공학석사학위논문

고해상도 영상 처리를 위한 HEVC 디코더의 효율적인 병렬 처리 기법

2014 년 2 월

서울대학교 대학원

공과대학 전기 · 컴퓨터공학부

최 준 철

고해상도 영상 처리를 위한 HEVC 디코더의 효율적인 병렬 처리 기법

지도교수 하 순 회

이 논문을 공학석사학위논문으로 제출함

2014 년 2 월

서울대학교 대학원

공과대학 전기·컴퓨터공학부

최 준 철

최준철의 석사학위논문을 인준함

2014 년 2 월

위 원 장 : 장 래 혁 (인)

부위원장 : 하 순 회 (인)

위 원 : 이 창 건 (인)

요약

4K 수준의 고해상도 영상, 3D 지원, Multi-view 등 차세대 TV에 대한 사용자의 요구사항이 지속적으로 증가하고 있으며 디지털 디스플레이 기술의 지속적인 발전으로 고화질 차세대 TV가 개발되고 있다. 또한 이러한 고해상도 영상 처리를 지원하기 위해 새로운 코덱(codec) 역시 개발되고 있으며, High Efficiency Video Coding(HEVC)가 그러한 코덱 중 하나이다. HEVC는 기존 h.264 코덱 대비 더 높은 영상 압축률 및 영상 품질을 제공하여 차세대 TV에 사용될 코덱으로 적합하다고 여겨지고 있으나, HEVC 기준 소프트웨어가 순차 수행되도록 구현되어 있기 때문에 최근의 SoC 설계 경향인 멀티코어 시스템을 충분히 활용하지 못하는 문제점을 가지고 있다. 본 논문은 멀티코어 기반 차세대 TV 플랫폼에서의 병렬화를 통한 성능 개선 및 SoC 설계 공간 탐색 등을 지원할 수 있는 소프트웨어 모델을 만드는 것을 목표로 하고 있으며, 이를 위해 유연한 시스템을 지원할 수 있도록 멀티코어를 최대한 활용하여 시간당 처리량을 최대화 하는 태스크 그래프 모델 기반의 병렬처리 기법을 제안한다. 또한 영상 처리 알고리즘이 여러 데이터에 동일한 연산을 반복하는 연산 패턴을 보인다는 점에 착안하여 Single Instruction Multi Data (SIMD) 명령어를 통한 병렬 처리 기법도 제안한다. 제안하는 기법의 우수성을 실험을 통해 확인하였으며, 실험 결과 2560x1600 해상도의 예제에서 순차 수행 대비 최대 15.59배, 3840x2160예제에서 15.85배의 성능 증가를 얻었다.

주요어 : 4K 해상도, HEVC, 멀티코어, 병렬처리, 설계 공간 탐색

학 번 : 2011-23385

목차

그림 목차

표 목차

제 1 장 서론

4K 수준의 고해상도 영상, 3D 지원, Multi-view 등 차세대 TV 에 대한 사용자의 요구사항이 지속적으로 증가하고 있으며 디지털 디스플레이 기술의 지속적인 발전으로 고화질 차세대 TV 가 개발되고 있다. 또한 이러한 고해상도 영상 처리를 지원하기 위해 새로운 코덱(codec) 역시 개발되고 있으며, High Efficiency Video Coding(HEVC)가 그러한 코덱 중 하나이다. HEVC 는 Moving Picture Experts Group (MPEG)와 ITU-T 의 Video Coding Experts Group (VCEG)가 Joint Collaborative Team On Video Coding (JCT-VC) 라는 이름으로 팀을 결성하여 개발에 착수한 차세대 동영상 인코딩 기술을 말한다. HEVC 는 기존 h.264 코덱 대비 더 높은 영상 압축률 및 영상 품질을 제공하여 차세대 TV 에 사용될 코덱으로 적합하다고 여겨지고 있으나, HEVC 기준 소프트웨어가 순차 수행되도록 구현되어 있기 때문에 최근의 SoC 설계 경향인 멀티코어 시스템을 충분히 활용하지 못하는 문제점을 가지고 있다. 차세대 멀티미디어 플랫폼에서 목표로 하고 있는 4K 해상도, 즉 Quad Full High Definition (QFHD)의 3840x2160 해상도의 영상들을 처리하기 위해서는 멀티코어 시스템은 필수적이기 때문에 HEVC 의 병렬 처리 기법의 필요성이 특히나 강조된다. 본 논문은 멀티코어 기반 차세대 TV 플랫폼에서의 병렬화를 통한 성능 개선 및 SoC 설계 공간 탐색 등을 지원할 수 있는 소프트웨어 모델을 만드는 것을 목표로 하고 있으며, 이를 위해 유연한 시스템을 지원할 수 있도록 멀티코어를 최대한 활용하여

시간당 처리량을 최대화 하는 태스크 그래프 모델 기반의 병렬처리 기법을 제안한다. 또한 영상 처리 알고리즘이 여러 데이터에 동일한 연산을 반복하는 연산 패턴을 보인다는 점에 착안하여 Single Instruction Multi Data (SIMD) 명령어를 통한 병렬 처리 기법도 제안한다.

본 논문에서는 HEVC 가 가지는 특징들에 대해 간략히 설명하고, HEVC 디코더의 효율적인 병렬 처리를 위해 HEVC 디코더를 태스크 단위로 분리 분석을 진행할 것이다. 태스크들 간의 의존관계 및 수행 시간 비율 분석을 통해 유연한 병렬 처리가 가능한 태스크 그래프 모델을 제시할 것이며 이를 통한 성능 향상을 실험을 통해 측정하여 성능의 우수성을 보일 것이다.

본 논문의 구성은 다음과 같다. 2 장에서 본 논문과 관련성을 가지는 HEVC 병렬화 기법을 제시한 유사 연구들에 대해 살펴 볼 것이며, 3 장에서는 HEVC 알고리즘에 대해 간략히 소개할 것이다. 4 장에서는 HEVC 디코더 알고리즘을 태스크 단위로 나누어, 각 태스크들이 하는 작업들과 태스크 내부의 의존관계성, 태스크 간의 의존관계성에 대해 분석하고, 수행시간 비율을 통해 병렬 처리 방향성에 대해 짚어볼 것이다. 5 장에서는 HEVC 디코더의 분석을 기반으로 태스크 그래프 모델 기반의 병렬 처리 기법과, SIMD 명령어를 사용한 병렬 처리 기법을 제안하며, 구체적인 구현 방법을 설명한다. 6 장에서 제안하는 기법이 어느 정도의 성능 향상을 가져오는지를 실험을 통해 측정할 것이다. 마지막으로 7 장에서 이 논문의 결론을 내린다.

제 2 장 관련연구

HEVC 디코더의 병렬 처리 기법으로 CPU 를 사용한 멀티 쓰레딩 방법, GPU 를 사용한 가속화, 혹은 SIMD 를 사용한 방법 등 다양한 방법이 제시되어 왔다. 이 장에서는 대표적인 기존 연구들을 살펴보며 각 기법이 어떤 특징을 가지며 우리의 기법과 다른 점이 무엇인지를 살펴본다.

HEVC 디코더를 멀티 쓰레딩으로 병렬 처리한 기법을 제시한 논문으로 [3]의 병렬화 기법이 있다. 이 기법은 LCU 단위의 wave-front 방식의 병렬화 기법이라는 점에서 제안하는 병렬 처리 기법과 유사하지만, 시간당 처리량을 최대화하는 것에 초점을 맞추지 않고 한 프레임의 처리 시간을 최소화하려 했다는 점에서 목적이 다르다. 멀티미디어 영상 처리는 반복적으로 동일 알고리즘을 수행한다는 점에서 볼 때 처리 시간의 최소화보다 시간당 처리량을 최대화하려 하는 우리의 방법이 더 적합하다. 또 다른 차이점은 이 논문에서 제시한 기법은 픽셀 복호화 부분만이 wave-front 형태로 병렬화되며, 다른 스테이지의 작업으로 넘어갈 때는 명시적인 배리어를 통한 동기화를 하기 때문에 배리어 동기화로 인한 CPU 활용도가 떨어지게 된다는 점이다. 반면 우리의 기법은 태스크들 간의 의존관계만 성립되면 작업이 수행될 수 있게 하여 명시적인 배리어 동기화가 존재하지 않는다. 특기할만한 장점은 이 기법에선 스트림 파싱이 LCU 단위로 처리되자마자 픽셀 복호화가 진행된다는 점이다. 이를 통해 전체 프레임이 파싱되기 전에 복호화가 진행될 수 있어 프레임 처리 시간이 감소한다. 우리의 기법은 전체 프레임이 파싱되고 난 후에 복호화가 진행되지만, 전체적으로 볼 때 파이프라이닝이 되어 시간당 처리량에 큰 영향을 주지 않는다.

또 다른 병렬화 기법을 제시한 논문으로 [4] 가 있다. 이 논문에서 제시한 방법은 HEVC 디코더 알고리즘의 모든 작업이 의존관계를 고려하여 wave-front 방식의 병렬 처리가 이루어진다는 점에서 우리의 방법과 아주 유사하다. 그러나 이 논문에서 제시하는

방식은 각 행이 엔트로피 슬라이스라는 특수한 슬라이스로 구분되도록 인코딩한 경우에만 사용 가능하다는 큰 제약이 있다. 이와 같은 방법으로 각 행이 다른 슬라이스에 배치되면 각 행간의 의존관계가 사라져 병렬화가 쉬워진다는 장점이 있으나, 그만큼 영상의 압축률이 크게 떨어지는 단점을 가진다. 또한 엔트로피 슬라이스는 표준 규격으로 결정된 main profile 에서 지원하지 않기 때문에 범용성이 크게 떨어진다는 문제점도 존재한다. HEVC 에서 지원하는 타일이나 슬라이스 기반의 병렬화를 이용한 논문 [5]도 있으나, 앞의 연구와 마찬가지로 압축률 등의 성능 손실이 발생하며 해당하는 방식으로 인코딩된 영상에만 적용 가능하다는 약점이 있다.

CPU 를 사용한 기법으로 가장 유사한 방법을 제시한 논문으로 [6]이 있다. [6]에선 wave-front 기법을 확장시킨 Overlapped WaveFront (OWF)를 제안하고 있다. 기존의 wave-front 방법은 한 프레임 내에서의 병렬화만을 고려했다면, OWF 는 다음 프레임의 참조영역이 처리가 완료되면 이전 프레임의 처리가 끝날 때까지 기다리지 않고 처리를 하는 방법이다. OWF 는 코어의 수가 많아지는 경우 wave-front 의 특징으로 인해 유휴 시간이 많이 발생하게 되는 부분을 메워줄 수 있어 성능을 향상시킬 수 있는 장점을 가진다. 그러나 [6]에서 제시하고 있는 기법은 순차수행을 할 수 밖에 없는 엔트로피 디코딩과 같은 부분에 대한 처리에 있어 우리의 기법과 차이를 보인다. 순차 수행을 담당하는 쓰레드가 하나밖에 없기 때문에, 순차 수행 처리가 병목이 되는 경우 병렬화의 성능이 높아도 성능 향상을 얻을 수가 없다. 우리의 기법은 엔트로피 디코딩과 같은 순차 수행이 필요한 부분의 다중 쓰레드 처리를 할 수 있기 때문에 병목이 제거된다.

HEVC 디코더 알고리즘을 SIMD 최적화를 통해 성능을 향상시키는 기법을 제시하는 논문[7] 또한 존재한다. [7]에선 HEVC 디코더에서 수행 시간 비중이 높은 움직임 보정, Adaptive Loop Filter(ALF)와 Deblocking Filter(DF)등의 필터, 정수변환 등의 처리에 SIMD 최적화를 적용하였다. SIMD 최적화를 통해 높은 성능을 얻었으나, 해당 기법은 멀티코어가 보급화된 현 시점에서 CPU 내의 모든 코어를 최대한 활용하지 못한다. 멀티 쓰레딩 처리와

SIMD 최적화를 모두 적용했을 때 최대의 성능을 얻을 수 있으며 본 논문에서는 SIMD 최적화까지 적용한 병렬 처리 기법을 제안한다.

HEVC 알고리즘을 GPU 를 통해 가속화 시키는 연구[8] 또한 이루어지고 있다. 논문 [8]에서는 작업간의 의존관계가 없는 DF 를 GPU 를 통해 병렬 실행시키는 방법을 제안하고 있으며, GPU 가속을 통해 2 배의 성능을 얻었다. 그러나 해당 기법은 GPU 에서의 작업을 위해 DF 전 후로 배리어를 통한 명시적인 동기화를 거쳐야 하기 때문에 멀티쓰레딩에서 얻을 수 있는 성능향상을 저해시킨다.

대표적인 기존 논문들과 비교할 때, 본 논문의 병렬화 기법은 다음과 같은 장점을 가진다. 첫째로, 제안하는 기법은 인코딩 옵션을 강제하지 않는다. 표준 규격인 main profile 을 준수하는 모든 예제 영상에 대해 적용이 가능하며, 타일과 슬라이스 병렬화처럼 압축률 손실과 같은 문제를 가지지 않는다. 둘째로, 멀티쓰레딩 병렬화와 SIMD 최적화가 모두 적용된다. CPU 활용도를 최대화하도록 쓰레드를 구성할 수 있으며, SIMD 를 지원하는 CPU 를 이용할 수 있다. 셋째로, 순차 수행하는 부분으로 인한 병목 현상이 발생하는 것을 막을 수 있다. 마지막으로, 태스크 그래프 모델을 도입하였기 때문에 다양한 아키텍쳐에 따라 태스크의 매핑이 자유롭다.

제 3 장 배경

본 논문에서 병렬화의 목표로 하고 있는 HEVC 코덱은 기존 코덱인 MPEG-4 와 H.264/AVC 와는 다른 특징적인 요소들을 포함하고 있다. 이 장에서는 HEVC 의 특징들과 본 논문의 진행에 앞서 이해가 필요한 내용들을 다루려고 한다. 그림 1 은 HEVC 인코더의 구조를 나타낸 블록 다이어그램이다.

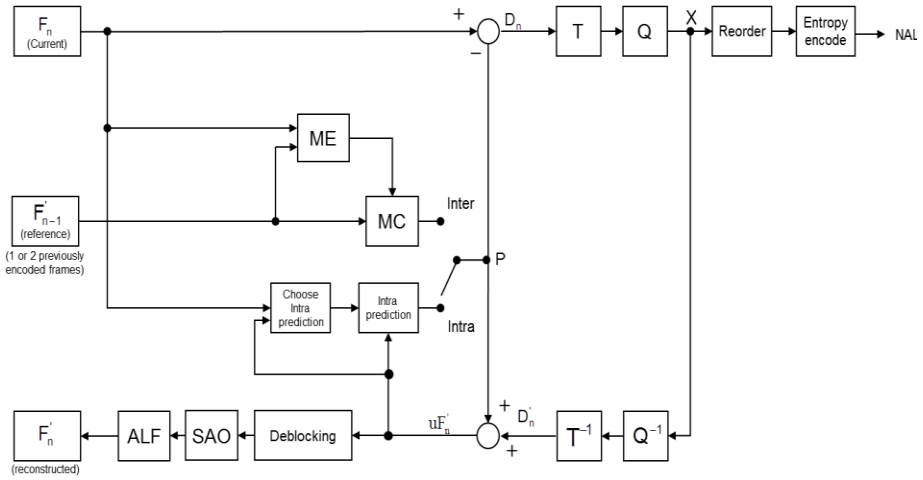


그림 1 HEVC 인코더의 구조

HEVC 에서는 정해진 크기의 매크로 블록이 사용되지 않고, 쿼드 트리 형태의 구조를 가진 Coding Unit (CU)가 사용된다. CU 는 재귀적으로 4 개의 쿼드 트리 형태로 쪼개질 수 있으며, 최대 깊이는 3 으로 정의된다. CU 중에 가장 큰 단위의 CU 를 Largest Coding Unit (LCU)라고 하며, 일반적으로 64x64 픽셀의 크기가 사용된다. 정해진 크기 단위가 아니라 영상의 복잡도에 근거하여 가변 크기를 사용할 수 있게 되었기 때문에 HEVC 는 기존 코덱 대비 높은 성능을 얻을 수 있게 되었다. CU 는 다시 인터 예측 또는 인트라 예측에서 쓰이는 단위인 Prediction Unit (PU)와 역변환에 쓰이는 단위인 Transform Unit (TU)로 나누어진다.

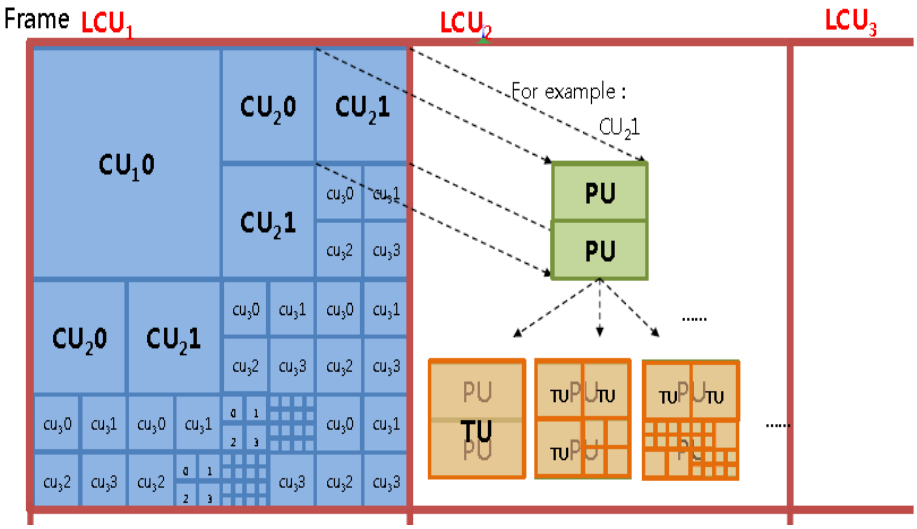


그림 2 HEVC의 Coding Unit (CU), Prediction Unit (PU), Transform Unit (TU) 구조

인코더 루틴에서 기존 코덱과 마찬가지로 움직임 추정, 변환, 양자화 등을 거친 뒤 엔트로피 코딩을 이용하여 부호화를 진행한다. HEVC 에서는 Context-based Binary Arithmetic Coding (CABAC)이 사용된다. 디코딩 루틴은 반대로 엔트로피 디코딩을 수행한 뒤, 역양자화, 역변환, 움직임 보정을 진행한다.

HEVC 에는 Deblocking Filter (DF)와 Sample Adaptive Offset (SAO) 2 가지 필터가 사용된다. 이전의 MPEG-4 나 H.264/AVC 의 경우 후처리 필터만이 존재했고, 필터링은 전체 알고리즘과는 별개로 적용되어 필터 후 이미지는 참조 프레임으로 사용되지 않았으나, HEVC 에서는 필터 후 이미지가 참조 프레임으로 사용되어 좀 더 정확한 이미지 복원이 가능해졌다. DF 는 CU 의 경계 부분에 적용되는 필터이며, 내부적으로 수직 방향과 수평 방향 필터로 나누어진다. SAO 는 독립적인 단위의 쿼드 트리를 이용해 진행되며, DF 와 달리 경계뿐만이 아닌 전 영역에 대해 수행될 수 있다. 이러한 필터들을 통해 움직임 추정과 보정만으로는 처리할 수 복원되지 않는 오차들을 줄일 수 있어 필터 후 이미지를 참조 프레임으로 사용할 때 복원의 정확도가 높아지는 장점이 있다.

제 4 장 HEVC 디코더 분석

4.1 HEVC 디코더 알고리즘 흐름 분석

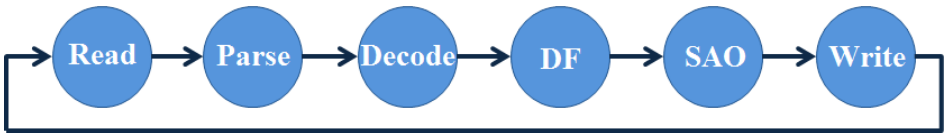


그림 3 간단한 순차수행 HEVC 태스크 플로우

HM 기준 소프트웨어 HM 11.0 디코더의 전체적인 수행 과정을 간단하게 태스크 그래프의 형태로 나타내면 그림 3과 같다. Read 태스크는 부호화된 파일 혹은 네트워크에서부터 네트워크 전송에 최적화된 자료구조인 Network Abstraction Layer (NAL) 단위로 구성된 바이트 스트림을 읽어낸다. 읽어낸 스트림을 전달받은 Parse 태스크는 엔트로피 디코딩을 수행하여 프레임 디코딩에 필요한 정보를 추출해낸다. 우선적으로 다수의 프레임의 디코딩에 사용되는 매개변수를 저장한 Sequence Parameter Set (SPS)를 디코딩하고, 하나의 프레임에 사용되는 매개변수들을 저장한 Picture Parameter Set (PPS)를 디코딩한다. 그 다음으로 슬라이스 헤더를 디코딩하고 SAO등의 필터에 사용될 매개변수들을 디코딩한다. 매개변수들의 디코딩이 완료된 후 프레임 내의 CU들에 대해 엔트로피 디코딩을 진행한다.

매개변수 및 CU에 대한 정보가 모두 추출된 다음엔 역양자화 및 역변환, 인트라 예측 혹은 인터 예측, 움직임 보정 등의 작업이 Decode 태스크에서 진행된다. 해당 과정이 완료되면 프레임의 YUV 정보가 대략적으로 구성되며, 마지막으로 필터를 통해 영상의 보정이 이루어진다. HEVC는 main profile 기준으로 Deblocking Filter (DF)와 Sample Adaptive Offset (SAO)의 2개의 필터를 가지고 있으며, 각각 DF 태스크, SAO 태스크가 해당 필터를 수행하게 된다. 이 과정까지 모두 완료되면 하나의 프레임에 대한 YUV정보가 모두 완성

되며, 이를 Write 태스크에서 출력 후 다음 프레임에 대해 위의 과정을 반복하여 수행한다.

4.2 HEVC 디코더 태스크 간 의존관계 분석

순차 수행으로 구현된 HM 기준 소프트웨어는 한 종류의 작업을 모두 처리한 뒤에 다음 작업으로 진행되도록 구성되어 있으며, 이러한 작업 순서를 기본으로 하여 태스크 간의 의존관계가 구성되어 있다. 태스크 단위로 멀티코어에서의 병렬 처리를 진행할 때는 태스크의 실행 순서가 정해지지 않게 되므로, 동일한 결과를 내는 병렬 알고리즘을 구성하기 위해서는 각 작업 간에 어떻게 의존관계가 구성되어 있는지를 파악해야만 한다.

4.2.1 태스크 내부 의존관계

위에서 구분 지은 태스크들은 내부적으로 또 다시 LCU 단위의 연산이 이루어지며, 해당 연산들 사이의 의존관계가 나타난다. 이 장에서는 태스크 내부 LCU 단위 연산 간의 의존관계를 분석한다.

4.2.1.1 Read, Parse 태스크 내부 의존관계

Read 태스크는 비트 스트림을 읽어 들이는 작업을 진행하기 때문에 기본적으로 순차적인 수행이 요구된다. 파일이나 네트워크로부터의 스트림을 임의 접근을 하려는 시도는 성능을 급격히 하락시킬 수 있다. Parse 태스크는 엔트로피 디코딩의 특성상 읽어 들인 스트림의 문맥에 따라 디코딩이 이루어진다. 그러므로 Parse 태스크 또한 순차적인 수행이 이루어져야 한다.

4.2.1.2 Decode 태스크 내부 의존관계

Decode 태스크는 역변환, 예측, 움직임 보정 등의 작업이 LCU 단위로 진행된다. 이 때 인트라 예측이 진행 되면 각 LCU는 상단 3개의 LCU와 좌측 LCU에 대해 내부 의존관계가 존재한다. 따라서 참조 LCU들의 Decode 태스크 수행이 완료되어야 해당 LCU에 대한 작업이 실행 될 수 있다.

4.2.1.3 DF 태스크 내부 의존관계

DF 태스크는 Deblocking Filter를 각 LCU단위로 진행하며, 내부적으로 수직 방향과 수평 방향 필터로 구분되어 있다. 수직 방향 필

터는 내부적으로는 의존관계를 가지지 않아 외부 의존관계만 만족되면 진행될 수 있으며, 수평 방향 필터는 좌측, 자신, 그리고 우측의 LCU에 대해 수직 방향 필터가 완료되어야 진행될 수 있다.

4.2.1.4 SAO 태스크 내부 의존관계

SAO 태스크는 Sample Adaptive Offset Filter를 각 LCU단위로 진행한다. SAO는 주변 LCU를 참조하긴 하나, SAO가 완료된 후가 아닌 DF가 완료된 후의 값을 참조하기 때문에 SAO 작업간의 내부 의존관계는 존재하지 않는다.

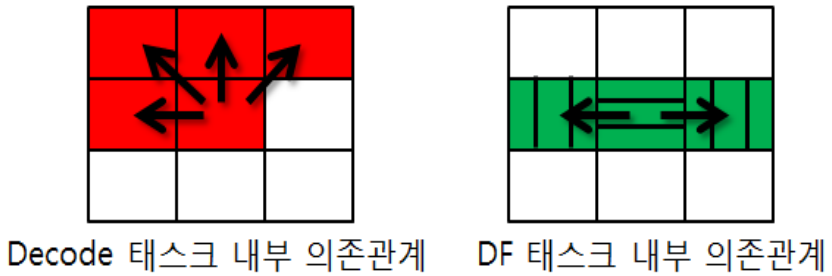


그림 4 태스크 내부 의존관계

4.2.2 태스크 간 의존관계

태스크 내부의 의존관계뿐 아니라 태스크들끼리의 의존관계 또한 존재한다. 이 또한 LCU단위의 의존관계 분석을 진행한다.

4.2.2.1 Parse 태스크와 Decode 태스크 간 의존관계

Parse 태스크가 프레임 간, 그리고 프레임 내부에서만 참조하는 매개변수들을 디코딩하고 나면 LCU들에 대한 엔트로피 디코딩을 진행한다. Parse 태스크가 끝난 LCU에 대해서는 바로 Decode 태스크가 실행 가능하다.

4.2.2.2 Decode 태스크와 DF 태스크 간 의존관계

DF 태스크는 앞서 언급한대로 수직 방향과 수평 방향 필터로 나뉘어 있으며, 의존관계에 의해 수직 방향 필터가 먼저 실행된다. 수직 방향 필터는 상단과 하단의 LCU에 대해 Decode가 완료된 후의

값을 필요로 하기 때문에 두 LCU에 대해 의존관계를 가진다.

4.2.2.3 DF 태스크와 SAO 태스크 간 의존관계

SAO 태스크는 매개변수에 따라 주변을 참조하는 방향이 결정된다. 0°, 45°, 90°, 135°의 네 가지 방향이 존재하기 때문에, 주변 8개의 LCU 모두가 DF 처리 완료되어야 SAO를 실행할 수 있다.

4.2.2.4 SAO 태스크와 Decode 태스크 간 의존관계

SAO 태스크는 순서상 Decode가 완료된 후에 진행되므로, 같은 프레임 내에서는 Decode는 SAO에 의존관계를 가지지 않는다. 그러나 외부 예측을 진행하는 경우, Decode 태스크는 디코딩이 완전히 완료된 이전 프레임들을 참조한다. 이 경우 Decode 태스크는 이전 프레임의 SAO 태스크에 대해 의존관계를 가진다. 이전 프레임에 대한 참조 범위는 고정되어 있지 않고 인코더 옵션으로 주어지기 때문에, 설정에 따라 의존관계가 달라지게 된다.

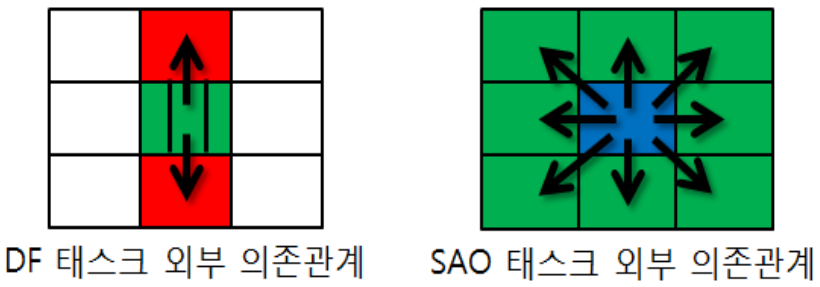
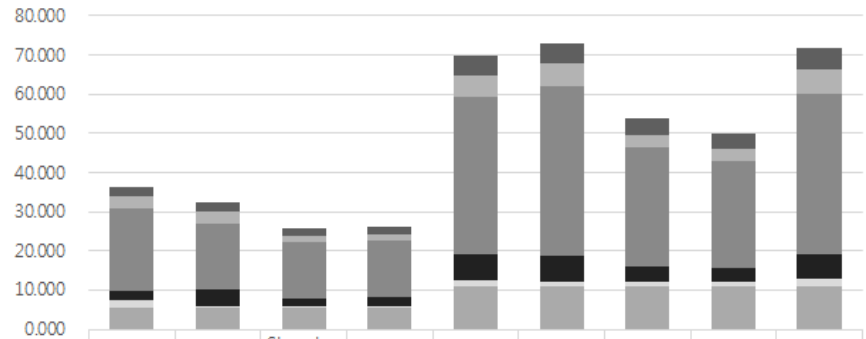


그림 5 태스크 외부 의존관계

4.3 태스크 별 수행시간 분석

그림 6과 7은 9개의 예제 영상들로 순차 수행 HM 소프트웨어를 실행했을 때 각 태스크의 수행시간을 표시한 그래프와, 각 태스크의 수행시간 비율을 표시한 그래프이다. NebutaFestival, PeopleOnStreet, SteamLocomotiveTrain, Traffic 예제는 2560x1600 해상도이고, 나머지 예제들은 3840x2160 해상도이다. 표에 나타나 있는 시간은 150프레임을 디코딩하는데 걸린 시간으로 단위는 초이다. 전체적인 양상을 보았을 때 가장 많은 시간을 차지하

는 태스크는 Decode 태스크이며, 그 뒤로 Parse 태스크와 DF 태스크가 비슷한 비율을 보였다. 그러므로 성능 향상을 위해선 Decode 태스크의 최적화 및 병렬화가 가장 중요하다고 볼 수 있다. 여기서 주의하여 보아야 할 점은 Read 태스크와 Parse 태스크인데, 이 두 태스크는 순차 수행을 필요로 하기 때문에 병렬성을 얻을 수 없다. 전반적으로 둘을 합쳐 13% 수준의 비율을 보이므로 순차 수행이 차지하는 비중이 높지 않다고 할 수 있으나, 4K 해상도에 대해 실시간 성능을 만족시키려면 순차 수행의 비중이 30FPS의 제한시간인 33.33ms를 넘어서는 안된다. 그러나 4K 예제들의 경우 Parse 태스크의 수행시간이 33.33ms를 넘기 때문에, 최적화를 충분히 하거나 높은 성능의 시스템을 써서 그 외의 부분을 30FPS안에 끝나도록 처리해도, 순차 수행 부분이 병목이 되어 실시간 성능을 만족시킬 수가 없게 된다. 그러므로 실시간 성능을 만족시키려면 순차 수행되는 Parse 태스크에 대하여 가속화할 수 있는 방법이 필요할 것이다.



	NebutaFestival	PeopleOnStreet	SteamLocomotiveTrain	Traffic	Crowd	DucksTakeOff	InToTree	OldTownCross	ParkJoy
Read	2.564	2.393	1.928	1.990	4.942	5.007	4.013	4.020	5.245
Parse	2.974	2.984	1.562	1.815	5.437	5.714	3.230	3.149	6.214
Decode	20.965	16.614	14.429	14.188	40.022	43.246	30.420	27.130	40.851
DF	2.638	4.321	1.981	2.236	6.925	6.545	4.036	3.725	6.514
SAO	1.784	0.569	0.495	0.577	1.387	1.442	1.028	0.979	1.719
Write	5.356	5.362	5.393	5.390	10.727	10.659	10.725	10.845	10.852
Etc	0.099	0.098	0.098	0.098	0.203	0.203	0.202	0.205	0.204

그림 6 예제 별 태스크의 수행 시간 누적 그래프와 표

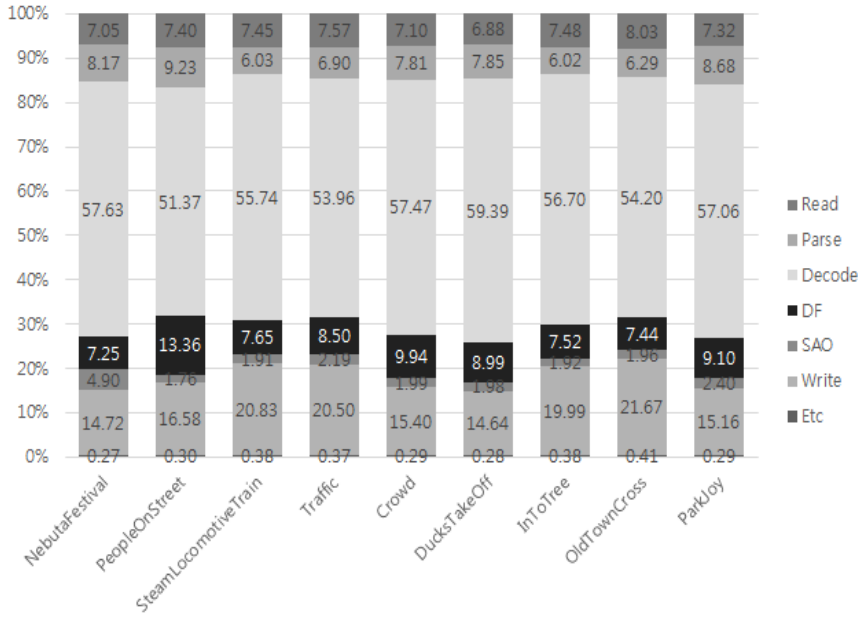


그림 7 예제 별 태스크의 수행시간 비율 그래프

제 5 장 제안하는 병렬 처리 기법

5.1 태스크 그래프 모델 기반 병렬 처리 기법

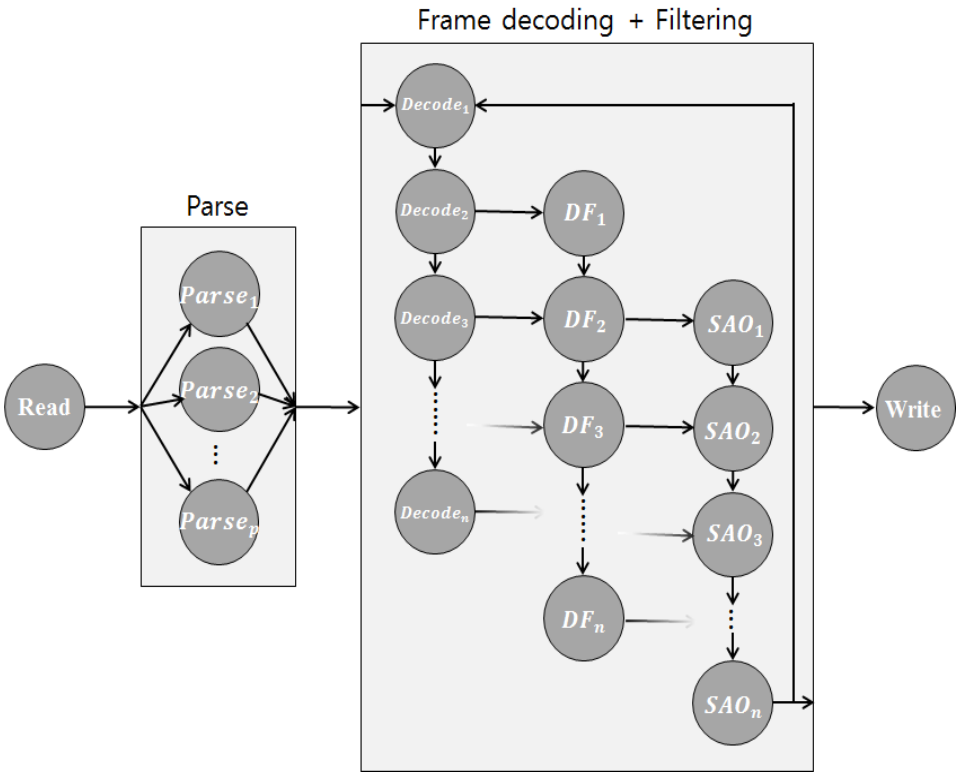


그림 8 제안하는 HEVC 태스크 그래프의 형태

그림 8 은 4 절에서 분석한 HEVC 의 태스크 의존 관계 및 수행시간비율을 통해 시간당 처리량을 최대화할 수 있도록 구성한 병렬화된 태스크 그래프를 나타낸다. 각 태스크가 하는 작업은 그림 3 에서의 태스크들과 동일하다. 차이점은 Decode 태스크, DF 태스크, SAO 태스크들이 LCU 행 단위로 분리되었다는 점이다. 그러므로 프레임 안에 n 개의 LCU 행이 존재한다면, 세 태스크들은 n 개의

행에 대해 하나씩 생성된다. 세 개의 태스크들은 각각 자신의 위 LCU 행의 동일 태스크에 내부 의존관계를 가지고 있고, DF 태스크는 한 줄 아래의 Decode 태스크에게, SAO 태스크는 한 줄 아래의 DF 태스크에게 태스크 간 의존관계를 가지고 있다. 해당 의존관계가 그래프에서 간선으로 나타나 있다. 이 때 간선은 해당 라인에 대해 태스크가 모두 종료해야 의존관계가 해소된다는 의미가 아니며, 내부적으로 LCU 단위로 의존관계가 정의되어있다. Decode 태스크가 SAO 태스크에 가지는 의존관계는 그림에서 가장 마지막 LCU 행을 처리하는 SAO 태스크에게 의존관계를 가진 것으로 표현했으나, 이는 외부 예측 시 참조 범위가 모든 범위일 경우에 대해서만 표현한 것이다. 실제로는 참조 범위를 고려하여 의존관계가 생성된다. 또 다른 차이점은 Parse 태스크를 여러 개 생성 가능하다는 점이다. Read 에서 하나의 프레임에 대한 스트림을 읽어들이고 이를 Parse 에 전달하면, Parse 는 내부적으로 가진 여러 개의 Parse 태스크에게 순서대로 배당한다. 즉 첫 번째 스트림은 $Parse_0$이, 두 번째 스트림은 $Parse_1$가, p 번째 스트림은 $Parse_p$가 처리한다. 이와 같은 방법을 통해 여러 개의 프레임에 대한 엔트로피 디코딩이 동시적으로 실행될 수 있게 된다.

위와 같이 태스크 그래프의 형태로 HEVC 디코더를 묘사하면, 각 태스크들 간의 의존관계가 명확하기 때문에 그래프 상에서 부모에 해당하는 태스크들이 모두 실행되고 나면 자연스럽게 실행 가능한 상태가 된다. 그림 8 은 8x4 의 LCU 들로 구성된 프레임에 대한 병렬화 양상을 나타낸 그림이다. 프레임에 대한 Read 태스크와 Parse 태스크, Decode, DF, SAO 태스크가 파이프라이닝 되어 실행되는 것을 볼 수 있다.

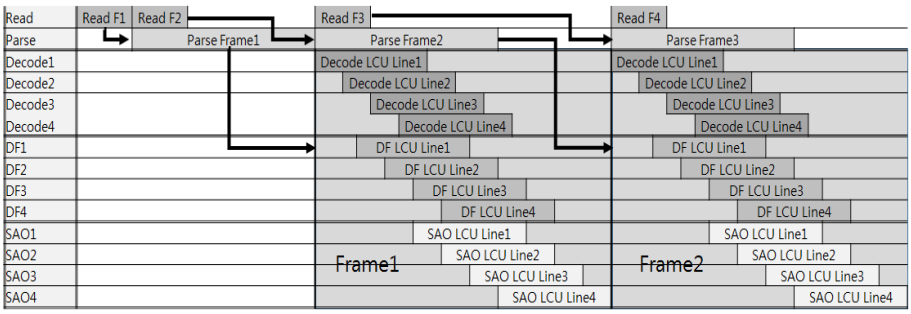


그림 9 제안하는 태스크 그래프 기반 병렬화 모델의 병렬화 양상

위와 같은 태스크 그래프 모델을 사용한 병렬 처리 기법의 또 다른 장점은 알고리즘이 완전히 독립된 태스크들과 그들간의 의존관계로 재구성이 되었기 때문에, 아키텍쳐에 따라 태스크들의 매핑과 스케쥴링 순서 등을 자유롭게 선택할 수 있다는 점이다. 이러한 장점을 통해 다양한 아키텍쳐에 대해 유연하게 적용이 가능하며, 가장 최선의 매핑과 스케쥴링 방법을 탐색해볼 수 있게 된다. 예를 들어 그림 9 와 같이 6 개의 코어를 가진 아키텍쳐를 가정해보자. 해당 아키텍쳐는 코어가 충분하지 않으므로, 각 코어에 태스크들을 어떻게 배치하고 스케쥴링을 할지를 결정해야만 한다. 위쪽의 경우는 Decode 태스크를 먼저 다 마치고, DF 를 실행하는 순서로 스케쥴링한 것이며, 아래쪽의 경우는 Decode 와 DF 를 섞어서 스케쥴링한 것이다. 위의 경우 각 코어에서 두 번째 Decode 를 실행하려면 의존관계로 인해 대기가 발생한다. 기다리는 시간 동안은 코어가 유휴 상태로 머물기 때문에 성능이 낮아지게 된다. 그러나 두 번째의 경우처럼 의존관계가 먼저 해결되는 DF 를 먼저 배치하면 유휴 상태가 발생하지 않고 더 빨리 마칠 수 있게 된다. 제안하는 방법은 이러한 방식으로 병렬화 동작을 분석한 뒤에 아키텍쳐의 특성에 맞춰 유연하게 더 좋은 성능을 낼 수 있는 방식을 선택하여 병렬 처리가 가능하다.

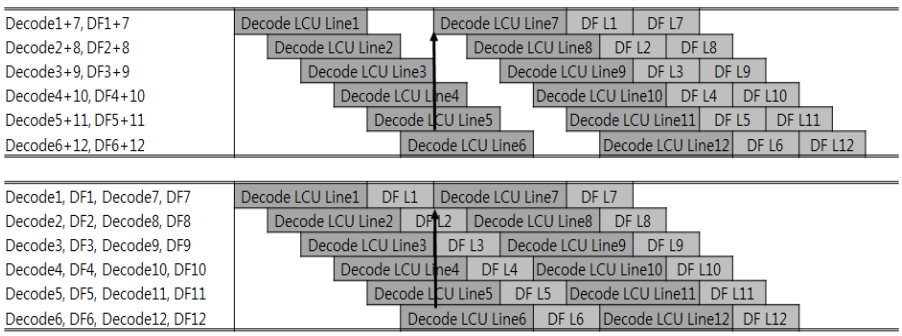


그림 10 스케쥴링 순서에 따른 병렬화 차이

앞서 수행시간 비율 분석에서 보았듯이 스트림 접근 또는 엔트로피 디코딩의 특성으로 인해 순차 수행이 필요한 Read 와 Write 에 병렬화 성능이 제한될 수 있다. 이를 위해 제안하는

방법에선 Parse 태스크를 다중으로 생성하여 배치한다. 그림 10 은 성능 병목 현상이 나타나는 상황을 보여준다. 코어가 충분히 많아 병렬화가 잘 되거나, 혹은 순차 수행되는 부분이 수행 시간이 큰 경우에 순차 수행이 병목으로 나타날 수 있다. 이런 경우 코어를 아무리 많이 써서 Decode, DF, SAO 태스크를 병렬화 시켜도, 순차수행이 필요한 Parse 태스크가 빨라지지 못하기 때문에 성능이 제한된다. 이러한 예제들에 대해서 Parse 태스크를 2 개 이상 배치하면 성능 병목 현상이 해소되고 성능이 비약적으로 향상된다. 이 경우 Read 태스크는 한번 수행해서 데이터를 읽을 때 여러 개 생성된 Parse 태스크들에게 한번씩 순서대로 데이터를 전달한다. Parse 태스크는 Read 태스크로부터 데이터를 전달받으면 바로 엔트로피 디코딩을 진행할 수 있기 때문에, 여러 프레임에 대하여 Parse 태스크가 병렬적으로 실행될 수 있게 된다. 이 경우 메모리의 여러 프레임에 대한 매개변수를 동시에 유지하고 있어야 하기 때문에 메모리 사용량이 늘어나지만, 동시에 여러 프레임의 YUV 정보를 유지하는 HEVC 알고리즘에서 매개변수가 차지하는 메모리 사용량 비율은 미미하기 때문에 성능에 악영향을 끼치지 못한다. 그림 11 은 같은 예에 대하여 Parse 태스크를 2 개 사용했을 때의 병렬화 예를 보여준다. 시간당 처리량이 Parse 태스크를 1 개 사용했을 때에 비하여 크게 향상될 수 있다는 것을 알 수 있다.

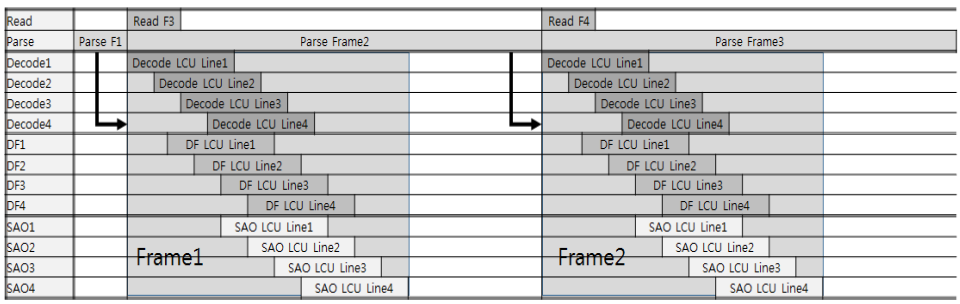


그림 11 Parse 태스크의 비중이 높은 경우의 성능 병목 현상

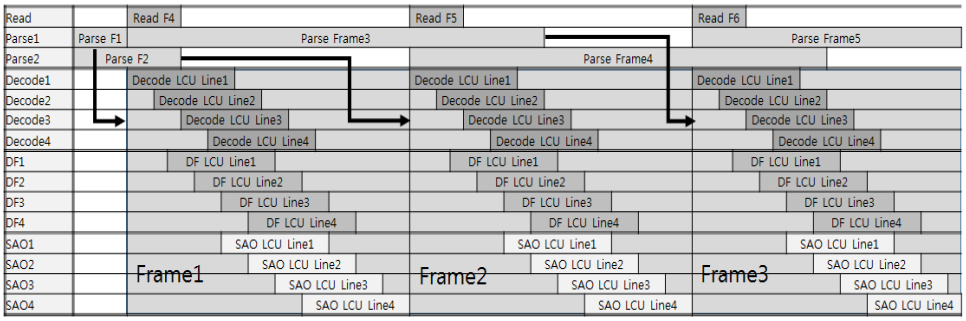


그림 12 Parse 태스크를 2개 사용했을 때의 병렬화 실행 양상

5.1.1 태스크 단위 변환 및 병렬화 구현

제안하는 태스크 단위의 병렬화를 진행하기 위해선 기존의 HM 소스코드를 각 태스크 단위로 독립적인 코드로 분리하고, 의존관계에 따른 처리를 추가해주어야만 한다. 기존의 HM 소스코드는 순차 수행을 가정하여 동작하기 때문에 중첩된 함수 호출 및 사용되는 데이터들을 우선적으로 분석해야 할 필요가 있다.

```
TAppDecTop::decode()
{
    InputByteStream byteStream
    while (!!byteStream)
    {
        InputNALUnit nalu = read(nalUnit)

        bool bNewPicture = m_cTDecTop.decode(nalu)
        if (bNewPicture)
        {
            m_cTDecTop.executeDeblockAndAlf(uiPOC, pcListPic);
            xWriteOutput(pcListPic, nalu.m_temporalId);
        }
    }
    xFlushOutput(pcListPic);
}

TDecTop::decode(InputNALUnit& nalu)
{
    switch (nalu.m_nalUnitType)
    {
        case NAL_UNIT_VPS:   xDecodeVPS(); return false;
        case NAL_UNIT_SPS:   xDecodeSPS(); return false;
```

```
        case NAL_UNIT_PPS:   xDecodePPS(); return false;
        case NAL_UNIT_APS:   xDecodeAPS(); return false;
        case NAL_UNIT_SEI:     xDecodeSEI(); return false;
        case NAL_UNIT_SLICE: return xDecodeSlice(nalu);
    }
    return false;
}

TDecTop::xDecodeSlice(InputNalUnit& nalu)
{
    .....
    m_cEntropyDecoder.decodeSliceHeader();
    if (isNextSlice())
        return true;
    ......
    m_cGopDecoder.decompressGop(nalu.m_Bitstream, pcPic, false);
    return false;
}

TDecTop::executeDeblockAndAlf(UInt&
ruiPOC,TComList<TComPic*>*& rpcListPic)
{
    m_cGopDecoder.decompressGop(NULL, pcPic, true);
}

TDecGop::decompressGop(TComInputBitstream*
pcBitstream,TComPic*& rpcPic, bool bExecuteDeblockAndAlf)

    if (!bExecuteDeblockAndAlf)
    {
        ..... substreams[ui] = pcBitstream->extractSubstream(....);
        m_pcEntropyDecoder->setBitStream();
        m_pcEntropyDecoer->resetEntropy();
        .....

        m_pcSliceDecoder->decompressSlice(pcBitStream,
substream,pcPic ... );
          }
          else { ..... }


TDecSlice::decompressSlice(TComInputBitstream* pcBitstream, .....)
{
```

```
    .....
    for( TComCU* pcCU in decoding_order)
    {
        .....
        pcSbacDecoder->loadContext();
        .....
        m_pcCuDecoder->decodeCU(pcCU);
        m_pcCuDecoder->decompressCU(pcCU);
        .....
        pcSbacDecoder->loadContext();
    }
}
```

위의 소스 코드는 HM 소스코드에서 태스크 단위로 중요한 의미를 가지는 부분들을 정리한 것이다. 가장 상위의 decode 함수에서는 스트림을 계속 읽어들이면서 읽어들인 스트림에 대해 엔트로피 디코딩을 수행한다. 읽어들인 스트림은 Parameter Set (PS)이거나 슬라이스 일 수 있으므로, 각각의 경우에 대해서 디코딩을 실행한다. 슬라이스를 엔트로피 디코딩 하는 경우는 스트림에서 우선 슬라이스 헤더를 디코딩한 뒤에, 슬라이스에 대해 엔트로피 디코딩과 움직임 보정 등의 작업을 수행한다. 만약 디코딩한 슬라이스 헤더가 다음 프레임에 대한 정보임이 확인되면, true 가 반환되어 하나의 프레임에 대해 모두 처리가 완료되었음을 인식하게 하며, 이 경우 완료된 프레임에 대해 필터링을 처리한다.

```
TDecGop::decompressGop(TComInputBitstream* pcBitstream,
TComPic*& rpcPic, bool bExecuteDeblockAndAlf)
{
    if (!bExecuteDeblockAndAlf)
    { ..... }
    else
    {
        // deblocking filter
        .....
        m_pcLoopFiler->loopFilterPic(pcPic);

        //SAO
        .....
        if(getUseSAO() && getSaoEnabledFlag())
            m_pcSAO->SAOProcess(pcPic, getSaoParam());
    }
}
```

```
TComLoopFilter::loopFilterPic(TComPic* pcPic)
{
    for (UInt i = 0; i < getNumCUsInFrame(); i++)
        xDeblockCU(pcCU, EDGE_VER);
    for (UInt i = 0; i < getNumCUsInFrame(); i++)
        xDeblockCU(pcCU, EDGE_HOR);
}

TComSampleAdaptiveOffset::SAOProcess(TComPic* pcPic)
{
 for (UInt i = 0; i < getNumCUsInFrame(); i++)
                processSaoUnit(pcCU);
}
```

필터링을 처리할 때는 DF 와 SAO 의 순서로 실행되며, 두 필터링은 모두 LCU 단위로 for 문을 돌아 처리되도록 되어있다. 위의 원본 HM 소스 코드를 태스크 단위로 코드를 재배치 하면 다음과 같다.

```
ReadTask::main()
{
    InputByteStream byteStream
    while (!!byteStream)
    {
        InputNALUnit nalu = read(nalUnit)

        bool bNewPicture = ReadTask::decode(nalu);
        if (bNewPicture)
        {
            send(ParseTask, BitstreamList);
            receive(WriteTask);
            BitstreamList.clear();
        }
        BitstreamList.add(nalu.m_Bitstream);
    }
}

ReadTask::decode(InputNALUnit& nalu)
{
    switch (nalu.m_nalUnitType)
    {
        case NAL_UNIT_VPS:   xDecodeVPS(); return false;
        case NAL_UNIT_SPS:   xDecodeSPS(); return false;
```

```
            case NAL_UNIT_PPS:   xDecodePPS(); return false;
            case NAL_UNIT_APS:   xDecodeAPS(); return false;
            case NAL_UNIT_SEI:   xDecodeSEI(); return false;
            case NAL_UNIT_SLICE: return ReadTask::decodeSlice(nalu);
        }
        return false;
}

ReadTask::decodeSlice(InputNalUnit& nalu)
{
        ...
        m_cEntropyDecoder.decodeSliceHeader();
      if (isNextSlice())
            return true;
        ......
        return false;
}
```

Read 태스크는 하나의 프레임에 대해 스트림을 읽어 들이기만 한 뒤에, 읽어 들인 정보를 Parse 태스크로 전달해 주어야 한다. 여기선 추가적으로 PS 및 슬라이스 헤더까지는 디코딩을 수행하며, 그 외의 경우 즉, 슬라이스에 대한 스트림은 기록해두었다가, 다음 프레임이 인식되면 쌓아두고 있던 슬라이스 스트림을 Parse 태스크로 전달한다. 그리고 Write 태스크로부터 프레임을 모두 처리했다는 정보가 도착하면 다시 Read 태스크를 재개한다.

```
ParseTask::main()
{
        receive(ReadTask, BitstreamList);
        for each (Bitstream bitstream in BitstreamList)
        {
            ParseTask::decodeGop(bitstream, pcPic);
        }
        send(DecodeTask, pcPic);
}

ParseTask::decodeGop(TComInputBitstream* pcBitstream, TComPic*&
pcPic)
{
        ......
        substreams[ui] = pcBitstream->extractSubstream(....);
        m_pcEntropyDecoder->setBitStream();
        m_pcEntropyDecoer->resetEntropy();
```

```
    .....

    ParseTask::decodeSlice(pcBitstream, substream, pcPic, ...);
}

ParseTask::decodeSlice(TComInputBitstream* pcBitstream, .....)
{
    .....
    for( Int iCUAddr = iStartCUAddr; iCUAddr < rpcPic->getNumCUsInFrame(); ...)
    {
        .....
        pcSbacDecoder->loadContext();
        .....
        m_pcCuDecoder->decodeCU(pcCU);
        .....
        pcSbacDecoder->loadContext();
    }
}
```

Parse 태스크는 Read 로부터 슬라이스에 대한 스트림을 받은 뒤, 엔트로피 디코딩 만을 처리하고 Decode 태스크로 전달해야 한다. 전달받은 스트림들에 대해 엔트로피 디코딩을 호출하며, 처리해야 되는 순서에 따라 각 LCU 를 엔트로피 디코딩한다. 이 때 decompressCU 가 호출되지 않는다는 것을 볼 수 있다.

```
DecodeTask::main()
{
    receive(ParseTask, pcPic);

    for (UInt i = 0; i < getNumCUsInFrame(); i++)
        m_pcCuDecoder->decompressCU(pcCU);

    send(DFTask, pcPic);
}

DFTask::main()
{
    receive(DecodeTask, pcPic);

    for (UInt i = 0; i < getNumCUsInFrame(); i++)
        xDeblockCU(pcCU, EDGE_VER);
    for (UInt i = 0; i < getNumCUsInFrame(); i++)
        xDeblockCU(pcCU, EDGE_HOR);
```

```
    send(SAOTask, pcPic);
}

SAOTask::main()
{
    receive(DFTask, pcPic);

    for (UInt i = 0; i < getNumCUsInFrame(); i++)
        processSaoUnit(pcCU);

    send(WriteTask, pcPic);
}

WriteTask::main()
{
    receive(SAOTask, pcPic);

    WriteOutput(pcListPic, nalu.m_temporalId);

    send(ReadTask);
}
```

Decode, DF, SAO 는 원래 코드에 있던 대로 for 문을 통해 각 LCU 에 대해 작업을 처리한다. Decompress 의 경우 기존엔 엔트로피 디코딩의 순서에 따라 처리가 되었으나, 엔트로피 디코딩이 분리되었기 때문에 엔트로피 디코딩 순서에는 무관하게 처리할 수 있게 되었다. 처리가 완료되면 각각 다음 태스크에게 전달하여 순서대로 작업이 진행될 수 있게 하며, 마지막으로 SAO 가 종료하면 Write 태스크가 활성화되어 출력 처리를 한 뒤에 Read 태스크를 활성화 시킨다.

```
ReadTask::main()
{
    while (!!m_byteStream)
    {
        InputNALUnit nalu = read(nalUnit)

        bool bNewPicture = ReadTask::decode(nalu);
        if (bNewPicture)
            break;

        BitstreamList.add(nalu.m_Bitstream);
```

```
    }

    send(ParseTask[iterationCount%p], BitstreamList);
    BitstreamList.clear();
}

ParseTask::main()
{
    receive(ReadTask, BitstreamList);
    for each (Bitstream bitstream in BitstreamList)
    {
        ParseTask::decodeGop(bitstream);
    }

    for each (Task task in DecodeTaskList, DFTaskList, SAOTaskList)
        send(task, pcPic);
}

DecodeTask::main
{
    receive(ParseTask[iterationCount%p], pcPic);

    for (int i = 0; i < getFrameWidthInCU(); ++i)
    {
        WaitDependency();
        pcCU = getCU(pcPic, i, idxY);
        m_pcCuDecoder->decompressCU(pcCU);
        SignalDependency();
    }
}

DFTask::main
{
    receive(ParseTask, pcPic);

    pcBeforeCU = null
    for (int i = 0; i < getFrameWidthInCU()+1; ++i)
    {
        WaitDependency();
        pcCU = getCU(pcPic, i, idxY);
        if (pcCU != null)
            xDeblockCU(pcCU, EDGE_VER);
        if (pcBeforeCU != null)
```

```
            xDeblockCU(pcBeforeCU, EDGE_HOR);
        pcBeforeCU = pcCU;
        SignalDependency();
    }
}

SAOTask::main
{
        receive(DFTask, pcPic);

        for (int i = 0; i < getFrameWidthInCU(); ++i)
        {
            WaitDependency();
            pcCU = getCU(pcPic, I, idxY);
            processSaoUnit(pcCU);
            SignalDependency();
        }

        if (idxY == getFrameWidthInCU - 1)
        {
            signal(DecodeTask[0]);
            send(WriteTask, pcPic);
        }
}

WriteTask::main
{
    receive(SAOTask, pcPic);

    WriteOutput(pcListPic, nalu.m_temporalId);

    send(ReadTask);
}
```

병렬화를 진행한 HM 의 수도코드는 위와 같다. 여기서 send 와 receive 로 처리되는 Edge 들은 데이터를 보관하는 버퍼를 가지며, 1 개의 데이터만을 들고 있을 수 있다. Signal 과 wait 로 처리되는 Edge 들은 세마포어를 통한 동기화 역할만을 한다.

Read 태스크는 더 이상 Write 태스크를 기다리지 않는다. Parse 태스크로 연결된 Edge 가 비어있기만 하면 작업을 처리해서 전송한다. 제안하는 병렬화 기법에서는 여러 개의 Parse 태스크를 지원하며, 이런 경우 순서대로 데이터를 전송하므로 send 시에는

어느 차례의 Parse 태스크로 보낼 것인지를 선택한다. Parse 태스크는 Read 로부터 스트림을 받으면 엔트로피 디코딩을 처리한 뒤에, 병렬화를 위해 확장된 Decode, DF, SAO 태스크들에 대해 엔트로피 디코딩된 결과를 전송한다. Decode, DF, SAO 태스크는 Parse 로부터 엔트로피 디코딩 결과를 받으면, 해당 프레임에 대해 활성화된다. 각 태스크는 하나의 LCU 행에 대해 작업을 진행하므로 LCU 행 개수만큼 for 문을 수행한다. 여기서 의존관계에 따라 WaitDependency 와 SignalDependency 가 수행되며, 이는 태스크 그래프에 있는 Edge 에 따라 wait 와 signal 이 구성된다. 의존관계에 따라 wave-front 로 수행이 마무리 되어 마지막 LCU 에 대해 SAO 가 수행되면 Write 태스크에 최종 결과가 전달되고, 첫 번째 Decode 태스크에게 다음 프레임을 시작할 수 있음을 알리는 Signal 이 보내진다.

5.2 SIMD 명령어를 사용한 병렬 처리 기법

영상 코덱은 각 픽셀에 대해 단순 연산을 반복 적용하는 작업이 많은 연산 집약적 소프트웨어이다. 이런 특성에 의해 연속된 여러 개의 데이터에 대해 동일한 작업을 하는 코드를 단일 명령어로 치환할 수 있는 SIMD 명령어를 적용하기가 쉽다. 시스템에서 이러한 SIMD 처리를 지원하는 경우 제안하는 병렬 처리 기법의 SIMD 처리를 적용할 수 있다. 그림 13 은 SISD 처리와 SIMD 처리의 차이점을 나타낸다. 각 픽셀에 대해 처리하는 작업을 f 라는 함수로 둔다면, SISD 는 각 픽셀에 대해 반복적으로 함수 f 를 적용하는 것으로 볼 수 있다. 그러나 SIMD 는 연속된 픽셀 묶음에 대해 함수 f 를 한번에 적용하는 것으로 함수 f 가 한번에 여러 개의 입력을 받아 동시 처리하여 출력을 내주는 것이다.

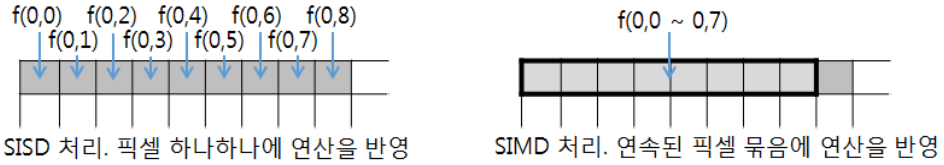


그림 13 SISD 처리와 SIMD 처리의 차이점

SIMD 기반 병렬 처리 기법은 순차 수행 및 파일 접근을 하는 Read 태스크, Parse 태스크, Write 태스크는 SIMD 를 적용하기가 적합하지 않으므로, 커다란 프레임에 대해 작업을 처리하는 Decode 태스크, DF 태스크, SAO 태스크에 대해 적용되었다. SIMD 최적화 작업은 프로파일링을 통해 연산의 비중이 높은 부분들에 대하여 적용되었으며, 대표적으로 역변환, 보간 필터, DF, SAO 내부의 코드들이 SIMD 연산으로 재구현되었다. 본 논문에서는 SIMD 최적화가 적용된 부분들 중 단순 작업을 제외하고 복잡도를 가지는 부분들 중 일부인 수평방향 DF 강 필터, SAO 0 도 Edge Offset(EO) 필터에 대해 구현을 정리한다.

5.2.1 수평방향 DF 강 필터 SIMD 병렬화

DF 는 수직방향 필터와 수평방향 필터로 나뉘어 있으며, 또 다시 내부적으로 필터의 강도 결정에 따라 강 필터와 약 필터로 나뉘어진다. 수직 방향 필터는 세로 경계 부분의 픽셀에 대해 수평방향 8 개의 픽셀을 참조하여 필터링을 하며, 수평 방향 필터는 반대로 가로 경계 부분의 픽셀에 대해 수직방향 8 개의 픽셀을 참조하여 필터링한다. 그림 14 는 수직 방향과 수평 방향 필터의 주변 픽셀 참조를 나타낸다. 필터링을 적용하는 위치를 m4 라고 한다면, 수직 방향은 m4 기준으로 왼쪽으로 4 개, 오른쪽으로 3 개와 m4 자신을 포함한 8 개의 픽셀을 이용해 계산하며, m1 에서 m6 까지 6 개의 값이 업데이트된다. 수평 방향은 필터링을 적용하는 위치를 m4 라 둔다면 m4 기준으로 위쪽으로 4 개, 아래쪽으로 3 개와 m4 자신을 포함한 8 개의 픽셀을 이용해 계산하며, m1 에서 m6 까지 6 개의 값이 업데이트된다.

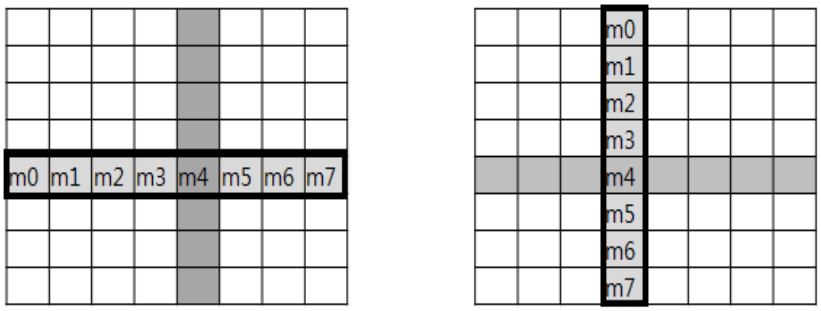


그림 14 DF 수직 방향 필터와 수평 방향 필터의 주변 픽셀 참조

이 때 수평 방향 강 필터는 6 개의 필터링 결과값을 다음과 같이 계산한다.

$$m_1^* = \frac{2m_0 + 3m_1 + m_2 + m_3 + m_4 + 4}{8}$$
$$m_2^* = \frac{m_1 + m_2 + m_3 + m_4 + 2}{4}$$
$$m_3^* = \frac{m_1 + 2m_2 + 2m_3 + 2m_4 + m_5 + 4}{8}$$
$$m_4^* = \frac{m_2 + 2m_3 + 2m_4 + 2m_5 + m_6 + 4}{8}$$
$$m_5^* = \frac{m_3 + m_4 + m_5 + m_6 + 2}{4}$$
$$m_6^* = \frac{m_3 + m_4 + m_5 + 3m_6 + 2m_7 + 4}{8}$$

정확히는 위의 계산에서 각 값에 최소 최대 범위 설정이 있으나 설명의 단순화를 위해 생략한다. 각 픽셀은 주변 픽셀에 대한 작업과 무관하므로 수평 방향 필터를 SIMD 로 구현 가능하다. 그러나 SIMD 는 한번의 연산에 128 비트 단위로 계산이 되기 때문에, 주어진 수식대로 계산한다면 연산량이 매우 많으므로 최적화를 위해 중복되는 연산을 제거할 필요가 있다. 연산에서 중복되는 계산을 최소화한다면 다음과 같이 구현할 수 있다.

$$\alpha = m_3 + m_4 + 2$$
$$\beta = m_2 + \alpha + m_5 + 2$$
$$m_2' = m_1 + m_2 + \alpha$$
$$m_5' = \alpha + m_5 + m_6$$
$$m_1' = 2(m_0 + m_1) + m_2' + 2$$

$$m_3' = m_2' + \beta$$
$$m_4' = m_5' + \beta$$
$$m_6' = 2(m_6 + m_7) + m_5' + 2$$
$$m_1^{new} = \frac{m_1'}{8}, \quad m_2^{new} = \frac{m_2'}{4}, \quad m_3^{new} = \frac{m_3'}{8}$$
$$m_4^{new} = \frac{m_4'}{8}, \quad m_5^{new} = \frac{m_5'}{4}, \quad m_6^{new} = \frac{m_6'}{8}$$

기존의 연산은 덧셈 30 번, 곱셈 4 번, 쉬프트 연산 6 번이 필요한데 반해, 위와 같이 구현하면 덧셈 17 번, 곱셈 2 번, 쉬프트 연산 6 번으로 절반 수준의 연산 횟수로 계산이 끝나게 된다.

5.2.2 SAO 0 도 EO 필터 SIMD 병렬화

SAO 는 각 LCU 에 대해서 내부의 모든 픽셀에 대해 필터링을 진행하는데, EO 에 따라 0 도, 45 도, 90 도, 135 도의 총 4 가지 방향을 참조하여 필터링된다. 0 도 EO 필터의 경우는 참조하는 각도대로 좌우 주변의 값을 이용하여 필터링 되며, 임의의 위치의 픽셀 P[i, j]에 대해 다음과 같은 연산이 적용된다.

EdgeType[i, j]=sign(P[i, j]–P[i, j+1])−sign(P[i, j−1]−P[i, j])+2

연산 결과인 EdgeType 을 인덱스로 하여 EO 테이블 및 클립 테이블에 접근해 필터링 결과값을 찾아낸다. 연산이 각 픽셀에 대해 좌우의 값을 사용하여 이루어 지고 있으므로, SIMD 를 통해 동시적으로 처리가 가능하며 이는 다음 그림 15 와 같으며, 수직방향의 참조가 없으므로 편의를 위해 수직방향의 인덱싱을 생략하였다.

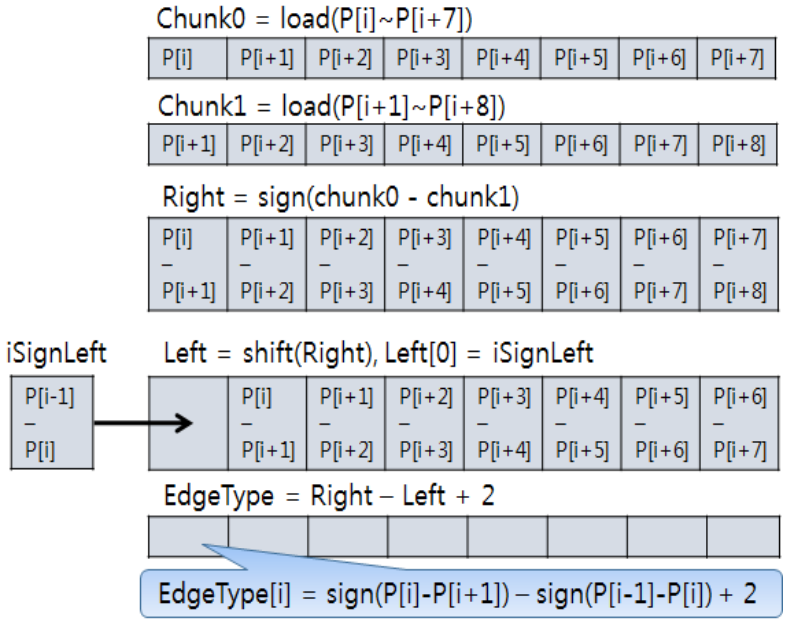


그림 15 SAO 0도 EO 필터 SIMD 병렬화 처리 과정

먼저 8 개의 픽셀을 한번에 읽어 들인다. 그리고 하나 오른쪽의 픽셀에서부터 다시 8 개의 픽셀을 읽어 들인다. 그 후 읽어 들인 두 픽셀묶음을 뺀 뒤에 부호를 구하면, 연산의 첫 번째 항이 완성된다. 연산의 두 번째 항에선 왼쪽을 참조하나, 하나 왼쪽에 대해 다시 읽어 계산하는 것은 비효율적이다. 방금 전의 계산 결과를 오른쪽으로 쉬프트를 하면, 로드 및 뺄셈, 부호추출의 과정 없이 바로 두 번째 항의 값이 나타난다. 문제는 쉬프트를 하면 첫 번째 항목에 대한 값이 없으므로, 이 값은 따로 기록해두었다가 대입한다. 여기서 쉬프트로 없어지는 가장 오른쪽 항목을 기록해두면, 바로 다음번 계산(P[i+8]부터 P[i+15])에 사용할 수 있다. 그 후 두 값을 뺀 뒤에 2 를 더해주면 원하는 결과값을 SIMD 를 통해 묶어 계산이 된다.

제 6 장 성능 분석 실험

제안하는 병렬 처리 기법을 검증하기 위해 HM 11.0 기준 소스 프로그램에 기법을 적용하여 성능 분석 실험을 진행하였다. 실험은 Intel Xeon E5-2640 Hexa Core (Hyper-threading 지원)CPU 가 2 개 장착된 환경에서 진행되었으며, 다양한 해상도의 예제들에 대해 측정되었다. 예제들은 모두 150 프레임씩 인코딩을 한 뒤에 디코딩 시간을 측정하였으며, 인코딩 옵션은 randomaccess 방식에 main profile, QP 22/32/37 을 적용하였다. 또한 디코딩 만의 성능을 확인하기 위해 Write 태스크에 의한 출력은 제거하고 진행하였다. 표 1 는 실험 환경을 자세히 정리한 표이다.

ref. software	HM 11.0	
CPU	Intel Xeon E5-2640 Hexa Core (Hyper-threading) 2개	
Clock Freq	2.5 GHz	
RAM	32.00 GB	
OS	Windows 7 Enterprise K 64bit	
Compiler	Visual Studio 2010	
Encoding Option	Main profile + randomaccess QP22/32/37, 150 Frame	
Sequences	2560x1600 (WQXGA)	NebutaFestival PeopleOnStreet SteamLocomotiveTrain Traffic
	3840x2160 (QFHD)	Crowd DucksTakeOff InToTree OldTownCross ParkJoy

표 1 실험 환경

6.1 Parse 태스크 수 변화에 따른 디코딩 성능

본 논문에서 제안하는 기법은 Parse 태스크의 수를 변화시킴으로써 순차수행이 되어야만 하는 엔트로피 디코딩을 동시 병렬적으로 실행해 성능 병목을 제거하였다. 이 실험을 통해 Parse 태스크의 수 변화에 따른 실제적인 성능 변화 여부와 그 정도를 확인한다. 실험은 성능에 영향을 줄 수 있는 다른 요소들을 모두 통제한 뒤, Parse 태스크의 수를 1 개에서 4 개까지 변화시켜가며 측정하였다. 모두 동일하게 매핑되고 같은 스케쥴링 순서를 가지며, 최대로 병렬화되도록 가질 수 있는 최대한의 쓰레드 수인 34 개씩의 쓰레드를 생성하였다. 디코딩 과정이 최대한으로 병렬화되어 빠르게 실행될 것이므로 엔트로피 디코딩에 의한 병목 현상을 가시적으로 보여줄 수 있다.

그림 16 과 그림 17 을 보면, 2560x1600 예제들과 3840x2160 예제들에 대하여 Parse 태스크 수의 변화에 따른 수행 시간 변화를 확인할 수 있다. 성능을 확인했을 때, Parse 태스크에 의한 성능 변화는 3 개까지는 대체로 상승하며 이후로는 거의 변화가 없는 것을 확인할 수 있다. 또한 QP 가 낮은 경우 Parse 태스크에 의한 성능 변화가 두드러지게 나타난다. QP 는 양자화 과정에 사용되는 수치로, 값이 낮을수록 영상을 원본에 가깝게 표현할 수 있게 되나 그만큼 압축률이 떨어져 부호화된 비트 스트림이 커지게 된다. 그렇기에 QP 가 낮게 인코딩이 된 경우 영상의 복호화 알고리즘이 차지하는 비율보다는 파일을 읽고 엔트로피 디코딩을 하는 부분의 비중이 높아지게 되며, 순차 수행이 많아지므로 병렬화로 인한 이득이 떨어진다. 큰 차이를 보이는 예제인 DucksTakeOff 의 경우 QP22 에서 Parse 태스크를 1 개 썼을 때와 4 개 썼을 때 2.14 배의 성능 차이가 나타났다. 큰 용량을 감수하고서라도 고품질의 영상을 서비스하려는 경우 이와 같은 엔트로피 디코딩의 동시 수행이 필수적이라 할 수 있으며, 높은 QP 를 사용하는 경우에도 성능 향상을 가져오기 때문에 해당 기법의 필요성이 부각된다.

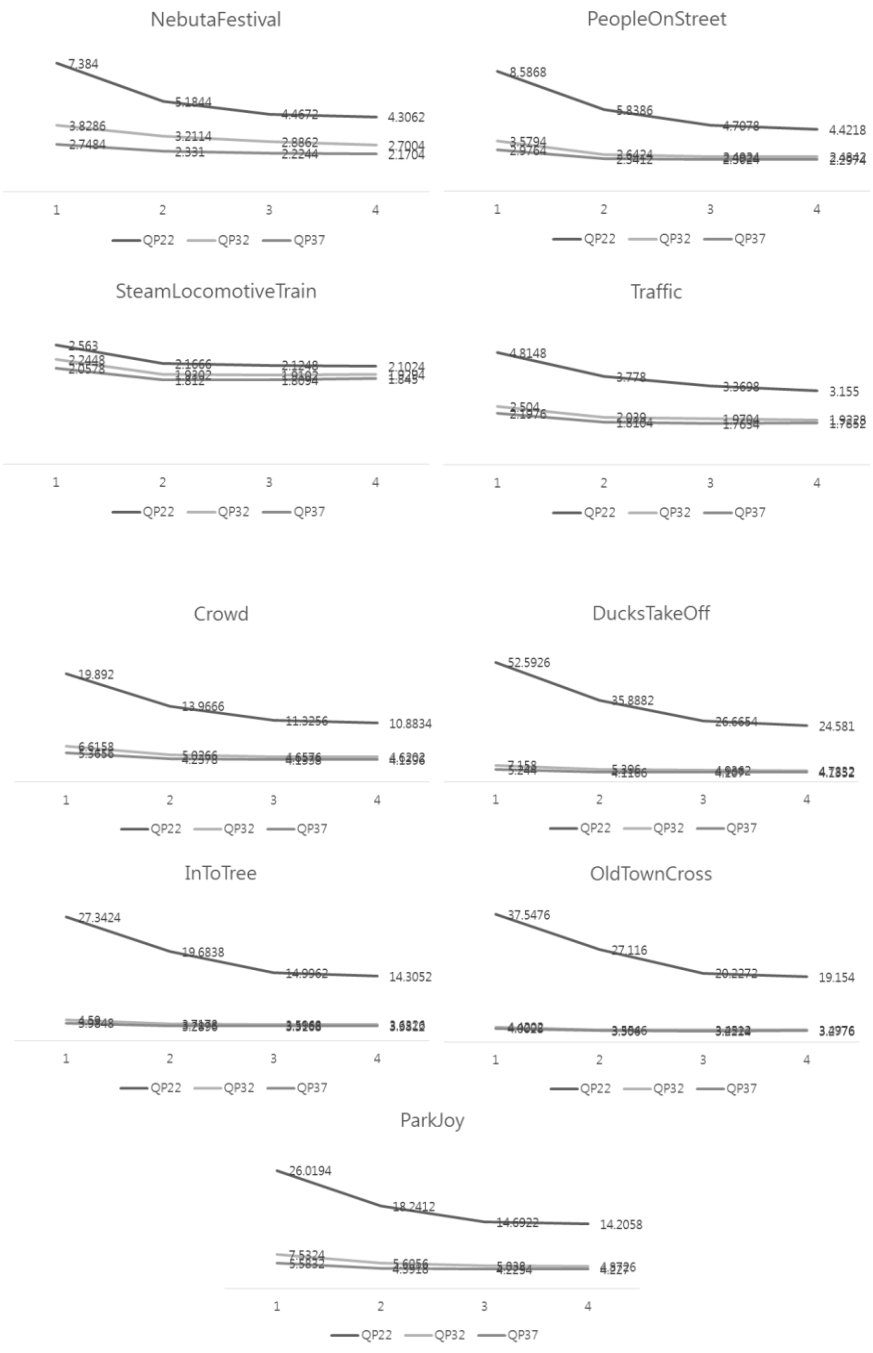


그림 16 Parse 태스크 수 변화에 따른 각 예제들의 수행 시간 변화

6.2 매핑 변경에 따른 성능 변화

실험 환경에선 Xeon E5 CPU 를 2 개 사용하였다. 메모리 모델은 공유메모리로 CPU 간의 명시적인 통신을 고려할 필요가 없으나, 두 CPU 가 서로 다른 캐시를 사용하며, Non Uniform Memory Access (NUMA)의 방식으로 공유메모리를 지원하기 때문에 태스크들을 어느 CPU 에 매핑 하는지에 따라 성능의 차이가 나타난다. 제안하는 기법은 태스크 그래프 기반의 병렬화 기법으로 태스크들의 매핑 및 스케쥴링이 자유롭다는 장점을 가지며, 본 실험 환경과 같이 매핑 및 스케쥴링에 따라 성능이 민감하게 변경될 수 있는 경우 쉽게 설계 공간 탐색을 통해 성능의 변화를 확인할 수 있다. 이번 실험은 태스크들의 매핑을 변경해보면서 성능의 변화를 살펴본다.

실험에는 4 가지의 매핑 설정을 사용하였다. 첫 번째 설정은 각 쓰레드가 자유롭게 CPU 에 매핑될 수 있는 방식이다. 이 설정에서는 태스크가 실행 가능할 때 OS 에 의해 사용 가능한 CPU 에 자유롭게 옮겨 다니며 배치된다. 두 번째 설정은 CPU1 에는 Read, Parse, Decode 태스크들을 배치하고 CPU2 에는 DF, SAO 태스크들을 배치한 방식이다. 세 번째 설정은 CPU1 에는 Read, DF, SAO 태스크들을 배치하고 CPU2 에는 Parse, Decode 를 배치한 방식이다. 마지막으로 네 번째 설정은 CPU1 에는 Read, Parse, DF, SAO 태스크들을 배치하고 CPU2 에는 Decode 태스크만을 배치한 방식이다. 모든 설정에서 Parse 태스크는 4 개를 사용하였으며, Decode, DF, SAO 는 각각 34 개의 Thread 를 사용하여 병렬화하였다. 각 설정들은 표 2 에 정리되어있다.

설정 1	특정 CPU 에 매핑되지 않음. 자유롭게 이동
설정 2	CPU1(Read, Parse, Decode), CPU2(DF, SAO)
설정 3	CPU1(Read, DF, SAO), CPU2(Parse, Decode)
설정 4	CPU1(Read, Parse, DF, SAO), CPU2(Decode)

표 2 실험에 사용된 매핑 설정

이 네 가지 설정들에 대해 각 예제들에 실험한 결과는 그림 17 과 같다. 여기서는 QP32 로 인코딩된 예제들을 사용하였다.

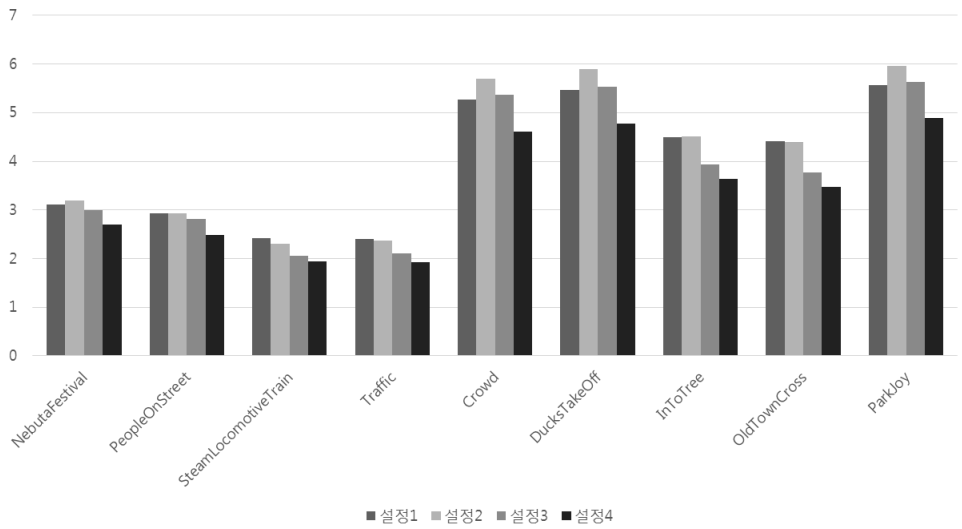


그림 17 매핑 변경에 따른 각 예제들의 수행 시간 변화

성능은 설정 4 가 모든 예제에서 가장 좋게 나타났으며, 그 다음으로 설정 3 이 좋은 성능을 보였다. 수행 시간의 비율로 보았을 때 Decode 태스크가 전체 수행 시간의 50%이상을 차지하기 때문에, Decode 태스크가 CPU 하나를 온전히 사용할 수 있게 배치한 설정 4 가 가장 높은 성능을 보인 것으로 볼 수 있다. Decode 가 Parse 와 같이 배치된 설정 2 와 3 의 경우 다른 태스크들에 의해 가장 비중이 높아 빨리 처리되어야 할 Decode 태스크가 지연되어 성능에 영향을 준 것으로 보인다. 여기서 또한 짚고 넘어가야 할 부분은 설정 1 의 성능이다. 다른 설정들이 일관된 수행 성능 순위를 가지는 것에 비해, 설정 1 은 예제에 따라 가장 나쁘기도 하고 아니기도 하다. 이는 설정 1 에서 태스크들이 자유롭게 CPU 들을 이동하며 배치되기 때문이다. 태스크들은 의존 관계에 따라 앞서 수행한 태스크의 결과물을 참조해야 한다. NUMA 방식에선 이런 선행 태스크와 후행 태스크가 서로 다른 CPU 에 매핑되어 있는 경우 평소보다 긴 접근시간을 가지고 수행하게 된다. 매핑이 고정되어 있지 않은 설정 1 과 같은 경우는 동적으로 매핑이 변경되면서 선행 태스크와 후행 태스크가 서로 다른 CPU 에 나타나게 될 수 있다. 그러므로 성능을 위해 수행시간을 고려한 정확한 매핑이 강조된다.

6.3 SIMD 병렬화에 의한 성능 향상

이번 장에서는 제안한 SIMD 병렬화에 의한 성능의 향상을 살펴본다. SIMD 병렬화만의 성능 향상을 확인하기 위해, 아무런 병렬화가 적용되지 않은 설정과 SIMD 병렬화만을 적용한 설정 간의 성능 차이를 측정한다. 각 예제 별로 측정한 수행 시간 및 SIMD 병렬화에 의한 성능 향상은 표 3 에 정리되어 있다.

Sequence	QP	Base	SIMD	SpeedUp
NebutaFestival	QP22	56.345	28.781	1.958
	QP32	40.923	19.152	2.137
	QP37	33.909	14.995	2.261
PeopleOnStreet	QP22	56.587	34.607	1.635
	QP32	36.224	19.965	1.814
	QP37	34.129	17.911	1.905
SteamLocomotiveTrain	QP22	30.364	14.295	2.124
	QP32	27.979	12.762	2.192
	QP37	26.377	11.965	2.204
Traffic	QP22	39.722	21.250	1.869
	QP32	28.347	13.427	2.111
	QP37	25.885	11.923	2.171
Crowd	QP22	121.837	72.957	1.670
	QP32	77.375	37.767	2.049
	QP37	69.439	32.275	2.151
DucksTakeOff	QP22	191.243	131.605	1.453
	QP32	83.358	39.079	2.133
	QP37	70.510	31.678	2.226
InToTree	QP22	128.209	75.760	1.692
	QP32	57.923	25.298	2.290
	QP37	52.623	22.427	2.346

OldTownCross	QP22	160.239	100.530	1.594
	QP32	52.205	23.162	2.254
	QP37	47.683	20.887	2.283
ParkJoy	QP22	137.194	80.688	1.700
	QP32	79.781	38.613	2.066
	QP37	69.383	31.959	2.171

표 3 각 예제 별 SIMD 병렬화에 의한 성능 변화

측정 결과 SIMD 병렬화 적용 시, 각 예제에서 최소 1.45 배, 최대 2.35 배 성능 향상을 확인할 수 있으며 평균 2.02 배 성능 향상이 나타나는 것을 알 수 있다. 예제 및 QP 설정에 따라 성능 배율이 차이가 나는 것은 SIMD 가 적용되는 부분들이 각 예제 및 설정에 따라 전체에서 차지하는 비율이 달라지기 때문이다. 성능이 낮게 나타난 부분은 SIMD 가 적용되는 부분이 전체에서 차지하는 비중이 낮으며, 반대로 높게 나타난 부분은 SIMD 병렬화의 비중이 높다고 볼 수 있다. 앞서 Parse 때와 같은 문맥으로 QP 가 낮게 인코딩된 경우는 디코딩 알고리즘보다 순차 수행인 엔트로피 디코딩의 비중이 높기 때문에, QP 가 낮을수록 SIMD 로 인한 성능 향상은 낮게 나타난다.

5.4 제안하는 기법에 의한 성능 향상

마지막으로 쓰레드 수를 증가시켜 가면서 성능의 변화를 확인하고, 제안한 기법을 모두 적용했을 때 얻을 수 있는 성능 향상을 확인한다. 각 설정에서 공통적으로 Read, Parse, Write 는 태스크 별로 각자 하나의 쓰레드를 가지며, Parse 태스크는 4 개를 사용한다. Decode, DF, SAO 는 쓰레드의 수를 2 에서 32 로 증가시키면서 성능의 변화를 확인한다. 이 때 DF 와 SAO 는 하나로 묶여 배치된다. 예를 들어 16 개의 쓰레드 설정인 경우, Decode 가 수행될 쓰레드가 16 개, DF+SAO 가 수행될 쓰레드가 16 개이다. 16 개의 쓰레드에는 각 LCU 라인이 하나씩 분배되어 배치되도록 설정하였다. 모든 설정에는 SIMD 병렬화가 적용되어 있다. 실험 결과는 그림 18 에 그래프로 정리되어 있다.

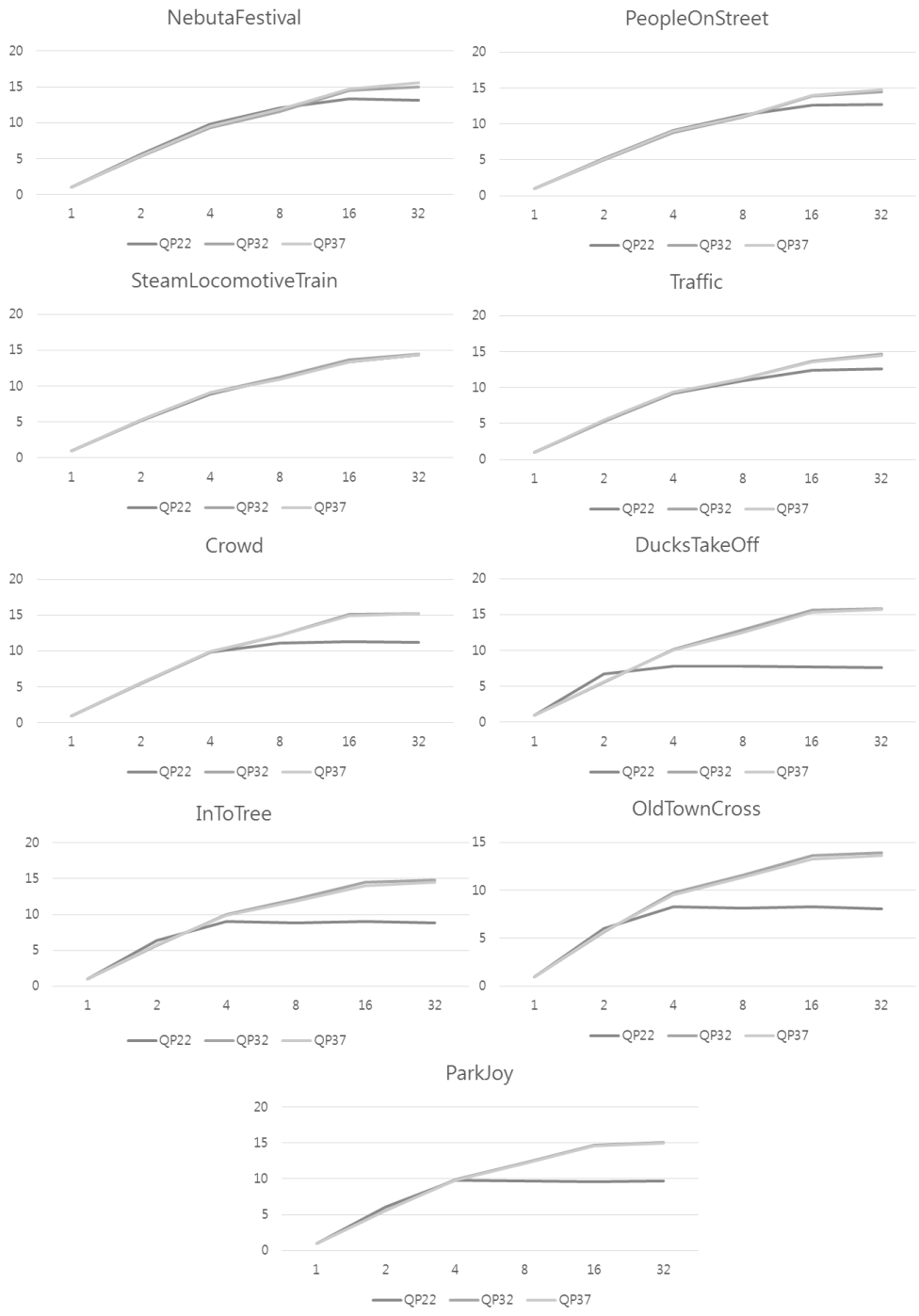


그림 18 쓰레드 수 변화에 따른 각 예제의 속도 향상 변화

성능을 보면 제안하는 기법을 적용했을 때 2560x1600 예제에서 최대 15.59 배의 성능을, 3840x2160 예제에서 최대 15.85 배의 성능을 얻었음을 확인할 수 있다. 최대 성능은 쓰레드의 개수가 보통 16 개를 넘은 이후부터 얻을 수 있었으며, 이후에도 그래프의 형태로 볼 때 계속적으로 병렬화가 된다면 성능이 증가할 수 있을 것이라고 예측된다. 그러나 4K 해상도에서 LCU 라인의 수가 34 이기 때문에 최대로 얻을 수 있는 병렬에 한계점에 도달하였으므로, CPU 자원을 더 활용할 수 있을 것으로 보이나 병렬화가 더 이상은 불가능하여 성능을 더 증가시킬 수 없게 된다. 매니 코어를 염두에 둔다면 더 세부적인 병렬화 기법이 필요할 것이라 생각된다. QP 에 따른 차이를 보면, QP 가 낮을수록 성능의 한계점에 수렴이 더 빨리 되는 것을 볼 수 있다. QP 가 낮을수록 엔트로피 디코딩에 의한 병목 현상이 심하기 때문으로 보인다. 병렬화를 아무리 해도 순차 수행에 의한 병목이 발생하여 성능이 증가하지 않게 된다.

표 4 와 표 5 는 각각 각 예제 별로 얻은 속도 향상 최고 수치와 최대 FPS 수치를 표시한 것이다. 이는 우리와 가장 비슷한 수준으로 병렬화의 자유도를 가지고 있는 [6]에서의 성능 10.7 배와 비교했을 때 매우 높은 속도 향상 수치이다. [6]과 비교해서 더 높은 성능 향상을 가져올 수 있는 이유는 첫째로 SIMD 기반 병렬화 기법이 적용되었기 때문이고 둘째로 Parse 에 의한 병목 현상을 제거하여 병렬화의 성능을 최대로 끌어냈기 때문이다. 속도 향상 수치가 QP22 로 인코딩된 4K 영상들에 대해선 낮은 수치를 보이는데, 이는 QP22 의 경우 엔트로피 디코딩에 의해 병목 현상이 나타나는 걸로 추측할 수 있다. 이 경우 더 많은 Parse 태스크를 사용함으로써 해결할 수 있으나, 저압축률의 고용량 영상 서비스가 일반적으로 이루어지지는 않는 편이기 때문에 본 논문에서는 이 이상의 성능 향상을 시도하지 않았다.

영상 별 FPS 수치를 보면 2560x1600 예제들은 60 FPS 를 상회하는 높은 디코딩 성능을 보였으며, 3840x2160 4K 예제들은 실시간 재생 기준인 30 FPS 에 근접한 성능을 보였다. 이를 통해 Xeon E5 CPU 를 2 개 사용하는 경우 4K 영상의 실시간 재생이 가능할 것으로 예측된다.

Sequence	QP			Max
	QP22	QP32	QP37	
NebutaFestival	13.403	15.024	15.589	15.589
PeopleOnStreet	12.706	14.509	14.804	14.804
SteamLocomotiveTrain	14.327	14.483	14.335	14.483
Traffic	12.606	14.715	14.430	14.715
Crowd	11.283	15.211	15.231	15.231
DucksTakeOff	7.808	15.849	15.704	15.849
InToTree	9.041	14.775	14.457	14.775
OldTownCross	8.303	13.958	13.639	13.958
ParkJoy	9.755	15.052	14.982	15.052

표 4 각 예제 별 얻은 순차 수행 대비 속도 향상 최고치

Sequence	QP			Max
	QP22	QP32	QP37	
NebutaFestival	35.682	55.070	68.959	68.959
PeopleOnStreet	33.682	60.082	65.065	65.065
SteamLocomotiveTrain	70.775	77.648	81.522	81.522
Traffic	47.604	77.865	83.621	83.621
Crowd	13.891	29.488	32.902	32.902
DucksTakeOff	6.124	28.519	33.409	33.409
InToTree	10.578	38.261	41.209	41.209
OldTownCross	7.772	40.105	42.904	42.904
ParkJoy	10.665	28.300	32.390	32.390

표 5 각 예제 별 얻은 최대 FPS

제 7 장 결론

본 논문에서는 고해상도 영상 처리를 위한 차세대 동영상 코덱인 HEVC 의 디코더 알고리즘을 멀티코어 시스템을 최대한으로 활용하여 시간당 처리량을 최대화 시킬 수 있는 병렬 처리 기법을 제안하였다. 또한 추가적으로 SIMD 명령어를 이용한 병렬 처리 기법 또한 제안하여 추가적인 성능 향상을 꾀하였다.

제안한 병렬 처리 기법은 실험적인 성능 분석과 알고리즘 분석에 기반하였다. HEVC 디코더 알고리즘을 태스크 단위로 분리하여 분석하고, 각 태스크 내부의 의존관계성과 태스크 간의 의존관계성에 대해 분석을 하여 태스크 그래프 모델을 완성시켰다. 또한 성능 병목 현상이 나타날 수 있는 병렬화 불가능한 태스크들을 고려하여 병목을 제거할 수 있는 방법도 태스크 그래프 모델 안에 표현하였다.

제안한 병렬 처리 기법의 우수성을 검증하기 위해, Hexa Core 가 2 개 장착된 실험 환경에서의 성능 측정 실험을 진행하였다. 쓰레드의 수를 증가시키며 성능 변화를 관찰한 결과, 코어의 수보다 더 많은 쓰레드를 사용했음에도 지속적으로 성능이 증가하는 추세를 보였다. 병목 현상을 제거하기 위한 다중 Parse 태스크 배치 기법의 성능과 필요성 또한 실험을 통해 확인되었다. 최종적으로 태스크 그래프 모델 기반 병렬 처리 기법과 SIMD 기반 병렬 처리 기법을 모두 적용했을 때 2560x1600 예제에서 최대 15.59 배 속도 증가, 3840x2160 예제에서 최대 15.85 배 속도 증가를 얻을 수 있었다. 또한 2560x1600 예제는 60FPS 의 서비스가 가능하며, 4K 도 인코딩 수준에 따라 30FPS 서비스가 가능할 것으로 예측되었다. Hexa Core CPU 2 개를 사용하는 환경이 일반적으로 보급된 환경은 아니나, 매니코어 추세가 일반화되면 4K 의 영상의 대중화도 그리 먼 일은 아닐 것으로 보인다.

참고문헌

[1] Joint Collaborative Team on Video Coding (JCT-VC), http://www.itu.int/en/ITU-T/studygroups/com16/video/Pages/jctvc.aspx

[2] JCTVC-L1003_v30: “High Efficiency Video Coding (HEVC) text specification draft 10 (for FDIS & Consent)”.

[3] 조현호, 유은경, 남정학, 심동규, 김두현, 송준호, "HEVC 복호화기를 위한 효과적인 병렬화 설계," 2012년도 대한전자공학회 하계학술대회, 제 35권 1호, 2012년 6월 28일.

[4] M. Alvarez-Mesa, C. C. Chi, B. Juurlink, V. George, and T. Schierl, "Parallel Video Decoding In The Emerging HEVC Standard," Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2012), Kyoto, Japan, March 2012.

[5] A. Fuldseth, M. Horowitz, S. Xu, K. Misra, A. Segall, and M. Zhou, “Tiles for Managing Computational Complexity of Video Encoding and Decoding,” in Proc. PCS’ 12, 2012

[6] C.C. Chi, M.A. Mesa, B. Juurlink, G. Clare, F. Henry, S.Pateux, T. Schierl, “Parallel Scalability and Efficiency of HEVC Parallelization Approaches”, IEEE Transactions on Circuits and Systems for Video Technology, IEEE TCSVT, Dec 2012.

[7] L. Yan, Y. Duan, J. Sun, Z. Guo “Implementation of HEVC decoder on x86 processors with SIMD optimization”. 2012 IEEE Visual Communications and Image Processing (VCIP), pp. 1-6, Nov. 2012.

[8] 한청희, 심동규, “CUDA 기반 HEVC 복호화기를 위한 디블로킹 필터의 병렬화 연구,” 광운대학교, 2013년 12월

[9] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, “Overview of the High Efficiency Video Coding (HEVC) standard,” IEEE Trans. Circuits Syst. Video Technol., vol. 22, no. 12, pp. 1648-1667, Dec.

2012.
[10] J-R Ohm, G. J. Sullivan, “High Efficiency Video Coding: The Next Frontier in Video Compression”. IEEE Signal Processing Magazine, pp.152-158, Jan. 2013.
[11] P.Bordes, G.Clare, F.Henry, M. Raulet, J. Vieron,“An overview of the emerging HEVC standard”, Proceedings of ISIVC, July 2012.
[12] Xiph.org Video Test Media, “http://media.xiph.org/video/derf/”

Abstract

An Efficient Parallelization Technique of HEVC Decoder for High Resolution Video Processing

Junchul Choi
School of Computer Science Engineering
College of Engineering
The Graduate School
Seoul National University

Next generation TV have been developed according to the advances of display technology. It is expected to have high resolution above 4K and to support additional functions such as 3D display and multi-view for enriching user experience. Efficient codec have been also developed in order to process high resolution videos faster than existing codecs, and High Efficiency Video Coding (HEVC) is the most representative one. HEVC is considered as the most appropriate video codec for next generation TV since HEVC provides more efficient video compression and higher video quality compared to existing codecs. However, HEVC reference software cannot fully utilize multi-core system resources, which is the trend of recent SoC design, because it is implemented to be executed sequentially on single-core. In this paper, we propose parallelization technique of HEVC decoder based on task graph model. Our technique aims for providing flexible software model to support design space exploration for the design of next generation TV SoC, and to increase throughput performance by parallel processing

on multi-core SoC platform. In addition to task graph based technique, we propose Single Instruction Multi Data (SIMD) based optimization. Performance increase by our technique is verified with experimental results. We achieves speed-up of 15.59 for 2560x1600 sequences and 15.85 for 3840x2160 sequences, compared to base reference software.

keywords : 4K resolution, HEVC, multi-core, parallel processing, design space exploration

student number : 2011-23385