회로 최적화를 위한 효율적인 희소 행렬 간 곱셈 연산에 관한 연구

임 은진, 김 경훈°
국민대학교 컴퓨터학부

Efficient Sparse Matrix-Matrix Multiplication for circuit optimization

Eun-Jin Im, Kyung-Hoon Kim
School of Computer Science, Kookmin Univ.

요 약

행렬 연산은 계산 과학을 사용하는 공학, 물리, 화학, 생명 과학, 경제학 등에서 다양하게 사용되고 있으며 이 행렬은 크기가 크고 대부분의 원소가 0 값을 갖는 희소 행렬일 경우가 많다. 본 논문에서는 희소 행렬의 연산 중, 회로 설계 시 최적화 과정에 사용되는 연산에서 문제가 되는 희소 행렬 A 와 블록 대각 행렬 H에 대하여 AHA^T 의 연산을 효율적으로 행하는 방법들을 검토하고 메모리 접근 횟수를 모델링하여 수행 속도와 메모리 사용량 면에서 비교한다.

1. 서론

행렬에 대한 연산은 계산 과학을 사용하는 공학, 물리, 화학, 생명 과학, 경제학 등에서 다양하게 사용되고 있다. 이 때 사용되는 행렬의 크기는 풀고자 하는 문제의 변수의 개수에 따라 결정되므로 실제적인 문제에서 크기가 매우 커서 모든 원소를 2차원 배열의 형태로 저장하는 밀집 행렬로 표현하였을 때 대부분의 컴퓨터에서 주 메모리의 크기를 넘어선다. 그러나 이를 잘 살펴보면 행렬의 원소는 연관이 있는 변수의 행과 열이 만나는 곳에서만 0이 아닌 값을 가지고 대부분의 경우에 0의 값을 가진다. 이러한 행렬은 0이 아닌 값만을 그 위치 정보와 함께 저장하는 희소 행렬의 형태로 저장된다. 희소 행렬의 형태로 저장을 하면 메모리의 사용량을 줄일 수 있을 뿐 아니라 행렬 연산을 하는데 있어서 0값에 대한 연산을 생략하기 때문에 계산량을 줄일 수 있다. 그러나 희소 행렬의 연산 속도 (0이 아닌 원소에 대한 연산만을 연산 개수에 포함시킬 때) 는 프로세서의 peak 성능보다 훨씬 낮고 밀집 행렬의 같은 연산보다도 낮은데 그 이유는 첫째로 모든 0이 아닌 원소에 대한 연산에서 그 원소의 행렬 내 위치를 표현하는 index 를 희소 행렬을 표현하는 자료 구조에서 읽어와야 하기 때문에 추가로 메모리를 접근해야 하는 오버헤드 때문이고, 둘째로는 연산 대상이 되는 벡터 혹은 행렬의 원소를 접근할 때 희소 행렬의 0이 아닌 원소의 행렬 내 위치 분포에 따라 비순차적인 메모리 접근이 이루어지기 때문이다.

따라서 희소 행렬 연산의 성능 개선은 계산 과학에서 중요한 영역이다. 그러나 밀집 행렬의 경우 BLAS [1]와 같이 많이 사용되는 연산들에 대한 표준화가 이루어져 이 표준화된 함수들의 성능 최적화에 대한 연구와 구현이 잘 이루어져 있는 반면에 희소 행렬 연산 경우에는 그와 같은 표준화가 이루어져 있지 않은 실정이다. 그 이유는 두 가지로 볼 수 있다. 첫째, 희소 행렬의 연산 속도는 행렬의 대칭, Hermitian과 같은 수학적 특성 뿐만 아니라 행렬 내에서 nonzero값들의 분포 형태와 밀접한 관계가 있는데 이러한 nonzero분포 형태를 분류, 판별하는 기준을 정하기가 어렵기 때문이다. 둘째, 희소 행렬의 복합 연산은 단순 연산 함수를 두 번 사용하는 것보다 전체 연산을 한 번에 수행하는 것이 성능 면에서 훨씬 효과적일 수 있는데 이러한 연산들까지 표준에 포함시키려면 그 범위가 너무 넓기 때문이다. 따라서 현재까지는 연산 속도가 문제시 되는 개별 행렬에 대한 개별 연산에 대한 성능 튜닝을 하는 것이 최상의 방법이다.

본 논문에서 우리는 회로 설계 시 최적화 과정에 사용되는 primal-dual 문제 [2]의 해를 구할 때 문제가 되는 희소 행렬 A 와 블록 대각 행렬 H에 대하여 AHA^t의 연산을 효율적으로 행하는 방법들을 검토하고 메모리 접근 횟수를 모델링하여 수행 속도와 메모리 사용량 면에서 비교한다.

2. 행렬의 특성과 연산

2.1 행렬의 특성과 연산

본 논문에서 다루는 AHA^T 연산은 회로 설계 소프트웨어에서 사용되는 다음과 같은 대칭 시스템의 해를 구하는 과정에서 사용된다.

$$\begin{bmatrix} 0 & B \\ B^t & AHA^t \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

여기서 A,B,H 행렬은 모두 희소 행렬로 이와 같은 시스템의 해를 구하는 것이 100-120번 정도 행해지는데 각 반복 시마다 AHA^t 의 곱을 계산하는 데 많은 시간이 걸린다. 각 반복 수행에서 A나 H의 구조, 즉 nonzero값의 위치들은 고정되고 값만 변화하며, H 행렬은 블록 대각 행렬로 블록의 크기는 1x1부터 39x39 까지 있으며 각 블록 H_i는 항등행렬 I와 행벡터 r_i에 대하여 $I+r_i r_i^t$의 형태를 가진다. 이는

$AHA^t = A(I+r_i r_i^t)A^t = AA^t + (Ar_i)(Ar_i)^t$

의 관계에 의하여 후자의 연산으로 대치될 수 있음을 보여준다.

2.2 희소행렬의 표현

희소 행렬을 저장하는 자료 구조는 행렬의 특성, 응용 분야의 특성에 따라 다양한데[1][3], 본 논문에서 다루는 행렬 A는 compressed sparse row (CSR) 혹은 compressed sparse column (CSC) 형태 중 선택하는 것이 적합하다. H 행렬의 경우, 마찬가지로 CSR 혹은 CSC 형식으로 저장하는 방법과 각 블록 $H_i=I+r_i r_i^t$의 성질을 이용하여 각 블록의 크기를 저장하는 1차원 배열과 다시 각 블록의 r_i 값들을 순차적으로 저장하는 1차원 배열로 저장하는 방법도 가능하다.

3. AHA^t 의 두 가지 연산 방법

3.1 two-phase scheme

다음의 알고리즘을 두 번 반복하여 적용한다. 즉 $P1=HA^t$ 를 계산하고 같은 알고리즘으로 $P=AP1$를 계산한다.

```
T(1:nrow(C))
mark(1:nrow(C))
indices(1:nrow(C))
T(1:nrow(C)) = 0 ;
for jj = 1:ncol(B)
count = 0 ;
   for B(ii,jj) != 0
      for A(kk,ii) != 0
         if mark(kk) == -1
```

```
            count = count + 1 ;
            mark(kk) = count ;
            indices(count) = kk ;
        end
        ll = mark(kk) ;
        T(ll) := T(ll) + alpha * A(ll,ii) * B(ii,jj)
    end
end
for kk = 1:count
    ii = indices(kk) ;
    C(ii,jj) = C(ii,jj) + alpha * T(kk) ;
    T(kk) = 0 ;
    mark(ii) = -1 ;
end
end
```

[그림 1] two-phase scheme의 유사코드

3.2 structure-aware scheme

H 행렬의 구조를 중심으로 $S=AHA^t$의 계산을 위하여 각 H_i 블록에 대하여

$$S = \sum_i A_i H_i A_i{}^t = \sum_i (A_i A_i{}^t + (A_i r_i)(A_i r_i)^t)$$

를 계산한다. 이 때 연산의 합은 두 개의 희소 행렬의 덧셈인데 nonzero값의 분포가 다른 두 개의 희소 행렬의 합을 계산하는 것은 데이터 구조 전체를 다시 구성해야 하므로 연산의 속도가 느리다. 이러한 문제점을 극복하기 위하여 summation을 전체 행렬에 대해 계산하는 대신 결과 product 행렬의 각 행에 대하여 다음 식과 같이 연산한다.

$$S = \sum_{k=1}^{n} \sum_{i\,for A_{ki}\neq 0} (A_i H_i A_i{}^t)_{k*}$$

$$= \sum_{k=1}^{n} \sum_{i\,for A_{ki}\neq 0} (A_i A_i{}^t + (A_i r_i)(A_i r_i)^t)_{k*}$$

여기서 A_{k*}는 A 행렬의 k번째 행을 나타낸다. 위와 같은 방법은 희소 행렬의 덧셈은 아직 효율적인 연산 방법이 개발되어 있지 않지만 희소 벡터의 덧셈은 효율적으로 구현할 수 있음에 착안하여 제안된 것이다. 바깥쪽 summation의 각 term은 희소 벡터의 덧셈으로 구현될 수 있기 때문이다.

3.3 두 방법의 비교

제안된 두 방법을 구현하였을 때의 수행 시간과 메모리 사용량을 비교해 보았다. 사용된 platform은 800 MHz Itanium I 으로 L1/L2/L3 캐쉬 크기는 각각 16K/96K/2MB 이다. Itanium I의 floating point unit은 4개이므로 peak performance는 3.2 Gflop/s 이다. 컴파일러는 Intel C compiler v5.0.1 을 사용하였다. 컴파일 옵션은 □03를 사용하였다. Two-phase scheme과 structure-aware scheme의 수행 시간과 메모리 사용량을 비교하면 표 1과 같다.

연산 방법	연산 시간	메모리 사용량
Two-phase scheme	9.1 초	105 MB
Structure-aware scheme	12.9 초	52 MB

[표 1] 두 가지 연산 방법의 비교

사용된 행렬 A, H 와 계산 결과 행렬 $P= AHA^T$ 의 크기와 nonzero element의 개수는 표 2와 같다.

행렬	크기	Nonzero 개수	Density
A	41392x244501	1,632,571	0.0161%
H	244501x244501	963,359	0.0016%
P	41392x41392	1,312,573	0.0077%

[표 2] 행렬들의 크기

현재 구현된 상태에서 two-phase scheme이 1.4배 정도 더 빠르나 기억용량은 2배 정도 더 많이 사용된다. 기억 장소 사용량은 논의의 편의성을 위하여 가장 많은 데이터를 차지하는 nonzero 요소를 저장하기 위한 공간만을 계산하면 two-phase scheme의 경우 (2*nz(A)+nz(H)+nz(P1)+nz(P)) * 8 만큼, structure-aware scheme 의 경우 (nz(A)+nz(H)*nz(P))*8 만큼이 필요하다. 이 때 nz(A)는 행렬 A의 nonzero의 개수이고 행렬 P1=HA^t 이다.

4. 관련 연구

희소 행렬의 연산의 비효율성은 간접적 데이터 구조와 비순차적 메모리 접근의 두 가지 이유 때문인데

이에 대한 연구는 [4][5][6]에서 희소 행렬과 밀집 벡터의 경우에 대하여 register blocking과 cache blocking의 방법이 제안되어 높은 성능 개선 효과를 보였고 [7]에서는 이와 같은 blocking의 효과를 높이기 위하여 행렬의 행과 열의 순서를 바꾸는 방법이 제안되었다. 희소 행렬간의 곱셈에 관한 연구는 [8][9]에서와 같이 일반적인 경우에 대한 library가 개발되어 있고 Java Sparse Benchmark[10] 에도 포함되어 있지만 본 연구에서는 block diagonal 행렬 H와 A행렬 간의 AHA^t 형태의 곱셈을 수행하는 특수한 구조의 연산을 다루고 있다. 희소 행렬 간의 곱셈은 응용 프로그램에 따라 행렬의 특수한 성질을 잘 이용하면 연산 속도를 높일 수 있는데 여러 분야의 응용에서 공통적으로 많이 사용되는 특성과 구조를 찾아 library화하는 것도 의미있는 연구가 될 것이다.

5. 결론 및 향후 연구

본 논문에서는 희소 행렬의 연산 중, 회로 설계 시 최적화 과정에 사용되는 연산에서 문제가 되는 희소 행렬 A 와 블록 대각 행렬 H에 대하여 AHA^T 의 연산을 효율적으로 행하는 방법들을 검토하고 메모리 접근 횟수를 모델링하여 수행 속도와 메모리 사용량 면에서 비교한다. Two-phase scheme과 structure-aware scheme의 두 가지가 제안되고 비교되었는데 two-phase scheme이 연산 속도는1.4배 정도 더 빠르나 기억용량은 2배 정도 더 많이 사용된다.

현재 구현된 코드는 더욱 개선의 여지가 있는데 크게 두 가지 방향의 개선이 가능하다. 첫째는 product 행렬 P가 대칭 행렬임을 이용하여 위나 혹은 아래 삼각 부분만 계산하면 메모리 사용량 뿐 아니라 연산량도 줄일 수 있다. 둘째는 이 연산이 nonzero값이 위치가 변하지 않고 반복됨을 이용하여 첫번째 곱셈에서 symbolic computation 을 통하여 product 행렬 P의 데이터 구조를 계산 하고 이후의 반복되는 곱셈에서 이 구조를 반복 사용하는 방법이다.

[참고문헌]

[1] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. Li, Z. Marmy, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, and J. W. von Gudenberg. Document for the Basic Linear Algebra Subprograms (BLAS) standard : BLAS Technical Forum, 2001. Chapter 3: www.netlib.org/blast.

[2] M. X. Goemans and D. P. Williamson, *The primal-dual method for approximation algorithms and its application to network design problems*, In Approximation Algorithms for NP-hard Problems, (D. S. Hochbaum), PWS Publishing Co., Boston, MA, 1995

[3] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997

[4] E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, May 2000

[5] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science*, volume 2073 of LNCS, pages 127-136, San Francisco, CA, May 2001. Springer

[6] 김 경훈, 김 병수, 임 은진. 희소 행렬 연산의 성능 최적화에 관한 연구. 2003년 정보과학회 춘계학술 발표 대회

[7] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of Supercomputing*, 1999.

[8] R.E. Bank and C.C. Douglas. SMMP: Sparse Matrix Multiplication Package. *Advances in Computational Mathematics*, **1** (1993), pp. 127-137.

[9] Preston Briggs. Sparse matrix multiplication ACM SIGPLAN Notices, Volume 31 , Issue 11 (November 1996) pp. 33 – 37

[10] JASPA (JAva SPArse benchmark) http://www.dl.ac.uk/TCSC/Staff/Hu_Y_F/JASPA