# SIMD 명령어를 이용한 영상의 평균화 필터 최적화

김준철, 최학남, 박은수, 김학일
정보통신공학부, 인하대학교

# Optimization of Image Averaging Filter Using SIMD Instructions

Junchul Kim, Xuenan Cui, Eunsoo Park, Hakil Kim
School of Information and Communication Engineering, Inha University

**Abstract** 본 연구에서는 영상의 잡음 제거에 우수한 평균화 필터 알고리즘을 SIMD(Single Instruction Multiple Data) 명령어를 이용하여 고속화 하였다. 먼저 순차 알고리즘의 속도 향상을 위한 최적화를 수행하고, SIMD에 적합 하도록 변환 하여 4 또는 16 개의 데이터를 동시에 연산이 가능하도록 하였다. 또한 나누기 연산을 줄이기 위하여 8비트의 데이터 타입과 함께 룩업 테이블(Lookup Table)을 이용하였다. 각각의 데이터 타입에 적합하게 알고리즘을 변환하여 비교하였고 실험을 통하여 기존의 순차 처리 방식에 비해 평균적으로 2.5배 이상의 속도 향상을 보았다.

## 1. 서 론

영상에 잡음이 있다고 할 때, 그 영상을 보고 알 수 있는 것은 잡음의 농도와 그 주변의 농도는 급격한 차이를 보인다는 점이다. 이러한 잡음의 성질을 이용하여 잡음 제거를 행하는 방법을 Smoothing이라고 부른다. 이를 위한 간단한 잡음 제거법이 평균화 필터링이다. 평균화 필터는 반복적인 루프 안에서 곱셈과 덧셈이 존재하여 병렬적인 연산들이 많다. 따라서 이러한 연산을 병렬로 동시에 수행함으로써 알고리즘의 수행 속도를 향상 시킬 수 있다[1].

Intel사는 MMX, SSE(streaming SIMD extensions), SSE2, SSE3, SSSE3(supplemental SSE3) 등과 같은 일련의 SIMD 기술을 발표하여 왔으며, 45nm 공정의 CPU 환경에서는 SSE4 명령어의 사용도 가능할 예정이다[2]. 본 연구에서는 잡음을 제거하기 위한 영상의 전처리 과정에서 기본이 되는 평균화 필터 알고리즘을 SIMD 구조의 다중 프로세서 환경에 맞게 병렬 고속화 하였다. 실험을 통하여 본래의 C++ 프로그램으로 구현된 경우와 다양한 SIMD 명령어를 이용하여 구현한 경우의 수행 속도를 비교하였다.

## 2. 본 론

### 2.1 평균화 필터 알고리즘(Averaging Filtering Algorithm)

임의의 점$(x,y)$에서 $m \times n$ 마스크의 응답이 $R$ 이라하면, 아래와 같이 표현 할 수 있다.

$$R = w_1 z_1 + w_2 z_2 + \ldots + w_{mn} z_{mn} = \sum_{i=1}^{mn} w_i z_i \quad (1)$$

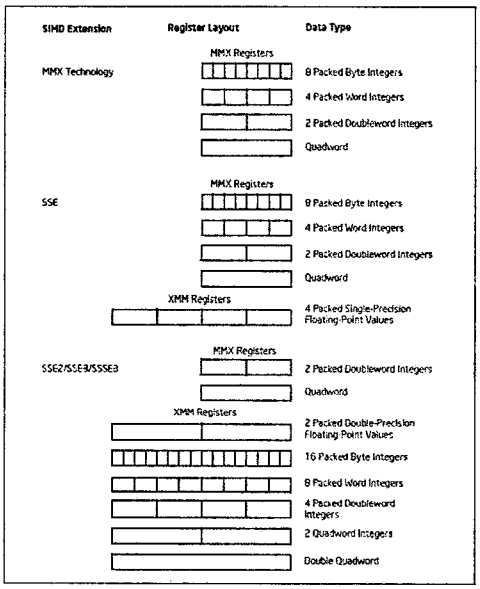
여기서 $w$들은 마스크의 계수이고 $z$들은 그 계수에 대응되는 영상의 명암도 값이고, $mn$은 마스크 계수의 전체 개수 이다. 식(1)에 3×3의 평균화 마스크를 대입하면 다음과 같은 응답을 얻을 수 있다.

$$R = \frac{1}{9}\sum_{i=1}^{9} z_i \quad (2)$$

### 2.2 MMX 및 SSEn기술

Intel PentiumII 프로세서에서 MMX 명령어를 지원하기 시작하여, Intel Pentium4 프로세서 및 듀오코어 환경에 이르러서는 MMX 명령어의 확장인 SSE, SSE2, SSE3, SSSE3 명령어를 지원한다. MMX명령어에서는 64비트의 레지스터(register)를 이용하여 8개의 8비트 정수 데이터를 동시에 처리 할 수 있고, SSE 에서는 128비트 레지스터를 이용하여 4개의 SIMD-FP(floating point) 연산을 할 수 있다. SSE2는 16개의 8비트 정수 데이터를 동시에 처리하는 SIMD 명령어들과 2개의 배정도(doble precision) 실수 데이터를 처리하는 명령어를 포함하고 있다. 90 nm 프로세서 기술 환경에서 포함된 SSE3 명령어에서는 13개의 명령어를 추가하여 SSE, SSE2, 및 x87-FP 기능을 강화 하였다. 또한 SIMD 정수 연산 기능을 강화하기 위한 32개의 명령어가 SSSE3 명령어에 포함되었다. 그림1에서 SIMD의 레지스터 형태 및 데이터 타입을 요약하여 보여주고 있다.

이러한 SIMD 명령어를 사용하기 위한 방법은 Inline Assembler, Automatic Vectorization, Intrinsics 함수 등이 있다. Inline Assembler는 SSE 명령어를 직접 이용하여 구현하므로 가장 좋은 속도 개선을 이룰 수 있지만, 프로세서 구조에 따른 유동성이 떨어지고 개발 시간이 오래 걸리는 단점이 있다. Automatic Vectorization은 Intel C/C++ 컴파일러를 통해 성능을 개선하고자 하는 루프를 자동적으로 SSE를 사용하여 구현하게 되는 가장 쉬운 방법이다. 하지만 이 방법은 데이터 의존성이 강한 루프의 경우는 적용하기 어렵다. Intrincs 함수는 Inline Assembler와 Auto Vectorization의 중간 방법으로써 Intel 사에서 SSE 명령어를 사용하기 쉽도록 내재하여 지원하고 있다. Intrincs 함수를 이용하면 SIMD컴파일러가 지원되는 Intel 프로세서 모든 구조에 적합하고, Assembler와 비슷한 성능을 가져 올 수 있다. 따라서 본 연구에서는 Intrincs 함수에 적합하도록 알고리즘을 변환하여 병렬 처리를 구현하였다[2][3].




**<그림 1> SIMD 명령어에 따른 레지스터 형태와 데이터 타입**

### 2.3 평균화 필터 구현

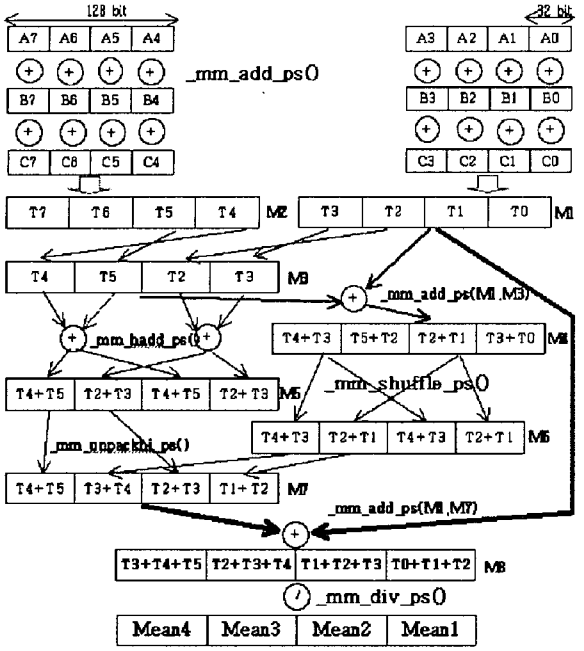
식 (2)에서 알 수 있듯이 평균화 필터를 적용하는 과정에는 나누기 연산이 필요하게 된다. 나누기 연산은 단정도(single precision) 실수 데이터나 배정도 실수 데이터 타입에서만 가능하다. 따라서 SIMD 명령어가 실수 타입에 적합하도록 알고리즘을 변환해야 한다. 이미지를 받아들이는 방법 및 처리 방법을 다음의 세 가지 형태로 변환하여 성능 개선 정도를 비교하였다.

#### 2.3.1 32비트의 실수 데이터를 이용한 평균화 필터

SSE Intrincs의 _mm_load_ps() 함수를 이용하여 32비트 4개의 실수 데이터를 받아들인다. 그림 2에서처럼 각 열에 대하여 덧셈 연산을 취하고 더해진 값들을 기본으로 하여 _mm_shuffle_ps(), _mm_add_ps(), _mm_unpackhi_ps() 등의 함수를 조합하여 4개의 마스크에 대한 평균값을 동시에 구할 수 있게 된다. 먼저 각 열의 합 M1과 M2를 _mm_shuffle_ps() 함수를 이용하여 M3과 같이 조합한다. 조합된 M3은 SSE3 명령어로 구성된 _mm_hadd_ps() 함수를 이용해 수평 덧셈을 하여 M5를 만들고, M1과 _mm_add_ps() 함수를 이용하여 각 열을 더해줘 M4를 만든다. M4는 다시 shuffle 연산을 통해 M6을 만들어 M5와 교차시켜 M7을 얻게 된다. 최종적으로 M7은 M1과 더한 후 나눗셈 연산을 통해 4개의 마스크의 평균값을 얻게 된다.

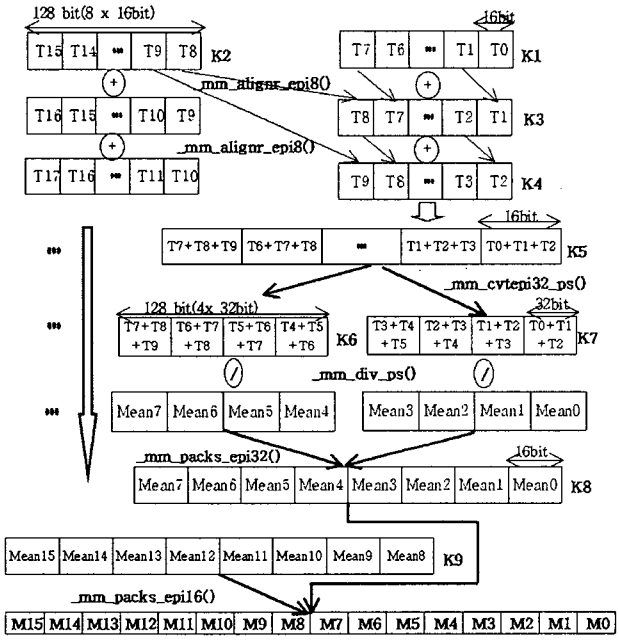
#### 2.3.2 8비트의 정수 데이터를 이용한 평균화 필터

SSE2와 SSE3를 이용하여 32비트의 실수 데이터를 가지고 평균화 필터를 구하는 과정과는 달리 이 과정에서 사용된 평균화 필터 구현은 SSSE3의 intrinsics 함수도 함께 이용하여 8비트의 정수를 받아 들였다. SSE2 이상에는 128비트의 레지스터를 이용하여 16개의 8비트 정수 데이터를 처리하는 SIMD 명령어가 포함되어 있다.




**<그림 2> 4개의 실수 데이타를 이용한 평균화 필터**

그러나, 마스크의 평균값을 구하는 중간의 덧셈 과정에서 값이 255보다 커져 8비트가 넘을 수가 있다. 또한 나누기 연산에서도 32비트가 필요하므로 비트 변환을 거쳐야만 16개의 화소를 동시에 처리를 할 수 있다. 처음 _mm_load_si128() 함수를 이용하여 8비트의 16개 화소를 받아들이고, unpack과 SSSE3의 가로 덧셈을 이용하여 8개의 16비트 값들로 나누어진다. 나누어진 값들은 그림 2에서 M1을 구한 것처럼 각 열을 더해서 그림 3의 K1과 K2를 구한다. 그 다음 K1을 오른쪽으로 이동하고 마지막 값은 K2에서 가져와 K3을 만들고 같은 방법으로 한 칸 더 이동하여 K4를 만든다. 이것은 하나의 레지스터에 8개의 마스크 결과가 들어 갈 수 있도록 하기 위한 방법이다. 이렇게 구해진 K1, K3, K4는 다시 각 열을 더해 K5를 구성한다. 이때 K5의 하나의 값은 하나의 마스크 즉, 9개의 화소를 모두 더한 값들이 된다.

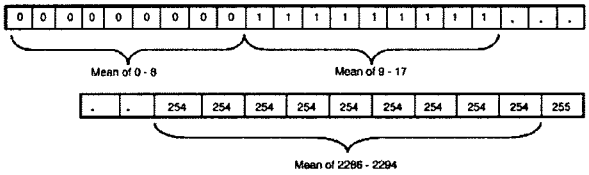



**<그림 3> 16개의 정수 데이타를 이용한 평균화 필터**

이 값들은 conversion 함수를 이용하여 4개의 32비트 값들로 만들어 나눗셈을 해주게 되고 역과정의 변환을 통해 8개의 16비트 값을 갖는 레지스터를 거쳐 16개의 8비트 값을 갖는 레지스터를 만들게 된다. 여기서 K9 구하는 과정은 앞에서 살펴본 방법과 같으나 K3과 K4를 K1에서 가져와 값을 채운 것처럼, 다음의 16 화소를 불러와 채우는 과정이 필요하다.

### 2.3.3 룩업 테이블(Lookup Table)을 이용한 평균화 필터[1]

2.3.2에서 살펴본 16개의 정수 데이터를 이용한 방법은 나누기 연산을 위하여 32비트 값으로 변환하는 과정이 필요하다. 하지만 룩업 테이블을 이용하게 되면 나누기 연산이 불필요하게 되어 그림 3에서 K5의 과정까지만 적용하여 반복하면 된다. 예를 들어 그림 4처럼 3x3 마스크의 경우 K5의 각 값들이 0과 8사이에 있으면 0이 되고 9와 17사이에 있으면 1이 되게 된다. 즉, K5를 룩업 테이블을 적용하여 8개의 마스크에 대한 평균값을 구하고 마찬가지로 다음 8개 마스크에 대한 평균값을 구한 후 하나의 레지스터에 16개의 값을 넣게 된다.




**<그림 4> 3x3마스크의 룩업 테이블**

## 3. 실험 결과

SIMD를 이용한 평균화 필터의 성능을 평가하기 위해 실험은 4048×4057 크기의 이미지를 가지고 Intel Core 2 Duo 2.4GHz 프로세서와 2 GHz DDR2 메모리 환경에서 이루어 졌다. 표 1은 일반적인 평균화 필터와 앞에서 언급한 세 가지 방법에 대하여 각각 수행 시간을 측정하여 비교하였다.

**<표 1> 각 방법에 대한 수행 시간 비교**

| 구현 방법 | 시간 (Clock cycle) | 속도 향상 |
|---|---|---|
| 표준 C++ | $22.5 * 10^{10}$ | 1.0 |
| 32 bit Float type | $9.6 * 10^{10}$ | 2.3 |
| 8bit Integer type | $6.9 * 10^{10}$ | 3.3 |
| Lookup Table | $13.2 * 10^{10}$ | 1.7 |

표에서 알 수 있듯이 기존의 순차 처리에 비해 평균적으로 2.5배의 향상을 얻을 수 있었고, 실수 데이터 연산이 필요한 알고리즘을 다양한 방법을 통해 성능 개선을 할 수 있었다. 하지만 4개의 실수 데이터를 SIMD를 이용하여 동시에 처리 할 경우 이론적으로는 4배의 속도 향상을 기대 하였으나 실제는 예상보다 낮은 향상을 가져 왔다. 이것은 4개의 실수 데이터를 이용하기 위한 명령어의 제약과 메모리와 레지스터 간 데이터 전달의 비효율성에 기인한 것으로 판단된다. 한편 16개의 정수 데이터를 이용하여 SIMD를 구현 할 경우 이론적으로는 16배의 속도 향상을 기대 하였으나 이보다 낮은 결과를 가져오는 것은 나누기 연산을 위한 데이터 비트 변환 과정과 실수 형을 통한 연산이 이루어졌기 때문이라 판단된다. 또한 SIMD 명령어와 룩업 테이블을 함께 이용한 방법에서도 성능 개선을 이루었지만, 배열로 이루어진 룩업테이블 접근 과정에서 속도 저하가 생겨 4개의 실수 데이터를 이용하여 구현한 방법보다 낮은 결과를 가져왔다.

## 4. 결 론

본 논문은 영상에서 잡음을 제거하는 전처리 과정에 효과적인 평균화 필터를 SIMD 명령어를 이용하여 고속화 하였다. 평균화 필터를 구현하는 정에서 필요하게 되는 나누기 연산의 수행 방법에 따라 4개의 실수 데이터 타입과 16개의 정수 데이터 타입으로 나누어 알고리즘을 구현하였다. 그리고 나누기 연산을 줄이기 위한 방법으로 16개의 정수 데이터 타입과 함께 룩업테이블도 사용하였다. Intel 프로세서의 호환성과 성능을 높이기 위해 Intrincs 함수를 이용한 알고리즘을 통해 평균적으로 2.5배 이상의 성능 개선을 볼 수 있었다. 실수 연산이 적고 8비트의 작은 정수형의 연산으로 이루어진 일반적인 영상처리에 SIMD 기술을 적용한다면 처리 속도를 크게 개선 할 수 있을 것이다.

## [참 고 문 헌]


[1] S. Rakshitn, A. Ghosh, B. Uma Shankar, "Fast mean filtering technique (FMFT)", Pattern Recognition, vol. 40, no. 3, pp. 890-897, 2007.

[2] *Intel64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, Intel, 2007.

[3] *Intel64 and IA-32 Architectures Software Developer's Manual Volum 2B: Instruction Set Reference N-Z*, Intel, 2007.

[4] *Intel 64 and IA-32 Architectures optimization Reference manual*, Intel, 2007.