HEVC 고속 복호화를 위한 SIMD 기반의 IDCT 병렬 프로그래밍 기법

*홍승보, **최기호, **박상효, **장의선‡
*한양대학교 컴퓨터공학부, **한양대학교 디지털 미디어 연구실
** esjang@hanyang.ac.kr

Parallelization mathod of IDCT with SIMD for fast HEVC decoding

*Seungbo Hong, **Kiho Choi, **Sang-hyo Park, **Euee Seon Jang
*Division of Computer Science and Engineering, Hanyang University
**Digital Media Laboratory, Hanyang University

요 약

최근 방송, 의료, 우주산업, 게임, UCC, 핸드폰 등 여러 사업 분야에 걸쳐 실제에 근접한 영상을 요구하고 있고 이것은 3D와 Ultra High Definition (UHD) 영상의 출현으로 현실화 되고 있다. UHD 급에 걸맞는 압축률을 위해 Joint Collaborative Team on Video Coding (JCT-VC) 에서는 MPEG-4 Part 10 AVC/H.264를 뒤이을 차세대 코덱으로 High Efficiency Video Coding (HEVC) 를 개발을 시작했다. HEVC는 기존 MPEG-4 Part 10 AVC/H.264코덱과 비교해 40%이상의 압축률을 나타내지만 복잡도 역시 상승했다. 특히 복호화기에서 복잡도는 중요한 요소이며, 역 코사인변환 (Inverse Discrete Cosine Transform, IDCT) 은 전체 복호화시간의 8% ~ 16%를 차지하는 알고리즘이다. 본 논문에서는 IDCT 의 수행시간을 줄이기 위해 병렬프로그래밍 중의 하나인 SIMD명령어를 사용하여 효율적으로 병렬화 프로그래밍을 하는 기법들을 제안한다. 본 제안 기법은 IDCT 수행시간을 평균 59% 단축하는 결과를 보였다.

1. 서론

2010년 1월부터 Joint Collaborative Team on Video Coding (JCT-VC) 에서는 차세대 코덱 High Efficiency Video Coding (HEVC) 의 표준화 작업을 시작했고, 3년이 지난 2013년 1월 HEVC 코덱 국제표준이 승인되었다. HEVC는 기존 MPEG-4 Part 10 AVC/H.264 코덱보다 약 40% 이상의 높은 압축 성능을 보인다[1].

기존 MPEG-4 Part 10 AVC/H.264 코덱에서 transform 의 기본 블록단위는 16x16 의 크기였으나, HEVC에서는 고화질 영상을 처리할 때 효율을 높이기 위해 변환을 위한 기본 블록의 단위가 32x32 ~ 4x4로 크기가 커지고 다양해졌다. 그러나 다양해진 블록단위만큼 역변환의 복잡도 역시 커졌다.

※ This work was supported by the strategic technology development program of MSIP. [KI001798, Development of Full 3D Reconstruction Technology for Broadcasting Communication Fusion]

역변환 과정에서 속도를 높이기 위한 방법 중 하나가 병렬 프로그래밍이다. 병렬프로그래밍 모델에는 하나의 명령어로 여러 개의 데이터를 처리할 수 있는 SIMD, 멀티스레드로 데이터를 처리하는 공유메모리 병렬프로그래밍, 여러 개의 PC메시지로 통신하여 대용량 메시지를 처리하는 메시지 패싱 프로그래밍 등이 있다. 특히 SIMD 명령어를 이용한 병렬 프로그래밍은 CPU의 영향을 가장 적게 받고, 싱글코어에서도 빠르게 동작해 간편하게 사용할 수 있다. 이론적으로는 각 pack 의 개수의 배 만큼 속도향상이 이루어지지만 실제로 레지스터의 선언과 해제, CPU와 메모리영역사이 데이터를 주고받는 오버헤드가 발생해 32bit int 형의 경우 10~30% 의 성능향상이 이루어진다[2].

IDCT의 커널함수인 patialButterflyInverse(32~4)의 수행시간은 전체 복호화 시간 중 All Intra Configuration (AI) 에서 15.9%, Random Access (RA) 에서 7.6%의 시간이 소요되는 함수이다. 특히 AI의 경우 각 함수 중 전체 복호화 시간에서 차지하는 비율이 가장 높다. 이 수치는 2번째로 비율이 높은 TComLoopFilter가 12.9% 인 것과 비교했을 때도

3%나 높은 수치이다[3]. 따라서 patialButterflyInverse 함수를 병렬화 한다면 전체 복호화 시간에도 상당한 영향을 줄 것이라 판단된다. 본 서에서는 SIMD 명령어를 이용하여 병렬화를 하기위해 문제가 되는 부분을 제시하고 이를 개선하고 적용시키기 위한 몇 가지 방법을 제안한다.

2. patialButterflyInverse 함수에서의 SIMD 명령어 적용시 문제점

2.1. 계산을 위해 미리 계산된 배열을 입력받는 문제

HM10.0 의 경우 계산의 편이성을 위해 g_aiT 배열에 미리 계산된 값들을 넣어놓고 함수가 진행되는 과정에서 이 값을 바로 꺼내어 쓸 수 있다. 이 배열은 16bit short형으로 구성된 2차원 배열이다. 함수의 계산과정에는 변환을 위한 입력 블록 (src, 32x32 ~ 4x4) 의 각 line별로 같은 값의 g_aiT 값이 곱해지므로 SIMD 를 이용할 경우 g_aiT 배열을 차례로 하나의 레지스터에 받아오기 때문에 그림1과 같이 하나의 레지스터에 서로 다른 값이 입력되게 된다. 이것을 차례로 받아온 각 line 의 src 와 pmulld 명령어를 이용하여 곱셈을 진행할 경우 계산 값이 다르게 나오는 문제가 생긴다.

source line 1	x4	x3	x2	x1
pmulld	x	x	x	x
g_aiT	y4	y3	y2	y1
	↓	↓	↓	↓
src * g_aiT	x4 * y4	x3 * y3	x2 * y2	x1 * y1

그림1. 문제점을 해결하지 않았을 때의 계산과정

2.2. 형태 확장 문제

IDCT 를 위한 src 는 16bit short 형으로 주어진다. 이 src 는 계산과정에서 32bit int 형으로 확장되어 배열에 입력된다. SIMD 계산을 위한 레지스터는 서로 같은 형태 간의 연산만 지원되고 그 결과 또한 같은 형태 이어야한다. 따라서 short 간 곱셈의 결과는 short 형이고 이때 이 값은 자동적으로 int 형으로 확장되지 않는다. 만약 서로 다른 형태 간의 연산이 가능하더라도 그 결과 값이 다른 형태로 확장 될 경우 쓰레기 값이 레지스터에 들어가게 된다.

2.3. 변수의 산술적 shift연산 문제

어셈블리언어에서는 산술적 오른쪽 shift 연산으로 shr 연산자를 정의하는데 shr reg, imm8 (레지스터, 상수) 혹은 shr mem, imm8 (메모리변수, 상수) 의 shift연산만을 제공한다[4]. 따라서 shr reg, mem (레지스터, 메모리변수) 와 같이 메모리 변수를 이용한 산술적 shift 연산은 불가능 하다. int shift 라는 변수가 만약 정형적이라면 미리 그 숫자를 알기 때문에 상관이 없지만 이 변수가 가변적이므로 우리는 shift 연산에서 문제를 겪게 된다.

2.4. 값 절삭의 병렬화 문제

HM10.0 에는 Clip3 메크로 함수가 정의되어 있다. 이 함수는 데이터 값을 short 형 범위인 -32768 ~ 32767로 절삭하는 역할을 담당한다. 함수의 최종적 결과 블록 (dst) 이 short 형이므로 값을 잘라주는 것이다. 이 함수는 각 함수가 진행되는 과정에서 오버헤드가 가장 크다.

2.5. 병렬계산 데이터를 직렬로 대입하는 문제

앞선 과정을 거쳐 계산이 완벽하게 된다고 해도 이 레지스터를 바로 dst 로 내보낼 수 없다. 그 이유는 첫째로 이때까지 병렬프로그래밍을 통하여 값이 계산되었기 때문에 레지스터에는 병렬로 값들이 저장 되어있는데, 우리가 내보내야 할 dst 는 src 와 마찬가지로 직렬로 저장되어야 하기 때문이고, 두 번째로 int형으로 계산된 값을 short형으로 저장해야 하기 때문이다.

3. SIMD명령어를 이용한 IDCT의 병렬프로그래밍 구현 제안방법

3.1. 새로운 배열의 선언

2.1 에서 지적한 첫 번째 문제는 g_aiT 배열이 미리 계산된 값이라는 점을 이용한다. 미리 계산된 값이기 때문에 각 배열의 값을 하나의 레지스터 속 pack 의 개수만큼 값으로 받는 새로운 배열을 선언하면 해결된다. 그리고 이 문제는 2.2 문제와도 연관된다. short 형 값이 계산을 통해 int 형으로 확장되기 때문에 처음 g_aiT 배열을 새로 선언할 때부터 미리 int 형으로 선언한다. int 형은 32bit 로 128bit 인 레지스터에 4개 pack 이 담기므로 새로운 배열을 선언할 때 4개씩 같은 값이 되도록 선언한다면 이와 관련된 문제들이 해결된다.

3.2. 비교연산을 이용한 데이터 형태 확장

2.2 에서 야기될 수 있는 쓰레기 값 입력 문제는 입력받은 src 를 미리 int 형으로 확장 하여 해결한다. Unpacking 조합 명령어인 punpcklwd (하위 4비트 조합) 와 punpckhwd (상위 4비트 조합) 를 이용하여 두 레지스터를 조합할 수 있는데 양의 정수인 경우 0으로 세팅된 16bit의 값 0x0000을 원래의 16bit 값 앞에 붙여주면 32bit로 확장이 가능하고, 음의정수인 경우 그 특성에 따라 1로 세팅된 16bit의 값 0xffff을 원래 16bit값 앞에 붙여주면 된다. 하지만 src는 음, 양의 정수의 값이 섞여 들어있으므로 이 과정을 위해서는 각 pack의 개별 연산이 이루어져야 한다. 확장과정에서 발생한 새로운 문제는 pcmpgtw 를 이용한 비교연산으로 해결할 수 있다. 그림2와 같은 방법으로 src 와 0으로 세팅된 새로운 배열을 비교해서 음의 정수인 경우 해당 위치에 있는 pack 에 0xffff를 양의 정수인 경우 0x0000을 결과 값으로 담는 레지스터를 생성하고 이 레지스터와 src 를 조합하여 int 형으로 확장한다.

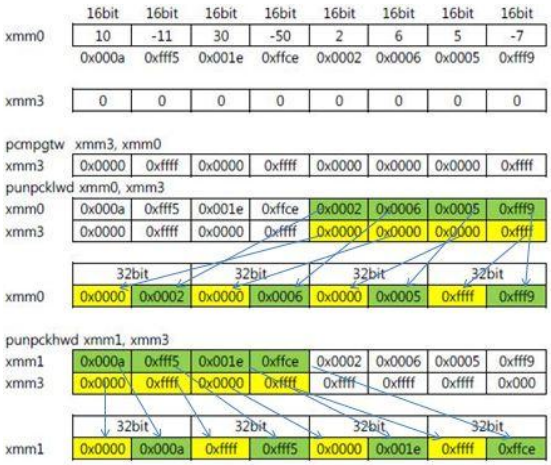

그림2. 레지스터의 형태 확장 과정

3.3. Intrinsic함수를 이용한 shift연산

2.3 의 문제는 SIMD Intrinsic 함수를 사용 하므로 해결할 수 있다. Intrinsic 함수는 제한 없는 레지스터 개수, 가독성, 프로그래밍 편리성, 메모리 변수 사용, 디버깅 환경 개선 등을 제공하기 위해 Advanced Reduced Instruction Set Computing Machine (ARM) 에서 제공하는 함수 이다 [5]. Intrinsic함수를 사용하게 되면 메모리 변수 shift 가 가능 하고 for 문을 사용한 루프연산을 할 수 있을 뿐 아니라 레지스터연산이 가능하므로 C 언어의 편이성과 SIMD 명령어의 병렬연산이 모두 가능하다. 하지만 Intrinsic 함수에 변수가 늘어나게 되면 연산속도가 느려지는 단점이 있으므로 본 논문에서는 Intrinsic함수의 사용을 최대한 자제하였다.

3.4. 비교연산과 비트연산을 이용한 절삭함수

2.4 에서 문제점으로 언급한 pack 끼리의 절삭연산을 위해서 3.2 의 방법처럼 미리 최대, 최소의 값으로 세팅된 레지스터를 이용하여 비교연산과 비트연산을 진행하여 절삭함수를 구현할 수 있는데. 데이터가 최댓값을 초과할 경우와 최솟값보다 작은 경우로 나눠 과정을 진행한다. 각 과정은 아래의 그림3, 그림4와 같이 진행된다.

가. 32767을 초과하는 값을 32767로 절삭

먼저 MAX (32767) 로 세팅된 레지스터를 선언해준다. MAX 와 데이터 값이 담긴 레지스터 (dst0) 를 대소 비교하여 MAX 보다 큰 값이 dst0 에 있었다면 해당위치의 pack 은 0xffffffff 으로, 범위를 넘지 않는 값에 대해서는 그 위치에 있는 pack 값이 0x00000000 으로 레지스터가 채워지게 된다. 이 레지스터를 MAX 와 AND 연산 한다면 그 pack 에는 32767이 담기고 나머지 pack 은 0으로 채워지며 . dst0 레지스터와 ANDNOT 연산을 수행하게 되면 32767을 넘지 않는 값들은 원래의 값을 갖고 범위를 넘는 값은 0이 된다. 계산된 두 개의 레지스터를 덧셈 명령어인 paddd 를 이용하여 더하고 과정을 마무리한다. 이 과정을 통해 32767이 넘는 값들은 32767로 절삭되게 한다.

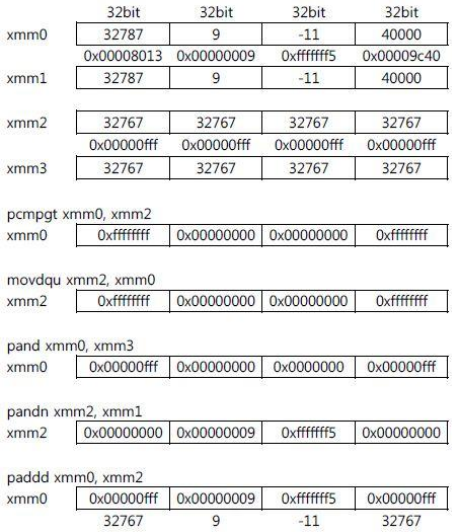

그림3. 범위를 초과하는 값을 절삭하는 과정

나. -32768미만의 값을 -32768로 절삭

이 과정은 가 에서 설명한 과정과 흡사하며 대소 비교과정을 MAX 레지스터 대신 MIN (-32768) 레지스터를 사용하여 앞선 과정에서의 비교연산 (pcmpgt xmm0, xmm2) 과는 반대로 바꿔 연산이 진행되며 (pcmpgt xmm2, xmm0) 이 결과 MIN 보다 작은 값에 대해 0xffffffff 을 pack 에 채우게 된다.

3.5.1 Unpack 조합연산을 이용한 정렬

병렬로 저장된 하나의 레지스터 내에서 pack 간 주소의 간격은 애초에 우리가 원하는 레지스터와 다음 레지스터에서 같은 위치에 있는 pack 간 주소의 간격과 같다. 다른 시점에서 보면 병렬로 저장된 레지스터와 다음레지스터의 pack 간 주소의 간격은 원래 우리가 원하는 레지스터 내 pack 간 주소의 간격과 같다. 이것을 이용하여 각 레지스터 내 pack 간 주소를 그림4와 같이 반 씩 줄여나가는 방법으로 병렬로 저장되어있는 데이터를 직렬로 저장되게 바꿔준다. 이 과정에서 3.2 에서 사용되었던 조합연산을 사용한다. 이 과정은 pack 간 주소의 차이가 1이 날 때 까지 수행하면 된다.

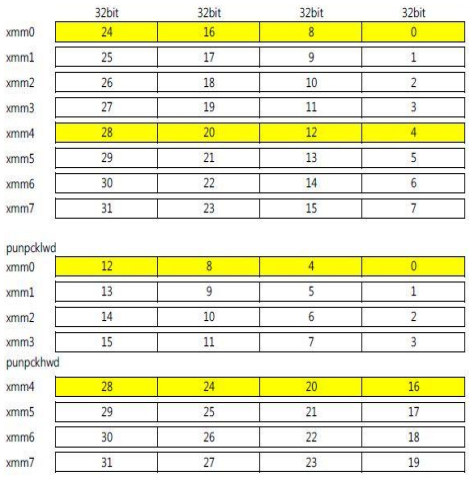

그림4. pack의 조합 과정 1

3.5.2. Packing 조합을 통해 결과블록생성

각 과정을 거쳐 각 레지스터의 구조는 원하는 결과블록과 유사한 구조가 되었지만 마지막으로 형태를 바꿔주는 문제가 남아있다. 이 문제는 Packing 조합 명령어인 packssdw 를 사용 하여 32bit int 형으로 구성된 레지스터 2개를 16bit short 형으로 구성된 레지스터 1개로 합쳐주면 된다. 이 때, int 형으로 구성된 레지스터가 short 형 범위를 넘어 서면 원하지 않는 값이 들어가지만 앞서 값의 절삭을 거쳤기 때문에 이 문제는 해결 된다. Packing 조합 과정에서 음의정수가 short 형 범위 내에 존재한다면 문제없이 Packing 조합 이 가능하다.

4. 실험 및 고찰

본 논문에서 제안한 방법으로 HM 10.0 버전에서 병렬 프로그래밍으로 변경한 paitalButterflyInverse 함수의 병렬화를 통한 역 코사인 변환의 시간복잡도의 감소를 확인하기 위해, 역 코사인 변환 함수 (xIT)를 CPU의 수행 시간으로 비교실험을 하였다. 실험을 위한 환경은 표1, 실험의 결과는 표2와 같다. 실험을 위해서 사용된 Bit stream은 Class C (832x480) 4개의 영상을 미리 HM 10.0에 의해 부호화된 결과를 이용하였다[6].

표1. 실험 환경 및 시간측정방법

CPU	Intel Core(TM) i3, 2.53GHz
OS	windows7 professional k 64bit
Memory	DDR3 4GB
구현대상	HM10.0
테스트 영상	Class C (WVGA) : BasketballDrill, BQMall, PartyScene, RaceHorses
프레임 수	BasketballDrill (500), BQMall (600), PartyScene (500), RaceHorses(300)
양자화 계수	22, 27, 32, 37
부호화 환경	Low-delay configuration

본 서에서 제안한 방법으로 병렬화한 역 코사인 변환함수에서 최고 평균 64.65 % (BasketballDrill), 최저 평균 55.53 % (PartyScene) 의 시간 단축을 나타냈다.

표2. xIT 함수 시간 단축

	양자화 계수별 xIT 함수 시간 단축 (%)				
	양자화 계수				평균
테스트 영상	22	27	32	37	
BasketballDrill	-60.54	-65.52	-62.62	-69.90	-64.65
BQMall	-56.10	-48.01	-58.37	-60.09	-55.64
PartyScene	-47.78	-57.51	-57.01	-59.80	-55.53
RaceHorses	-60.06	-55.81	-56.56	-61.56	-58.50
평균	-56.98	-58.05	-58.99	-60.84	-58.72

앞서 서론에서 밝힌 것처럼 이론상 10 ~ 30% 의 시간감소를 해야 하는데 결과를 보면 평균 58.72% 의 높은 시간 단축의 결과를 보였다. 따라서 기존의 함수에서 발생하는 오버헤드가 병렬화한 함수에서 발생하는 오버헤드에 비해 30% 정도 더 발생하는 것을 알 수 있다.

5. 결론

SIMD 명령어를 적용하여 역 코사인 변환함수의 병렬화를 통해 시간을 평균 58.72% 의 시간 감소를 보였다. 특히 SIMD명령어는 SIMD를 제공하는 CPU이면 싱글코어에서도 하드웨어의 특성을 적게 따르며 동작하기 때문에 사회 전반적으로 여러 분야에 거쳐 유용하게 이용될 수 있어 그 효율성은 더해질 것으로 예상된다.

기존 SIMD 명령어는 128bit 레지스터 xmm0~7을 사용하였는데 AVX [7]이상의 버전에서 레지스터의 크기는 256bit로 확장되었고, 사용할 수 있는 레지스터의 개수 또한 ymm0~15로 2배 확장되었다. 기존 SSE 프로그래밍의 문제점인 레지스터 개수와 크기의 확장을 통해 앞으로 더 빠른 속도와 더 높은 시간단축이 가능한 프로그램 개발이 가능할 것으로 전망된다.

본 논문에서 소개한 제안방법은 단지 역 코사인 변환 함수뿐 아니라 범용적인 방법으로써 다양한 프로그램에서 SIMD를 이용하여 병렬화를 진행하는 과정에서 사용할 수 있을 것 이라 예상된다.

참고문헌

[1] 효요성 외1명, *"UHD 고화질 영상 압축 기술"*, 진셈미디어, 서울 pp.1-30, 2013.

[2] 정영훈, *"SIMD 병렬프로그래밍"*, 프리렉, 부천, pp.3-148, 2012.

[3] Frank Bossen, et al., "HEVC Complexity and Implementation Analysis", *IEEE Trans. on Circuits and Systems for Video Technology*, vol.22, No 12. pp.1685-1699 December 2011.

[4] Kip R. Irvine., *"assembly Language for intel-based computers"*, pp.265-268

[5] 김완수 외1명, "Advanced SIMD를 이용한 화면 간 예측 고속화 방법", *Jornal of IKEEE.* Vol.16, No.4, pp.100-106 382~388, December 2012.

[6] [online] ftp://ftp.kw.bbc.co.uk/hevc/hm-10.0-anchors/bitstreams/

[7] Intel Corp. *Introduction to Intel® Advanced Vector Extensions.*