석사학위논문

CPU 기반의 SIMD 명령을 사용한 실시간 최대휘소투영 가시화

2018년

한성대학교 일반대학원

정보시스템공학과

정보시스템공학전공

이 세 희

석사학위논문
지도교수 계희원

CPU 기반의 SIMD 명령을 사용한 실시간 최대휘소투영 가시화

Realtime maximum intensity projection using CPU-based SIMD instruction set

2018년 6월 일

한성대학교 일반대학원

정보시스템공학과

정보시스템공학전공

이 세 희

석사학위논문
지도교수 계희원

CPU 기반의 SIMD 명령을 사용한 실시간 최대휘소투영 가시화

Realtime maximum intensity projection using CPU-based SIMD instruction set

위 논문을 공학 석사학위논문으로 제출함

2018년 6월 일

한성대학교 일반대학원

정보시스템공학과

정보시스템공학전공

이 세 희

이세희의 공학 석사학위논문을 인준함

2018년 6월 일

심사위원장 _____(인)

심 사 위 원 _____(인)

심 사 위 원 _____(인)

국 문 초 록

CPU 기반의 SIMD 명령을 사용한 실시간 최대휘소투영 가시화

한 성 대 학 교 대 학 원
정 보 시 스 템 공 학 과
정 보 시 스 템 공 학 전 공
이 세 희

MIP 렌더링은 관찰 방향을 자주 바꿔가며 깊이감을 획득하므로 빠른 가시화 속도가 필수적이다. 의료 데이터는 수백, 수천 장의 영상 집합으로 아주 큰 데이터이기 때문에 일반적인 방법으로는 고속 가시화를 할 수 없다.

본 연구는 CPU의 병렬처리 연산인 SIMD의 AVX 명령어 집합을 사용하여 CPU 기반으로 대용량 볼륨 데이터의 실시간 MIP 가시화가 가능함을 보인다. 우리는 연속된 배열의 값을 여러 개 불러와 한 번의 명령만으로 연산을 수행하는 SIMD를 수행하기 위해 광선의 방향에 따라 볼륨을 기울여 볼륨과 영상을 평행하게 만들어 연산하는 shear-warp 분해 기법을 이용한 고속 MIP 렌더링을 제안한다. 볼륨은 대개 X, Y, Z 순으로 저장되어 있다.

SIMD를 활용하려면 순차적인 배열을 불러와야 한다. 하지만 볼륨의 관찰 방향에 따라 순차적으로 데이터에 접근하지 못하는 경우가 생긴다. 이를 해결하기 위한 전처리 과정으로 AVX 명령어를 사용한 고속 전치 방법을 제안한다. 이는 한 번에 연산하는 데이터 수가 많을 뿐 아니라 연산의 횟수를 줄이

면서 병목현상을 크게 완화했다. 행렬 전치(matrix transposition)는 일반적인 영상처리에서 흔히 사용되므로, 제안 방법은 볼륨 가시화 외에 다른 영상 처리에도 범용으로 적용 가능하다.

최근접 보간을 이용한 볼륨 렌더링은 빠르고 편하지만, 영상이 부드럽지 못하고 계단 현상이 발생한다. 이러한 화질 개선을 위해 삼차원 선형 보간을 수행한다. 삼차원 보간은 8개의 각 복셀마다 여러 번의 연산이 이루어지는데, shear-warp의 특성을 이용하여 각 복셀의 가중치를 미리 계산하여 연산 횟수를 줄이는 방법을 제안한다. 보간 연산에는 실제 복셀의 정숫값의 밀도와 실숫값의 가중치를 곱하는 연산이 필요하다. 하지만 본 논문에선 AVX2로 대부분의 연산을 수행하는데, 이는 정수와 실수의 곱셈 연산을 지원하지 않는다. 이를 해결하기 위해 실수의 가중치를 정수로 변환한 후 정수끼리의 AVX2 곱셈 연산을 수행하고 이후 원래 값을 얻는 방법을 제안한다. 또한, 삼차원 보간 시 하나의 픽셀을 보여주기 위해 주변 8개의 복셀을 접근하여 계산해야 하며 이때 메모리 접근에 대한 비효율이 발생한다. 이러한 성능 저하를 최소화하기 위해 볼륨 메모리에 접근하는 순서가 최대한 순차적이게 전처리 과정을 거친다. 관찰 방향에 따른 볼륨 세 벌을 렌더링 전에 미리 준비했으며 각 볼륨들은 렌더링과 보간에 필요한 복셀들의 거리를 최소화시켜 저장한다. 저장되는 영상에서 또한 연산된 값이 이미지에 저장될 때 이미지 픽셀 간 거리를 최소화하여 같은 위치의 배열에 접근하는 횟수가 많게 하면서 메모리 접근 효율을 극대화한다. 이를 확인하기 위한 새로운 알고리즘을 제안하며 성능 비교를 수행한다.

【주요어】 볼륨 렌더링, 영상처리, MIP, shear-warp, CPU 병렬처리, SIMD, AVX, 전치, max, 보간, 메모리 재배치

목 차

표 목 차

그 림 목 차

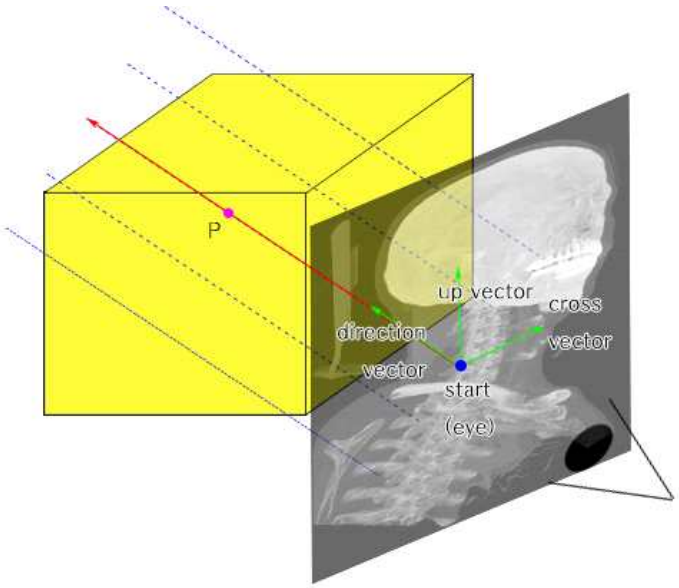
제 1 장 서 론

제 1 절 MIP

의료영상 시스템에서 볼륨 렌더링은 필수적이고 기본적인 렌더링 방법이다. 여러 장의 단면 영상이 높이 방향으로 쌓여있는 데이터인 볼륨 데이터를 여러 방향으로 관찰하며 사용자가 원하는 효과를 내어 영상을 생성하는 기법을 볼륨 가시화 혹은 볼륨 렌더링(Volume Rendering)이라고 한다(Sabella, 1988). CT나 MR로 얻은 수백 장의 인체 단면 영상들을 하나씩 관찰하는 것은 진단에 힘이 들기 때문에 볼륨 가시화를 통해서 진단에 도움이 되는 영상을 생성한다.

최대 휘소 투영(MIP : Maximum Intensity Projection) 볼륨 가시화는 삼차원 볼륨 데이터에서 가장 큰 밀도 값을 얻어와 화면에 출력하는 기법이다(Belina, 2009). 관찰자가 바라보는 방향으로 가상의 투사선을 만들어 볼륨 데이터를 통과시키고 이 투사선을 지나는 복셀(voxel)들의 값을 비교하여 가장 큰 값을 계산한다. 계산된 밀도 값들은 영상에 대응하는 위치에 밝깃값으로 저장되고 화면에 출력된다. CT 데이터에서 조영된 혈관이나 골격계는 x선 흡수도가 높기 때문에 큰 값을 갖고, 피부나 공기는 낮은 값을 갖게 된다(Kye, 2017). MIP를 통해 혈관의 구조나 뼈의 생김새를 명확하게 관찰할 수 있다(Belina, 2009).

예를 들어, [그림 1-1], [그림 1-2]와 같이 바라보는 위치에서의 투사선을 생성하고, 투사선에 대응하는 볼륨 데이터의 값을 비교하며 가장 큰 값인 230을 픽셀에 저장한다. CT 또는 MRI로 얻은 인체 단면 데이터 여러 장을 종합하면 3차원 볼륨 데이터로 얻을 수 있다. 이를 MIP 기법을 적용하면 인체 조직, 조영된 혈관, 골격을 관찰하기 좋은 영상을 얻을 수 있다.



[그림 1-1] 투사선 생성 및 MIP 데이터 추출 과정.

[그림 1-2] 투사선 생성 과정.

MIP 영상에는 깊이 정보가 없기 때문에, 깊이 단서(depth cue)를 얻기 위해 관찰방향을 자주 바꿔가며 진단하게 된다. 그러므로 지금까지의 연구에서는 빠르게 영상을 생성하기 위한 많은 노력이 있었다. 우선, 전처리 작업을 통해 최종 영상에 반영될 확률이 낮은 데이터를 사전에 제거하는 방법이 있다. Schreiner(1993)는 수 분의 전처리 작업을 수행하여 데이터를 밀도 값 순서로 미리 정렬하고 높은 값부터 투영(projection)하는 방법을 적용하는 고속 알고리즘을 제안하였다. Mroz(1999)는 수분의 전처리 과정으로 불필요한 데이터의 50% 이상을 제거하는 1% 손실 압축 방법을 적용하였다. Mroz(2000)는 관찰 방향에 따라 불필요한 데이터를 제거하고 데이터 블록인 cell을 밀도 값 순서로 정렬하여 cell projection 하였다.

제 2 절 연구 내용

MIP 영상의 생성 과정은 샘플링과 누적연산에서 대부분의 수행시간이 소요된다. 샘플링은 볼륨 데이터의 일정 간격마다 각 위치에서 값을 얻는 것이다. 샘플링 방법으로 최근접 화소 보간(Nearest Neighbor Interpolation)은 연산이 간단하지만, 화질이 나쁘기 때문에 일반적으로 잘 알려진 선형 보간(Linear Interpolation)을 사용한다(Mroz, 2000). 샘플링 과정의 효율을 향상시키고 캐시 메모리의 활용을 향상시키기 위해 object order 렌더링 기법을 적용하는 방법도 연구되었다. 삼차원 샘플링의 비용을 절감하기 위해 Fang(2002)은 shear-warp 기법을 적용하였고, Peker (2003)는 이차원 샘플링을 적용하였다. Mora(2005)는 이론적인 분석과 함께 object order ray casting을 사용한 사용 예를 들었고, Kiefer(2006) 역시 Object order ray casting을 사용하며 광선의 진행 방향을 경사진 방향으로 하여 캐시 효율을 향상시켰다.

이러한 기존 연구는 일관된 흐름으로 정리할 수 있다. 즉, 영상 데이터를 블록으로 구분한 다음, 오랜 전처리 시간을 들여 불필요한 블록을 제거한다(Mroz, 1999). MIP는 렌더링 순서에 무관하게 동일한 영상을 얻기 때문에, 데이터를 내림차순으로 정렬하고 그 순서로 렌더링을 수행한다(Schreiner, 1993; Mroz, 2000; Peker, 2003). 따라서 마지막 순서의 어두운 값의 데이터를 렌더링에서 제거될 확률이 높아진다. 각 블록은 블록 기반 광선 투사법 (Mora, 2005; Kiefer, 2006)을 이용하여 블록 단위로 연산이 수행하므로 캐시 메모리의 효율이 향상된다.

이러한 노력에도 불구하고 대부분의 연구는 수 분의 오랜 전처리가 필요한 단점이 있으며, 가시화 속도가 대화적 속도에 미치지 못하고, 2fps 정도의 수준에 그쳤다.

최근 GPU의 성능 향상으로 GPU를 이용한 볼륨 가시화 연구가 증가하고 있다. 초기 3D Texture와 같은 GPU 기능을 사용한 가시화는 워크스테이션에서 가능하였으나(Dachille, 1998), 곧 상용 GPU에서 가능해졌다(Rezk-Salama, 2000). GPU를 사용하는 것은 다음과 같은 단점이 존재한다.

먼저 알고리즘 개발 및 적용 과정에서 CUDA(Mensmann, 2010)나 DirectX(Vollrath, 2005)와 같은 특별한 라이브러리 설치 및 개발 방법이 필요하다. 그리고 완성된 애플리케이션을 병원 등 사용자가 활용할 때, 적합한 하드웨어가 설치되어 있어야 한다. 마지막으로 의료영상 데이터가 GPU에 적재되어야 하는데, GPU 메모리의 크기는 CPU에 비해 작기 때문에 4D-CT와 같은 대용량의 데이터에는 직접 볼륨 렌더링(Direct Volume Rendering)이 불가능(Zhao, 2014)하다는 문제가 있다.

이러한 관련 연구들의 단점을 극복하기 위해 CPU의 SIMD 기능을 이용하여 연산 병렬화가 수행되었다(Kye, 2009). 이 방법은 전혀 전처리 과정 없이 CPU 병렬처리를 활용한 방법이다. MIP에서는 최대 밝기를 계산하기 위해 밝깃값의 크기 비교 명령이 매우 많이 사용되는데, 이 명령은 CPU에서 조건 분기명령으로 해석되어 매우 비효율적이다. SIMD에서는 크기 비교를 고속의 산술연산으로 처리하므로 큰 속도 향상을 얻는다. 부가적으로, 한 번에 여러 데이터의 비교가 동시에 일어나므로 그 성능 향상은 더욱 커진다. 그 결과 GPU 없이 CPU만으로 대화적 속도의 가시화가 가능하였으나, 관찰 방향에 따라 속도의 편차가 매우 크다는 단점이 있다.

본 연구는 범용 CPU 기반의 고속 MIP 알고리즘을 제안한다. 고속 선형 보간 샘플링과 누적을 최신 CPU에서 SIMD 기법인 AVX2를 이용하여 수행한다. 제안 방법은 shear-warp 분해 기법이 인접한 샘플들의 가중치가 동일한 점에 착안하여, 동일한 가중치로 여러 데이터를 동시에 샘플링하는 효율적인 방법을 제안한다. shear-warp를 기반으로 입력 의료영상 데이터와 출력 MIP 영상 데이터의 메모리 접근을 모두 순차적으로 수행하도록 좌표계 변환을 수행하였다. 그 결과 메모리 접근의 효율이 향상된다. 그리고 MIP에서 가장 많이 사용되는 크기 비교 연산을 AVX의 고속 병렬 산술연산으로 대체한다. 한 번의 연산으로 256bit의 데이터를 처리할 수 있어 가시화 속도가 향상된다.

상기 연구 결과를 종합하면, 제안 방법은 상용 영상처리 라이브러리와 기존 연구에 비해 매우 빠르며, GPU 없이도 임상 시험을 실시간 가시화로 원활하게 수행할 수 있는 수준이다. 본 연구는 특별한 하드웨어가 필요하지 않

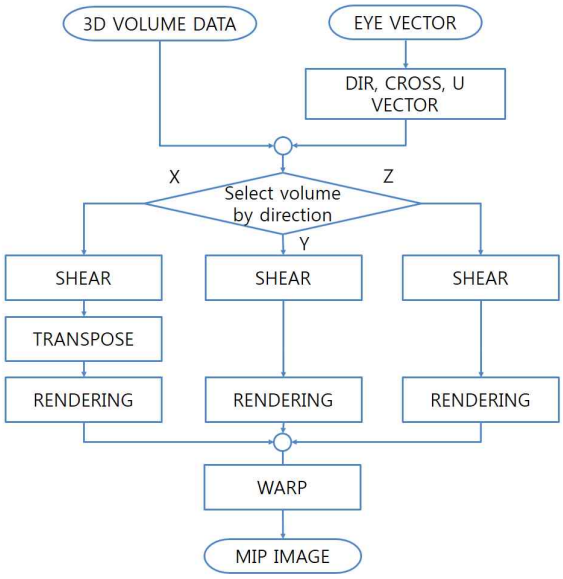
아 노트북, 데스크톱 등 어느 하드웨어에서나 적용할 수 있는 장점이 있다.

본 논문에서는 입력된 하나의 볼륨만으로 MIP를 수행하면서 행렬 전치 연산을 효율적으로 수행하는 방법을 제안한다. 이때 메모리 접근 효율을 향상하는 대신 특정 관찰 방향(sagittal view)에서 행렬 전치 연산이 필요하다. 제안 방법은 16×16 크기 행렬을 64회의 연산으로 행렬 전치 연산을 수행하는 방법을 개발하였다. 본 연구는 SIMD를 이용한 n×n의 전치 연산이 최소 $n\log_2 n$회의 연산이 필요함을 증명하고 SIMD의 버전 별로 관찰 방향에 따른 속도 변화를 경감한다.

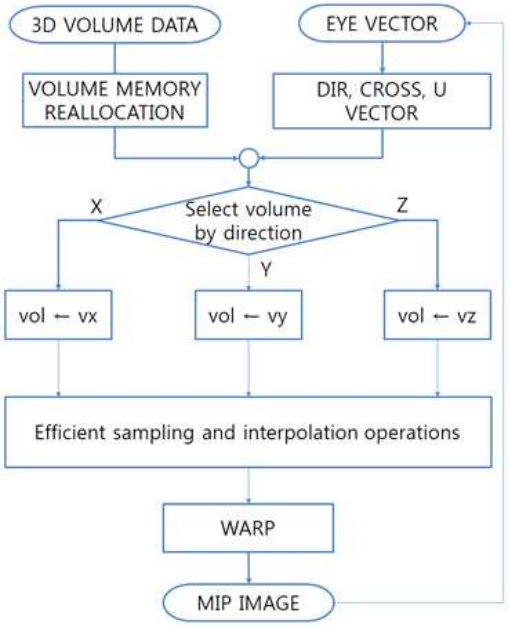
그리고 화질 개선을 위해 삼차원 보간을 수행한다. 이는 한 번의 샘플링 시 1개의 복셀 값만 얻어 오는 최근접 보간에 비해 삼차원 보간은 8개의 복셀의 값을 얻어오므로 성능 저하가 발생한다. 그래서 좀 더 효율적인 볼륨 렌더링을 위해 세 벌의 볼륨을 가지고 연산하는 방법을 제안한다. 이때, 볼륨 데이터는 용량이 크므로, 연산 속도 못지않게 메모리 참조의 효율을 향상하는 것도 중요하다. 본 연구는 볼륨 가시화 과정에서 메모리 참조의 효율성을 개선하는 새로운 메모리 재배치 방법을 제안한다. 제안 방법은 관찰 방향에 따라 메모리 참조의 지역성을 높여서 더 빠른 속도로 영상의 생성이 가능하다.

제 3 절 논문 구성

2장에서는 본 연구의 기반인 SIMD 연산과 shear-warp를 이용한 MIP의 구현 방법에 대해 간략하게 설명한다. 3장에서는 한 벌의 볼륨으로 AVX를 이용하여 고속 MIP를 수행하는 것을 제안하며, 4장에서는 세 벌의 볼륨으로 고속 선형 보간 MIP를 제안한다. 5장은 실험 결과로 제안 방법들을 증명하고 마지막 6장에서 결론과 향후과제를 제시한다.



[그림 1-3] 고속 MIP의 시스템 개요도.



[그림 1-4] 고속 선형 보간 MIP의 시스템 개요도.

제 2 장 관련 연구

이 단락에서는 제안 연구의 기본이 되는 SIMD 연산과 이를 이용한 MIP 알고리즘을 설명한다. 그리고 shear-warp를 소개하며 병렬 가시화에 대해 설명한다.

제 1 절 SIMD

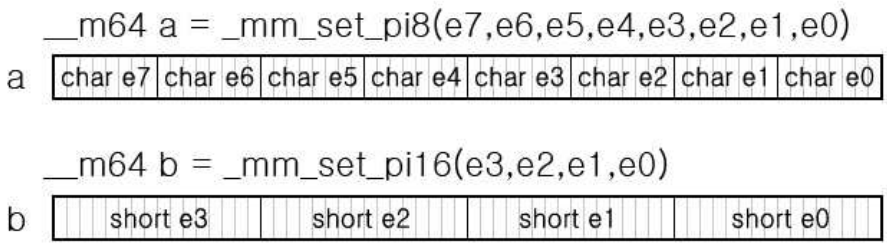
SIMD는 하나의 명령어를 여러 데이터에 대해 동시에 병렬로 수행할 수 있는 기술 또는 하드웨어 구조를 말한다. 상용 CPU에서는 인텔이 1997년 펜티엄 용으로 SIMD 확장 명령 세트인 MMX(MultiMedia eXtension)를 개발하였다. 이후 1999년에는 펜티엄3부터 16바이트 용량인 SSE (Steaming Simd Extensions)가 사용 가능해졌다. SSE는 부동 소수점 계산의 병렬 수행이 가능하였고, 정수 데이터 처리는 SSE2가 개발되는 2001년에 가능해졌다. 2008년에 도입된 AVX(Advanced Vector Extensions)는 32바이트를 한 번에 다룰 수 있으며 부동 소수점 연산을 처리가 가능하다. 2011년 인텔의 샌디브릿지 CPU부터 사용 가능한 AVX2 는 32바이트 단위로 정수데이터를 병렬처리할 수 있다.

본 연구는 정수 형인 볼륨 데이터를 다루기 위하여 AVX2 명령어 집합을 이용한다. 2011년 출시된 인텔의 2세대 코어 CPU부터 사용 가능하므로 현존하는 대다수의 컴퓨터에서 별도의 장비 없이 본 연구를 적용할 수 있다.

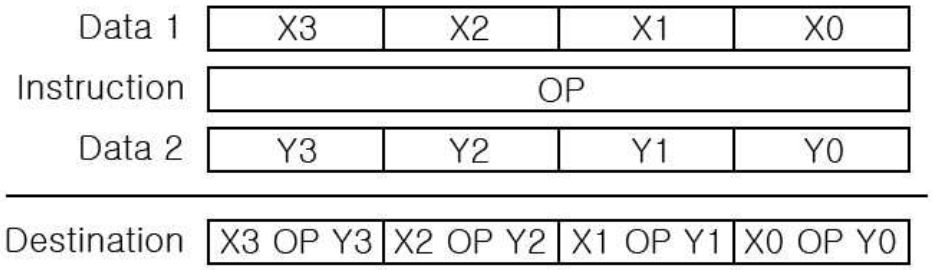
1) SIMD의 사용

MMX는 64bit 크기의 데이터를 이용할 수 있다. SIMD 명령어는 사용자가 원하는 비트 단위로 자유롭게 나누어 사용할 수 있다. 예를 들어, 16bit의 short 변수들을 MMX로 프로그래밍하면 4개의 값을 동시에 연산할 수 있으며 8bit인 character 변수들은 8개의 값을 동시에 연산할 수 있다. 이러한 병

렬 연산으로 한 번에 많은 연산을 수행할 수 있어서 SIMD 연산은 속도가 빠른 장점이 있다.

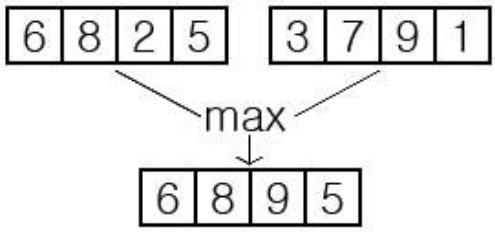


[그림 2-1] 하나의 64 bit 변수에 여러 데이터를 입력하는 SIMD 명령.



[그림 2-2] SIMD를 이용한 멀티 데이터 연산.

또한 SIMD의 max 연산을 적용하면 두 쌍의 값들을 비교하여 더 큰 값들만을 한 번에 추출할 수 있다. 일반적으로 오랜 시간이 소요되는 조건 분기(conditional branch) 명령을 산술연산으로 대체할 수 있어서 더욱 효율적이다. 이렇게 SIMD 연산을 MIP에 적용하면 한 번에 여러 데이터의 최댓값을 얻을 수 있고, 비효율적인 조건 분기를 제거할 수 있어서 효과적이다.



[그림 2-3] MMX를 이용한 max 연산.

단, SIMD 연산을 효율적으로 수행하는 조건이 있다. 먼저, 데이터가 메모리상에 연속적으로 존재해야 한다. [그림 2-3]에서 데이터값 6, 8, 2, 5가 차례대로 메모리에 존재해야 한다. 만약 데이터가 메모리에 흩어져 있다면 이를 하나의 레지스터에 옮기는 과정에서 큰 비효율이 발생한다. 두 번째 제한은 원소 단위로만 병렬 연산이 가능한 것이다. 위의 예제에서 6과 3, 8과 7중 큰 값을 계산하는 수직연산(vertical operation)은 효율적이나 하나의 레지스터에 속한 값인 6, 8, 2, 5 중 가장 큰 값 8을 얻는 수평 연산(horizontal operation)은 매우 효율이 낮다. 이 두 제약을 만족하며 MIP를 구현하는 방법이 2.1.2절에 소개되어 있다.

2) MMX를 이용한 MIP 구현

본 절에서는 좌표계의 변환을 이용하여 MMX를 사용한 MIP 구현에 대해 설명한다. 우선, 볼륨 데이터의 좌표축을 X, Y, Z축, 영상의 좌표축을 i, j, k축으로 가정한다. 물리적인 데이터는 볼륨 데이터는 X축, 영상데이터는 i축을 따라 순차적으로 배열되어 있다고 가정한다. 관찰자의 관찰 방향은 k축이 된다. 만약 관찰 방향 k축이 볼륨 데이터의 좌표축 Z와 정확하게 일치한다면 MMX를 이용하여 MIP를 구현할 수 있다. 우선 좌표계 변환을 정리한다. Z축이 k 축에 대응되므로, 볼륨의 X축은 이미지의 i축에, 볼륨의 Y축은 이미지의 j축에 대응시킨다. 그러면 순차적으로 존재하는 X축 방향의 데이터 4개의 복셀은, 순차적으로 존재하는 영상의 i축 4개의 픽셀에 1:1로 대응된다. 이제 연속한 4개 복셀 데이터를 한 번에 MMX 레지스터에 읽어, 이를

영상에 연속된 4개의 픽셀과 비교하고 갱신하면 비교/분기 명령어 없이 한 번에 4개 복셀의 처리가 완료된다. 이 과정을 각 Y축과 Z축 방향으로 반복하면 영상이 생성된다.

[표 2-1] 알고리즘 / MMX를 이용한 Z축 기준 MIP

```
1:  FOR each slice (Z)
2:    FOR each scanline (Y)
3:      get IMAGE_POS from (Y,Z)
4:      get VOL_POS from (Y,Z)
5:      FOR X += 4
6:        IMAGE_POS+X <- MAX( VOL_POS+X,  IMAGE_POS+X)
```

비슷하게 관찰자가 볼륨 데이터의 Y축 방향으로 관찰할 때는 다음과 같이 좌표계를 변환한다. Y축이 영상의 깊이 방향인 k축과 대응되므로, 볼륨의 X축과 Z축은 각각 이미지의 i축과 j축과 대응되도록 하면, X축과 i축이 대응되어 앞과 마찬가지로 효율적이 된다. 그러나 X축 방향으로 관찰 시엔 비효율이 발생하게 된다. 볼륨 데이터의 X축 방향으로 연속한 데이터 4개를 읽으면, 영상에서는 깊이 k 방향으로 대응되므로, 효율적인 i축에 순차적으로 저장할 수 없다. 만약 shear-warp를 사용하지 않고 광선 투사법(SabelIa, 1988)을 적용하여 k축으로 진행하는 하나의 광선을 병렬 처리하려고 할 수도 없다. k축 4개의 값 중 가장 큰 값을 한 픽셀에 저장하는 것은 수평 연산이므로 효율이 낮기 때문이다.

이를 해결하기 위한 방법으로 관련 연구(Kye, 2009)에서는 4×4 크기의 타일 단위로 행렬 전치 연산을 수행한다. X, Y축 방향으로 4×4 복셀을 MMX 레지스터 4개에 담고, 이를 전치 행렬을 구하는 연산으로 X축과 Y축을 변경하면, X축이 i축과 대응시킬 수 있다. 즉, 수평 연산을 수직 연산으로 변환하여 SIMD 명령을 적용할 수 있다.

이때, 4×4타일 단위로 데이터를 전치하는 행렬 전치 연산이 필요한데, 전체 연산시간 상당히 많은 부분을 차치하고 있어서 개선이 필요하다. 본 연구

는 AVX2를 이용하여 16×16 타일 단위로 한 번에 데이터를 전치하는 방법을 제안하고 있다. 기존 연구에서 행렬 또는 영상의 전치는 integer나 float 단위로 8×8 타일의 전치는 연구되었으나(McFarlin, 2011) 우리는 16×16 타일을 전치한다.

제 2 절 shear-warp

볼륨 데이터에서 샘플링을 수행하기 위해서는 우선 볼륨 메모리상의 위치를 계산해야 한다. shear-warp 기법은 볼륨 데이터의 메모리를 순차적으로 읽어 주소 계산이 간단하게 한다. 복셀의 크기와 pixel의 크기가 동일하며, 볼륨 데이터의 직교축에 평행한 중간 이미지(intermediate image)를 사용하여, 볼륨 메모리를 순차적으로 읽은 결과를 중간 이미지에 순차적으로 기록할 수 있다. 완성된 중간 이미지는 이차원 영상처리(warp)를 수행하여 최종 영상을 생성한다(Lacroute, 1994; Fang, 2002). shear-warp의 특징은 다음과 같다.

① 주소 계산이 간단하여 효율이 높다.

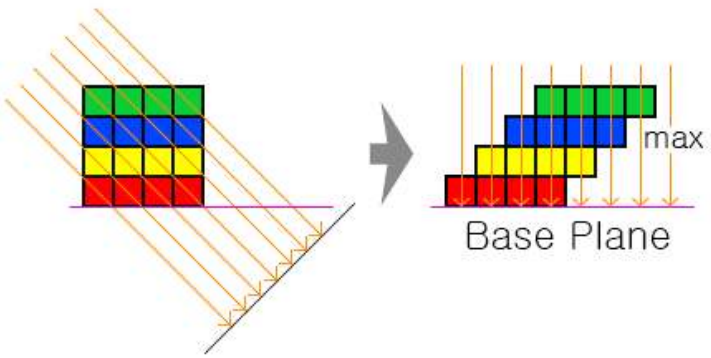
② 볼륨 데이터를 순차적으로 읽으면 영상데이터의 위치도 순차적으로 접근하게 된다.

③ shear-warp에서의 보간은 보간 가중치(weight)가 슬라이스(slice)마다 동일하다.

본 연구는 AVX2기능을 이용하여 shear-warp 렌더링을 발전시켜 더 높은 성능을 제공한다.

관찰자는 기울어진 방향으로 영상 데이터를 관찰하는 경우가 일반적이다. 본 연구는 shear-warp 분해 기법을 사용해서 가시화를 수행한다. shear-warp 기법은 관찰자의 관찰 방향이 볼륨 데이터와 비스듬하게 위치하는 경우, 좌표계를 두 단계로 나누어 변경하는 방법이다. 우선, 볼륨 좌표계를 영상 좌표계로 대응하는 관찰 행렬을 M_{view}라고 가정한다.

$$\begin{bmatrix} Img_x \\ Img_y \\ Img_z \\ 1 \end{bmatrix} = \begin{bmatrix} & M_{view} & \\ & & \end{bmatrix} \begin{bmatrix} Vol_x \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix} \quad (1)$$



[그림 2-4] 투사선을 기준으로 한 볼륨 shearing.

관찰 방향이 z축과 나란하도록 좌표계를 shearing 하는 변환을 적용한다. 그때의 새로운 좌표계에서는 [그림 2-4]와 같이 볼륨 데이터가 기울어진 형태로 위치하게 된다.

$$\begin{bmatrix} Temp_x \\ Temp_y \\ Temp_z \\ 1 \end{bmatrix} = \begin{bmatrix} \\ M_{shear} \\ \\ \end{bmatrix} \begin{bmatrix} Vol_x \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix} \qquad (2)$$

새로운 좌표계에서는 각 볼륨 데이터의 슬라이스는 평행하게 미끄러져(shearing) 중간 이미지 Temp에 대응된다. 관찰 방향에 따른 시작 지점을 계산하면, 병렬 연산을 수행하여 영상을 생성할 수 있다. 이렇게 생성된 영상은 shearing 된 영상이므로 최종 출력 영상은 2차원 영상처리를 수행하여 보정해야 하며 이를 warping이라고 한다. 식 3과 같이 M_{view}행렬과 M_{shear}^{-1}행렬의 곱을 이용하여 Temp 영상을 입력으로 하여 최종 출력 영상을 이차원 영상처리를 수행할 수 있다.

$$\begin{bmatrix} Img_x \\ Img_y \\ Img_z \\ 1 \end{bmatrix} = \begin{bmatrix} \\ M_{view} \\ \\ \end{bmatrix} \begin{bmatrix} \\ M_{shear} \\ \\ \end{bmatrix}^{-1} \begin{bmatrix} \\ M_{shear} \\ \\ \end{bmatrix} \begin{bmatrix} Vol_x \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix} \quad (3)$$

$$= \begin{bmatrix} \\ M_{view} \\ \\ \end{bmatrix} \begin{bmatrix} \\ M_{shear} \\ \\ \end{bmatrix}^{-1} \begin{bmatrix} Temp_x \\ Temp_y \\ Temp_z \\ 1 \end{bmatrix}$$

제 3 절 병렬 가시화

볼륨 가시화에서 하나의 픽셀의 계산은 다른 픽셀의 계산에 영향을 주지 않기 때문에 병렬화할 수 있는 여지가 많아 이를 이용하는 다양한 방법이 연구되었다.

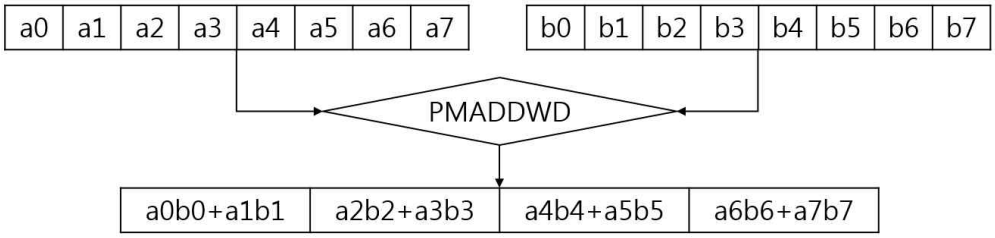
게임 등에 널리 사용되는 그래픽스 처리장치는 볼륨 가시화 연산에 적용할 수 있다(Dachille, 1998; Rezk-Salama, 2000). OpenGL 또는 DirectX의 shader 프로그램을 사용하거나, OpenCL 또는 CUDA 등의 라이브러리를 사용하여 고속으로 볼륨 가시화를 수행할 수 있으나, 하지만 상대적으로 추가 비용이 필요하며 사용자 시스템에 적합한 하드웨어가 설치되어 있어야 한다는 점과 메모리 용량이 CPU에 비해 부족한 점이 단점이다(Zhao, 2014).

CPU만을 이용한 방법으로는, Xeon Phi와 같은 60 core 프로세서(Hofmann, 2014), Xeon Westmere EX 40 core, 80 threads(Treibig, 2013)를 이용하여 CT reconstruction과 같은 병렬 처리를 대규모로 수행 가능하다. 그러나 서버급 CPU를 사용해야 해서 비용문제가 있다.

CPU의 SIMD 기능을 사용하면 추가 비용 없이 효율적으로 영상 처리를 수행(Chen, 2012)할 수 있다. 특히 AVX기능을 이용하면, 256bit의 레지스터(Zekri, 2014)를 이용할 수 있다. Treibig(2013)은 AVX를 이용하여 CT 역투영(back projection)을 구현하여 GPU와 비교하였고 결과는 상용 하드웨어가 고도로 조정된 GPU 구현과 경쟁할 수 있다고 주장하였다. CPU SIMD의 효율적인 수행을 위한 전제 조건은 효율적 메모리 참조와 연산 방법이다(Kye,

2017).

효율적으로 메모리를 참조하려면, 같은 데이터를 비슷한 시기에 여러 번 읽거나, 데이터를 물리적 메모리 순서에 순차적으로 읽어야 한다. SIMD 연산은 [그림 2-2]에서 볼 수 있듯이 수직적 계산 명령이며, 수평 한 배열에 들어있는 값들 중에 가장 큰 값을 얻는 것은 매우 비효율적이다. Chen(2012)은 CT reconstruction에서 보간을 수행하기 위해 PMADDWD연산을 사용하였다. SIMD를 이용하여 효율적이나, 이차원 보간을 하려면 a0b0+a1b1와 a2b2+a3b3를 더해야 하므로 수평 연산을 추가로 수행하고, 세로로 배치되어 있는 복셀을 가로로 재배치하는 작업을 수행해야 한다. 그리고 곱셈 결과에서 데이터 요소의 정밀도가 2배가 되므로 병렬화 정도가 떨어진다. 그리고 반올림 연산이 되지 않아 정밀도가 약간 감소한다.



[그림 2-5] PMADDWD 연산.

제 3 장 고속 MIP

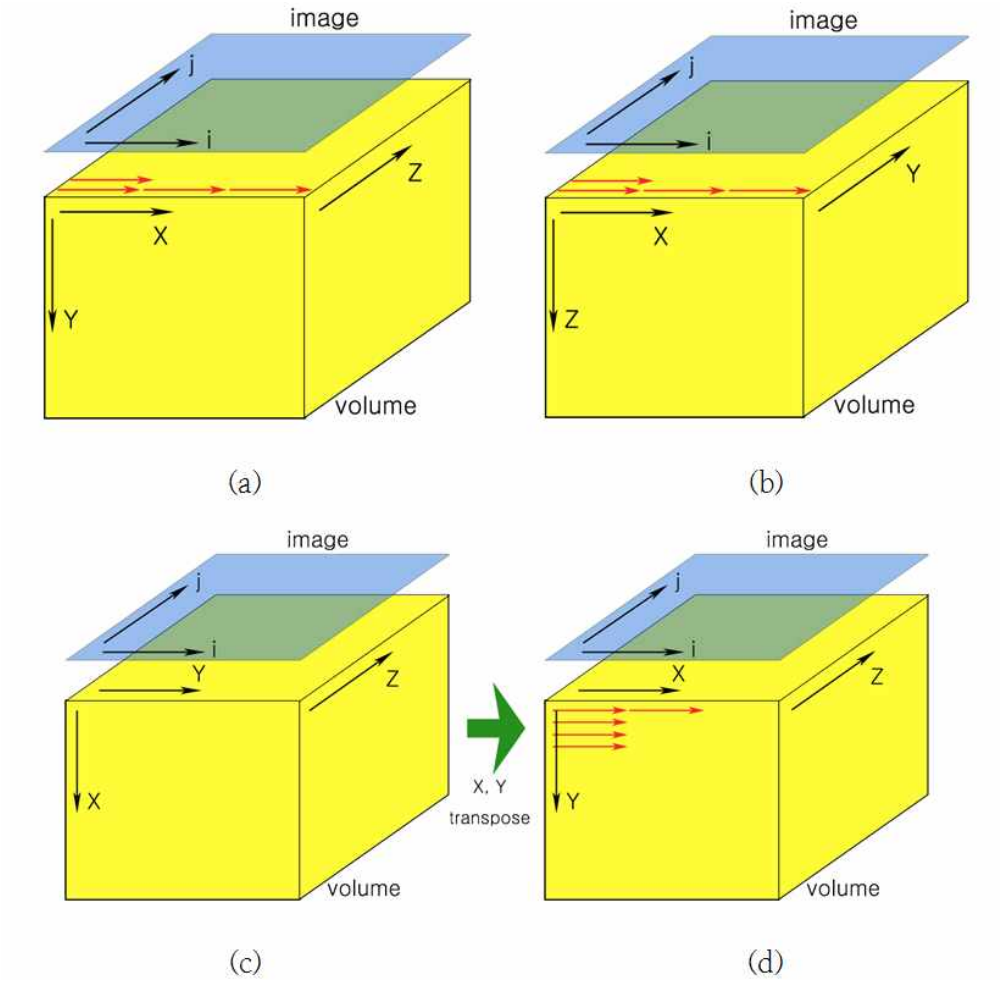
제 1 절 다양한 관찰 방향에 따른 렌더링 방법

볼륨은 크게 X축, Y축, Z축 방향에서 관찰할 수 있다. 하지만 SIMD는 한번 데이터를 읽고 쓸 때 배열의 연속된 값을 다룰 수 있어서 X축의 값들을 차례대로 불러오는 것이 자연스럽다.

본 절에서는 2장 1절의 2)와 마찬가지로 볼륨 데이터의 좌표축을 X, Y, Z축 영상의 좌표축을 i, j, k축으로 가정한다.

Y축 방향으로 관찰하는 경우, 볼륨의 X축과 영상의 i축이 메모리 순서대로 배열되어 있으므로 볼륨 데이터의 X축과 영상의 i 축이 대응되도록 하는 것이 효율이 높다. 따라서 영상의 i축을 볼륨의 X축으로, 영상의 j축을 볼륨의 Z축으로 대응하고 Y축 방향으로 1씩 증가하며 연산한다. 마찬가지로 Z축 방향의 관찰에서는 볼륨의 X축과 영상의 i축을 대응시키고 볼륨의 Y축과 영상의 j 축을 대응시켜 Z축 방향으로 증가시키며 연산한다.

X축 방향으로 관찰하는 경우, Y축과 Z축을 먼저 대응시켜 계산한 이후에 X축 방향으로 1씩 증가해야 하므로 X축이 i축에 대응될 수 없다. 그러므로 X축을 순서대로 읽을 수 없는 X축 방향의 관찰에 문제가 생긴다. 이를 해결하기 위한 방법으로 X축과 Y축을 기준으로 모든 복셀들을 전치한다.



[그림 3-1] 관찰 방향별 이미지 대응 및 볼륨 데이터 읽는 순서 비교: (a) Y축 방향으로 관찰, (b) Z축 방향으로 관찰, (c)X축 방향으로 관찰, (d)X축 방향 관찰을 위한 X, Y축 전치.

이렇게 좌표계 변환을 수행하면 항상 볼륨의 X축과 Temp 영상의 i축이 1 : 1 대응이 되기 때문에 좌표 계산식이 다음과 같이 간단해진다. 즉, 영상 좌표는 vol_x=0일 때의 좌표를 계산해두고 vol_x값을 더하면 영상 좌표를 얻을 수 있다. 매번 좌표를 계산하는 행렬 곱셈 연산 대신 덧셈 연산 한 번으로 좌표 계산이 종료된다.

$$\begin{bmatrix} Temp_x \\ Temp_y \\ Temp_z \\ 1 \end{bmatrix} = \begin{bmatrix} \\ M_{shear} \\ \\ \end{bmatrix} \begin{bmatrix} Vol_x \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix} \tag{4}$$

$$= \begin{bmatrix} \\ M_{shear} \\ \\ \end{bmatrix} \begin{bmatrix} 0 \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix} + \begin{bmatrix} \\ M_{shear} \\ \\ \end{bmatrix} \begin{bmatrix} Vol_x \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} Temp_x | \, Vol_x = 0 \\ Temp_y \\ Temp_z \\ 1 \end{bmatrix} + \begin{bmatrix} Vol_x \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

이를 정리하면 다음과 같다.

Y축 방향의 관찰할 땐 X축 기준의 스캔 라인을 16바이트마다 위치를 계산하여 max 연산을 하고 한 스캔 라인을 모두 연산하면 Z축 방향으로 추가해가며 연산을 한다. 한 면을 모두 연산하면 Y축 방향으로 1씩 증가하여 연산하면 된다.

X축 방향의 경우에는 X축 기준의 스캔 라인을 16바이트마다 전치한 후 전치된 타일을 바탕으로 위치를 계산하여 max 연산을 하게 된다. 한 타일의 연산이 끝나면 X축 방향으로 확장하고 Y축 방향으로 넓혀서 연산한다. 한 면이 연산을 마치면 Z축 방향으로 전진하며 계산한다.

[표 3-1] 알고리즘 / AVX2를 이용한 Y축 기준 MIP

```
1:  FOR Y
2:    FOR Z
3:      GET IMAGE_POS
4:      GET VOL_POS
5:      FOR X += 16
6:        IMAGE_POS+X <- MAX(VOL_POS + X,
                                    IMAGE_POS + X)
```

[표 3-2] 알고리즘 / AVX2를 이용한 X축 기준 MIP

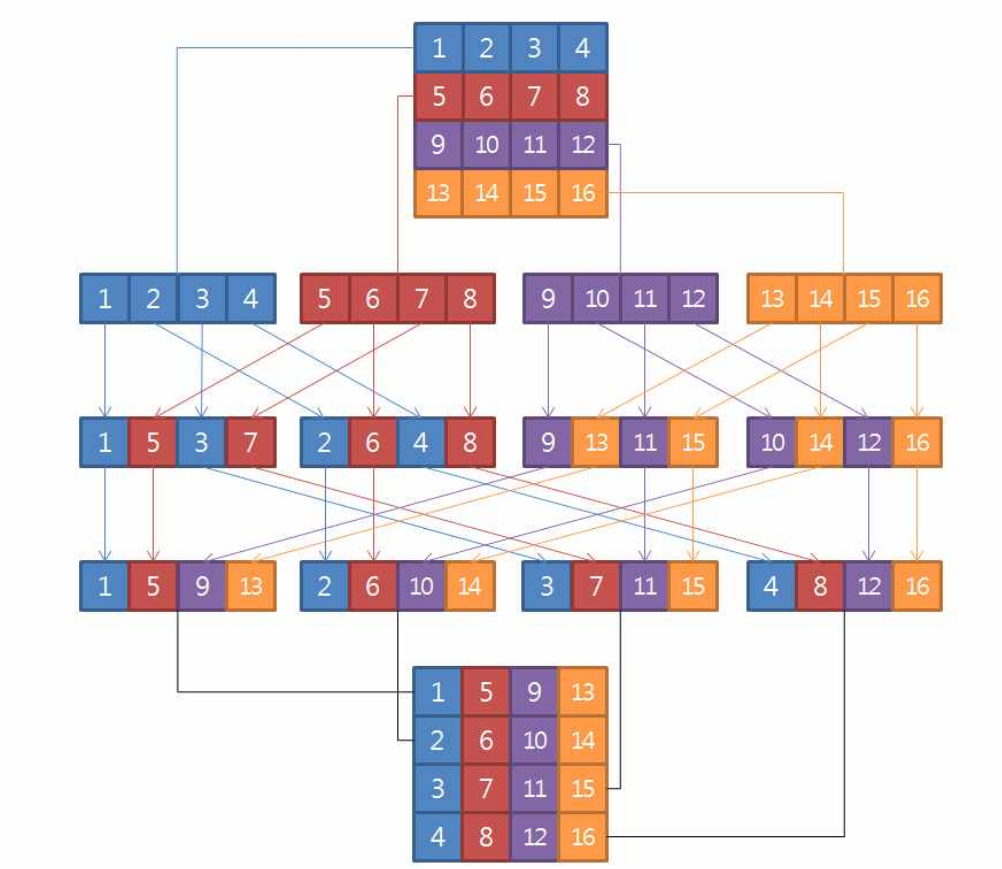
```
1:  SET TILE_SIZE
2:  FOR Z
3:    FOR Y += TILE_SIZE
4:      FOR X += TILE_SIZE
5:        MAKE TILE
6:        TRANSPOSE(EACH_TILE)
7:        FOR T to TILE_SIZE
8:          GET   IMAGE_POS
9:          GET   TILE_POS
10:         IMAGE_POS <- MAX(TILE_POS,
                              IMAGE_POS+Y+TILE_SIZE)
```

한편, 전치 연산은 X축으로 관찰할 때만 발생하므로, 관찰자는 X축으로 관찰할 때 부자연스럽게 느려진다고 느끼게 된다. 이 문제를 해결하려면, 전치 연산의 효율을 높여야 한다.

제 2 절 고속 행렬 전치

관찰자가 sagittal 방향에서 데이터를 관찰할 경우 전치 연산이 부차적으로 수행되어야 한다. 이에 따라, 사용자는 갑작스러운 성능 저하를 느끼게 되고 전체 성능이 제한된다. 따라서 최대한 행렬 전치 연산을 최적화하는 방법이 필요하다.

본 연구는 AVX2를 이용하여 n=16일 때 $16\times\log_2 16=64$회의 연산으로 전치 연산을 수행하는 방법을 보인다. 그리고 이 방법이 최소 횟수 연산임을 아래에서 증명한다.



[그림 3-2] SIMD를 이용한 4×4 2차원 배열 전치.

예를 들어, 하나의 8바이트 MMX 레지스터에는 4개의 복셀이 저장될 수 있으므로, 기존 4×4 크기의 행렬은 [그림 3-2]처럼 4개의 레지스터가 필요하다. 4개의 레지스터에 입력된 값이 1번 레지스터에 {1,2,3,4}, 2번 레지스터에 {5,6,7,8}, 3번 레지스터에 {9,10,11,12}, 4번 레지스터에 {13,14,15,16} 이라면 전치 연산을 통해 최종적으로 1번 레지스터에 {1,5,9,13}, 2번 레지스터에 {2,6,10,14}, 3번 레지스터에 {3,7,11,15}, 4번 레지스터에 {4,8,12,16}를 얻어야 한다. SIMD의 데이터 교환 연산은 두 레지스터를 입력으로 다양한 조합을 통해 하나의 출력 결과를 생성한다. 1, 5, 9, 13은 각각 4개의 서로 다른 레지스터에 흩어져 있으므로 한 번의 연산으로는 결과를 얻을 수 없다. 그러므로 최소 두 번의 연산이 필요하며, 실제 [그림 3-2]를 보면 중간 단계인

{1, 5, 3, 7}, {9, 13, 11, 15}을 생성한 후에 {1, 5, 9, 13}을 얻을 수 있다. 각 출력 레지스터를 생성하는데 두 단계의 연산이 사용되어 총 연산 횟수는 8회이다.

4개의 레지스터에 흩어진 값을 하나의 레지스터로 모으는데 최소 두 단계의 연산이 필요하므로, 만약 8×8의 타일을 전치 연산을 하려면 8개의 레지스터에 흩어진 값을 모아야 한다. 이는 두 단계로 가능하지 않으므로 최소 세 단계의 연산이 필요하다. 4개 흩어진 값을 모은 결과 레지스터를 두 개 모아 다시 한 번 조합 연산을 수행해야 8개 흩어진 값을 모을 수 있기 때문이다. 마찬가지로, 16×16의 타일이라면 네 단계의 연산이 필요하다.

이를 일반화하면, 각 출력 레지스터는 n개의 입력 레지스터에 흩어진 값을 모아야 하므로, 최소한 $\log_2 n$회의 연산을 필요로 한다. 따라서 n×n 행렬의 전치 연산을 수행하려면 최소 $n \log_2 n$의 연산이 필요하다는 것을 증명하였다.

이렇게 n^2개의 픽셀에 n log n 회의 연산이 필요하므로, 각 픽셀에 대한 오버헤드를 고려하면 $ov(n) = \frac{n\log_2 n}{n^2} = \frac{\log_2 n}{n}$ 이다. 이 값은 n에 대해서 단조 감소하고 $\lim_{n\to\infty} ov(n) = \lim_{n\to\infty} \frac{\log_2 n}{n} = 0$ 이다. 따라서 한 번에 더 큰 레지스터를 사용할수록 n이 증가하고 오버헤드가 감소하여 상대적인 전치 연산의 오버헤드는 0으로 수렴하게 된다.

먼저 SSE2를 이용한 8×8 크기의 타일의 전치 연산부터 설명한다. SSE2는 하나의 16바이트 SSE 레지스터에 8개의 복셀을 저장할 수 있고 8×8 복셀 타일을 저장하기 위해 8개의 레지스터가 동시에 사용된다. 총 반복연산은 3회가 적용되어 $8 \times \log_2 8 = 24$회의 명령으로 전치 연산을 수행할 수 있다. 아래는 SSE2로 구현한 전치 연산 함수와 각 연산을 시행했을 때 데이터 변화의 예를 보인다.

[표 3-3] 알고리즘 / 8×8 전치

```
1:  TRANSPOSE_8x8(VA[0:127], VB[0:127],  VC[0:127], VD[0:127],
                  VE[0:127], VF[0:127], VG[0:127], VH[0:127])
2:  {
3:    SET TA[0:127], TB[0:127],  TC[0:127], TD[0:127], TE[0:127],
        TF[0:127], TG[0:127], TH[0:127];
4:    // OPERATION A
5:    TA[0:127] <- UNPACK(VA[0:15],  VB[0:15], VA[16:31],
        VB[16:31], VA[32:47], VB[32:47], VA[48:63], VB[48:63])
6:    TB[0:127] <- UNPACK(VA[64:79],  VB[64:79], VA[80:95],
        VB[80:95], VA[96:111], VB[96:111], VA[112:127],
        VB[112:127])
7:    TC[0:127] <- UNPACK(VC[0:15],  VD[0:15], VC[16:31],
        VD[16:31], VC[32:47], VD[32:47], VC[48:63], VD[48:63])
8:    TD[0:127] <- UNPACK(VA[64:79],  VB[64:79], VA[80:95],
        VB[80:95], VA[96:111], VB[96:111], VA[112:127],
        VB[112:127])
9:    TE[0:127] <- UNPACK(VE[0:15],  VF[0:15], VE[16:31],
        VF[16:31], VE[32:47], VF[32:47], VE[48:63], VF[48:63])
10:   TF[0:127] <- UNPACK(VE[64:79],  VF[64:79], VE[80:95],
        VF[80:95], VE[96:111], VF[96:111], VE[112:127],
        VF[112:127])
11:   TG[0:127] <- UNPACK(VG[0:15], VH[0:15],  VG[16:31],
        VH[16:31], VG[32:47], VH[32:47], VG[48:63], VH[48:63])
12:   TH[0:127] <- UNPACK(VG[64:79],  VH[64:79], VG[80:95],
        VH[80:95], VG[96:111], VH[96:111], VG[112:127],
        VH[112:127])
```

```
13:    //    OPERATION B
14:    VA[0:127] <-    UNPACK(TA[0:31], TC[0:31], TA[32:63],
           TC[32:63])
15:    VB[0:127] <-    UNPACK(TA[64:95], TC[64:95], TA[96:127],
           TC[96:127])
16:    VC[0:127] <-    UNPACK(TB[0:31], TD[0:31], TB[32:63],
           TD[32:63])
17:    VD[0:127] <- UNPACK(TB[64:95], TD[64:95],    TB[96:127],
           TD[96:127])
18:    VE[0:127] <-    UNPACK(TE[0:31], TG[0:31], TE[32:63],
           TG[32:63])
19:    VF[0:127] <- UNPACK(TE[64:95],    TG[64:95], TE[96:127],
           TG[96:127])
20:    VG[0:127] <-    UNPACK(TF[0:31], TH[0:31], TF[32:63],
           TH[32:63])
21:    VH[0:127] <- UNPACK(TF[64:95],    TH[64:95], TF[96:127],
           TH[96:127])

22:    //    OPERATION C
23:    TA[0:127] <-    UNPACK(VA[0:63], VE[0:63])
24:    TB[0:127] <-    UNPACK(VA[64:127], VE[63:127])
25:    TC[0:127] <-    UNPACK(VB[0:63], VF[0:63])
26:    TD[0:127] <- UNPACK(VB[64:127],    VF[63:127])
27:    TE[0:127] <-    UNPACK(VC[0:63], VG[0:63])
28:    TF[0:127] <-    UNPACK(VC[64:127], VG[63:127])
29:    TG[0:127] <-    UNPACK(VD[0:63], VH[0:63])
30:    TH[0:127] <-    UNPACK(VD[64:127], VH[63:127])
31: }
```

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

(a)

1	9	2	10	3	11	4	12
5	13	6	14	7	15	8	16
17	25	18	26	19	27	20	28
21	29	22	30	23	31	24	32
33	41	34	42	35	43	36	44
37	45	38	46	39	47	40	48
49	57	50	58	51	59	52	60
53	61	54	62	55	63	56	64

(b)

1	9	17	25	2	10	18	26
3	11	19	27	4	12	20	28
5	13	21	29	6	14	22	30
7	15	23	31	8	16	24	32
33	41	49	57	34	42	50	58
35	43	51	59	36	44	52	60
37	45	53	61	38	46	54	62
39	47	55	63	40	48	56	64

(c)

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

(d)

[그림 3-3] SSE2를 이용한 8×8 transpose 과정:
(a)ORIGINAL, (b)AFTER OPERATION A,
(c)AFTER OPERATION B, (d)AFTER OPERATION C.

본 연구는 이를 확장하여 AVX2를 이용한 16×16타일 전치 연산 방법을 개발하였다. (구체적인 내용은 부록을 참고하시오) 이렇게 AVX2는 32바이트 크기로 16 복셀을 저장할 수 있고, 16×16 타일을 전치하기 위해 16개 레지스터를 동시에 사용하였다. 전치 연산을 수행하기 위해, 명령을 4단계 수행하면, $16 \times \log_2 16 = 64$회의 명령으로 16×16 타일의 전치 연산이 가능하다.

만약 레지스터 크기가 작은 MMX로 16×16 타일의 전치 연산을 수행한다고 가정하면, 4×4의 타일을 총 16회 전치하여 16×16 타일을 생성해야 한다. 이는 8(명령) × 16(회) = 128회이므로, 본 연구는 AVX2연산을 이용하여 두 배 적은 명령을 달성하였다. 게다가 본 제안 방법은 한 번에 대량의 메모리를 다루므로 기존 방법보다 메모리 접근의 효율이 높아서 추가의 성능 향상이 발생한다.

제 3 절 AVX2를 이용한 최댓값 계산과 렌더링

AVX를 이용하면 32바이트 데이터를 한 번에 처리할 수 있게 되었다. 의료 영상 볼륨 데이터는 한 복셀이 2바이트이므로 한 번에 16개의 복셀을 처리할 수 있어서 적은 명령어의 호출로 영상을 획득할 수 있다. 한 번에 32바이트, 즉 16개 복셀을 동시에 처리하므로, 512×512의 일반적인 영상 크기를 고려했을 때, 512 ÷ 16 = 32번의 반복으로 하나의 스캔 라인이 처리된다. SIMD를 적용하지 않으면 512번의 비교 연산이 필요한 데 비해 본 연구는 32번의 max 연산으로 종료된다. 또한, SIMD를 사용하지 않으면 if 조건문을 사용해야 하는데 이는 비용이 많이 들어가는 연산이다. SIMD를 사용하면 MAX 연산을 사용함으로써 조건문이 사라져 추가적인 성능 향상이 발생한다.

[표 3-4] 알고리즘 / SIMD를 적용하지 않은 일반적인 방식의 메모리 접근

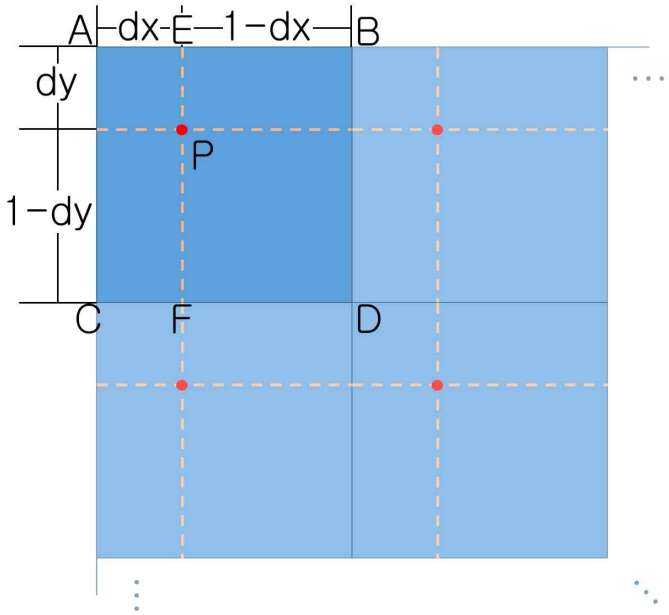
```
1:  FOR(X=0; X<VOL_X; X+=1)
2:     IF VOL_POS + X > IMAGE_POS + X
3:        IMAGE_POS + X <- VOL_POS + X
```

[표 3-5] 알고리즘 / SIMD AVX2에서의 메모리 접근

```
1:  FOR(X=0; X<VOL_X; X+=16)
2:     IMAGE_POS+X <- MAX( VOL_POS + X,  IMAGE_POS + X)
```

제 4 장 고속 선형 보간 MIP

제 1 절 보간



[그림 4-1] 가중치가 일정한 경우의 이차원 보간.

본 장에선 삼차원 보간을 수행하며 이를 설명하기 위해 먼저 이차원 보간을 설명한다. [그림 4-1]과 같이 인접한 4개의 복셀 A, B, C, D 사이의 샘플링 위치 P에서의 값을 구하기 위해선 샘플과 복셀 간의 거리 dx, dy를 가중치로 삼아, 식 5와 같이 계산할 수 있다. 이 경우, 6번의 곱셈 연산을 사용한다.

$$E = (1-dx)\times A + dx\times B$$
$$F = (1-dx)\times C + dx\times D$$
$$P = (1-dy)\times E + dy\times F \qquad (5)$$

본 연구에서 사용하는 shear-warp 방식은 각 슬라이스에 대해 dx, dy가 동일한 성질이 있다(Lacroute, 1994). 본 연구는 Fang(2002)의 연구와 같이

가중치를 미리 계산하여 반복적으로 재사용한다. 식 5를 전개하고, (1-dx)×(1-dy), dx×(1-dy), (1-dx)×dy, dx×dy를 각각 weightA, weightB, weightC, weightD로 치환하면 식 6과 같이 네 번의 곱셈으로 줄게 된다.

$$\begin{aligned} P &= (1-dx)\times(1-dy)\times A + dx\times(1-dy)\times B \\ &\quad + (1-dx)\times dy\times C + dx\times dy\times D \\ &= weightA\times A + weightB\times B + weightC\times C + weightD\times D \end{aligned} \quad (6)$$

[표 4-1] 알고리즘 / 한 슬라이스에서의 이차원 보간

```
1:  get dx, dy
2:  wA <- (1-dx) × (1-dy)
3:  wB <- dx × (1-dy)
4:  wC <- (1-dx) × dy
5:  wD <- dx × dy
6:  FOR Y =0 to HEIGHT
7:    FOR X =0 to WIDTH
8:      P <- Data[y][x]×wA + Data[y][x+1]×wB + Data[y+1][x]×wC
            + Data[y+1][x+1]×wD
```

1) AVX2를 이용한 고속 보간

본 연구에서 제안하는 고속 보간 방법을 위와 마찬가지로 2차원 보간에 대해 우선 설명하고, 이후 삼차원 보간으로 확장한다. 의료영상에서 복셀의 값은 2바이트의 정수형이고, 가중치 값들은 실수형이지만 결괏값은 정수형으로 출력되어야 한다. AVX2에서는 정수와 실수의 곱셈을 직접 수행하는 명령이 없다. 따라서 가중치를 정수형으로 변환하여 정수형 사이의 곱으로 연산하는 것이 일반적이다. 예를 들어, 가중치는 0~1까지의 실수 가중치에 적당한 값 32768(=2^{15})을 곱해 0~32768 범위의 정수로 만들어 곱을 수행한 후, 나중에 32768로 나누어(또는 15bit shift 연산) 최종값을 얻는다. 식 6의 첫 번

째 항을 자료형을 고려하여 다시 쓰면 식 7과 같다. 식 7에서 나누기 또는 shift 연산을 수행하기 전에 반올림(round-off)을 수행하면 정밀도를 향상할 수 있다.

$$\begin{aligned} weight_{float} \times A_{int} &= weight_{float} \times 32768 \times A_{int} \div 2768 \\ &\approx (weight_{float} \times 32768)_{int} \times A_{int} \rangle\rangle_{(round-off)} 15 \end{aligned} \quad (7)$$

본 연구는 _mm256_mulhrs_epi16(Intel) 명령을 활용하여, 효율적 보간을 수행한다. [표 4-2]는 _mm256_mulhrs_epi16 명령을 자세하게 풀어쓴 것이다. 우선, MULHRS16은 두 16bit 정수를 곱한 결과에 15bit shift를 수행하되, 결과의 14bit에서 반올림을 수행한다. 본 연구에서는 MULHRS16에 로 대입하므로 결과는 식 7과 같다.

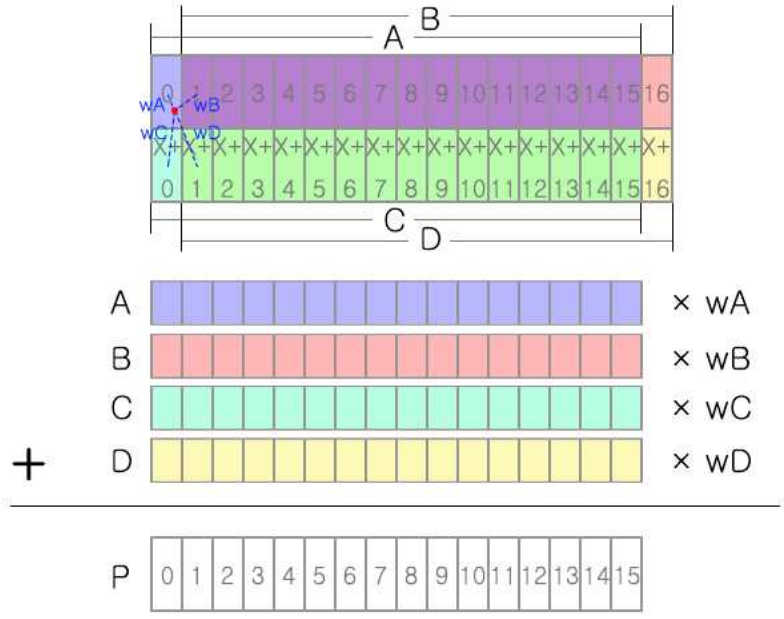
[표 4-2] 알고리즘 / _mm256_mulhrs_epi16

```
MULHRS16 (x, y) := ((x *  y) >> 14 + 1) >> 1

_mm_mulhrs_epi16  (X[0:15], Y[0:15]) :=
1:  clear DST[0:15]
2:  FOR i=0 to 15
3:    DST[i] <- MULHRS16(X[i], Y[i])
```

그리고 _mm256_mulhrs_epi16는 MULHRS16를 16회 병렬적으로 수행하므로, 나란한 16개 점에 대해서 식 6의 weightA×A 결과를 얻을 수 있다. 이제 마찬가지로 식 6의 B, C, D에 대해서도 같은 식을 적용한 후, 결괏값을 _mm256_add_epi16을 이용하여 16개씩 더하면 16개의 픽셀에 대한 보간이 완료된다. AVX를 이용하여 수정된 보간은 [표 4-3]이다. [표 4-3]의 일곱 번째 줄에서 보듯이 한 번에 16개의 샘플을 연산하므로 [표 4-1]과 비교하면 매우 효율적이다. 이를 정리하면 [그림 4-2]와 같다.

[표 4-3] 알고리즘 / 한 슬라이스에서 AVX를 이용한 이차원 보간

```
1:  get dx, dy using viewing direction
2:  wA[0:15] <- (1-dz) × (1-dx) × (1-dy) × 32768
3:  wB[0:15] <- dx × (1-dy) × 32768
4:  wC[0:15] <- (1-dx) × dy × 32768
5:  wD[0:15] <- dx × dy × 32768
6:  FOR Y =0 to HEIGHT
7:    FOR   X =0 to WIDTH, X <- X + 16
8:      A[0:15] <- Load Data[z][Y][X:X+15]
9:      B[0:15] <- Load Data[z][Y][X+1:X+16]
10:     C[0:15] <- Load Data[z[Y+1][X:X+15]
11:     D[0:15] <- Load Data[z][Y+1][X+1:X+16]

12:     P[Y][X] <- ADD(ADD(MULHRS(wA,A),   MULHRS(wB,B)),
                       ADD(MULHRS(wC,C),   MULHRS(wD,D)))
```



[그림 4-2] AVX를 이용한 이차원 보간.

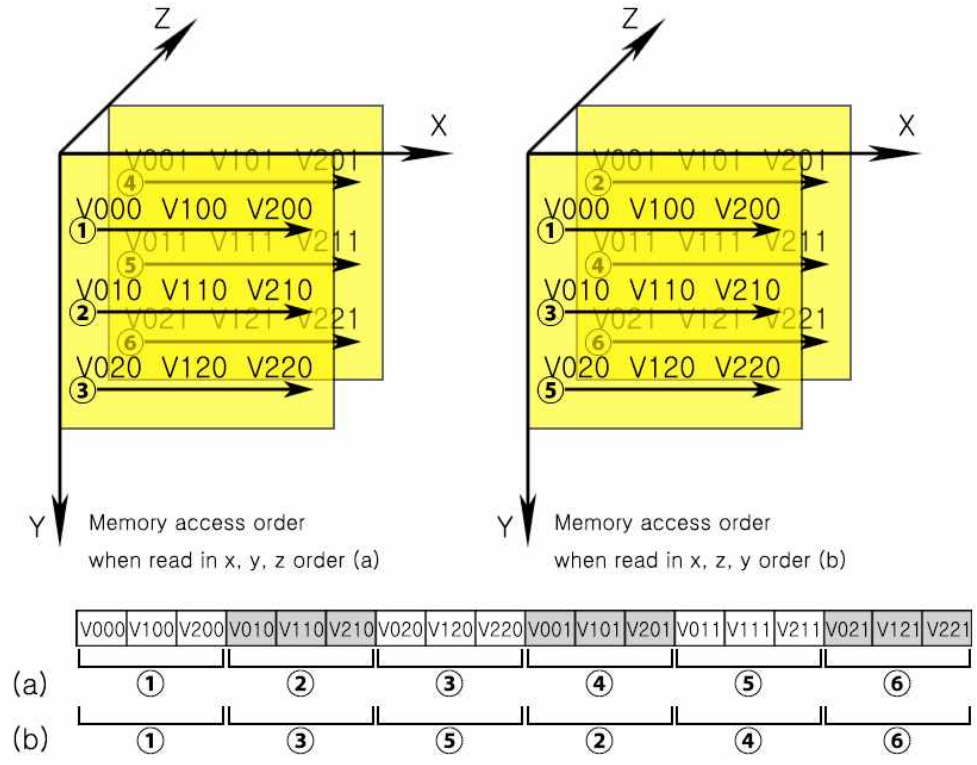
이를 삼차원으로 확장하면, 보간을 위한 가중치 값이 8개가 되는 것만 차이가 난다. 그 결과는 [표 4-4]와 같다.

[표 4-4] 알고리즘 / 한 슬라이스에서 AVX를 이용한 삼차원 보간

```
1:  get dx, dy, dz using viewing direction
2:  wA[0:15] <- (1-dx) × (1-dy) × (1-dz) × 32768
3:  wB[0:15] <- dx × (1-dy) × (1-dz) × 32768
4:  wC[0:15] <- (1-dx) × dy × (1-dz) × 32768
5:  wD[0:15] <- dx × dy × (1-dz) × 32768
6:  wA'[0:15] <- (1-dx) × (1-dy) × dz × 32768
7:  wB'[0:15] <- dx × (1-dy) × dz × 32768
8:  wC'[0:15] <- (1-dx) × dy × dz × 32768
9:  wD'[0:15] <- dx × dy × dz × 32768
10: FOR Y =0 to HEIGHT
11:     FOR X =0 to WIDTH, X <- X + 16
12:         A[0:15] <- Load Data[z][Y][X:X+15]
13:         B[0:15] <- Load Data[z][Y][X+1:X+16]
14:         C[0:15] <- Load Data[z[Y+1][X:X+15]
15:         D[0:15] <- Load Data[z][Y+1][X+1:X+16]
16:         A'[0:15] <- Load Data[z+1][Y][X:X+15]
17:         B'[0:15] <- Load Data[z+1][Y][X+1:X+16]
18:         C'[0:15] <- Load Data[z+1][Y+1][X:X+15]
19:         D'[0:15] <- Load Data[z+1][Y+1][X+1:X+16]
20:         P[Y][X] <- ADD(ADD(ADD(MULHRS(wA,A),
                       MULHRS(wB,B)), ADD(MULHRS(wC,C),
                       MULHRS(wD,D))), ADD(ADD(MULHRS(wA',A'),
                       MULHRS(wB',B')), ADD(MULHRS(wC',C'),
                       MULHRS(wD',D'))))
```

제 2 절 메모리 재배치

세 벌의 데이터를 사용하는 Shear-warp 볼륨 가시화는 중간 이미지와 볼륨 데이터의 접근 방향이 나란하도록 메모리가 배치된다. 그리고 연산 순서는 [그림 4-3](a)와 같이 관찰자가 바라보는 방향에서 가까이 존재하는 1, 2, 3번 주사선을 순서대로 읽어 가시화하고, 멀리 있는 4, 5, 6번 주사선을 나중에 연산한다. 이것은 직접 볼륨 가시화의 경우 알파 블렌딩(alpha blending)을 수행하므로 front-to-back 투영이 올바르기 위함이다. 그러나 MIP와 같은 최댓값 계산은 데이터의 비교 순서와 결과가 무관하므로 렌더링 순서를 [그림 4-3](b)와 같이 변경해도 출력 영상에 차이가 없다. 이에 착안하여 본 연구는 더 높은 메모리 참조 효율성을 갖는 새로운 방법을 제안한다.



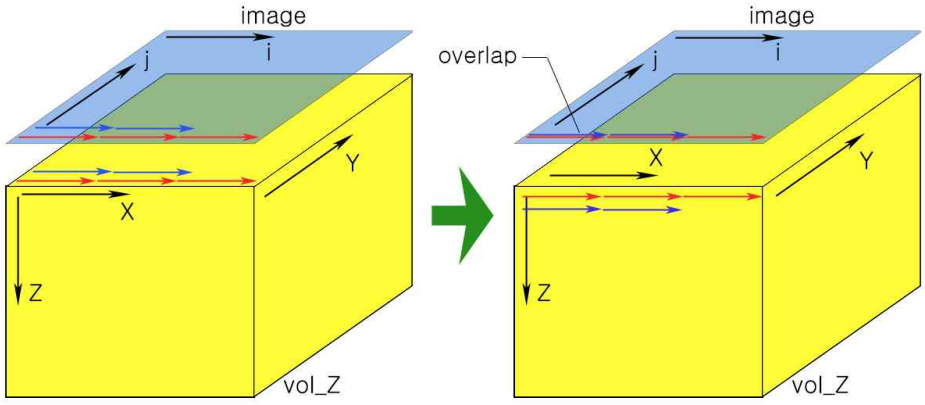
[그림 4-3] 메모리 접근 순서 비교:
(a) X-Y-Z 순서 (b) X-Z-Y 순서.

1) 메모리 접근 순서에 따른 속도 비교

메모리 접근의 순서에 따라 전체 시스템 효율은 크게 달라진다. Z 방향으로 관찰하는 경우 볼륨 데이터는 [그림 4-3]과 같이 메모리에서 1차원 배열로 저장되어 있으므로, [그림 4-3](a)와 같이 X, Y, Z 순으로 읽으면(X → Y → Z) 1차원 배열을 처음부터 끝까지 순서대로 읽도록 효율적으로 설계되어 있다. 반면 [그림 4-3](b)와 같이 X, Z, Y 순으로 읽게 되면(X → Z → Y) 1차원 배열 내에서 참조 지점이 계속해서 먼 거리를 옮겨 다니는 비효율이 발생하게 된다.

[표 4-5] 메모리 접근 순서에 따른 차이

For Z axis For Y axis For X axis Image <- max(image, vol[z][y][x])	For Y axis For Z axis For X axis Image <- max(image, vol[z][y][x])
X → Y → Z 순서 렌더링	X → Z → Y 순서 렌더링

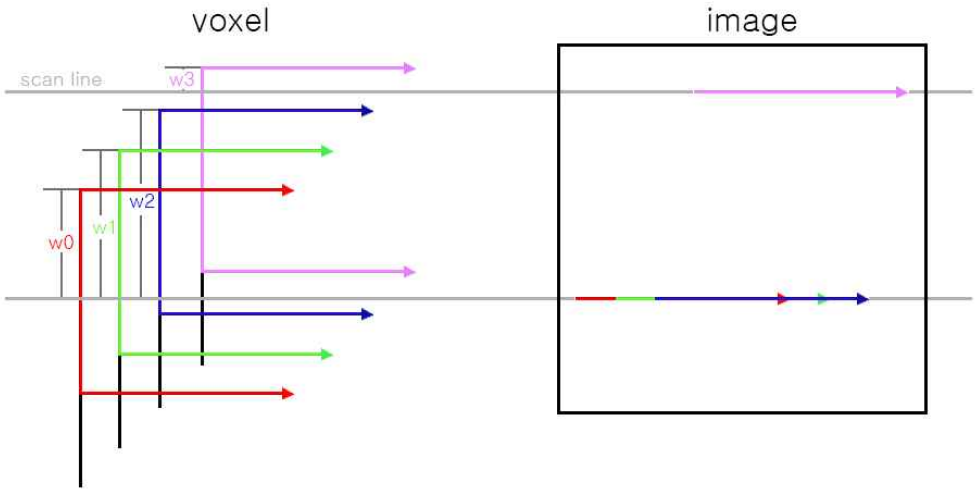


[그림 4-4] 볼륨에서의 메모리 접근 순서 변경에 따른 이미지 배열 접근.

한편, 볼륨을 읽어 계산한 후에 이미지에 누적을 수행할 때의 메모리 참조

순서도 효율에 영향을 준다. [그림 4-4]는 Z축 방향으로 관찰하고 있다. X → Y → Z 순으로 계산하고 영상에 누적한다면, Y가 증가하는 경우 영상에 대응하는 j 좌표도 증가한다. 그러나 X → Z → Y 순으로 읽어서 이미지에 저장하면 Z가 증가하여도 j 좌표가 변화 없으므로 영상에서 중첩(overlap)된다. 이때 영상에서는 이미 한번 접근하여 기록한 메모리 영역을 재사용하므로, 캐시 메모리의 효율이 향상된다.

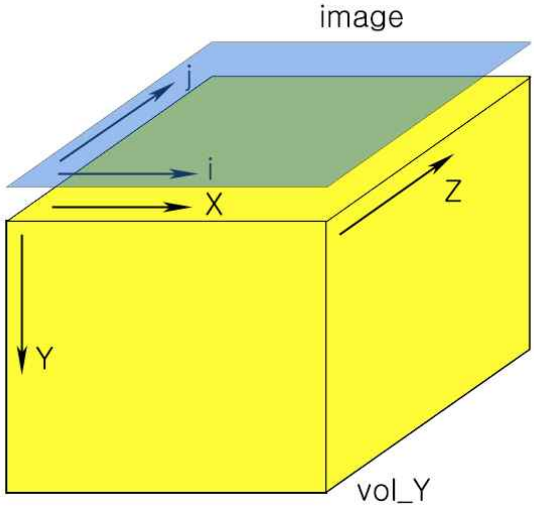
관찰 방향이 정확한 Z축 방향에서 약간 기울어지면, 오버랩(overlap) 되는 횟수가 달라진다. [그림 4-5]는 X → Z → Y 순으로 렌더링하는 경우, 이미지 오버랩을 자세히 보여주고 있다. 우선 두 줄의 빨간 스캔 라인을 읽어 가중치 w0를 알고리즘 3-2의 dy로 삼아 section 3과 같이 보간하여 출력 스캔 라인을 만들고 오른쪽 이미지에 누적한다. 다음으로 Z축 방향으로 한 칸 전진하여 데이터를 읽으면 그 위치는 초록 스캔 라인에 위치한다. 가중치 w1을 dy로 삼아 마찬가지로 보간하면, 가중치는 다르지만, 오른쪽 이미지와 같이 출력 메모리의 위치는 중첩된다. 다시 Z 방향으로 전진하여 파란 스캔 라인의 경우에도 중첩이 일어나며, 그다음 보라 스캔 라인의 출력은 다른 스캔 라인이 되어 중첩되지 않는다. 이렇게 중첩이 일어나는 횟수가 많을수록 직전 스캔 라인에서 참조한 출력 영상 배열 위치를 재참조하므로 효율이 증가하게 된다.



[그림 4-5] 근접한 위치에 있는 복셀들을 이미지 스캔 라인에 덮어쓰는 법.

위의 두 가지를 종합하면 볼륨과 이미지 모두 가시화 반복 순서에 따라 메모리 참조 효율이 달라진다는 사실을 알 수 있다. X → Y → Z 순으로 읽으면 볼륨에서의 메모리 접근은 순차적이 되므로 빠르지만, 이미지에서 메모리 접근은 상대적으로 느리다. 반면에, X → Z → Y 순으로 읽으면 볼륨의 메모리 접근은 부분적으로만 순차적이 되어 비효율적이지만 이미지에서 메모리 접근은 같은 좌표를 재참조할 확률이 증가하므로 매우 효율적이 된다. 다음 절에서는 볼륨 메모리 참조와 영상 메모리 참조가 모두 효율적인 방법을 제안한다.

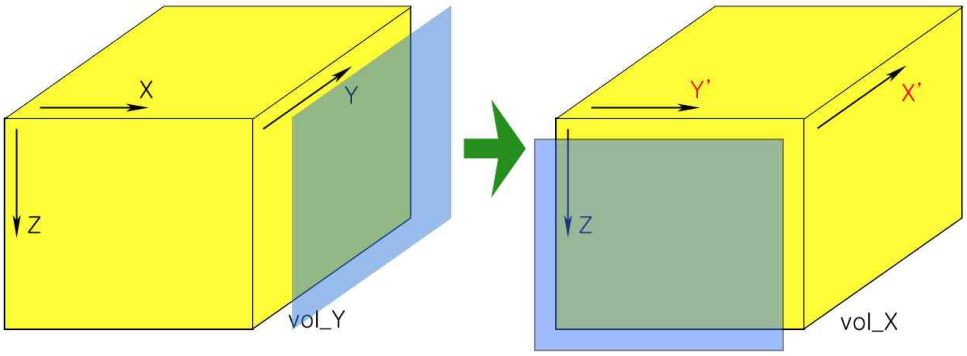
2) 축 별 메모리 배치 변경



[그림 4-6] 효율적인 Y축 방향의 볼륨 관찰.

관찰 방향이 Y축일 때 볼륨 데이터의 메모리 배치를 바꾸고 X → Z → Y 순으로 가시화하는 것이 일반적이다(Lacroute, 1994). 그러나 [그림 4-6]의 볼륨 데이터를 그대로 사용하고 X → Y → Z 순으로 메모리에 접근한다고 가정하자. 그러면 볼륨 데이터의 접근은 순차적이 되므로 효율적이고, Y의 증가에 영상 좌표 j가 변하지 않는 오버랩이 발생하여 영상 데이터의 접근도 효율적이 된다. 이에 착안하여 본 절에선 3가지 관찰 방향에 대해 모두 효율

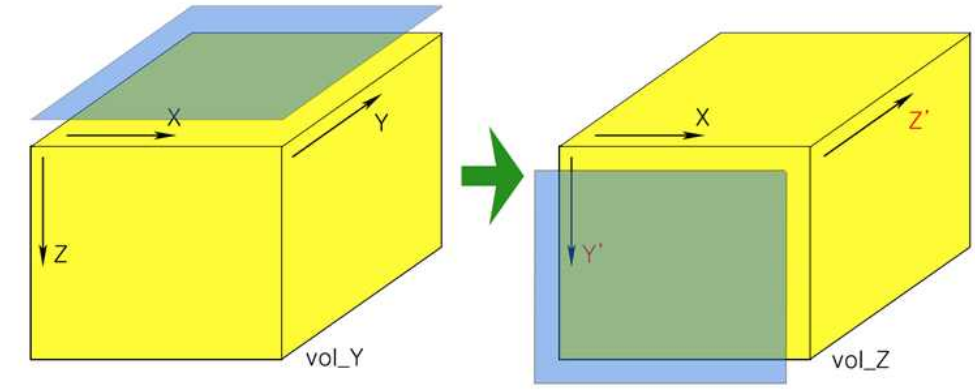
이 높은 메모리 재배치 방법을 제안한다. 가시화 연산을 시작하기 전에 세 개의 볼륨 세트를 다음의 규칙에 따라 생성한다.



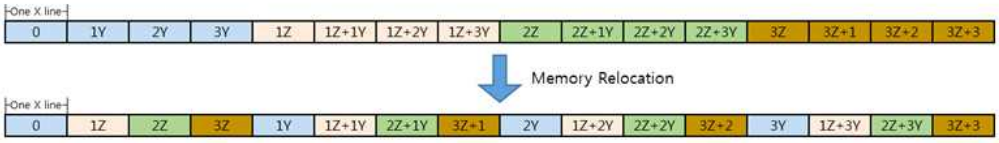
[그림 4-7] X 방향으로 관찰되는 볼륨에서의 데이터 재배치.

먼저, 일반적으로 디스크에 저장된 배열 상태는 일반적으로 [그림 4-3]의 하단 배열과 같이 저장되어 있고 기존 연구에서는 Z축 방향으로 관찰할 때 사용되나, 본 연구에 따르면 Y축 관찰 방향의 렌더링에 가장 효율적이다. 따라서 원본 데이터를 vol_Y라고 정의하였다. 이를 이용하여 X축 기준 관찰에 효율적인 볼륨 vol_X를 만든다. vol_Y를 기준으로 Y축과 X축을 교환하여 Y':=X, X':=Y로 vol_X에 저장한다. x 방향으로 투영이 일어나므로 메모리 교환이 일어나면 X' 방향으로 투영이 일어나서 Y'Z평면에 영상이 생성된다.

가시화 순서는 Y' → X' → Z (또는 X → Y → Z) 로 진행한다. 볼륨 데이터의 경우 순차적으로 접근하므로 효율적이고, 영상에서도 오버랩이 발생하여 효율적이다.



(a)



(b)

[그림 4-8] Z 방향으로 관찰되는 볼륨에서의 데이터 재배치:
(a) Z방향으로 관찰되는 볼륨을 효율적인 접근을 위해 Y, Z축 기준으로 데이터를 변경한 모양, (b) 데이터 재배치를 하기 위한 1차원 배열의 모습.

마찬가지 방법으로 vol_Z를 만든다. vol_Y를 기준으로 Y축과 Z축을 교환하여 Y'=Z, Z'=Y로 한다. 가시화 순서는 X → Z' → Y (또는 X → Y → Z)로 수행하면 마찬가지로 볼륨 데이터는 순차적으로 접근하고, 영상에서는 오버랩이 발생하여 효율적이다. 전치 방법은 [그림 4-8](b)와 같이 스캔 라인 단위로 메모리를 복사하면 된다.

이러한 전치 알고리즘은 프로그램 수행 시 단 한 번만 수행되므로 효과적이다. X축과 Y축의 교환은 효율적으로 영상의 전치를 수행하는 3.2절의 방법과 동일하게 AVX2를 이용하여 16x16 크기의 타일 단위로 전치를 시행한다. Y축과 Z축의 교환은 배열을 순차적으로 접근하지 못해 3.2절을 활용하지 못해 일일이 메모리 복사를 수행한다. 하지만 X 방향으로 인접한 데이터의 메모리 복사이므로 간단하며 매우 빠르다. 단, 기존 shear-warp 알고리즘과 동일하게 메모리를 3배 사용한다는 점은 단점으로 지적되나, 최근 일반적인 컴

퓨터의 메모리가 8~16GB 이상인데, 일반적으로 사용하는 큰 볼륨 데이터의 크기는 512×512×1024의 경우 512MB라서 3배의 메모리를 사용하여도 충분하다.

제 5 장 실험

제 1 절 실험 환경

이번 장에서는 3장에서 제안한 고속 MIP 알고리즘의 성능을 비교하기 위한 실험은 5.1장에서 확인하고 5.2장에서는 4장에서 제안한 고속 선형 보간 MIP를 검증한다. 5.3장에서는 각각의 결과 영상에 대한 화질 비교할 것이다. 실험 환경은 i5-4570 @ 3.2 Ghz CPU, 16GB의 메모리를 장착한 개인용 컴퓨터에서 수행하였다. 프로그램 개발은 window7에서 Visual Studio 2015를 사용하여 c++로 직접 작성하였다. [표 5-1]은 실험한 볼륨 데이터들의 정보이다.

[표 5-1] 볼륨 데이터 세트

Name	Dimension	Pixel Size [mm]	Slice thickness [mm]	Size (MB)
복부 #1	512×512×277	0.570.57	1.00	138.5
복부 #2	512×512×110	0.630.63	3.00	55.0
두부	512×512×552	0.420.42	1.00	276.0
하지	512×512×1165	0.780.78	1.50	582.5

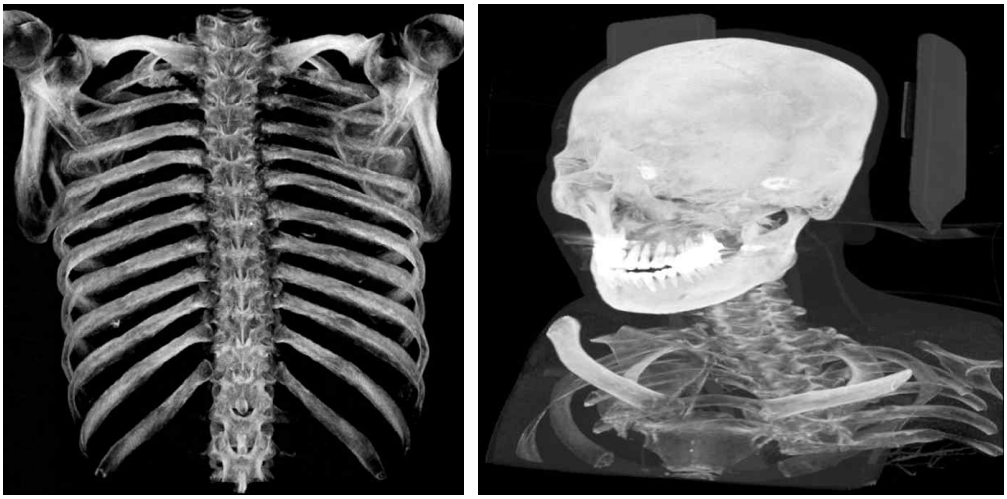
제 2 절 고속 MIP 실험 결과

관찰 방향에 따라 성능 차이가 있기 때문에 관찰 방향을 다양하게 바꿔가며 가시화 속도를 측정한다. 난수를 발생하여 총 21곳의 관찰자 위치를 결정한다. 즉, 주 관찰 방향 X, Y, Z축 각각에 대해 7가지 경우가 고르게 배치되어 총 21곳의 관찰자 위치를 [표 5-2]에서와 같이 설정하며 관찰 위치는 볼륨의 정중앙이다.

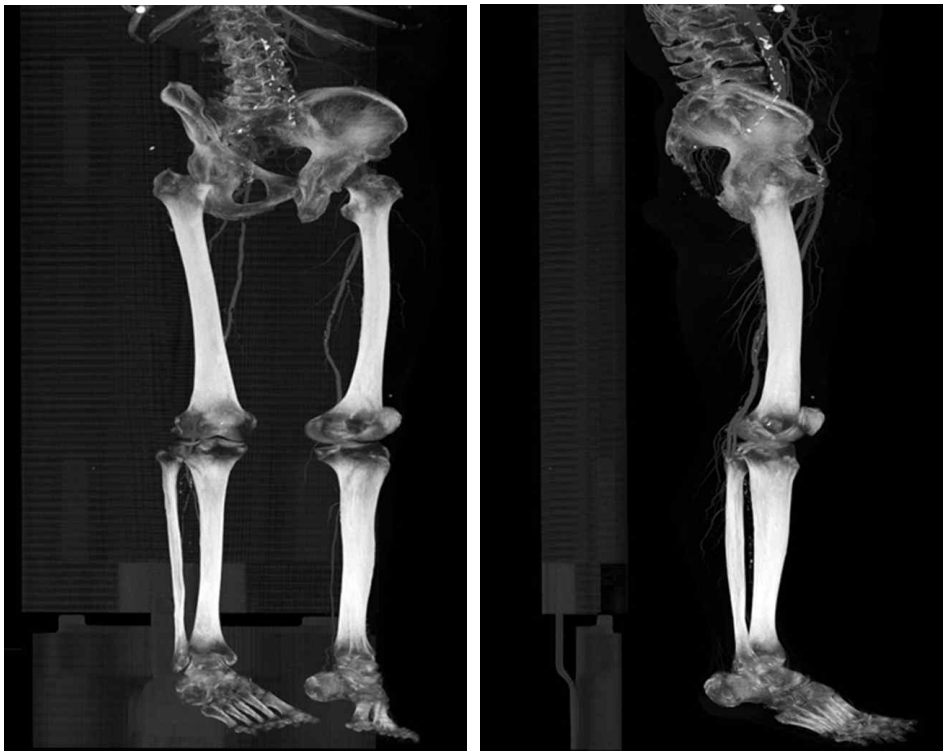
[표 5-2] 실험에 사용한 관찰자 위치

	X축 관찰 위치	Y축 관찰 위치	Z축 관찰 위치
1	(0, 184, 63)	(265, 191, 157)	(294, 503, 411)
2	(114, 293, 269)	(44, 57, 281)	(122, 333, 273)
3	(368, 288, 60)	(232, 168, 99)	(464, 141, 511)
4	(48, 54, 323)	(146, 39, 1)	(53,268, 347)
5	(2, 123, 174)	(138, 69, 112)	(44, 262, 357)
6	(187, 197, 198)	(467, 499, 235)	(237, 259, 323)
7	(54, 217, 147)	(141, 529, 178)	(316, 35, 590)

결과 영상은 다음과 같다. [표 5-2]에 제시된 다양한 관찰자 위치와 관찰 방향에 대하여 제안 기법을 사용하여 렌더링 된 MIP 영상은 MMX(Kye, 2009)와 SSE2 기법으로 렌더링 된 MIP 영상과의 차 영상이 0으로 동일한 것으로 확인되었다.



(a) (b)



(c) (d)

[그림 5-1] 여러 데이터를 여러 각도에서 MIP한 결과 영상.

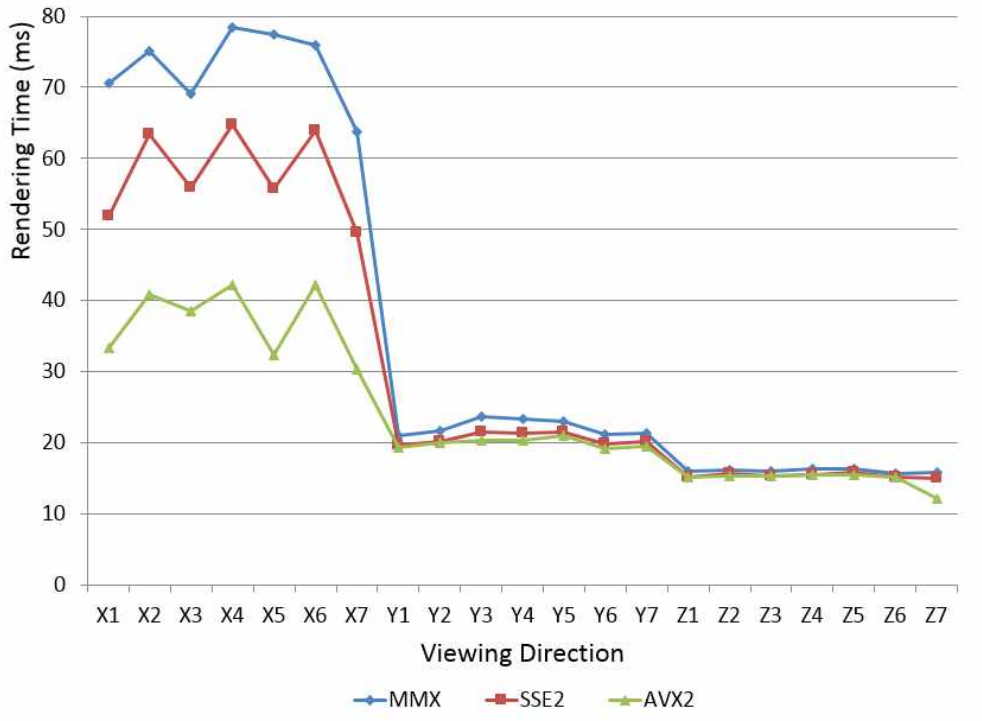
[표 5-3]은 제안 알고리즘의 성능을 평가하기 위한 결과로 복부 #1 데이터를 이용하여 각 관찰자 위치 21곳에 대해 가시화를 각각 100번 수행하고 평균 MIP 렌더링 시간을 측정한 결과이다. MMX(Kye, 2009)보다 SSE2(Chen, 2012)가, SSE2 보다 제안 기법인 AVX2가 빠르게 렌더링되는 것을 확인할 수 있다. 제안 기법인 AVX2는 MMX에 비해 평균적으로 약 1.55배 빨라졌다. 본 논문에서 제안한 고속 MIP(Fast MIP)는 FMIP로 표시한다.

[표 5-3] MMX, SSE2, AVX2의 렌더링 시간 비교

(시간 단위 : ms = 10^{-3}sec)

MMX	SSE2	AVX2(FMIP)
37.038	31.273	23.972

[그림 5-2]는 구체적인 성능을 분석하기 위해 각 관찰 방향에 대한 렌더링 속도를 그래프로 나타내었다. 이 그래프 또한 각 관찰점에서 복부 #1 데이터를 100번 가시화하고 평균한 결과이다. 제안 알고리즘과 비교 알고리즘은 shear-warp 분해를 기반으로 하므로 관찰 방향에 따라 차이가 매우 크다는 것을 알 수 있다. 메모리를 순차적으로 접근하는 Y 혹은 Z 관찰 축의 경우 성능 향상이 10% 정도인데, X축의 경우 성능 향상이 2배에 달한다. X축 방향의 관찰은 전치를 해야 하기 때문에 다른 방향보다 현저히 느린데, 본 연구의 고속 전치가 속도 향상의 효과를 보이는 것이다. 기존 방법의 경우, Y 혹은 Z축으로 관찰하다가 X축으로 관찰방향이 변경되면 다른 관찰방향에 비해 약 3.8배 느려지는 속도 감소를 겪게 된다. 이를 본 연구는 약 2.1배 차이로 크게 경감하였다.



[그림 5-2] MMX, SSE2, AVX2를 이용한 X, Y, Z축 기준의 MIP 렌더링 속도 비교.

[표 5-4]는 각각의 방법으로 전치 연산을 1억 번을 반복 수행한 후의 평균 시간이다. 메모리 복사 시간은 포함되지 않은 순수 연산 시간이다. 기존 기법인 MMX(Kye, 2009)의 전치와 제안 기법인 AVX2를 이용한 전치의 계산 시간을 비교하여 보았다. 참고로, 기존 기법인 AVX(Zekri, 2014)의 경우 i7 2670 @ 2.2GHz로 동작한 결과이며 float 연산을 4×4 단위로 수행한다. 한 번에 많은 연산을 수행하는 제안 방법은 다른 방법에 비해 우수하다. Zekri와 비교하면 2배 이상 빠르고 CPU i5와 i7의 차이를 무시하고 단순하게 CPU 성능을 clock으로 비교하여도, 제안 방법이 1.5배가량 우수하다.

[표 5-4] MMX와 AVX2를 이용한 전치 성능 향상 결과
(시간 단위 : ns = 10^{-9}sec)

	MMX 4×4	AVX 4×4	AVX 16×16
1회 연산 시간	4.721	3.6	27.62
16×16 연산 시간	75.536	57.6	27.62
16×16에 소요된 clock	271.7	126.7	88.4

[표 5-5]는 볼륨 데이터에 대해 일관적으로 시간이 소요된다는 점을 보이기 위해, 다양한 데이터에 대해서 100회 가시화 한 시간을 측정하였다. 관찰 방향은 각각 X축 방향은 (0.926509, 0.260581, 0.271438), Y축 방향은 (-0.131742, 0.951469, -0.278122), Z축 방향은 (00102672, -0.667368, -0.737617)을 이용하여 100번 가시화를 한 렌더링 시간들의 평균을 표시하였다. 볼륨 데이터들의 크기와 비교해보면 데이터의 크기와 소요되는 시간이 선형으로 비례하다는 것을 알 수 있다. 예를 들어 두부 데이터는 복부 #1 데이터의 약 두 배 크기인데 렌더링 시간도 대략 두 배 더 걸리는 것으로 측정된다. 이는 볼륨 기반으로 명령을 수행하는 shear-warp의 특성이며, 추후 다른 볼륨 데이터를 이용한다고 하더라도 예상 수행 시간을 추정할 수 있다. 최종적으로 본 연구를 통해 512×512×552 크기의 두부 데이터가 일반적인 CPU 기반으로 20fps 이상의 속도로 가시화 될 수 있다는 것을 보여주었다.

[표 5-5] 각 데이터에 따른 X, Y, Z축 별 MMX, SSE2, AVX2 렌더링 시간 비교 (시간단위 : ms = 10^{-3}sec)

		복부 #1	복부 #2	두부	하지
X축 방향	MMX	70.54734	28.01430	135.01820	284.33343
	SSE2	51.96790	20.48483	95.94551	203.81860
	AVX2	33.33280	13.18679	65.82183	138.73640
Y축 방향	MMX	20.99611	7.66047	44.62732	121.41248
	SSE2	19.69106	7.23729	40.19813	108.11024
	AVX2	19.27156	7.03868	39.13795	104.23639
Z축 방향	MMX	15.96361	6.29126	30.19485	64.89549
	SSE2	15.21291	5.99895	29.25030	62.87185
	AVX2	15.12345	6.04863	29.14376	62.54959

제 3 절 고속 선형 보간 MIP 실험 결과

제안 방법의 성능 향상을 확인하기 위해 [표 5-6]은 렌더링 시간을 측정하여 제시하였다. 회전축에 대해 5도씩 회전하면서 측정한 가시화 시간의 평균을 측정하였다. Chen(2012)의 연구에서 사용된 방법 MADD으로 표시하였고, 4장에서 제안한 고속 선형보간(Fast Linear Interpolation MIP)을 FLIMIP로 표시한다. 메모리 재배치(Memory Reordering) 기법은 세 벌의 데이터를 전처리 과정에서 생성한다. 전처리에 필요한 시간은 1초 이내로 프로그램의 수행에 영향을 주지 않는다. 메모리 재배치의 성능 검증을 위해 보간을 완료하였지만, 메모리 재배치를 하지 않은 알고리즘을 notMR로 표시한다. 가시화 속도는 같은 렌더링을 100번 반복한 평균 시간을 측정한다.

[표 5-6] 보간 방법과 메모리 재배치에 따른 성능 차이표
(시간 단위 : ms = 10^{-3}s)

Name	MADD	notMR	FLIMIP	MADD ÷ FLIMIP
복부 #1	92.537	19.692	18.112	5.11배
복부 #2	37.643	7.717	7.389	5.09배
두부	191.208	41.563	38.644	4.95배
하지	401.743	96.847	87.805	4.58배
평균	181.195	41.160	37.793	4.93배

모든 실험에 대하여 볼륨은 세 벌씩 생성되며 볼륨을 읽는 loop 순서는 항상 for(Z) for(Y) for(X)이다. MADD와 notMR에서는 아무것도 하지 않고 불러온 볼륨이 vol_Z가 된다. 이 볼륨은 Z축 방향의 관찰에서 사용된다. Y축 관찰을 위한 볼륨은 Z와 Y를 교환하여 Z':=Y, Y':=Z가 되게 만든다. X축 방향의 관찰에는 Z':=X, Y':=Z, X':=Y인 볼륨을 만들면 같은 형식의 볼륨이 만들어진다.

FLIMIP에서는 이미지 배열의 순서에 따른 성능도 고려해야 하기 때문에 세 벌의 볼륨이 기존의 모양과 다르다. [그림 4-7]과 [그림 4-8]와 같이 기존 볼륨이 vol_Z가 아닌 vol_Y가 되어 Y축 관찰 방향에서 사용된다. Z축 방향에 필요한 볼륨이 Z':=Y, Y':=Z가 되어 vol_Z가 될 것이다. X축 관찰을 위한 vol_X는 Z':=Z, Y':=X, X':=Y로 만들어져 렌더링하게 된다.

MADD는 [그림 2-3]과 같이 a0b0+a1b1와 a2b2+a3b3를 더해야 이차원 보간이 된다. 이때 가로 방향 덧셈이 필수적이다. SIMD에서 가로방향 덧셈(MADD)은 세로 방향 덧셈에 비해 큰 비효율이 발생한다. 또한, 한 점에 인접한 네 개의 복셀(a0, a1, a2, a3)을 SIMD 데이터로 불러올 때 한 번에 불러오기 위해 세로로 배치되어있는 복셀들을 가로로 볼륨 재배치하는 비효율이 발생한다. PMADDWD을 할 때마다 하나의 데이터 값이 두 배로 커져서 데이터양이 늘어나는 문제가 생긴다. 그리고 이 방법은 반올림 계산을 하지 않아 정밀도가 떨어진다.

이러한 이유로 MADD 방법은 우리가 제안한 FLIMIP에 비해 약 4.93배

느린 성능을 보인다. 우리의 코드는 가로방향 덧셈을 전혀 수행하지 않고 저장되어 있는 데이터를 그대로 불러와서 사용하고 계산하기 때문에 MADD의 방법보다 더 좋은 성능을 보이게 된다.

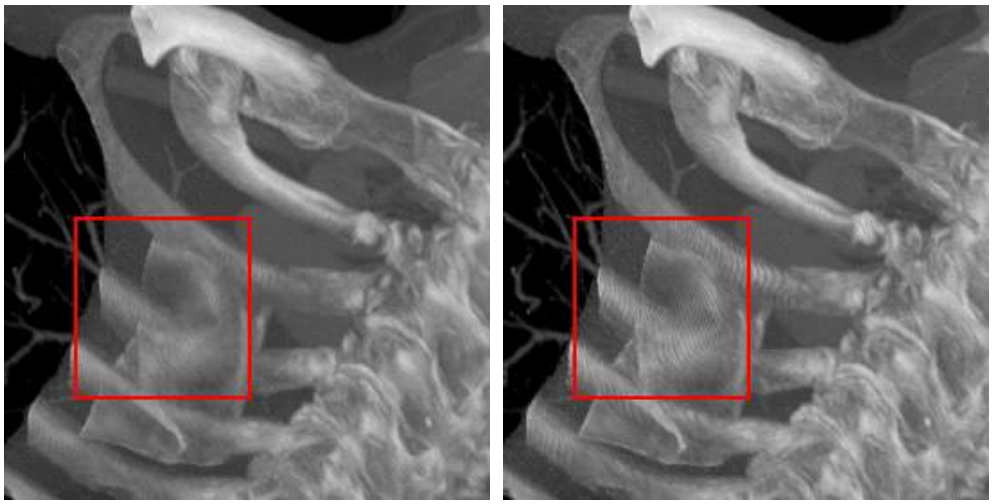
[그림 5-3] 복부#1에 대한 각도별 notMR, FLIMIP성능 측정 비교.

본 연구에서 MR의 추가적인 성능 향상을 설명하기 위해 [그림 5-3]을 나타내었다. MR을 적용하기 전에는 메모리를 읽고 쓸 때 어느 방향에서 보던 매번 다른 위치의 메모리를 참조했기 때문에 같은 축 안에서의 성능은 거의 일정하다. 반면 4장의 2절을 적용하면 바라보는 방향에 따라 같은 위치를 참조하는 횟수가 조금씩 다르므로 약간의 성능 차이가 발생한다. 사용하는 볼륨 데이터의 축 방향과 관찰 방향이 완전히 평행하면 볼륨에서 X, Y축 데이터를 한 장 처리하는 동안 이미지 데이터는 같은 위치를 참조하게 된다. 관찰 방향이 볼륨 데이터의 45도 방향으로 움직일수록 데이터를 참조하는 위치도 바뀌는 횟수가 조금씩 늘어나 [그림 5-3]과 같이 점차적인 성능 차이를 보이게 된다. 대각선 방향에서 관찰 시 FLIMIP가 효율이 약간 떨어지지만, 매우

일시적이며 일반적으로 볼륨 데이터를 관찰할 때 정면에서 보는 경우가 많아 실제 사용자의 체감 속도는 MR을 적용한 것이 더 빠르게 느껴질 것이다.

복부 #1의 데이터의 경우 볼륨의 크기가 512×512×277이다. MR의 경우 삼차원 보간을 위한 다음 슬라이스 접근을 위해 Y축 방향관찰에선 X×Y를, Z축 관찰에선 X×Z를 건너뛰게 된다. 반대로 FI는 Y축 방향 관찰에서 X×Z를, Z축 방향 관찰에선 X×Y를 건너뛴다. 이때 X×Z보다 X×Y가 더 크기 때문에 각각의 경우에 대해 약간의 성능 하락이 있으며, Z축 방향의 관찰이 더 우수한 성능을 가진다.

제 4 절 화질 비교



(a) (b)

[그림 5-4] 같은 데이터, 각도, windowing에서 보간 방법에 따른 화질 비교 영상:
(a) 삼차원 보간, (b) 최근접 보간.

일반적으로 영상의 삼차원 선형 보간이 최근접 보간에 비해 화질이 좋다고 알려져 있다. 결과 화면에서도 보간을 한 영상이 하지 않은 영상에 비해 잡음이나 계단 현상이 줄어들어 화질이 개선됨을 알 수 있다.

3차원 보간을 하면 샘플링 횟수가 8번으로 늘어나고 추가적인 연산이 필요하여 성능이 떨어지는 문제가 발생하여 세 벌의 메모리를 미리 만들어 렌더링을 수행하였다. [표 5-3]과 [표 5-6]을 보면 복부#1 데이터 기준으로 FMIP 수행 시 평균 23.972ms가 수행되었고 FLIMIP 방법으로는 평균 18.112ms가 소요되었다. 샘플링 횟수가 8배가 늘어났음에도 약 30%의 성능 향상이 일어났다. [표 5-5]에서 볼 수 있듯 FMIP는 X축 방향의 관찰에서 특히 낮은 성능을 보였지만 FLIMIP는 축에 상관없는 효율을 보여준다. Z축 방향의 관찰은 FMIP가 더 좋은 성능을 보인다. 하지만 FLIMIP는 삼차원 보간을 사용하여 화질을 개선했으며 X축의 성능 향상을 고려하면 이 또한 아주 좋은 결과이다.

이와 같은 실험 결과에 따라, 메모리의 제한이 있는 시스템에서 빠르게 결과 영상을 확인하고 싶은 경우 FMIP 방법을 사용하는 것이 좋으며, 메모리가 충분하며 고화질 영상을 시간 편차 없이 생성하고자 하는 경우 FLIMIP 방법을 사용하면 효율적이다.

제 6 장 결론 및 향후 과제

본 연구는 CPU 기반으로 실시간 MIP가 가능한 것을 보였다. AVX2를 이용하여 16개의 복셀을 한 번에 max 연산함으로써 반복문의 횟수를 줄이고 조건문을 삭제시켰다. 그 결과 512×512×552 크기의 볼륨 데이터가 20fps 이상의 실시간 속도로 수행된다. 구체적으로 본 연구는 고속 행렬 전치(matrix transposition) 방법을 개발하였다. 이 방법은 렌더링 속도를 향상할 뿐만 아니라, 관찰 방향에 대한 속도 편차를 줄여 사용자의 사용성을 대폭 향상시킨다. 행렬 전치는 영상처리에서 흔하게 사용되므로, 제안 방법은 볼륨 가시화 외에 다른 영상 처리에도 범용으로 적용할 수 있다.

그리고 CPU 기반의 고화질 영상도 실시간으로 생성이 가능한 방법을 제안한다. SIMD에서 지원하지 않는 정수와 실수 간의 곱셈을 하기 위한 알고리즘을 제안하여 삼차원 선형 보간에 성공하였으며 보간에 필요한 연산을 최소화하였다. 또한, 1초 이내의 전처리 시간을 사용하면, 관찰 방향에 무관한 실시간 고속 가시화가 가능하다. 구체적인 방법으로 본 연구는 미리 전처리된 세 벌의 볼륨으로 관찰 방향에 따라 최적의 볼륨을 선택하는 방법을 제안한다. 이 방법은 약간의 전처리를 통해 모든 가시화에서 더 빠르고 손실 없이 영상을 가시화할 수 있어 매우 큰 이득이다. 하지만 이에 그치지 않고 영상 메모리에서 또한 읽고 쓸 때 발생하는 메모리 효율성을 고려하여 성능을 극대화했다.

현재 실험에서는 하나의 코어만을 사용하였다. OpenMP와 같은 멀티 코어 CPU 기법을 적용하면 하나의 스레드가 수행하던 일을 여러 개의 스레드가 수행하여 효율이 높아진다. 현재 실험에 사용된 i5-4570은 4개의 코어와 스레드를 사용하므로 멀티코어 기법을 활용하면 추가 성능 향상을 얻을 수 있다.

최근 출시된 i9-X 시리즈 CPU는 18코어, 36 스레드를 갖췄다. 이 장비를 이용하여 멀티 코어 기법을 적용하면 더 높은 성능 향상이 일어난다. 또한, 이 장비는 AVX2의 상위 버전인 AVX-512를 탑재하였다. AVX-512는

64바이트를 한 번에 연산할 수 있는 명령어들을 가지고 있어 AVX2보다 한 단계 더 빠른 렌더링을 수행할 수 있을 것으로 예상한다. 본 연구의 제안 방법을 이용하면, 향후 CPU의 성능 향상에 대응하여, GPU와 비교해도 손색이 없는 렌더링 속도를 얻을 수 있을 것으로 예상된다.

참 고 문 헌

1. 국외문헌

Belina, S., Cuk, V., Klapan, I. (2009). Virtual Endoscopy and 3D Volume Rendering in the Management of Frontal Sinus Fractures. Collegium Antropologicum, 33, 43－51.

Chen, K., Duan, Y., Yan, L., Sun, J., Guo, Z. (2012). Efficient SIMD optimization of HEVC encoder over X86 processors. In Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific, 1-4.

Dachille, F., Kreeger, K., Chen, B., Bitter I., Kaufman, A. (1998). High-Quality Volume Rendering Using Texture Mapping Hardware. SIGGRAPH/Eurographics Workshop on Graphics Hardware'98, 69-76.

Fang, L., Wang, Y., Qiu, B., Qian, Y. (2002). Fast Maximum Intensity Projection Algorithm Using Shear Warp Factorization and Reduced Resampling. Magnetic Resonance in Medicine, 47, 696-700.

Hofmann, J., Treibig, J., Hager, G., Wellein, G. (2014). Performance engineering for a medical imaging application on the Intel Xeon Phi accelerator. In Architecture of Computing Systems (ARCS), 2014 Workshop Proceedings 1-8.

Kiefer, G., Lehmann, H., Weese, J. (2006). Fast Maximum Intensity Projections of Large Medical Data Sets by Exploiting Hierarchical Memory Architectures. IEEE Transactions on Information Technology in Biomedicine, 10(2), 385-394.

Kye, H. (2009). Efficient Maximum Intensity Projection using SIMD Instruction and Streaming Memory Transfer. Journal of Korea Multimedia Society, 12(4), 512-520.

Kye, H., Lee, S., Lee, J. (2017). CPU-based real-time maximim intensity projection via fast matrix transposition using parallelization operations with the AVX instruction set. Multimedia Tools and Application.

Lacroute, P., Levoy, M. (1994). Fast volume rendering using a shear-warp factorization of the viewing transformation. Proceeding SIGGRAPH '94 Proceedings of the 21st annual conference on Computer graphics and interactive techniques, 451-458.

McFarlin, D.S. Arbatov, V., Franchetti, F., Püschel, M. (2011). Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. Proceedings of the international conference on Supercomputing, 265-274.

Mensmann, J., Ropinski, T., Hinrichs, K.H. (2010). An Advanced Volume Raycasting Technique using GPU Stream Processing. International Conference on Computer Graphics Theory and Applications, 190-198.

Mora, B., Ebert, D.S. (2005). Low-Complexity Maximum Intensity Projection. ACM Transactions on Graphics, 24(4), 1392-1416.

Mroz, L., König, A., Gröller, E. (1999). Real-Time Maximum Intensity Projection. Data Visualization '99, 135-144.

Mroz, L., Hauser, H., Gröller, E. (2000). Interactive High-Quality Maximum Intensity Projection. Computer Graphics forum, 19(3), 341-350.

Pekar, V., Hempel, D., Kiefer, G., Busch, M., Weese, J. (2003). Efficient Visualization of Large Medical Image Datasets on Standard PC Hardware. Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization, 135-140.

Rezk-Salama, C., Engel, K., Bauer, M., Greiner G., Ertl, T. (2000). Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stagerasterization. Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware'00, 109-118.

Sabella, P. (1988). A Rendering Algorithm for Visualizing 3D Scalar Fields. ACM SIGGRAPH 1988 Proceedings of the 15th annual conference on Computer graphics and interactive techniques, 51-58.

Schreiner, S., Galloway, R. L. Jr. (1993). A fast maximum-intensity projection algorithm for generating magnetic resonance angiograms. IEEE Transactions on Medical Imaging, 12(1), 50-57.

Treibig, J., Hager, G., Hofmann, H. G., Hornegger, J., Wellein, G. (2013). Pushing the limits for medical image reconstruction on recent standard multicore processors. The International Journal of High Performance Computing Applications, 27(2), 162-177.

Vollrath, J.E. Weiskopf, D., Ertl, T. (2005) A generic software framework for the gpu volume rendering pipeline. Proceedings of Vision, Modeling, and Visualization, 391-398.

Zekri, A. S. (2014). Enhancing the Matrix Transpose Operation Using Intel Avx Instruction Set Extension. International Journal of Computer Science & Information Technology (IJCSIT), 6(3), 67-78.

Zhao, K. Sakamoto, N., Koyamada, K. (2014). Fused Visualization for Large-scale Timevarying Volume Data with Adaptive Particle-based Rendering. AsiaSim 2014, 14th International Conference on Systems Simulation, 228-242.

인터넷 사이트

Intel Intrinsics Guide, https://software.intel.com (Accessed May 2018).

부 록

부록 1. 8×8 영상의 행렬 전치. SSE2명령어와 실행 예제

8×8 행렬 SSE2 전치 코드 ([표 3-3]에 대한 원래 소스 코드)

```
1:  Transpose_8x8(v_A, v_B, v_C, v_D, v_E, v_F, v_G, v_H)
2:  {
3:      __m128i t_A, t_B, t_C, t_D, t_E, t_F, t_G, t_H;
4:      // OPERATION A
5:      t_A = _mm_unpacklo_epi16(v_A, v_B);
6:      t_B = _mm_unpackhi_epi16(v_A, v_B);
7:      t_C = _mm_unpacklo_epi16(v_C, v_D);
8:      t_D = _mm_unpackhi_epi16(v_C, v_D);
9:      t_E = _mm_unpacklo_epi16(v_E, v_F);
10:     t_F = _mm_unpackhi_epi16(v_E, v_F);
11:     t_G = _mm_unpacklo_epi16(v_G, v_H);
12:     t_H = _mm_unpackhi_epi16(v_G, v_H);

13:     //    OPERATION B
14:     v_A = _mm_unpacklo_epi32(t_A, t_C);
15:     v_B = _mm_unpackhi_epi32(t_A, t_C);
16:     v_C = _mm_unpacklo_epi32(t_B, t_D);
17:     v_D = _mm_unpackhi_epi32(t_B, t_D);
18:     v_E = _mm_unpacklo_epi32(t_E, t_G);
19:     v_F = _mm_unpackhi_epi32(t_E, t_G);
20:     v_G = _mm_unpacklo_epi32(t_F, t_H);
21:     v_H = _mm_unpackhi_epi32(t_F, t_H);

22:     //    OPERATION C
```

```
23:     t_A = _mm_unpacklo_epi64(v_A, v_E);
24:     t_B = _mm_unpackhi_epi64(v_A, v_E);
25:     t_C = _mm_unpacklo_epi64(v_B, v_F);
26:     t_D = _mm_unpackhi_epi64(v_B, v_F);
27:     t_E = _mm_unpacklo_epi64(v_C, v_G);
28:     t_F = _mm_unpackhi_epi64(v_C, v_G);
29:     t_G = _mm_unpacklo_epi64(v_D, v_H);
30:     t_H = _mm_unpackhi_epi64(v_D, v_H);
31: }
```

부록 2. 16×16 영상의 행렬 전치. AVX2명령어와 실행 예제

16×16 행렬 AVX2 전치 코드

```
1:  Transpose_16x16(v_A, v_B, v_C, v_D, v_E, v_F, v_G, v_H,
                    v_I, v_J, v_K, v_L, v_M, v_N, v_O, v_P)
2:  {
3:      __m256i t_A, t_B, t_C, t_D, t_E, t_F, t_G, t_H,
              t_I, t_J, t_K, t_L, t_M, t_N, t_O, t_P;

4:      //    OPERATION A
5:      t_A = _mm256_unpacklo_epi16(v_A, v_B);
6:      t_B = _mm256_unpackhi_epi16(v_A, v_B);
7:      t_C = _mm256_unpacklo_epi16(v_C, v_D);
8:      t_D = _mm256_unpackhi_epi16(v_C, v_D);
9:      t_E = _mm256_unpacklo_epi16(v_E, v_F);
10:     t_F = _mm256_unpackhi_epi16(v_E, v_F);
11:     t_G = _mm256_unpacklo_epi16(v_G, v_H);
12:     t_H = _mm256_unpackhi_epi16(v_G, v_H);
13:     t_I = _mm256_unpacklo_epi16(v_I, v_J);
14:     t_J = _mm256_unpackhi_epi16(v_I, v_J);
15:     t_K = _mm256_unpacklo_epi16(v_K, v_L);
16:     t_L = _mm256_unpackhi_epi16(v_K, v_L);
17:     t_M = _mm256_unpacklo_epi16(v_M, v_N);
18:     t_N = _mm256_unpackhi_epi16(v_M, v_N);
19:     t_O = _mm256_unpacklo_epi16(v_O, v_P);
20:     t_P = _mm256_unpackhi_epi16(v_O, v_P);

21:     //    OPERATION B
22:     v_A = _mm256_unpacklo_epi32(t_A, t_C);
23:     v_B = _mm256_unpackhi_epi32(t_A, t_C);
```

```
24:   v_C = _mm256_unpacklo_epi32(t_B, t_D);
25:   v_D = _mm256_unpackhi_epi32(t_B, t_D);
26:   v_E = _mm256_unpacklo_epi32(t_E, t_G);
27:   v_F = _mm256_unpackhi_epi32(t_E, t_G);
28:   v_G = _mm256_unpacklo_epi32(t_F, t_H);
29:   v_H = _mm256_unpackhi_epi32(t_F, t_H);
30:   v_I = _mm256_unpacklo_epi32(t_I, t_K);
31:   v_J = _mm256_unpackhi_epi32(t_I, t_K);
32:   v_K = _mm256_unpacklo_epi32(t_J, t_L);
33:   v_L = _mm256_unpackhi_epi32(t_J, t_L);
34:   v_M = _mm256_unpacklo_epi32(t_M, t_O);
35:   v_N = _mm256_unpackhi_epi32(t_M, t_O);
36:   v_O = _mm256_unpacklo_epi32(t_N, t_P);
37:   v_P = _mm256_unpackhi_epi32(t_N, t_P);

38:   // OPERATION C
39:   t_A = _mm256_unpacklo_epi64(v_A, v_E);
40:   t_B = _mm256_unpackhi_epi64(v_A, v_E);
41:   t_C = _mm256_unpacklo_epi64(v_B, v_F);
42:   t_D = _mm256_unpackhi_epi64(v_B, v_F);
43:   t_E = _mm256_unpacklo_epi64(v_C, v_G);
44:   t_F = _mm256_unpackhi_epi64(v_C, v_G);
45:   t_G = _mm256_unpacklo_epi64(v_D, v_H);
46:   t_H = _mm256_unpackhi_epi64(v_D, v_H);
47:   t_I = _mm256_unpacklo_epi64(v_I, v_M);
48:   t_J = _mm256_unpackhi_epi64(v_I, v_M);
49:   t_K = _mm256_unpacklo_epi64(v_J, v_N);
50:   t_L = _mm256_unpackhi_epi64(v_J, v_N);
51:   t_M = _mm256_unpacklo_epi64(v_K, v_O);
```

```
52:     t_N = _mm256_unpackhi_epi64(v_K, v_O);
53:     t_O = _mm256_unpacklo_epi64(v_L, v_P);
54:     t_P = _mm256_unpackhi_epi64(v_L, v_P);

55:     // OPERATION D
56:     v_A = _mm256_permute2x128_si256(t_A, t_I, 0x20);
57:     v_B = _mm256_permute2x128_si256(t_B, t_J, 0x20);
58:     v_C = _mm256_permute2x128_si256(t_C, t_K, 0x20);
59:     v_D = _mm256_permute2x128_si256(t_D, t_L, 0x20);
60:     v_E = _mm256_permute2x128_si256(t_E, t_M, 0x20);
61:     v_F = _mm256_permute2x128_si256(t_F, t_N, 0x20);
62:     v_G = _mm256_permute2x128_si256(t_G, t_O, 0x20);
63:     v_H = _mm256_permute2x128_si256(t_H, t_P, 0x20);
64:     v_I = _mm256_permute2x128_si256(t_A, t_I, 0x31);
65:     v_J = _mm256_permute2x128_si256(t_B, t_J, 0x31);
66:     v_K = _mm256_permute2x128_si256(t_C, t_K, 0x31);
67:     v_L = _mm256_permute2x128_si256(t_D, t_L, 0x31);
68:     v_M = _mm256_permute2x128_si256(t_E, t_M, 0x31);
69:     v_N = _mm256_permute2x128_si256(t_F, t_N, 0x31);
70:     v_O = _mm256_permute2x128_si256(t_G, t_O, 0x31);
71:     v_P = _mm256_permute2x128_si256(t_H, t_P, 0x31);
72: }
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112
113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128
129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144
145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176
177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192
193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208
209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224
225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256

(a)

1	17	2	18	3	19	4	20	9	25	10	26	11	27	12	28
5	21	6	22	7	23	8	24	13	29	14	30	15	31	16	32
33	49	34	50	35	51	36	52	41	57	42	58	43	59	44	60
37	53	38	54	39	55	40	56	45	61	46	62	47	63	48	64
65	81	66	82	67	83	68	84	73	89	74	90	75	91	76	92
9	85	70	86	71	87	72	88	77	93	78	94	79	95	80	96
97	113	98	114	99	115	100	116	105	121	106	122	107	123	108	124
101	117	102	118	103	119	104	120	109	125	110	126	111	127	112	128
129	145	130	146	131	147	132	148	137	153	138	154	139	155	140	156
133	149	134	150	135	151	136	152	141	157	142	158	143	159	144	160
161	177	162	178	163	179	164	180	169	185	170	186	171	187	172	188
165	181	166	182	167	183	168	184	173	189	174	190	175	191	176	192
193	209	194	210	195	211	196	212	201	217	202	218	203	219	204	220
197	213	198	214	199	215	200	216	205	221	206	222	207	223	208	224
225	241	226	242	227	243	228	244	233	249	234	250	235	251	236	252
229	24	230	246	231	247	232	248	237	253	238	254	239	255	240	256

(b)

1	17	33	49	2	18	34	50	9	25	41	57	10	26	42	58
3	19	35	51	4	20	36	52	11	27	43	59	12	28	44	60
5	21	37	53	6	22	38	54	13	29	45	61	14	30	46	62
7	23	39	55	8	24	40	56	15	31	47	63	16	32	48	64
65	81	97	113	66	82	98	114	73	89	105	121	74	90	106	122
67	83	99	115	68	84	100	116	75	91	107	123	76	92	108	124
69	85	101	117	70	86	102	118	77	93	109	125	78	94	110	126
71	87	103	119	72	88	104	120	79	95	111	127	80	96	112	128
129	145	161	177	130	146	162	178	137	153	169	185	138	154	170	186
131	147	163	179	132	148	164	180	139	155	171	187	140	156	172	188
133	149	165	181	134	150	166	182	141	157	173	189	142	158	174	190
135	151	167	183	136	152	168	184	143	159	175	191	144	160	176	192
193	209	225	241	194	210	226	242	201	217	233	249	202	218	234	250
195	211	227	243	196	212	228	244	203	219	235	251	204	220	236	252
197	213	229	245	198	214	230	246	205	221	237	253	206	222	238	254
199	215	231	247	200	216	232	248	207	223	239	255	208	224	240	256

(c)

1	17	33	49	65	81	97	113	9	25	41	57	73	89	105	121
2	18	34	50	66	82	98	114	10	26	42	58	74	90	106	122
3	19	35	51	67	83	99	115	11	27	43	59	75	91	107	123
4	20	36	52	68	84	100	116	12	28	44	60	76	92	108	124
5	21	37	53	69	85	101	117	13	29	45	61	77	93	109	125
6	22	38	54	70	86	102	118	14	30	46	62	78	94	110	126
7	23	39	55	71	87	103	119	15	31	47	63	79	95	111	127
8	24	40	56	72	88	104	120	16	32	48	64	80	96	112	128
129	145	161	177	193	209	225	241	137	153	169	185	201	217	233	249
130	146	162	178	194	210	226	242	138	154	170	186	202	218	234	250
131	147	163	179	195	211	227	243	139	155	171	187	203	219	235	251
132	148	164	180	196	212	228	244	140	156	172	188	204	220	236	252
133	149	165	181	197	213	229	245	141	157	173	189	205	221	237	253
134	150	166	182	198	214	230	246	142	158	174	190	206	222	238	254
135	151	167	183	199	215	231	247	143	159	175	191	207	223	239	255
136	152	168	184	200	216	232	248	144	160	176	192	208	224	240	256

(d)

1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255
16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256

(e)

AVX2를 이용한 16x16 전치 과정:

(a) ORIGINAL, (b) AFTER OPERATION A, (c) AFTER OPERATION B, (d) AFTER OPERATION C, (e) AFTER OPERATION D.

ABSTRACT

Realtime maximum intensity projection using CPU-based SIMD instruction set

Lee, Se-Hee

Major in Information System Engineering

Dept. of Information System Engineering

The Graduate School

Hansung University

Rapid visualization speed is quite essential for MIP (Maximum Intensity Projection) rendering since the acquisition of perceptual depth can require frequent changes in viewing direction. Since medical data is very large in size, with hundreds or thousands of image sets, high-speed visualization can not be performed through traditional methods. In this paper, we propose a CPU-based real-time MIP visualization of mass volume data that uses parrallelization operations through SIMD's AVX instruction set. We propose fast MIP rendering through the use of a shear-warp technique that tilts the volume according to the direction of the ray, and positions the volume and image parallel in order to perform a SIMD that calls multiple values in a contiguous array, and computes using only one instruction. The volumes are usually stored via X, Y, Z

order. To use SIMD, you have to call sequential arrays. However, there is a case where the data can not be sequentially accessed according to direction observation of the volume. To solve this problem, we propose a fast transposition method using AVX instruction. Not only is there a large number of data to be computed at once, but the bottleneck is greatly mitigated by the reduction of the number of operations necessary. Since matrix transposition is commonly used in general image processing, the proposed method is applicable to other image processing algorithms for the sake of faster processing.

Volume rendering using the nearest interpolation is fast and easy, but the resulting image is not smooth, and staircasing typically occurs. Three dimensional linear interpolation was performed in order to improve image quality. To perform a three-dimensional interpolation, several operations are performed for each of the eight voxels. We propose a method for the reduction of the number of operations necessary by pre-calculating the weights of each of the voxels using the characteristics of the shear-warp used in this study. The interpolation operation requires multiplication of the density, which is an integer value of an actual voxel, by weight, which turns out to be a real number value. However, in this paper, most operations are performed via AVX2, which does not support multiplication of integers and real numbers. To solve this problem, we propose a method for conversion of the weights of real numbers into integers, then performing the AVX2 multiplication between the integers, which reveals the original values. Also, in order to show one pixel when three-dimensional interpolation is present, eight voxels in the vicinity should be approached and calculated. In this instance, memory inefficiency occurs. To minimize this performance degradation, the order of accessing the volume memory must be preprocessed in order to be made as sequential as possible. We have prepared the three volume sets

according to the observation direction before rendering, and each volume minimizes the distance of the voxels necessary for rendering and interpolation. Also, when the calculated values are stored in the image, the distances between the image pixels are minimized, and the number of accesses to the same position array is increased, thereby maximizing memory access efficiency. We propose a new algorithm that can verify this, and that can also perform performance comparison.

KEYWORD : Volume Rendering, Image Processing, MIP, shear-warp, CPU parallel processing, SIMD, AVX, transposition, max, Interpolation, Memory Reallocation