논문 2011-48SP-2-5

TMS320C64x+ DSP에서의 H.264/AVC 디블록킹 필터 최적화

(Optimization for H.264/AVC De-blocking Filter on the TMS320C64x+ DSP)

이 진 섭*, 강 대 범**, 심 동 규***, 이 수 연***

(Jin-Seop Lee, Dae-Beom Kang, Dong-Gyu Sim, and Soo-Youn Lee)

요 약

H.264/AVC의 디블록킹 필터는 복호화기 전체의 계산 복잡도 중 큰 비중을 차지하기 때문에, H.264/AVC 복호화기의 실시간 구현을 위해서는 디블록킹 필터의 계산 복잡도를 줄이는 것이 매우 중요하다. 디블록킹 필터의 계산 복잡도가 높은 이유는 여러 단계의 조건 분기문이 많고 메모리 접근이 자주 일어나기 때문이다. 따라서, 본 논문에서는 분기문과 메모리 접근을 최소화하는 디블록킹 필터의 구조를 제안하고, 필터 연산의 병렬화를 위해 소프트웨어 파이프라이닝이 가능하도록 하는 최적화 방법을 제안한다. 제안하는 방법은 TMS320C64x+ 계열의 DSP의 TMS320DM6467 EVM 보드에 구현하여 최적화를 진행하였다. 실험 결과, 최적화된 디블록킹 필터는 FFmpeg의 디블록킹 필터와 비교하였을 때 평균 약 46%의 사이클이 감소되었다.

Abstract

It is important to reduce computational complexity of de-blocking filter for real-time implementation, because it accounts for a great part of total computational complexity of the decoder. Because there are a lot of conditional branches and memory accesses in a decoding loop, it is not easy to speed up the de-blocking filter. Therefore, this paper presents a new algorithm of de-blocking filter minimizing conditional branches and memory accesses. The proposed structure of de-blocking filter enables filter operation to parallelize by software pipelining. The proposed optimization method was implemented on a TMS320DM6467 EVM board and we achieved approximately 46% cycle reduction, compared with that of FFmpeg.

Keywords : 디블록킹 필터, H.264/AVC, TMS320C64x+ DSP, 소프트웨어 파이프라이닝, 병렬 처리

Ⅰ. 서 론

H.264/AVC는 ISO/IEC (International Organization for Standardization/International Electrotechnical Commission) 산하 MPEG (Moving Picture Experts Group)과 ITU-T (International Telecommunication Union Telecommunication)의 VCEG (Video Coding Experts Group)이 함께 구성한 JVT (Joint Video Team)에서 개발한 동영상 압축 표준으로, 현존하는 동영상 압축 표준 중, 가장 우수한 압축 성능을 가진다. 특히 위성 및 지상파 DMB와 상용 IPTV 서비스 등의 다양한 멀티미디어 서비스에서 사용되고 있다.

H.264/AVC는 영상 내 혹은 영상 간의 차분영상을 구하여 정수변환 후 양자화를 거쳐 엔트로피 부호화를 통해 영상을 압축한다. H.264/AVC는 이러한 부호화 과정을 매크로블록 단위 혹은 하위 블록으로 나누어 부호화과정을 거치므로 블록 경계에서 왜곡이 발생되어 객관적/주관적인 화질을 저하시킨다. 디블록킹 필터는 이러한 블록 경계를 부드럽게 하여 왜곡을 제거함으로써

* 학생회원, 광운대학교 임베디드소프트웨어공학과
(Dept. of Embedded Software Engineering, Kwangwoon University)
** 정회원, 삼성전자(주)
(Samsung Electronics)
*** 정회원, 광운대학교 컴퓨터공학과
(Dept. of Computer Engineering, Kwangwoon University)
※ 이 논문은 2009년도 광운대학교 연구년 지원에 의해 연구되었음
접수일자: 2010년11월17일, 수정완료일: 2011년2월15일

주관적/객관적 화질뿐만 아니라 부호화 성능도 향상시킬 수 있다.

디블록킹 필터는 H.264/AVC 베이스라인 프로파일(baseline profile)을 기준으로 복호화기 전체 연산량의 33%를 차지할 만큼 계산 복잡도가 매우 높다[1]. 이러한 계산 복잡도로 인하여 H.264/AVC 복호화기의 실시간 복호화를 위해서는 디블록킹 필터의 계산 복잡도를 줄이는 연구가 매우 중요하다. 디블록킹 필터의 계산 복잡도가 높은 이유는 블록 간의 경계강도 (boundary strength)를 구하는 과정과 필터링 연산과정에서 여러 단계의 분기문 들과 다수의 메모리 접근을 사용하기 때문이다.

이러한 문제를 해결하기 위한 하드웨어 구조를 제안하는 연구들이 지속적으로 이루어지고 있으나,[2~6] 소프트웨어적으로 디블록킹 필터를 최적화하는 연구는 많지 않은 실정이다. J. Lou 등은[7] 슬라이스 타입별로 블록 경계강도를 구하는 방법을 달리하여 인트라 슬라이스에서 필요 없는 블록 경계강도 결정을 위한 분기문 들을 제거하고, 슬라이스 타입별로 경계강도의 발생 확률이 큰 순서대로 조건을 검사하도록 하여 디블록킹 필터의 속도를 향상시키는 방법을 제안했다. J. Lou 등의 연구는 소프트웨어 레벨에서 알고리듬 단계에서만 최적화하는 알고리듬이 적용되었다. Z. Yang 등은[8] 일반적인 DSP 아키텍처에서 파이프라인 구조를 이용하여 필터 연산을 분기문 없이 구현하는 방법을 제안하였다. 필터 적용 여부를 결정하는 조건에 상관없이 모든 필터 출력 값을 미리 계산해놓고, 마지막에 필터 조건을 이용한 조건부 실행 (conditional execution)으로 최종 결과 값 하나를 선택하는 방식으로 분기문을 제거하였다. 이 연구에서는 부분적으로 알고리듬 수준의 최적화를 다루고 있지만, 전반적으로 일반적인 DSP 아키텍처의 파이프라인 구조에 특화된 알고리듬을 이용하여 최적화를 수행하였다.

본 논문에서는 멀티미디어 관련 임베디드 분야에서 널리 사용되고 있는 DSP 아키텍처에서 실시간 실행 가능한 H.264/AVC 복호화기의 구현을 위해, 아키텍처 수준과 알고리듬 수준을 모두 고려한 최적화 알고리듬을 제안한다. 먼저, 조건 분기문을 최소화하기 위한 알고리듬 수준에서의 전체적인 디블록킹 필터 구조를 제안하고, 이 구조 안에서 TMS320C64x+ DSP 아키텍처의 특징들을 효율적으로 사용하는 방법들을 제안한다.

본 논문은 다음과 같은 순서로 구성된다. Ⅱ장에서 TMS320C64x+ DSP의 간단한 아키텍처 소개와 소프트웨어 파이프라이닝 방법에 대해서 설명한다. Ⅲ장에서는 H.264/AVC의 표준 디블록킹 필터링 방법에 대해 설명하고, Ⅳ장에서는 제안하는 알고리듬 수준의 전반적인 디블록킹 필터의 구조와 소프트웨어 파이프라인 구조를 고려한 필터 연산의 병렬화에 대해 자세히 설명한다. Ⅴ장에서는 제안한 방법의 성능을 검증하기 위한 실험 결과를 기술하고, 마지막으로 Ⅵ장에서 결론을 맺는다.

Ⅱ. TMS320C64x+ DSP의 주요 특징들

1. TMS320C64x+ DSP 아키텍처의 구성

TMS320C64x+ DSP는 VLIW (Very Long Instruction Word) 아키텍처로써 8개의 기능 유닛(functional unit)을 가지고 있어 최대 8개의 명령어를 동시에 처리할 수 있다. 또한, 대부분의 명령어들이 SIMD (Single Instruction Multiple Data) 형태로 구성되어 있어 프로그램의 병렬성을 극대화할 수 있다. TMS320C64x+ DSP는 그림 1과 같이 두 개의 데이터 경로로 구성되어 있으며, 각 데이터 경로마다 32개의 레지스터를 포함하는 레지스터 파일과 4개의 기능 유닛(.L, .S, .M, .D)으로 구성된다.[9]

TMS320C64x+ DSP는 C64x에 비해 각 기능 유닛마다 곱하기, 정렬, 비트 조작 등의 기능을 하는 효율적인 명령어들이 다수 추가되었다.[10]

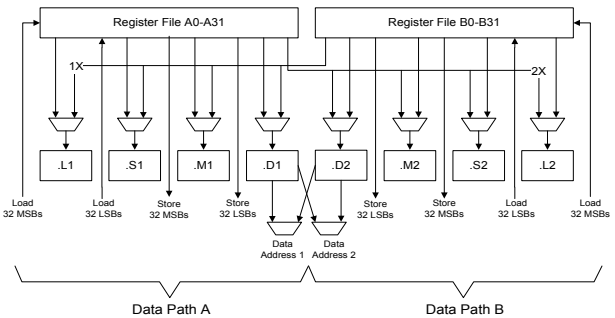

그림 1. TMS320C64x+ DSP 데이터 경로
Fig. 1. Data paths of TMS320C64x+ DSP.

2. 소프트웨어 파이프라이닝

소프트웨어 파이프라이닝은 루프의 반복들 간의 병렬화를 위한 명령어 스케줄링 (instruction scheduling)의 한 방법이다. 소프트웨어 파이프라이닝 방법을 통해 구현된 루프는 그림 2와 같이 프롤로그, 커널과 에필로

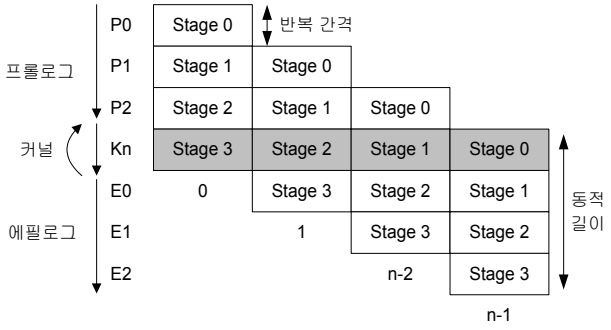

그림 2. 소프트웨어 파이프라이닝된 루프의 실행 흐름
Fig. 2. Software-pipelined execution flow for loop operation.

그 부분으로 구성되어 있다. 커널은 파이프라인의 상태가 계속 유지되는 상태를 말하며 프롤로그와 에필로그는 각각 커널이 동작하기 전/후의 상태를 말한다.

그림 2와 같이 루프의 한 반복이 4개의 단계로 이루어진다고 했을 때, 병렬로 수행되는 연속된 반복 간에 사이클 차이를 반복 간격 (iteration interval)이라고 한다. 반복 간격이 작으면 작을수록 반복 간의 병렬성이 극대화된다. 루프의 한 반복이 완료되는데 필요한 명령어의 총 사이클 수를 동적 길이 (dynamic length)라고 한다.

TMS320C64x+ DSP에는 그림 2와 같이 실행되는 루프에 대해 소프트웨어 파이프라이닝 전용의 하드웨어 버퍼에 명령어를 저장하여, 병렬적으로 실행될 수 있도록 하는 SPLOOP (software-pipelined loop) 기능이 추가되었다.[10] SPLOOP는 소프트웨어 파이프라이닝을 위해 루프 하나의 반복을 위한 명령어들을 저장할 전용 하드웨어 버퍼와, 버퍼에 저장되어 있는 명령어들을 선택적으로 실행할 수 있도록 하는 선택기로 구성된다. SPLOOP 기능을 사용함으로써 얻을 수 있는 장점은 프롤로그와 에필로그를 따로 기술하지 않아도 되므로 코드 사이즈가 줄어들고, 매 사이클마다 루프의 명령어들을 메모리에서 가지고 (fetch) 오지 않고 전용 하드웨어 버퍼에서 가지고 오기 때문에 전체적인 실행 속도를 향상시킬 수 있다. SPLOOP 기능을 사용하기 위해서는 몇 가지 조건들이 따른다. SPLOOP 버퍼의 크기가 한정되어 있으므로 루프의 동적 길이가 48 사이클을 넘어서는 안 되며 반복 간격이 14 사이클을 넘어서는 안 된다. 마지막으로 루프 안에 함수 호출 및 분기문이 없어야 한다.

Ⅲ. 디블록킹 필터 알고리듬

H.264/AVC의 디블록킹 필터는 각 매크로블록 내에서 4×4 블록의 경계에 대해 필터링이 적용된다. 각 블록에 대해, 수직 방향의 블록 경계인 경우 왼쪽에서 오른쪽 방향으로 필터링되고, 수평 방향의 블록 경계인 경우 위쪽에서 아래쪽으로 필터링이 적용된다. 디블록킹 필터는 크게 블록 경계강도 결정과 필터 연산 모듈로 나눌 수 있다. 먼저 4×4 블록 단위로 블록 경계강도를 구하고, 구해진 블록 경계강도를 필터링 강도의 결정을 위한 인자로 사용하여 휘도/색차 블록에 대해 필터 연산이 적용된다. 다음은 각 모듈의 동작에 대해 각각의 설명을 보인다.

1. 블록 경계강도 결정

디블록킹 필터는 필터 강도를 결정하기 위해 4×4 블록 각각에 대해 블록 경계강도를 구한다. 블록 경계강도는 블록 경계를 기준으로 인접한 블록 P와 Q 블록의 예측 모드와 움직임 벡터, 참조 프레임, 잔여 변환 계수와 같은 코딩 상태들에 따라 결정된다. 그림 3은 인접한 블록 P와 Q의 화소들과, 이 블록들에 대한 경계강도의 결정 조건을 나타낸다.

경계강도가 4인 경우 강한 필터링 (strong filtering), 1 또는 2, 3인 경우 약한 필터링 (week filtering)이 적용되고, 0인 경우엔 필터링이 적용되지 않는다.

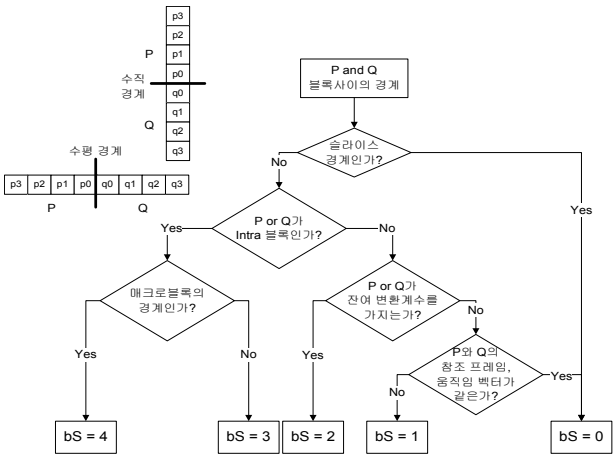

그림 3. 블록 경계강도 조건
Fig. 3. Condition for block boundary strength.

2. 필터 연산

블록 경계강도가 결정되면 연속된 8개 화소 값에 의해 그림 4와 같이 두 개의 필터 중 하나가 적용된다. 0

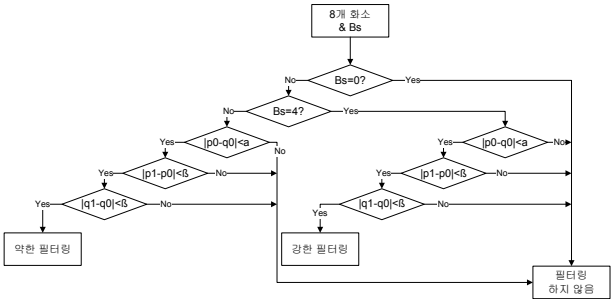

그림 4. 필터링 모드 결정 조건
Fig. 4. Condition for decision of filtering mode.

이 아닌 블록 경계강도의 경우, 블록 경계가 에지(edge)인지 아닌지를 판단하는 연속된 화소 값의 차이와 임계치 값 α, β와 비교를 위한 여러 단계의 분기문을 통해 필터 적용 여부를 결정하게 된다. 양자화 파라미터 (QP)에 의해 결정되는 α, β와 비교했을 때 화소들의 차이 값이 작을 경우, 블록 경계가 에지가 아닌 양자화에 의한 에러로 판단하고 저주파 통과필터 (low-pass filter)가 적용된다. 그렇지 않은 경우에는 블록 경계를 에지로 판단하여 필터를 적용시키지 않는다.

Ⅳ. 제안하는 디블록킹 필터 최적화 방법

본 장에서는 디블록킹 필터를 최적화하기 위한 알고리듬 수준의 구조적 최적화와 아키텍처 수준의 최적화 방법을 제안한다. 먼저 알고리듬 수준에서 블록 경계강도를 구하기 위해 필요한 여러 단계의 분기문을 최소화하는 디블록킹 필터 구조에 대한 최적화 방법을 제안하고, 이를 바탕으로 하여 아키텍처 수준에서 DSP의 8개 기능 유닛들을 최대한 동시에 사용할 수 있도록 하는 소프트웨어 파이프라이닝 방법을 필터 연산에 적용함으로써 필터 연산 루프를 병렬화할 수 있는 최적화 방법을 제안한다. 다음은 각각의 최적화 방법에 대한 자세한 설명을 보인다.

1. 알고리듬 수준의 구조적인 최적화

디블록킹 필터는 매크로블록 내의 4×4 블록 각각의 블록 경계강도를 구하기 위해 그림 3과 같이 다수의 조건 비교문이 필요하다. 대부분의 DSP에서는 분기 예측(branch prediction) 기법이 사용되지 않으므로, 분기문은 성능저하의 가장 큰 원인이 된다. 이러한 이유로 블록 경계강도를 구하기 위해 필요한 분기문들을 최소화하고, 필터 연산 모듈에서 블록 경계강도에 따라 필터

표 1. 매크로블록 타입에 따른 구분
Table 1. Classification by type of macroblock.

현재 매크로블록 타입	왼쪽 매크로블록 타입	위쪽 매크로블록 타입
스킵 모드	×	×
인트라 모드	×	×
인터 모드	인트라 모드	인트라 모드
		인터 모드
	인터 모드	인트라 모드
		인터 모드

링 모드를 구분하는 분기문을 제거할 수 있는 알고리듬 수준의 구조적인 최적화 방법을 제안한다.

먼저 블록 경계강도를 구하기 위한 여러 단계의 분기문들을 최소하기 위해, 현재 매크로블록 타입과 주변 매크로블록 타입의 조합에 따라 불필요하게 적용되는 분기문을 제거한다. 표 1은 현재 및 주변 매크로 블록 타입의 조합 종류를 보인다.

J. Lou 등은[7] 슬라이스 타입별로 블록 경계강도를 구하는 방법을 달리하여 필요 없는 조건 분기문을 제거하고, 경계강도의 발생 확률이 큰 순서대로 조건을 검사하도록 하는 방법을 제안하였다. 슬라이스 단위로 나누게 되면 인트라 슬라이스에 대해서는 매크로블록 경계 여부만을 확인하면 되므로 분기문이 많이 줄어들지만, 인터 슬라이스에는 인트라블록이 존재할 수 있으므로 전체적으로 분기문이 줄어드는 효과가 작다. 이러한 이유로 주변 매크로블록의 타입까지 고려함으로써, 더 많은 분기문을 제거할 수 있다. 표 1에 나와 있는 각 조합은 매크로블록 경계일 때에만 주변 블록 타입이 현재 블록의 경계강도를 구할 때 영향을 미친다. 매크로블록 경계부터 차례대로 필터가 적용되므로, 별도의 분기문 추가 없이 매크로블록 경계와 내부를 같은 루프에서 처리하지 않고 별도로 처리하여 매크로블록 내부에서는

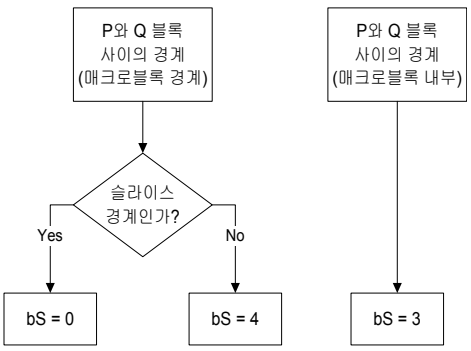

그림 5. 인트라 블록의 블록 경계강도 결정 조건
Fig. 5. Condition of boundary strength for intra block.

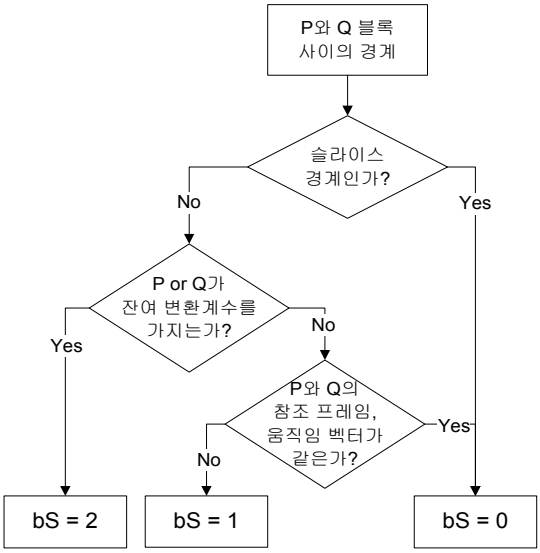

그림 6. 현재 및 주변 매크로블록 타입이 모두 인터 모드인 블록 경계강도 결정 조건

Fig. 6. Condition of boundary strength of which type of both current and neighbor MB are inter block.

현재 매크로블록에 대한 검사만 할 수 있다.

그림 5는 현재 매크로블록 타입이 인트라 블록일 때 매크로블록 경계/내부에 대한 블록 경계강도를 결정하기 위한 블록도이다.

그림 6은 현재 및 주변 매크로블록 타입이 모두 인터 모드이면서 블록 사이의 경계가 매크로블록 경계인 경계강도를 결정하는 조건이다. 매크로블록 경계인 경우에는 슬라이스 경계 여부를 검사해야 하고, 매크로블록 내부인 경우에는 슬라이스 경계 여부에 대한 조건 또한 제거할 수 있다.

그리고 현재 매크로블록 타입이 인터 모드이고 주변 매크로블록 타입이 인트라 모드인 경우에, 매크로블록 경계에 대해서는 그림 5의 왼쪽 그림과 같은 결정 조건을 사용하고, 매크로블록 내부에 대해서는 그림 6의 조건들 중에서 슬라이스 경계 여부를 결정하는 조건이 제거된 결정 조건을 사용한다.

그리고 현재 매크로블록 타입이 스킵 블록인 경우에는 매크로블록 내부의 블록은 현재 블록과 인접한 블록간의 움직임 벡터와 참조 프레임이 같을 수밖에 없고 잔여 변환계수를 가지지 않기 때문에 필터링이 항상 적용되지 않는다. 따라서 매크로블록 경계에만 필터링을 적용하면 된다. 그림 7은 스킵 매크로블록의 블록 경계강도를 결정하는 조건에 대한 블록도이다. 현재 매크로블록 타입이 스킵 블록인 것을 이미 알고 있으므로, Q가 인트라 블록인지와 잔여 변환계수를 가지는지에 대한 조건은 검사하지 않아도 된다.

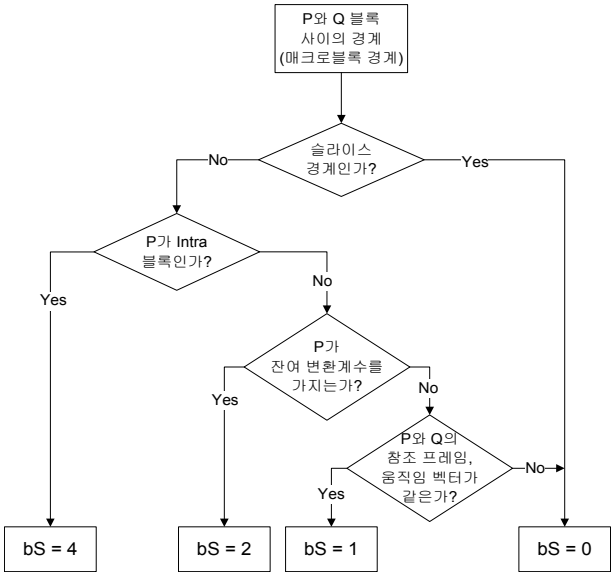

그림 7. 스킵 블록의 블록 경계강도 결정 조건

Fig. 7. Condition of boundary strength for skip block.

이와 같은 방법으로, 휘도블록의 경우 매크로블록내의 4개의 4×4 블록들의 블록 경계강도를 구한 후 4개의 블록에 대해 16번 루프를 돌면서 필터 연산이 적용된다. 이 때 이 4개의 블록들의 블록 경계강도는 서로 다를 수 있으나, 항상 같은 필터링 모드를 갖는다. 즉 4개 블록들의 경계강도를 순차적으로 구하지 않고, 한 번에 구한다면 4개 블록 전체에 적용될 필터링 모드를 알 수 있으므로, 강한 필터링 모드와 약한 필터링 모드의 필터 연산 모듈을 각각 다른 함수로 구현하고, 필터링 모드에 해당하는 필터 연산 함수를 바로 호출함으로써 그림 4에서 볼 수 있는 필터링 모드를 결정하기 위한 조건 분기문을 제거할 수 있다. 인트라 블록의 경우 4개 4×4 블록들의 경계강도는 3 또는 4로 항상 같으므로 쉽게 구할 수 있지만, 움직임 예측 블록의 경우 블록들의 경계강도는 서로 다른 값을 가질 수 있으므로, 이를 구하기 위한 효과적인 방법이 필요하다.

그림 8은 스킵 매크로블록의 경계에 해당하는 4개의 4×4 블록들의 블록 경계강도를 구하는 방법에 대한 블록 다이어그램으로써, 다른 매크로블록 타입에 대해서도 비슷한 방법이 적용된다. 기존에 스킵 매크로블록내의 4개 블록의 경계강도를 구하기 위해 루프를 돌면서, 각 루프 당 4개의 분기문이 필요하므로 총 16개의 분기문이 필요했었다. 제안하는 방법에서는 4개 블록의 경계강도를 구하기 위해 필요한 값들을 모두 불러온 후에, 단 4개의 분기문만으로 구할 수 있다. P 블록이 인

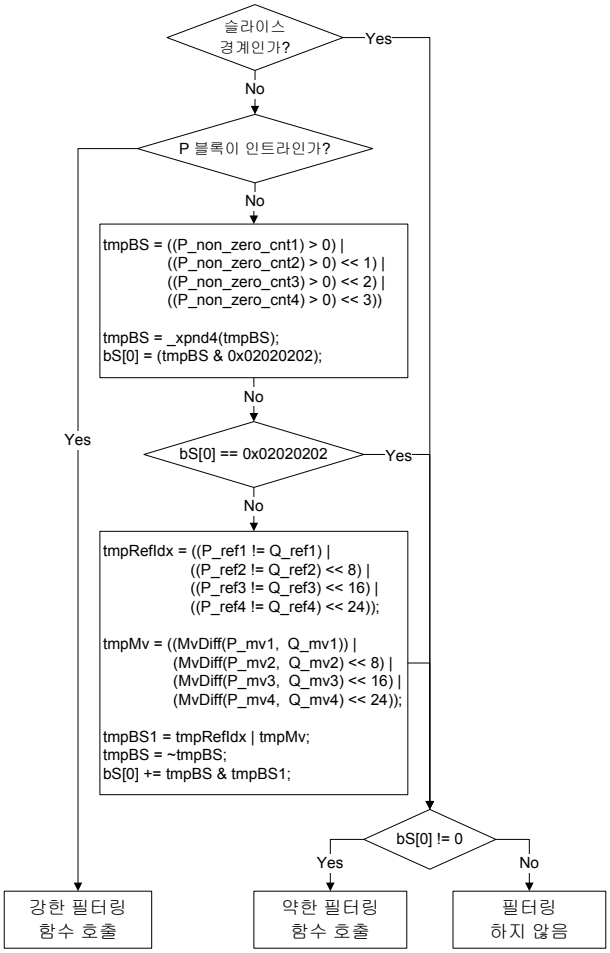

그림 8. 제안하는 블록 경계강도 결정 방법
Fig. 8. Proposed method for determining block boundary strength.

트라인 경우 바로 강한 필터링이 적용되고, 인트라가 아닌 경우에는 4개 블록들의 0이 아닌 계수의 개수(P_non_zero_cnt)를 사용하여 경계강도가 2인 블록에 대해서 bS[0]에 1바이트 단위로 해당하는 위치에 저장한다. 4개 블록의 경계강도가 모두 2인 경우에는 다른 조건 검사 없이 바로 약한 필터링을 적용시키고, 그렇지 않은 경우에는 P, Q 블록의 참조 프레임과 움직임 벡터 값을 비교하여 경계강도가 1인지 0인지에 대한 값을 bS[0]의 나머지 위치에 저장한다. 4개의 블록 경계강도가 바이트 단위로 저장되어 있는 bS[0]를 사용하여 필터링 함수에서 필터 연산이 수행된다.

앞에서 제안한 방법을 사용하여 4개의 4×4 블록들의 블록 경계강도를 한 번에 구하고, 구하는 즉시 4개의 블록들의 필터링 모드가 결정되므로 추가적인 분기문없이 강한 필터링 함수와 약한 필터링 함수로 구분할 수 있게 된다.

2. 소프트웨어 파이프라이닝을 고려한 필터 설계

이번 절에서는 기존의 순차적이던 필터 연산을 병렬처리가 가능하도록 하기위해 TMS320C64x+ DSP의 SIMD 명령어를 효과적으로 사용하면서 소프트웨어 파이프라이닝 구조를 가지는 필터 연산 방법을 제안한다. 필터 연산의 기본적인 구조는 Z. Yang 등이 제안한[6] 필터 적용 여부를 결정하는 조건에 상관없이 모든 필터 출력 값을 미리 계산해놓고, 마지막에 필터 조건을 이용한 조건부 실행으로 최종 결과 값만을 메모리에 저장하도록 하는 방식을 기본으로 하여, 그림 9와 같이 크게 네 단계로 나눌 수 있다.

한 개의 매크로블록에 대한 필터 연산을 위해 휘도 블록의 경우에 이 단계 과정을 16번 반복하고, 색차 블록의 경우에는 앞의 네 단계 과정을 4번 반복하여 필터 연산이 수행된다. 색차 블록의 경우 앞의 과정을 8번이 아닌 4번 반복하는 이유는, 휘도 블록의 필터 연산에 비해 색차 블록의 필터 연산 방법이 훨씬 간단하여 하나의 반복 내에서 사용되는 명령어 개수가 많지 않기 때문에 필터 연산을 루프 언롤링 (loop unrolling)하여 기능 유닛들을 동시에 최대한 많이 사용할 수 있도록 하기 위함이다. 같은 이유로 휘도 블록에 대해서 루프 언롤링을 하지 않는 이유는 이미 8개의 기능 유닛들을 최대한 동시에 사용하고 있기 때문에 오히려 루프 횟수를 줄이면 루프의 반복들 간의 중첩 횟수가 줄어들어 소프트웨어 파이프라이닝 효과가 감소될 수 있기 때문이다.

Z. Yang 등이 제안한[8] 방식에서는 블록 경계강도 값과 상관없이 강한 필터링과 약한 필터링 모드의 모든

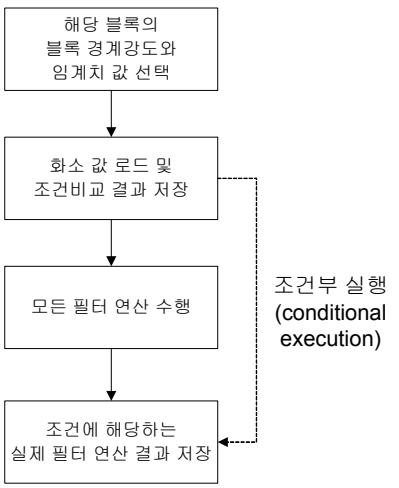

그림 9. 필터 연산의 수행 순서
Fig. 9. Execution order of filter operation.

필터 연산을 처리한 후 마지막에 조건비교 결과에 해당하는 필터 연산 결과만을 메모리에 저장한다. 이 방식은 블록 경계강도 값에 따라 필터링 모드를 구분하지 않고 처리하기 때문에 이를 구분하는 분기문이 필요가 없다는 장점이 있다. 하지만 앞 절에서 설명한 것처럼 4개의 블록들은 항상 같은 필터링 모드를 가지므로 강한 필터링과 약한 필터링 모드의 필터 연산을 같은 루프 안에서 모두 처리할 필요가 없다. 두 모드의 필터 연산 함수를 각각 다른 함수로 구현함으로써, 두 모드의 필터 연산을 합쳐놓은 것보다 빠르게 동작할 수 있다. 하지만 Z. Yang 등의 제안한 명령어 그래프는 두 모드의 필터 연산들끼리 서로 의존성이 많아서 두 모드를 나눈다고 하더라도 사이클 상의 이점을 가져올 수 없다. 따라서 본 논문에서는 각 모드에 최적화된 명령어 그래프를 제안한다.

강한 필터링 모드의 경우 블록 경계강도가 4일 경우에만 선택이 되는데, 이는 인트라 블록에서만 발생하므로 다른 필터링 모드에 비해 빈도가 매우 낮다. 따라서 본 논문에서는 휘도 블록의 수평/수직 방향의 강한 필터링 함수에 대해서는 Z. Yang 등이 제안한[8] 구조와 명령어들을 그대로 사용하고, 나머지 필터링 함수들에 대해 본 논문에서 제안하는 방식을 사용한다. 본 논문에서는 휘도 블록의 수직방향 약한 필터링 연산에 대해서 자세히 다룬다. 나머지 함수들도 이와 비슷한 방식으로 이루어진다.

가. 블록 경계강도와 임계치 값 선택

그림 10은 휘도 블록의 수직방향 약한 필터 연산 과정 중에서 첫 번째 단계인 현재 필터링이 취해질 블록에 해당되는 블록 경계강도 cur_BS와 임계치 값 cur_TC0, minusTC0를 선택하는 과정에 대한 명령어 그래프이다.그림 10에서 점선으로 표시된 부분은 cnt_flag 값에 따라 ADD 명령어를 조건부 실행한다는 의미이다. iBS와 TC0에는 각각 4개의 블록에 대한 블록 경계강도와 임계치 값이 저장되어 있다. 따라서 16번 루프를 돌면서 새로운 블록을 만날 때마다 iBS와 TC0에서 현재 블록에 해당하는 값을 가져와야 한다. 이를 분기문으로 처리하면 소프트웨어 파이프라이닝 구조가 깨지기 때문에 조건부 실행을 사용하도록 한다. 먼저 매 반복마다 tmp_cnt (0x2221)를 오른쪽으로 시프트하여 최하위 비트가 1일 경우에만 iBS와 TC0를 시프트하여 현재 블록 경계강도와 임계치 값을 구한다.

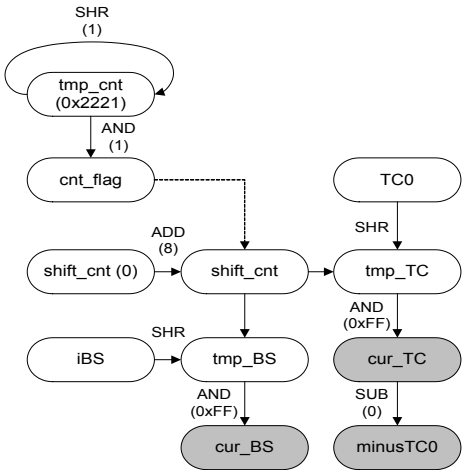

그림 10. 블록 경계강도, 임계치 값에 대한 명령어 그래프

Fig. 10. Instruction graph to get bS and tC.

나. 화소 로드 및 필터링 조건 비교결과

그림 11은 그림 4에 나와 있는 P, Q 블록 모두에 대한 필터링 여부를 결정하는 조건비교 결과인 flag와 p0, q0 화소 각각에 대해 필터링 여부를 결정하는 조건 비교 결과인 ap, aq를 구하는 과정이다. 이 조건 비교 결과들을 구하기 위해 사용되는 중간 결과 값들 xp0p1p2,

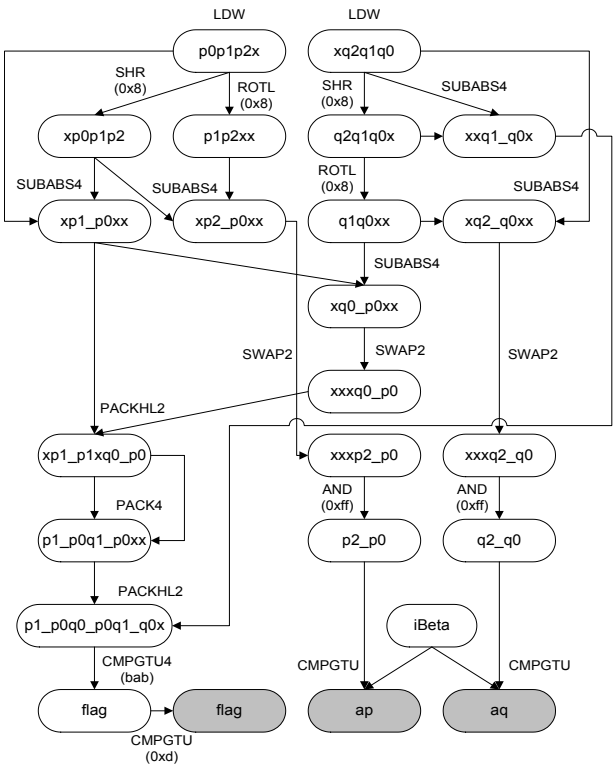

그림 11. 화소 값 및 조건비교 결과들을 구하는 과정

Fig. 11. Process to get pixels and condition comparison.

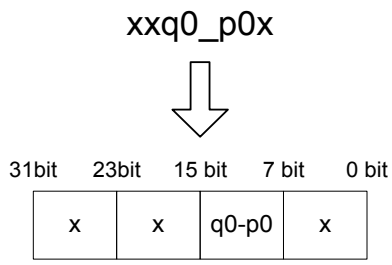

그림 12. 레지스터의 구성
Fig. 12. Register structure.

p1p2xx, q2q1q0x, q1q0xx는 3번째 단계인 실제 필터 연산에서도 사용된다.

레지스터 이름에서 ×는 32비트 레지스터에서 의미 없는 8비트 값을 의미한다. 그림 12에 나와 있는 것처럼 ××q0_p0× 레지스터는 상위 16비트와 하위 8비트에는 의미 없는 값이 들어있고, 중간에 q0-p0 값이 저장되어 있다는 의미이다.

명령어를 선택함에 있어서 동일한 명령어를 여러 번 사용하는 것은 좋지 않다. 그 이유는 각 명령어마다 실행되는 기능 유닛 등이 다르므로, 데이터 경로 A와 B에 각각 4개씩 있는 .L, .S, .D, .M 기능 유닛들을 고르게 사용하여 8개 기능 유닛을 최대한 많이 사용하도록 해야 한다. 때문에 LDW 명령어로 불러온 화소 값들을 시프트 할 때도 .S 유닛을 사용하는 SHR 명령어 외에도 .M 유닛을 사용하는 ROTL 명령어를 사용하여 기능 유닛을 최대한 고르게 사용한다. .M 기능 유닛의 명령어들은 대부분 2사이클 이상 걸려야 결과가 나오는 단점을 가지고 있지만, .S와 .L 기능 유닛들의 명령어에 비해 사용되는 빈도가 낮기 때문에 최대한 많이 사용하는 것이 좋다.

LDW 명령어를 사용하여 P 블록과 Q 블록의 화소 값들을 p0p1p2x, xq2q1q0 레지스터에 각각 저장 한 후 SHR, ROTL 명령어들을 사용하여 화소 값들을 정렬한다. 정렬한 화소 값들을 SUBABS4 명령어를 사용하여 화소 값들의 차의 절대 값을 xp1_p0xx, xp2_p0xx, xq0_p0xx, xq2_q0xx 레지스터들에 저장한다.

앞에서 구한 화소 값들의 차의 절대 값들을 PACKHL2, PACK4, SWAP2 명령어들을 사용하여 필터링 조건비교 결과 값들을 p1_p0q0_p0q1_q0x, p2_p0, q2_q0 레지스터에 저장한다. 우선 전체적인 필터링 여부를 결정하는 조건비교 결과를 구하기 위해 p1_p0q0_p0q1_q0x 레지스터와 상위 비트부터 8비트씩 β, α, β 순서로 저장되어 있는 레지스터와 CMPGTU4 명령어를 사용하여 비교 결과를 flag에 저장한다. 만약 |p1-p0|, |q0-p0|, |q1-q0| 값들이 각각 β, α, β 값들보다 모두 클 경우 flag에는 111x 값이 저장된다. 하위 1비트에는 의미 없는 비교 결과이므로, flag와 0xd 값을 비교하여 flag가 클 경우 비교 결과가 참이 되어 flag에는 1이 저장된다. ap, aq 또한 이와 같은 방식으로 구한다.

다. 모든 필터 연산 수행

그림 13은 실제로 필터 연산이 적용된 p0, p1, q0, q1 화소들을 구하는 과정을 보인다. 시프트된 화소 값이 저장되어 있는 p0p1p2x, xq2q1q0, xp0p1p2 들은 그림 11에 나와 있는 조건비교 결과들을 구하는 과정 중에 구해지는 값들이다. 이 값들을 SHLMB, SHRMB, PACKLH2 명령어들을 사용하여 필터링이 적용될 화소들을 각각의 레지스터에 모아서 저장한 후, 이미 저장해 둔 상수 값인 0x02fc0101, 0x0101dc02, 0xff04fc01 들과 DOTPSU4 명령어를 통해 4탭 필터를 적용한 뒤 ADD, SHR 명령어를 통해 나온 diff1, diff2, diff3 값들을 MIN2, MAX2 명령어를 사용하여 클리핑 연산을 취한다. 마지막으로 클리핑된 값들인 diff1, diff2, diff3 값들과 실제 화소 값인 p0, p1, q0, q1 들과 ADD, SUB 연산을 통해 최종 필터링된 화소 값인 Luma_P0,

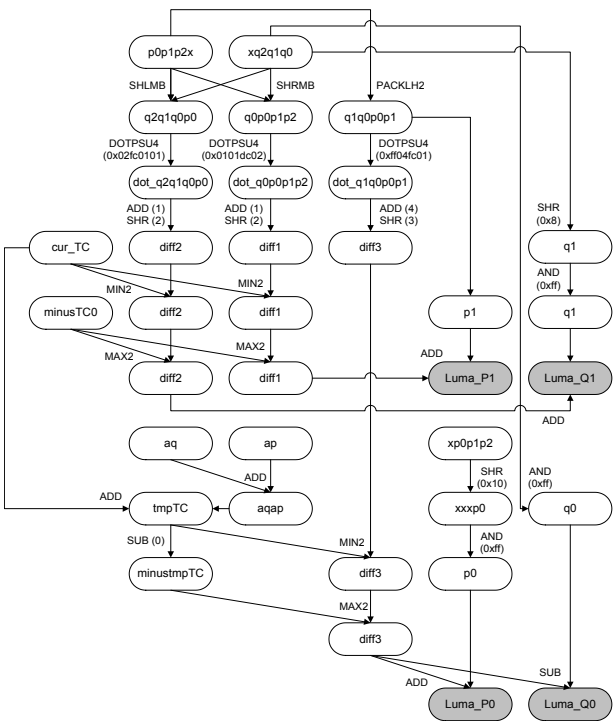

그림 13. 필터링된 화소 값을 구하는 과정
Fig. 13. Process to get filtered pixels.

Luma_P1, Luma_Q0, Luma_Q1 들을 구한다.

라. 필터 연산 결과 저장

마지막으로 그림 14는 앞에서 구했던 조건비교 결과 값들을 조건으로 하여 실제 필터링 결과 값만을 조건부 저장하는 과정을 보인다. 필터링된 화소 값들 중에 Luma_P0와 Luma_Q0는 flag만을 조건으로 사용하고, Luma_P0는 flag와 ap를 더한 값을 조건으로 사용하며, Luma_Q0는 flag와 aq를 더한 값을 조건으로 사용하여 조건이 만족되는 필터링 결과만을 메모리에 저장되도록 한다. 현재 블록의 블록 경계강도가 0일 때는 flag 값을 0으로 치환하여 필터링이 적용되지 않도록 한다.

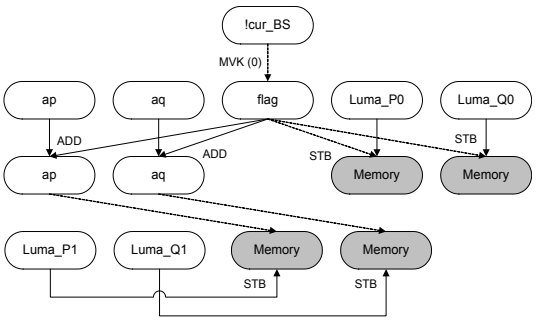

그림 14. 필터링된 화소 값들 중에 조건비교 결과에 해당하는 화소 값만을 조건부 저장하는 과정

Fig. 14. Process for conditional storing only pixel value corresponding to the condition comparison's result among filtered pixel values.

나머지 휘도 블록의 수평방향의 약한 필터링 모드, 색차 블록의 수직/수평방향의 강한 필터링/약한 필터링 모드들에 대한 필터 연산 또한 앞에서 보인 휘도 블록의 수직방향 약한 필터링 모드의 필터 연산 방식과 거의 동일하게 이루어진다.

Ⅴ. 제안하는 디블록킹 필터의 실험 결과

본 논문에서 제안하는 디블록킹 필터 최적화 방법의 성능을 평가하기 위하여 실험은 크게 3가지로 나누어 진행하였다. 첫 번째는 블록 경계강도를 구하는 부분에 있어서 J. Lou의 방법과의 분기문 개수를 비교한 실험이고, 두 번째는 필터 연산의 루프를 소프트웨어 파이프라이닝을 적용하여 병렬화한 성능을 측정하기 위해 각 필터별로 Z. Yang의 방법과 제안한 방법 간의 사이클을 비교하는 실험이며 마지막으로 전체적인 디블록킹 필터의 속도 향상에 대한 실험이다.

슬라이스 타입별로 블록 경계강도를 구하는 방법을 달리한 J. Lou의 방법과 본 논문에서 제안한 방법과의 분기문의 개수를 비교하면 다음과 같다. 인트라 슬라이스의 인트라 매크로블록 내의 모든 블록에 해당하는 수평/수직 블록 경계강도를 구하기 위해서 J. Lou의 방법은 76개의 분기문이 필요하고, 본 논문의 방법은 인트라 블록 경계에서 슬라이스 경계인지 검사하기 위해 수평/수직 방향으로 4개 블록마다 한 개씩 필요하므로, 단 두 개의 분기문만이 필요하다. 그리고 인터 슬라이스의 한 개의 매크로블록에 대해 J. Lou의 방법은 128개의 분기문이 필요하고, 본 논문에서 제안하는 방법에 의해 현재 및 주변 매크로블록의 타입에 따라 필요한 각각의 분기문의 개수는 표 2와 같다.

본 논문에서 제안하는 블록 경계강도를 구하는 방법은 현재 및 주변 매크로블록 타입에 대해서 구분하는 분기문이 매크로블록 당 6개의 분기문이 추가로 필요하

표 2. 인터 슬라이스의 한 개의 매크로블록 당 필요한 분기문 개수의 비교

Table 2. Comparison of the number of branches required per a macroblock about inter slice.

현재 매크로블록 타입	주변 매크로블록 타입	분기문 개수		
		경계	내부	합계
스킵 모드	×	6	0	6
인터 모드	인터 모드	4	6	10
인터 모드	인트라 모드	2	6	8

표 3. 필터 연산들에 대한 소프트웨어 파이프라이닝 사이클 비교

Table 3. Comparison of the software-pipelined cycles for each filter operation.

	필터 연산	커널	루프 횟수	프롤로그 에필로그	전체 사이클
휘도/수직	Z. Yang	15	16	22	262
	강한 필터	12	16	24	216
	약한 필터	13	16	13	221
휘도/수평	Z. Yang	16	16	31	287
	강한 필터	13	16	39	247
	약한 필터	15	16	19	259
색차/수직	Z. Yang	11	8	24	112
	강한 필터	6	4	18	42
	약한 필터	9	4	18	54
색차/수평	Z. Yang	12	8	26	112
	강한 필터	8	4	16	48
	약한 필터	10	4	20	60

기 때문에 표 2의 분기문 개수에 각각 6개씩 더하면, 매크로블록 당 평균 14개의 분기문이 필요하므로, 기존의 128개에 비해 분기문을 많이 줄일 수 있다.

표 3은 Z. Yang의 방법과 제안한 방법 간의 필터 연산에 대한 소프트웨어 파이프라이닝된 루프의 사이클을 비교한 결과이다. 휘도 블록의 수직 필터의 경우 약 17% 사이클이 감소하였다. 강한 필터링의 경우 약한 필터링보다 훨씬 간단한 필터 구조를 가지고 있음에도 불구하고 Z. Yang의 방법으로 구현한 강한 필터링과 제안한 방법의 약한 필터링의 사이클이 크게 차이가 나지 않는 것으로 보아 Z. Yang의 방법과 비교하여 본 논문에서 제안한 방법이 효과적인 것을 알 수 있다. 휘도 블록의 수평 필터의 경우 화소 값을 LDW 명령어를 사용하여 연속으로 읽어올 수 없으므로, 그만큼 메모리 접근을 더 많이 해야 하기 때문에 수직 필터에 비해서 약간 더 많은 사이클이 걸린다. 색차 필터는 Z. Yang의 방법보다 약 56% 사이클이 감소한 것을 볼 수 있다.

그리고 제안하는 최적화 방법을 사용한 디블록킹 필터의 전체적인 성능 측정을 위해 H.264/AVC 오픈소스 라이브러리인 FFmpeg[15] 복호화기를 기반으로 구현하였다. TMS320C64x+ 계열의 DSP가 장착된 TMS320DM6467 EVM 보드 상에서 TI에서 제공하는 통합 개발 환경인 CCS (Code Composer Studio) v3.3을 사용하여 실험을 진행하였고, 실험 조건으로는 4개의 D1 (720×480)과 CIF (352×288) 해상도의 영상들을 4개의 QP 값으로 각각 100장씩 부호화한 비트스트림을 사용하였다. 컴파일러 옵션은 -o3만을 사용하였고, 정확한 수행 사이클을 측정하기 위하여 표준 라이브러리를 사용하지 않고 TMS320C64x+ 계열의 DSP에서만 제공하는 TSCL (Time Stamp Counter Low Half) 레지스터를 직접 읽어서 디블록킹 필터의 사이클을 측정하였다.

자세한 실험 조건은 표 4에 나타내었다. 각 방법 별로 디블록킹 필터의 복호화 사이클은 감소율은 식(1)과 같이 계산하였다.

표 4. 실험 조건
Table 4. Experimental conditions.

해상도	영상	실험 조건
D1 (720×480)	football	Baseline 프로파일, QP={22, 28, 32, 38} CAVLC IPPP
	susie	
	popple	
CIF (352×288)	foreman	
	news	
	paris	

표 5. FFmpeg과 최적화한 디블록킹 필터의 성능 비교
Table 5. Performance comparison of optimized deblocking filter and FFmpeg.

해상도	영상	QP	Deblocking filter (thousand cycle)		ΔCycle (%)
			FFmpeg	Proposed	
D1	football	22	2,268,559	1,284,341	43.4
		28	2,262,452	1,261,904	44.2
		32	2,063,463	1,193,620	42.2
		38	1,839,567	985,551	46.4
	susie	22	1,832,378	1,085,117	40.8
		28	1,587,475	839,793	47.1
		32	1,450,628	713,840	50.8
		38	1,349,011	542,031	59.8
	popple	22	2,286,347	1,552,521	32.1
		28	1,553,350	820,174	47.2
		32	1,456,668	765,666	47.4
		38	1,334,853	630,353	52.8
	D1 평균				46.2
CIF	foreman	22	566,489	404,217	28.6
		28	485,012	312,627	35.5
		32	443,177	270,891	38.9
		38	400,905	206,782	48.4
	news	22	383,990	191,090	50.2
		28	358,950	158,425	55.9
		32	342,763	143,688	58.1
		38	325,774	119,516	63.3
	paris	22	454,230	274,609	39.5
		28	423,794	234,894	44.6
		32	394,105	229,591	41.7
		38	357,966	179,334	49.9
	CIF 평균				46.2
전체 평균					46.2

$$\Delta Cycle(\%) = \frac{C_{FFmpeg} - C_{method}}{C_{FFmpeg}} \times 100 \qquad (1)$$

표 5는 FFmpeg 복호화기의 디블록킹 필터 부분과 최적화한 디블록킹 필터와의 사이클을 측정하여 성능을 비교한 결과로써, 평균 1.8배의 속도 향상이 이루어진 것을 볼 수 있다. 영상 각각의 복잡도에 따라 속도 향상의 비율이 약간씩 다르긴 하지만, 영상의 해상도와 상관없이 전체적으로 비슷한 속도 향상이 이루어진 것을 알 수 있다.

일반적으로 QP가 높을수록 스킵 비율이 올라가므로 블록 경계강도가 0이 가장 많이 나온다. 따라서 QP가 높을수록 디블록킹 필터를 적용하지 않는 블록이 많아

지므로 속도가 향상된다. 본 논문에서 제안하고 있는 필터 연산 방법은 분기문을 제거하기 위해 연속된 4개 블록의 블록 경계강도가 모두 0이 아닌 경우를 제외한 모든 블록에 대해서 필터링을 취함에도 불구하고, 표 5의 실험 결과를 보면 QP가 높아질수록 속도 향상의 비율이 높아지고 있다. 그 이유는 필터 연산 내에서 분기문을 통해 블록 경계강도가 0일 경우 필터링을 취하지 않도록 하는 것보다 블록 경계강도 값에 상관없이 필터 연산의 필터링 루프를 소프트웨어 파이프라이닝을 통해 병렬화시키는 것이 더욱 효과적이라는 것을 알 수 있다. 또 다른 이유는 스킵 블록인 경우에 기존의 블록 경계강도 결정 조건에 비해 분기문이 줄어들었으며, 추가적인 분기문 없이 매크로블록의 내부일 경우에는 필터 연산을 하지 않도록 하고 있기 때문이다.

VI. 결 론

본 논문에서는 H.264/AVC 복호화기에서 계산상의 큰 비중을 차지하고 있는 디블록킹 필터를 TMS320C64x+ DSP에서 최적화하는 방법을 제안하였다. 최적화를 위해 두 가지 접근 방법으로 진행하였다. 첫 번째로 알고리듬 수준에서의 최적화를 위해 디블록킹 필터의 전체적인 구조상에서 조건 분기문과 메모리 접근을 최소화하였고, 아키텍쳐 수준에서의 최적화를 위해 필터 연산을 소프트웨어 파이프라이닝이 가능한 구조로 구현함으로써 루프의 반복간에 병렬성을 극대화시키는 방법을 사용하였다. 실험 결과 필터 연산 부분은 Z. Yang의 방법과 비교하여 휘도블록의 경우 약 14%의 좋은 성능을 보였고, 색차블록의 경우 약 56%의 좋은 성능을 보였다. 이는 기존의 방법에 비해 TMS320C64x+ DSP의 특징을 고려하였기 때문에 더 좋은 성능을 가져올 수 있었다. 그리고 전체적인 디블록킹 필터의 성능은 FFmpeg과 비교하였을 때 약 1.8배 속도가 향상되는 것으로 나타났다. 앞으로의 연구 방향은 DSP의 2단계 캐시 구조를 고려하고, DMA를 효율적으로 사용하여 메모리 접근을 더욱 줄임으로써 더욱 향상된 디블록킹 필터 구조를 갖도록 하는 것이다.

참 고 문 헌

[1] M. Horowitz, A. Joch, F. Kossentini and A. Hallapuro, “H.264/AVC baseline profile decoder complexity analysis,” *IEEE Trans. Circuits and Systems for Video Tech.* Vol. 13, no. 7, pp. 704-716, Jul. 2003.

[2] S. Y. Shih, C. R. Chang and Y. L. Lin, “A near optimal deblocking filter for H.264 advanced video coding,” *in Proc. Asia South Pacific Design Automation Conference*, pp. 170-175, Jan. 2007.

[3] B. Sheng, W. Gao and D. Wu. “An implemented architecture of deblocking filter for H.264/AVC,” *in Proc. of IEEE International Conference on Image Processing*, vol. 51, pp. 249-255, 2005.

[4] T. Song, Y. Hayashi and T. Shimamoto, “Fast deblocking filter implementation method for H.264/AVC,” *International Journal of Innovative Computing, Information and Control*, Vol. 5, no. 11, Nov. 2009.

[5] C. C. Cheng and T. S. Chang, “An in-place architecture for the deblocking filter in H.264/AVC,” *IEEE Transactions on circuits and Systems*, Vol. 53, no. 7, Jul. 2006.

[6] C. L. Hsu and Y. S. Huang, “A fast-deblocking boundary-strength based architecture design of deblocking filter in H.264/AVC applications,” Jounal of Signal Processing Systems, Vol. 52, no.3, pp. 211-229, Sept. 2008.

[7] J. Lou, A. Jagmohan, D. He, L. Lu and M. T. Sun, “Statistical analysis based H.264 high profile deblocking speedup,” *IEEE International Symposium on Circuits and Systems*, 2007.

[8] Z. Yang, W. Gao, Y. Liu and D. Zhao, “Deeply pipelined DSP solution to deblocking filter for H.264/AVC,” *IEEE transactions on Consumer Electronics*, Jul. 2006.

[9] Texas Instruments, “TMS320DM6446 digital media system-on-chip,” *SPRS283G*, Dec. 2005.

[10] Texas Instruments, “TMS320C64x/C64x+ DSP CPU and instruction set reference guide,” *SPRU732D*, Jul. 2007.

[11] 강대범, 황정우, 심동규, “TMS320C64x+를 이용한 MPEG-4 코덱 최적화,” *제 20회 신호처리합동학술대회논문지*, 1권, 158쪽, 2007년 10월

[12] 강대범, 심동규, 박호종, 심영석, “Davinci를 위한 Sorenson H.263 비디오 디코더 최적화,” *제 21회 신호처리합동학술대회논문지*, 21권, 1호, 155쪽, 2008년 9월

[13] 강대범, 심동규, “멀티미디어 DSP를 위한 AVS 비디오 복호화기 구현,” *전자공학회논문지*, 46권 SP편, 제5호, 151-161쪽, 2009년 9월.

[14] 이진섭, 서정한, 심동규, “SVC 디블록킹 필터의 DSP 최적화 구현,” *전자공학회하계학술대회논문*

지, 33권, 제1호, 301-303쪽, 2010년 6월.
[15] FFmpeg, “http://www.ffmpeg.org”, Mar. 2009.

저 자 소 개

이 진 섭(학생회원)
2007년 성결대학교 멀티미디어학과 학사 졸업
2009년～현재 광운대학교 임베디드소프트웨어학과 석사과정
<주관심분야 : 영상처리, 영상압축, DSP>

강 대 범(정회원)
2008년 광운대학교 컴퓨터공학과 학사 졸업
2010년 광운대학교 컴퓨터공학과 석사 졸업
2010년～현재 삼성전자 무선사업부 선행개발팀 연구원
<주관심분야 : 영상 신호처리, 영상압축, DSP, ASIP>

심 동 규(정회원)
1999년 서강대학교 전자공학과 공학박사
1999년～2000년 (주) 현대전자
2000년～2002년 (주) 바로비젼
2002년～2005년 Univ. of Washington
2005년～현재 광운대학교 컴퓨터공학과 (부교수)
<주관심분야 : 영상 신호처리, 영상 압축, 컴퓨터 비전>

이 수 연(정회원)
1983년 교토대학 공학 박사
1973년～현재 광운대학교 컴퓨터공학과 (정교수)
<주관심분야 : 정보공학 XML, 영상압축, 컴퓨터비젼>