Packing Buffer for Efficient Irregular Data access in SIMD Processors

장호석, 임지은, 성원용

Seoul National University

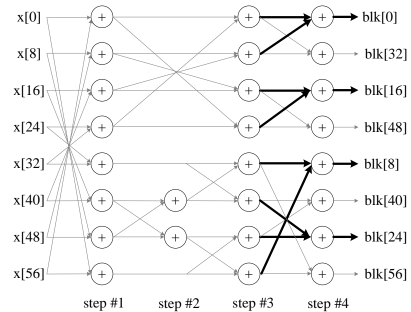
초록

The performance of an SIMD (Single Instruction Multiple Data) processor is bounded by the memory bottleneck; most of which is due to the overhead for preparing aligned vector data. In this paper, we have added a hardware unit to an SIMD processor to reduce the alignment overhead. The proposed packing buffer contains a small size multi-port memory block for which multiple addresses are generated by using a vector index register. Since the packing buffer has a small size, it requires neither complex hardware nor increased CPU cycle time. DSP benchmarks are used to measure the performance efficiency.

1. 서론

SIMD(Single Instruction Multiple Data) 프로세서는 하나의 명령어를 이용하여 여러 개의 데이터를 처리하는 구조를 가지기 때문에 비교적 간단한 하드웨어를 필요로 하며, 멀티미디어 어플리케이션과 같이 데이터 수준의 병렬도가 큰 프로그램 처리에 매우 효율적이다. 최근에는 분할된 데이터 경로 (data-path)를 갖는 SIMD 프로세서 구조가 많이 사용되고 있는데 Intel Pentium4, Texas Instruments TMS320C64x, ARM Cortex 등이 그 예이다[1][2]. 분할된 데이터 경로를 이용하면 한 개의 명령어를 이용하여 여러 개의 정렬된 데이터를 처리할 수 있다. 하지만 데이터들이 항상 정렬되거나 일정한 규칙을 가지고 저장되는 것은 아니다. 데이터들이 정렬 되지 않았을 경우에는 이 들을 정렬하기 위한 추가의 명령어가 필요하기 때문에 SIMD 프로세서의 효율이 급격히 떨어진다. SIMD 프로세서에서 복잡한 데이터 접근은 크게 두 가지로 분류될 수 있는데, 그것은 비정렬 접근 (nonaligned access)과 불규칙적 데이터 접근 (irregular data access)이다. 이 중 불규칙적인 데이터 접근이 더 복잡한데 그 이유는 일반적으로 데이터 주소가 작은 범위 안에 있음에도 불구하고 순서대로 정렬되어 있지는 않기 때문이다. 불규칙 데이터 접근의 경우에는 데이터를 정렬시키기 위해서는 추가적인 메모리 접근 연산뿐만 아니라 pack 과 shuffle 과 같은 특수한 instruction 이 필요하다. 불규칙한 데이터 접근의 한가지 예로 그림 1 에 Chen-Wang DCT 알고리즘의 신호흐름도를 나타내었다[3]. 여기에는 상당한 데이터 병렬성이 존재하고 있으나, SIMD 코드 생성은 불규칙한 데이터읽기 때문에 매우 힘들어진다. 그리고 이 경우에 SIMD 코드 생성, 소위 벡터화(vectorization),을 자동화하기가 매우 어려워진다. SIMD 코드를 생성하기 위한 컴파일러로는 Intel compiler, ARM 의 RealView compiler, GNU 의 gcc compiler 가 있다[4][5]. 하지만 이러한 컴파일러를 이용한 소프트웨어 기반 접근은 모든 오버헤드를 쉽게 제거하지 못한다.

코드의 질을 향상시키기 위한 다른 접근 방법으로는 데이터를 자동으로 정렬시키기 위한 특수한 하드웨어를 사용하는 것이다. 본 논문에서는 SIMD 프로세서의 불규칙적인 데이터 정렬을 쉽게 처리하기 위한 패킹(packing) 버퍼를 제안한다. 본 패킹 버퍼는 작은 크기의 다중포트(multi-port) 메모리를 가지고 있으며 vector index 레지스터로부터 다수의 주소를 공급받는다. 본 패킹 버퍼의 크기는 작기 때문에 복잡한 하드웨어를 필요로 하거나 CPU 의 동작 속도를 감소시키지 않는다. 또한 우리는 작은 크기의 packing 버퍼를 이용하기 위한 컴파일러 개발환경을 개발하였으며 이것은 여러 벤치마크를 이용한 성능 측정을 지원한다.



(a) Chen-Wang DCT 의 수직 방향 signal flow diagram

```
for (i = 0 i < 8; i ++)
{
        ...
        blk[0] = (x[0] + x[8]) >> 8;
        blk[8] = (x[32] + x[56]) >> 8;
        blk[16] = (x[16] + x[24]) >> 8;
        blk[24] = (x[48] + x[40]) >> 8;
        ...
}
```

(b) 8x8 Chen Wang DCT 의 C source code

그림 1. 불규칙적인 데이터 접근의 예

2. Packing 버퍼의 하드웨어 구조

본 패킹버퍼의 구조가 그림 2 에 나타나있으며, index 레지스터를 가진 작은 다중 포트 메모리를 포함하고 있다. 배열에 불규칙한 데이터 접근이 있을 경우 index 레지스터에 배열의 index(첫번째 주소)를 저장하고 base 레지스터의 입력 operand 에 대한 SIMD 데이터 읽기 (load)를 수행한다. 각 데이터의 주소는 base 주소와 index 를 더해서 얻을 수 있고 이 주소는 다중 포트 메모리로 전송되어 원하는 벡터가 얻어진다. 이 다중 포트 메모리는 작은 용량으로 좋은 성능을 얻기 위해 일반적인 메모리에 통합되어야 한다. 이러한 하드웨어 지원이 packing 버퍼인데 packing 버퍼의 크기는 클 필요가 없다. MPEG2 비디오 인코딩에서는 2D-DCT 와 IDCT 의 처리를 위해서 128 byte 로 충분하다.

Packing 버퍼는 두 가지 방식으로 동작할 수 있는데, 하나는 on-chip 메모리 블럭으로 동작할 수도 있고 또 다른 방법으로는 캐쉬처럼 동작할 수도 있다. 첫 번째 경우에는 동적 메모리 관리 체계가 필요하다. 동적 메모리 관리 소프트웨어는 copy 와 copy-back 기능을 가지고 있으며 본 연구에서 개발된 SIMD 컴파일러 개발환경에서 그 기능을 지원한다. 이러한 방식은 추가적인 소프트웨어 오버헤드가 필요하지만 packing 버퍼에 분리된 주소 공간을 제공하고 L1 캐시와의 데이터 동기(coherency) 문제를 고려하지 않아도 되므로 런타임 시의 동작이 간단하고 효율적인 장

이 논문은 지식경제부 출연금으로 ETRI 와 시스템반도체산업진흥센터에서 수행한 IT SoC 핵심설계인력양성사업, 그리고 삼성반도체 주식회사의 지원을 받아 수행한 연구결과 입니다.

점이 있다.

Packing 버퍼가 캐쉬처럼 동작할 경우에는 tag 메모리 블럭과 태그비교(tag-comparing) 로직이 추가로 필요하게 된다. 그래서 이 경우에는 packing 캐쉬로 불리기도 한다. Packing 캐쉬를 포함하는 캐쉬 메모리 시스템은 그림 3 에 나타나있다. 불규칙적인 데이터 접근이 필요할 경우 base 주소와 index 가 packing 캐쉬로 보내지고 그에 해당하는 상위 비트를 tag 와 비교하게 된다. Tag 를 비교하는 로직을 간단하게 하기 위해 packing 캐쉬의 캐쉬 라인은 index 전체를 커버할 수 있도록 충분히 커야 한다. 만약 각 index 가 i 비트라면 index 된 전체 데이터를 위해 캐쉬 라인은 2^{i+1} byte 를 가져야 한다. 예를 들어 index offset 이 6bit unsigned integer 로 나타내어질 때 packing buffer 는 최소한 16bit 데이터를 위한 128 byte 에 해당하는 64 개의 데이터를 가져야 한다. Packing 캐쉬 구조는 캐쉬 메모리와 coherency 문제가 발생할 수 있다. 즉 coherency 를 위해 일반적인 메모리 접근 시에도 packing 캐쉬를 접근해야 되며 원하는 데이터가 packing 캐쉬에 이미 존재하고 있을 때만 data 가 제공된다. Packing 캐쉬는 자동으로 수행되는 copy 나 copy-back 기능이 필요하지 않고, 따라서 packing 캐쉬가 동적 프로그램 실행 환경에서 더 안정적일 수는 있지만 메모리에 접근할 때 항상 추가적인 tag 비교가 필요하다.

3. 실험 결과

성능 평가를 위한 SIMD 프로세서는 제안된 하드웨어를 지원하도록 설계되었다. 프로세서는 ARMv4 구조를 기반으로 하고 128bit SIMD ALU 와 16 개의 128-bit 벡터 레지스터를 포함하고 있다.

벤치마크로 DSP 커널이 사용되었다. DSP 커널은 16-tap FIR 필터, 12th order IIR 필터, 16x16 2D-DCT 와 1024-point FFT 가 포함되어 있으며 이러한 벤치마크들은 복잡한 데이터 접근 패턴을 포함하고 있다. IIR filter 는 stride access 를 포함하고 있고 Chen-Wang 알고리즘을 사용한 2D-DCT 벤치마크는 불규칙적 접근 패턴을 포함한다.

128 Byte 라인의 packing 버퍼를 사용할 경우 hit 일 때는 1 cycle, 1 miss 일 때는 L1 캐쉬 전체를 fetch 할 때는 108 cycles 가 걸린다. 이 때 외부 메모리는 166MHz, 32-bit wide SDRAM 모듈을 가정했으며 프로세서는 500MHz 로 동작한다. SDRAM 의 burst 동작 때문에 miss cycle 은 라인 길이에 따라 선형적으로 증가하지는 않는다.

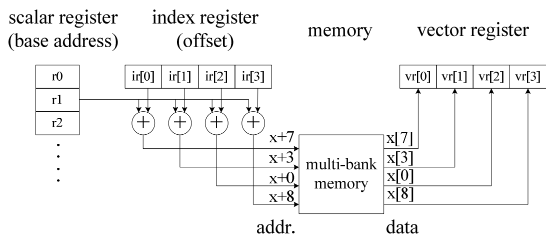


그림 2. index 레지스터 기반의 4-port 메모리

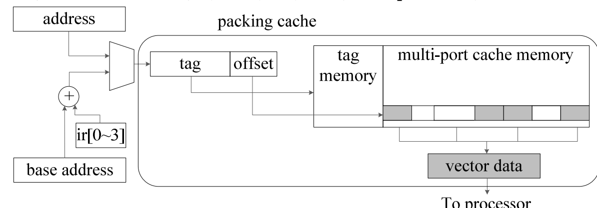


그림 3. 불규칙 데이터 접근을 위한 캐쉬 메모리 시스템

표 1 은 DSP 커널에서 측정한 성능 결과를 보여준다. 첫번째 결과는 통상의 단일 포트 메모리를 이용한 결과이며, 두번째는 통상의 단일 포트 메모리에 비정렬 데이터 악세스를 돕기 위한 split buffer 를 추가한 경우이며, 최종적으로 패킹버퍼가 추가된 경우의 결과이다. FIR 필터의 경우에는 불규칙 데이터 접근이 없으므로 패킹버퍼에 의한 성능향상이 없다. IIR 벤치마크는 stride access 를 가지고 있고 이것은 packing 버퍼에 의해 잘 지원되기 때문에 성능이 22% 정도 향상되었다. 2D-DCT 벤치마크에서는 대부분의 벡터 데이터 접근 연산이 불규칙적이기 때문에 packing 버퍼를 사용했을 경우 성능 향상이 크다. FFT 벤치마크의 bit-reverse shuffling 부분은 vectorize 하기가 매우 어렵고 캐쉬 miss rate 가 27.4%로 매우 높다. 그래서 packing 버퍼를 사용할 경우 섞기(shuffling) 연산이 index 레지스터 기반의 메모리 접근 명령어에 의해 벡터화 된다. 또한 packing 버퍼가 shuffling 부분의 캐쉬 miss rate 를 19.8%까지 낮춰준다. 일반적으로 멀티미디어 데이터 처리는 temporal locality 가 낮고 spatial locality 가 크기 때문에 라인 길이가 긴 packing 버퍼를 사용할 경우 더 좋은 성능을 보인다.

		conventianal SIMD	with split buffer	with split and packing buffer
FIR	cycles	70,674	39,703	39,703
	speed -up	1	1.78	1.78
IIR	cycles	126,142	43,235	33,562
	speed-up	1	2.92	3.76
2D-DCT	cycles	73,268	63,442	38,696
	speed-up	1	1.15	1.89
FFT	cycles	243,664	243,664	205,752
	speed-up	1	1	1.18

표 1. DSP 커널의 성능

4. 결론

SIMD 컴파일러를 이용한 벡터화 연산은 메모리에 의해서 그 효율성이 제한되는데 그 이유는 정렬된 벡터 데이터를 만드는 오버헤드가 크기 때문이다. 본 논문에서는 packing 버퍼/캐쉬를 이용한 효율적인 메모리 시스템을 설계하여 불규칙적인 데이터 접근의 오버헤드를 줄이도록 하였다. 또한 제안된 구조와 설계된 SIMD 컴파일러를 이용하여 DSP 커널의 성능향상이 테스트 되었다.

참고문헌

[1] *TMS320C64x Technical Overview*. Texas Instruments, 2000

[2] G. Hinton, et al., The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 5(1):1-13, 2001.

[3] Z. Wang. Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-32(4):803-816, 1984.

[4] A.J.C. Bik, et al., Automatic Intra-Register Vectorization for the Intel Architecture. International Journal of Parallel Programming, 20(2):65-98.

[5] D. Nuzman and A. Zaks. Autovectorization in GCC-Two Years Later. In *Proceedings of the 2006 GCC Developers Summit*, pages 145-58, 2006.