韓國 컴퓨터情報學會 論文誌
第11卷 第3號, 2006. 7.

2006-11-3-1-17

삼중 행렬 곱셈의 효율적 연산

임 은 진*

An Efficient Computation of Matrix Triple Products

Eun-Jin Im*

요 약

본 논문에서는 회로 설계 소프트웨어에서 사용되는 primal-dual 최적화 문제의 해를 구하기 위해 필요한 삼중 행렬 곱셈 연산 ($P = AHA^t$)의 성능 개선에 관하여 연구하였다. 이를 위하여 삼중 행렬 곱셈 연산의 속도를 개선하기 위하여 기존의 2단계 연산 방법을 대신하여 1단계 연산 방법을 제안하고 성능을 분석하였다. 제안된 방법은 희소 행렬 H의 블록 대각 구조의 특성을 이용하여 부동 소숫점 연산량을 감소시킴으로써 성능 개선을 이루었으며 더불어 메모리 사용량도 기존 방법에 비하여 50% 이하로 감소하였다. 그 결과 Intel Itanium II 플랫폼에서 기존 2단계 연산 방법과 비교하여 속도 면에서 주어진 실험 데이터 집합에 대하여 평균 2.04 의 speedup 을 얻었다. 또한 본 논문에서는 플랫폼의 메모리 지연량과 예측된 캐쉬 미스율을 이용한 성능 모델링을 통하여 이와 같은 성능 개선 수치의 가능 범위를 보이고 실측된 성능 개선을 평가하였다. 이와 같은 연구는 희소 행렬의 성능 개선 연구를 기본 연산이 아닌 복합 연산에 적용하는 연구로써 큰 의미가 있다.

Abstract

In this paper, we introduce an improved algorithm for computing matrix triple product that commonly arises in primal-dual optimization method. In computing $P = AHA^t$, we devise a single pass algorithm that exploits the block diagonal structure of the matrix H. This one-phase scheme requires fewer floating point operations and roughly half the memory of the generic two-phase algorithm, where the product is computed in two steps, computing first $Q=HA^t$ and then $P=AQ$. The one-phase scheme achieved speed-up of 2.04 on Intel Itanium II platform over the two-phase scheme. Based on memory latency and modeled cache miss rates, the performance improvement was evaluated through performance modeling. Our research has impact on performance tuning study of complex sparse matrix operations, while most of the previous work focused on performance tuning of basic operations.

• 제1저자 : 임은진
• 접수일 : 2006.06.20, 심사일 : 2006.07.05, 심사완료일 : 2006.07.17
* 국민대학교 컴퓨터학부 조교수

▸ Keyword : 삼중행렬곱셈(Matrix Triple Products), 성능 개선(Performance Improvement), 성능 모델링(Performance Modeling), 슈어-컴플멘트 행렬(Schur-Complement matrices), 희소 행렬 곱셈(Sparse Matrix Multiplication).

Ⅰ. 서 론

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 컴퓨터 시스템에서 주 메모리의 크기를 넘어선다. 그러나 실제로 이들 행렬의 대부분의 원소는 0의 값을 가지는 경우가 대부분이어서 0이 아닌 원소의 개수가 전체 행렬의 원소 개수의 10% 이하인 경우가 많다.[1] 따라서 이러한 행렬은 0이 아닌 값만을 그 위치 정보와 함께 저장하는 희소 행렬의 형태로 저장할 수 있다. 희소 행렬의 형태로 저장하는 경우 메모리의 사용량을 줄일 수 있을 뿐 아니라 행렬 연산을 하는데 있어서 0값에 대한 연산을 생략하기 때문에 계산량도 줄일 수 있는 이점이 있다. 이러한 희소 행렬의 표현 방법은 다양하게 개발되어 있어 대표적으로는 compressed sparse row, compressed sparse column, coordinate format 등이 있으며 이 외에도 행렬 내부의 0 아닌 값의 분포 형태에 따라 희소 행렬을 표현하기에 유리한 skyline format 등이 있다. [2]

이러한 희소 행렬의 연산 속도는 0이 아닌 원소에 대한 연산 개수를 연산 시간으로 나누어 계산하는데 이는 실제 프로세서의 peak 성능에 비교하여 훨씬 못 미치는 성능을 보인다. 또한 밀집 행렬에 대한 연산 속도와 비교해서도 속도가 낮은데 그 주된 이유는 (1) 희소 행렬 원소를 접근하기 위해 필요한 간접적 자료 구조의 접근에서 오는 비효율성과 (2) 밀집 행렬 연산에서 볼 수 있는 행렬 원소 접근의 근접성이 희소 행렬에서는 상대적으로 미약하다는데서 그 원인을 찾을 수 있다. [3]

이러한 행렬 연산의 성능 개선은 계산 과학의 여러 문제에서 대규모 연산에 소요되는 계산 시간의 단축에 필수적인 영역이다. 그러나 행렬이 밀집 행렬로 표현되는 경우에는 빈번히 사용되는 연산들에 대해서는 BLAS (Basic Linear Algebra Subprograms standards) [4] 표준화가 이루어져 있어서 널리 사용되고 있기 때문에 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 비영값들의 분포 형태와 밀접한 관계가 있는데 이러한 비영값들의 분포 형태를 분류, 판별하는 기준을 정하기가 어렵기 때문이다. 둘째, 희소 행렬의 복합 연산은 단순 연산 함수를 여러 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 대하여 그 특성을 분석하여 성능 튜닝을 하는 것이 최상의 방법이다.

본 논문에서 우리는 회로 설계 시 최적화 과정에 사용되는 primal-dual optimization 문제 [5]의 해를 구하는 과정에서 필요한 희소 행렬 간의 삼중 곱셈, 즉 희소 행렬 A와 블록 대각 행렬 H에 대하여 $P=AHA^t$의 연산의 성능 개선 방법에 대하여 연구한다. 기존의 연산 방법은 두 번의 곱셈 연산 ($Q=HA^t$, $P=AQ$) 을 통하여 행렬곱을 계산하는데 대하여, 제안되는 방법은 H의 블록 구조와 A와 A^t 의 대칭성을 이용하여 한 번에 행렬곱을 계산하는 1단계 방법이다. 본 논문은 1단계 방법을 구현하고 이를 기존의 2단계 방법과 비교한다. 또한 실제 platform에서 연산을 수행하여 연산 속도를 측정하고, 한 편으로는 이론적으로 얻을 수 있는 성능 개선을 메모리 지연량과 캐쉬 미스율을 근거로 모델링하여 실제 얻은 성능 개선을 평가한다. 이와 같은 성능 개선의 평가 방법은 앞으로 다른 연산의 성능 개선치를 평가하는 기준 프레임워크로 사용될 수 있다.

본 논문의 구성은 다음과 같다. 2절에서 해결하고자 하는 시스템의 특성과 이 특성에 맞는 희소 행렬의 표현 방법을 소개한다. 3절에서는 먼저 기존의 2단계 연산 방법을 효율적으로 구현하는 알고리즘을 소개하고, 다음으로는 어떻게 삼중 곱셈을 1단계로 연산하는지를 수학적으로 고찰한 후 1단계 연산 방법의 구현 알고리즘을 소개한다. 4절에서는 문제 규모를 달리하는 5개 데이터 집합에 대하여 1단계 연산 방법과 2단계 연산의 연산 성능을 비교하고 성능 모델

링에 의하여 얻을 수 있는 성능 개선 정도를 예측하여 예측치에 대한 실제 성능 개선치를 평가한다.

II. 행렬과 연산의 특성

2.1 행렬의 삼중 곱셈 연산

본 논문에서 다루는 $P = AHA^t$ 연산은 회로 설계 소프트웨어에서 사용되는 다음과 같은 대칭 시스템의 해를 구하는 과정에서 사용된다.

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

Primal-dual optimization 방법 [5]은 회로 설계, 최적 control, 포트폴리오 최적화 등의 방법에서 자주 쓰이는 방법이다. 위에 보여지는 대칭 시스템은 primal-dual optimization 문제를 연립 방정식으로 표현하고 Schur complement를 이용하여 그 해를 구할 때 사용된다. 여기서 A, B, H 행렬들은 모두 전체 원소중 비영값(nonzero element) 의 비율이 0.1 % 이하인 희소 행렬들이다. 이 시스템의 해를 구하기 위해서 직접 연산 방법(direct method)를 사용하려면 연산 과정의 행렬 분해(factorization) 에서 발생하는 fill-in 값들을 저장하기 위한 메모리 요구량의 증가를 실제적으로 해결하는 데 문제가 있다. 따라서 반복적 방법 (iterative method)을 사용하여 해를 구하게 되는데 일반적으로 이와 같은 시스템의 해를 구할 때 $P=AHA^t$ 의 연산이 100-120번 정도 행해지므로 현재 이 삼중 곱셈 연산에 소요되는 연산 시간이 전체 연산 시간의 70% 이상을 차지한다.

이 때, 각 반복 수행에서 행렬 A와 H의 구조, 즉 비영값 (nonzero element) 의 행렬 내 위치들은 고정된 채로 그 값만 변화하며, H 행렬은 블록 대각 행렬로 블록의 크기는 1x1부터 39x39 까지의 정사각형 블록들로 구성되어 있다. 그림 1에서 행렬 A와 H의 비영값의 분포를 그림으로 볼 수 있는데 이 중 전체 사각형이 행렬이고 색있는 부분은 비영값 (nonzero element) 의 존재를, 흰 부분은 해당 위치의 원소값이 0 임을 나타낸다.

이 중 H 행렬의 대각 블록을 부분 확대하여 보여주고 있는데 각각의 블록을 왼쪽 위부터 H_i 라 칭하고 그 크기를 n_i x n_i 로 표현할 때, 각 블록 H_i는 다시 크기 n_i x n_i인 대각 행렬 D_i와 크기 n_i x 1인 행벡터 r_i 에 대하여 $H_i = D_i + r_i \cdot r_i^t$ 의 형태를 가진다. 이는

$$A_i H_i A_i = A_i (D_i + r_i r_i^t) A_i^t = A_i D_i A_i^t + (A_i r_i)(A_i r_i)^t$$

의 관계에 의하여 후자의 연산으로 대치될 수 있음을 보여준다. 여기서 A_i 는 H_i 가 전체 H행렬의 c 번째 열에서 시작할 때 A행렬의 c번째 열부터 n_i개의 열로 이루어진 A의 부분행렬로써, $A_i = A_{*,c:c+n_i-1}$ 로 표현된다.

2.2 희소 행렬의 표현 방법

희소 행렬을 저장하는 자료 구조는 행렬의 특성, 응용 분야의 특성에 따라 다양하며[2][3], 자료 구조의 선택은 일차적으로는 행렬의 저장에 필요한 기억 장치의 사용량에 영향을 미치고, 이차적으로는 행렬에 대한 연산의 수행 속도와 밀접한 관련을 가진다. 이들 희소 행렬의 자료 구조는 크게 블록 구조를 갖는 자료 구조와 그렇지 않은 자료 구조로 나누어 볼 수 있다. 행렬의 블록 구조란 그림 1의 H행렬에서 보는 바와 같이 행렬 전체를 여러 개의 작은 사각형의 블록 구조로 나누어 볼 때 비영값(nonzero element)들이 일부의 블록 내부에 밀집되어 존재하는 행렬 구조를 지칭한다.

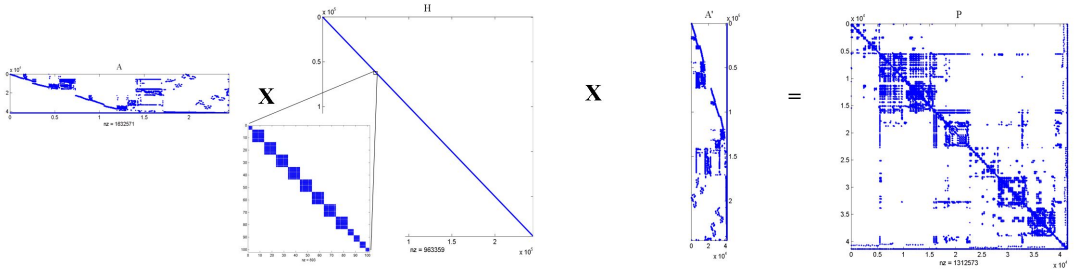

그림 1. $P=AHA^t$ 연산에 사용된 희소 행렬 A와 H, 그리고 P 의 구조
Fig 1. Nonzero Structure of matrices A,H, and P in computing $P=AHA^t$

본 논문에서 다루는 행렬 A는 행렬의 비영값들이 작은 사각 블록 내에 모여 있는 규칙성을 갖는 블록 구조를 가지고 있지 않으므로 이 중 후자에 속하는 compressed sparse row (CSR) 혹은 compressed sparse column (CSC) 형태 중 선택하는 것이 적합하다. CSC 형태에서 희소행렬은 column-major 순으로 저장된 비영값들의 1차원 배열 (그림 2의 value) 과 해당 원소의 행 (row) 위치를 저장하는 배열 (그림 2의 row_idx), 그리고 각 열의 시작 위치를 나타내는 간접 인덱스 배열 (그림 2의 col_start) 로 표현된다.

블록 대각 행렬인 H 행렬의 경우는, 대각 블록 H_i 의 원소들을 저장하는 대신, $H_i = D_i + r_i r_i^t$ 로 표현되는 성질을 이용하여 D_i 와 r_i 를 저장하는 자료 구조를 제안하여 사용한다. 이 자료 구조에서 행렬 H는 각각 대각 행렬 D_i 의 대각 원소들과, r_i의 원소들을 저장하기 위한 2개의 1차원 배열 (그림 3의 diag_v, rv_v) 과 다시 이 배열들에 대하여 블록의 시작위치를 나타내는 간접 인덱스 배열 (그림 3의 blk_start) 로 표현된다. 이와 같은 자료 구조를 본 논문에서 Block Diagonal Vector 형태 (BDV format) 로 명명하였다.

다음의 그림 2와 3에서 각각 CSC 와 BDV 방법에 의하여 다음의 블록 다각 행렬 H를 표현한 예를 보인다.

$$H = \begin{bmatrix} h_{11} & 0 & 0 & 0 \\ 0 & h_{22} & h_{23} & 0 \\ 0 & h_{32} & h_{33} & 0 \\ 0 & 0 & 0 & h_{44} \end{bmatrix}$$

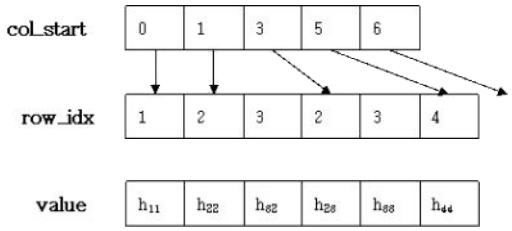

그림 2. CSC 형태의 희소 행렬 표현
Fig 2. Representation of matrix H in Compressed Sparse Column format

행렬 H는 다음과 같은 대각 블록 H_1,H_2,H_3 로 구성되어 있다. 블록 크기가 1x1인 경우에는 벡터 r은 영벡터이다.

$$H_1 = [h_{11}] = [d_{11}],$$
$$H_2 = \begin{bmatrix} h_{22} & h_{23} \\ h_{32} & h_{33} \end{bmatrix} = \begin{bmatrix} d_{21} & 0 \\ 0 & d_{22} \end{bmatrix} + \begin{bmatrix} r_{21} \\ r_{22} \end{bmatrix} \begin{bmatrix} r_{21} & r_{22} \end{bmatrix},$$
$$H_3 = [h_{33}] = [d_{33}]$$

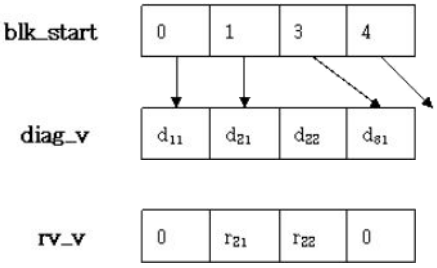

그림 3. BDV 형태의 희소 행렬 표현
Fig 3. Representation of matrix H in Block Diagonal Vector format

III. 삼중 곱셈의 두 가지 구현 방법 : 1단계 연산과 2단계 연산

3.1 기존의 방법: 2단계 연산 방법

기존의 연산 방법은 $P = AHA^t$ 를 계산하기 위하여 먼저 $Q = HA^t$ 를 계산하고 같은 알고리즘으로 $P = AQ$ 를 계산하는 2단계 연산 방법이다. 두 개의 희소 행렬 A,B의 곱 C를 계산하는 효율적인 알고리즘은 행렬 C의 각 열을 순서대로 계산하는 방법이 이미 사용되고 있다. 이와 같은 알고리즘은 고성능과 사용자 편리성 때문에 널리 사용되는 수학 연산 소프트웨어 MATLAB에서 희소 행렬 곱셈을 수행하는 알고리즘으로 구현되어 있다. [6] 연산 방법은 다음과 같다.

그림 4에서 보여지는 것처럼 행렬 C의 i번째 열 C*i 를 계산하기 위하여는 두 번째 행렬 B의 i번째 열 B*i 의 원소들만 필요하다. 따라서 행렬 A의 각 열 A*j 에 대하여 순서대로 비영값들을 읽으면 이 값들은 B*i 의 j번째 원소, 즉 B_{ji} 에 곱해져서 C*i의 j번째 원소에 누적된다. 이와 같은 연산 방법이 효율적인 이유는 전체 곱셈을 위하여 행렬 B의 원소들은 오직 한 번씩만 메모리에서 읽혀져서 재사용되며,

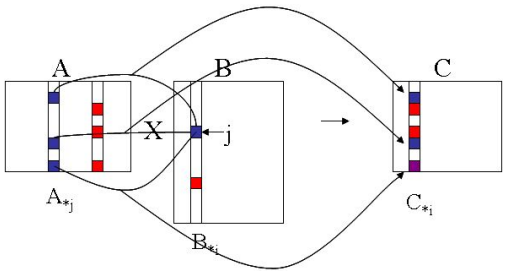

그림 4. C = AB 연산에서 C의 i번째 열의 계산
Fig 4. Formation of the i-th column of matrix C in computing C = AB

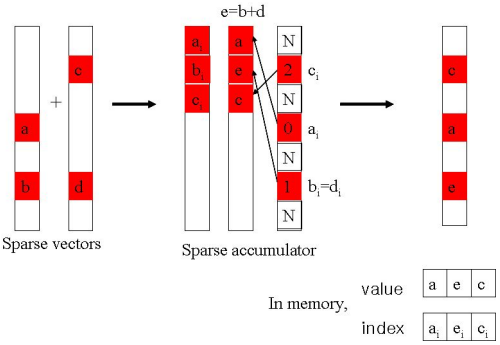

그림 5. sparse accumulator를 이용한 희소 벡터 덧셈
Fig 5. Summation of sparse vectors using a sparse accumulator

A_{*j} 는 비영값 B_{ji} 에 대하여 한 번씩 읽혀지므로 행렬 A의 원소 A_{ij} 는 0이 아닌 Bj*의 갯수만큼 읽혀져서 메모리를 경제적으로 접근하기 때문이다.

3.1.1 Sparse Accumulator를 이용한 희소 벡터의 덧셈 연산

그림 4는 효율적인 희소 행렬 곱셈 연산에서 i번째 열의 연산을 표현한다. 이 때 A_{*j} 와 A_{*k} 가 각각 B_{ji}와 B_{ki} 에 곱해서 얻어지는 희소 벡터들을 합산하는 데에는 sparse accumulator 에 의한 연산이 사용된다.

그림 5에서 보는 것처럼 sparse accumulator 는 각각 index 와 value를 저장하는 2개의 배열들과 참조 배열로 구현된다. 참조배열은 벡터의 각 원소들의 실제 위치에 대하여 index, value 배열 내에서의 위치를 저장하고 있는 배열이다. 그림에서 해당 위치의 원소 값이 0인 경우는 N(null) 로 표현되고 있다. 그림 왼쪽의 두 벡터가 더해지는 경우에, 첫 번째 벡터의 a, b 원소들이 순서대로 index, value 배열에 저장될 때 이 원소들의 저장 위치는 참조배열의 a_i, b_i 위치에 기록된다. 두 번째 벡터의 c, d원소가 저장될 때 d의 경우는 $d_i = b_i$ 이므로 새로운 위치에 저장되는 대신 참조배열이 가리키는 value 배열의 1번 위치의 값을 e = b + d 로 갱신한다. 이와 같은 연산 결과로 오른 쪽에 보이는 벡터는 왼쪽의 두 벡터의 합을 갖게 된다. 그림 5에서 왼쪽과 오른쪽에 보이는 벡터들은 실제로는 희소 벡터로 표현되는 벡터들이다.

본 절에서는 삼중 곱셈을 위한 2단계 연산 방법을 설명하였다. 삼중 곱셈을 위한 2단계 연산 방법은 product(A, product(H, A^t)) 로 표현될 수 있다. 이와 같은 구현은 H행렬의 블록 대각 특성을 고려하지 않고 두 개의 행렬의 원소들을 열에 따라 순서적으로 접근하므로 A와 H 행렬 모두를 Compressed Sparse Column (CSC) 형식으로 저장하는 것이 적합하다.

3.2 개선된 방법 : 1단계 연산 방법

개선된 방법에서는 블록 대각 행렬 H의 구조를 이용하여 삼중 행렬 곱셈을 one-pass 로 연산한다. 즉 H 행렬의 대각 블록을 왼쪽 위부터 H_i 라 칭하고 그 크기를 n_i x n_i 로 표현할 때, 각 블록 H_i는 다시 크기 n_i x n_i인 대각행렬 D_i와 크기 n_i x 1인 행벡터 r_i 에 대하여 $H_i = D_i + r_i \cdot r_i^t$ 의 형태를 가지는 사실을 이용하면 삼중 행렬 곱 $P=AHA^t$ 는 H 행렬의 블록 구조를 중심으로 다음 식에 의하여 계산될 수 있다.

$$P = \sum_{i}^{\#of\ blocks\ in\ H} P_i = \sum_i A_i H_i A_i^t$$

$$P = \sum_i A_i H_i A_i^t = \sum_i (A_i D_i A_i^t + (A_i r_i)(A_i r_i)^t)$$

이와 같은 연산을 구현할 때 문제점은 P 행렬을 구하기 위하여 H행렬의 블록의 개수만큼의 희소 행렬들의 합을 구하는 것이다.

그 이유는 그림 6에서 보는 바와 같이 다른 색으로 구분되는 $P_1=A_1H_1A_1^t$ 와 $P_2=A_2H_2A_2^t$ 의 비영값들의 분포가 다르기 때문인데, 이와 같이 비영값들의 분포가 다른 두 개의 희소 행렬의 합을 계산하기 위해서는 데이터 구조 전체를 매번 다시 구성해야 하므로 연산의 수행 시간이 증가한다.

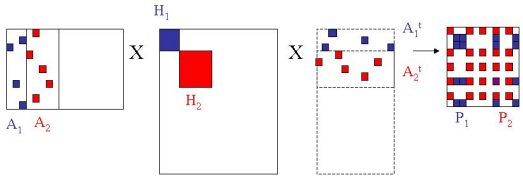

그림 6. $A_1H_1A_1^t + A_2H_2A_2^t$ 의 계산
Fig 6. Computing $A_1H_1A_1^t + A_2H_2A_2^t$

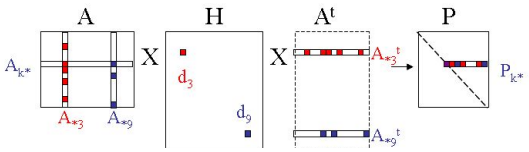

그림 7. 행-중심 행렬 연산의 경우, $P=AHA^t$ 의 k번째 행의 연산
Fig 7. Computing k-th row of $P=AHA^t$ in a row-wise computation

이러한 문제점을 극복하기 위하여 행렬합을 전체 행렬에 대해 계산하는 대신 결과 product 행렬의 각 행에 대하여 다음 식과 같이 연산한다.

$$P_{k\bullet} = \sum_{j}^{\#\ of\ col.s\ in\ A} (A_{\bullet j} d_j A_{\bullet j}^{\ t})_{k\bullet} + \sum_{i}^{\#\ of\ non-unit\ blocks\ of\ H} (B_{\bullet i} B_{\bullet i}^{\ t})_{k\bullet}$$

$$= \sum_{j} a_{kj} d_j A_{\bullet j}^{\ t} + \sum_{i} b_{ki} B_{\bullet i}^{\ t}$$

$$= \sum_{j:a_{kj}\neq 0} a_{kj} d_j A_{\bullet j}^{\ t} + \sum_{i:b_{ki}\neq 0} b_{ki} B_{\bullet i}^{\ t}$$

여기서 X_{k*}는 X 행렬의 k번째 행을, X_{*k}는 X 행렬의 k 번째 열을 나타내며, A_i 는 H행렬의 i번째 블록 H_i 가 전체 H행렬의 c 번째 열에서 시작할 때 A행렬의 c번째 열부터 n_i 개의 열로 이루어진 A의 부분행렬로써, $A_i = A_{*,c:c+n_i-1}$ 로 표현된다. 그리고 $B_{*i} = A_i r_i$ 를 나타낸다.

위와 같은 방법은 희소 행렬의 덧셈은 아직 효율적인 연산 방법이 개발되어 있지 않지만 희소 벡터의 덧셈은 앞 절에서 설명된 sparse accumulator 를 이용하여 효율적으로 구현할 수 있음에 착안하여 제안된 것이다. 바깥쪽 summation의 각 term은 희소 벡터의 덧셈으로 구현될 수 있기 때문이다.

이와 같은 구현에서 A_i 는 H_i 가 전체 H행렬의 c_i 번째 열에서 시작할 때 A행렬의 c_i번째 열부터 n_i개의 열로 이루어지므로 A행렬은 열(column) 중심의 Compressed Sparse Column 형식으로 저장하는 것이 적합하며 H행렬은 Block Diagonal Vector 형식으로 저장하는 것이 적합하다.

3.2.1 1단계 연산 방법에서 대칭성 활용

개선된 방법에서는 블록 대각 행렬 H의 구조를 이용할 뿐만 아니라, H가 대칭 행렬인 점을 이용, 행렬곱 P 역시 대칭 행렬인 성질에 따라, P*k의 계산에서 a_{kj} d_j $A_{*j}^{\ t}$ 와 b_{ki} $B_{*i}^{\ t}$ 를 연산하는 대신 a_{kj} d_j $A_{k:mj}^{\ t}$ 와 b_{ki} $B_{k:mj}^{\ t}$ 를 연산하여 연산량을 감소하였다. 이 때 $X_{k:mj}$ 는 X 행렬의 j번째 열의 k부터 m번째 행을 지칭한다.

이와 같은 대칭성의 활용은 1단계 연산 방법에서만 활용 가능한데 이는 2단계 연산의 경우 중간곱행렬 $Q=HA^t$ 는 대칭성을 갖지 않기 때문이다.

또한 이와 같은 구현에서 희소 행렬 A 의 비영값들에 대한 접근 속도를 개선하기 위하여 A 의 각 열에서 다음 비영값의 위치를 기억하는 보조 인덱스 배열을 사용하였다.

Ⅳ. 실험과 성능 분석

앞 절에서 설명된 삼중 행렬 곱셈 연산의 두 가지 구현의 실제 성능을 비교하기 위하여, 실제의 회로 설계 과정에서 생성되는 A,H 행렬들을 이용하여 실험하였다. 표 1에 보여지는 5개의 A,H 행렬 집합들은 크기가 다른 회로 설계에서 얻어지는 행렬들이다. 회로의 크기가 증가할수록 (A,H 행렬의 크기가 증가할수록) A,H 행렬의 비영값의 비율은 낮아진다.

그림 8에서 각 실험 집합에 대하여 2단계 연산 방법과 1단계 연산 방법의 부동소수점연산 횟수를 비교하였다. 1단계 연산 방법에서 부동소수점연산 횟수가 모두 50% 정도로 감소함을 알 수 있다. 또한 프로세스의 메모리 사용량 1단계

Set	# of rows in A	# of columns in A	A의 비영값 갯수	H의 비영값 갯수
1	8648	42750	361K	195K
2	14872	77406	667K	361K
3	21096	112150	997K	528K
4	39768	217030	1913K	1028K
5	41392	244501	1633K	963K

표 1. 실험에 쓰인 행렬들
Table 1. matrix sets used in the performance measurement

연산 방법에서 50% 정도 감소하는데 그 이유는 1단계 연산에서 A의 전치행렬을 암시적으로 사용하고, 또한 H행렬과 이로 인한 곱행렬 P의 대칭성을 활용하고 있기 때문이다.

그림 9에서 900 MHz Itanium II에서 각 데이터 집합에 대하여, 2단계 연산에 대한 1단계 연산의 성능 개선 (speedup)을 비교하였다. 사용된 platform은 L1/L2/L3 캐쉬 크기가 각각 32K/256K/1.5MB이다. Itanium II의 부동소수점연산장치는 모두 4개이므로 이론적 최고 연산 속도는 3.6 Gflops/s가 가능하다. 컴파일러는 linux 상에서 intel C compiler v7.0을 사용하였고 컴파일 옵션은 -O3를 사용하였다. 실험 결과 일관되게 1.9~2.1의 성능 개선치를 보이고 있는데 이는 주로 그림 8에서 보여지는 부동 소수점 연산의 감소와 또한 이에 따른 메모리 접근 회수의 감소에 기인한다.

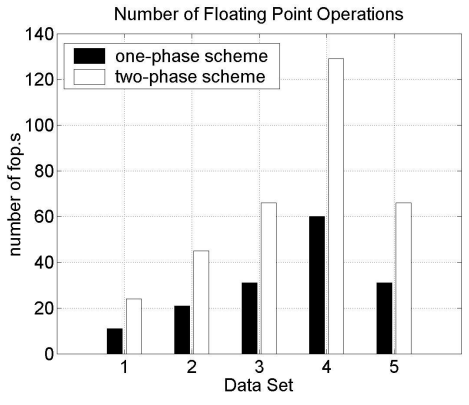

그림 8. 1단계 연산 방법과 2단계 연산 방법에서 실험 데이터 집합에 대한 부동 소수점 연산 횟수 비교
Fig 8. comparison of the number of floating point operations in one-phase scheme and in two-phase scheme.

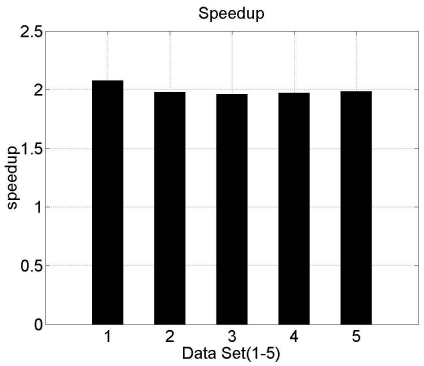

그림 9. 실험 데이터 집합에서 2단계 연산 방법에 대한 1단계 연산 방법의 성능 개선치
Fig 9. Speedup of one-phase scheme over two-phase scheme for the data set

이와 같이 개선된 연산 속도가 이론적으로 도달할 수 있는 성능에 얼마만큼 도달하고 있는지를 성능 모델링을 통하여 얻어진 예측치에 대하여 백분율로 그림 10에 보인다.

마지막으로 1단계 연산과 2단계 연산의 전처리에 필요한 오버헤드를 분석해 본다. 전처리 시간은 전체 반복 연산에 대하여 초기에 단 한 번 요구되는 시간인데 그림 11에서 각 데이터 집합에 대한 1단계 연산과 2단계 연산의 전처리 시간을 실제 연산 (1번의 곱셈) 에 소요된 시간에 대한 비율로 보여준다. 이 비율 상으로도 1단계 연산의 전처리 시간 비율이 2단계 연산보다 작은 것으로 나타나는데, 실제 연산 시간이 1단계 연산의 경우 2단계 연산의 50% 정도로 짧은 것을 고려하면 1단계 연산이 전처리 시간에 있어서도 훨씬 개선되었음을 알 수 있다. 이 전처리 과정에 필요한 연산들을 분석하면 다음과 같다.

(1) 1 단계 연산의 경우

(1-1) P행렬의 비영값의 개수만을 계산하여 P행렬을 저장하기 위한 메모리 할당량을 계산하기 위하여 소요되는데 이는 실제 곱셈 구현에서 연산 부분만 제외하고 실행함으로써 얻어진다.

(1-2) 모든 i에 대하여 $B_i = A_i r_i$ 의 벡터를 저장하는 것도 역시 희소행렬의 형태로 저장되는데 여기 필요한 메모리 할당량을 계산한다.

(1-3) 이 연산이 반복될 때 행렬의 비영값이 위치가 변하지 않고 값만 변화하는 점을 이용하여 이후의 반복되는 곱셈에서 반복 사용될 수 있는 데이터 구조를 미리 구성한다.

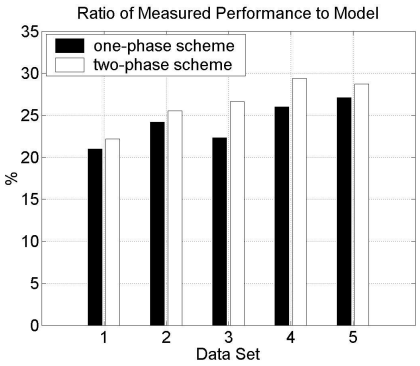

그림 10. 성능 모델에 의한 이론적 성능 최고치에 대한 측정된 성능의 백분율

Fig 10. Percentage of measured performance over theoretical peak performance predicted by a performance model

(2) 2단계 연산의 경우

(2-1) Q와 P행렬의 비영값의 개수만을 계산하여 Q와 P 행렬을 저장하기 위한 메모리 할당량을 계산하기 위하여 소요되는데 이는 1단계 연산에서와 마찬가지로 실제 곱셈 구현에서 연산 부분만 제외하고 실행함으로써 얻어진다.

(2-2) 행렬 A로부터 A의 전치행렬 (A^t) 을 계산하여 저장한다.

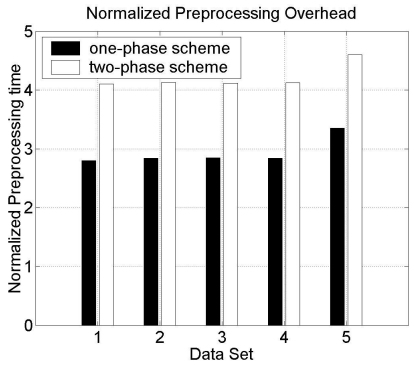

그림 11. 1단계 연산 방법과 2단계 연산 방법에서의 전처리 오버헤드 비율

Fig 11. Normalized prepocessing overhead time in one-phase scheme and in two-phase scheme

V. 결론

희소 행렬 간의 곱셈은 본질적으로 비효율적인 메모리 접근 때문에 밀집 행렬 연산과 비교해서도 성능이 대체로 낮다. 이와 같은 문제를 해결하기 위하여 밀집 행렬의 경우 [7]와 마찬가지로 기본 행렬 연산에 대한 표준 인터페이스를 제안하고 성능 최적화가 적용된 라이브러리를 개발하는 연구가 진행되어왔다. 한 편 희소 행렬 연산의 성능은 비영값들의 분포와 밀접한 영향을 받기 때문에 응용 분야에 따라 생성되는 희소 행렬의 특수한 성질을 잘 이용하면 연산 속도를 높이는 것이 가능하다.

희소 행렬의 연산의 비효율성은 간접적 데이터 구조와 비순차적 메모리 접근의 두 가지 이유 때문인데 이에 대한 연구는 [2][3][8]에서 희소 행렬과 밀집 벡터의 경우에 대하여 register blocking과 cache blocking의 방법이 제안되어 높은 성능 개선 효과를 보였고 Pinar and Heath [9]는 이와 같은 blocking의 효과를 높이기 위하여 행렬의 행과 열의 순서를 바꾸는 방법이 제안되었다. 희소 행렬간의 곱셈에 관한 연구는 [10][11]에서와 같이 일반적인 경우에 대한 라이브러리가 개발되어 있지만 본 연구에서는 대각 블록 행렬 H와 A행렬 간의 AHA^t 형태의 곱셈을 수행하는 특수한 구조의 연산을 다루고 있다.

본 논문에서는 희소 행렬의 연산 중, 회로 설계 시 최적화 과정에서 primal-dual optimization 문제의 해를 구하기 위하여 Schur complement를 계산할 때 희소 행렬 A 와 블록 대각 행렬 H에 대하여 AHA^t 의 연산을 효율적으로 행하는 방법들을 검토하고 수행 시간과 메모리 사용량 면에서 비교하였다. 이와 같은 연구는 지금까지의 연구와 달리 주어진 문제 도메인에 특수한 성질을 이용하여 하위의 연산 단계에서 도출할 수 없는 상위 알고리즘 단계의 성능 개선을 사용한 연구로 의미가 있다.

참고문헌

[1] J. W. Demmel. "Applied Numerical Linear Algebra". SIAM, 1997

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

2000

[3] 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

[4] 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.

[5] 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

[6] J. R. Gilbert, C. Moler and R. Schreiber, "Sparse Matrices in Matlab: Design and Implementation". SIAM J. Matrix Analysis and Applications, 13:333-356, 1992

[7] 이 원주, 김 선욱, 김 형래, "Parsec 기반 시뮬레이터를 이용한 다중처리시스템의 성능 분석", 한국컴퓨터정보학회 논문지, 제 11권, 제 2호, pp. 35-42, May 2006

[8] Richard W. Vuduc. "Automatic performance tuning of sparse matrix kernels". PhD thesis, University of California, Berkeley, Dec. 2003

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

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

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

저 자 소 개

임 은 진

1991년 서울대학교 컴퓨터 공학과 (공학사)
1993년 서울대학교 대학원 컴퓨터공학과 (공학석사)
2000년 U.C.Berkeley Ph.D.
2001년 ~ 현재 : 국민대학교 컴퓨터학부 조교수
관심분야 : 컴퓨터 시스템, 컴파일러, 성능 모델링