저궤도 위성을 위한 HW 행렬 곱셈기의 구현과 성능 측정

이윤기[*], 김지훈[**]

HW Matrix Multiplier Implementation & Performance Measurement for Low Earth Orbit Satellite

Yunki Lee[*], Jihoon Kim[**]

요 약

지금까지 저궤도 위성의 자세제어 SW는 자세제어 연산을 위해서 CPU Resource로 있는 FPU를 사용하였으며, 이 결과 SW Throughput의 상당 부분이 행렬 곱셈 연산에 사용 되었다. 향후 위성에서 제어 주기가 더 짧아지고, 연산 량이 증가하면, 심각한 영향을 받을 수 있기 때문에 곱셈 전용 HW구현이 필요하게 되었다. 본 논문에서는 부동소수점 행렬 곱셈을 전용으로 수행하는 HW를 구현 및 성능 측정을 수행한 결과를 제시하며 추가적인 성능 향상을 위한 방법들과 향후 과제를 언급한다.

Key Words : Flight Software, Floating Point Unit, Hardware, Matrix, Multiplier

ABSTRACT

Until now, AOCS SW has used FPU which is one of CPU resources for satellite attitude control. And most of the SW Throughput was consumed to calculate Matrix Multiply. As SW throughput margin is decreasing seriously with shorter control period and more computational burden at next satellite programs, a dedicated HW matrix multiplier is absolutely required. This paper represents results of HW implementation & performance measurement and mentions several techniques for performance improvement, further works.

Ⅰ. 서 론

국내에서 개발한 저궤도 위성의 OBC에서 수행되는 자세제어 FSW (비행소프트웨어)는 센서들로부터 측정된 쿼터니언과 각속도를 바탕으로 EKF (Extended Kalman Filter) 연산 결과에 의존하여 위성의 자세제어를 수행하게 된다. 이때 기존 Atmel사의 ERC32 CPU Resource인 FPU (Floating Point Unit, FPU Queue가 없는 Meiko-FPU)를 사용한 FSW로 (without 운영체제) EKF 연산에 필요한 다양한 Matrix연산을 20MHz Board Clock으로 측정한 결과는 다음 표 1과 같다. (6 by 6) * (6 by 6)연산이 거의 1.3ms로 측정이 되었으며, SW의 반복 단위 시간 62.5ms안에서 46ms라는 꽤 많은 시간이 EKF의 행렬 곱셈 연산에 사용이 되었다. [1] 따라서 위성 성능향상을 위해서 SW계산에 의존하지 않고, Double Precision 기반의 Floating Point Matrix곱셈 전용 FPGA로직을 개발을 위한 타당성 연구가 진행되었다. [2]

표 1. 기존 FSW의 Matrix곱셈 연산 소모 시간

Executed Matrix Calculation	Time
(3 by 6) * (6 by 3)	323 usec
(6 by 3) * (3 by 3)	365 usec
(3 by 6) * (6 by 6)	635 usec
(6 by 6) * (6 by 3)	644 usec
(6 by 3) * (3 by 6)	688 usec
(6 by 6) * (6 by 6)	1300 usec

지금껏 HW를 Matrix곱셈기를 구현하는 연구는 대부분 Xilinx FPGA에서 구현을 하였으며, 주로 Parallel곱셈기를 구현하여 연산의 속도를 높이는 것에 많은 주안점을 두었다. [3~5] 하지만 위성 본체 컴퓨터에서 위성 자세제어에 사용되는 SW를 구동하는 경우에는 Xilinx FPGA들이 속도는 높으나, 우주방사능 내성이 약하여 Reliability가 낮아지는 단점이 있어서 주로 Microsemi의 Radiation Tolerant FPGA에 구현

*한국항공우주연구원 위성전자팀 (yklee@kari.re.kr,),
**충남대학교 전자공학과 (jihoonkim@cnu.ac.kr), 교신저자 : 김지훈.
접수일자 : 2015년 6월 1일, 수정완료일자 : 2015년 6월 24일, 최종게재확정일자 : 2015년 6월 29일

해야 하는 제약이 있다. 참고문헌 [5]는 Memory Load/Store, Move, Accumulate (덧셈/뺄셈), 곱셈, 곱셈-Accumulate 그리고 Zero Fill등의 Instruction Set Architecture를 구현함으로서 Matrix곱셈을 이용한 일반화된 구조를 제시하였다. 본 논문에서는 일반화된 곱셈기는 아니지만, 다양한 행렬 Size에 적용 가능한 HW곱셈기를 구현한 후, 성능 측정을 수행한 결과를 제시한다.

Ⅱ. HW행렬 곱셈기의 구현

아래 그림 1은 차세대 위성용 CPU로 선정된 Atmel사의 AT697F와 Memory Bus의 IO영역에 구현된 HW 행렬 곱셈기 (Microsemi사의 RTAX2000S FPGA에 구현) 의 연결을 보여준다.

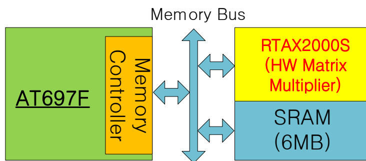


그림 1. HW행렬 곱셈기 구현 구조

1. 주요 구현 요구조건

표 2는 HW행렬 곱셈기 구현을 위한 주요 요구조건을 요약한 것으로서 RTAX2000S를 기준으로 약 50%이내의 조합회로를 사용하여 최대 (6 by 6) * (6 by 6) 연산을 기존 1.3ms에서 약 150us 이내로 줄이는 것을 주요 목표로 하며, 더불어 RAM Block에 대한 EDAC Protection, 부동소수점 연산의 오류 Detection등의 요구조건을 가진다. [6]

표 2. HW행렬 곱셈기 구현의 주요 요구조건

Requirements		Description
General	1	- (N by M) * (M by K) Double Precision Matrix Multiplier with Variable Size <N = 1~6, M = 1~6, K = 1~6>
Performance	2	- Less than 150us for (6 by 6) * (6 by 6) → Board Clock 80MHz → Multiplier Core Clock 10MHz
Resource	3	- Less than 50% Combinational Cell at RTAX2000S
RAM Block	4	- A, B, C Matrix RAM Block with BCH (64,8) EDAC Protection with Test Capa. → A, B Ram Write Protection during Core is busy.
Fault Handling	5	- IEEE-754 Double Precision Exception Detection with First Occurred Position. <+-QNAN, +-SNAN, Indeterminate, +-Infinity, +-Denormalized> - Double Bit Detection with First Occurred Position. - Multiplier Core & Its Controller Reset by Host for Fault Recovery

2. HW행렬 곱셈기 FPGA로직 설계

그림 2는 HW행렬 곱셈기의 FPGA 내부 로직 구조로서 A*B=C Matrix연산을 수행할 때, A, B, C를 저장할 EDAC RAM Block들과 곱셈기 Core로직으로 구성된 Write/Read Datapath와 2개의 Controller들로 구성된다.

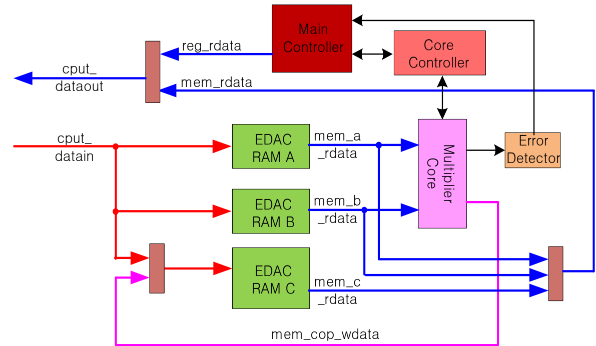


그림 2. HW행렬 곱셈기 구현 구조

실제 행렬 곱셈기 Core는 RTAX2000S의 용량 부족으로 병렬 곱셈을 구현하지 않았으며, VHDL-93에서 사용할 수 있는 "FloatFixLib" 참고문헌 [7]에 포함된 Double Precision 부동 소수점 곱셈기와 덧셈기를 1개씩만 사용하였다. 이에 따라 RAM Read, Input Latch, 곱셈, 덧셈으로 이어지는 4단 Pipeline으로 설계하였다. 다음 그림 3과 4는 각각 행렬 곱셈기 HW 및 Pipeline구조를 나타내고 있다.

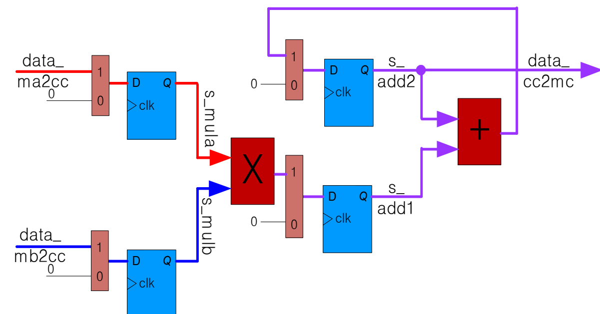


그림 3. HW행렬 곱셈기 Core 구조

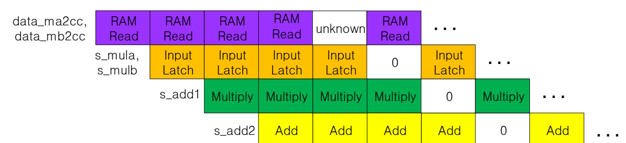


그림 4. HW행렬 곱셈기 Pipeline 구조

Core Controller는 다음 그림 5와 같이 I, J, K Counter를 두어서 SW가 설정한 값 까지 증가하며, 매번 I, J, K값으로 접근할 A, B, C Matrix의 주소를 계산하게 된다. 또한 I, J, K Counter는 차후 EDAC Error 발생, IEEE-754 Exception발생시에 발생한 주소 위치를 Generation하는데 사용이 된다.

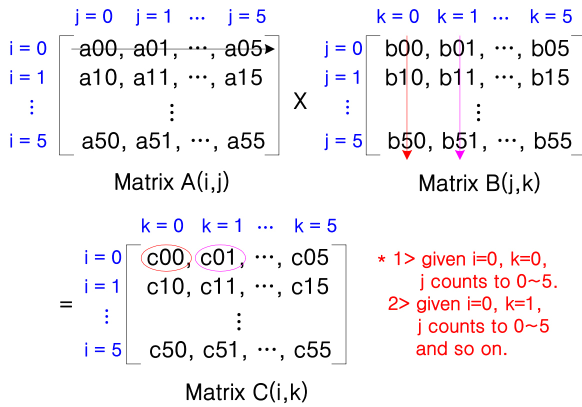


그림 5. Core Controller의 I, J, K Counter

A, B, C Matrix를 저장하는 공간은 FPGA 내부 RAM Block을 사용하였으며, EDAC Test Enable비트가 설정된 후에는 EDAC RAM은 Error를 Injection하는 Datapath를 만들어서 EDAC이 잘 동작하는지 Test할 수 있도록 구성하였다. 다음 그림 6은 EDAC RAM Block에 대한 설계 구조이다.

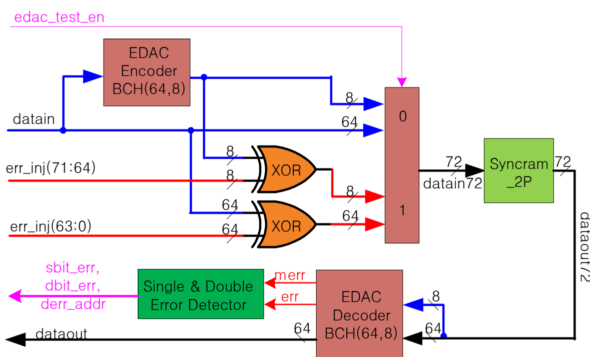


그림 6. 내부 EDAC RAM Block 구조 for A,B,C Matrix

3. HW행렬 곱셈기 구현 Simulation과 성능

(6 by 6) * (6 by 6)의 이상적인 결과를 얻기 위해서 A, B Writing에 소모되는 시간과 Core가 연산되는 시간, 그리고 C Reading에 소모되는 시간을 다음 그림 7과 같이 Simulation하여 얻을 수 있었다.

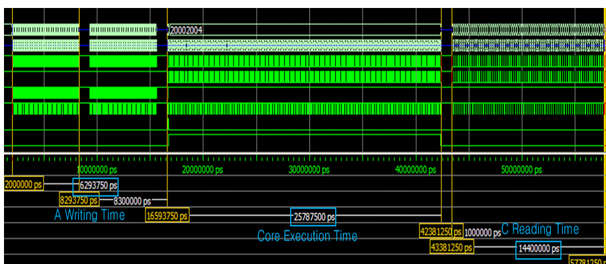


그림 7. HW행렬 곱셈기 이상적인 Simulation결과

☐ A 혹은 B Matrix 쓰기 시간 : 약 6.3us
☐ Core곱셈 연산 시간 : 약 25.7 us
☐ C Matrix 읽기 시간 : 약 14.4us

따라서 가장 이상적으로는 2번의 Writing, 연산, 1번의 Reading을 한다면, 2*6.3us + 25.7us + 14.4us = 52.7us이내에서 완료 될 수 있을 것으로 보인다.

4. HW행렬 곱셈기 구현 결과

HW행렬 곱셈기는 RTAX2000S, CQFP256, Speed -1 Grade FPGA를 대상으로 Synplify Pro F-2012.03A- SP1-2 합성기로 합성하였으며, Actel Designer 9.1.5.1 SP5로 Military온도, 300Krad 조건으로 Place&Routing을 수행하였다. P&R후 용량은 조합회로가 약 57%, 순차회로가 약 10%을 소비하였으며, RAM Block 약 9%를 사용하였다. STA (Static Timing Analysis) 결과는 80MHz Clock Path에 대해서 약 73.153MHz 결과가 나왔으나, -Slack Path 22개 부분이 모두 2 Clock Period인 Multi-Cycle Path이므로 문제되지 않으며, 10MHz Clock Path에 대해서는 12.520MHz로 +Slack을 가지게 되었다.

5. 연산 결과 비교를 위한 GUI-SW

HW행렬 곱셈기는 여러 가지 오류 상황과 연산 결과 정확성을 위해서 다음 그림 8과 같이 비교 대상이 되는 GUI SW를 만들었으며, 여러 가지 경우에 대한 연산결과가 GUI SW와 일치함을 확인하였다.

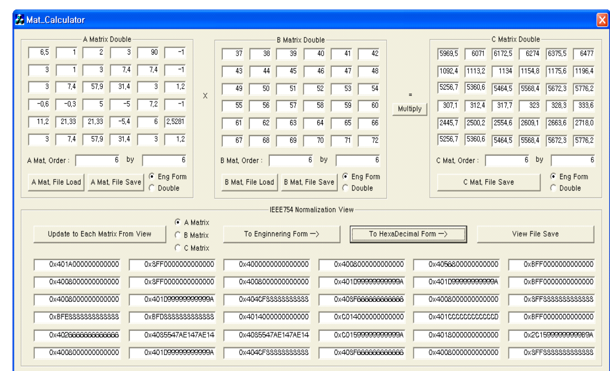


그림 8. HW행렬 곱셈기 성능 비교 위한 GUI-SW

Ⅲ. HW행렬 곱셈기의 성능 측정

HW행렬 곱셈기의 성능측정은 그림 9와 같은 AT697F가 탑재된 검증 보드에서 수행되었으며, 80MHz Clock, SRAM Wait Cycle은 Write 0, Read 3, No SW Compile Optimization (Compiler : BCC 4.4.2 <1.0.36c>)으로 수행되었다.

그림 9. HW행렬 곱셈기 성능 측정 보드

참고로 Atmel사의 AT697F는 1단 Cache만 가지는 간단한 32Bit SPARC V8계열 CPU로서 I-Cache가 4-Way Associative로 구성된 32KB공간을 가지며, 하나의 Cache Line는 8개의 32Bit 명령어를 가진다. D-Cache는 2-Way Associative로 구성된 16KB공간을 가지며, 하나의 Cache Line은 4개의 32Bit Data를 저장할 수 있다.

1. HW곱셈기와 SW계산의 비교

실제 EKF필터에서 쓰이는 곱셈 행렬 Size 6가지에 대해서 ERC32 및 AT697F FPU를 활용한 운영체제 없는 SW수행의 경우와 HW 행렬 곱셈기를 호출한 경우에 대한 성능 측정 결과는 다음 그림 10과 같다.

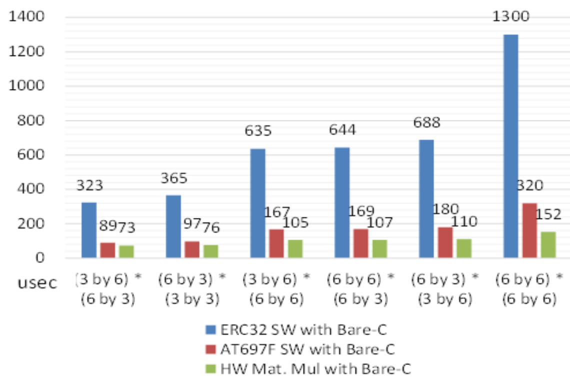


그림 10. ERC32 (SW), AT697F (SW), AT697F (HW) 경우에 대한 성능 측정 결과

측정 결과를 요약하면, (6 by 6) * (6 by 6)계산에 대해서 SW 320us, HW 152us로 약 2배정도 밖에 빠르지 않았다. 먼저 AT697F의 FPU를 이용한 단순 SW계산 시간이 ERC32 FPU이용한 경우보다 약 4배가량 작게 나오며, 이는 ERC32 (20MHz)가 AT697F (80MHz)보다 4배의 느린 Clock을 사용하였기 때문으로 보인다. 둘째는 HW행렬 곱셈기를 사용했을 경우에 (6 by 6) * (6 by 6)인 경우가 약 152us로서 실제 A 혹은 B Writing과 C Reading, Core연산을 측정한 결과 다음과 같이 A, B Writing, C Reading에서 이상적인 Simulation보다 훨씬 큰 값이 나오는 것을 확인할 수 있었다.

- ☐ A 혹은 B Matrix 쓰기 시간 : 약 41 us (> 6.3 us)
- ☐ Core곱셈 연산 시간 : 약 27 us (≅ 25.7 us)
- ☐ C Matrix 읽기 시간 : 약 44 us (> 14.4 us)

2. SW 코드의 최적화

한편 SW Compile을 Option 2로 최적화 시켜서 돌리면, A, B Writing과 C Reading 시간이 꽤 줄어드는 것을 확인할 수 있으며, 이에 따라서 A혹은 B Writing Code는 그림 11과 같이 C코드를 강제로 최적화된 어셈블리어 코드로 작성하고, C Reading Code는 그림 12와 같이 C코드를 최적화된 어셈블리어 코드로 작성하여 전체 SW는 역시 No Compile Optimization으로 수행하였다.

```
test_addr = (UADDR *)a;
for(i=0; i<72;i++)
    io_mat_ram_a_addr[i] = *(test_addr+i);
```

```
asm volatile("         sethi  %hi(io_mat_ram_a_addr), %l1");
asm volatile("         ld [%l1 + %lo(io_mat_ram_a_addr)], %l1");
asm volatile("         clr %l3");
asm volatile("jumpa :  ld [%i0 + %l3], %l4");
asm volatile("         st  %l4, [%l1 + %l3]");
asm volatile("         add %l3, 4, %l3");
asm volatile("         cmp %l3, 0x120");
asm volatile("         bne,a   jumpa");
asm volatile("         nop");
```

그림 11. A 혹은 B Matrix의 IO Area Writing코드의 최적화

```
test_addr = (UADDR *)c;
for(i=0; i<72;i++)
    *(test_addr+i) = io_mat_ram_c_addr[i];
```

```
asm volatile("         sethi  %hi(io_mat_ram_c_addr), %l1");
asm volatile("         ld [%l1 + %lo(io_mat_ram_c_addr)], %l4");
asm volatile("         clr %l1");
asm volatile("jumpc :  ld  [%l4 + %l1], %l3");
asm volatile("         st  %l3, [%i5 + %l1]");
asm volatile("         add %l1, 4, %l1");
asm volatile("         cmp %l1, 0x120");
asm volatile("         bne,a   jumpc");
asm volatile("         nop");
```

그림 12. C Matrix의 IO Area Reading코드의 최적화

그 결과, (6 by 6) * (6 by 6)인 경우가 약 152us이던 것이 87us로 줄어들어서 SW Compiler의 성능이 매우 좋지 않음을 확인했다. 다음은 SW코드 일부 최적화 후 각 단계 시간을 측정한 것이며, 아직도 A, B Writing, C Reading에서 이상적인 Simulation보다 훨씬 큰 값이 나오는 것을 확인할 수 있었다.

- ☐ A 혹은 B Matrix 쓰기 시간 : 약 17 us (> 6.3 us)
- ☐ Core곱셈 연산 시간 : 약 27 us (≅ 25.7 us)
- ☐ C Matrix 읽기 시간 : 약 25 us (> 14.4 us)

3. Data Cache Pre-Loading

위성 자세 제어를 위한 EKF로직은 기본적으로 앞에서 계산된 A, B Matrix값을 행렬 곱셈의 입력으로 사용하여 HW 행렬 곱셈기를 위해 IO Area에 Writing이 되는데, 이때 계산

된 값은 Data Cache에 미리 로드되어 있을지 여부가 불확실하다. (대부분의 경우는 앞의 행렬 계산 결과가 A, B로 다시 입력되므로, Cache에 Load되어있다.) 만약 모든 A, B Matrix값이 예전에 사용된 적이 있어서 모두 Read Hit가 된다면, SRAM Reading의 시간이 없어질 수 있어서 가장 빠른 성능을 낼 수 있다. 따라서 Data Cache에 이미 올라온 상황을 모사하기 위해서 미리 IO Area에 적을 Data값을 모두 읽어본 후 측정한 결과 71us로 줄어드는 것을 확인하였다. 다음은 각 단계별 측정 값으로서 이제 A와 B Matrix의 Writing시간은 거의 이상적인 Simulation값에 근접함을 알 수 있으며, C Matrix는 읽은 후에 AT697F Data Cache의 Write Through정책으로 인해서 Cache와 SRAM에 Writing하는 과정 때문에 시간 소요가 Simulation결과 보다 거의 2배 이상 걸린 것으로 예상이 된다.

- □ A 혹은 B Matrix 쓰기 시간 : 약 9 us (> 6.3 us)
- □ Core곱셈 연산 시간 : 약 27 us (≅ 25.7 us)
- □ C Matrix 읽기 시간 : 약 25 us (> 14.4 us)

4. Element Based C Matrix Reading

지금까지 구현한 FPGA로직은 A와 B Matrix를 HW행렬 곱셈기로 입력한 후에 연산이 끝났다는 Register값을 Polling하고 있다가 C Matrix값을 읽는 방식이었다. 하지만 FPGA에서 A*B의 일부 계산된 요소 값이 나올 때 마다 AT697F가 읽을 수 있도록 해준다면, (AT697F의 IO Area는 IO Bus Ready방식을 활용하여 계산이 완료되지 않으면 No Ready Assertion하도록 FPA구현) Core를 계산하는 시간 27us안에 C Matrix Reading을 모두 수행할 수 있다. 따라서 FPGA로직을 요소단위로 읽을 수 있도록 변경하였고 SW는 A, B Matrix Writing이 끝난 후부터 계속해서 C Matrix Reading을 수행한 결과 최종 46us로 줄어들었다. 다음은 각 단계별 시간이다.

- □ A 혹은 B Matrix 쓰기 시간 : 약 9 us (> 6.3 us)
- □ Core곱셈 연산 및 C Matrix Reading시간 : 약 28 us (≅ 25.7 us)

5. 강화된 HW곱셈기와 SW계산의 비교

앞의 일부 SW코드의 최적화와 Data Cache Pre-Loading은 Best 경우 상황으로서 Simulation결과와의 일치 여부를 보기위한 것이었으며, 실제는 FPU를 이용한 SW도 같은 조건하에서는 시간이 많이 줄어들 수 있다. 따라서 요소 기반의 C Matrix Reading으로 바뀐 FPGA로직을 바탕으로 ERC32 SW Case를 제외하고 각 행렬 Size에 대해서 측정 결과는 다음 그림 13과 같다.

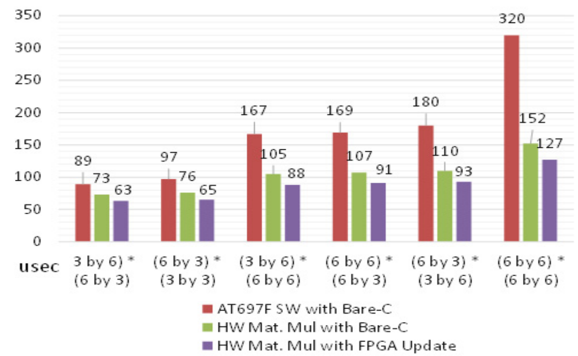


그림 13. AT697F (SW), AT697F (HW), AT697F (HW Update) 경우에 대한 성능 측정 결과

Ⅳ. 향후 계획

측정 결과 320us/127us = 약 2.5배 SW계산보다 빨라졌지만 FPGA 조합회로 용량을 약 50%를 사용하여 구현된 것에 비하면 월등한 성능향상이 없었다. 따라서 향후에는 좀 더 일반화된 HW행렬 연산기를 개발할 필요성이 있는데 그 이유는 대부분의 시간을 IO Write/Read에 소비되기 때문에 연속된 행렬 연산 (+, -, x, Transpose, Move)을 FPGA내부에서 연속으로 처리할 수 있다면, 전체 응용 행렬 연산 성능에 큰 향상을 가져올 수 있다. 다음은 실제로 EKF함수에서 Prior Covariance를 계산하는 함수의 Flow를 예제로 보여준다.

① State Transition Matrix A(k)와 과거 계산된 Covariance Matrix P(k-1<최종>)를 행렬곱하기.
(<6 by 6> * <6 by 6>)
수식 1 : { temp1 = A(k) * P(k-1<최종>) }

② State Transition Matrix A(k)를 행렬 Transpose.
(Transpose of <6 by 6>)
수식 2 : { temp2 = Transpose of A(k) }

③ ①의 결과와 ②의 결과를 행렬 곱하기.
(<6 by 6> * <6 by 6>)
수식 3 : { temp3 = temp1*temp2 }

④ ③의 결과와 상수 Process Noise Matrix Q를 더하여 현 예측 Covariance Matrix구하기.
(<6 by 6> + <6 by 6>)
수식 4 : { P(k<예측>) = temp3 + Q }

즉, 이러한 연속된 행렬 연산 동작에서 ④를 제외한 ① ~ ③의 과정은 다음 표 3과 같이 HW와 SW에 의해서 동작 분담을 한다면 IO Write/Read를 줄임으로서 FPU SW로만 계산한 것 보다 높은 성능향상을 가져올 수 있다.

표 3. 예측 Covariance 계산의 ① ~ ③ HW/SW동작

Step	Formula	Sub-Step	Operation
1	temp1 = A1(A<k>) X B1(P<k-1최종>)	1	SW Write A1 to IO
		2	SW Write B1 to IO
		3	HW Multiplies A1 & B1 and Store to C1 (temp1)
2	temp2 = Transpose of A(k)	1	HW Transpose A1 to B1 (temp2)
3	temp3 = temp1*temp2	1	HW Copies C1 to A1 (temp1)
		2	HW Multiplies A1 & B1 and Store to C1 (temp3)
		3	SW Reads C1 from IO

V. 결 론

본 논문에서는 차세대 저궤도 위성에 사용하기 위한 AT697F CPU의 FPU를 이용하여 Double Precision부동 소수점 Matrix연산을 사용할 경우에 대한 FSW Throughput 부족 문제를 해결하기 위해서 HW 행렬 곱셈기를 FPGA에 구현하였고, 그 성능을 측정하였다. 측정결과 실제 AT697F FPU를 이용한 SW계산 보다 약 2.5배 가량 빨라진 것을 확인할 수 있었다. 또 실제 HW곱셈과 덧셈의 연산 시간보다 더 중요한 것이 일부 SW Code의 Optimization으로 IO Write, IO Read Time을 줄이는 것이며, SW를 Cache Miss가 많이 나지 않도록 Coding하는 것이 필요함을 파악하였다. 본 연구 결과를 바탕으로 일반화된 HW행렬 연산기를 구현하는 것을 향후 과제로 파악되고 있다.

참 고 문 헌

[1] "LEO ACS FSW Throughput Reexamination“, 2008.10.12, KARI IOC

[2] 이윤기, 김지훈, “Double-Precision기반의 HW Matrix곱셈기 구현에 관한 타당성 연구”, 춘계 항공우주 학술대회, 2014, pp. 837-840.

[3] Ju-Wook Jang, Seonil B. Choi and Viktor K. Prasanna. "Energy- and Time-Efficient Matrix Multiplication on FPGAs'' IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 13, no. 11, pp. 1305 - 1319, 2005.

[4] Shivangi Tiwari and Nitin Meena "Efficient Hardware Design for Implementation of Matrix Multiplication by using PPI-SO'' International Journal of Innovative Research in Computer and Communication Engineering, vol. 1, Issue 4, pp. 1020 - 1024, 2013.

[5] Nirav Dave, Kermin Fleming, Myron King, Michael Pellauer, Muralidaran Vijayaraghavan. "Hardware Acceleration of Matrix Multiplication on a Xilinx FPGA'' MEMOCODE 2007. 5th IEEE/ACM International Conference. pp. 97-100

[6] David Bishop,“Floating Point Package User's Guide”, http://www.vhdl.org/fphdl/vhdl.html,

[7] “IEEE-754 Standard”, 1985, IEEE

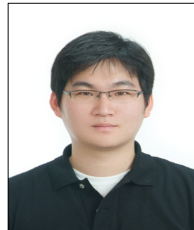
저자

이 윤 기(Yunki Lee) **정회원**

・2003년 2월 : 경북대학교 전기전자공학 학사졸업
・2005년 2월 : 한국과학기술원 전자공학 석사졸업
・2005년 3월 ~ 현재 : 한국항공우주연구원

<관심분야> : 위성컴퓨터, 위성시스템

김 지 훈(Ji-Hoon Kim)



・2004년 2월 : KAIST 전자전산학 학사졸업
・2009년 8월 : KAIST 전기 및 전자공학 박사졸업
・2009년 7월 ~ 2010년 2월 : 삼성전자 책임연구원
・2010년 3월 ~ 현재 : 충남대학교 전자공학과 부교수

<관심분야> : 디지털시스템, System on Chip