제 18 회 한국정보처리학회 추계학술발표대회 논문집 제 9 권 제 2 호 (2002.11)

SIMD 를 이용한 영상의 고속 이진화에 관한 연구

김두식, 이상호, 김병근
한국전자통신연구원 정보화기술연구본부 자동구분처리연구팀
e-mail : {doosik, ileesh, iros}@etri.re.kr

A Study on High-speed Image Binarization Using SIMD

Doo-Sik Kim, Sang-Ho Lee, Byeong-Geun Kim
Automatic Sorting and Processing Research Team,
Electronics and Telecommunications Research Institute

요 약

영상 이진화란 명도 영상(gray-scaled image)을 이진 영상(bi-leveled image)으로 변환하는 것을 말한다. 영상 이진화는 문서 인식, 비디오 영상 분석 등과 같이 영상처리 분야에서 많이 사용되는 기본적인 영상 처리 과정에 해당한다. 본 논문은 Intel 사의 Pentium 계열 프로세서에서 지원하는 SIMD(Single-Instruction Multiple-Data) 기술을 이용하여 영상 이진화를 고속으로 수행하는 방법을 소개한다. 우편 영상에 대하여 실험한 결과, SSE2 명령어로 구현된 프로그램은 기존의 C 언어로 구현된 프로그램에 비하여 4 배 이상의 속도 향상을 보였다.

1. 서론

영상 이진화란 명도 영상(gray-scaled image)을 이진 영상(bi-leveled image)으로 변환하는 것을 말한다. 본 논문은 Intel 사의 Pentium 계열 프로세서에서 지원하는 SIMD (Single-Instruction Multiple-Data) 기술을 이용하여 영상 이진화를 고속으로 수행하는 방법을 소개한다. Intel 사는 MMX, SSE, SSE2 등과 같은 일련의 SIMD 기술을 발표하여 왔으며, 16 개의 데이터를 동시에 처리할 수 있는 SSE2 명령어를 사용할 경우에 기존 코드에 비하여 이론상 최대 16 배의 속도 향상을 기대할 수 있다.

2. 본론

2.1. 영상 이진화 개념

명도 영상의 각 화소는 0 부터 255 까지의 명도 값을 갖는다. 이에 반하여 이진 영상의 각 화소는 0 또는 255 의 값을 갖는다. 일반적으로, 0 의 명도 값을 갖는 화소는 흑 화소에, 255 의 명도 값을 갖는 화소는 백 화소를 의미한다. 이때, 명도 영상을 임의의 임계값(threshold value)을 기준으로 이진 영상으로 변환하는 것을 이진화라고 한다[1-4]. 앞에서 기술한 이진화를 수행하는 기능을 C 언어를 이용하여 구현하면 코드 1 과 같다.

$$B(x, y) = \begin{cases} 255 & \text{if } G(x, y) \geq t \\ 0 & \text{otherwise} \end{cases}$$

수식 1. 명도 영상의 이진화를 위한 식

```
<입력>
  unsigned char *image : 영상의 포인터
  int width : 영상의 폭
  int height : 영상의 높이
  unsigned char threshold : 이진화 임계 값
<코드>
  for (int j = 0; j < height; j++)
    for (int i = 0; i < width; i++, image++)
      *image = (*image >= threshold) ? 0xFF : 0;
```

코드 1. 이진화를 C 언어로 구현한 예

2.2. Intel 기술을 이용한 이진화의 고려 사항

Intel 사의 Pentium 프로세서에는 다양한 최적화 기법들을 도입하고 있다[5]. 이 중에서 프리패치(pre-patch) 기법은 순차적으로 실행되는 명령문을 미리 중앙 연산 장치(CPU)로 읽어서 실행 속도를 향상시키는 기법이다. 이 때 조건 분기가 포함된 명령문의 경우에는 이러한 최적화 기법을 효과적으로 적용할 수 없다. 앞에서 언급한 이진화 과정은 반복 문 내에 조건 문을 포함하는 대표적인 경우이며, 조건문을 제거함으로써 속도 향상을 꾀할 수가 있다.

또한, Intel 사의 Pentium 프로세서는 SIMD 기능을 포함하고 있다. SIMD 란 여러 개의 데이터 처리를 단일 명령어로 처리하는 기법이다. 즉 대량의 데이터를 한꺼번에 처리하는 경우에 SIMD 명령어를 사용하여 처리 속도를 향상시킬 수 있다. 영상 이진화 과정도 많은 수의 화소에 대하여 동일한 처리를 수행하므로 SIMD 명령어를 적용할 수 있는 예가 된다.

2.3. MMX, SSE/SSE2 기술 요약

Intel 사는 점차적으로 발전된 Pentium 계열의 프로세서들을 공개하면서 이와 함께 MMX 기술을 발전시켜왔다[6]. 초기의 Pentium 프로세서에 처음으로 MMX 기술을 도입하였으며, Pentium III 프로세서에는 *SSE(Streaming SIMD Extensions)* 기술, *Pentium 4* 프로세서에는 SSE2(Streaming SIMD Extensions 2) 기술로 확장하였다. 그 동안의 기술 변화를 간략히 소개하자면, MMX 에는 64 비트의 레지스터(register)를 이용하여 8 개의 8 비트 정수 데이터를 동시에 처리하는 SIMD 명령어들이 포함되어 있고, SSE 에는 128 비트의 레지스터를 이용하여 4 개의 단정도 실수 데이터를 동시에 처리하는 *SIMD* 명령어들이 포함되어 있으며, *SSE2* 에는 128 비트의 레지스터를 이용하여 16 개의 8 비트 정수 데이터를 동시에 처리하는 SIMD 명령어들과 2 개의 배정도 실수 데이터를 동시에 처리하는 SIMD 명령어들이 포함되어 있다.

2.4. SIMD 명령어 사용을 위한 inline 어셈블리 코드와 intrinsic 함수

C 언어 환경 하에서 C 소스 내에 어셈블리 코드를 직접 작성하는 것을 인라인 어셈블리 코드라고 한다. 인라인 어셈블리 코드는 속도가 비교적 빠른 프로그램을 작성할 수 있는 반면에 복잡한 어셈블리 명령어를 직접 사용해야 하므로 프로그램을 작성하는데 어려움이 있을 수 있다. 따라서, Intel 사에서는 보다 쉬운 코딩을 돕기 위해 대부분의 SIMD 명령어에 대응하는 내재 함수를 지원하고 있다.

```
_asm {                                  __m64 r1, r2, r3;
  pminub   xmm1, xmm7
  pcmpeqb xmm1, xmm7                    r3 = _mm_min_epu8(r1, r2);
}                                       r3 = _mm_cmpeq_epi8(r1, r2);
```

(a) 인라인 어셈블리 (b) Intrinsic 함수 사용 예

코드 2. 인라인 어셈블리 코드와 내재 함수의 예

2.5. SIMD 기술을 이용한 이진화 구현

SIMD 명령어를 이용하면 앞에서 언급한 이진화 연산을 다수의 화소들에 대하여 동시에 처리할 수 있다. 즉, MMX/SSE 명령어를 이용하면 8 개의 화소를, SSE2 명령어를 이용하면 16 개의 화소를 동시에 처리할 수 있다. 일례로, 그림 1 과 같이 단일 명령어를 이용하여 8 개 또는 16 개의 화소 값에 대하여 임계 값보다 크거나 같으면 0xFF 를, 그렇지 않으면 0 을 출력하는 연산을 수행할 수 있다.

$$B(x+i,y)=\begin{cases}255 & \text{if } G(x+i,y)\ge t\\ 0 & \text{otherwise}\end{cases} \quad \text{for } 0<i<L \quad \text{where } L=\begin{cases}8 \text{ in MMX}\\ 16 \text{ in SSE2}\end{cases}$$

수식 2. SIMD 를 이용하여 영상을 이진화 하는 식

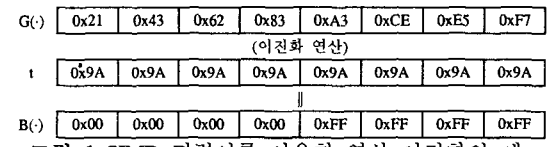

그림 1. SIMD 명령어를 이용한 영상 이진화의 예

2.6. MMX/SSE를 이용한 이진화

MMX 명령어 중에서 두 개의 64 비트 레지스터에 저장된 각 8 비트 정수 데이터에 대하여 크기 비교를 수행하는 명령어는 *PCMPGTB* 이다. 그러나 이 명령어는 8 비트 정수를 무부호(unsigned) 값이 아닌 유부호(signed) 값으로 간주하므로 0 부터 255 의 값을 화소 값으로 갖는 명도 영상의 이진화 문제에 직접 적용하기가 곤란하다. 즉, 명도 영상의 화소 값이 0x21 이고 임계 값이 0x9A 인 경우에 이진 영상의 화소 값은 0x00 이어야 하지만(그림 1), PCMPGTB 명령어를 사용하면 0x9A 를 음수로 간주하므로 이진 영상의 화소 값이 0xFF 가 되는 문제가 발생한다(그림 2).

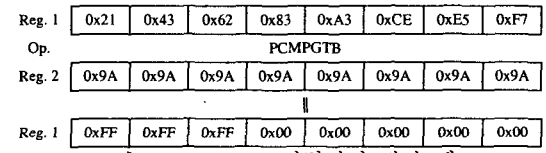

그림 2. PCMPGTB 명령어의 결과 예

이러한 문제를 해결하기 위하여 SSE 명령어에 해당하는 PMINUB 와 MMX 명령어에 해당하는 PCMPEQB 를 함께 사용할 수 있다. *PMINUB* 명령어는 두 피 연산자 레지스터에 저장된 대응되는 각 8 비트 무부호 정수 값 중에서 작은 값을 취하며, PCMPEQB 명령어는 두 피 연산자 레지스터에 저장된 대응되는 각 8 비트 무부호 정수 값을 비교하여 같으면 0 을 다르면 0xFF 를 출력한다. 그림 3 은 PMINUB 명령어와 PCMPEQB 명령어를 이용하여 이진화를 수행한 예를 보여주며, 영상의 명도 값과 이진 값의 부호와 상관없이 두 개의 명령어를 조합함으로써 올바른 이진화 결과를 출력할 수 있음을 보여주고 있다.

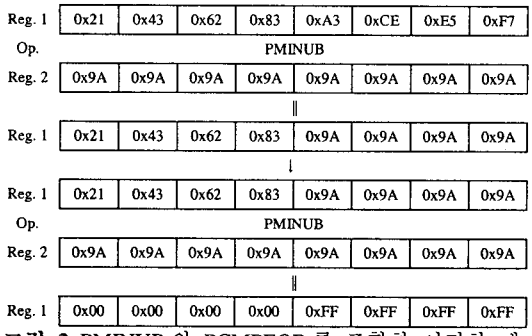

그림 3. PMINUB 와 PCMPEQB 를 조합한 이진화 예

즉, 수식 1 에서 사용한 크기 비교 연산을, 수식 3 에서와 같이 최소값 연산과 비교 연산의 조합으로 변환하여 동일한 결과를 얻을 수 있다.

$$B(x,y) = \text{equal}(\min(G(x,y), t), t)$$

$$\text{where } \min(a,b) = \begin{cases} a & \text{if } a < b \\ b & \text{otherwise} \end{cases} \text{ and } \text{equal}(a,b) = \begin{cases} 255 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$

수식 3. 크기 비교 연산자를 사용하지 않은 이진화 식

코드 3 은 MMX 와 SSE 에서 지원하는 내재 함수들을 이용하여 이진화를 수행하는 예를 보여준다.

```
<입력>
  unsigned char *img : 영상의 포인터 (8 바이트로 정렬)
  int width : 영상의 폭 (8 바이트로 정렬)
  int height : 영상의 높이
  unsigned char threshold : 이진화 임계 값

<단계>
1. 임계값을 레지스터에 저장
  __m64 reg2 = _mm_set1_pi8(threshold);

2. 영상을 레지스터에 저장
  __m64 reg1 = _mm_set_pi32(*(int*)img, *(int*)(img + 4));

3. 최소값 연산
  reg1 = _mm_min_pu8(reg1, reg2);

4. 비교 연산
  reg1 = _mm_cmpeq_pi8(reg1, reg2);

5. 결과 저장
  *(long*)img = reg1.m64_u32[1];
  img += 4;
  *(long*)img = reg1.m64_u32[0];
  img += 4;

6. img 포인터 증분

7. 처리 종료 조건 시까지 2~6 단계 반복

8. MMX 상태 초기화
  _mm_empty();
```

코드 3. MMX 와 SSE 의 Intrinsic 함수를 이용한 예

이때, MMX 와 SSE 에서 메모리로부터 데이터를 읽거나 메모리에 데이터를 기록하기 위해 사용하는 메모리 포인터는 8 바이트 단위로 정렬되어야 한다. 즉, 메모리 포인터의 값이 8 의 배수이어야 한다. 그렇지 않으면, 위 프로그램의 실행 시에 명령어 특권 오류(instruction privileged error)가 발생한다. MMX 와 SSE 에서 8 바이트 단위로 정렬되어 있지 않은 경우에 대한 메모리 접근 명령어 집합이 지원되기는 하지만, 이러한 명령어들을 사용하면 실행 속도가 크게 저하되므로, 고속 연산 효과를 기대하기 위해서는 메모리를 8 바이트 단위로 정렬하는 것이 바람직하다. 또한, MMX 와 FPU(Floating Point Unit)는 동일한 레지스터 집합을 공유하므로 MMX 를 이용한 SIMD 명령어를 수행한 후에 실수 연산을 하기 위해서는 _mm_empty() 함수를 호출하여 레지스터를 초기화하여야 한다. 그렇지 않으면, MMX 에 해당하는 명령어 수행 이후에 오는 실수 연산들이 잘못된 결과를 출력하게 된다.

한편, 코드 4 는 코드 3 를 인라인 어셈블리 코드로 구현한 예를 보여준다.

```
<입력>
  unsigned char *img : 영상의 포인터 (8 바이트로 정렬)
  int width : 영상의 폭 (8 바이트로 정렬)
  int height : 영상의 높이
  unsigned char threshold : 이진화 임계 값

<단계>
1. 임계값을 레지스터에 저장
  movd mm2, threshold
  punpcklbw  mm2, mm2
  punpcklwd  mm2, mm2
  punpckldq  mm2, mm2

2. 영상을 레지스터에 저장
  mov    esi, img
  movq   mm1, mmword ptr [esi]

3. 최소값 연산
  pminub  mm1, mm2

4. 비교 연산
  pcmpeqb  mm1, mm2

5. 결과 저장
  movq   mmword ptr [esi], mm1

6. img 포인터 증분

7. 처리 종료 조건 시까지 2~6 단계 반복

8. MMX 상태 초기화
  _mm_empty();
```

코드 4. MMX 와 SSE 를 인라인 코드로 구현한 예

2.7. SSE2를 이용한 이진화

SSE2 에는 128 비트의 레지스터를 이용하여 16 개의 8 비트 정수 데이터를 동시에 처리하는 SIMD 명령어들이 포함되어 있다. 그러나 SSE2 에도 MMX/SSE 의 경우에서처럼 두 무부호 정수를 직접 크기 비교하는 명령어가 없기 때문에 두 개의 명령어를 조합하여

무부호 정수를 크기 비교하는 것과 동일한 결과를 얻도록 하여야 한다. 즉, 피 연산자 레지스터에 저장된 각 16 개의 8 비트 무부호 정수 데이터의 각 쌍에서 작은 값을 취하는 PMINUB 명령어와 각 쌍의 값이 동일한 값인지를 비교하는 PCMPEQB 명령어를 사용하여 이진화를 수행한다. 이 과정은 MMX/SSE 에서와 유사하나, SSE2 는 FPU 와 별도의 레지스터를 사용하므로 MMX 의 경우에서와 달리 _mm_empty() 함수를 호출할 필요가 없다.

한편, 메모리로부터 데이터를 읽거나 메모리에 데이터를 기록하기 위해 사용하는 메모리 포인터는 MMX/SSE 와 마찬가지로 16 바이트 단위로 정렬되어 있어야 하며, 그렇지 않을 경우에는 영상의 좌우 측 영역을 처리하는 고전전인 방법을 추가해야 한다.

```
<입력>
  unsigned char *img : 영상의 포인터 (16 바이트로 정렬)
  int width : 영상의 폭 (16 바이트로 정렬)
  int height : 영상의 높이
  unsigned char threshold : 이진화 임계 값

<단계>
1. 임계값을 레지스터에 저장
  movd xmm2, threshold
  punpcklbw xmm2, xmm2
  punpcklwd xmm2, xmm2
  pshufd xmm2, xmm2, 0

2. 영상을 레지스터에 저장
  mov  esi, img
  movdqu  xmm1, xmmword ptr [esi]

3. 최소값 연산
  pminub  xmm1, xmm2

4. 비교 연산
  pcmpeqb  xmm1, xmm2

5. 결과 저장
  movdqu  xmmword ptr [esi], xmm1

6. img 포인터 증분

7. 처리 종료 조건 시까지 2~6 단계 반복
```

코드 5. SSE2 명령어를 이용한 이진화 코드의 예

3. 실험 및 결과 분석

표 1 은 4 개의 실험 우편 영상에 대하여 앞에서 언급한 네 가지 방법으로 이진화를 수행한 시간을 측정한 결과를 보여준다.

표 1. 각 방법에 대한 영상의 이진화 시간 (ms)

구현 / 영상	표준 C	MMX/SSE (intrinsic)	MMX/SSE (inline)	SSE2
평균 시간	3.53	3.81	1.03	0.78

예상한 바와 같이, 표준 C 로 구현한 방법은 SSE2 로 구현한 방법에 비하여 이진화 처리를 하는데 4 배 이상의 시간을 소요했다. 한편, 기대했던 바와 달리 MMX/SSE 를 intrinsic 함수로 구현한 방법이 가장 많은 처리 시간을 보였는데, 이는 영상 정보를 읽고 쓰는 과정에서 메모리와 레지스터 간 데이터 전달의 비효율성에 기인한 것으로 판단된다. 즉, 빈번한 메모리 접근이 요구되는 상황에서는 메모리와 레지스터 간의 데이터 전송 방식에 대해 효율성을 고려해야 한다.

결국, SSE2 가 기존 방식에 비해 16 개의 데이터를 동시에 처리할 수 있는 장점이 있는 반면, 이진화 문제에 있어서는 단일 명령어가 아닌 두 개의 명령어로 나누어 처리해야 하고, 메모리 접근 속도의 한계로 인하여, 이론적인 16 배속 향상이 아닌 4 배속 향상에 머문 것으로 판단된다.

4. 결론

Intel 사에서 개발한 Pentium 프로세서 계열의 CPU 에서 지원하는 SIMD 명령어를 이용하여 영상 처리 과정에서 비교적 많은 처리 시간을 필요로 하는 이진화 과정의 처리 속도를 4 배 가량 향상시킬 수가 있었다. 본 논문에서 언급한 MMX, SSE/SSE2 기술은 Intel 사의 프로세서에서만 지원하는 기능이므로 이를 사용할 경우에 호환성 문제가 발생할 것으로 우려할 수 있으나, Intel 사의 제품이 시장의 많은 부분을 차지한다는 것을 고려한다면, 호환성 문제가 크게 장애가 될 염려는 없을 것으로 보인다. 그리고, 현재 소수 시장을 형성하고 있는 AMD 계열의 프로세서에서도 이와 유사한 형태의 SIMD 기술을 지원하므로, 향후 필요하다면 AMD 계열의 프로세서에 적용할 수 있도록 확장 개발하는 데에 어려움이 없을 것으로 예상된다. 특히, 최근의 프로세서 개발 기술의 발전 동향을 살펴볼 때, SIMD 기술의 지원은 보편화될 것으로 예상되므로, 이러한 SIMD 기술을 적극 도입하여 영상 처리 분야에서 시스템의 처리 속도를 향상시키는 것이 유용할 것으로 판단된다.

참고문헌

[1] J. N. Kapur, P. K. Sahoo, and A. K. C. Wong, "A new method for gray-level picture thresholding using the entropy of the histogram", Computer Vision, Graphics and Image Processing, vol. 29, pp. 273~285, 1985.

[2] J. Kittler and J. Illingworth, "Minimum error thresholding", Pattern Recognition, vol. 19, no. 1, pp. 41~47, 1986.

[3] T. Kurita, N. Otsu, and N. Abdelmalek, "Maximum likelihood thresholding based on population mixture models", Pattern Recognition, vol. 25, no. 10, pp. 1231~1240, 1992.

[4] A. Jain, S. Bhattacharjee, and Y. Chen, "On texture in document images", *Proc. of Computer Vision and Pattern* Recognition, pp. 677-680, 1992.

[5] "Intel® Pentium® 4 and Intel® Xeon™ processor optimization reference manual (order number 248966-007)", http://www.intel.com/design/pentium4/manuals/248966.htm.

[6] "IA-32 Intel® architecture software developer's manual, volume 1: basic architecture (order number 245470)", http://www.intel.com/design/pentium4/manuals/245470.htm