolation이 가능하다는 점이다. Gaussian image를 down-sampling한 뒤 외부 메모리에 저장할 경우 bandwidth를

효과적으로 줄일 수 있다. 마지막으로 Gaussian image의 significant bits가 GMO data에 비해 짧다는 점이다. 이 점을 이용하여 Gaussian image의 pixel 크기를 SIFT feature의 정확도 감소를 최소화 하는 범위에서 줄일 수 있다.

표 3.1 External Gaussian image buffer를 사용하는 SIFT 하드웨어의 외부 메모리 접근량

Operation	Source image load	Next octave source write	Gaussian image write	Gaussian image read	Descriptor write	**Total**
Memory access (Mbits)	12.90	2.30	87.09	247.56	3.07	**352.93**

표 3.2 External GMO buffer를 사용하는 SIFT 하드웨어의 외부 메모리 접근량

Operation	Source image load	Next octave source write	GMO write	GMO read	Descriptor write	**Total**
Memory access (Mbits)	12.90	2.30	94.35	250.51	3.07	**363.14**

3.2 외부 메모리 bandwidth를 줄이기 위한 방안

본 논문은 SIFT 하드웨어는 Gaussian image buffer를 외부 메모리에 위치시키는 구조를 제안한다. 이 경우, descriptor를 생성하기 위해 Gaussian image를 local-patch 단위로 읽어오는 동작으로 인해 외부 메모리 bandwidth가 급격하게 증가하게 된다. 이런 외부 메모리 bandwidth 증가 문제를 해결하기 위해 본 논문은 3가지 bandwidth 최적화 방안을 제안한다.

3.2.1 Local-patch 재사용 방안

External Gaussian image buffer에 대한 외부 메모리 접근량을 줄이기 위해, 연속해서 처리되는 key-point의 local-patch 사이의 중복 데이터를 재사용하는 방안을 제안한다. 그림 3.2 (a)는 local-patch 사이의 중복 데이터가 재사용되는 예를 보여준다. 검은 색 block은 검출된 key-point를 의미하고, 이를 표현하는 kp_k에서 k는 k번째로 key-point가 검출되었음을 의미한다. 연속해서 검출된 key-point kp_k 와 kp_{k+1}는 순서대로 descriptor 생성이 이뤄지고, 이때 external Gaussian image buffer에서 읽어온 흰색 block으로 표시된 2개의 local-patch 사이에는 회색으로 표시된 영역과 같이 중복 영역이 존재할 수 있다. 만일 kp_k 의 descriptor를 생성하기 위해 읽어온 local-patch를 internal buffer에 저장해두면, kp_{k+1}의 descriptor를 생성할 때 회색 영역에 존재하는 pixel 데이터를 외부 메모리에서 가져오지 않고, internal buffer에 저장된 것을

재사용할 수 있다. 이 방안을 통해 감소되는 외부 메모리 접근량을 증가시키기 위해서는 연속된 local-patch 사이의 중복 영역을 늘리는 것이 중요하다.

만일 연속적으로 검출된 key-point의 scale index가 서로 다르다면, 이에 해당하는 local-patch 역시 서로 다른 Gaussian image에 속한다. 그림 3.2 (b)는 이와 관련된 예를 보여준다. 연속적으로 검출된 kp_k, kp_{k+1}, kp_{k+2}가 각각 scale 1, scale 2, scale 3에 속해 있다면, 이에 해당하는 local-patch 역시 서로 다른 scale의 Gaussian image에 속하게 된다. 이 그림이 설명해주는 것과 같이 연속적으로 검출된 key-point들이 좌표 상으로 중복 영역이 발생할 만큼 가까이 위치하였다고 하더라도 서로 다른 scale의 에서 검출되었다면 local-patch 사이에 중복된 데이터는 존재하지 않는다.

이런 상황에서 local-patch 재사용률을 높이기 위해서 본 논문은 낮은 scale의 Gaussian image를 적절한 Gaussian sigma로 filtering하면 높은 scale의 Gaussian image를 생성할 수 있다는 특성을 이용한다 [28]. scale 1의 Gaussian image는 σ_1의 Gaussian filter로 filtering된 image이고, 생성해야 하는 scale i의 Gaussian image에 해당하는 sigma가 σ_i일 경우, 추가적인 Gaussian filter의 sigma값 σ'_i은 식 (3.1)으로 결정된다 [31].

$$\sigma'_i = \sqrt{{\sigma_i}^2 - {\sigma_1}^2} \tag{3.1}$$

External memory에는 scale 1의 Gaussian image만 저장한다. 그리고 모든 scale의 key-point에 대하여 scale 1의 Gaussian image에서 동일한 위치의 local-patch를 읽어온다. 단 scale 2, 3의 key-point에 대해서는 local-patch의 경계에 위치한 pixel의 추가적인 Gaussian filtering을 고려하여 조금 더

큰 local-patch를 읽어온다. 이러한 방안을 이용하면 그림 3.2 (c)와 같이 key-point의 scale index와 상관 없이 local-patch 사이 중복 데이터가 발생하며, 이를 통해 외부 메모리 접근량을 효과적으로 줄일 수 있다

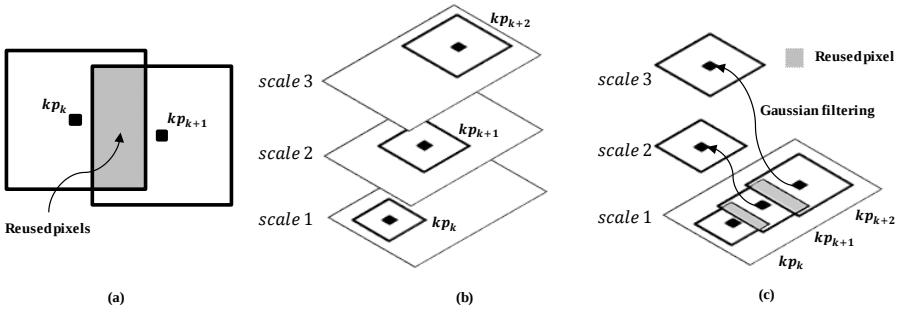


그림 3.2 Local-patch 재사용 방안에 대한 설명

앞서 설명한 local-patch 재사용 방안을 SIFT 하드웨어에 구현하기 위해 관련된 module을 설계하였고, 이는 그림 3.3 (a)과 같다. Local-patch 재사용 module은 patch boundary calculator, load requester, overlap detector, local-patch reuse buffer, Gaussian filter로 구성된다. 만일 kp_k가 검출되어 이에 대한 local-patch를 external Gaussian image buffer에서 읽어온다고 가정하자. 그러면 patch boundary calculator가 kp_k의 local-patch의 경계를 알려주는 st_k와 ed_k를 계산하여 출력해준다. st_k는 이 local-patch의 왼쪽 상단 꼭지점의 좌표를 의미하고, ed_k는 하단 오른쪽 꼭지점의 좌표를 의미한다. 모든 local-patch는 정사각형 형태이므로 이 st_k와 ed_k로 local-patch의 경계를 정의할 수 있다. st_k와 ed_k를 참조하여 load request는 kp_k의 local-patch의

pixel을 순차적으로 요청한다. 요청된 pixel의 좌표 p_{req}를 overlap detector에 전달하면, overlap detector는 boundary register에 저장된 이전 key-point의 local-patch의 경계 정보 st_{k-1}, ed_{k-1}를 참조하여 p_{req}가 재사용 가능한 데이터인지 아닌지 판단한다. 판단 방식은 p_{req}가 이전 local-patch 내부에 위치하는지 아닌지 확인하는 것이고, 이는 식 (3.2)과 같다. $p_{req}(x)$와 $p_{req}(y)$는 각각 p_{req}의 x좌표와 y좌표를 의미한다. 마찬가지로 $st_{k-1}(\mathrm{x})$와 $st_{k-1}(y)$는 st_{k-1}의 x좌표와 y좌표를 뜻하고, $ed_{k-1}(\mathrm{x})$와 $ed_{k-1}(y)$는 ed_{k-1}의 x좌표와 y좌표를 의미한다.

$$st_{k-1}(\mathrm{x}) \le p_{req}(\mathrm{x}) \le ed_{k-1}(\mathrm{x}) \text{ and } st_{k-1}(\mathrm{y}) \le p_{req}(\mathrm{y}) \le ed_{k-1}(\mathrm{y}) \quad (3.2)$$

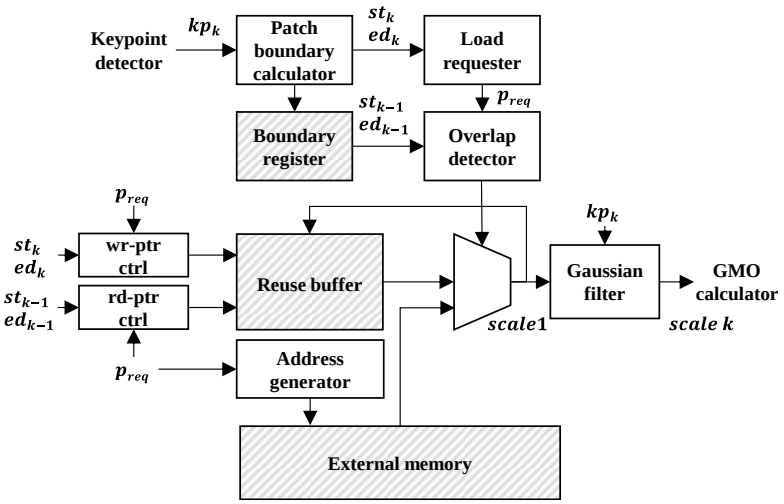
제안된 local-patch 재사용 방안은 key-point가 어떤 scale에 검출되든 scale index 1의 Gaussian image에서 local-patch를 가져온다. 그렇기 때문에 scale이 증가하면서 local-patch 크기가 커져야 하는 것이 st_k와 ed_k에 반영되어야 한다. Scale 1, 2에서 검출된 key-point의 경우 local-patch에 추가적인 Gaussian filtering이 수행되어, scale 1, 2의 Gaussian image pixel을 계산해야 한다. 이때 local-patch 경계에 위치한 pixel의 filtering을 위해서는 local-patch 외부의 pixel이 추가적으로 필요하다. 이러한 이유로 증가하는 local-patch 크기 역시 st_k와 ed_k에 반영되어야 한다.

p_{req}가 kp_{k-1}의 local-patch 내부에 위치해 있다고 판단되면, 이 pixel은 이미 reuse buffer에 저장되어 있다는 것을 뜻한다. 따라서 이 pixel은 external Gaussian image buffer가 아닌 reuse buffer가 제공되어 Gaussian filter로

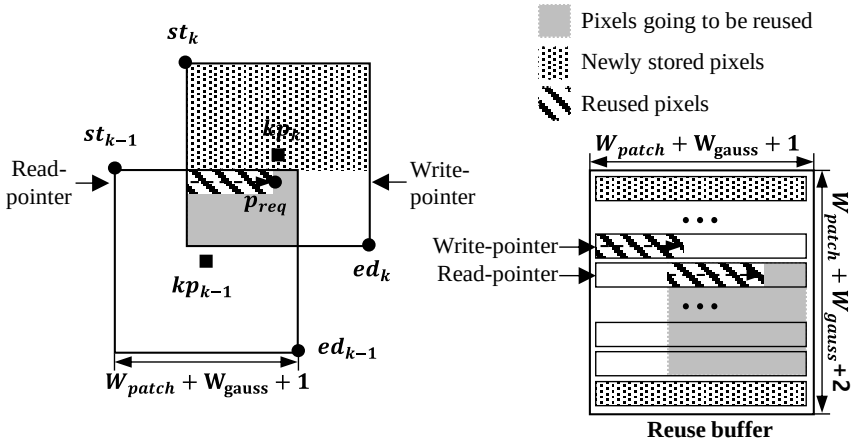
전달된다. Reuse buffer의 read-, writer-pointer는 p_{req}와 st_k, ed_k, 그리고 st_{k-1}, ed_{k-1} 정보를 가지고 운영된다. Read-pointer는 현재 요청된 p_{req} 좌표의 pixel이 reuse buffer 어느 위치에 저장되어 있는지 알려준다. Writer-pointer는 확보된 pixel이 다음 key-point를 위해 reuse buffer의 어느 위치에 저장되어야 하는지 알려준다. 만일 p_{req}가 kp_{k-1}의 local-patch 외부에 위치해 있다고 판단되면 이 pixel은 external Gaussian image buffer에서 가져온다. 확보된 pixel은 write-pointer가 알려주는 reuse buffer 주소에 저장된다. 동시에 이 pixel은 Gaussian filter로 전달되어 적절한 scale의 Gaussian image pixel로 변경되어 GMO calculator에 전달된다.

그림 3.3 (b)는 연속해서 처리되는 두 key-point의 local-patch 사이 중복 데이터를 검출하는 과정과 이때 reuse buffer가 어떻게 동작하는지 설명해준다. Local-patch를 재사용하기 임시 저장하는 공간인 reuse buffer는 내부 메모리로 구현되며, write-pointer와 read-pointer로 제어된다. 각각의 pointer는 2개의 address register로 구성된다. 하나는 내부 메모리의 local-patch의 한 row의 시작 지점 주소를 가리킨다. 이 address는 내부 메모리의 마지막 row 시작 지점을 가리킨 다음, 더이상 pointer가 증가할 memory 공간이 없을 경우 첫번째 rows의 시작 지점을 가리킨다. 이렇게 순환되는 pointer를 사용하면 reuse buffer는 local-patch의 첫번째 row를 실제 내부 메모리의 첫번째 row 영역에 저장하지 않아도 된다. 다른 address register는 기존 address register가 가리키는 row에서 현재 데이터가 접근되는 또는 접근될 column 주소를 가리킨다. Write-pointer는 새로 확보된 pixel을 저장할 공간을 가르시며, kp_k의 local-patch를 위한 것이다. 만일 p_{req}가 외부 메모리에서 제공되었다면, 이것은 write-pointer가 가르기는 자리에 저장되고, writer-

pointer는 증가한다. Read-pointer는 kp_{k-1} 의 local-patch를 순차적으로 접근하기 위한 것으로, read-pointer가 가르기는 pixel이 재사용 가능하면 읽혀져서 재사용된다. 재사용 가능한 pixel은 write-pointer가 가르기는 자리에 저장된다. 재사용할 수 없다면 read-pointer는 그냥 증가되며, 이 pixel은 garbage value가 된다. Reuse buffer가 정상적으로 동작되려면 write-pointer는 read-pointer를 앞서 갈 수 없다. 그림 3.3 (b)의 reuse buffer에서 볼 수 있듯이, 이 buffer는 local-patch보다 하나의 row를 더 저장할 수 있는 크기다. 이 추가적인 공간으로 인해 read-pointer와 write-pointer가 빈번한 충돌이 방지되며, 이로 인한 stall cycle이 사라진다. Reuse buffer의 크기는 scale 3인 key-point의 local-patch width (W_{patch})와 sigma 값이 σ'_3 인 Gaussian filter의 tap size (W_{gauss})에 의해 결정된다.



(a)



(b)

그림 3.3 Local-patch 재사용을 위한 module과 reuse buffer 동작 설명

일반적으로 SIFT 하드웨어는 검출되는 key-point 순서대로 descriptor를 생성한다. 따라서 key-point 검출 순서대로 local-patch를 읽어오게 되며, 이 순서는 Gaussian image의 재사용률에 큰 영향을 미친다. 그림 3.4 (a)는 frame-based raster scan order로 key-point를 검출한 상황을 보여준다. 검은 색 box는 key-point를 의미하며, key-point kp_k의 k는 검출 순서를 알려준다. key-point 주변 흰색 box는 local-patch를 의미한다. 점선 화살표는 각각의 pixel line에서 key-point의 검출 방향을 알려주며, 실선 화살표는 pixel line scan의 순서를 알려준다. 그림 3.4 (a)의 상황에서는 frame-based raster scan order를 이용하면 세로 방향의 local-patch 재사용률이 낮아진다. 왜냐하면 세로 방향으로 인접한 key-point가 연속으로 검출될 확률이 낮기 때문이다. kp_k의 local-patch와 kp_{k+r}의 local-patch는 세로 방향으로 가까이 위치해 있기 때문에 중복 데이터가 존재한다. 그러나 kp_{k+1}의 local-patch가 kp_{k+r}의 local-patch보다 먼저 읽혀져서, kp_{k+r}를 처리할 때 kp_k의 local-patch는 reuse buffer에서 지워지게 된다.

하드웨어 구현에 있어서 block 단위로 연산을 수행하는 것은 일반적인 접근법이다. 따라서 SIFT 하드웨어도 block 단위로 동작하도록 설계되었다. 이 사항은 key-point 검출 순서에 큰 영향을 미친다. 만일 W_{block} 크기의 block을 기준으로 block-based raster scan order로 key-pointer를 검출하게 되면, 그림 4.3 (b)와 같이 세로 방향으로 local-patch가 재사용될 가능성이 커진다. W_{block} 기준 block-based raster scan은 source image를 가로 방향으로 W_{block}의 width의 다수의 block으로 나눈 후, 각각의 block 안에서 raster scan order로 key-point를 검출하는 것을 말한다. 이와 같은 검출 순서를 이용하면, 그림 4.3 (b)와 같이 세로 방향으로 인접해 있으나, 연속적으로 검출되지 못했던

두 key-point가 연속적으로 검출될 가능성이 높아진다. 세로 방향으로 local-patch가 재사용될 가능성은 W_{block}가 source image의 width 대비 감소할수록 증가한다.

Local-patch 재사용률과 W_{block} 사이의 관계를 확인하기 위하여 Mikolajczyk *et al.* 가 제시한 8개의 Oxford test image들을 이용하여 실험을 진행하였다 [32]. 그 결과는 그림 3.5와 같다. 이 그래프의 가로축은 W_{block}를 나타내고, 세로축은 재사용되는 pixel의 수를 보여준다. 이 그래프에서 알 수 있듯이 W_{block}가 감소할수록 재사용되는 pixel의 수가 감소하며, W_{block}가 32일 때 가장 많은 수의 pixel이 재사용된다. W_{block}가 32보다 더 짧아지면 오히려 재사용 pixel 수는 감소한다. 결과적으로 이 실험 결과를 통해 local-patch의 재사용률을 최대화 하기 위해서는 W_{block}가 32로 선정되어야 한다. 따라서 본 논문에서는 W_{block}를 32로 선정한다.

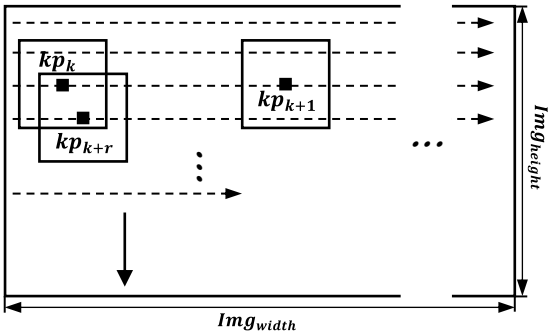
Block 단위로 key-point를 검출하는 것은 block 경계 가까이 위치한 pixel들이 2번 외부 메모리에서 읽혀져야 한다는 단점을 지닌다. 따라서 W_{block}를 너무 짧게 선정하게 되면, local-patch 재사용으로 인해 감소된 외부 메모리 bandwidth가 block 경계 근처 pixel이 2번 읽혀져서 증가하는 bandwidth에 의해 효과가 감소될 수 있다. 그림 3.6은 W_{block}와 block 단위 연산에 의한 불필요하게 외부 메모리에 접근되는 데이터량을 보여준다. 이 그래프에서 알 수 있듯이 W_{block}이 32로 선정될 경우, redundancy가 과도하다. 결론적으로 이 실험 결과를 통해 W_{block}는 최소한 96 이상이 되어야 redundancy가 급격하게 감소한다는 것을 알 수 있다.

본 논문은 W_{block}를 32로 유지하면서 불필요한 외부 메모리 접근량을 줄이기 위하여 key-point의 출력 순서를 검출 순서와 다르게 하는 방안을 제안한다.

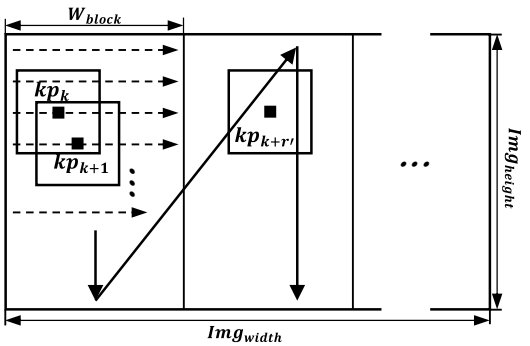
이와 같은 key-point 재배열은 그림 3.4 (c)와 같이 이뤄진다. Key-point 검출은 $3 \times W_{block} = 96$ 단위의 block width로 block-based raster scan order로 이뤄진다. 그러면 redundant한 중복 외부 메모리 접근은 그림 3.6에서 볼 수 있듯이 급격하게 감소하게 된다. 검출된 key-point는 병렬로 배치된 3개의 FIFO에 저장된다. W_{block} 단위의 첫번째 block에서 검출된 key-point는 첫번째 FIFO에 저장된다. 두번째 block에서 검출된 key-point는 두번째 FIFO에 저장되고, 마찬가지로 세번째 block에서 검출된 key-point는 마지막 FIFO에 저장된다. 한편, 첫번째 FIFO에 저장된 key-point가 모두 처리될 때까지 나머지 FIFO에 저장된 key-point는 대기한다. 첫번째 FIFO의 key-point가 처리되면 두번째 FIFO의 key-point가 처리된다. 마찬가지로 두번째 FIFO의 데이터 처리가 완료되면, 마지막 FIFO에 데이터가 처리된다. 이와 같은 규칙으로 key-point 검출과 key-point 처리가 수행되면, key-point 검출은 마치 W_{block}가 96인 것과 같이 동작하여 redundant한 외부 메모리 접근량이 급감하게 된다. 동시에 key-point 처리는 W_{block}가 32인 것 같이 수행되므로 local-patch 재사용률은 최대화 된다.

Redundant한 외부 메모리 접근량을 좀 더 감소시키기 위하여, key-point FIFO를 좀 더 많이 운영하는 방식을 고려해볼 수 있다. 그러나, 이런 접근법은 key-point FIFO를 구성하는데 사용되는 내부 메모리의 크기를 증가시킨다. 또한 과도하게 많은 FIFO를 운영하게 되면 예기치 못한 stall cycle이 key-point FIFO의 출력을 처리하는 GMO calculator에서 발생할 수 있다. 예를 들어 첫번째와 두번째 block에 key-point가 존재하지 않아 이에 해당하는 FIFO1과 FIFO2가 비어 있고, FIFO3에만 key-point가 존재한다고 하더라도, 전체 block에 대한 key-point 검출이 종료되지 않으면 FIFO3에 존재하는 key-

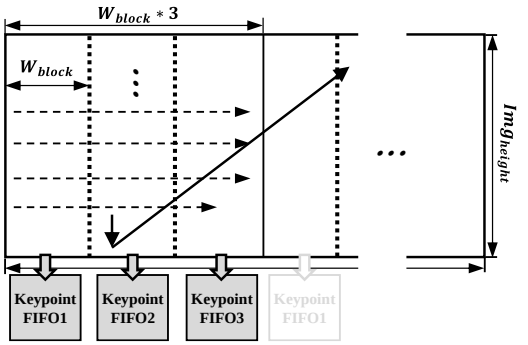
point는 처리될 수 없다. 이런 이유로 GMO calculator의 stall time은 증가되며, FIFO를 많이 운영할수록 stall time은 길어지게 된다. 따라서 본 논문은 block의 경계 인근 pixel에 대한 중복 외부 메모리 접근이 급감하고, 더 이상 효과적으로 감소되지 않는 시점인 W_{block}가 96인 점을 선정한다



(a) Frame-based raster scan order



(b) Block-based raster scan order



(c) Key-point reordering with three FIFOs

그림 3.4 세가지 data scan order와 데이터 재사용률 가능성의 고려

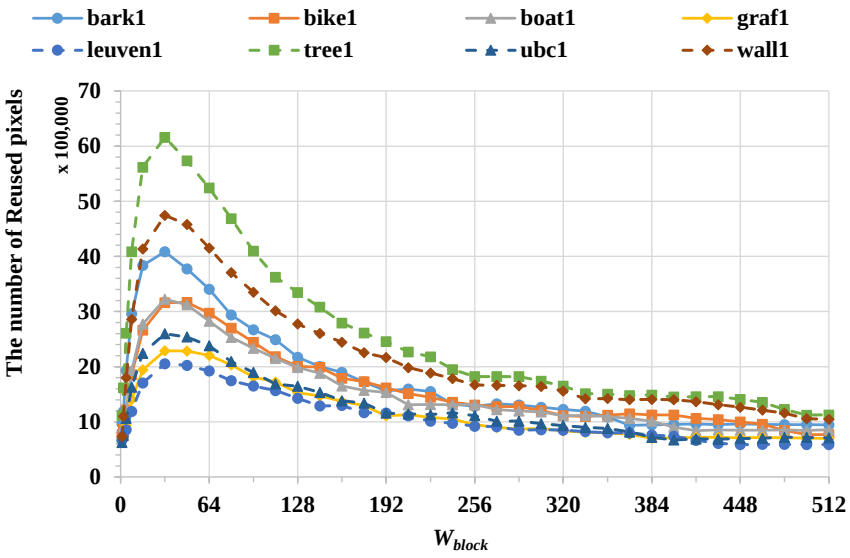


그림 3.5 W_{block}와 재사용되는 pixel 수와의 관계

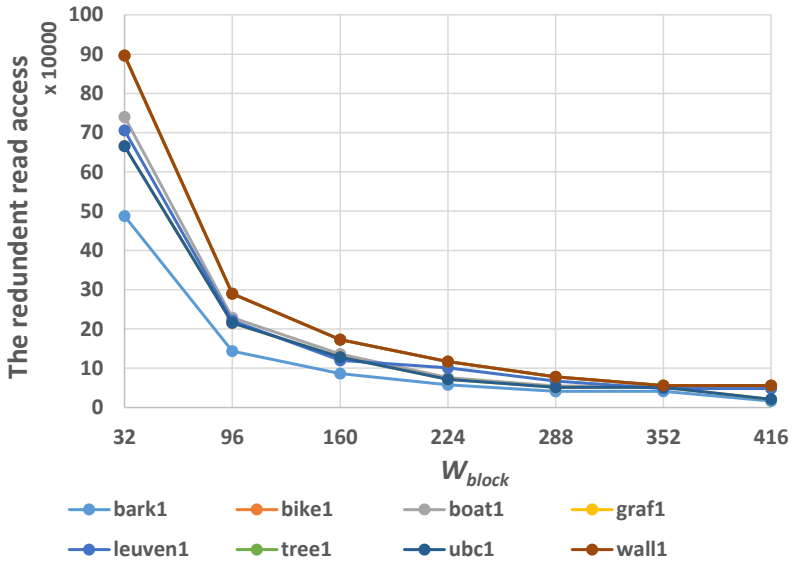


그림 3.6 W_{block}와 block 단위 연산에 의한 불필요한 중복 접근량

3.2.2 Local-patch down-sampling 방안

외부 메모리 접근량을 감소시키기 위하여, 본 논문은 external Gaussian image buffer에 저장되는 image를 가로, 세로 방향으로 각각 1/2 down-sampling하는 방안을 제안한다. 이 경우, Gaussian image를 외부 메모리에 쓰고 읽어 오기 위한 bandwidth가 75%감소할 수 있다. Local-patch down-sampling 방안이 적용하였을 때 동작은 그림 3.7과 같다. Gaussian filter에 의해 생성된 Gaussian image는 down-sampler에 의해 가로 세로 방향으로 1/2 down-sampling된다. 그리고 down-sampling된 image는 external Gaussian image buffer에 저장된다. Key-point가 검출되고, key-point 주변 local-patch가 필요하면 interpolator는 외부 메모리에서 Gaussian image의 일부 데이터를 가져오고, interpolation을 수행하여 온전한 local-patch를 생성한다. 제안된 SIFT 하드웨어의 interpolator는 linear interpolation을 통해 local-patch를 재구성한다. 완성된 local-patch는 GMO calculator에 전달된다. 이와 같은 기능은 SIFT 하드웨어에서 외부 메모리 bandwidth를 줄이는 역할을 수행하는 BW optimization module 내부에 구현된다.

일반적으로 image를 down-sampling하기 전에 anti-aliasing filtering을 수행하여 영상에서 고주파 성분을 제거한다. 이는 aliasing 문제로 인한 영상의 왜곡을 방지하고 SIFT 알고리즘의 정확도 감소를 막는다. 반면 Gaussian image는 별도의 anti-aliasing filtering을 수행할 필요가 없다. 왜냐하면 Gaussian image는 이미 Gaussian filter를 통해 low-pass filtering된 영상이기 때문이다. 제안된 SIFT 하드웨어는 외부 메모리에 scale 1인 Gaussian image를 저장한다. 따라서 local-patch down-sampling 방안은 Gaussian image를 down-sampling해야 한다. Scale 1인 Gaussian image는

Gaussian sigma가 $\sigma_1 = \sigma_0 \times 2^{1/3}$인 finite impulse response (FIR) Gaussian filter $g(n)$로 filtering된 영상이다. $g(x, \sigma_1)$의 주파수 영역을 살펴보기 위해 $g(n)$의 discrete Fourier transform (DFT) 결과인 $|G(\Omega)|$를 살펴보면 그림 3.8과 같다. 이해의 편리성을 위해 그림 3.8에서 제시된 DFT의 결과는 sampling domain인 가로축을 radian으로 변형하였다. $|G(\Omega)|$의 30dB stop-band corner frequency를 보면 $0.223 \times 2\pi$ radian이다. 즉 stop-band corner frequency가 $1/2\pi$ 보다 작으므로, 이 filter로 low-pass filtering된 scale index 1인 Gaussian image는 1/2 down-sampling되어도 aliasing 문제가 발생하지 않는다. 그러므로, 제안된 ASIFT 하드웨어는 추가적인 anti-aliasing filter를 추가하지 않고 Gaussian image에 대한 down-sampling을 수행한다.

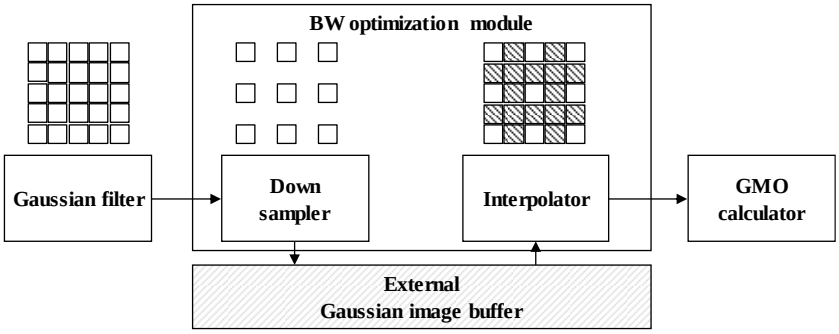


그림 3.7 Local-patch sampling 방안의 적용

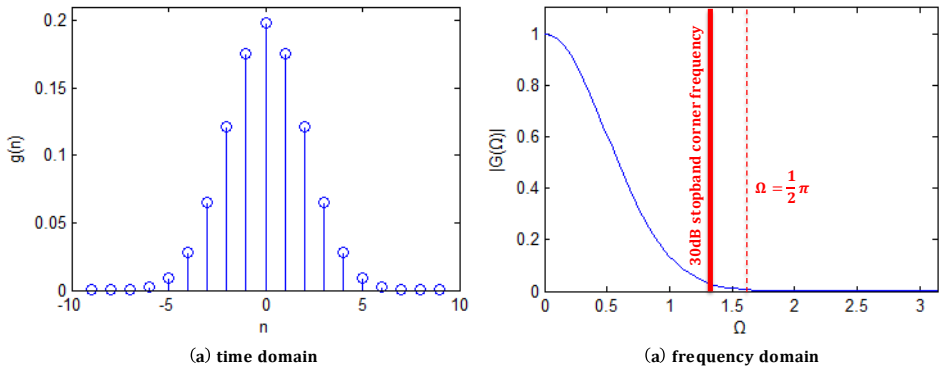


그림 3.8 scale index가 1인 Gaussian filter의 time/frequency domain

3.2.3 Gaussian image의 less signification bits 제거

기존에 제안된 대부분의 SIFT 하드웨어는 Gaussian image의 한 pixel을 저장하는데 3bytes를 할당한다. 이중에서 1byte는 integer part를 저장하는 공간이고, 나머지 2bytes는 fractional part를 저장하는 공간이다. 만일 Gaussian image pixel에 할당된 bit 수를 감소하여도 SIFT feature의 정확도에 큰 영향이 없다면 효과적으로 외부 메모리 bandwidth를 효과적으로 줄일 수 있는 방안이 될 수 있다.

외부 메모리에 저장되는 Gaussian image pixel에 할당된 bit 수를 감소시켰을 때 local-patch에 포함된 pixel의 세밀도가 감소할 수 있다. 하지만 SIFT 알고리즘은 local-patch 내부 영상의 pixel gradient를 histogram에 누적하여 descriptor를 생성하기 때문에 각각의 pixel의 fraction-bit 데이터를 제거한다고 해도 descriptor에 미치는 영향은 크지 않을 수 있다. 이 말은 곧 Gaussian image pixel에 할당된 bit 수를 일정 정도 제거하여도 SIFT 알고리즘의 성능에 미치는 영향은 미미할 수 있다는 것이다. 이 부분을 실험적으로 확인하기 위하여, Mikolajczyk *et al.*가 제시한 8개의 Oxford test image set과 'matching score'라는 image feature 알고리즘의 정확도를 측정하는 metric을 이용하여 실험을 진행하였다. 그 결과는 그림. 3.9와 같다. 할당된 bit 수가 24일때 실험에 사용된 test image set에 대한 평균 matching score는 32.2%정도다. Matching score는 검출된 key-point 수 대비 얼마나 많은 correct matches를 확보하였는지 보여주는 metric이기 때문에 높을수록 image feature 알고리즘의 성능이 높음을 의미한다. 그림 3.9에서 보여주듯이 할당된 bit를 하나씩 줄여갈 때 matching score는 32% 이상을 유지하는 것을 볼 수 있다. 하지만 8-bit 미만인 경우 급격하게 matching score가 감소한다. 이를

통해 Gaussian image pixel에 할당된 bit 중 integer part에 할당된 bit가 중요하고, fractional part에 할당된 bit는 덜 중요하다는 것을 확인할 수 있다. 따라서 제안된 SIFT 하드웨어는 Gaussian image pixel의 less significant bits이 fraction part를 외부 메모리에 저장하지 않는다. 이를 통해 Gaussian image를 외부 메모리에 저장 또는 읽는데 발생하는 bandwidth가 66.6% 감소하게 된다.

Gaussian image 뿐만 아니라 SIFT 알고리즘이 생성하는 다양한 중간 연산 데이터를 SIFT 하드웨어에 저장된다. 이 데이터들에 할당된 bit 수는 표 3.3과 같다.

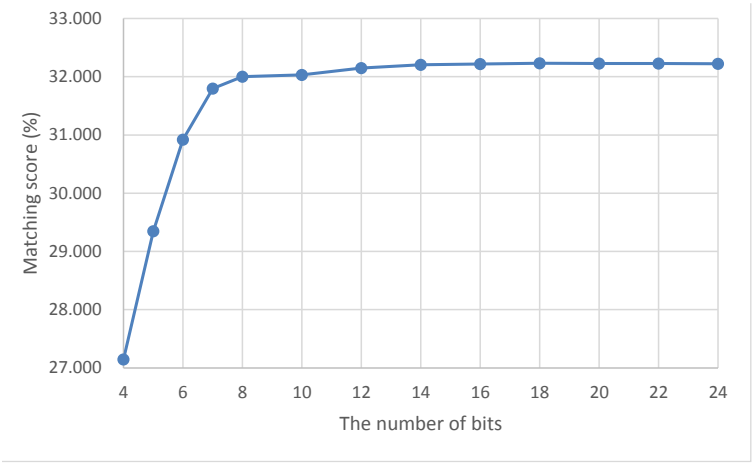


그림 3.9 Gaussian image pixel에 할당된 bit 수 대비 matching score

표 3.3 SIFT 알고리즘의 중간 데이터에 할당된 bit 수

Data	**Word length (Bits)**	
	Integer	**Fraction**
Source image	8	0
DoG pixel	9	16
Gradient magnitude	9	8
Gradient orientation	9	8
Each element of descriptor	8	0

3.2.4 Bandwidth 최적화 방안이 적용된 SIFT 하드웨어 구조

본 논문이 제안하는 실시간 SIFT 하드웨어 구조는 그림 3.10과 같다. 제안된 SIFT 하드웨어는 Gaussian image를 외부 메모리에 저장한다. 이때 급격하게 증가되는 외부 메모리 bandwidth를 줄이기 위한 3가지 방안은 회색 block으로 표시된 module들에 구현되었다. 빗금 친 block으로 표시된 것은 내부, 외부 메모리를 의미하고, 흰색 block은 일반적인 연산 module을 의미한다.

제안된 SIFT 하드웨어는 외부 메모리에서 source image를 입력 받아 source buffer에 저장한다. Gaussian filtering을 수행하기에 충분한 데이터가 source buffer에 쌓이면 이 데이터는 Gaussian filtering (GF) & DoG generator에 전달된다. GF & DoG generator는 Gaussian pyramid를 생성하고, 이를 이용하여 DoG image들을 생성한다. 생성된 DoG image들은 DoG buffer에 전달되고, 동시에 scale index가 1인 Gaussian image는 BW optimization module로 보내진다. 실제 SIFT 하드웨어는 다음 octave에 대한 SIFT 연산을 수행하기 위하여, scale index가 3인 Gaussian image를 별도로 외부 메모리에 저장한다. 그림 3.10은 표현의 간결성을 위해 다음 octave를 위해 필요한 동작을 수행하는 관련 module은 표현하지 않았다. DoG buffer에 key-point를 검출하기 충분한 데이터가 쌓이면 key-point detector를 이를 읽어가서 key-point를 검출한다. 검출된 key-point는 3개의 key-point FIFO에 저장된다. BW optimization module가 전달 받은 scale index가 1인 Gaussian image는 sub-sampler에 의해 down-sampling되고, truncator에 의해 fractional-bit가 제거된다. 그러면 Gaussian image의 데이터 크기는 기존의 8.3%가 된다. 따라서 external Gaussian image buffer에 데이터를 쓰기 위해 접근하는 데이터량 역시 기존 대비 8.3%로 감소하였다.

검출된 key-point는 3.2.1절에서 설명한 규칙에 따라 적절한 key-point를 BW optimization module에 전달한다. 그러면 BW optimization module은 외부 메모리에서 해당된 local-patch를 읽어온다. 이때 overlap detector는 외부 메모리에서 읽어와야 하는 pixel이 reuse buffer에 존재하는지 검사한다. 만일 reuse buffer에 저장한다면 이를 읽어내서 interpolator에 전달한다. 이와 동시에 이 pixel은 다음 local-patch를 처리할 때 재사용되기 위해 reuse buffer의 다른 주소에 저장된다. 만일 요청된 pixel이 reuse buffer에 없다면 외부 메모리에서 이를 읽어온다. 확보된 down-sampling된 local-patch는 interpolator에 의해 완전한 형태로 계산된 다음 Gaussian filter로 전달된다. 만일 현재 처리하는 key-point가 scale 1보다 크다면, 확보된 local-patch의 scale을 높이기 위해 Gaussian filtering이 이뤄진다. Scale이 맞춰진 local-patch는 GMO calculator에 전달되고, 계산된 GMO data는 GMO buffer에 저장된다. 최종적으로 descriptor generator는 GMO 데이터를 이용한 해당 key-point의 descriptor를 생성한다.

그림 3.10에서 보여주는 제안된 SIFT 하드웨어는 그림 3.1에서 보여준 Huang *et al.* 이 제시한 SIFT 하드웨어와 내부 메모리 크기가 다르다. 제안된 SIFT 하드웨어는 block-based raster scan order로 key-point를 검출하여 local-patch 재사용률을 높였다. 이를 위해 관련된 연산을 수행하는 GF & DoG generator와 key-point detector 역시 block 단위로 동작한다. 따라서 이 module에 입력 데이터와 출력 데이터를 저장하는 source buffer와 DoG buffer의 width는 image width에서 96으로 짧아졌다. Source buffer의 경우, block의 경계에 위치한 pixel의 Gaussian filtering을 위해서는 block 밖 15 pixels (=Gaussian filtering window의 tap 수의 절반)이 추가적으로 필요하다.

또한 3x3 DoG window를 구성하기 위해 1개의 pixel이 더 필요하다. 이렇게 총 16 pixels이 96 크기의 block의 좌우 경계에 더 필요하다. 따라서 하나의 block을 처리하기 위해서 외부 메모리에서 읽어와야 하는 데이터의 width는 128이 된다. Source buffer의 height는 Gaussian filtering을 수행하기 위해 31 pixels(=Gaussian filtering window의 tap 수)이 필요하다. 여기에 1 pixel line이 추가적으로 필요하다. 이는 외부 메모리에서 읽어온 데이터를 저장하는 공간과 GF & DoG generator에 전달한 데이터를 읽어오는 공간이 항상 분리되기 위해 완충 영역이다. 결과적으로 Source buffer의 크기는 $128 \times 32 \times 8bits$=32.8Kbits로 결정된다.

3×3 크기의 DoG window를 구성하기 위하여, block의 경계에 위치한 pixel은 block 밖의 1개의 추가 pixel이 필요하다. 따라서 DoG buffer의 width는 block의 width=96과 좌우 1개씩 추가된 pixel을 더한 98로 결정된다. DoG buffer의 height는 3×3 DoG를 구성하기 위해 3이 되어야 한다. 단, 1 pixel line은 3개의 DoG pixel registers로 대체될 수 있다. 따라서 DoG buffer의 height는 2가 된다. DoG image는 총 5개 scale이 존재하므로, DoG buffer의 크기는 $98 \times 2 \times 5 \times 25\text{bits}$ =24.5Kbits다. 제안된 하드웨어는 GMO buffer를 2개 사용한다. GMO buffer는 GMO calculator가 쓰고, descriptor generator가 읽는 중간 buffer다. 그런데, descriptor generator는 GMO buffer의 데이터를 읽고 있을 때, GMO calculator는 GMO buffer에 데이터를 쓸 수 없다. 왜냐하면 descriptor generator는 dominant orientation을 계산하는데 1번, gradient histogram을 계산하는 1번, 총 2번 읽혀야 하기 때문이다. 따라서 GMO buffer가 descriptor에 독점되고 있을 때 GMO calculator가 데이터를 쓸 추가적인 GMO buffer가 필요하다. 이를 통해 GMO generator와 descriptor

generator의 utilization이 증가될 수 있다. 결과적으로 GMO buffer는 240.24Kbits의 크기를 갖게 된다.

제안된 SIFT 하드웨어는 Huang *et al.*이 제시한 SIFT 하드웨어와 내부 메모리 크기 외에 key-point detector와 GMO calculator의 의존성이 다르다. Huang *et al.*이 제시한 하드웨어는 key-point를 고려하지 않고 GMO 데이터를 내부 메모리에 저장한다. 따라서 GMO buffer의 크기가 크다. 대신 Gaussian image buffer가 최소한의 데이터만 가지고 있어도 되므로 그 크기가 작다. 제안된 SIFT 하드웨어는 GMO buffer의 크기를 작게 하고, 대신 Gaussian image buffer에 전체 image를 저장하도록 설계하였다. 대신 이 buffer를 외부 메모리에 위치시켰다. GMO buffer의 크기를 최소화 하기 위해, 이 buffer는 1개의 local-patch만을 위한 데이터를 저장하도록 하였다. 따라서 GMO calculator는 key-point detector에 의존하여 동작하게 되었다. 이러한 의존성은 GMO calculator가 key-point를 전달 받은 다음 동작을 시작하므로 이에 따른 delay가 발생할 수 있다. 하지만 이는 SIFT 하드웨어의 전체 동작 시간에는 큰 영향을 미치지 못한다. 왜냐하면 descriptor generator가 전체 동작의 bottle-neck이기 때문이다.

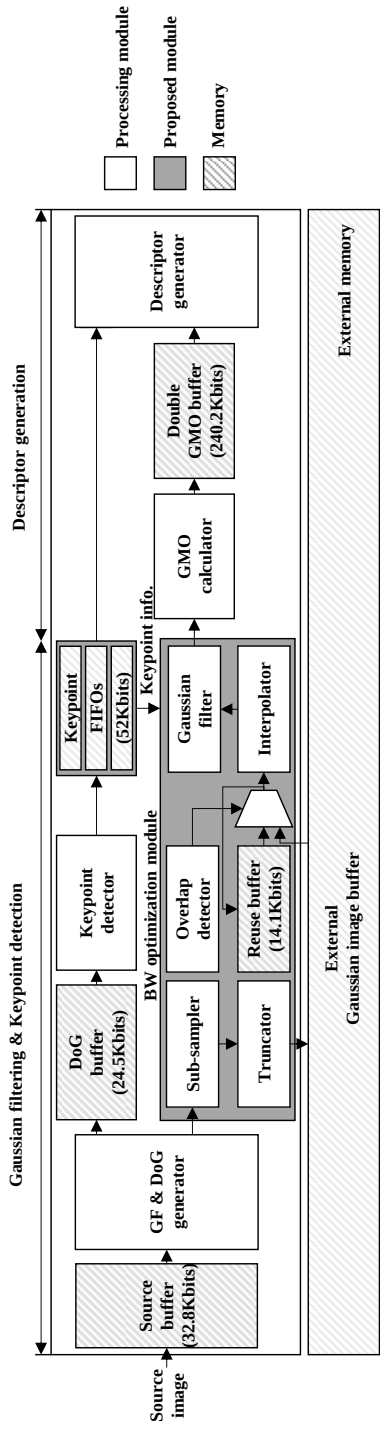


그림 3.10 제안된 SIFT 하드웨어의 block diagram

3.3 SIFT 하드웨어에 대한 실험 결과

3.3.1 SIFT 하드웨어의 스펙

제안된 SIFT 하드웨어는 기존에 제안된 SIFT 하드웨어 중에서 정확도와 속도 측면에서 state-of-the-art인 Huang *et al.* 의 하드웨어와 하드웨어 비용 측면을 비교한다 [26]. 이는 표 3.4과 같다. 두번째 열은 Huang *et al.* 이 제시한 SIFT 하드웨어의 메모리 크기를 보여준다. 세번째 열은 제안된 SIFT 하드웨어의 baseline이라고 할 수 있는 Kim *et al.*의 하드웨어 구조를 메모리 크기를 보여준다 [28]. 단, 세번째 열의 하드웨어는 Kim *et al.* 이 제안한 하드웨어와 Gaussian filter의 크기, 연산의 기본 block 단위 등이 달라 실제 메모리 크기는 다르다. Baseline 하드웨어와 Huang *et al.* 의 하드웨어를 비교하면, block 단위 연산으로 인해 source buffer와 DoG buffer의 크기가 크게 감소하였다. 또한 하나의 local-patch의 데이터만 저장하는 구조 이므로 GMO buffer의 크기가 크게 감소하였다. 반면 Gaussian image buffer는 크게 증가하였다. 그러나 제안된 하드웨어는 Gaussian image buffer를 외부 메모리에 저장하기 때문에 하드웨어 비용이 증가하지 않는다. 표 3.4의 (*)는 외부 메모리를 의미한다. 네번째 열은 baseline 하드웨어에 local-patch reuse 기능이 추가되었을 때의 메모리 크기를 보여준다. 이 방안을 이용할 경우 Gaussian image buffer에 scale 1번인 Gaussian image를 저장하므로 그 크기가 줄었다. 단 이러한 메모리 크기 감소는 Kim *et al.* 이 제안한 구조에 따른 것이므로 본 논문의 contribution은 아니다 [26]. Local-patch를 재사용하기 위해 reuse buffer가 추가되었고, 관련 연산을 위해 logic의 크기가 증가하였다.

다음으로 local-patch sampling 방안 적용되면서 Gaussian image buffer, reuse buffer의 크기가 크게 감소하였고, 이 동작을 제어하기 위한 1.4K gates의 logic이 추가되었다. 마지막으로 fraction-bit truncation을 통해 Gaussian image buffer와 reuse buffer의 크기가 감소하였다. 이로 인한 matching score 감소량은 0.2%에 불과하므로, 이로 인한 성능 감소는 미미하다. 결과적으로 제안된 SIFT 하드웨어는 Huang *et al.* 이 제안한 하드웨어에 비해 10.93% 더 적은 내부 메모리를 사용한다는 것을 확인하였다. 제안된 SIFT 하드웨어는 130nm process technology를 통해 합성할 경우 동작 가능한 최대 주파수는 190MHz다. 사용하는 내부 메모리 총량은 396.03Kbits다. 이 크기는 표 3.4에서 제시한 내부 메모리 크기보다 조금 더 크다. 그 이유는 표 3.4에 포함되지 않는 processing module에서 사용하는 내부 메모리가 조금 더 있기 때문이다.

표 3.4 메모리 크기 비교

Temporary buffer	**Huang [26] (Bits)**	**No BW optimization [28] (Bits)**	**Proposed BW optimization (Bits)**		
			Local-patch reuse (A)	**A + local-patch sampling (B)**	**B + fraction-bit truncation**
Source buffer	128.00K	32.77K	32.77 K	32.77 K	32.77 K
DoG buffer	230.40K	24.50K	24.50 K	24.50 K	24.50 K
Gaussian image buffer	138.24K	29.03M*	9.68M*	2.42M*	806.40K*
Key-point buffer	52.48K	52.03K	52.03K	52.03K	52.03K
GMO buffer	2778.88K	240.24K	240.24K	240.24K	240.24K
Reuse buffer	0.00	0.00	144.77K	49.92K	14.08K
Trade-off	-	-	Logic increase by 14K gates	Logic increase by 1.4K gates	Matching score decrease by 0.2%

* External memory

3.3.2 외부 메모리 bandwidth 요구량 분석

제안된 SIFT 하드웨어가 외부 메모리에서 Gaussian image를 읽어오는데 드는 데이터량은 표 3.5와 같다. 이 실험은 그림 3.11이 보여주는 8개의 Oxford test image를 사용하여 측정되었다. 표 3.5의 두번째 열은 각각의 test image에서 검출된 key-point의 수를 보여주고, 세번째 열은 test image의 크기를 보여준다. 네번째 열은 제안된 BW 최적화를 적용하지 않은 baseline 하드웨어의 외부 메모리 접근량을 보여준다. 표의 마지막 행은 평균적인 test image 크기 대비 외부 메모리 접근량의 비율을 보여준다. 제안된 bandwidth optimization을 적용하지 않으면 평균적으로 source image의 24.74배의 외부 메모리 접근이 발생한다. 5번째 열은 local-patch 재사용 방안이 적용되었을 때 Gaussian image 읽기를 위한 외부 메모리 접근량을 보여준다. 이 경우 외부 메모리 접근량이 50% 감소하여 source image 크기 대비 12.39배로 줄어들었다. 여기에 추가적으로 local-patch sampling 방안을 적용하면 평균 외부 메모리 접근량은 source image 크기 대비 3.52배로 줄어들었다. 마지막 열은 제안된 모든 bandwidth 최적화 방안이 적용되었을 때 Gaussian image를 읽어오는데 소모되는 외부 메모리 접근량을 보여준다. 평균적으로 source image 대비 1.17배 외부 메모리 접근이 발생하였고, 이것은 제시된 방안이 적용되지 않았을 때 대비 95% 감소된 수치다. 결과적으로 제안된 bandwidth 최적화 방안은 극적으로 외부 메모리 접근량을 감소시키며, 평균적으로 source image의 1.17배의 외부 메모리 접근만 발생하게 됨을 확인하였다. 그러므로 제안된 SIFT 하드웨어는 실질적으로 다양한 SoC에 병합되어 사용될 수 있는 구조라고 말할 수 있다.

(a) ‘Bikes’ (b) ’Tree’

(c) ‘Graf’ (d) ’Wall’

(e) ‘Bark’ (f) ’Boat’

(g) ‘Leuven’ (h) ’Ubc’

그림 3.11 Oxford test images

표 3.5 External Gaussian image에 대한 읽기 접근량

Image	Number of key-points	Image size (Bytes)	No BW optimization [28] Read access size (pixels)	Local-patch reuse (A) Read access size (pixels)	A + Local-patch sampling (B) Read access size (pixels)	B + Fraction-bit truncation (C) Read access size (pixels)
Bark1	1841	376,832	14,962,596	6,642,693	1,920,684	640,228
Bike1	1577	690,432	1,344,768	7,302,942	2,031,717	677,239
Boat1	1550	565,760	12,985,770	6,703,269	1,918,128	639,376
Graf1	1134	512,000	9,629,103	5,323,380	1,485,765	495,255
Leuven1	1036	537,600	8,767,632	4,856,745	1,366,344	455,448
Trees1	2867	690,432	23,087,499	10,809,540	3,112,689	1037,563
Ubc1	1260	512,000	10,458,594	5,291,511	1,500,756	500,252
Wall1	2308	690,432	18,583,203	9,406,875	2,673,168	891,056
Average read accesses / pixel			**24.74**	**12.39(50.08%)**	**3.52(14.24%)**	**1.17(4.75%)**

3.3.3 동작 속도

제안된 SIFT 하드웨어의 동작 속도를 기존에 Huang *et al.*이 제시한 SIFT 하드웨어의 동작 속도와 비교한다. SIFT 알고리즘을 하드웨어로 구현하는 주 목적은 고속으로 알고리즘을 수행하는 것이므로 가장 중요한 평가 항목이라고 말할 수 있다. 우선, 두 하드웨어를 공평한 조건에서 동작 속도를 비교하기 위하여, 제안된 하드웨어의 key-point detection 단계 (Gaussian filtering 및 key-point detection)와 descriptor generation 단계 (GMO calculator 및 descriptor generation)가 Huang *et al.* 이 제시한 SIFT 하드웨어와 마찬가지로 순차적으로 동작하도록 제어 방법을 변경하였다 [26]. 다시 말해, key-point가 검출되어 descriptor가 생성되기 시작하면, key-point detection 단계에 포함된 모듈들은 동작을 멈춘다. 그리고 descriptor 생성이 완료되면, key-point detection 단계가 재개된다. 또한 Huang *et al.* 이 제안한 SIFT 하드웨어의 최대 동작 속도는 100MHz이므로 제안된 SIFT 하드웨어의 동작 주파수를 100MHz로 설정하였고, 동일하게 1bytes/cycle로 source image를 읽어오도록 외부 메모리 latency 세팅을 설정하였다. 마지막으로 동작 속도 측정에 사용된 source image는 Huang *et al.* 의 하드웨어가 VGA (640x480) 크기의 source image에서만 동작되므로, VGA 크기의 source image를 사용하였다.

표 3.6에는 위에서 설명된 환경에서 측정된 두 하드웨어의 동작 속도가 제시되어 있다. 실험에 사용된 test image는 Oxford test image를 VGA 크기로 scaling시킨 다음 사용하였다. 두번째 열에 제시된 Huang *et al.*이 제시한 하드웨어의 동작 속도는 [26]에서 제시한 하드웨어 동작 속도를 대표하는 식 (3.3)을 통해 계산된 식이다. 식 (3.3)은 다음과 같다.

Processing time of one VGA image
= 3.4 ms + # of features x 0.0331 ms (3.3)

세번째 열과 네번째 열은 제안된 순차적 연산 단계로 제어로 SIFT 하드웨어가 각각 [2]에서 제시된 descriptor generation 모듈을 사용할 때, 본 연구에서 설계된 descriptor generation 모듈을 사용할 때의 동작 속도를 보여준다. 세번째 열에서 제시된 동작 속도는 제안된 하드웨어의 Gaussian filtering DoG generation, key-point detection 에 걸리는 시간과 식 (3.3)에서 제시된 descriptor generator의 동작 시간 정보를 합하여 계산되었다. 두번째 열과 세번째 열을 비교하면, 제안된 SIFT 하드웨어의 descriptor generator를 제외한 나머지 부분의 동작 속도가 Huang *et al.*이 제시한 하드웨어보다 0.89배 느리다는 것을 확인할 수 있다. 이런 결과가 나온 이유는 제안된 SIFT 하드웨어는 Gaussian image 뿐 만 아니라 다음 octave를 수행하는데 필요한 down-sampling된 source image를 모두 외부 메모리에 저장하기 때문이라고 추측된다. 반면 Huang *et al.*의 하드웨어는 모든 메모리 공간을 외부 메모리보다 접근 속도가 빠른 내부 메모리로 사용한다.

네번째 열은 제안된 하드웨어를 순차적 연산 제어를 통해 동작 시켰을 때의 동작 속도를 보여준다. 이 경우 제안된 SIFT 하드웨어는 Huang *et al.*의 descriptor generator를 사용할 때에 비해 1.5배, Huang *et al.*의 전체 하드웨어에 비해 1.3배 더 빠른 동작 속도를 보여준다. 이와 같이 속도 향상이 이뤄진 부분은 본 연구에서 구현한 descriptor generator가 Huang *et al.*이 설계한 descriptor generator보다 더 빠르다는 것을 의미한다. 마지막 열은 제안된 SIFT 하드웨어의 key-point detection 단계와 descriptor generation

단계가 병렬적으로 동작할 때의 동작 속도를 보여준다. 이때의 동작 속도는 Huang *et al.*의 하드웨어에 비해 1.96배 더 빠르다. 이와 같이 동작 속도가 향상된 것은 key-point detection 단계와 descriptor generation 단계가 병렬적으로 동작할 수 있도록 설계되었기 때문이다. 제안된 하드웨어는 external Gaussian image buffer 사용한다. 외부 메모리는 저장 공간이 넓기 때문에 Gaussian image 전체를 저장할 수 있다. 따라서 GF & DoG generator와 key-point detector는 GMO calculator와 descriptor generator의 상태와 무관하게 높은 utilization으로 동작이 가능하다. GMO calculator와 descriptor generator는 역시 key-point와 Gaussian image 정보만 확보되면 앞선 모듈과의 큰 의존성 없이 동작이 가능하다. 따라서 제안된 SIFT 하드웨어는 key-point detection 단계와 descriptor generation 단계가 병렬적으로 동작이 가능한 것이다. 반면 Huang *et al.*의 SIFT 하드웨어는 Gaussian image buffer의 내부 메모리로 구현하여, 반응 속도는 빠르지만, 하드웨어 비용을 절약하기 위해 메모리 공간을 최소한으로 한정하였다. 일반적으로 Descriptor generation 단계는 Gaussian filtering 연산 보다 느리다. 따라서 Gaussian filtering과 관련된 모듈은 descriptor generation 단계가 Gaussian image buffer의 데이터를 소비해줄 때까지 기다릴 수 밖에 없는 의존성이 발생한다. 이런 이유로 인해 이 하드웨어는 key-point detection 단계와 descriptor generation 단계가 병렬적으로 동작할 수 없다.

표 3.7은 제안된 SIFT 하드웨어의 최대 동작 속도를 보여준다. 동작 주파수는 SIFT 하드웨어의 합성 결과를 참조하여 190MHz로 설정하였고, 사용된 외부 메모리는 2.46bytes/cycle의 bandwidth로 데이터를 제공하도록 구성하여 실험하였다. 실험에 사용된 test image는 Oxford test image 8개를

VGA (640x480)크기와 HD (1280x720)크기로 scaling한 것이다. 표 3.7의 세번째 열은 VGA크기의 image를 1초에 몇 장 처리할 수 있는지 보여준다. 제안된 하드웨어는 전체 image에 대하여 평균적으로 114.68fps로 처리 가능하다. Source image의 크기를 HD로 증가시켰을 경우의 동작 속도는 6번째 열에서 보여준다. 제안된 SIFT 하드웨어는 HD image는 평균적으로 40.83fps의 속도로 처리 가능함을 보여준다. 결론적으로 제안된 SIFT 하드웨어는 HD 영상에 대해 실시간 동작이 가능하다고 말 할 수 있다.

표 3.6 VGA image에 대한 동작 속도 (동작 주파수 100MHz 기준)

Image	Huang [26] Serial (ms)	Proposed Serial w/ DG in [26] (ms)	Proposed Serial (ms)	Proposed Parallel (ms)
Graf1	28.79	33.18	23.60	15.35
Bark1	51.99	56.38	36.67	26.45
Boat1	33.75	38.14	2`6.03	17.64
Tree1	41.17	45.56	29.87	20.56
Leuven1	26.17	32.56	21.53	13.18
Ubc1	28.49	32.88	22.76	14.70
Bike1	29.25	33.64	23.62	15.33
Wall1	30.77	35.16	23.15	14.59
Average	**33.80**	**38.19**	**25.90**	**17.23**

DG: descriptor generator

표 3.7 제안된 SIFT 하드웨어의 최대 동작 속도

(동작 주파수 190MHz)

Image	640x480 (VGA)		1280x720 (HD)	
	Number of key-points	Proposed (Frames/sec)	Number of key-points	Proposed (Frames/sec)
graf1	767	123.36	1781	55.41
bark1	1468	69.93	4592	22.37
boat1	917	108.93	2856	36.93
tree1	1141	89.55	4042	25.92
leuven1	688	141.57	1666	58.63
ubc1	758	131.05	2311	44.83
bike1	781	124.03	1998	50.57
wall1	827	129.06	3282	31.96
Average	**918.38**	**114.68**	**2816**	**40.83**

3.3.4 Feature matching 정확도

본 논문에서 제안된 bandwidth 최적화 방안들과 하드웨어 구현을 위해 floating-point 단위 연산을 fixed-point 단위 연산으로 변경시키는 일은 생성되는 SIFT feature의 정확도에 영향을 미칠 수 있는 사항들이다. 따라서 제안된 각각의 상황들에 SIFT feature의 정확도에 어떤 영향을 미치는지 실험을 통해 확인한다. 정확도 평가는 Mikolajczyk *et al.* 이 제시한 'matching score' metric을 이용하여 수행되고, Oxford test image를 이용하였다.

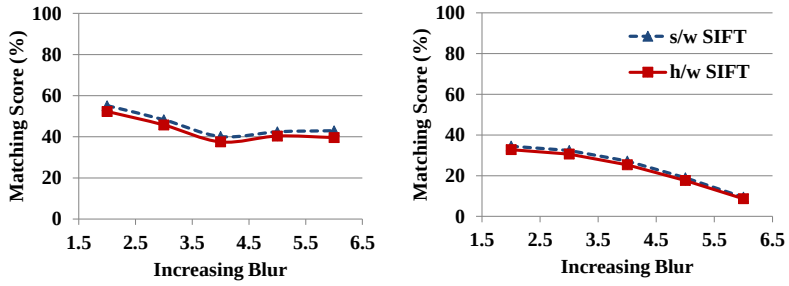
표 3.8은 제안된 bandwidth 최적화 방안이 matching score에 미치는 영향을 확인하기 위해 진행한 시험 결과를 보여준다. 두번째 열은 software로 구현된 SIFT 알고리즘의 matching score를 보여준다. 이 software는 Hess가 구현하여 공개한 SIFT software이다 [33]. 3번째 열은 fixed-point 연산

방식을 이용하는 baseline 하드웨어의 정확도를 보여준다. 표 3.8에서 볼 수 있듯 fixed-point 연산을 이용하면 floating-point 연산에 비해 평균 1.75% matching score가 감소한다. 4번째 열은 local-patch sampling 방안을 적용하였을 때 matching score를 보여준다. 이 경우 비록 그 정도가 미미하지만 약간의 matching score가 향상되었다. 그 이유는 Gaussian image에 대한 down-sampling과 interpolation을 통해 image blurring이 이뤄졌고, 이것은 pixel gradient를 명료성을 높여주었기 때문이라고 사료된다. Down-sampling과 interpolation에 의한 효과는 $\sigma = 0.713$로 Gaussian filtering하는 효과와 동일하다. 실제로 Gaussian image를 $\sigma = 0.713$로 다시 한번 더 Gaussian filtering 할 경우, 5번째 열과 동일한 32.23%의 평균 matching score를 보여주었다. 5번째 열은 Gaussian image pixel의 fractional part를 제거하였을 때의 효과다. 이 경우 0.23%의 matching score 감소가 발생하였다. Local-patch 재사용방안은 matching score에 아무런 영향을 미치지 않기 때문에 표 3.8에 제시하지 않았다.

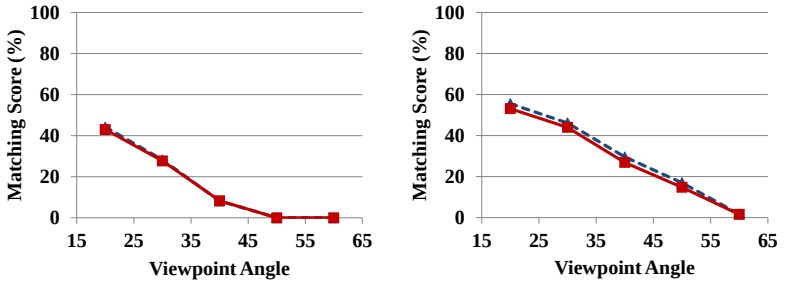
그림 3.12은 제안된 SIFT 하드웨어와 Hess에 의해 공개된 SIFT 소프트웨어의 matching score를 다양한 영상 변환 조건에서 보여준다. SIFT 하드웨어의 key-point detector는 key-point 검출 단계에서 key-point localization을 수행하지 않는다. 공평한 비교를 위해 SIFT 소프트웨어에서도 동일 연산을 수행하지 않도록 변형하였다. 그림 3.12에서 실선은 SIFT 하드웨어의 matching score를 보여주고, 점선은 SIFT 소프트웨어의 matching score를 보여준다. 평균적으로 두 SIFT 알고리즘 구현의 matching score 차이는 1.74%이다. 이것은 미미한 수준의 정확도 감소라고 볼 수 있다.

표 3.8 Bandwidth 최적화 방안이 matching score에 미치는 영향 (%)

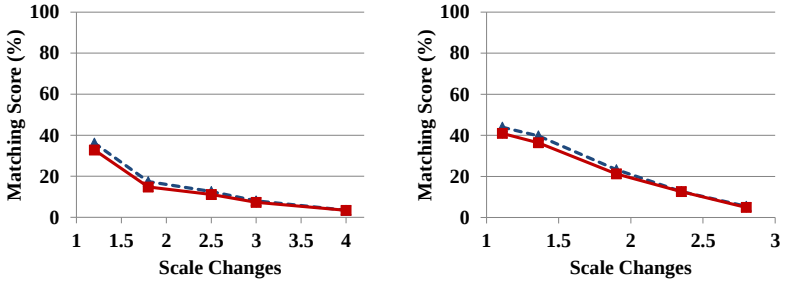
Test sets	s/w SIFT [17]	With fixed-point	With sampling	Without fraction-bit
Bikes	45.78	43.00	43.56	43.11
Tree	24.49	23.07	23.08	23.00
Graf	16.12	15.82	15.94	15.79
Wall	30.00	28.25	28.34	28.07
Bark	15.56	13.93	14.02	13.89
Boat	25.00	23.56	23.47	23.19
Leuven	61.27	59.38	59.85	59.63
Ubc	51.73	48.92	49.54	49.35
Average	**33.74**	**31.99**	**32.23**	**32.00**



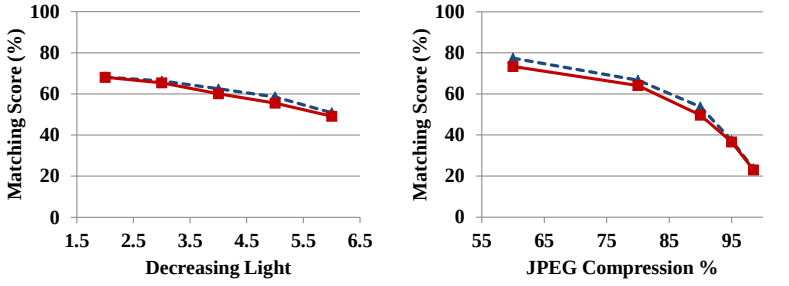
(a) ‘Bikes’ (b) ’Tree’



(c) ‘Graf’ (d) ’Wall’



(e) ‘Bark’ (f) ’Boat’



(g) ‘Leuven’ (h) ’Ubc’

그림 3.12 SIFT 하드웨어와 소프트웨어의 matching score 비교

제 4 장 ASIFT 하드웨어 구조

4장에서는 제안된 SIFT 하드웨어에 affine transform 기능을 추가하여 실시간으로 동작이 가능한 ASIFT 하드웨어 구조를 제안한다. 기존 affine transform 연산이 하드웨어로 구현될 경우 불합리한 이유와 이 문제를 해결한 새로운 affine transform 연산 방식을 제안한다. 그리고 효율적인 연산 재활용과 외부 메모리 bandwidth 감소를 통해 ASIFT 하드웨어가 높은 utilization으로 동작할 수 있는 전체 구조를 제안한다.

4.1 ASIFT 하드웨어에 적합한 affine transform 방식

4.1.1 새로운 affine transform 방식

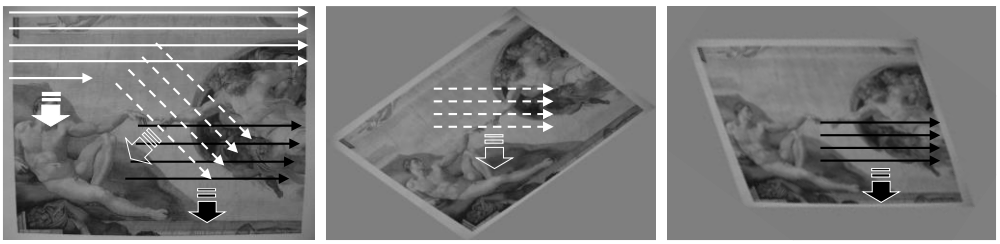
ASIFT 하드웨어가 고속으로 동작하기 위해서는 가장 복잡한 연산을 수행하는 SIFT generation module이 높은 utilization으로 동작할 수 있어야 한다. 그러기 위해서는 SIFT generation module가 입력 데이터를 요청할 때 affine transform module이 latency 없이 영상 데이터를 제공해줄 수 있어야 한다. 그림 4.1은 3종류의 image scan order를 보여준다. 그림 4.1 (a)는 source image와 함께 외부 메모리에 pixel이 저장된 순서를 흰색 실선 화살표를 통해 보여준다. 그림 4.1 (b)는 일반적인 affine transform 연산을 통해

transform된 source image와 SIFT generation module이 요청하는 pixel 순서를 점선으로 된 흰색 화살표로 보여준다. 이에 대응한 pixel의 순서는 그림 4.1 (a)의 점선으로 된 흰색 화살표와 같다. SIFT generation module이 이 pixel들을 요청할 경우 affine transform module은 외부 메모리에서 저장된 순서와 다르게 source image를 읽어와야 하고, 이때 외부 메모리 latency가 증가한다. 증가된 latency를 감소시키기 위하여, 본 논문은 새로운 affine transform 방식을 제안한다. 이 방식은 ASIFT 알고리즘이 카메라 포즈를 affine transform으로 모델링할 때 0으로 고정한 spin parameter (ψ)를 적절히 조절한다. 그 결과는 그림 4.1 (c)가 보여주는 또다른 affine transform된 image다. 그림 4.1 (c)의 검은색 실선 화살표는 이 image에 대한 SIFT generation module의 pixel 접근 순서를 보여준다. 이에 대응하는 pixel의 순서를 source image에서 찾아보면 그림 4.1 (a)의 검은색 실선 화살표와 같다. 여기서 확인할 수 있는 것은 이 pixel 접근 순서는 source image의 pixel이 외부 메모리에 저장된 순서와 방향이 같다는 것이다. 결과적으로, 이러한 데이터 접근은 기존 affine transform을 이용할 때 보다 외부 메모리에서 source image를 읽어오는 측면에 있어서 훨씬 효율적이고 빠르다.

그림 4.1 (c)와 같이 affine transform된 image를 얻기 위해서, 본 논문은 카메라의 spin parameter ψ를 $-\mu$만큼 회전시킨다. 다시 말해, 기존 affine transform A를 이용하여 transform된 image를 $-\mu$만큼 회전시킨다. 이 새로운 transform을 matrix B라고 표현한다. 새로운 affine transform matrix B를 이용한 transform 과정은 그림 4.2가 자세히 보여준다. 그림 4.2 (a)는 matrix A로 transform된 image를 보여준다. Parameter μ는 x축과 transform된 image의 의 위쪽 경계선 사이의 각도를 의미한다. 따라서 이 transform된

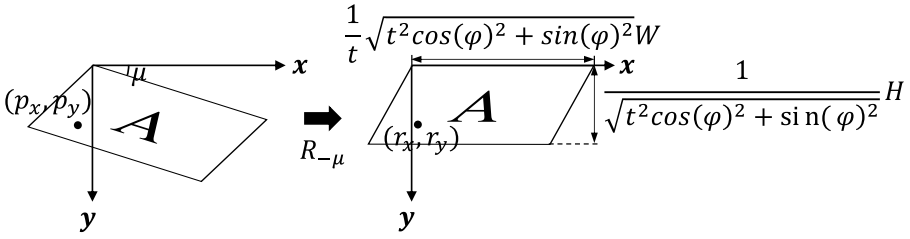
image를 $-\mu$만큼 회전시키게 되면 이 image의 위쪽 경계선이 x축과 평행하게 된다. 그러면 그림 4.1 (c)에서 보여준 새로운 형태의 transformed image가 생성된다. 식 (4.1)은 source image에 속한 pixel의 좌표 q와 이에 대응하는 matrix B를 통해 transform된 image의 pixel 좌표 r 사이의 관계를 보여준다.

$$r = Bq = R_{-\mu}T_{1,1/t}R_{\varphi}q, \qquad \mu = acos\left(\frac{cos\varphi}{\sqrt{cos\varphi^2 + \frac{sin\varphi^2}{t^2}}}\right) \tag{4.1}$$



(a) (b) (c)

그림 4.1 두 종류의 affine transform된 영상에 대한 raster scan 방향과 이에 대응하는 source image에서는 scan 방향



(a) (b)

그림 4.2 기존 affine transform matrix A와 새로운 matrix B 사이의 관계

SIFT 알고리즘은 rotation-invariant한 특성이 있다. 따라서 matrix A에 의해 transform된 image인 그림 4.1 (b)에서 추출한 SIFT feature와, matrix B의 새로운 rotation matrix $R_{-\mu}$에 의해 한 단계 더 회전한 그림 4.1 (c)에서 추출한 SIFT feature는 정확도에 있어서 이론적으로 차이가 없다. 하지만 구현에 있어서 새로 추가된 rotation matrix는 μ의 각도에 따라 interpolation을 필요로 하고, 이러한 연산 과정으로 pixel 값이 달라지는 문제는 SIFT feature의 정확도에 작은 영향을 미칠 수 있다. 이 부분을 검증하기 위하여 Mikolajczyk *et al.*이 [32]에서 제시한 'matching score' metric과 Morel *et al.*이 [15]에서 제시한 test image들을 이용하여 affine transform matrix 변경에 따른 정확도 차이를 측정하였다. 그 결과, matrix A를 사용하는 ASIFT 알고리즘 구현과 matrix B를 사용하는 ASIFT 알고리즘 구현의 matching score 차이는 0.2로 의미를 가질 수 없는 수준이었다. 결론적으로, 새로운 affine transform 방식을 이용하여도 ASIFT 알고리즘의 정확도에는 변화가 없다고 말할 수 있다.

식 (4.1)에 제시된 affine transform matrix B는 식 (4.2)와 같이 scaling matrix와 skewing matrix로 표현될 수 있다.

$$B = S_g T_{sx,sy} = \begin{bmatrix} 1 & g \\ 0 & 1 \end{bmatrix} \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} \quad (4.2)$$

Scaling matrix $T_{sx,sy}$ 는 $W \times H$ 크기의 image를 $sxW \times syH$ 크기로 scaling하는 기능을 담당한다. 여기서 scaling parameter (sx, sy)는 $0 < sx, sy \leq 1$인 범위에 해당한다. S_g는 skewing matrix로 image는 g만큼 기울이는 기능을 담당한다. 각각의 parameter sx, sy, g는 식 (4.3)을 통해 계산된다.

$$sx = \frac{1}{t}\sqrt{t^2 cos(\varphi)^2 + sin(\varphi)^2}, sy = \frac{1}{\sqrt{t^2 cos(\varphi)^2 + \sin(\varphi)^2}}, \quad (4.3)$$
$$g = tan\tau = \left(\frac{1}{t} - t\right)\sin(\varphi)\cos(\varphi)$$

새로운 affine transform matrix B 는 메모리 접근과 메모리 크기의 측면에서 기존 affine transform matrix보다 효율적이다. 그림 4.3은 식 (4.2)를 이용하여 affine transform을 수행하는 과정을 단계별로 보여준다. 그림 4.3 (a)는 source image를 보여준다. 일반적으로 source image의 데이터 크기가 크기 때문에 외부 메모리에 저장되어 있다. 새로운 affine transform matrix를 이용하여 source image의 pixel이 저장되어 있는 순서와 affine transform module이 source image의 pixel을 읽어오는 순서가 viewpoint와 무관하게 항상 raster scan order로 일치한다. 이 경우, 그림 4.1 (b)와 달리 외부 메모리 latency가 감소한다. 읽혀진 source image는 그림 4.3 (b)와 같이 $T_{sx,sy}$에 의해 down-sampling된다. Down-sampling된 image는 source image와 크기가 같거나 더 작다. 게다가 scaled image는 affine transform된 image와 달리 모든 viewpoint에 대하여 직사각형 형태이기 때문에 물리적 구조가 사각형 형태인

일반적인 메모리 장치에 저장되기 적합하다. 본 논문에서 제안하는 affine transform module은 ASIFT 연산에 사용되는 내부 메모리에 scaled image를 저장하여 메모리 운용의 효율성을 높인다.

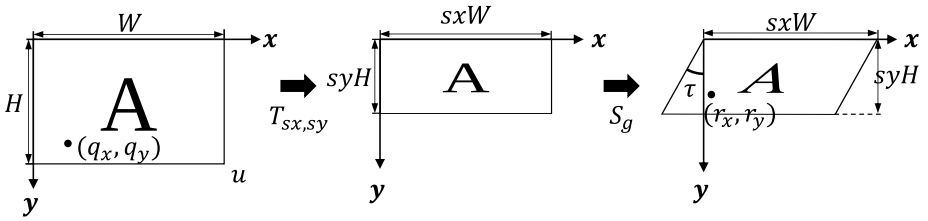


그림 4.3 새로운 affine transform matrix B를 이용한 transform 과정

4.1.2 내부 image buffer의 메모리 공간 최적화

Scaled image는 skew transform이 적용된 다음 square-shaped kernel를 이용해 SIFT feature 연산에 필요한 filtering 연산이 적용된다. 그 과정은 그림 4.4가 보여준다. 그림 4.4 (a)는 내부 image buffer에 저장된 scaled image를 보여준다. 이 image는 그림 4.4 (b)와 같이 S_g에 의해 skewing transform된 다음 내부 메모리에 저장된다. skewed image가 filtering을 수행하기 충분할 정도로 진행되면 square-shaped kernel로 구성된 filter에 의해 filtering된다. 이는 그림 4.4 (c)와 같다. 그림 4.4 (c)에서 검은 색 원은 filtering이 될 pixel을 의미하고, 주변 diamond로 표시된 pixel들은 이 filtering 연산에 필요한 이웃한 pixel들이다. 지금 설명된 바와 같이 일반적인 transform 과정과 filtering 과정이 affine transform module 내에서 이뤄진다면, skewed image를 저장하는 내부 메모리는 기울어진 image의 형태를 유지하기 위해 불필요한 background 공간까지 메모리에 할당해야 한다. 이로 인해 이 메모리의 width는 scaled image의 width보다 커져야 한다. 그림 4.4에 제시된 예의 경우, scaled image의 width는 16인 반면 skewed image를 저장해야 하는 내부 메모리의 width는 23이 되어야 한다. Affine transform module은 표 2.1에서 보여주는 모든 viewpoint에 대한 affine transform을 수행할 수 있어야 한다. 그러기 위해서는 skewed image를 저장하는 내부 메모리의 width는 source image의 width의 1.39배가 되어야 한다. 이것은 사용되지 않을 데이터까지 내부 메모리에 할당됨을 의미한다.

만일 scaled image에 skewing transform을 적용한 다음 square-shaped kernel을 적용하는 것 대신 scaled image에 skewing transform이 적용된 kernel을 적용한다면, 앞 서 설명한 불필요한 메모리 공간을 제거할 수 있다. 이 skewed kernel은 일반적인 square-shaped kernel에 S_{-g} 를 적용하여

생성한다. Skewed kernel을 이용한 image filtering은 그림 4.5을 가지고 설명할 수 있다. 그림 4.5 (a)와 (c)의 (c', r') 좌표의 검은색 원은 filtering이 수행된 pixel을 의미하며, 이는 그림 4.4 (c)의 (c, r)와 대응되는 pixel이다. 그림 4.5 (a)의 점선으로 표시된 기울어진 box는 skewed kernel의 filtering window를 의미하며, 그 안에 위치한 흰색과 회색의 diamond 문양은 filtering에 사용될 이웃한 pixel들을 의미한다. 흰색 pixel들은 integer-point에 위치한 pixel이고, 회색 pixel들은 integer-point 가 아닌 곳에 위치한 pixel들을 의미한다. 이 pixel들은 이웃한 integer-point의 pixel들을 참조하여 계산해야 한다. Skewed kernel은 scaled image를 raster scan order로 한 pixel씩 처리하고, filtering된 pixel은 그림 4.5 (b)와 같이 메모리에 저장된다. Filtering된 pixel들 중 integer-point에 위치한 pixel들은 흰색 원으로 표시되었고, 그렇지 않은 pixel들은 회색 원으로 표시되었다. Integer-point가 아닌 위치에서 존재하는 pixel들이 메모리에 저장되는 이유는, 다음 skewed kernel로 filtering할 때 필요한 pixel들이기 때문이다. 따라서 이 pixel들을 메모리에 저장해 두면 다음 skewed kernel을 이용해 filtering을 수행할 때, interpolation을 수행하여 재 생성할 필요가 없어진다. 이 pixel들은 좌측의 integer-point 좌표에 위치한 pixel을 저장할 공간이 비어 있기 때문에 이 공간에 저장한다. 그러면 non integer-point의 pixel을 저장하기 위한 추가적인 메모리 공간이 필요하지 않다. 첫번째 skewed kernel을 이용한 filtering이 완료되면, 다음 skewed kernel을 이용한 filtering이 시작되며, 이는 그림 4.5 (c)와 같다. 앞서 설명한 것과 같이 이때 필요한 non integer-point 좌표의 pixel들은 그림 4.5 (b)의 회색 pixel들과 같이 이미 메모리에 존재하기 때문에 이를 취한 추가적인 interpolation logic이 필요하지 않다.

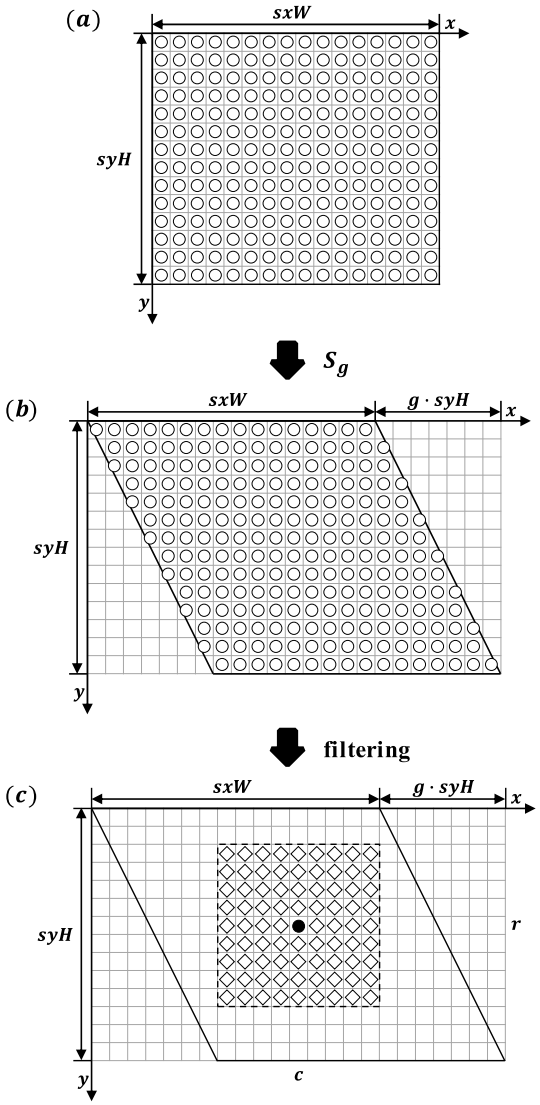
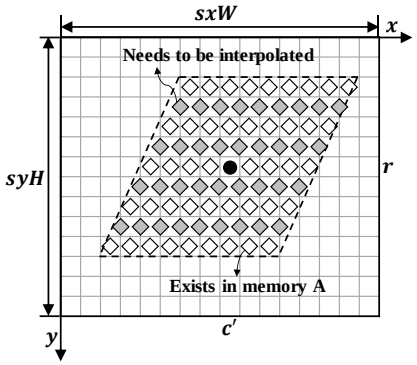
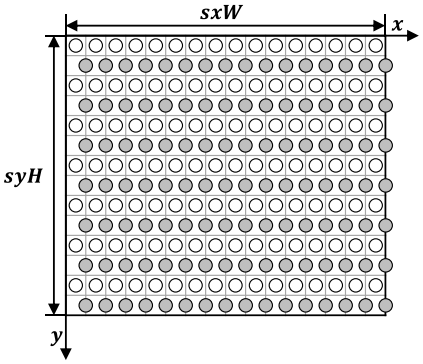


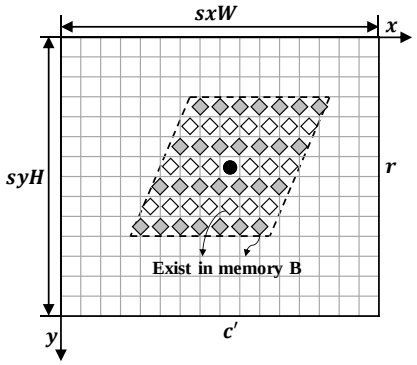
그림 4.4 Scaled image에 대한 skew transform 연산과 square-shaped kernel을 이용한 filtering 연산 설명



(a)



(b)



(c)

그림 4.5 Skewed kernel을 이용한 scaled image에 대한 filtering 연산 설명

4.2 ASIFT 하드웨어의 구조

4.2.1 기본 하드웨어 구조 및 scaling 연산 재사용

제안하는 ASIFT 하드웨어의 기초적인 구조는 그림 4.6과 같다. ASIFT 하드웨어는 크게 affine transform module과 SIFT generation module로 구성된다. ASIFT 하드웨어는 앞 절에서 설명한 새로운 affine transform 방식을 이용하여 연산을 수행한다. SIFT generation module은 3장에서 제안한 SIFT 하드웨어를 이용하여 구성된다. 여기에 앞서 설명한 skewed kernel을 이용한 filtering 연산 기능이 추가되었다.

그림 4.7이 보여주는 것과 같이 제안하는 ASIFT 하드웨어는 1개의 affine transform module과 1개의 SIFT generation module로 이뤄졌다. 따라서 다양한 viewpoint에 대한 ASIFT 연산을 수행하기 위해 2개의 module이 반복적으로 이용된다. Affine transform module은 외부 메모리에서 source image를 읽어온 다음 이를 down-sampling한다. 이 과정에서 affine transform module의 출력 데이터의 크기는 입력 데이터의 크기보다 작아진다. 이러한 입력 대비 출력 데이터 크기의 감소는 latency를 야기하고, 이로 인해 SIFT generation module의 utilization이 감소한다. 이러한 latency를 감소시키기 위해, 본 논문은 이전 viewpoint를 위해 수행된 scaling 연산 결과를 재활용할 수 있는 하드웨어 구조를 제안한다. 이를 위해 affine transform module에는 source image, pre-scaled image, scaled image 중 하나의 입력 데이터를 선택할 수 있는 source mux가 존재한다. Source image는 $ASIFT(0)$를 위해

선택된다. Scaled image는 이전 viewpoint의 ASIFT 연산에서 생성된 것으로, 만일 현재 연산 중인 viewpoint의 scaling parameter가 이전과 동일할 때 선택된다. 이 경우, 읽혀진 scaled image는 생성하고자 한 것과 동일한 것이므로 아무런 scaling 연산 없이 바로 SIFT generation module로 전달된다. Pre-scaled image는 나머지 경우에 선택되는 입력 데이터다. ASIFT 하드웨어가 $ASIFT(0)$를 수행하기 위해 source image를 읽어왔을 때, source image는 SIFT generation module 뿐 만 아니라 low-pass filter에도 제공된다. Affine transform 과정에서 scaling이 수행되기 전 aliasing 문제를 완화시키기 위해 source image의 고주파 성분을 제거해야 한다. 이 역할을 low-pass filter가 수행한다. Filtering된 source image는 pre-scaler에 의해 scaling parameter $(sx, sy) = (1, 1)$ 와 $(sx, sy) = (0.5, 1)$을 이용해 pre-scaled image로 변환된다. 만일 현재 처리하고 있는 viewpoint의 sx가 0.5이하라면 0.5x1 크기의 pre-scaled image가 affine transform module의 입력 데이터로 사용된다. 반대의 경우 1x1 크기의 pre-scaled image가 선택된다. 이와 같은 하드웨어 구조를 이용하게 되면 affine transform module은 생성해야 하는 scaled image와 가장 유사한 입력 데이터를 선택할 수 있다. 이는 affine transform module의 입출력 데이터의 크기 차이를 줄여줘서 latency를 감소시킬 뿐 만 아니라, 외부 메모리 접근량도 감소시키기 때문에 ASIFT 하드웨어의 utilization을 향상시키는데 기여한다.

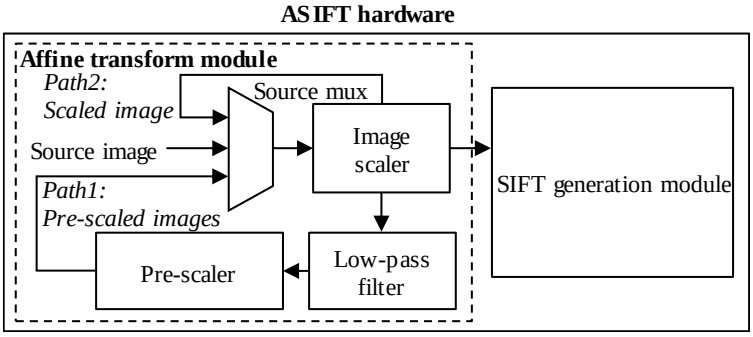
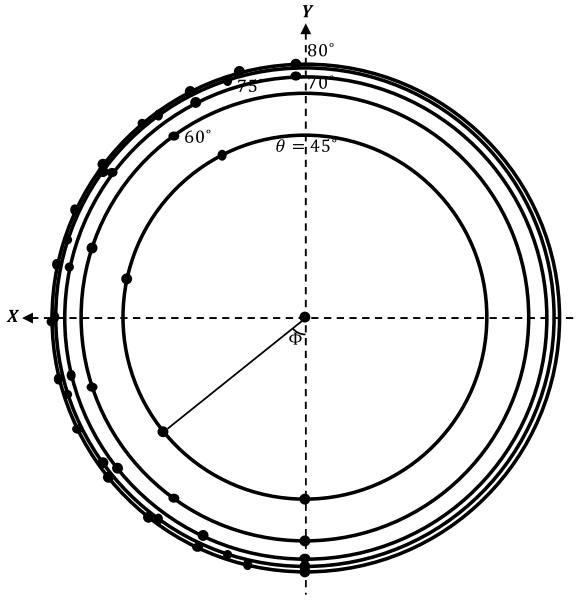


그림 4.6 ASIFT 하드웨어의 기본 구조

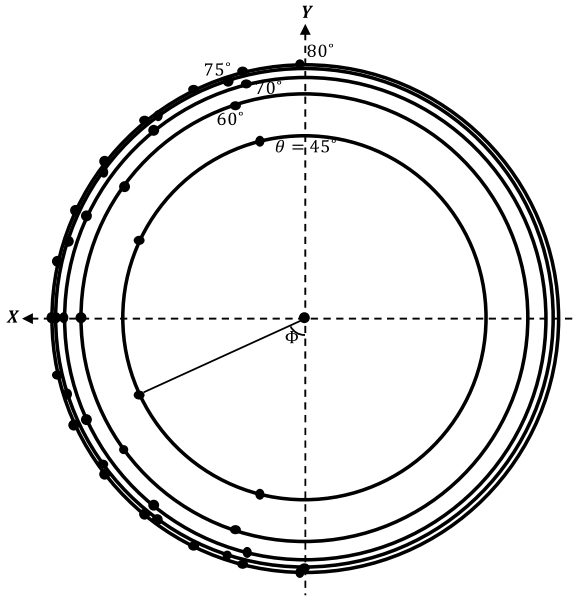
4.2.2 Affine transform parameter의 구성

4.2.2.1 Longitude offset 변경을 통한 scaled image 재사용률 향상

Scaled image의 재사용률을 높이기 위하여, 본 논문은 longitude offset은 변경한다. 만일 두 viewpoint의 latitude가 동일하고, longitude가 90° 를 기준으로 대칭된다면 두 viewpoint의 scaling parameter (sx, sy)는 같다. 그러면 하나의 viewpoint를 처리할 때 생성된 scaled image가 대칭되는 viewpoint를 처리할 때 온전히 재사용될 수 있다. 재사용될 수 있는 scaled image의 수를 늘리기 위해, 본 논문은 longitude offset은 변경한다. Latitude가 각각 45° , 60° , 69° , 80° 일 때, 이에 해당하는 longitude의 offset을 기존 0° 에서 각각 14° , 18° , 14, 1° 로 변경한다. 그림 4.7은 longitude 변경 전과 후의 viewpoint 위치를 보여준다. 그림 4.7 (a)는 Morel *et al.*이 제시한 viewpoint를 보여주고, (b)는 longitude를 변경했을 때의 viewpoint를 보여준다. 예를 들어, 기존 (sx_5, sy_5)는 (1, 0.5)이고, (sx_9, sy_9)는 (0.86, 0.58)이다. Longitude offset을 변경하고 나면 (sx_5, sy_5)와 (sx_9, sy_9) 모두 (0.92, 0.52)로 변경된다. 그러면 viewpoint 5를 위해 생성된 scaled image는 viewpoint 9에 대한 ASIFT 연산을 수행할 때 재사용될 수 있다. 결과적으로 Longitude offset 변경을 통해 재사용될 수 있는 scaled image 수가 6개에서 17개로 증가한다.



(a)



(b)

그림 4.7 Longitude offset를 변경하기 전 viewpoint (a)와 변경 후 viewpoint (b)

4.2.2.2 Tilt sampling step 증가에 의한 ASIFT 계산량 감소

ASIFT 알고리즘은 다양한 시점의 image를 시뮬레이션을 하기 때문에 SIFT 알고리즘 대비 처리해야 pixel의 수가 14배 증가하였다. 이러한 과도한 계산 요구량 때문에 1개의 SIFT generation module을 사용하는 ASIFT 하드웨어는 실시간으로 모든 viewpoint에 대한 ASIFT 연산을 수행하기 어렵다. 따라서 본 논문에서는 viewpoint를 선정할 때 사용되는 tilt sampling step (Δt)을 기존의 $\sqrt{2}$에서 2로 변경함을 통해 요구되는 계산량을 기존의 43%로 감소시켰다. Δt를 증가시키는 것은 ASIFT 알고리즘의 정확도에 영향을 줄 수 있다. 특히 fully affine invariant한 특성에 큰 영향을 미칠 수 있다. 이런 우려되는 지점을 검증하기 위하여 Morel *et al.*이 제시한 test image들을 이용하여 matching score를 측정하였다. 실험 결과, 평균 matching score는 기존의 83.57%로 감소하였다. 하지만 Morel *et al.*이 제시한 모든 viewpoint 차이에 대하여 모두 충분한 숫자의 correct matches를 확보하는 것을 확인하였다. 다시 말해, latitude가 76° 이하인 모든 상황에서 Δt 를 2로 결정하여도 안정적인 matching이 가능하다는 것을 확인하였다. 결론적으로, Δt를 $\sqrt{2}$에서 2로 변경하는 것이 감소된 연산량에 따라 불가피하게 matching score 하락하지만, 여전히 fully affine invariant한 특성은 유지된다고 말할 수 있다.

제안된 ASIFT 하드웨어는 Δt가 2일 때 선택되는 모든 viewpoint에 대한 시뮬레이션과 SIFT 연산을 수행한다. 이때의 viewpoint는 표 4.1이 보여준다. 첫번째 열부터 세번째 열까지는 각각 viewpoint index와 latitude, longitude를 보여준다. 그리고 4번째 열과 5번째 열은 viewpoint의 scaling parameter

(sx, sy)를 보여준다. 마지막 열은 viewpoint 사이의 scaled image 공유 가능 여부에 따라 맺어진 pair 정보를 보여준다. P_i라고 표시된 두 개의 viewpoint는 scaling parameter가 동일하여 scaled image를 공유할 수 있는 쌍을 의미한다. 반대로 U_j 로 표시된 단일 viewpoint는 scaled image를 공유할 수 없는 viewpoint를 뜻한다.

표 4.1 Tilt sampling step이 2일 때 scaling parameter와 scaled image 공유가 가능한 viewpoint pair 정보

Viewpoint index	Latitude (Degree)	Longitude (Degree)	sx	sy	Pair info.
0	0	0	1.00	1.00	U_0
1	60	18	0.96	0.52	P_0
2		54	0.71	0.70	P_1
3		90	0.50	1.00	U_1
4		126	0.71	0.70	P_1
5		162	0.96	0.52	P_0
6	76	0	1.00	0.25	U_2
7		18	0.95	0.26	P_2
8		36	0.82	0.30	P_3
9		54	0.62	0.40	P_4
10		72	0.39	0.64	P_5
11		90	0.25	1.00	U_3
12		108	0.39	0.64	P_5
13		126	0.62	0.40	P_4
14		144	0.82	0.30	P_3
15		162	0.95	0.26	P_2

4.2.3 ASIFT 하드웨어 구조 설명

본 논문에서 제안하는 ASIFT 하드웨어의 구조는 그림 4.8과 같다. 제안된 ASIFT 하드웨어는 affine transform module의 affine transform 과정에서 발생하는 latency를 감소시켜 SIFT generation module의 불필요한 대기 시간을 없앴다. 이러한 구성은 ASIFT 하드웨어의 utilization을 최대화 시켜 동작 속도 향상에 기여한다. 그림 4.8에서 흰색 block은 일반적으로 processing module을 의미하고, 빗금 친 block은 affine transform 연산과 관련된 module을 의미한다. 회색 block은 내부 메모리로 구성된 module이고, 검은색 block은 외부 메모리를 의미한다.

제안된 ASIFT 하드웨어의 동작은 그림 4.9에 제시된 연산 순서에 따라 수행된다. 그림 4.9에서 원 안에 표시된 숫자는 viewpoint index를 뜻한다. P_i로 표시된 viewpoint index 쌍은 scaled image를 공유할 수 있는 viewpoint를 묶은 것이다. 반대로 U_j라고 표시된 viewpoint index는 scaled image를 공유할 수 없다는 것을 뜻한다. 빗금 친 원으로 표시된 viewpoint는 scaled image를 생성해야 하는 viewpoint를 알려주고, 회색으로 표시된 viewpoint는 pre-scaled image를 생성해야 하는 것을 말한다.

ASIFT 하드웨어의 동작은 그림 4.8과 그림 4.9를 가지고 설명한다. Affine transform module에 포함된 source mux는 현재 처리해야 하는 viewpoint index에 적합한 입력 데이터를 source image, pre-scaled image, scaled image 중에서 선택한다. ASIFT 하드웨어가 첫번째로 $ASIFT(0)$를 수행해야 하므로 source image를 선택한다. B_0 는 identity matrix이므로 vertical scaler는 source image를 bypass하여 source buffer에 저장한다. Horizontal

scaler 역시 source buffer에 저장된 source image를 bypass하여 scaled image buffer에 저장한다. 그 다음, low-pass filter는 source image의 고주파 성분을 제거한다. Low-pass filtering된 image는 external 1x1 pre-scaled image buffer에 저장된다. 이 image는 pre-scaling parameter sx_{ps}를 1로 선택한 1x1 pre-scaled image라고 한다. 동시에, 1x1 pre-scaled image는 horizontal 1/2 scaler로 전달되어 가로 방향으로 1/2 down-sampling된다. 그리고 이 image는 external 0.5x1 pre-scaled image buffer에 저장된다. 이 image는 pre-scaling parameter sx_{ps}를 0.5로 선택한 0.5x1 pre-scaled image라고 한다. 2개의 pre-scaled image가 생성되는 연산과 동시에 scaled image buffer에 저장된 source image는 SIFT generation module에도 제공되어 SIFT feature 생성이 시작된다. B_0는 identity matrix이므로 SIFT generation module의 affine transform과 관련된 module들은 skew transform을 적용하지 않은 상태로 연산을 수행한다.

$ASIFT(0)$가 완료된 후, 나머지 viewpoint index는 모두 적절한 scaled image를 가져와서 동작해야 해야 한다. 따라서 source mux는 path1 또는 path2를 선택하게 된다. $ASIFT(1)$를 연산하기 위해서 source mux는 path1를 선택하고, pre-scaled image mux는 $sx_{ps} = floor(2 \times sx_i)/2$인 pre-scaled image를 선택한다. sx_1는 0.96이므로 sx_{ps}가 1인 pre-scaled image를 선택한다. Vertical scaler는 선택된 image를 vertical scaling parameter sy_i만큼 down-sampling한 다음 source buffer에 저장한다. Affine transform module에 구현된 모든 scaler들은 nearest neighbor (NN) interpolation 방식을 이용하여 image scaling을 수행한다. 따라서 vertical scaler는 선택된 image에서 적절한 pixel line을 선택하여 읽어오는 방식으로 수직 방향

scaling을 수행한다. sy_1는 0.52이므로 source buffer에는 1x0.52 크기로 scaling된 pre-scaled image가 저장된다. 마지막으로 horizontal scaler는 source buffer에 저장된 image를 horizontal scaling parameter sx_i/sx_{ps}를 이용하여 image scaling을 수행한다. $ASIFT(1)$의 경우, sx_i는 0.96이고, sx_{ps}는 1이므로 0.96이 horizontal scaler의 scaling parameter로 결정된다. 결과적으로 0.96x0.52 크기로 scaling된 image가 scaled image buffer에 저장된다. 이 scaled image는 SIFT generation module에 제공된다. 동시에 scaled image는 external scaled image buffer에 저장되어 $ASIFT(5)$에서 재사용될 준비를 한다. SIFT generation module은 $-g_1$ 만큼 skewing된 filtering kernel을 이용하여 SIFT generation을 수행한다.

$ASIFT(1)$가 완료된 다음 그림 4.9가 보여주는 것과 같이 $ASIFT(5)$가 진행된다. $ASIFT(5)$를 수행할 때, source mux는 이전에 생성된 0.96x0.52 크기의 scaled image를 재사용하기 위해 path2를 선택한다. 이 경우 추가적인 scaling 연산이 필요하지 않으므로 vertical scaler와 horizontal scaler는 bypass한다.

Affine transform module에 구현된 low-pass filter는 Gaussian filtering kernel을 이용하여 고 주파수 성분을 제거한다. Morel *et al.*은 low-pass filter의 Gaussian sigma(σ)를 t에 비례하여 증가시켰다. 본 논문에서는 구현의 용이성을 위해 σ를 0.5로 고정한다.

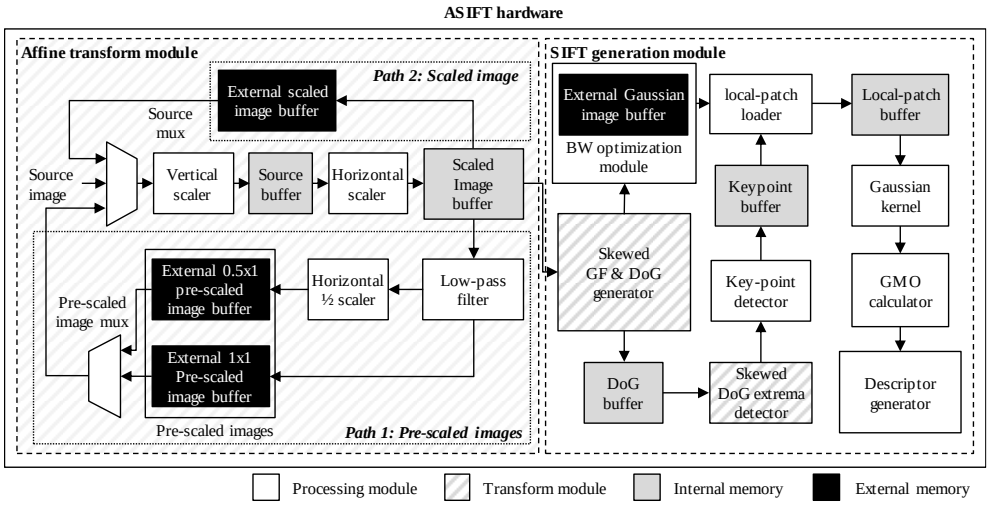


그림 4.8 ASIFT 하드웨어 구조

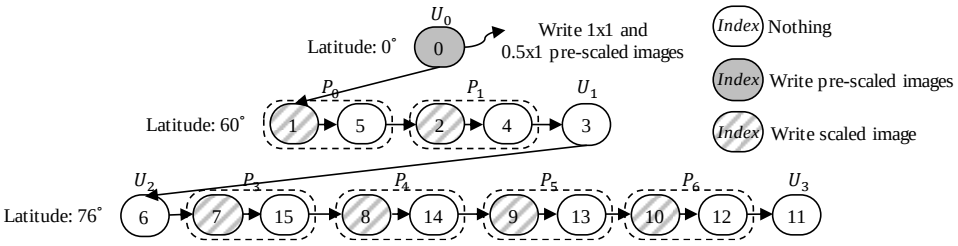


그림 4.9 ASIFT 하드웨어의 동작 순서

4.3 ASIFT 하드웨어에 대한 실험 결과

4.3.1 새 affine transform 방식에 의한 메모리 latency 감소

제안된 affine transform 방식을 이용할 경우, affine transform module은 모든 viewpoint에 대해 raster scan order로 source image를 읽어올 수 있다. 이로 인해 기존 affine transform 방식을 이용할 때에 비해 외부 메모리 latency를 크게 줄이면서 source image를 읽어오는 것이 가능해졌다. 이 실험은 ASIFT 하드웨어가 VGA 크기의 image를 처리하는 상황을 가정한다. ASIFT 하드웨어는 SDRAM을 외부 메모리로 사용하는 SoC 환경에 통합되었다. 이 SDRAM의 initial latency는 11 cycle이고, 다음 data 부터는 1cycle에 한 word씩 제공할 수 있다. 이 SoC의 bus system은 32 bits를 한 word로 구성하며, 한번의 burst transfer로 16개의 word를 제공한다.

기존 affine transform을 이용하면 연속적으로 읽어와야 하는 word의 주소가 불연속되는 상황이 많아 데이터를 읽어오는데 latency가 길며, 읽어온 word에 저장된 4개의 pixel 중 필요하지 않은 데이터가 포함될 수 있다. 반면 제안된 affine transform 방식을 이용하면, 외부 메모리에 저장된 pixel의 순서와 affine transform module이 읽어와야 하는 pixel의 순서가 같아서 burst transfer를 최대로 활용할 수 있다. 따라서 source image를 읽어오는데 소모되는 cycle 수가 크게 감소된다. 추가적으로 제안된 affine transform 방식을 이용하게 되면 pre-scaling 방안과 scaled image reuse 방안을 이용할 수 있게 되며, 이로 인해 읽어온 한 word 내의 대부분의 pixel이 valid한 경우가 많아진다.

표 4.2는 제안된 affine transform 방식의 적용 유무에 따라 외부 메모리에서 source image를 읽어오는데 얼마나 cycle이 소모되는지를 보여준다. 첫번째 열은 viewpoint index를 보여주고, 두번째 열은 기존 affine transform 방식을 사용한 경우의 외부 메모리 접근 cycle 수를 보여준다. 모든 viewpoint에 대한 전체 cycle 수는 12.09M cycle이다. 만일 제안된 affine transform 방식을 사용하면 외부 메모리 접근을 위한 전체 cycle 수는 5.36M cycle로 기존의 55.7%가 감소하였다. 네번째 열은 제안된 affine transform module이 pre-scaling 방안을 적용하였을 때의 cycle 수를 보여주며, 전체 cycle 수는 3.58M cycle로 감소하였다. 여기에 scaled image reuse 방안까지 이용하면 전체 cycle 수는 2.35M cycle로 감소한다. 이것은 제안된 방안은 전혀 이용하지 않았을 경우의 19.46%로 크게 감소된 양이다.

표 4.2 외부 메모리에서 source image를 읽어오는데 소모되는 cycle 수

v.i.	Original affine transform	Proposed affine transform (A)	A+Pre-scaling (B)	B+scaled image reuse
0	124,800	124,800	124,800	124,800
1	799,551	**96,224**	96,224	96,224
2	1,416,965	**580,272**	580,272	580,272
3	1,686,069	**844,800**	**62,400**	62,400
4	1,417,482	**580,272**	580,272	**61,152**
5	801,988	**96,224**	96,224	**60,016**
6	31,200	31,200	31,200	31,200
7	395,288	**57,040**	57,040	57,040
8	568,348	**182,304**	182,304	182,304
9	708,389	**327,360**	327,360	327,360
10	807,796	**514,976**	**227,088**	227,088
11	843,029	**844,800**	**422,400**	422,400
12	808,082	**514,976**	**230,128**	**30,400**
13	708,873	**327,360**	327,360	**29,952**
14	568,447	**182,304**	182,304	**29,952**
15	401,291	**57,040**	57,040	**30,008**
Total	**12,087,598**	**5,361,952**	**3,584,416**	**2,352,568**

4.3.2 Affine transform module의 출력 bandwidth 향상

Affine transform module의 출력 bandwidth를 높이기 위해, 본 논문에서는 pre-scaling 방안과 scaled image 재활용 방안을 사용하는 affine transform module 구조를 제안하였다. 표 4.3은 제안된 affine transform module 구조의 출력 bandwidth를 평가하기 위한 지표를 제시한다. 이 표는 affine transform module의 입력 데이터의 크기 대비 출력 데이터의 크기 비율을 보여준다. 만일 이 bandwidth ratio가 1이라면, 출력 bandwidth는 affine transform이 외부 메모리에서 읽어오는 입력 bandwidth와 같다는 것을 의미한다. 다시 말해, 이 비율이 1이라는 것은 SIFT generation module은 affine transform module이 존재하지 않고 외부 메모리에서 직접 scaling이 완료된 image를 읽어오는 것과 같은 속도로 데이터를 읽어갈 수 있다는 것을 의미한다. 표 4.3의 첫번째 열은 viewpoint index를 보여준다. 두번째 열은 affine transform module이 오직 source image만 선택하여 읽어갈 때의 bandwidth ratio를 보여준다. 이 경우, bandwidth ratio는 평균적으로 0.37이 된다. 제안된 affine transform module은 NN interpolation을 이용해 scaling 연산을 수행한다. 이 방법은 bandwidth ratio를 높이는데 효과적이다. 왜냐하면 외부 메모리에 저장된 image에서 적절한 pixel line을 읽어오면 vertical scaling이 완료되기 때문이다. NN interpolation을 사용하면 세번째 열이 보여주는 것과 같이 bandwidth ratio가 평균 0.72가 된다. 네번째 열은 NN interpolation과 더불어 pre-scaling method를 이용할 때의 bandwidth ratio를 보여준다. 이때의 평균 bandwidth ratio는 0.81이다. 마지막으로 scaled image 재사용 방안이 추가될 때는 bandwidth ratio는 0.89가 된다. 이것은 제안된 구조를 사용하지 않았을

때에 비해 2.4배 증가한 값이다.

표 4.3 입력 데이터 크기 대비 출력 데이터의 크기 비교

v.i.	None	NN based scaling (A)	A + Pre-scaling (B)	B + scaled image reuse
0	1.00	1.00	1.00	1.00
1	0.49	**0.95**	0.95	0.95
2	0.49	**0.70**	0.70	0.70
3	0.50	0.50	**1.00**	1.00
4	0.49	**0.70**	0.70	**1.00**
5	0.49	**0.95**	0.95	**1.00**
6	0.25	**1.00**	1.00	1.00
7	0.25	**0.95**	0.95	0.95
8	0.24	**0.80**	0.80	0.80
9	0.24	**0.60**	0.60	0.60
10	0.24	**0.38**	**0.75**	0.75
11	0.25	0.25	0.50	0.50
12	0.24	**0.38**	**0.75**	**1.00**
13	0.24	**0.60**	0.60	**1.00**
14	0.24	**0.80**	0.80	**1.00**
15	0.25	**0.95**	0.95	**1.00**
Average	**0.37**	**0.72**	**0.81**	**0.89**

4.3.3 ASIFT 하드웨어의 스펙과 동작 속도

표 4.4는 본 논문에서 제안하는 ASIFT 하드웨어의 design specification을 보여준다. 이 하드웨어는 130nm process technology를 통해 합성되었다. SIFT 하드웨어의 gate count는 574K이고, 동작 가능한 최대 주파수는 190MHz다. 사용하는 내부 메모리 총량은 481.36Kbits다.

표 4.4 제안된 ASIFT 하드웨어 specification

Technology	130nm
Maximum operating frequency	190MHz
Gate count(except memory)	574K
Internal memory size	481.36Kbits
External memory size	9.67Mbits

제안하는 ASIFT 하드웨어의 동작 속도를 측정하기 위해, VGA (640x480) 크기의 source image를 처리하는데 소요되는 시간을 RTL 시뮬레이션을 이용해 측정하였다. 이 시뮬레이션에서 동작 주파수는 제안하는 하드웨어의 최대 동작 속도인 190MHz로 설정하였고, source image은 Mikolajczyk *et al.* 이 제안한 Oxford test image들을 사용하였다. 동작 속도 측정 시뮬레이션이 수행된 환경은 4.3.1 절에서 설명한 SoC 환경이 이용되었고, 사용되는 외부 메모리의 bandwidth는 2.46bytes/cycle이다. 시뮬레이션 결과는 표 4.5와 같다. 첫번째 열은 test image의 이름을 보여주고, 두번째 열은 key-point의 수를 보여준다.

마지막 열은 1frame을 처리하는데 소요되는 연산 시간을 보여준다. 표 4.5가 보여주는 것과 같이 제안된 ASIFT 하드웨어는 2,500개의 key-point가 검출되는 영상에 대하여 30fps의 동작 속도로 ASIFT 알고리즘을 수행할 수 있다.

표 4.5 제안된 ASIFT 하드웨어의 VGA image에 대한 동작 속도

Test images	Key-points	Operating time (ms)
Leuven 1	2,228	31.57
Wall 1	2,435	32.03
Ubc 1	2,574	33.86
Graf 1	2,644	34.39
Bike 1	2,678	34.65
Boat 1	3,188	38.13
Tree 1	3,220	39.28
Bark 1	4,386	47.99

4.3.4 Feature matching 정확도

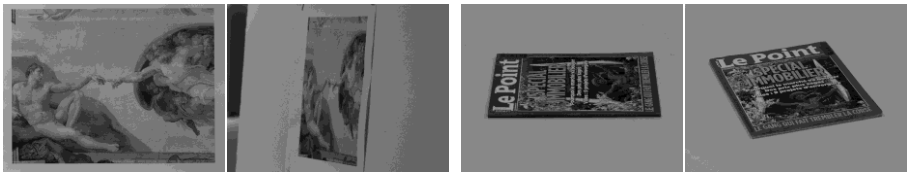
본 논문에서 제안한 ASIFT 하드웨어 구현을 위한 여러 방안은 feature matching 정확도에 영향을 미칠 수 있다. 이것을 평가 하기 위해, original ASIFT 알고리즘을 구현한 소프트웨어와 제안된 방안이 적용된 소프트웨어 및 하드웨어의 feature matching 정확도를 비교한다. 실험에서 사용된 original ASIFT 소프트웨어는 Morel *et al.*이 [15]에서 공개한 affine transform function과 Hess가 [33]에서 공개한 SIFT generation function을 이용하여 구현되었다. Feature matching 정확도는 Mikolajczyk *et al.*가 [32]에서 제안한 'matching score' metric을 이용하여 평가하였고, 실험에서 사용된 test image는 Morel *et al.*이 공개한 test image들을 이용하였다. Matching score를 측정하는 소프트웨어는 Mikolajczyk *et al.*이 공개한 소프트웨어를 수정하여 사용하였다. 구체적으로 feature matching function이 ASIFT에 적합하도록 수정되었다. ASIFT에 적합한 feature matching function은 하나의 시뮬레이션 된 image에서 추출된 feature를 하나의 set으로 구성하고, feature set 단위로 matching이 수행된다. 예를 들어 image 1과 image 2가 matching된다면, image1에 속한 하나의 feature는 image 2의 서로 다른 feature set에 포함된 서로 다른 다수의 feature와 matching될 수 있다. 이 경우 feature matching function은 1개의 correct matches를 발견하였다고 평가한다.

Morel *et al.* 이 제시한 test image는 그림 4.10과 같이 Painting_zoom1, Painting_zoom1R, Magazine_zoom4, Painting_zoom10, Painting_zoom10R, Magazine_tilt2, Magazine_tilt4로 구성된다. Painting_zoom1, Painting_zoom1R, Magazine_zoom4, Painting_zoom10, Painting_zoom10R

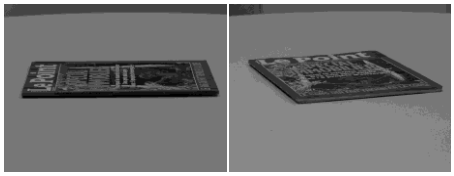
는 평면 물체를 정면에서 촬영한 장면부터 latitude 각도를 0° 에서부터 80° 로 증가시키면서 촬영된 장면들로 구성된다. Magazine_tilt2와 Magazine_tilt4는 magazine 책을 $t = 2$와 $t = 4$인 상태에서 longitude를 0° 에서부터 90° 까지 증가시키면서 촬영한 장면들로 구성되었다. 이러한 test image들을 이용하여 동일한 평면 물체를 서로 다른 viewpoint에서 촬영한 상태에서 correct matching이 가능한지 평가할 수 있게 해준다. 특히 Magazine_tilt2와 Magazine_tilt4의 경우 촬영된 두 image 모두 기울어진 상태에서 matching이 가능하지 확인할 수 있는 상황을 제공한다. 이것은 한 image가 정면에서 촬영된 것보다 훨씬 matching하기 어려운 상황이다.

(a) Painting_zoom1 (b) Painting_zoom1R

(c) Magazine_zoom4 (d) Painting_zoom10



(e) Painting_zoom10R (f) Magazine_tilt2



(g) Magazine_tilt4

그림 4.10 Morel *et al.*이 제시한 test image들 [15]

4.3.4.1 제안된 affine transform 방식과 parameter에 따른 matching 정확도 비교

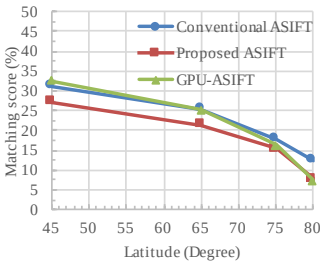
표 4.6은 ASIFT 하드웨어 구현의 효율성을 높이기 위해 제안된 다양한 방안들에 따른 ASIFT 알고리즘의 matching score 변화를 보여준다. 첫번째 열은 실험에서 사용된 test image의 이름을 보여준다. 두번째 열은 original affine transform 방식을 사용할 때의 matching score를 보여준다. 세번째 열은 본 논문에서 제안한 새로운 affine transform 방식을 이용하였을 때의 matching score를 보여준다. 표 4.6이 보여주는 것과 같이 제안된 affine transform 적용 유무에 따라 달라지는 matching score 차이는 0.2로 미미하다. 여기에 추가적으로 4.2.2.1에서 설명한 것과 같이 longitude offset을 변경할 경우 평균 matching score는 약간 상승한다. 이것은 viewpoint의 offset을 변경하면서 viewpoint 들의 분포가 좀 더 균일해지면서 생긴 효과다. 마지막으로, affine transform module의 image scaler들이 NN interpolation 기반 image scaling을 수행할 경우, 이것이 matching score에 미치는 영향을 5번째 열이 보여준다. 평균적으로 matching score는 0.76 감소한다. 마지막으로 6번째 열은 3장에서 제안한 local-patch sampling 방안을 추가적으로 적용할 경우의 matching score를 보여준다.

표 4.6 제안된 방안들로 인한 ASIFT 알고리즘의 matching score 변화 비교

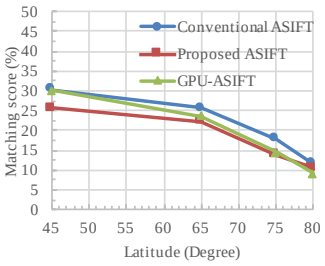
Test image set	Original affine transform	Modified affine transform (B)	B + Modified longitude offset (C)	C + NN interpolation (D)
painting_zoom1	21.70	22.00	23.82	23.02
painting_zoom1R	21.41	21.63	24.21	23.44
magazine_zoom4	35.16	33.96	36.50	35.34
painting_zoom10	22.09	21.61	26.66	25.59
painting_zoom10R	19.83	19.72	25.07	23.72
magazine_tilt2	19.83	19.72	25.07	14.28
magazine_tilt4	5.02	4.94	7.18	6.83
Average	**18.19**	**17.99**	**21.13**	**20.38**

4.3.4.2 Tilt 관련 parameter의 변경에 따른 matching 정확도 비교

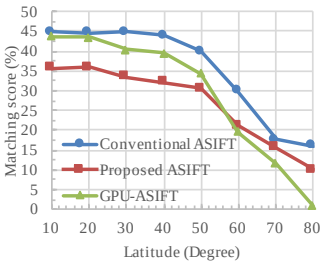
본 논문에서는 ASIFT 하드웨어의 연산량 합리화를 위해 tilt sampling step(Δt)을 기존의 $\sqrt{2}$ 에서 2로 변경하였다. 이러한 Δt 의 변경은 ASIFT 알고리즘의 연산량을 기존의 43%로 감소시킨다. 이전 GPGPU를 이용한 ASIFT 알고리즘 구현 연구 (GPU-ASIFT)에서는 Δt를 변경하는 대신 t_{max}를 기존의 $4\sqrt{2}$에서 2로 변경하였다. 이로 인해 ASIFT 알고리즘의 연산량은 기존의 45%로 줄어들었다. 이것은 본 논문에서 Δt를 2로 변경한 것과 거의 동일한 연산량 감소 효과를 가져온다. 이와 같이 t 와 관련된 두가지 접근법에 따른 feature matching의 정확도는 그림 4.11에서 보여준다. Conventional ASIFT는 ASIFT 알고리즘 그대로 구현된 경우를 말하고, proposed ASIFT는 $\Delta t = 2$인 경우를, GPU-ASIFT는 $t_{max} = 2$ 를 의미한다. 실험 결과, Δt 를 2로 증가시킬 경우 matching score는 original ASIFT 알고리즘 대비 83.57%로 감소한다. GPU-ASIFT의 경우 latitude 또는 longitude의 각도가 작은 경우 proposed ASIFT보다 더 높고 conventional ASIFT와 거의 유사한 matching score를 보여준다. 반면 latitude 또는 longitude의 각도가 증가할수록 matching score는 급격하게 감소하는 상황이 발생하며, proposed ASIFT 보다 더 많이 감소한다. 특히 proposed ASIFT의 경우 모든 경우에 대하여 correct matches를 확보하였지만, GPU-ASIFT의 경우 magazine_tilt4의 경우 longitude가 30° 이상인 경우 제대로 correct matches를 획득하지 못하는 것을 보여준다. 이를 통해 $\Delta t = 2$로 설정하는 경우에는 비록 matching score가 83.57% 감소하지만, fully affine invariant한 특성은 여전히 유지되고 있음을 볼 수 있다.



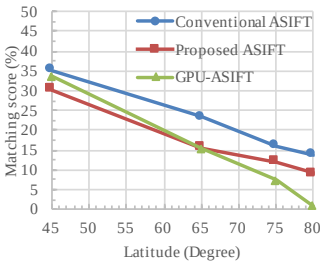
(a) Painting_zoom1



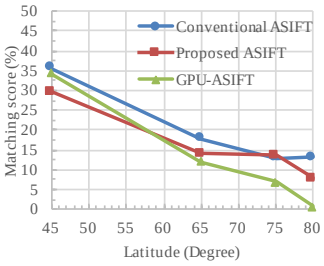
(b) Painting_zoom1R



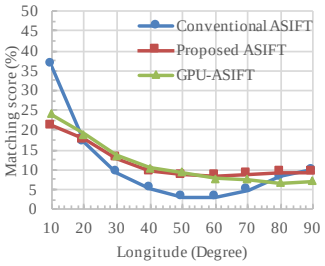
(c) Magazine_zoom4



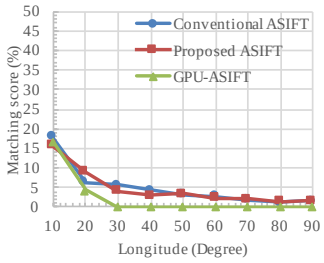
(d)Painting_zoom10



(e) Painting_zoom10R



(f)Magazine_tilt2

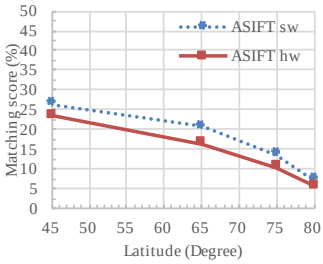


(g) Magazine_tilt4

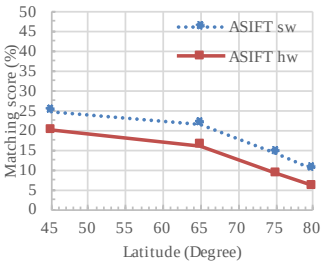
그림 4.11 original ASIFT, $\Delta t = 2$인 ASIFT, $t_{max} = 2$인 ASIFT (GPU-ASIFT)의 matching score 비교

4.3.4.3 ASIFT 하드웨어와 ASIFT 소프트웨어의 matching 정확도 비교

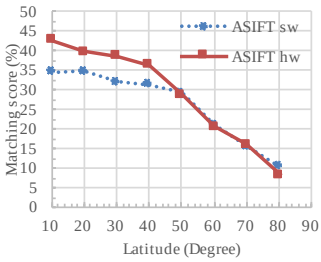
소프트웨어로 구현된 ASIFT 알고리즘을 모든 사칙연산은 floating-point 기반 연산으로 수행하는 반면 ASIFT 하드웨어는 fixed-point 기반 사칙 연산을 기반으로 계산을 수행한다. 이러한 변화는 ASIFT 하드웨어의 정확도에 좋지 않은 영향을 미친다. 그림 4.12는 ASIFT 하드웨어와 ASIFT 하드웨어와 동일한 parameter로 setting된 ASIFT 소프트웨어의 matching score를 비교한 데이터를 보여준다. 여기서 사용된 ASIFT 하드웨어는 SIFT generation module의 외부 메모리 bandwidth 최적화 방안(local-patch sampling 및 fractional bits 제거)이 추가적으로 적용되었다. 두 ASIFT 알고리즘 구현의 평균 matching score를 비교하면, ASIFT 하드웨어가 소프트웨어에 비해 97.98% matching score가 낮은 것으로 측정되었다. 그러나 ASIFT 하드웨어는 모든 latitude와 longitude 변화에 대하여 correct matches를 확보할 수 있는 것으로 확인되었다. 다시 말해 ASIFT 하드웨어는 fully affine invariant한 특성을 유지하고 있다는 것이 확인되었다.



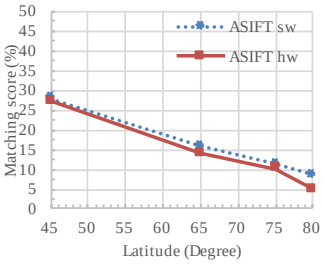
(a) Painting_zoom1



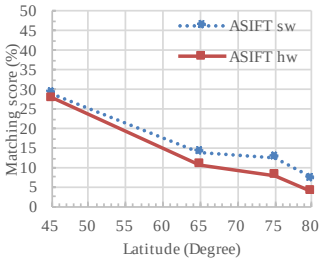
(b) Painting_zoom1R



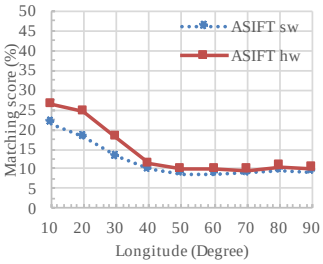
(c) Magazine_zoom4



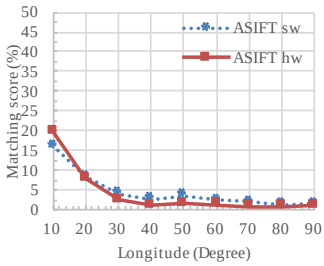
(d)Painting_zoom10



(e) Painting_zoom10R



(f)Magazine_tilt2



(g) Magazine_tilt4

그림 4.12 ASIFT 소프트웨어와 ASIFT 하드웨어의 matching score 비교

제 5 장 결론

본 논문에서는 ASIFT 알고리즘을 실시간으로 수행할 수 있는 ASIFT 하드웨어의 구조를 처음으로 제안하였다. ASIFT 알고리즘은 두 영상 사이에서 scale change, rotation, viewpoint change의 image transformation이 발생하여도 안정적으로 image matching을 수행할 수 있는 높은 안정성을 보이는 local feature다. 그러나 이러한 안정성을 확보하기 위하여 다양한 viewpoint에 대한 시뮬레이션, Gaussian scale space 계산 등 굉장히 많은 연산을 수행해야 하므로 software로 구현할 경우 실시간 처리에 어려움을 겪게 된다. 따라서 전용 하드웨어 설계를 통해 알고리즘 연산 가속에 대한 연구가 필요하였다.

본 논문은 상용화 가능한 ASIFT 하드웨어 구조를 설계하기 위하여 첫번째로 기존에 제안된 SIFT 하드웨어가 실시간성을 확보하기 위해 과도한 내부 메모리를 사용하는 문제를 해결하였다. 이를 위해 SIFT 연산에 사용되는 메모리 공간을 내부/외부 메모리를 혼용하여 구성하고, 외부 메모리 bandwidth가 증가하면서 외부 메모리 latency로 인한 동작 속도 저하 문제를 해결하기 위한 bandwidth 최적화 방안을 제안하였다. 또한 효율적인 SIFT 하드웨어 설계로 동작 주파수 향상과 key-point detection step과 descriptor generation step의 병렬 동작이 가능해져 기존의 state-of-the art 하드웨어보다 더 빠른 HD (1280x720) 크기의 영상을 30fps로 처리할 수 있는 하드웨어를 설계하였다.

두번째로, 제안된 SIFT 하드웨어를 ASIFT 연산에 높은 utilization으로 이용하기 위해 throughput을 높인 affine transform module 구조를 제안하였다.

일반적인 affine transform 연산은 rotation 연산을 포함하며, 이는 외부 메모리에 저장된 source image를 불연속적인 메모리 주소로 접근하게 만든다. 이러한 외부 메모리 접근은 latency를 빈번히 발생시키고, 이로 인해 affine transform module의 동작 속도가 저하된다. 본 논문은 SIFT 알고리즘이 rotation-invariant하다는 특성을 이용해 affine transform 과정에서 rotation 단계를 제거하였고, 이로 인해 외부 메모리 latency로 인한 속도 저하를 해결하였다. 또한 affine transform 과정 중 image scaling을 수행 시, 이전에 수행된 scaling 연산 결과를 재사용할 수 있는 구조를 제안하여, affine transform module의 동작 속도를 향상시켰다. 결과적으로 제안된 affine transform module은 출력 데이터를 SIFT 하드웨어에 고속으로 제공할 수 있게 되었고, 최종적으로 제안하는 ASIFT 하드웨어는 VGA (640x480) 크기의 영상을 30fps의 속도로 처리할 수 있다.

SIFT 알고리즘이나 ASIFT 알고리즘은 높은 신뢰도의 correspondence 정보를 제공해줄 수 있지만, 실시간으로 동작해야 하는 시스템에서는 많은 연산 요구량 때문에 적극적으로 활용되지 못하였다. 본 논문에서 구현된 ASIFT 하드웨어는 VGA 영상을 30fps를 처리할 수 있는 속도를 갖추고 있기 때문에 이러한 상황에서 적극적으로 활용될 경우 순수한 software로만 구성된 computer vision system보다 더 빠른 속도와 정확성을 제공해줄 수 있을 것으로 기대된다.

참고 문헌

[1] T. Tuytelaars, and K. Mikolajczyk, "Local Invariant Feature Detectors: A Survey," Foundations and Trends® in Computer Graphics and Vision 3.3 (2008): 177-280.

[2] K. Grauman, and B. Leibe, "Visual Object Recognition," Morgan & Claypool Publishers (1st edition), April, 2011, ISBN-13: 978-1598299687

[3] D. Lowe, "Distinctive image features from scale-invariant keypoints," International Journal of Computer Vision, vol. 2, no. 60, pp. 91-110, 2004.

[4] H. Bay, A. Ess, and T. Tuytelaars, "Speeded-up robust features (SURF)," Computer Vision and Image Understanding, no. 110, pp. 346-359, 2008.

[5] E. Rosten and T. Drummond. Machine learning for high- speed corner detection. In European Conference on Com- puter Vision, volume 1, 2006.

[6] J. Matas, O. Chum, M. Urban, and T. Pajdla, "Robust wide-baseline stereo from maximally stable extremal regions," in Proceedings of the British Machine Vision Conference, pp. 384–393, 2002.

[7] T. Tuytelaars and L. Van Gool, "Wide baseline stereo matching based on local, affinely invariant regions," in Proceedings of the British Machine Vision Conference, pp. 412–425, 2000.

[8] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. "ORB: an efficient alternative to SIFT or SURF," *In Proc. IEEE Int. Conf. Comp. Vis.*, 2011, pp. 2564-2571.

[9] Lowe, David G. "Object recognition from local scale-invariant features." *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*. Vol. 2. Ieee, 1999.

[10] Zhou, Huiyu, Yuan Yuan, and Chunmei Shi. "Object tracking using SIFT features and mean shift." *Computer vision and image understanding* 113.3 (2009): 345-352.

[11] Brown, Matthew, and David G. Lowe. "Recognising panoramas." *ICCV*. Vol. 3. 2003.

[12] Ke, Yan, and Rahul Sukthankar. "PCA-SIFT: A more distinctive representation for local image descriptors." *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*. Vol. 2. IEEE, 2004.

[13] Derpanis, Konstantinos G., Erich TH Leung, and Mikhail Sizintsev. "Fast scale-space feature representations by generalized integral images." *Image Processing*,

2007. ICIP 2007. IEEE International Conference on. Vol. 4. IEEE, 2007.

[14] Zhang, S., et al. "Fast SIFT Algorithm for Object Recognition." *Computer Systems & Applications* 6 (2010): 022.

[15] J.-M. Morel and G. Yu, "ASIFT: A new framework for fully affine invariant image comparison," SIAM Journal on Imaging Sciences (SIIMS), vol. 2, no. 2, pp.438-469, 2009.

[16] M. Lalonde, D. Byrns, L. Gagnon, N. Teasdale, and D. Laurendeau, "Real-time eye blink detection with GPU-based SIFT tracking," *in Proc. 4th Can. Conf. Comput. Robot Vis.*, 2007, pp. 481–487.

[17] S. Heymann, K. Müller, A. Smolic, B. Froehlich, and T. Wiegand, "SIFT implementation and optimization for general-purpose GPU," *in Proc. Int. Conf. Central Eur. Comput. Graph., Visual. Comput. Vis.*, 2007, pp. 144–159.

[18] P. H. Hsu, Y. C. Tseng, and T. S. Chang, "Low memory cost bilateral filtering using stripe-based sliding integral histogram," *in Proc. IEEE Int. Symp. Circuits Syst.*, 2010, pp. 3120–3123.

[19] B. Rister, G. Wang, M. Wu, and J. R. Cavallaro, "A fast and efficient SIFT detector using the mobile GPU," *in Proc. IEEE Int. Conf. Acoustics, Speech, Sig. Proc.*, 2013, pp. 2674-2678.

[20] G. Wang, B. Rister, and J. R. Cavallaro, "Workload Analysis and Efficient OpenCL-based Implementation of SIFT Algorithm on a Smartphone," *in Proc. IEEE Glob. Conf. Sig. Info. Proc.*, 2013, pp. 759-762.

[21] M. Mark, K. Keutzer, and P. Wang, "Image feature extraction for mobile processors," *in Proc. IEEE Int. Symp. Workload Char.*, 2009, pp. 138-147.

[22] V. Bonato, E. Marques, and G. A. Constantinides, "A parallel hardware architecture for scale and rotation invariant feature detection," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 12, pp. 1703–1712, Dec. 2008.

[23] L. Yao, H. Feng, Y. Zhu, Z. Jiang, D. Zhao, and W. Feng, "An architecture of optimized SIFT feature detection for an FPGA implementation of an image matcher," *in Proc. Int. Conf. Field-Program. Technol.*, 2009, pp. 30–37.

[24] K. Mizuno, H. Noguchi, G. He, Y. Terachi, T. Kamino, H. Kawaguchi, and M. Yoshimoto, "Fast and low-memory-bandwidth architecture of SIFT descriptor generation with scalability on speed and accuracy for VGA video," *in Proc. Int. Conf. Field Programmable Logic Appl.*, 2010, pp. 608-611.

[25] L.-C. Chiu, T.-S. Chang, J.-Y Chen, and Y.-C. Chang, "Fast sift design for real-time visual feature extraction," *IEEE Trans. Image Proc.*, vol. 22, no. 8, pp.3158-3167, Aug. 2013.

[26] F. C. 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, Mar. 2012.

[27] M. Qasaimeh, A. Sagahyroon, and T. Shanableh. "A parallel hardware architecture for Scale Invariant Feature Transform (SIFT)," *in Proc. IEEE Int. Conf. Mult. Comp. Syst.*, 2014, pp. 295-300.

[28] 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.

[29] E.S. Kim and H.-J. Lee, "A Practical Hardware Design for the Keypoint Detection in the SIFT Algorithm with a Reduced Memory Requirement," *in Proc. IEEE Int. Symp. Circuits Syst.*, 2012, pp.770-773.

[30] V. Codreanu, F. Dong, and B. Liu, "GPU-ASIFT: A fast fully affine-invariant feature extraction algorithm," in proc. of IEEE High Performance Computing and Simulation (HPCS), 2013, pp. 474-481.

[31] T. Lindeberg, "Scale-space for discrete signals," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 12, no. 3, pp. 234-254, Apr 01, 1990.

[32] K.Mikolajczyk,T.Tuytelaars,andC.Schmid,"AComparisonofAffine Region Detectors," *Int. J. Comput. Vis.*, vol. 65, no. 1-2, pp. 43-72, Nov. 2005.

[33] R.Hess,"AnOpen-SourceSIFTLibrary,"*inProc.Int.Conf.Multi.*,2010, pp. 1493–1496.

Abstract

A Novel Hardware Architecture for Real Time Extraction of Local Features

Joohyuk Yum
Electrical and Computer Engineering
The Graduate School
Seoul National University

In the computer vision applications, scene matching with local features is an essential function for object recognition, motion estimation, and 3D reconstruction. Scale-Invariant Feature Transform (SIFT), proposed by Lowe, is one of the most widely used local features because it is invariant to scale, and rotation. However, SIFT does not guarantee the matching performance when an axis orientation of the camera is changed. In order to maintain the performance of SIFT against the change of viewpoint, an affine invariant extension of SIFT (ASIFT) is proposed by Morel *et al*. ASIFT simulates many images which are obtained by using affine transforms for various viewpoints of the camera. Then, SIFT features are extracted from the simulated images. Because ASIFT explores many simulated images considering various viewpoints, ASIFT achieves affine-invariance, and more correspondences can be found comparing with the conventional SIFT.

In order to achieve real-time ASIFT extraction, this paper proposes

fast SIFT hardware architecture. In the research area, a number of SIFT acceleration works have proposed, and most of all satisfy real-time operation (30frames/sec). However previous works use a large amount of internal memory, which limits a wider use of SIFT. In order to reduce internal memory size, a new SIFT hardware architecture was proposed, which reduces the amount of required internal memory by storing temporary data into external memory. The use of external memory demands an excessive bandwidth for the external memory. In order to reduce the external memory bandwidth, this paper proposes three schemes: local-patch reuse, local-patch sub-sampling, and fraction-bit truncation. As a result, the proposed hardware processes HD-sized (1280x720) video at 30fps with 3,300 key-points. This is achieved because the external memory access is reduced to only 4.75% of that without the proposed schemes and also because the size of the internal memory is reduced to 10.93% compared to the size required in the previous work.

In order to increase the operating speed of the ASIFT hardware, the affine transform module should provide a pixel of affine transformed images to the SIFT generation module without delay. A proposed affine transform method modifies the order of data read into the raster scan order, which reduces the latency of an external memory access. In the proposed ASIFT hardware architecture, scaled images are reused for multiple viewpoints, which improves the efficiency of external memory

access. As a result, the amount of cycles for accessing the external memory is reduced by 80.54% and bandwidth between the affine transform module and the SIFT hardware is increased by 240%. In order to achieve the real-time operation, the amount of computation is reduced to 43% by changing a parameter for view simulation, but the proposed method maintains the characteristics of fully affine invariance. The proposed hardware processes a VGA-sized (640x480) video at 30 fps with 2,500 key-points.

Keywords : SIFT, ASIFT, Hardware accelerator, Bandwidth optimization, Data reuse, Affine transform hardware

Student Number : 2012-30218