VLIW 시뮬레이터 상에서의 디지털 신호처리 행렬 연산에 대한 병렬화 알고리즘

송 진 희† · 전 문 석††

요 약

본 논문에서는 행렬 또는 벡터 곱셈을 선형 프로세서나 VLIW 시뮬레이터로 분할 및 배치하는 알고리즘을 제안한다. 먼저 입력 행렬이나 벡터를 임의 크기의 프로세서 배열에 배치하는 기법에 대해 논의하고, 문제 크기를 프로세서 배열 크기로 분할하는 알고리즘을 보인다. 이 알고리즘을 VLIW 시뮬레이터 상에서 실행하고 알고리즘의 효율성을 보이도록 한다. 그 결과 우리가 설계한 VLIW 시뮬레이터 상에서의 수행이 선형 프로세서 상에서 보다 병렬화 성능이 향상됨을 알 수 있었다.

A Parallelising Algorithm for Matrix Arithmetics of Digital Signal Processings on VLIW Simulator

Jin-Hee Song† · Moon Seog Jun††

ABSTRACT

A parallelising algorithm for partitioning and mapping methods of matrix/vector multiplication into linear processor array/VLIW simulator is presented in this paper. First, we discuss the mapping methods for input matrix or vector into the arbitrarily size of processor arrays. Then, we show partitioning the algorithmss of the large size of computational problem into the size of the processor array. We execute the algorithm on VLIW simulator and show to effectiviness of algorithm. The result which we achived better parallelising performance on our VLIW simulator design than on linear processor array.

1. 서 론

VLSI 기술 향상으로 인하여 다양한 신호 처리 알고리즘에 있어서의 처리율을 증진하기 위해 여러 개의 상용 프로세서들을 사용하는 것에 관심이 집중되고 있다. 따라서 디지털 신호 처리에 대한 문제와 Kalman Filtering에 의한 최적의 선형 최소 편차 예측치는 레이다 신호 처리를 포함한 미사일 제어 유도와 비행 예측, 목표물 예측 또는 추적에 응용된다.이것은 많은 입력 데이타가 필요하거나 대규모 계산이 요구되는 과학 계산 분야인 VLSI설계, 로보트 시스템, 일기예보, 영상 처리 및 비전 등에 적용될 수 있다. 디지털 신호 처리를 위한 특수 목적의 DSP 구조와 병렬화에 대한 연구가 계속되고 있으나 DSP 구조는 범용성이 없다[1]. 따라서 DSP에 비해 상대적으로 융통성이 많은 VLIW 병렬 컴퓨터에 대한 연구들이 진행되고 있으며, VLIW는 특히 신호 처리 분야에 적합하다. 이는 복잡한 계산 위주의 신호 처리 문제들을 한 명령어 내에서 동시에 많은 병렬 처리를 함으로써 속도 향상과 컴퓨터 효율을

※ 이 연구는 1994-'97년도 한국 과학재단의 특정 연구 과제 연구비 지원(과제번호 94-0100-14-3)으로 수행되었음.
† 정 회 원 : 신흥대학 전산정보처리과 조교수
†† 종신회원 : 숭실대학교 전자계산학과 부교수
논문접수 : 1997년 10월 6일, 심사완료 : 1998년 6월 1일

능일 수 있기 때문이다.

행렬-행렬 곱셈(Matrix-Matrix Multiplication)은 VLIW 병렬 컴퓨터에서 많이 사용되는 디지탈 신호 처리 중 한 가지로서 이에 대한 VLSI 배열 프로세서(Processor Arrays)와 병렬 알고리즘에 대한 연구들이 많이 수행되고 있다[2-4]. 디지털 신호 처리에 대한 병렬처리 연구와 주어진 문제 크기를 프로세서의 크기로 분할하여 처리하는 입력 자료 분할 기법이 제안되었다[5].

행렬 곱셈에 사용되는 시스톨릭 배열은 메쉬형, LU 분해에 사용되는 육각형 시스톨릭 배열 및 삼각 방정식을 위한 선형 시스톨릭 배열로 연결하여 사용한다. 시스톨릭 배열의 일반적인 구성 셀의 수는 n^2이다. 따라서 문제 크기가 클 경우 배열을 구성하는 셀 수도 증가하여 H/W 설계가 복잡해 진다.

본 논문에서는 선형 접속 형태의 k-개의 배열 프로세서 또는 VLIW 시스템에서 행렬-행렬 곱셈과 Kalman Filter의 행렬-벡터 연산을 수행하기 위한 입출력 자료의 배치와 분할 기법을 소개한다. 또한 제안된 기법에 의한 연산 처리를 VLIW 컴퓨터 상에서 수행하여 응용 프로그램 수준에서의 병렬 수행 결과인 처리 속도의 향상에 대해 살펴본다. 일반 시스톨릭 어레이에서의 낮은 PE 사용율에 비해 제안한 알고리즘에 의한 선형 접속 형태의 프로세서의 사용율은 100%에 이른다. 그러므로 실행 시간에 대한 향상을 기대할 수 있다.

2. 행렬 곱셈을 위한 고정 크기의 선형 프로세서 배열

시스템을 구성하는 각 프로세서들은 인접한 프로세서들끼리 서로 연결된다. 선형 접속된 각 프로세서들은 규칙적으로 연결되어 있으며, 각 프로세서에 입출력되는 데이타들은 규칙적인 흐름을 가진다. 또한 입력된 자료들은 전역 시간에 동기적으로 수행 및 전달되는 특성을 가지고 독립적으로 수행된다. 각 PE는 입력된 자료에 대해 곱셈과 덧셈 연산을 수행한다.

사용 PE의 갯수가 증가할수록 하드웨어의 설계는 복잡해지며, 사용 PE의 수와 하드웨어 설계 복잡도 간에는 trade-off이 존재하게 된다. 또한 기존의 배열 프로세서는 어느 한 사용 시점에서의 PE 사용율은 낮은 것으로 알려져 있다. 행렬 곱셈을 문제-크기 시스톨릭 어레이로 설계할 경우 메쉬 접속 어레이 상에서는 PE 사용율이 33%이다[6]. 그러므로 처리할 행렬의 크기에 따라 사용 PE 수를 무한정 확장하는 것보다 적정 수의 사용 PE를 제한하여 처리할 수 있는 실용적인 측면을 고려할 필요성이 생긴다.

본 논문에서는 문제의 크기와는 관계없이 처리할 프로세서의 수가 제한되는 경우 행렬 연산을 위한 입력 자료에 대한 배치 방법을 소개하고자 한다.

2.1 행렬-행렬 곱셈을 위한 입력자료 배치

(n x n) 크기의 이차원 행렬 $A=\{a_{ij}\}$와 $B=\{b_{kj}\}$를 곱한 결과 행렬을 C_{ij}라 하자. 결과 행렬에 대한 수식은 다음과 같다.

$$C_{i,j}=\sum_{k=1}^{n} a_{i,k}b_{k,j} \quad 1\le i, j \le n \quad \ldots (1)$$

시스톨릭 어레이를 설계하기 위해서는 주어진 입력 문제(행렬), 입력 자료들에 대한 배치, 행렬 곱셈을 수행하기 위한 공간 함수가 필요하다. 각 PE는 곱셈 및 덧셈을 수행하며, 또한 계산 결과를 기억할 수 있어야 한다.

사용 가능한 PE의 수를 $1\le k\le n$로 제한되며, 해당 프로세서들을 가장 간단한 방법인 선형으로 연결할 때, 연결된 각 PE는 주어진 행렬 A의 원소들을 왼쪽에서 오른쪽으로 전달한다고 가정하자. 행렬 A의 원소가 입력되는 첫번째 PE에는 (n-k) 크기의 버퍼가 연결되며, 이 버퍼는 시스톨릭 어레이의 경계 위치에 있는 두 개의 PE와 상호 연결된다. 행렬 A의 원소들은 한 행씩 PE와 버퍼에 순차 배치된 상태에서 (n-1)회 이동(shift)한다. 행렬 B의 원소들은 top-down방향으로 PE에 순차 입력되며, 각 PE에는 행렬 B의 열이 수식 ((d-1)k+p))에 의해 전달된다(d: 분할 수행 단계, p: 각 프로세서의 인덱스, k: 프로세서 갯수). 입력된 행렬 원소들의 연산 결과는 각 PE에 누적되며, 각 PE는 n번의 곱셈을 수행한 후, 누적값을 동시에 출력한다. PE가 누적된 곱셈 결과값을 출력한 후 누적합은 '0'으로 초기화하며,입력 자료가 모두 처리될 때까지 반복 수행한다. 행렬 곱셈을 수행하기 위한 공간 함수(행렬)와 입력 자료의 배치 함수들에 대해 살펴보자.

(4 x 4) 행렬 A, B에 대한 곱셈을 수행하기 위한 입력 함수의 초기 배치는 다음과 같다.

초기 배치에서는 행렬 A의 첫 행에 대한 배치만 보였으며, 2행부터는 열의 위치는 변화없이 행 번호만 다

은 채 행 순서로 순차 배치될 것이다. 즉, 행렬 A의 첫 행 원소들이 n-1번 shift 처리된 후, 각 PE는 A, B의 곱셈 결과인 C의 첫 행을 동시에 k개씩 출력한다. 그 후, 각 PE에는 A의 두 번째 행이 초기 상태와 같이 배치되고, 직전에 PE에 입력되었던 행렬 B의 열 원소들은 A의 모든 행들이 연산 완료될 때까지 상기 과정을 반복 수행하며, C의 다음 번 결과들이 동시에 출력된다. 이와같이 A의 모든 행에 대해 반복 처리를 하면 곱셈 결과인 C의 모든 원소들이 계산 출력된다.

행렬 A(n x n)의 입력 자료 처리에 관련된 버퍼는 n의 크기가 PE 수(k)보다 큰 경우에만 FIFO 방식에 따라 사용된다(k<n). A의 원소는 행 단위로 순차적인 순환 입력이 되고, B의 원소들은 각 PE에 동시에 입력된다. PE에서 A, B의 각 원소들이 한 번 연산 수행이 이루어진 후, A의 원소들은 오른쪽으로 한 번씩 이동되며 n번의 곱셈이 수행될 때까지 모든 PE는 그 결과를 누적한다. C는 일정 횟수(n)가 지난 후, 각 PE에서 동시 출력된다.

각 PE에 할당되는 행렬 A와 B 입력 자료들에 대한 배치는 다음에 의한다.

【 행렬 A의 입력 배치 】

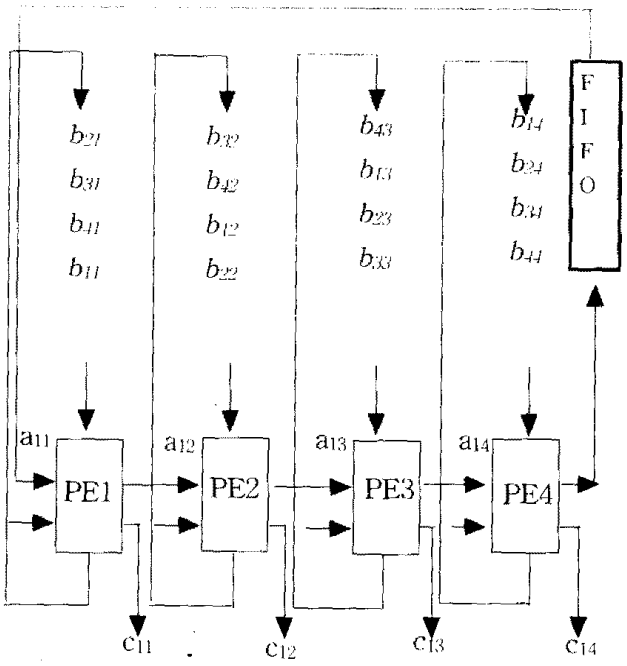
행렬 A는 한 행이 모두 연산 완료될 때까지 순차적으로 입력 처리되며, 1열부터 PE^1~PE^k와 경계에 위치한 프로세서에 연결된 버퍼로 n열까지 차례로 할당된 후 이동 처리된다. 그러므로 행렬 A는 각 행마다 최초의 열 배치 이후에는 n-1회의 이동 순회 처리만 고려하면 된다. 각 입력 행의 결정은 수행 단계 s와 동일하다.

PE_{ij}^{Ast}: p번째 PE에 입력되는 행렬 A의 원소

- s : 수행 단계(1≤s≤n, s = i Mod n)
- p : 프로세서 인덱스(1≤p≤k)
- i : 행렬 A의 입력 행번호(s = i)
- j : 행렬 A의 입력 열번호

【 행렬 B의 입력 배치 】

행렬 B는 각 PE에 대한 열들을 결정하고, 각 열에 대한 행의 초기 할당 후 다음에 수행할 행을 결정한다. B의 열은 n/d번 분할 수행되며 1열부터 PE^1~PE^k에 차례로 할당된다. 그러므로 행렬 B는 각 열이 최초 배치되면 이후의 행과 열은 수식(2)~(4)에 의한다.



(그림1) (4x4) 행렬의 초기 입력 자료 배치
(Fig.1) Initial input data allocate of (4x4) Matrix

PE_{ij}^{Bdt}: p번째 PE에 입력되는 행렬 B의 원소

- k : 프로세서 수
- d : 분할 수행 단계(1≤d≤n/k),
- p : 프로세서 인덱스(1≤p≤k))
- i : 행렬 B의 입력 행번호(1≤i≤n)
- j : 행렬 B의 입력 열번호(1≤j≤n, j = (d-1)k+p)

· 분할 수행 단계 : $1 \le d \le n/k$

· PE^p에 입력될 연산 대상 행 :

```
if (i Mod n == 1) then
  i = n
else
  i = i + 1
end if                          ···(2)
```

· PE^p에 할당될 열 : $j = (d-1)k+p$. ···(3)

· PE^p의 최초 배치 원소 :

$PE^p = B_{ij}$ $(i = j = (d-1)k+p)$ ···(4)

다음은 (4 × 4) 입력 행렬에 대한 곱셈 처리 과정을 단계별로 나타낸 것이다.

	【PE^1】	【PE^2】	【PE^3】	【PE^4】
단계 1〉	$a_{11} \cdot b_{11}$	$a_{12} \cdot b_{22}$	$a_{13} \cdot b_{33}$	$a_{14} \cdot b_{44}$
단계 2〉	$a_{12} \cdot b_{11}$	$a_{11} \cdot b_{12}$	$a_{12} \cdot b_{23}$	$a_{13} \cdot b_{31}$
단계 3〉	$a_{13} \cdot b_{11}$	$a_{11} \cdot b_{12}$	$a_{11} \cdot b_{13}$	$a_{12} \cdot b_{21}$
단계 4〉	$a_{11} \cdot b_{21}$	$a_{13} \cdot b_{32}$	$a_{11} \cdot b_{13}$	$a_{11} \cdot b_{11}$
단계 5〉	c_{11}	c_{12}	c_{13}	c_{11}
단계 6〉	$a_{21} \cdot b_{11}$	$a_{22} \cdot b_{22}$	$a_{23} \cdot b_{33}$	$a_{24} \cdot b_{11}$
단계 7〉	$a_{24} \cdot b_{11}$	$a_{21} \cdot b_{12}$	$a_{22} \cdot b_{23}$	$a_{23} \cdot b_{34}$
단계 8〉	$a_{23} \cdot b_{31}$	$a_{24} \cdot b_{42}$	$a_{21} \cdot b_{13}$	$a_{22} \cdot b_{24}$
단계 9〉	$a_{22} \cdot b_{21}$	$a_{23} \cdot b_{32}$	$a_{24} \cdot b_{43}$	$a_{21} \cdot b_{14}$
단계10〉	c_{21}	c_{22}	c_{23}	c_{24}
단계11〉	$a_{31} \cdot b_{11}$	$a_{32} \cdot b_{22}$	$a_{33} \cdot b_{33}$	$a_{34} \cdot b_{44}$
단계12〉	$a_{34} \cdot b_{11}$	$a_{31} \cdot b_{12}$	$a_{32} \cdot b_{23}$	$a_{33} \cdot b_{34}$
단계13〉	$a_{33} \cdot b_{31}$	$a_{34} \cdot b_{42}$	$a_{31} \cdot b_{13}$	$a_{32} \cdot b_{21}$
단계14〉	$a_{32} \cdot b_{21}$	$a_{33} \cdot b_{32}$	$a_{34} \cdot b_{43}$	$a_{31} \cdot b_{14}$
단계15〉	c_{31}	c_{32}	c_{33}	c_{34}
단계16〉	$a_{41} \cdot b_{11}$	$a_{42} \cdot b_{22}$	$a_{43} \cdot b_{33}$	$a_{44} \cdot b_{41}$
단계17〉	$a_{44} \cdot b_{41}$	$a_{41} \cdot b_{12}$	$a_{42} \cdot b_{23}$	$a_{43} \cdot b_{34}$
단계18〉	$a_{43} \cdot b_{31}$	$a_{44} \cdot b_{42}$	$a_{41} \cdot b_{13}$	$a_{42} \cdot b_{24}$
단계19〉	$a_{42} \cdot b_{2}$	$a_{43} \cdot b_{32}$	$a_{44} \cdot b_{43}$	$a_{41} \cdot b_{14}$
단계20〉	c_{41}	c_{42}	c_{43}	c_{44}

(4 x 4) 행렬 연산에서 프로세서 수를 k=4, 분할 단계 d=n/k=1, 프로세서 인덱스 p가 $1 \le p \le 3$으로 가정하자. (그림2)와 같이 각 PE^p에는 행렬 A의 1행 원소들이 왼쪽에서 오른쪽으로 순차적으로 배치된다. k=n이므로 입력 버퍼에 배치되는 A의 원소는 없다. PE에 입력될 행렬 B의 초기 입력 열은 식(3)에 의해 PE^1 = 1열(j=1), PE^2 = 2열, PE^3 = 3열, PE^4 = 4열이 배치된다. B의 열에 대한 결정은 d=1이므로 식(4)에 의해 ① $PE^1 = (1-1)4+1=1(B_{11})$ ② $PE^2=(1-1)4+2=2(B_{22})$ ③ $PE^3=(1-1)4+3=3(B_{33})$ ④ $PE^4=(1-1)4+4=4(B_{44})$가 차례로 배치된다. 〈단계1〉에서 각 PE의 A, B 원소들이 곱셈을 수행하고 A의 원소들은 오른쪽으로 1회 이동한다. 〈단계2〉에서 각 PE의 B원소에 대한 결정은 (2)에 의해 PE^1만이 (i=1, 1 Mod 4 = 1)이 되어 i=4 즉, 4열이 선택(b_{41})된다. 나머지 PE들은 i Mod 4+1이므로 〈단계1〉에서의 열 번호에서 1이 작은 열들이 선택되어($PE^2 = b_{12}$, $PE^3 = b_{23}$, $PE^4 = b_{31}$)가 배치 처리되며, 수행 결과는 〈단계1〉의 결과 값에 누적된다. 이 과정을 n회(4번) 수행하고 〈단계5〉에서 각 PE들은 누적된 결과를 출력한다. 〈단계1~5〉의 과정을 행렬 A의 n행만큼 반복 수행하면 분할 1단계가 완료된다. 상기 예에서는 다음 분할이 필요하지 않으므로 작업이 종료되나 (n x m)행렬에서 k<m이면 각 PE에 식(2)~(4)에 의해서 식(3), (4), (2)의 순서에 따라 B의 원소들이 결정되며, A의 다음 행 원소들이 초기 배치 형태로 배치되어 상기 과정을 반복 수행하게 된다.

2.2 Kalman Filtering의 상태 및 측정 벡터 연산을 위한 입력 자료 배치

Kalman filter는 n-차원의 상태 벡터 x(k)와 m-차원의 측정 벡터 y(k)로 구성되며 다음 방정식으로 표현된다.

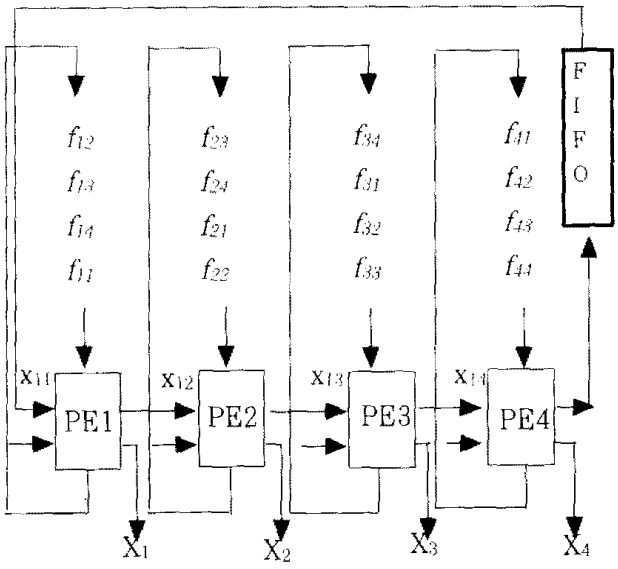
$$\begin{cases} x(k+1) = F(k)x(k) + w(k) & \cdots\cdots(5) \\ y(k) = C(k)x(k) + v(k) & \cdots\cdots(6) \end{cases}$$

위에서 F(k), C(k)는 각각 (n x m)과 (m x n) 차원의 계수 행렬이다. w(k)는 n-차원의 잡음 벡터이고, v(k)는 m-차원의 측정 잡음 벡터이다. 각각의 공분산 행렬 $R_w(k)$와 $R_v(k)$는 서로 독립적이며, 잡음 w, v는 상관 관계가 없다고 가정한다.

Kalman filter는 측정 벡터{y(1), y(2), ···y(k)}를 알면 상태 벡터 x(k+1)=x(k+1|k)의 최상의 예측치를 구할 수 있다. 방정식 갱신에 필요한 많은 계산들은 적어도 $O(n^3)$이 요구되어 병렬 처리를 필요로 한다. 따라서 Paige와 Saunders에 의해 제안된 Kalman filter의 최저 평방식을 기본으로 여러 가지 시스톨릭 배열들이 제안 및 연구되고 있다[7-11]. Kalman Filter는 최저 평방식을 사용하기 위한 시스템과 측정상 발생하는 잡음을 제거, Givens 전환에 의한 축 회전법에 따라 행렬의 부 대각선 원소들을 차례로 0으로 만들기, 행렬 C(k)를 삼각 행렬 R(k)로 무효화시키는 QR 무효화, RCD를 사용한 LU 분해, 데이터 Pre-whitening을 위한 사후 곱셈화 등의 복잡한 과정을 거쳐야 한다. 이러한 처리는 이차원의 시스톨릭 배열 상에서 수행에 관해 주로 연구되어 왔으며, 이 장에서는 Kalman filter 방정식의 행렬과 벡터에 대한 입력 자료를 k개의 선형 프로세서 어레이에서 처리할 경우에

대한 배치 기법에 대해 제안하려 한다.

(m x n)의 계수 행렬에 대한 입력은 2.1에서 소개한 행렬 B의 배치 방법과 동일하다. 즉, 상태 전이 행렬 F(k)와 측정 행렬 C(k)에 대한 배치는 2.1에서의 행렬 B와 같은 형태로 배치하며, 벡터 x(k)는 행렬 A의 형태로 배치한다. 단, (m x n)차원의 F(k) 또는 C(k)는 각 행을 중심으로 1행부터 차례로 PE^p에 프로세서 인덱스 p에 순차적으로 배치되며, 남은 행(전체 분할 횟수 : m/k)들은 순차적으로 분할 수행된다. 벡터 x(k) 역시 PE^p에 순차적으로 배치되고 남은 원소들은 버퍼에 차례로 입력되어 n-1회의 순환 이동 연산이 이루어지면 각 PE^p의 결과가 출력된다. n이 전체 프로세서 수보다 클 때, 이에 대한 행렬-벡터에 대한 프로세서 할당은 (그림2)와 같다.



(그림 2) (4x4) 행렬-벡터에 대한 초기 배치
(Fig. 2) Initial allocate of (4x4) Matrix-Vector

(그림2)에서의 행렬과 벡터에 대한 배치는 다음 기호들에 따른다.

【 행렬 F의 입력 배치 】

PE_{ij}^{Fdp}
: p번째 PE에 입력되는 행렬 F의 원소
- k : 프로세서 수
- d : 분할 수행 단계($1 \le d \le m/k$),
- p : 프로세서 인덱스($1 \le p \le k$))
- i : 행렬 F의 입력 행번호($1 \le i \le m$,
  $i = (d-1)k+p$)
- j : 행렬 B의 입력 열번호($1 \le j \le n$)

· 분할 수행 단계 : $1 \le d \le m/k$
· PE^p에 할당될 행 : $i = (d-1)k+p$ ···(7)
· PE^p에 입력될 연산 대상 열 :

```
if (j Mod n == 1) then
  j = n
else
  j = j + 1
end if
```
···(8)

· PE^p의 최초 배치 원소 :
$PE^p = C_{ij}$ 또는 F_{ij} $(i=j=(d-1)k+p)$ ···(9)

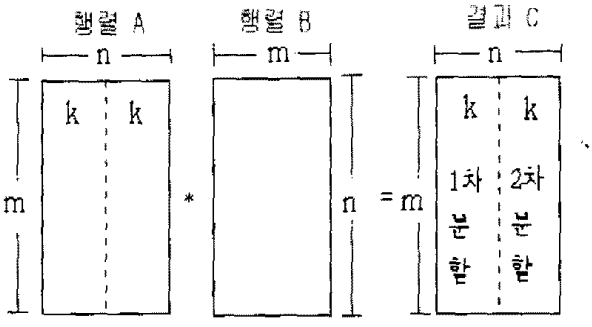
【벡터 X의 입력 배치】
· PE^p에 할당될 행 :
① $i = (d-1)k+p$ ···(10)
② If i>p then allocate to buffer with FIFO scheduling by row i ··· (11)

(그림3)과 같은 배치 방식에 의하면 행렬-벡터 연산에 대한 행 중심의 처리가 완료되고, 벡터에 대한 배치는 그대로 둔 채 행렬 F(k)의 나머지 모든 행들도 같은 방식으로 배치함으로 연산을 완료할 수 있다.

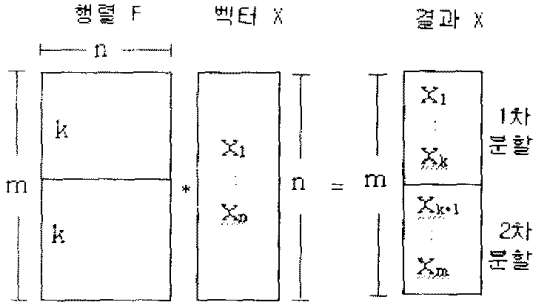
2.3 행렬 곱셈의 분할 기법

(n × n)행렬-행렬 곱셈이나 (m x n)의 행렬-벡터에 대해 k개의 PE로 연산을 수행하려면 분할 알고리즘이 필요하다.

【 행렬-행렬에 대한 분할】



【 행렬-벡터에 대한 분할】



상기에서와 같이 제안된 알고리즘에서는 행렬-행렬 곱셈은 결과 행렬 C의 모든 행이 열 중심으로, 행렬-벡터에서는 해당 행렬의 행 중심으로 k개씩 분할되어 차례로 수행된다.

이 때의 행렬에 대한 전체 분할 수행 횟수는 각각 ⌈n/k⌉, ⌈m/k⌉가 된다. 즉, (n x n)행렬-행렬 곱셈에서는 첫 번째 분할 단계에서 {(C_{11}, C_{12}, ... , C_{1k}), (C_{21}, C_{22}, ... , C_{2k}), · · · , (C_{n1}, C_{n2}, ... , C_{nk})}가 계산된다. 두 번째는 {($C_{1,k+1}$, $C_{1,k+2}$, ... , $C_{1,2k}$), ($C_{2,k+1}$, $C_{2,k+2}$, ... , $C_{2,2k}$), · · · , ($C_{n,k+1}$, $C_{n,k+2}$, ... , $C_{n,2k}$)}이 계산되며, d번째 분할 단계에서 {($C_{1,(d-1)k+1}$, $C_{1,(d-1)k+2}$, ... , $C_{1,dk}$), ($C_{2,(d-1)k+1}$, $C_{2,(d-1)k+2}$, ... , $C_{n,dk}$)}가 계산된다.

(m x n)의 행렬-벡터 연산에서는 {(X_1, X_2, ... , X_k), ($X_{(d-1)k+1}$, $X_{(d-1)k+2}$, ... , $X_{(d-1)k+k}$), · · · , ($X_{(d-1)k+1}$, $X_{(d-1)k+2}$, ... , X_m)}가 분할 수행된다. 따라서 입력 자료 행렬의 행 크기와는 관계없이 k개로 구성된 선형 프로세서 배열 상에서 프로세서의 수(k)와 행렬(m x n) 또는 행렬(n x n) 크기만 상수로 지정하면 입력 자료의 크기나 프로세서의 수 등에 무관하게 프로그램을 수행할 수 있다.

3. VLIW 시뮬레이터 상에서의 행렬 연산

2.1과 2.2에서 제안된 알고리즘을 VLIW 시뮬레이터 상에서 병렬 수행에 대해 살펴보자. 실험에 사용 한 시뮬레이터는 SUN SPARC 10에 Solaris 2.5.2 운영체제를 사용하였다. 시스템의 Functional Unit은 정수연산 처리기(IU: Integer Unit)와 부동 소수점 처리기(FU: Floating Pointer Unit)로 구성된다.

실험에 사용된 프로그램은 C 언어로 작성되었으며, 2.1과 2.2의 행렬-행렬에 대한 곱셈과 Kalman Filter의 행렬-벡터에 대한 곱셈을 입력 행렬 크기를 변화시켜 제안된 입력 배치 기법에 따라 수행할 수 있도록 작성되었다.

【알고리즘1. Matrix-Matrix Multiplication】

```
/* 1. 행렬 A, B의 입력 자료는 파일
   2. k : 프로세서 개수
      m, n : 행렬의 크기(m x n 또는 n x m)
   3. PE : 프로세서
   4. SUM : 각 프로세서의 연산 결과 누적 처리기
   5. t_part : 행렬 연산에 필요한 전체 분할 횟수
      c_part : 현재의 분할 수행 인덱스
      shift_cnt : 행렬 A의 각 요소에 대한 Shift 횟
                  수(= n-1)
   6. i, j : 행렬의 행, 열을 표시하는 인덱스      */

단계1.  declare constant k, n;
        /* 전체 분할 횟수 결정 */
        t_part = ⌈ n / k ⌉ ;        c_part = 1;
        shif_cnt = 0;
단계2.  Read A, B
        /* 행렬 A, B의 입력 자료 읽기와 프로세서,
        연산 누적 처리기의 초기화 */
단계3.  Initialize all PEs and Sums
단계4.  /* 행렬 연산 수행 */
        If c_part 〉 t_part then goto 15;
단계5.  Allocate first row elements of A to each
        PEs by sequential column(FIFO)
단계6.  Select columns of B to each PEs
단계7.  Select rows of B to each PEs
단계8.  Allocate elements of B to PEs
단계9.  Multiply each PEs
        Accumulate each results of PEs to
        Sums
단계10. Shift to right elements of A
        shift_cnt++
단계11. If (shift_cnt 〈= n-1 ) then
          goto 단계6;
단계12. Select next row elements of A
단계13. Initialize all PEs and Sums
단계14. If ( row of A ≠ n ) then
          Allocate the row elements of A to PEs
```

```
        goto 단계6
    else
        c_part++;
        goto 단계3;
    endif
단계15. stop.
```

【알고리즘1】에 의한 C 프로그램을 작성하여 순차 수행 및 VLIW 시뮬레이터 상에서 수행한 결과 〈표1〉과 같다.

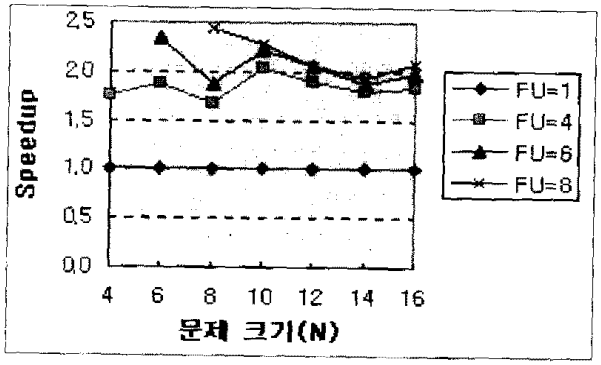
FU와 문제 크기에 각각 변화를 주고 실험한 결과 동일한 프로그램에 대한 명령어 실행 사이클 수가 달라짐을 알 수 있다. 순차 실행의 경우는 시뮬레이터의 단위 처리기를 IU=1, FU=1로 수행한 것이며, VLIW 실행의 경우에는 (IU=4, FU=4), (IU=6, FU=6), (IU=8, FU=8)로 각각 설정하였다. 문제 크기(N)는 행렬 입력 크기를 나타낸 것이다. 프로그램의 실행 사이클에 대한 비교를 하여 보면, 알고리즘의 행렬 연산에 대한 반복 수행문에 대한 Loop 변수를 증가함에 따라 실행 속도가 조금씩 증가되었다. 이것은 한 번에 병렬 처리할 수 있는 명령문의 수를 증가시키므로써 명령어 사이클 수를 감소시킬 수 있음을 알 수 있는 것이다. 【알고리즘1】에서는 최대 병렬 처리할 수 있는 반복 명령문의 수를 문제 크기인 n으로 지정하였다.

〈표 1〉 FU와 문제 크기 변화에 따른 속도
(행렬-행렬 곱셈)

〈Table 1〉 The Speedup based on FU numbers and Problem Sises(Matrix-Matrix Multiplication)

FU 수 / N	순차 실행	VLIW 실행			
		k=n	k⟨n	Max(k)	Min(k)
4x4	1	1.76		1.76	1.76
6x6	1	2.34	1.87	2.34	1.87
8x8	1	2.43	1.78	2.43	1.68
10x10	1		2.23	2.40	2.04
12x12	1		2.05	2.16	1.90
14x14	1		1.93	2.03	1.80
16x16	1		2.07	2.20	1.83

〈표1〉의 결과에 따르면 두 행렬에 대한 곱셈 연산에 있어 문제 크기(n)와 프로세서(FU)의 수(k)를 동일하게 한 경우(k = n)에 실험에 사용된 VLIW 시뮬레이터 상에서서는 순차 실행에 비해서 평균적으로 2.18배 정도의 속도 향상을 보여주었다. 문제 크기보다 FU 수를 적게 할당한 경우(k < n)에도 1.99배의 속도 향상이 되었으며, FU를 8로 한 경우(Max(k))에도 2.19배의 속도 향상이 되는 것으로 나타났다. 또한 FU 할당 수를 가장 작게 한 경우(k=4)에도 1.84배 정도의 속도 향상이 되는 것으로 나타났다.



(그림3) 행렬-행렬 연산에서의 속도 향상(행렬-행렬)
(Fig.3) The Speedup in the Matrix-Matrix arithmetic

(그림3)에서 n이 커지고 할당된 FU 수(1≤k≤8)가 늘어날수록 실행 속도는 점차 증가되는 것을 알 수 있다.

【알고리즘2. Matrix-Vector Multiplication】

```
/* 1. 행렬 F(m x n), 벡터 X(n)의 입력 자료의 형
      태는 파일
   2. k : 프로세서 개수
      m : 행렬 F의 행 크기
      n : 행렬 F의 열 크기
   3. PE : 프로세서
   4. SUM : 각 프로세서의 연산 결과 누적 처리기
   5. t_part : 행렬 연산에 필요한 전체 분할 횟수
      c_part : 현재의 분할 수행 인덱스
      shift_cnt : 행렬 A의 각 요소에 대한 Shift 횟
                  수(= n-1)
   6. i, j : 행렬의 행, 열을 표시하는 인덱스     */

단계1.  declare constant k, m, n;
        /* 전체 분할 횟수 결정 */
        t_part = ⌈ m / k ⌉ ;
```

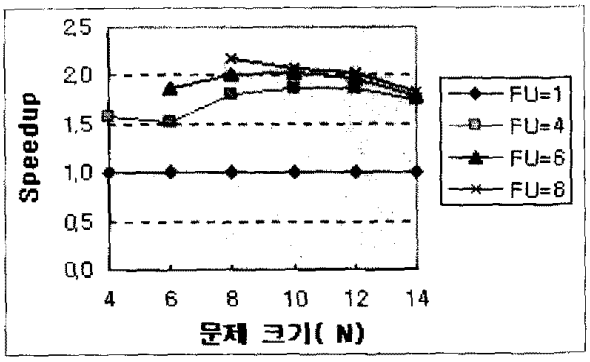
```
        c_part = 1;
        shif_cnt = 0;
단계2.  Read F, X
        /* 행렬 A, B의 입력 자료 읽기와 프로세
           서, 연산 누적 처리기의 초기화 */
단계3.  Initialize all PEs and Sums
단계4.  /* 행렬 연산 수행 */
        If c_part > t_part then goto 14;
단계5.  Allocate elements of X to each
        PEs with sequential(FIFO)
단계6.  Select columns of F to each PEs
단계7.  Select rows of F to each PEs
단계8.  Allocate elements of F to PEs
단계9.  Multiply each PEs
        Accumulate each results of PEs to Sums
단계10. Shift to right elements of X
        shift_cnt++
단계11. If ( shift_cnt <= n-1 ) then goto 단계6 ;
단계12. Initialize all PEs and Sums
단계13. c_part++;
        goto 단계4;
단계14. stop.
```

〈표 2〉 FU와 문제 크기 변화에 따른 속도
〈Table 2〉 The Speedup based on FU numbers and Problem sizes

FU 수 / N	순차 실행	VLIW 실행			
		k=n	k<n	Max(k)	Min(k)
4x4	1	1.59		1.59	1.59
6x6	1	1.86	1.51	1.86	1.51
8x8	1	2.18	1.92	2.18	1.81
10x10	1		2.03	2.16	1.86
12x12	1		2.01	2.14	1.87
14x14	1		1.84	1.93	1.75
16x16	1		1.63	1.63	1.63

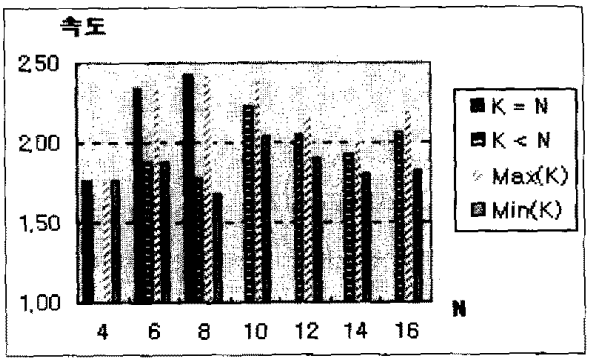
【알고리즘2】 에 의한 C 프로그램을 작성하여 순차 수행 및 VLIW 상에서 수행한 결과는 〈표2〉와 같다. 시뮬레이터에서의 실험 조건은 【알고리즘 2】 에서와 동일한 조건에서 수행하였으며, 실험 결과 속도 향상율도 비슷한 것으로 나타났다.

(그림4)에서 문제 크기(n)를 증가시키고, 시스템에 할당하는 FU의 크기를 증가할수록 처리 속도는 순차 실행에 비해 2배 정도의 처리 향상을 보임을 알 수 있다.



(그림 4) FU 증가에 따른 속도 변화(행렬-벡터)
(Fig. 4) The Speedup based on the FU numbers(matrix-vector)

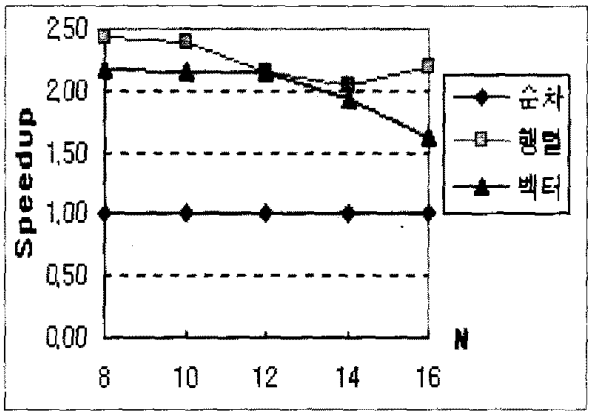
할당된 FU와 N에 대한 변화를 각각 다르게 하여 측정한 결과, (k = n)의 경우는 순차 실행에 비해서 평균적으로 1.88배 정도의 속도 향상을 보여주었다. 문제 크기보다 FU 수를 적게 할당한 경우(k 〈 n)에는 1.82배의 속도 향상이 되었으며, FU를 8로 한 경우(Max(k))에는 1.93배의 속도 향상이 되는 것으로 나타났다. 또한 FU 할당 수를 가장 작게 한 경우(k=4)에도 1.72배 정도의 속도 향상이 되는 것으로 나타났다. 실험 결과 행렬 연산과 벡터 연산에 대한 처리 속도의 차는 비슷한 것으로 나타났다. 따라서 FU 수와 문제 크기(n)에 따라 처리 속도가 어떻게 달라지는지 살펴본 결과 (그림5)와 같이 할당된 FU가 늘어날수록 실행 속도는 평균 2배 이상으로 증가되는 것을 알 수 있었다.



(그림 5) FU 수와 문제 크기에 따른 속도(행렬-행렬)
(Fig. 5) The Speedup based on the FU numbers and Problem Sizes(matrix-matrix)

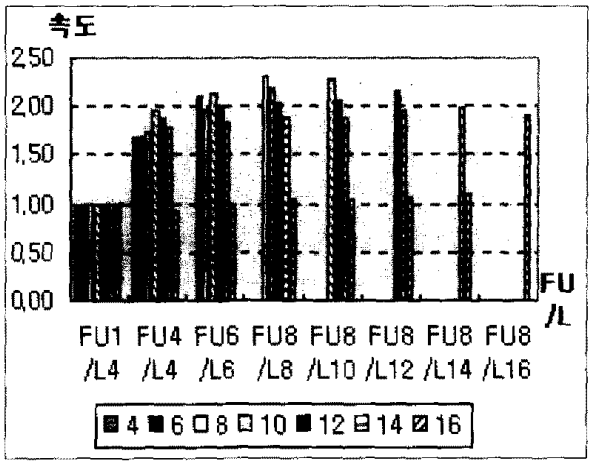
본 논문에서 제시한 두 알고리즘에 대한 실행 속도에 대한 분석을 하기 위해 해당 시뮬레이터 상에서 최

내 (FU=8, L=8)로 두고 n의 크기를 변화하여 그 처리 속도를 비교한 결과 (그림6)과 같이 행렬 연산이 벡터 연산보다 처리 속도가 조금 높은 것으로 나타났다. VLIW 상에서의 병렬 처리 속도는 할당된 FU와응용 프로그램 내의 반복처리문에서의 Loop 변수 범위와도 관련이 있다. 동일한 프로그램을 순차 실행에서는 FU=1, Loop 변수(L)의 범위를 4로 두고 실행하였고, VLIW 상에서 수행할 때는 FU와 L을 각각 다르게 하여 처리하여 보았다. 그 결과 두 가지 요소에 따른 평균 속도는 (그림7)과 같았다. 즉, FU를 최대 8로 하고, L의 범위를 증가할수록 처리 속도는 향상되는 것으로 나타났다.



(그림 6) 고정 크기(FU)에서의 문제 크기에 따른 실행 속도

(Fig. 6) The Speedup for the Problem Sizes on the fixed FU numbers



(그림 7) FU와 LOOP변수에 따른 평균 속도

(Fig. 7) The Average Speedup based on the FU numbers and Loop variables

4. 성능 및 비용 분석

4.1 선형 프로세서 배열

행렬 곱셈의 병렬 처리에 있어서 각 수행 단계에서의 PE 사용율이 낮을수록 결과 출력에 소요되는 시간이 증가하는 것은 명확하다. 제안된 알고리즘은 (n × n)의 행렬-행렬 곱셈이나 (m x n)의 행렬-벡터 연산에 있어서 순차 처리를 할 경우 시간 복잡도는 전체 $O(n^3)$, $O(mn^2)$번의 곱셈을 수행해야 한다. 이것을 k개의 PE를 사용해서 병렬 처리를 한다고 가정하면 $O(n^3/k)$, $O(mn^2/k)$번에 처리할 수 있으며, k=n 일 경우 $O(n^2)$, $O(mn)$의 시간 복잡도로 모든 행렬의 곱셈 처리를 할 수 있다. 또한 PE의 사용율이 100%이므로 병렬 처리의 효율성을 높일 수 있을 것으로 판단된다.

4.2 VLIW 시뮬레이터

제안된 기법에 따라 해당 프로그램을 VLIW용 시뮬레이터에서 Integer와 Floating pointer 프로세서 수(k)를 프로그램의 for문 상한 범위로 사용해서 동일한 프로그램을 각각 다른 수치로 실행하여 본 결과, 【알고리즘1】과 【알고리즘2】의 실행 명령어 사이클에 대한 분석은 약간의 비율 차이만 있을 뿐 동일하였다. 즉, 행렬-벡터 연산을 수행하는 Loop 변수의 범위를 증가함으로써 명령어 사이클 수는 감소되는 추세였으며, 연산 처리문에 대한 Loop 문의 상한첨자 범위를 문제 크기 n과 동일하게 할 경우 평균 43%의 최대 속도 향상을 보였다. 그러나 연산 처리문에 대한 Loop 문에서의 상한 첨자 범위를 문제 크기보다 작게 설정할 경우에는 평균 25% 정도의 속도 향상을 보였다. Code 내에 분기문이 없으면 원시 코드의 최적화와 스케줄이 용이해지므로 더 나은 처리 속도의 향상을 기할 수 있다.

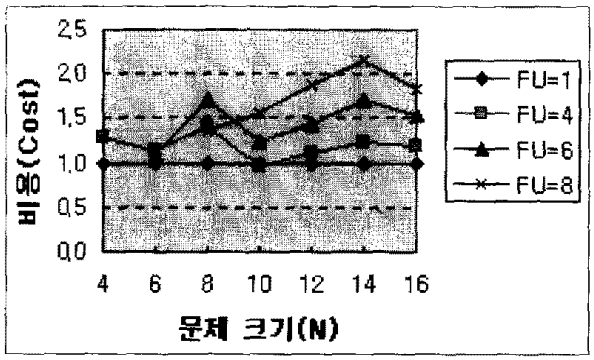
일반적인 병렬 수행 비용의 산정은 식(12)에 의한다.

비용 = PE 수 * 실행 시간^2 ····· (식 12)

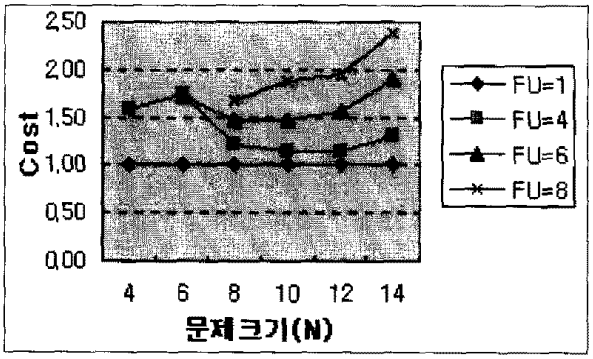
(식 12)에 의해 산출된 병렬 수행 비용은 일반적으로 순차 실행의 1.5 ~ 2배 정도에 이르는 것으로 알려져 있다. 비록 병렬 수행에 소요되는 비용이 높아도 H/W 제작에 의한 비용 절감 효과를 고려한 특수 목적

컴퓨터에서는 나름대로의 의의가 존재한다.

VLIW 시뮬레이터 상에서의 실험에 대한 【알고리즘1】과 【알고리즘2】에 대해서 프로세서를 증가시켜 실험한 경우 각각에 대한 비용 산출은 각각 (그림8), (그림9)와 같았다.

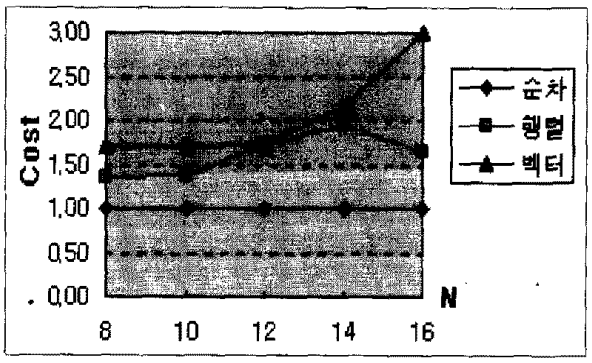


(그림 8) FU 증가에 따른 비용 변화 (행렬-행렬)
(Fig. 8) The Cost based on the FU numbers (Matrix-Matrix)



(그림 9) FU 증가에 따른 비용 변화(행렬-벡터)
(Fig. 9) The Cost based on the FU numbers (Matrix-Vector)

신호처리 목적의 DSP 시스템은 VLIW 시스템에 비하여 특정 처리에만 사용할 수 있어 범용성이 없다. 행렬 곱셈, FIR 필터, IIR 필터 등의 6가지에 대한 DSP 알고리즘에 대한 몇 가지 연구 결과가 발표되었다. [12]에서는 서로 다른 최적화 컴파일러를 사용한 결과 응용 프로그램의 최적화가 평균 1.81의 속도 증가를 보였고, 커널 수준에서의 실행 시간은 2.9배 정도의 속도 향상을 보이는 것으로 나타났다. [13]의 DSP 구조에서는 동일 문제에 대해 응용 프로그램 수준에서는 1.66배, 커널 수준에서는 2.87배의 성능 향상을 나타내었다. 이것은 기계어 구성과 입력 자료의 병렬화 가능한 알고리즘 구성에 따라 차이를 나타낼 수 있다. 또한 순차 실행과 VLIW 시뮬레이터 상에서의 【알고리즘1】, 【알고리즘2】에 대한 각각의 처리 비용에 대한 것을 그래프로 살펴보면 (그림10)과 같다.



(그림 10) 고정 크기(FU)에서의 문제 크기에 따른 비용
(Fig. 10) The Cost based on the Problem Sizes on the fixed FU numbers

순차 실행에 비해 VLIW 상에서의 수행이 행렬 연산에서는 평균 1.61, 벡터 연산에서는 2배 정도 높은 것으로 나타났다.

5. 결 론

임의의 (n × n)행렬-행렬 곱셈 또는 (m x n) 행렬-벡터 곱셈에 사용되는 많은 병렬 알고리즘들은 $n^2/2$에서 n^2에 이르는 PE를 선형 또는 직교, 육각 등의 접속을 통해 처리하고 있다. 그러나 n의 수가 커지면 하드웨어적인 구현에 문제가 발생될 수 있으며, 연산 처리 단계에서 각 PE에 idle time이 발생하여 전체 처리 시간이 증가된다. 이러한 문제점을 개선하기 위한 방법으로 프로세서 수를 k로 고정하였을 경우의 선형 접속 프로세서에 관한 연구가 필요하다. (n x n) 행렬 곱셈에서 N^2의 PE를 사용할 경우 3N-2의 실행 시간이 요구되었다[14]. (n x n) 행렬 곱셈을 위한 네트워크 구성별 프로세서 배열과 사용율에 대한 비교 연구가 발표되었다. Contraflow의 Mesh-array에서는 (2N-1)N의 배열로서 N/2(2N-1)의 사용율을 나타냈으며, undirectional flow의 Mesh-array에서는 (2N-1)N의 배열과 N/(2N-1)의 사용율이, undirectional flow의 Hex-array에서는 $(2N-1)^2$ 배열과 $N2/(2N-1)^2$의 사용율을 보이고 있었다[14]. 그러나 제안한 알고리즘을 선형 배열로 접속하였을 때 100%의 사용율을 보임으로써 전체 실행 속도를 개선할 수 있음을 보

였다.

본 논문에서는 행렬-행렬 곱셈 또는 행렬-벡터 곱셈을 대상으로 VLIW 컴퓨터와 선형 프로세서 배열상에서 동시에 실행 가능한 새로운 입력 자료에 대한 배치 알고리즘과 분할 기법을 제안하였다. 제안된 기법에 따르면 VLSI 구조나 시스톨릭 어레이에서 발생되는 PE의 낮은 효율성을 향상시킬 수 있음을 보였다. 또한 C로 작성된 행렬-행렬 곱셈과 행렬-벡터 곱셈에 대한 응용 프로그램을 VLIW 시뮬레이터에서 실행하여 순차처리와 비교하여 처리 속도가 2배 정도 향상됨을 보였다. 이에 대한 처리 비용도 평균 1.8배 정도 증가함을 알 수 있었다.

본 논문에서 제안한 행렬-행렬 및 행렬-벡터 연산을 VLIW 시뮬레이터에서 구현한 결과, VLIW 시스템의 성능은 시스템의 사용 유니트 수, 스케쥴링 방식 등 여러 가지 요인에 의존적이지만 제안된 기법에 의한 응용 프로그램에서는 순차 처리에 비해 평균 2배 정도의 처리 속도가 향상됨을 알 수 있었다.

그러나 동일한 프로그램에 대해서도 컴파일러의 최적화 방법, 스케쥴링 등에 따라 병렬 처리의 속도가 제한될 수 있다. 실험에서 사용한 시뮬레이터의 경우, 문제 크기 n > 16으로 증가할 경우에 컴파일러의 최적화 기법의 문제점으로 인하여 더 높은 성능 향상을 보이지 못했다. 그러나 알고리즘에서 병렬 처리할 수 있는 반복 처리문의 Loop 변수를 크게하면 할수록, 또한 할당하는 PE 수를 증가시킬수록 프로그램 내에 분기문이 없을 경우에는 그 처리 속도가 많은 향상을 가져올 수 있음을 알 수 있다.

참 고 문 헌

[1] Kai Hwang, "Advanced Computer Architecture : Parallelism, Scalability, Programmability", McGraw-Hill International Editions, 1993.

[2] H. T. Kung, Chales E. Leiserson, "Algorithms for VLSI Processor Arrays", symposium on Sparse Matrix Computations and Their Applications", 1978.

[3] F. Wichmann, "An Experimental Parallelizing Systolic Compiler for Regular Programs", IEEE, pp.92-99, Sept. 1993.

[4] A. Abnous, N. Bagherzadeh, "Pipelining and Bypassing in a VLIW Processor", IEEE Transaction on Parallel and distributed Systems, Vol.5, No.6, pp.658-664, June 1994.

[5] A. Capitanio, N. Dutt & A. Nicolau, "Partitioning of Variables for Multiple Register-File VLIW Architectures", Int. Conf. on Parallel Processing, pp.298-301, 1994.

[6] H. J. Whitehouse, J. M. Speiser, K, Bromley, "Signal Processing : Theory and Algorithms," Naval Ocean Systems Center, San Diego, California, pp.25-41.

[7] Hen-Geul YEH, "Systolic Implementation on Kalman Filters", IEEE Trans. on Acoustics, Speech, and Signal Processing, Vol.36., no.9, Sept. 1988.

[8] A. Andrews, "Parallel Processing of the Kalman Filter", Int. Conf. Parallel Process., Aug. 1981,pp.216-220.

[9] R. H. Travassos and A. Andrews, "VLSI Implementation of Parallel Kalman Filters," AIAA Guid. Cont. Conf., Advanced Avionics Session, San Diego, Aug, 1982.

[10] M. J. Chen & K.Yao, "On realization and implementation of Kalman filtering by systolic arrays," in Proc. 21st Conf. Inform. Sci., Syst.(Jones Hopkins University), Mar, 1987, pp.375-380

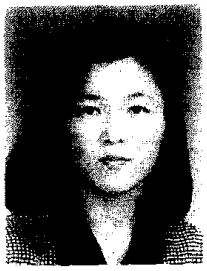
[11] F.M.F.Gaston and G.W.Irwin, "A systolic square root information kalman filter.", in Proc. Int. Conf. Systolic Array, New York : IEEE press, May 1988, pp.643-652

[12] Mazen A.R. Saghir, Paul Chow, Corinna G.Lee, "Application-Driven design of DSP Architectures and Compilers," in Proc. of the International Conference on Acoustics, and Signal Processing, pp.II-437 to II-440, April, 1994.

[13] Mazen, A. R. Saghir, Paul Chow, and Cori-nna G. Lee, "Towards Better DSP Archite-ctures and Compilers," in Proc. of the 5th International Conference on Signal Processing

Applications and Technology, pp.I-658 I664, Oct. 1994.

[14] Fabian Klass, Uri Weiser, "Efficient Systolic Array for Matrix Multiplication", Proc. of the Int. Conference on Parallel Processing,VolIII, pp.III-21-III215, Aug. 1991.



송 진 희

1988년 서울산업대학교 전산과(학사)

1990년 한국외국어대학교 교육대학원 전산과(석사)

1997년 숭실대학교 대학원 전산과 박사과정 수료

1992년~현재 신흥대학 전산정보처리과 조교수

관심분야 : 병렬알고리즘, 병렬 컴퓨터 구조, 계산기하학, 그래프 이론

전 문 석

1980년 숭실대학교 전산과(학사)

1986년 Univ. of Maryland 전산과(석사)

1988년 Univ. of Maryland 전산과(박사)

1989년 Morgan State Univ. 전산수학과 조교수

1991년 New Maxico State Univ. 부설 Physical Science Lab. 책임연구원

1991년~현재 숭실대학교 컴퓨터학부 부교수

관심분야 : 병렬 알고리즘, 병렬컴퓨터구조, 대규모 집적회로, 병렬처리 이론