논문 2009-46SP-2-8

SSE 명령어를 이용한 영상의 고속 전처리 알고리즘

(Fast Image Pre-processing Algorithms Using SSE Instructions)

박 은 수*, 최 학 남*, 김 준 철*, 임 유 청**, 김 학 일***

(Eunsoo Park, Xuenan Cui, Junchul Kim, Yucheong Im, and Hakil Kim)

요 약

본 논문에서는 SSE (Streaming SIMD Extensions) 명령어를 이용한 고속 영상처리 알고리즘을 제안한다. SSE 명령어를 지원하는 CPU는 128비트 크기의 XMM 레지스터를 보유하고 있으며 이에 속한 데이터는 SIMD(Single Instruction Multiple Data) 방식으로 한 번에 병렬로 처리 될 수 있다. 영상처리에서 폭넓게 활용되는 평균 필터, 소벨 수평방향 외곽선 검출, 이진 침식 알고리즘을 SIMD 방식으로 효과적으로 처리 할 수 있는 알고리즘을 제시하였고, 수행 시간을 측정하였다. 보다 객관적인 수행 속도 평가를 위해 현재 많이 사용되고 있는 영상처리 라이브러리와의 수행 속도를 비교하였다. 비교에 사용된 라이브러리는 SISD(Single Instruction Single Data)방식으로 동작하는 OpenCV 1.0, SIMD 방식을 지원하는 고속 영상처리 라이브러리인 IPP 5.2와 MIL 8.0에서 각각 수행 시간을 측정하고 제안하는 알고리즘의 처리 속도와 비교하였다. 실험결과 제안하는 알고리즘은 SISD방식의 영상처리 라이브러리에 비해 평균 8배의 성능향상을 보였으며, SIMD 방식의 고속 영상처리 라이브러리와 비교 하였을 때 평균 1.4배의 성능향상을 보였다. 따라서 제안하는 알고리즘은 고가의 영상처리 라이브러리와 추가적인 하드웨어의 구입 없이도 고속으로 동작해야 하는 실제 영상 처리 어플리케이션에 효과적으로 적용될 수 있음을 보였다.

Abstract

This paper proposes fast image processing algorithms using SSE (Streaming SIMD Extensions) instructions. The CPU's supporting SSE instructions have 128bit XMM registers; data included in these registers are processed at the same time with the SIMD (Single Instruction Multiple Data) mode. This paper develops new SIMD image processing algorithms for Mean filter, Sobel horizontal edge detector, and Morphological erosion operation which are most widely used in automated optical inspection systems and compares their processing times. In order to objectively evaluate the processing time, the developed algorithms are compared with OpenCV 1.0 operated in SISD (Single Instruction Single Data) mode, Intel's IPP 5.2 and MIL 8.0 which are fast image processing libraries supporting SIMD mode. The experimental result shows that the proposed algorithms on average are 8 times faster than the SISD mode image processing library and 1.4 times faster than the SIMD fast image processing libraries. The proposed algorithms demonstrate their applicability to practical image processing systems at high speed without commercial image processing libraries or additional hardwares.

Keywords : SSE, SIMD, fast image processing, IPP, MIL

I. 서 론

최근 디지털 카메라 기술의 발전과 보다 높은 해상도를 갖는 비전 어플리케이션의 요구로 인하여 디지털 영상의 크기가 급속도로 증가하고 있다. 특히, 영상 분석과 관련된 어플리케이션에 사용되는 이미지인 의료영상, 위성영상과 같은 경우 그 크기가 수백에서 수천 MB에 달하며 산업용 제품의 불량을 검사하기 위한 머신 비전 어플리케이션은 생산량의 증가를 위해 이러한 대용량 영상의 분석을 실시간으로 처리 할 수 있는 기술을 요구한다. 대용량의 영상 데이터를 효과적이고 효율적으로 처리하기 위하여 고려되어야 할 점은 영상 데

* 학생회원, *** 정회원, 인하대학교 정보공학과
(Dept. Information Eng., Inha University)
** 정회원, (주)삼성전기
(Samsung electro-mechanics)
※ 본 연구는 2008년도 인하대-삼성전기 산학 협력단의 지원으로 수행 되었습니다.
접수일자 :2008년7월25일, 수정완료일:2008년3월3일

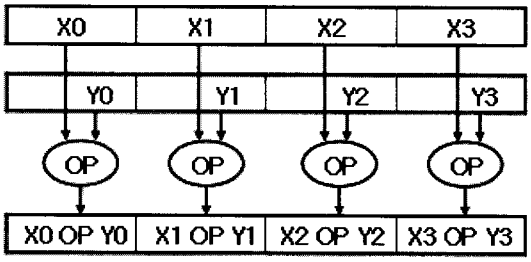


그림 1. SIMD 연산구조
Fig. 1. Architecture of SIMD operation.

이터를 처리할 수 있는 프로세서의 성능과 이 프로세서의 성능을 최적으로 만들어 줄 수 있는 영상 처리 알고리즘이다. 대부분의 영상 데이터 처리를 위한 프로그래밍은 루프 구조를 이용하여 반복적인 명령을 수행하기 때문에 시간 지역성과 공간 지역성을 갖고 있다[1]. 이런 지역성을 효과적으로 처리하기 위한 이미지 처리 하드웨어 구조로 SIMD (Single Instruction Multiple Data)가 있다. SIMD는 그림 1과 같이 하나의 명령어로 다수의 데이터를 처리 할 수 있는 프로그래밍 구조를 말한다. 현재 대표적인 프로세서인 인텔과 AMD의 CPU는 SIMD 연산에만 사용되는 16개의 128비트 레지스터와 SSE (Streaming SIMD Extensions)로 불리는 SIMD 명령어 집합을 보유하고 있다. SSE는 프로그램의 복잡성을 줄이며 다양한 연산을 수행하는 함수를 추가하여 SSE2, SSE3, SSSE3 (Supplemental SSE3), SSE4의 형태로 지속적으로 발전하였고 현재도 보다 높은 효율의 SIMD 연산을 위한 개발이 진행 중이다[2~3].

이러한 발전과 함께 SIMD 구조를 통한 프로그램 성능 향상에 관한 기존의 연구가 있었다[4~5]. 그러나 기존의 연구는 기본적인 순차처리에 대한 처리속도 비교만을 제시하여 실제 어플리케이션에서 가장 많이 사용되는 이미지 처리 라이브러리들 간의 비교 평가가 힘들며, 현재 CPU는 그 당시 보다 높은 처리 속도를 갖고 있기 때문에 개선된 하드웨어 환경에서의 성능평가에 대한 비교가 필요하다. 또한 영상 분석의 정확성을 위해 활용 빈도가 높은 이미지 전처리 알고리즘에 대해서 적절한 SIMD 프로그래밍 방법과 알고리즘을 제시하지 못함으로 다양한 어플리케이션으로의 활용 가능성을 보여주지 못하고 있다.

본 논문에서는 SIMD 연산을 효과적으로 수행 할 수 있는 프로그래밍 방법론을 설명하고 SSE명령어 집합을 이용한 효율적인 영상처리 알고리즘을 제안한다. 알고리즘의 구현은 현재 CPU 시장의 80%를 점유하고 있는 인텔 프로세서를 기반으로 하였다. 본 논문의 Ⅱ장에서는 SSE명령어를 이용한 SIMD 프로그래밍 방법론을 설명하고 Ⅲ장에서는 제안하는 SIMD 영상처리 알고리즘의 구현 방법을 설명한다. Ⅳ장에서는 성능평가를 위해 사용된 영상 처리 라이브러리들을 설명하고 제안하는 알고리즘과의 처리 속도결과를 비교한 후 Ⅴ장에서 결론을 맺는다.

Ⅱ. SSE 프로그래밍

2.1. SSE 기술과 XMM 레지스터

인텔에서는 멀티미디어 고속처리 및 고속 통신을 위하여 PentiumⅡ에서부터 SIMD 연산을 가능하게 하는 MMX (Multi-Media eXtension) 기술을 제공하기 시작하였다. MMX는 64비트 전용 레지스터를 이용하여 8개의 8비트 정수형 데이터를 동시에 처리할 수 있도록 설계 되었다. SSE는 MMX의 확장 버전으로써 PentiumⅢ에서 부터 지원 되었고, 70개의 명령어를 포함하고 있으며, 그림 1과 같이 SIMD 연산을 가능하게 하는 8개의 128비트 XMM 전용 레지스터를 보유하고 있다. PentiumⅣ 에서부터 지원된 SSE의 확장 버전인 SSE2에서는 배 정밀도 실수 데이터 (double precision floating point) 와 8비트, 16비트, 32비트 데이터를 위한 새로운 명령어들이 추가 되었다. 90*nm* 프로세서 기술 환경에서 포함된 SSE3에서는 DSP 연산을 쉽게 할 수 있는 명령어와 캐쉬 명령과 같은 프로세스 관리 명령어들이 추가 되었으며, SSE3의 확장 버전인 SSSE3 (Supplemental SSE3) 에서는 최적화된 16개의 새로운 명령어들이 추가 되었다. 또한 45*nm* 프로세서 환경에서 포함된 SSE4에서는 54개의 소수점 연산을 포함한

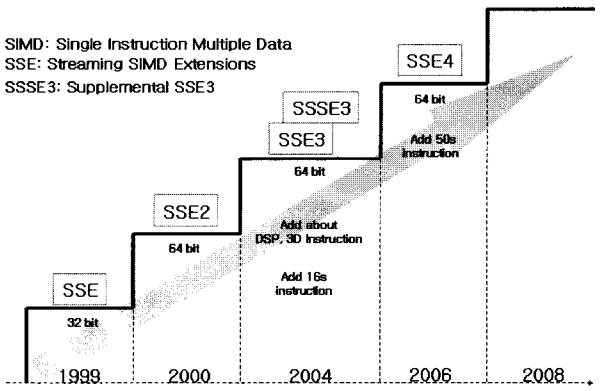


그림 2. SSE 발전과정
Fig. 2. Process of SSE development.

연산들이 추가 되었다. 그림 2는 SSE의 발전과정을 나타낸다.

2.2 SSE 프로그램 시 고려할 점

SIMD연산을 이용하여 어플리케이션의 성능을 높이기 위해서는 SSE 명령어를 사용하여 프로그램을 작성하여야 한다. SSE로 프로그래밍 하기 전 고려해야 할 점은 첫째로 현재의 프로그램이 SIMD의 연산으로 인해 성능 면에서 이점이 있는지의 여부이다. 둘째, 프로그램에서 다루는 데이터의 타입이 정수형인지 실수형인지의 여부와 그 데이터의 크기를 고려해야 한다. 또한 SIMD 연산을 하기 위해 이러한 데이터들을 어떻게 정렬하고 배열 할 것인지의 여부도 중요한 고려사항이다[6]. 일반적으로 SSE 프로그래밍 시 많은 이점을 가지고 있는 코드는 정수나 실수형의 연속적인 배열 형태를 갖고 있으며 루프구조를 이용하여 반복적으로 접근하는 형태의 코드들이다. 기존의 코드를 SSE의 연산 과정으로 바꿀 때 고려해야 할 점과 절차를 그림 3에 나타내었다[6].

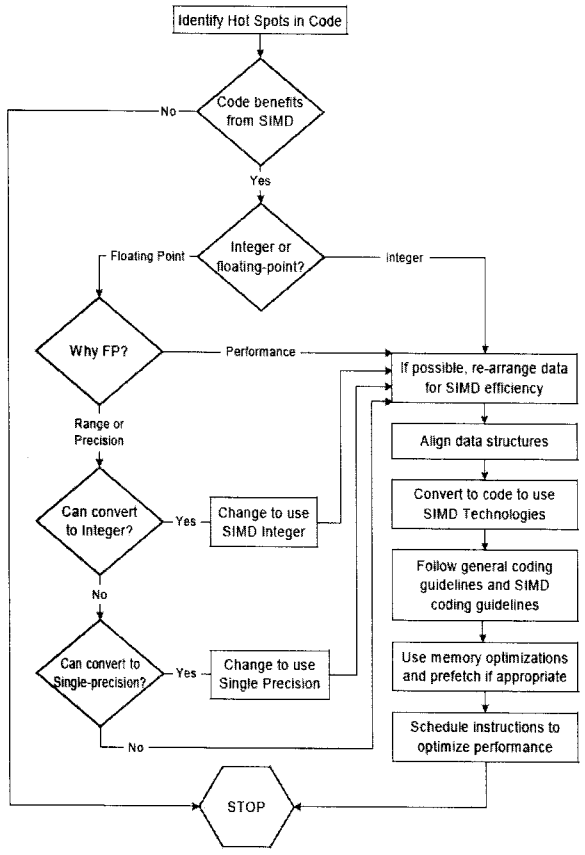


그림 3. SSE 프로그래밍 과정으로의 변환 절차
Fig. 3. Converting process for SSE programming.

2.3. SSE 코딩 기술

SSE 명령어를 이용한 SIMD기술은 새로운 방법의 코딩 알고리즘을 필요로 한다. SIMD 연산의 이점을 충분히 활용하기 위해선 XMM 레지스터의 크기에 맞게 데이터들을 정렬하고 벡터화 시켜야 한다. SSE 명령어를 이용하여 프로그램을 작성하는 방법은 다음과 같이 4가지로 나뉜다.

- 어셈블리 코드로 작성하는 방법
- Intrinsics 코드로 작성하는 방법
- 클래스를 활용하는 방법
- 자동 벡터화를 사용하는 방법

목적 프로세서에 맞추어 하드웨어에 근접한 언어인 어셈블리어로 프로그램을 작성할 경우 프로그래머가 원하는 작업만을 수행 할 수 있도록 코드를 작성할 수 있다. 따라서 오버헤드 없이 프로그램 성능을 충분히 향상 시킬 수 있지만 프로그램 작성이 어렵고 유지 보수가 힘들며 이식성이 떨어지는 단점이 발생한다.

또 다른 프로그래밍 방법인 Intrinsics 명령어로 프로그램을 작성하게 될 경우 C/C++과 같은 스타일로 프로그램을 작성 할 수 있기 때문에 어셈블리 언어보다 상대적으로 쉽게 프로그램을 작성할 수 있다. 또한 Intrinsics는 레지스터의 할당과 명령 스케쥴링과 같은 기능은 고성능의 컴파일러에게 의존하며 프로그래머로 하여금 알고리즘을 보다 쉽게 작성 할 수 있도록 해주며 가독성을 높여준다. 컴파일러는 Intrinsics 명령어를 이에 직접적으로 매칭 되는 어셈블리 명령어로 해석하게 된다. 따라서 Inrinsics를 이용하면 어셈블리 언어로 작성한 프로그램에 가까운 성능을 나타낼 수 있게 되며 계속적으로 발전되는 컴파일러의 지원을 받아 인텔 기반의 구조를 갖는 프로세서간의 이식성을 제공해줄 수 있다.

또 다른 방법인 클래스를 활용하는 방법은 직접적으로 Intrinsics을 사용하는 방법보다 사용이 쉽고 훨씬 자연스러운 C++ 코드 스타일로 작성되기 때문에 유연한 인터페이스를 제공하게 된다.

마지막으로 자동 벡터화를 수행하는 방법이 있다. 이는 C/C++의 스타일로 작성된 코드를 인텔 C++ 컴파일러의 -QAX, -QRESTRICT 스위치를 사용하여 자동으로 SSE의 명령어 코드로 바꿔주는 방법이다. 그러나 데이터 구조의 의존성이나 루프 구조에 따라 이 기능이

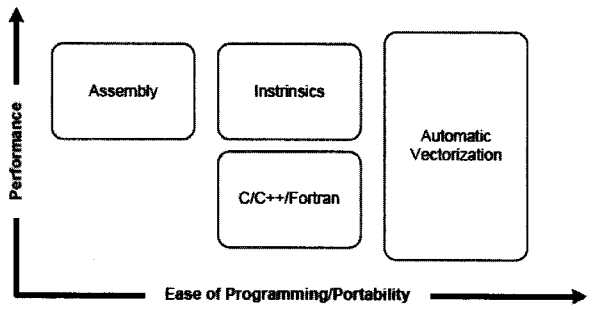


그림 4. 성능 대비 코딩의 이식성과 프로그래밍의 편리성
Fig. 4. Performance versus ease of programming and portability.

활성화 되지 않는 경우가 발생 할 수 있게 된다. 그림 4는 SSE의 구현 방법에 따른 어플리케이션의 성능 대비 코딩의 편리성과 이식성의 상관관계를 보여준다[6].

본 논문에서는 Intrinsics를 이용하여 영상처리 알고리즘을 구현 하였다. 그 이유는 자동 벡터화 방법을 사용 할 경우 영상처리 알고리즘의 데이터 상관성과 알고리즘의 복잡도를 고려하였을 때 높은 성능을 얻기는 힘들 것이라 판단되기 때문이다. 이 방법은 어플리케이션 마다 성능변화의 폭이 크기 때문에 프로세서 기반의 SSE 명령어를 통한 성능향상을 비교하는 작업에 적합하지 않다. 비슷한 이유로 클래스를 활용한 SSE의 프로그래밍 방법은 프로그래머가 원하지 않는 오버헤드를 발생 시킬 수 있기에 배제하였다. 또한 어셈블리 언어로 개발 할 경우 매우 빠르게 발전하는 프로세서의 변화에 따라 이식성이 중요하게 작용할 것이고 프로그램의 유지 보수가 힘들기 때문에 배제하게 되었다.

Ⅲ. 영상처리 알고리즘의 구현

SSE 명령어를 이용한 영상처리 알고리즘 구현 시 정렬되지 않은 영상 데이터는 정렬된 데이터에 비해 빠른 수행 성능을 얻을 수 없기 때문에 영상 데이터를 128비트 크기에 맞게 정렬 시켜야 한다[8]. 본 실험에 사용된 영상은 그레이 스케일의 영상으로 데이터 타입은 [0, 255]의 밝기 값을 표현 할 수 있는 1바이트 크기의 부호 없는 정수형이다. 이 데이터 타입은 SIMD 연산을 위해서 128비트의 데이터 타입인 __128i 형태로 바꾸어야 한다. 128비트 크기의 XMM 레지스터에는 16개의 픽셀 밝기 정보를 로드할 수 있으며 이는 한 번의 연산으로 처리 될 수 있다. 본 논문에서는 SSE 을 적용하여 평균 필터, 소벨 수평방향 외곽선 검출, 이진 침식 알고리즘을 Intirinsics 명령어를 사용하여 구현한다. 일반적인 영상처리 알고리즘인 이웃 픽셀과 그 중심 값을 이용하는 한 픽셀 단위의 영상 데이터 접근 방법으로는 SIMD 연산을 구현 할 수 없기 때문에 새로운 알고리즘을 적용하여야 한다.

3.1 평균 필터

평균 필터는 영상 내에서 잡음을 제거하거나 영상을 몽롱화 하는데 주로 이용된다. 본 연구에서는 3×3 평균 필터를 SIMD 연산을 이용하여 수행 할 수 있도록 구현 하였다. 실제 평균 필터의 응답을 수식 (1)에 나타내었다.

$$R = \frac{1}{9}\sum_{i=1}^{9} z_i \tag{1}$$

평균 필터는 수식 (1)에서와 같이 나눗셈 연산이 필요하기 때문에 실수형 데이터로 알고리즘이 구현 되어야 하며 이 데이터에 맞는 SSE 명령어를 사용하여야 한다.

3.1.2 32비트 실수 데이터를 이용한 평균 필터

처리 되어야 할 영상이 한 픽셀 당 32비트의 크기를 갖고 있으며 128비트로 정렬되어 있다고 가정한다. 데이터가 픽셀 당 32비트 크기를 갖고 있지 않다면 32비트 데이터로의 타입 변환과 128비트 단위로의 정렬 과정이 포함된다. 128비트 XMM 레지스터에는 32비트 픽셀이 4개가 포함 될 수 있다. 알고리즘의 수행 과정을

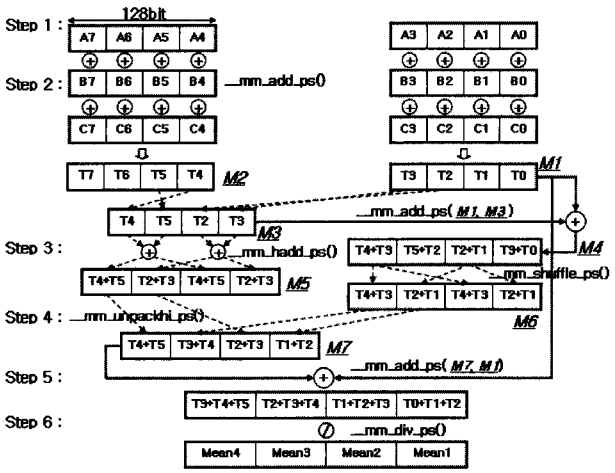


그림 5. 4개의 실수 데이터를 이용한 평균 필터
Fig. 5. Mean Filter using 4 floating point data.

그림 5에 나타내었다. 그림 5의 *A, B, C* 로 나타낸 배열은 평균 필터를 통과하게 될 이미지의 행을 말한다. 알파벳 뒤에 붙은 인덱스는 픽셀 데이터의 연속성을 나타내기 위해 사용되었다.

전체적인 과정은 열 방향으로 픽셀들을 더한 후 이 결과를 다시 행 방향으로 더한다. 이렇게 평균 필터의 마스크에 속하는 9개의 픽셀들을 모두 더한 후 9로 나누어 평균값을 구한다. 정확한 연산을 위해 Intrinsics명령어인 _shuffle_ps와 같은 데이터 재배열 명령문을 사용하여 연산하였다. 아래에 이러한 과정을 *Step*으로 나누어 설명하였다.

Step 1 (픽셀 데이터의 로드) : SSE Intrinsics의 _mm_load_ps 함수를 이용하여 2개의 128비트 XMM 레지스터에 각각 4개씩의 픽셀 데이터를 로드한다.

Step 2 (세로축 픽셀 데이터의 합) : 이미지 데이터의 3개의 행을 더해서 세로 축 방향의 합을 계산한다. 이 결과를 그림 5에서 *M1, M2*로 나타내었다.

Step 3 (가로 축 픽셀 데이터의 합) : *M1, M2* 를 이용하여 *M3* 를 만든다. 이 *M3* 레지스터를 _mm_hadd_ps 명령어를 이용하여 가로 축 으로의 합을 계산한다. _mm_hadd_ps 명령어를 사용한 결과는 그림 5에서 *M5* 로 나타내었다.

Step 4 (데이터의 재배열) : *M5* 와 *M6* 의 레지스터에 _mm_unpackhi_ps 명령어를 이용하여 재배열된 레지스터 *M7* 을 만든다.

Step 5 (최종 3×3 마스크의 합) : *Step4* 에서 만들어진 *M7* 과 *Step2* 에서 만들어진 *M1*을 더하여 최종 3×3 마스크 내에 위치한 픽셀 데이터의 합을 구한다.

Step 6 (마스크 합의 나눗셈) : *Step5* 에서 만들어진 4개의 마스크 합의 결과를 9로 나누어 최종 평균 필터의 응답을 구한다.

3.1.2. 8비트 정수 데이터를 이용한 평균 필터

8비트 정수데이터를 이용하여 평균 필터를 구현 할 경우 나눗셈의 연산 결과로 발생되는 결과 데이터는 실수 형이기 때문에 이를 적절하게 변환해 주어야 한다. 8비트의 데이터일 경우 128비트 XMM 레지스터에 총 16개의 픽셀이 로드 될 수 있다. 한 픽셀의 크기가 32비

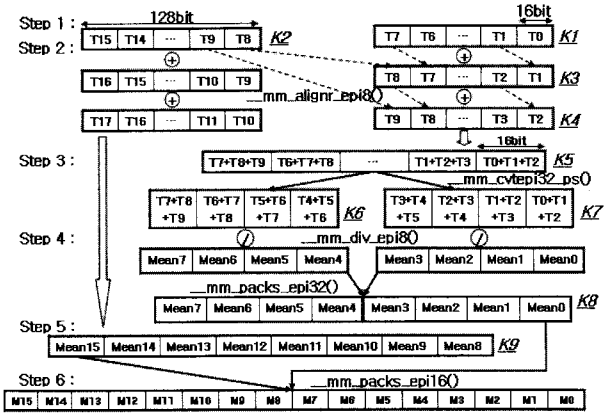


그림 6. 16개의 정수 데이터를 이용한 평균 필터
Fig. 6. Mean Filter using 16 integer data.

트의 경우 보다 더 많은 데이터를 로드 하고 처리 될 수 있기 때문에 처리 시간을 많이 줄일 수 있다. 전체적인 8비트 정수 데이터를 이용한 평균 필터의 구현 과정을 그림 6에 나타내었다. 먼저 16개의 8비트 픽셀 데이터를 XMM 레지스터로 로드 한 후 unpack 명령어를 통하여 8개의 16비트 픽셀 데이터 형태로 바꾸어 준다. 이렇게 바뀐 8개의 픽셀 데이터를 그림 5의 *Step2* 에서 *M2* 레지스터를 구한 것처럼 열 방향으로 더해준다. 이 결과를 그림 6에서 *K1*과 *K2*로 나타내었다. 그림 5에서 행 방향으로 덧셈하기 위하여 가로 방향 덧셈 연산자를 사용했지만 8비트 정수 데이터를 이용한 알고리즘 경우 XMM 레지스터를 한 픽셀 데이터 크기만큼 이동 시킨 후 열 방향으로 더한다. 이 경우 앞선 평균 필터의 경우보다 구현에 있어서 직관적이며 많은 데이터를 로드 할 수 있어서 빠른 성능을 보인다. 그림 6의 아래에 전체 알고리즘 과정을 설명하였다.

Step 1 (XMM 레지스터로 데이터 이동) : unpack 명령어를 이용하여 128비트로 정렬 된 8비트 정수형 데이터를 128비트로 정렬 된 16비트 정수형 데이터로 바꾸어 준다. 이를 그림 6의 *K1* 과 *K2* 로 나타내었다.

Step 2 (픽셀 데이터의 이동) : 이미지 데이터의 열 방향 덧셈을 위해서 그림 6의 *K3* 와 *K4* 레지스터를 만들어야 한다. *K3* 레지스터를 만드는 방법은 _mm_alignr_epi8 명령어를 이용하여 만들 수 있다. 이 명령어는 내부적으로 *K1* 과 *K2* 레지스터를 합쳐서 임시 레지스터 안에서 256비트의 크기로 만든 후 이 레지스터를 입력 받은 바이트 크기만큼 이동 시킨다. 그리고 이

레지스터의 하위 128비트만을 결과로 내놓게 된다. *K3* 를 만든 것과 같은 방식으로 *K3* 와 *K2* 레지스터를 이용하여 *K4* 레지스터를 만들 수 있다.

Step 3 (최종 3×3 마스크의 합) : 그림 6의 *K1, K3, K4* 를 세로 축 방향으로 더한다. 이 결과는 최종 마스크에 속한 9개의 픽셀들의 합이며 이를 *K5* 레지스터로 나타내었다.

Step 4 (나눗셈) : *K5* 레지스터는 8개의 16비트 데이터이다. 이를 _mm_ cvtepi32 명령어를 이용하여 32비트의 크기의 실수형 데이터 *K6* 와 *K7* 레지스터로 분리하고 여기에 나눗셈을 수행 한다. 이렇게 나누어진 결과를 다시 _mm_packs_epi32 명령어를 이용하여 128비트 크기의 *K8* 레지스터로 합쳐준다.

Step 5 (*Step* 의 반복) : *Step1* 부터 *Step4*의 과정을 *K2* 레지스터와 그 하위 픽셀 레지스터들에 똑같이 적용하여 *K9* 레지스터를 만든다.

Step 6 (2개의 결과 레지스터의 합침) : *K9* 레지스터와 *K8* 레지스터를 _mm_packs_epi16 명령어를 이용하여 합친다. 이 결과는 최종 16개의 평균 필터 결과를 나타낸다.

3.2 소벨 수평방향 외곽선 검출

소벨 연산자는 영상처리 과정 중 대상 이미지의 외곽선를 검출하는데 사용된다. 소벨 연산의 결과는 수직방향의 기울기와 수평방향의 기울기를 구한 후 이 기울기의 크기를 구함으로써 얻어 질 수 있다. 수학적으로 수직과 수평방향의 기울기는 수식 (2)와 같이 표현 할 수 있다.

$$G_x=\begin{bmatrix}-1 & 0 & +1\\ -2 & 0 & +2\\ -1 & 0 & +1\end{bmatrix}*A \text{ and } G_y=\begin{bmatrix}+1 & +2 & +1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{bmatrix}*A \quad (2)$$

여기서 A는 처리해야할 이미지를 나타내고 G_x와 G_y는 각각 수직 방향과 수평 방향의 기울기를 나타낸다. G_x와 G_y를 이용하여 외곽선의 크기인 G를 수식 (3)과 같이 얻을 수 있다.

$$G=\sqrt{G_x^2+G_y^2} \quad (3)$$

실제 구현에서는 수식 (3)을 구하는 것은 계산적으로 복잡하고 수행 시간이 오래 걸리기 때문에 수식 (4)와 같이 절대 값을 이용하는 방법을 주로 사용하게 된다[7].

$$G=|G_x|+|G_y| \quad (4)$$

본 논문에서는 소벨 알고리즘의 구현을 $|Gy|$, 즉 수평방향의 기울기의 크기를 구하는 것으로 한정하였다. 그 이유는 성능 비교에 사용되는 라이브러리들 중에는 최종 소벨 결과인 수식 (4)의 G 를 구하는 함수가 존재하지 않고 각 방향 $|G_x|$와 $|G_y|$의 연산만을 지원하는 것이 있기 때문이다. 따라서 수식 (4)의 G를 구하는 과정은 프로그래머가 따로 작성해줘야 한다. 이는 G 연산에 대한 프로그래머의 개입으로 인해 정확한 라이브러리의 성능측정을 어렵게 하는 요인이 된다.

3.2.1 16비트 정수 데이터를 이용한 소벨 수평방향 필터

16비트 정수 데이터를 사용하기 때문에 총 8개의 픽셀을 한 번에 로드 할 수 있다. 이미지가 16비트 크기로 정렬되었을 때의 소벨 수평 방향 필터의 구현상 이점은 소벨 마스크의 계수 크기가 크지 않기 때문에 연산과정에서의 더 큰 데이터 타입으로의 변환이 필요 없다는 점이다. 전체적인 소벨 알고리즘 처리 과정은 그림 7과 같다.

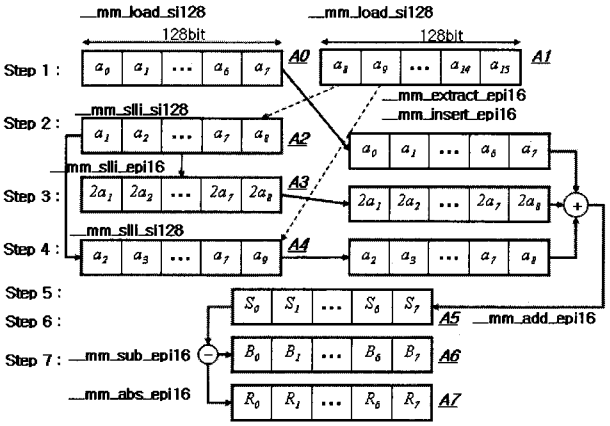


그림 7. 8개의 정수 데이터를 이용한 소벨 수평 방향 필터

Fig. 7. Sobel horizontal Filter using 8 integer data.

그림 7의 상단 *Step1* 은 128비트 크기 데이터 2개를 불러 온 모습을 나타낸다. 처리과정은 아래와 같은 *Step* 으로 나뉜다.

Step 1 (XMM 레지스터로의 데이터 이동) : 2개의 XMM 레지스터에 각각 8개의 픽셀들을 로드 한다. 그

림 7의 *A0, A1* 레지스터가 이를 나타낸다.

Step 2 (픽셀 이동과 삽입) : 로드된 *A0* 레지스터를 _mm_sllisi128 명령어를 이용하여 한 픽셀 크기만큼 이동시키고 그 자리에 빈칸을 생성해 둔다. *A1* 레지스터에 _mm_extract_epi16 명령어를 이용하여 하나의 픽셀 a_8 을 추출 하고 이 추출된 픽셀 데이터를 _mm_insert_epi16을 이용하여 앞서 발생된 빈칸에 채워 넣어 *A2* 레지스터를 완성한다.

Step 3 (소벨 필터의 계수 2를 곱한다) : *A2* 레지스터에 2를 곱한다. 2의 곱을 계산하기 전에 미리 2의 값을 8개 포함하고 있는 XMM 레지스터를 만들어 두어야 한다. 한 번의 연산으로 8개의 2의 곱을 얻을 수 있으며 이를 그림 7의 *A3* 레지스터로 표현 하였다.

Step 4 (픽셀의 이동과 삽입): *A2* 레지스터를 한 픽셀 크기만큼 이동 시킨 후 빈칸을 생성하고 *Step 2* 에서와 같이 빈칸을 *A1* 레지스터를 이용하여 채워 넣는다. 이 결과로 그림 7의 *A4* 레지스터를 생성할 수 있다.

Step 5 (소벨 수평방향 필터의 상단 부분 계산) : 수식 (2)의 소벨 수평 방향 마스크의 상단 [+1, +2, +1]을 계산하는 과정이다. 그림 7의 *A0, A3, A4*를 열 방향으로 더하면 그 결과가 바로 소벨 수평 방향 마스크의 상단 부분의 계산 결과이며 이를 *A5* 레지스터로 표기했다.

Step 6 (Soel 수평방향 마스크의 하단 부분계산) : 위의 [*Step 1-Step 5*] 를 상단 방향 마스크를 계산했던 데이터보다 2행 아래에 있는 픽셀데이터에 적용하여 *A6* 레지스터를 만든다.

Step7 (최종 계산) : *A5* 와 *A6*를 _mm_sub_epi16 명령어를 이용하여 뺄셈을 취한 후 _mm_abs_ epi16 명령어를 이용하여 절대값을 취해준다. *A7* 레지스터는 8개의 소벨 수평방향 필터의 응답, 즉 수식 (4)의 $|G_y|$ 를 계산 한 것이다.

3.2.2 8비트 정수 데이터를 이용한 소벨 수평방향 필터

한 픽셀 당 8비트로 표현되는 이미지는 평균 필터를 구현 했을 때와 마찬가지로 한 번에 16개의 픽셀을 로드 할 수 있다. 소벨 수평방향 필터의 처리 속도를 높이기 위하여 SSSE3에 추가된 명령어인 곱과 합 명령어와 XMM 속한 데이터를 순서를 쉽게 변형 시킬 수 있는 명령어를 활용하여 수행 시간을 단축시켰다. 그림 8은 소벨 수평방향 필터의 상단 부분 연산의 결과 중 홀수 인덱스를 갖는 결과를 생성하는 과정을 나타낸다. 전체

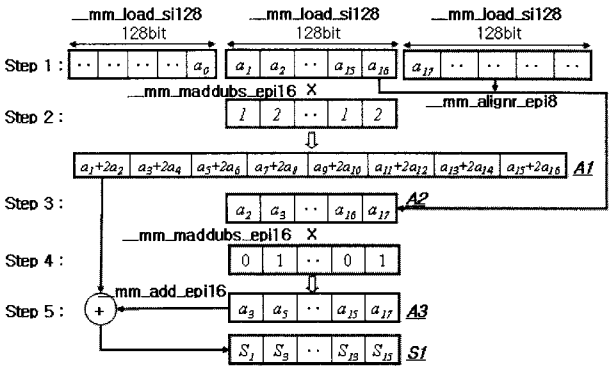


그림 8. 16개의 정수와 곱과 합 명령어를 사용한 소벨 수평방향 필터

Fig. 8. Horizontal Sobel Filter using 16 integer and Multiply and Add instruction.

적인 수행 과정을 아래에 설명 하였다.

Step 1 (XMM 레지스터로 데이터 이동) : 3개의 XMM 레지스터를 로드한다. 3개를 로드하는 이유는 양쪽의 가장 자리 부분을 처리해주기 위함이다. 가운데 로드된 16개의 픽셀에 대해서 수평방향 소벨 필터의 상단 부분을 계산하게 된다.

Step 2 (곱과 합) : 상수 1, 2를 반복적으로 갖는 128비트 XMM 레지스터를 준비한 후 *Step1*에서 로드된 픽셀들과 곱과 합 연산을 수행한다. 이 결과를 그림 8의 *A1* 레지스터로 표현 하였다.

Step 3 (데이터 재배열) : _mm_alignr_epi8 명령어를 이용하여 *A2* 레지스터를 만든다.

Step 4 (곱과 합) : 상수 0, 1을 반복적으로 갖는 128비트 XMM 레지스터를 준비한 후 *A2* 레지스터와 연산한다. 그 결과를 그림 8의 *A3*로 표현 하였다.

Step 5 (홀수 인덱스의 소벨 수평방향 마스크의 상단 계산) : *A1* 레지스터와 *A3* 레지스터를 더하면 홀수 인덱스를 갖는 소벨 수평방향 필터의 상단 부분 계산 결과를 얻을 수 있다. 이 연산 결과 생성된 레지스터를 *S1*으로 표시 하였다.

Step 6 (상수계수의 순서를 바꾸어 앞의 *Step*을 반복 적용) : 위의 단계를 수행한 결과 홀수 인덱스를 갖고 있는 *S1* 레지스터를 얻을 수 있다. 위 단계 중 *Step2*의 곱과 합 연산의 계수를 16개의 2, 1을 계수로 하는 레지스터로 바꾸어 적용하고, *Step3*의 데이터 재배열 연산에서는 그림 8의 *Step1* 부분에서 가장 왼쪽에 위치한 128비트 XMM 레지스터와 _mm_alignr_epi8 명령어를 수행한다. *Step4* 에서는 0, 1의 연속적인 계수가 아닌

1, 0의 연속적인 계수를 갖는 XMM 레지스터와 곱과 합 연산을 수행하고 *Step5* 단계를 거치면 그림 9의 *Step6*의 S2 와 같은 짝수 인덱스를 갖는 레지스터를 얻을 수 있다.

Step 7 (소벨 수평방향 마스크의 하단 부분 계산) : 소벨 수평 방향 마스크의 하단 부분을 계산하기 위해 그림 8의 *Step1* 에서와 같이 소벨 수평방향 필터 마스크의 하단부분에 해당하는 2행 아래의 3개의 레지스터를 읽어 들인 후 [*Step1* -*Step6*]의 과정을 똑같이 적용한다. 그 결과를 그림 9의 *B1, B2* 레지스터로 나타내었다.

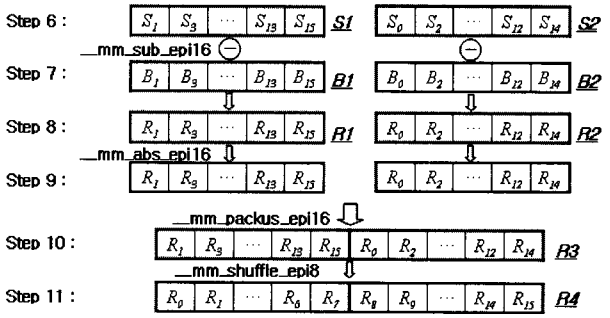


그림 9. 최종 소벨 수평방향 필터의 결과
Fig. 9. Final result of Horizontal Sobel Filter.

Step 8 (레지스터의 차를 계산) : *S1* 과 *B1* 레지스터와의 차를 구하고 *S2* 와 *B2* 레지스터와의 차를 구한다. 그 결과를 그림 9의 *R1, R2* 레지스터로 표현 하였다.

Step 9 (절대치를 취함) : 이 *R1, R2* 레지스터에는 소벨 수평방향 마스크의 계산결과가 저장되어 있다. 수식 (4)의 $|G_y|$를 연산하기 위해서 절대치를 취한다.

Step 10 (16개의 데이터 형태로 만듦) : 2개의 XMM 레지스터를 하나로 합쳐서 16개 픽셀 데이터를 갖고 있는 XMM 레지스터 하나로 만든다. 그 결과는 그림 9의 *R3* 레지스터로 표현 하였다. *R3* 레지스터는 짝수 인덱스와 홀수 인덱스를 번갈아 가지고 있다.

Step 11 (데이터 재배열) : _mm_shuffle_epi8 명령어를 이용하여 데이터를 순서대로 재배열 한다. 이는 16개의 마스크가 한 번에 처리된 최종 소벨 수평 방향 필터의 결과를 나타내며 *R4*로 표현하였다.

위의 알고리즘을 통하여 SSSE3에 새롭게 추가된 명령어를 활용하여 더 빠른 성능을 보이는 소벨 수평 방향 필터제작이 가능함을 보일 수 있었다.

3.3 이진 팽창 & 침식

이진 팽창(Binary dilation)과 침식(erosion)은 형태학 기반 이미지 처리의 가장 기본적인 알고리즘이다. 두 가지 팽창과 침식 연산을 기초로 하여 열림(Opening), 닫힘(Closing). 경계 추출, 영역 채움, 볼록 깍지(Convex hull), 농밀화, 세선화 등 다양한 영상처리 알고리즘들을 구현 할 수 있게 된다. 이진 팽창과 침식 연산의 수학적인 정의는 아래의 수식 (5), (6)과 같이 집합 이론을 이용하여 표현 할 수 있다. 여기서 A는 영상 내의 처리 대상이 되는 이미지를 가리키고 B는 사용되는 구조요소 이며, z는 구조요소 B를 이동 시킬 때의 결과 영상 내의 좌표를 나타낸다. 이진 팽창연산은 영상 A의 경계부분을 확장하고, 침식 연산은 경계부분을 축소하는 성질을 가지고 있다.

$$A \oplus B = \{z \mid (\hat{B})_z \cap A \neq \phi\} \tag{5}$$

$$A \ominus B = \{z \mid (B)_z \subseteq A\} \tag{6}$$

본 연구에서는 이진 팽창과 침식 연산을 구현하기 위해서 그림 10과 같은 가장 일반적인 3×3 정사각형 형태의 구조요소를 사용하였다.

침식 연산의 경우 그림 10의 구조 요소 마스크가 이미지를 지나가면서 해당 마스크의 범위에 속하는 값들의 *AND* 연산을 하게 되고 이 연산 결과를 마스크 중심에 위치한 이미지 픽셀에 반환한다. 팽창일 경우 *AND* 연산이 *OR* 연산으로 바뀌면 되기 때문에 본 논문에서는 침식 알고리즘만을 설명한다. 알고리즘이 단순하고 똑같은 연산을 여러 데이터에 적용하기 때문에 SIMD로 구현하기에 적합하다.

1	1	1
1	1	1
1	1	1

그림 10. 실험에 사용된 구조요소
Fig. 10. Structuring element used in experiment.

3.3.1 8비트 데이터를 이용한 침식 알고리즘

그림 10과 같은 3×3 구조요소를 통과하는 이미지 픽셀들을 P_x로 표현하였을 때 이를 그림 11에 나타내었다. 팽창 알고리즘을 통과한 그림 11의 이미지들은 *OR* 연산을 통해서 수식 (7)과 같이 구현 될 수 있고, 침식

P_1	P_2	P_3
P_4	P_5	P_6
P_7	P_8	P_9

그림 11. 3×3 구조요소를 통과하는 3×3 이미지 픽셀
Fig. 11. 3×3 image pixels passing through 3×3 Structuring element.

은 *AND* 연산을 통해서 수식 (8)을 통해 구현 될 수 있다.

$$P_S = \bigcup_{x=1}^{9} P_x \tag{7}$$

$$P_S = \bigcap_{x=1}^{9} P_x \tag{8}$$

침식 알고리즘 구현의 핵심은 9개의 픽셀의 *AND* 연산을 구하는 것이다. 비트 연산이기 때문에 128비트 레지스터로 16개의 픽셀이 로드 된 이후 데이터 타입 변환이 필요 없다. 전체적인 알고리즘의 수행 과정은 아래와 같다.

Step 1 (XMM 레지스터로 두 쌍의 128비트, 16개의 데이터를 로드) : 그림 12와 같이 두 쌍의 128비트 레지스터 *P1*과 *P2*를 로드한다. 한 개의 레지스터엔 16개의 픽셀 데이터가 있다.

Step 2 (행 방향 *AND* 연산을 위한 데이터 재배열) : 행 방향의 *AND* 연산을 위해서 그림 12의 *P2* 레지스터를 로드 한 후 *P1* 과 _mm_alignr_ epi16 명령어를 이용하여 *P3* 와 *P4* 레지스터를 생성한다.

Step 3 (행 방향 *AND* 연산 수행): 행 방향 *AND* 연산을 _mm_and_epi128 명령어를 이용하여 수행한다. 이 결과 생성된 *P4*는 행 방향으로 연속된 3개의 픽셀 값에 대한 *AND* 연산결과를 저장하고 있다.

Step 4 (2번째 행과 3번째 행에 이전 *Step* 반복 적용) : *Step1-Step3* 의 과정을 두 번째 행과 세 번째 행에 반복 적용하여 그림 12의 *T*와 *U* 레지스터를 생성한다. 이들 레지스터 한 칸은 모두 행 방향으로 연속된 3개의 픽셀의 *AND* 연산을 저장하고 있다.

Step 5 (열 방향으로의 *AND* 연산) : 3개의 행에 대하여 각각 연속된 3픽셀의 *AND* 연산의 결과를 *S, T, U*가 저장하고 있으므로 이 세 개의 레지스터를 *AND* 연산한다면 열 방향으로의 *AND* 연산이 되어 최종 침식의 결과를 얻을 수 있다. 이를 그림 12의 *V* 레지스터로 나타내었다.

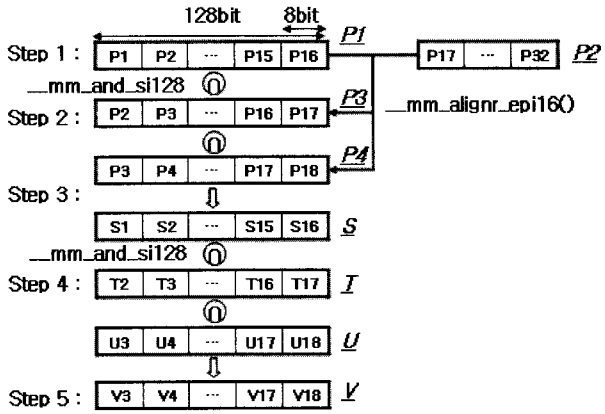


그림 12. 8비트 데이터를 이용한 SIMD 침식 알고리즘
Fig. 12. SIMD Erosion algorithm using 8bit data.

Ⅳ. 성능 평가

4.1 성능평가에 사용된 영상 처리 라이브러리

실제 대부분의 상용화된 영상 처리 어플리케이션 들은 라이브러리를 이용한다. 이는 보다 안정적으로 프로그램을 수행 할 수 있으며 고속의 성능을 보장 받을 수 있기 때문이다. 본 논문의 실험에 사용된 영상 처리 라이브러리는 MIL 8.0, IPP 5.2, OpenCV 1.0이며 이 라이브러리들의 간략한 특징들을 아래에 설명한다.

4.1.1 MIL

MIL (Matrox Image Library)은 매트록스 사에서 개발하고 판매하는 고성능 영상처리 라이브러리이다. MIL에 사용된 이미지 처리 함수들은 인텔의 MMX, SSE, SSE2를 이용하여 개발 되었다. 또한 오딧세이와 같은 영상처리 전용보드를 제공하며 이 보드에 최적화된 ONL(Odyssey Native Library) 을 활용하면 보다 높은 성능향상을 볼 수 있다. MIL의 영상처리 기능은 점대 점(point-to-point), 통계적, 공간적, 형태학적, 계층적(geometric) 변환 및 고속 푸리에 변환 등의 다양한 기능을 포함하고 있다. 영상 분석 기능은 블랍 분석, 켈리브레이션, 광학 문자 인식(OCR), 계층적 패턴매칭, 외곽선 분석 등이 가능하다[9].

4.1.2 IPP

IPP(Integrated Performance Primitives)는 MMX,

SSE의 SIMD 기술을 이용하여 인텔에서 개발된 최적화된 고성능 연산 라이브러리이다. IPP는 영상처리뿐만 아니라 신호처리, 행렬처리, 3차원 자료처리 등을 위한 1000여개의 최적화된 함수들을 제공한다[10]. 또한 라이브러리 구매 비용이 비싸지 않고 이 라이브러리를 활용하여 개발된 제품에 대한 로열티를 따로 받지 않으므로 실제 상업용 어플리케이션 제품 개발 시 많이 사용 되고 있다.

4.1.3 OpenCV

OpenCV는 인텔에서 개발한 공개용 영상처리 라이브러리로 "Open Source Computer Vision Library"의 약자이다. OpenCV는 영상 처리, 컴퓨터 비전, 기계 학습 등 매우 다양한 기능을 가지고 있으며 이를 간편하게 조작할 수 있다[11]. OpenCV가 설치된 플렛폼에 IPP가 설치되면 OpenCV의 함수 처리 속도가 증가하게 되는데 이는 함수가 수행되면서 동적으로 연동된 IPP 함수를 호출하기 때문이다. 또한 OpenCV는 소스가 공개되어 있으므로 개발이 편리하고 다양한 의견을 교류할 수 있어 많은 비전 관련 연구자들이 이용하고 있다.

4.2 이미지 처리 라이브러리들과의 성능 비교

제안하는 알고리즘의 성능을 평가하기 위해 사용된 CPU는 인텔 코어 2 듀오 데스크탑 프로세서 이며 클럭 스피드는 2.40*GHz* 이다. 알고리즘이 적용되는 이미지는 크기가 4057 × 4048인 대용량의 PCB(Printed Circuit Board) 이미지를 사용하였고 실험에서는 OpenCV 라이브러리 외에 SISD(Single Instruction Single Data) 방식으로 동작하는 최적화된 순차 코드를 작성하여 제안하는 SIMD 방식으로의 프로그램이 SISD 방식에 비해 어느 정도의 속도 향상을 보이는지를 확인하였다. 또한 상용 라이브러리인 MIL, IPP는 내부적으로 SIMD 기반으로 프로그래밍 된 고속처리 라이브러리이기 때문에 제안하는 SIMD 알고리즘과의 성능비교에 적합하다.

표 1에 각 라이브러리들을 이용한 영상처리 알고리즘의 처리 속도를 *ms* 단위로 나타내었다. 표 1에는 열림 연산과 닫힘 연산의 수행시간이 추가 되었다. 열림 연산의 경우 팽창 연산을 적용 후 그 결과에 침식 연산을 적용하는 알고리즘이고, 닫힘 연산의 경우 침식 연산을 적용한 결과에 팽창연산을 적용한 연산이다. 따라서 표1의 열림과 닫힘 연산의 처리 시간은 제안하는 팽창과 침식 연산을 순차적으로 적용한 후 이 처리 시간을 측정 하였다. 평균 필터와 소벨 수평방향 외곽선 검출의 경우 2가지 방법으로 구현하였기 때문에 본 논문에서의 첫 번째 구현을 *A*로 두 번째 구현을 *B*로 표기하였다. 표 1을 보면 2가지 구현 모두 SISD 방식으로 동작하는 최적화된 순차코드와 OpenCV에 비해 평균 필터에서는 2배 이상의 성능 향상을 보였고 소벨 수평방향 외곽선 검출에서는 12배 이상의 성능 향상을 보였다. 침식과 팽창의 경우 OpenCV에 비해 10배 정도 빠른 성능을 보였으며 최적화된 순차코드에 비해 5배 정도의 성능 향상을 나타냈다. 전체적인 SISD 영상 처리 방법과 비교했을 때 제안하는 SIMD 방식은 평균적으로 8배 이상의 성능향상을 가져왔다.

표 1. 수행 시간 비교
Table 1. Performance comparison.

	Sequential	OpenCV 1.0	MIL 8.0	IPP 5.2	SSE	
Mean	117.29	65.93	37.39	7.64	*A*	*B*
					50.99	30.28
Sobel Horizontal	168.89	192.06	30.12	14.81	*A*	*B*
					23.37	14.32
Erosion	56.62	104.33	56.62	11.49	10.47	
Dilation	70.4	101.56	15.05	11.6	10.1	
Opening	134.79	206.11	34.45	24.44	20.41	
Closing	135.68	205.8	34.39	23.81	20.21	

표 1을 통해 상용 라이브러리이며 SIMD 기술을 사용한 고속의 MIL과 IPP는 SISD 방식의 OpenCV 라이브러리 보다 훨씬 빠른 속도를 보임을 알 수 있다. 또한 실험에 사용된 라이브러리 중 IPP가 가장 높은 성능을 보임을 볼 수 있다. 제안하는 SIMD 알고리즘과 고속 SIMD 이미지 처리 라이브러리의 수행 시간을 비교하였을 때 침식연산에서 MIL 라이브러리에 비해 5배 이상의 가장 큰 성능 향상을 확인 할 수 있었다. 또한 제안하는 알고리즘은 IPP 라이브러리와는 큰 성능차이를 볼 수 없었지만 평균 필터를 제외하고 모두 앞선 성능을 나타냄을 확인 할 수 있었다. 제안하는 알고리즘은 MIL과 IPP의 SIMD 연산에 비하여 평균 1.4배 정도의 성능 향상을 가져왔다.

평균 필터 알고리즘의 경우 제안하는 방법은 결과의 정확도를 위해 실수형 데이터로 변환하는 과정에서 레지스터의 많은 이동을 초래하였기 때문에 만족할 수준의 성능향상을 기대하기 힘들었다. 결과의 정확도를 낮

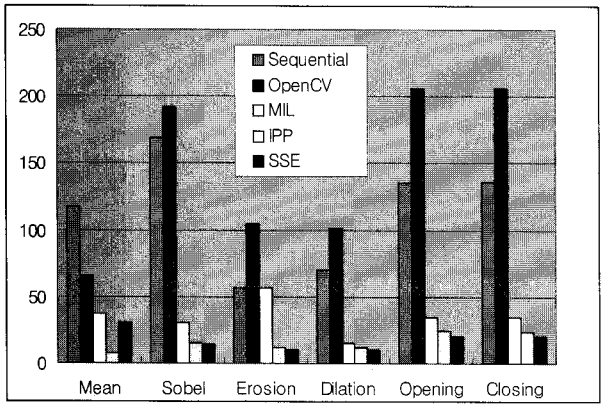


그림 13. 수행 성능 평가 그래프
Fig. 13. Graph of performance evaluation.

추어 정수형으로 처리 한다면 처리 시간을 많이 향상 시킬 수 있을 것이다. 그림 13은 성능 비교를 쉽게 확인하기 위해서 표1을 그래프로 표현해 보았다.

그림 13의 가로축은 각각의 실험에 사용된 이미지 처리 알고리즘들을 나타내고 세로축은 *ms* 단위의 시간을 나타낸다. 각 알고리즘에 5개의 막대그래프가 있는데 순서대로 최적화된 순차처리, OpenCV, MIL, IPP, 제안하는 SSE 구현 방법을 나타낸다. 그림 13을 보면 제안하는 SSE 명령어를 이용한 SIMD 알고리즘 방법이 상용 라이브러리와 순차처리 코드에 비해 빠르게 영상처리 알고리즘을 수행 할 수 있음을 보여준다.

V. 결 론

본 논문에서는 SSE 명령어를 이용한 고속 영상처리 알고리즘을 제안하였다. SIMD 방식으로 이미지를 처리하기 위해서는 기존 영상처리 알고리즘 효과적인 변형이 필요하다. 일반적으로 많이 사용되는 영상처리 알고리즘인 평균 필터, 소벨 필터, 침식과 팽창을 SIMD 방식에 맞는 알고리즘으로 변형하였고 이 과정을 레지스터 이동에 관한 그림으로 설명하였다. 제안하는 SIMD 알고리즘과 SISD 알고리즘의 처리 속도를 비교하기 위해서 최적화된 순차처리 코드와의 수행 속도를 비교 하였고, 실제 적용 가능성을 보이기 위해 상용화된 SIMD 방식의 MIL, IPP 라이브러리와의 처리 속도를 비교를 하였다.

본 논문에서 제안하는 SIMD 알고리즘은 SISD 방식보다는 전체적으로 8배의 성능향상을 가져왔으며, SIMD 방식의 고속 라이브러리와는 평균 필터를 제외하고 모두 앞선 수행 시간을 보여 실제적인 영상처리 어플리케이션에 적용 가능함을 증명하였다. 따라서 영상처리 라이브러리에 의존도를 줄여 어플리케이션의 유연성을 확보할 수 있으며 빠른 수행 속도를 갖는 어플리케이션 제작도 가능함을 보였다. 이를 활용하면 고가의 라이브러리와 추가적인 하드웨어의 구입 없이 영상처리 알고리즘 제작이 가능하기 때문에 실제 어플리케이션의 경쟁력 확보에 도움이 될 수 있을 것이라고 판단된다.

향후 평균 필터의 나눗셈 연산 알고리즘의 성능을 개선하고 SSE4의 추가된 기능을 활용하여 수행 시간을 향상시킬 계획이다. 또한 제안하는 알고리즘에 OpenMP와 MPI를 통한 병렬처리 방법을 추가한 보다 빠른 영상처리 알고리즘을 연구 중이다.

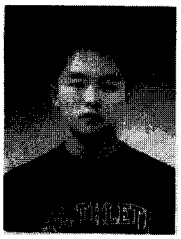
참 고 문 헌

[1] Randle Hyde, *WRITE GREAT CODE*, No Starch Press, 2004.
[2] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture", May, 2007.
[3] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2: Basic Architecture", May, 2007.
[4] Johan Skoglund and Michael Felsberg, "Fast image processing using SSE2", Proceedings of the SSBA Symposium on Image Analysis, 2005.
[5] J´er^^ome Landr´e and Fr´ed´eric Truchetet, "Optimizing signal and image processing applications using Intel libraries", Proceedings of SPIE, vol 6356, 2007.
[6] Intel Corporation, "Intel® 64 and IA-32 Architectures Optimization Reference Manual", May, 2007
[7] Asadollah Shahbahrami. Ben Juurlink, Stamatis Vassiliadis, "Performance Impact of Misaligned Accessesin SIMD Extensions", ProRISC 2006, pp. 334-342, 2006.
[8] Matrox, "Matrox Imaging Library 8.0 User Guide", June, 2005.
[9] Intel Corporation, "Intel Integrated Performance Primitives for Intel Architecture Reference Manual, Volume2 : Image and Video Processing", January, 2007.
[10] Intel Corporation, "Open Source Computer Vision Library Reference Manual", December, 2001.
[11] G. Conte, S. Tommesani, E Zanichelli, "The long and winding road to high-performance image

processing with MMX/SSE", Proceeding of CAMP, pp. 302-342, 2000.
[12] Asadollah Shahbahrami Ben Juurlink Stamatis Vassiliadis, "Efficient Vectorization of the FIR Filter", ProRisc 2005, pp. 432-437, 2005.
[13] 조상현, 박창준, 최홍문 등, "인텔 MMX 기술을 이용한 영상처리 루틴의 고속화", 전자기술연구지, 경북대 공대, vol. 22, pp.154-161, 2001.
[14] S. Rakshitn, A. Ghosh, B. Uma Shankar, "Fast mean filtering technique (FMFT)", Pattern Recognition, Vol. 40, No. 3, pp. 890-897, 2007.
[15] Timothy Furtak, Jose Nelson Amaral, Robert Niewiadomski, "Using SIMD Registers and Instructions to Enable Instruction-Level Parallelism in Sorting Algorithms", Proceeding of SPAA, June, 2007.
[16] Deepu Talla, Lizy Kurian John, Doug Burger, "Bottlenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements", IEEE Transactions on Computer, Vol. 52, No. 8, pp. 1015-1031, August, 2003.
[17] Joe H. Wolf III, "Programming Methods for the Pentium® III Processor's Streaming SIMD Extensions Using the VTune Performance Enhancement Environment", Intel Technology Journal, 1999.

저 자 소 개

박 은 수(학생회원)
2007년 인하대학교 정보통신 공학부 학사 졸업.
2007년~현재 인하대학교 정보공학과 석사 과정.
<주관심분야 : 컴퓨터 비전, 병렬 처리>



김 준 철(학생회원)
2008년 인하대학교 정보통신공학부 학사 졸업.
2008년~현재 인하대학교 정보공학과 석사 과정.
<주관심분야 : 컴퓨터 비전, 로봇 비전, 패턴 인식>

김 학 일(정회원)
1983년 서울대학교 제어계측 공학과 학사 졸업.
1985년 (미) 퍼듀대학교 전기 컴퓨터공학과 석사 졸업.
1990년 (미) 퍼듀대학교 전기 컴퓨터공학과 박사 졸업.
1990년 9월~현재 인하대학교 공과대학 교수
2001년 2월~현재 한국생체인식포럼 시험평가분과 위원장
2002년 1월~현재 한국정보보호학회 생체인증연구회 위원장
2003년 3월~현재 ISO/IEC JTC1/SC37 (생체인식) WG5(성능평가) Rapporteur Group
2005년 4월~현재 ITU-T SG17 Q.8 (Telebiometircs) Rapporteur
<주관심분야 : 생체인식, 생체인식 표준화, 정보보호>

최 학 남(학생회원)
2004년 연변대학교 응용수학학과 학사 졸업.
2007년 상명대학교 컴퓨터과학과 석사 졸업.
2007년~현재 인하대학교 정보공학과 박사 과정.
<주관심분야 : 컴퓨터 비전, 로봇 비전, 머신비전>

임 유 청(정회원)
1998년 한국 해양대 기계공학과 학사 졸업.
2000년 한국 해양대 기계공학과 가시화 정보공학 석사 졸업.
2003년 일본 동경대학교 기계공학과 가시화 정보공학 수료.
2004년~현재 (주)삼성전기 생산기술 센터 (화상응용)
<주관심분야 : 머신비전, 광학자동검사>