**Tiny TPU Custom RTL Design**

1. **개요 및 요약**

**Google TPU(In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 17)를 참고하여 CNN, DNN 연산에 기반 되는 General Matrix-Matrix Multiplication 연산에 대한 효율을 높일 수 있는 아키텍처를 설계한다.**

**해당 아키텍처를 RTL Design으로 설계하고 아키텍처에 대한 적절한 Custom Instruction Set을 할당한다.**

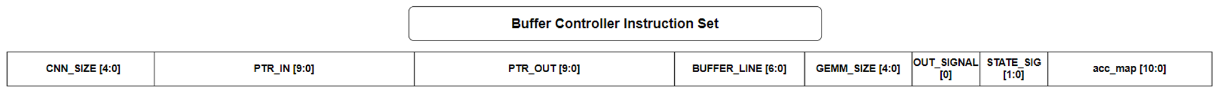
**Instruction Sequence를 설정하여 CNN, DNN 연산을 수행하고 시뮬레이션 결과를 살펴본다.**

스케치, 디자인, 직사각형, 예술이(가) 표시된 사진

자동 생성된 설명텍스트, 도표, 스크린샷, 평면도이(가) 표시된 사진

자동 생성된 설명

**Fig 1. Custom TINY TPU RTL Design & Block Diagram**



**Fig 2. Custom Instruction Set for Buffer Controller**

**목차**

1. **Google TPU Architecture**
2. **Tiny TPU Architecture & RTL Design**
3. **Simulation**
4. **Google TPU Architecture**
5. **개요 및 설명**

**TPU Architecture는 현대 인공지능 모델에 자주 사용되는 CNN, DNN 등 인공지능 모델 연산의 효율을 높이기 위해 개발된 구조이다.**

**전통적인 컴퓨터 구조인 Von Neumann의 Computer Architecture를 벗어나 연산에 따른 속도 및 전력량 등 전반적인 성능을 높이는 구조라 할 수 있다.**

**기존 컴퓨터 구조에서 연산을 하는 주체는 CPU이고, 데이터를 저장하는 곳은 DRAM이라 하면, 데이터의 흐름은 CPU-> DRAM -> CPU -> DRAM … 이런 식의 Data Flow를 나타낼 것이다.**

**하지만 이러한 구조에서는 연산에 따른 반복적 DRAM (혹은 Cache)에 대한 접근이 필수적이며, 많은 양의 데이터를 한 번에 처리할 때는 해당 구조에서 많은 DRAM ACCESS를 유발 할 것이다.**

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**Fig1. Computation 과 Data Access에 대한 전력량 비교**

**Fig 1을 살펴보면 Data Access 관점에서 기존 연산과 비교하면 엄청난 전력량을 소모하는 것을 알 수 있다. 이러한 관점에서 Data Access를 줄이면 상당한 전력량 이득을 차지할 수 있다.**

**실제 Memory Access는 많은 오버헤드를 유발한다. 먼저 앞서 말한 전력량 또한 해당 오버헤드들로 인해 영향을 받는다.**

**실제 CPU에서 L1Cache에 대해 Data Fault가 발생하게 되면 L2Cache, L3Cache, 나아가 DRAM까지 접근하게 될 것이다. 이러한 순차적 Cache내 데이터를 읽는 과정과 DRAM내 Row Buffer를 open 하는 과정까지, 많은 오버헤드를 발생시킨다고 할 수 있다.**

**하지만 일반적으로 인공지능 모델의 연산에서는 데이터 재사용이 자주 일어나게 된다.**

**특히 CNN 알고리즘을 살펴보면 해당 알고리즘에서는 Weight들이 서로 공유되어 input Feature maps에 계산되게 된다.**

**그렇다면 Weight들은 재사용되는 데이터이며, 또한 Input Feature Maps 또한 연속적으로 다른 Weight(Filters)들과 연산이 되어야 하기에, 이 또한 재사용 데이터라 할 수 있다.**

1. **TPU 구조 설명**

텍스트, 도표, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Fig 2. Google TPU Architecture Block Diagram**

**Fig 2을 살펴보면 Google TPU의 대략적 구조를 살펴볼 수 있다. 여기서 중요한 것이 실제 CPU와 TPU는 서로 다르다는 것이다. TPU는 DRAM과 같은 외부장치라 생각해야 한다.**

**즉, GPU가 기존 컴퓨터 구조에서 외부 장치에 해당되어 GPU 관련 Device Driver를 설치해야 하는 것처럼, TPU 구조 또한 GPU와 유사하게 외부 장치 취급을 받으며 PCIe Interface를 통해 DRAM으로부터 데이터를 받거나 TPU내 Control Signals를 받게 된다고 할 수 있다.**

**즉, 인공지능 모델 추론같이 연산 량이 매우 방대한 경우, CPU에서 연산을 처리하지 않고 외부장치로 데이터를 보내 연산을 진행시키는 것이다.**

**이러한 개념을 기반으로 설명하자면, DRAM으로부터 온 데이터는 Local buffer로 저장된다.**

**해당 buffer내 데이터는 TPU내 Instruction에 따른 Control에 따라 Matrix Multiply Unit으로 들어가게 된다. 해당 MMU(Matrix Multiply Unit)내 존재하는 PE Array들에 데이터가 들어가며 연산이 이뤄진다. (PE Array는 Processing Element Array이다.)**

도표, 텍스트, 스크린샷, 평면도이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

**Fig 3. PE Array 확대 & PE 구조**

**여기서 중요한 부분은 한번 연산 된 결과가 바로 buffer에 저장되지 않고 PE의 결과물이 다른 PE의 피연산자로 들어가거나 PE내 피연산자들이 공유된다는 점이다.**

**이러한 방식을 통해 데이터 재활용을 최대화 시켜주는 것이 PE Array, 즉 MMU의 장점이라 할 수 있다.**

텍스트, 도표, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**Fig 4. Systolic array & PE array 내 연산 과정**

**Fig 4을 살펴보면 앞서 말한 것과 같이 하나의 PE내 피연산자들이 다른 PE의 피연산자로 사용되게 되는 것을 확인할 수 있다.**

**또한 이러한 데이터 재활용 구조는 어떤 데이터를 재활용하는 가에 따라 Data Flow가 달라지게 된다.**

**위 Fig 4의 PE array 구조에서는 Output-stationary dataflow구조라 되어 있다.**

**해당 구조에서는 PE내 연산 된 결과물이 해당 PE내 존재하는 것을 의미한다.**

**연산의 결과는 PE 내 존재하는 어느 정도의 크기를 가진 Register에 임시적으로 저장되며, 연산이 종료되면 PE Array를 타고 다음 연산(Activation function)을 하거나 Data Buffer로 돌아가게 된다.**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**Fig 5. Type of Data Flow based on Data Reuse**

**Fig 5을 보면 어떤 데이터를 재사용하는 가에 따라 데이터 플로우가 정해지는 것을 확인할 수 있다. Weight Stationary에서는 인공지능 모델내 Weight를 재사용하는 방식이다. 해당 방식은 PE내 존재하는 Register에 weight값을 저장하고 들어오는 피연산자와 weight를 연산 시켜준다.**

**그리고 연산의 결과물을 다음 PE 로 보내주게 된다. 이러한 방식은 DNN 및 CNN구조 둘 다 적용 가능하다.**

텍스트, 스크린샷, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

**Fig 6. CNN 연산을 위한 Transformed Filters and Transformed Input features**

**결국 CNN, DNN 모두 Matrix Multiplication의 형태로 연산이 되어야 한다. CNN 연산을 직관적으로 살펴보면 Filter와 해당 Filter내 존재하는 데이터의 연산으로 적은 양의 메모리를 소모하며 연산 되는 것처럼 보이지만 실제 Matrix Multiplication 형태로 연산이 된다.**

**이를 참조하여 해당 프로젝트의 Tiny TPU는 Output stationary 기반으로 CNN, DNN, GEMM에 대한 Matrix Multiplication을 수행하는 Processing Unit 제작을 목표로 하였다.**

1. **Tiny TPU Architecture & RTL-Design**

**위에서 설명한 Output stationary 방식을 기반으로 Systolic array로 배열된 PE Array를 통해 CNN, DNN 등 연산에 대한 Matrix Multiplication을 진행하는 Processing Unit을 설계한다.**

**해당 유닛에서는 Buffer Controller, Instruction Buffer, Data buffer, Matrix Multiplication Unit, Activation Unit이 존재한다.**

텍스트, 도표, 스크린샷, 평면도이(가) 표시된 사진

자동 생성된 설명

**Fig 7. Tiny TPU Custom RTL Block Diagram**

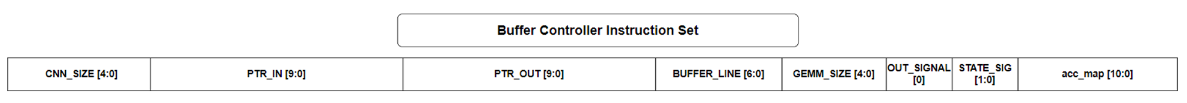
**해당 구조에서 Buffer Instruction File은 TPU 구조내 PCIe Interface로 들어오게 될 MMU에 대한 Control Signal들의 집합이다.**

**Buffer Controller는 전체 하드웨어 컨트롤에 관장하며, Buffer Instruction내 존재하는 Control Signals을 기반으로 data buffer내 신호를 보내 MMU내 연산이 이뤄나게 한다.**

**MMU는 내부에 16x16의 Processing Element 유닛들이 존재하게 된다. 해당 유닛들은 Systolic Array 형식으로 배치되어 있으며 buffer를 통해 들어온 데이터를 Output Stationary기반으로 연산하게 된다.**

**Activation Unit에서는 Relu 연산이 이뤄나게 된다. CNN 및 DNN 알고리즘 구조에서는 연산 결과에 Activation Function을 사용하는 것이 일반적이다. 이는 인공지능 모델의 계층이 깊어지며 계층 간에 비선형 함수의 연산을 넣어줘 더 깊은 층을 쌓기 위함이다.**

**이를 구현하기 위해 Activation 단을 설치해주었다.**



**Fig 8. Custom Buffer Control Instruction Set**

**이렇게 5가지의 하드웨어로 이루어져 있으며, 각각의 하드웨어는 Fig 8의 Instruction내 Signal에 따라 유기적으로 동작한다.**

**CNN\_SIZE는 Output feature map의 Row 개수 – Col 개수에 해당한다.**

**Ptr\_in은 Data Buffer 내 존재하는 Row\_Data의 인덱스 위치 값을 의미한다.**

**Ptr\_out은 Data Buffer 내 존재하는 Row\_Data의 마지막 인덱스 위치 값을 의미한다.**

**Buffer\_line 은 Row\_Data 시작점과 Col\_Data 시작점의 인덱스 위치 차이 값을 의미한다.**

**GEMM\_SIZE는 Output feature maps의 ROW 개수를 나타낸다.**

**Out\_signal은 MMU 연산을 진행하라는 신호이다.**

**State\_sig는 MMU내 연산 종류를 나타내며 현재 프로젝트에서는 Output stationary 기반으로 연산이 진행되기에 모두 00에 해당한다.**

**Acc\_map: acc\_map은 MMU 및 Activation 연산 결과를 Buffer내 어느 인덱스부터 저장할 지 결정하는 신호라 할 수 있다.**

**이러한 신호 8개를 통하여 전체 하드웨어 5가지 요소를 제어하게 된다.**

1. **Instruction Buffer**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Fig 9. Instruction Buffer RTL Design**

**해당 구조에서는 Buffer Controller 내 존재하는 Address 신호에 따라 Instruction Buffer 내 Instruction을 가져오는 식으로 진행하였다.**

**Instruction Buffer는 다른 장치 (DRAM)에서 들어오는 Instruction 정보를 Buffer(Queue)형식으로 저장하고 Buffer Controller 내 존재하는 Address 신호에 따라 명령어를 전달해주는 역할 하게 된다. 이렇게 함으로써, 먼저 들어온 명령어가 먼저 호출되게 할 수 있으며 MMU 및 Act내 연산 과정을 기다릴 수 있게 된다.**

**Instruction의 길이는 51 bits에 해당하며 이 51bits의 명령어를 101개의 Queue Array내 저장할 수 있다.**

**해당 프로젝트 내에서는 Dram 같은 장치를 따로 구현하지 않았기에 미리 설정해둔 Instruction File을 불러오는 식으로 진행하였다.**

1. **Buffer Controller**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Fig. 10. Buffer Controller RTL Design**

**해당 구조에서는 Data buffer에서 들어오는 Finish-Sign을 토대로 다음 Instruction Fetch 과정을 거치게 된다.**

**Instruction Fetch가 완료되면 Instruction 내 존재하는 신호들을 분류하여 Data\_buffer로 전달하는 과정을 가진다.**

**Output으로 설정된 Signal들은 data\_buffer로 전달되며 이는 앞서 말한 Instruction Set내 존재하는 신호들에 해당한다.**

1. **Data Buffer**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Fig 11. Data Buffer**

**해당 구조에서는 MMU 및 Activation Module에서의 주고받는 데이터를 관리하는 모듈이다. 뿐만 아니라 Instruction에 따른 결과 신호들에 따라 어디까지 데이터를 보내고 받는지 결정하게 된다.**

**Clk: clock에 맞춰 data의 입력 및 출력이 진행된다.**

**Rst: buffer내 존재하는 data를 0으로 초기화 시키거나 미리 설정해둔 Data.txt file을 읽는 과정을 진행한다.**

**Act\_in\_sig: 해당 신호는 activation에서 출력되는 데이터를 받을 때, 해당 받는 데이터들이 Valid한지 확인해주는 신호이다.**

**해당 신호가 16개 존재하는데 이는 MMU내 존재하는 16x16 PE array의 row 개수와 동일하다.**

**Data\_in: 해당 신호는 activation에서 출력되는 데이터를 의미한다. 해당 데이터의 Valid/InValid는 act\_in\_sig에 따라 결정된다.**

**Col, row: 해당 신호는 MMU로 보내는 data들에 해당하게 된다. 해당 신호들을 통해 MMU는 데이터를 받고 연산을 진행하게 된다.**

**Instruction에 따른 signals: 해당 신호들은 MMU내에서 연산을 어느 정도 하고 어디부터 Valid한 신호인지 의미하게 된다.**

**또한 Output Stationary로 진행할지 Weight Stationary로 진행할지 결정되게 된다.**

1. **MMU (Matrix Mulitplication Unit)**

텍스트, 스크린샷, 폰트, 대수학이(가) 표시된 사진

자동 생성된 설명

**Fig 12. Matrix Multiplication Unit**

**해당 유닛은 Data\_buffer로부터 받은 데이터 row, col에 대한 연산을 진행한다.**

**연산된 결과는 Valid 신호와 함께 Activation Module로 보내게 된다.**

**Clk: clock에 따라 PE가 동작하기에 clk 신호를 받아온다.**

**Rst: PE내 존재하는 Register를 Reset 시켜주는 동작 및 연산 동작을 모두 무효화시키는 작업이다.**

**Row, col: Data\_buffer로부터 받은 데이터들에 해당하게 된다.**

**Weight: Weight stationary를 고려해주어 해당 신호가 존재한다. 해당 신호는 State가 Weight Stationary 기반 연산을 수행하라 할 때, buffer로부터 weight를 받게 된다.**

**Act\_outsig : 해당 신호는 Activation으로 가는 MMU 연산결과의 Valid를 나타낸다.**

**Out\_acc: MMU의 연산 결과를 의미한다.**

**4-1) MMU내 존재하는 Processing Unit**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Fig 13. Processing Unit**

**해당 유닛은 MMU에 배치된 Processing Unit에 해당한다.**

**Clk: clock에 동기화되어 연산이 이루어진다.**

**Rst: 연산 결과 및 과정을 초기화 시키는 Reset 신호이다.**

**Flag: 해당 신호는 data buffer로부터 들어온 데이터들이 Valid 한지 의미한다.**

**state: Instruction으로부터 나오는 State 신호를 의미하며, 해당 신호의 값에 따라 Output, Row, Weight Stationary 연산이 진행되게 된다.**

**Col\_in, row\_in : 해당 신호들은 Data Buffer로부터 들어오는 피연산자 신호라 할 수 있다.**

**Gemm-valid: 해당 신호는 연산을 진행해야 하는 지 말아야 하는지 결정하는 신호이다.**

**Gemm-valid2: 해당 신호는 연산의 종료를 나타내는 신호이다.**

**Row\_result, col\_result : PE가 Systolic array로 배치되어 있어 하나의 row output이 다른 pe의 row input이 되게 된다. 또한 col output이 다른 pe의 col input이 되게 되는데 이러한 시스템을 나타내기 위한 신호이다.**

**4-2) PE\_LAST**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Fig. 14. Processing Unit in last col**

**해당 유닛은 PE Array내 마지막 col에 해당하는 Processing Unit을 의미한다.**

**Clk: clock에 동기화되어 동작한다고 할 수 있다.**

**Rst: 연산 결과 및 과정을 초기화 시키는 Reset 신호이다.**

**Gemm\_valid: 연산의 유무를 결정하는 신호라 할 수 있다.**

**Flag: 해당 신호는 Buffer로부터 들어오는 data가 Valid한지 의미한다.**

**State: Instruction으로부터 나오는 State 신호를 의미하며, 해당 신호의 값에 따라 Output, Row, Weight Stationary 연산이 진행되게 된다.**

**PE\_LAST은 다음 col이 존재하지 않기에 Row\_RESULT가 존재하지 않는다.**

**즉, 가로로 전달할 데이터가 존재하지 않는다.**

**Gemm\_valid2: 해당 신호는 Activation Unit에 연산 결과값이 Valid한지 나타내는 신호이다.**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**Fig 15. Processing Unit in Last row**

**해당 유닛은 PE array내 마지막 row에 배치되는 Processing Unit이다.**

**마지막 row에 기존 pe와 다른 Unit이 존재하는 이유는 이 Row의 결과들이 Activation Unit의 input으로 들어가기 때문이다.**

**PE는 16X16의 형태를 띄게 되며, 출력 데이터는 16x16 Matrix로 고정되어 있다. 하지만 실제 연산의 결과는 4x4 Matrix에 대한 연산일 수도 있고 3x3 Matrix에 대한 연산일 수도 있다. 이러한 것을 고려해주는 것이 pe\_last\_row모듈의 역할이다.**

텍스트, 스크린샷, 폰트, 대수학이(가) 표시된 사진

자동 생성된 설명

**Fig 16. Activation Unit**

**해당 유닛은 MMU의 결과를 받아 비선형 함수(Relu)의 연산 결과를 Buffer로 보내는 유닛이다.**

**Act\_out\_valid 신호는 Buffer로 전달되는데, 해당 신호를 토대로 Buffer에서는 들어오는 데이터가 Valid한지 아닌지를 결정하게 된다.**

**또한 act\_out 신호는 Relu 함수의 연산 결과물에 해당하며 해당 신호 또한 Buffer로 들어가 저장된다.**

1. **Simulation**

**위 Custom Tiny TPU RTL Design을 통해 CNN 및 GEMM 연산을 진행해본다.**

1. **GEMM (General Matrix-Matrix Multiplication)**

**해당 구조를 통해 GEMM 연산을 수행하고자 한다.**

**Target Matrix는 4x4 Matrix – 4x4 Matrix간의 연산이다.**

텍스트, 폰트, 스크린샷, 화이트이(가) 표시된 사진

자동 생성된 설명

**Fig 17. 4x4 Matrix – 4x4 Matrix Multiplication**

**해당 구조는 Output stationary를 착안하여 연산을 진행하게 된다.**

**연산의 결과는 4x4 Matrix이며 해당 연산의 결과는 Buffer내 저장되어야 한다.**

**전체 하드웨어는 Instruction Set을 기준으로 작동한다. 그렇기에 먼저 Instruction Set을 작성해주어야 한다.**

라인, 직사각형, 번호, 평행이(가) 표시된 사진

자동 생성된 설명

**Fig 18. 4x4 GEMM Example**

**위 Fig 0의 Matrix 구조를 연산을 진행한다.**

**해당 구조 연산을 진행하기 위해 먼저 Instruction Set을 작성해야 한다.**

**000000000101000000000000001010000010010000110010000**

**위 Instruction은 해당 4x4 GEMM 연산을 위한 명령어이다. 명령어를 살펴보면**

**Cnn\_size =0, ptr\_in =0, ptr\_out =40, buffer\_line= 40, gemm\_size =4, out\_signal=1, state\_signal =00, acc\_map=400임을 알 수 있다.**

**이를 해석하자면, Output stationary 기반 연산을 진행하며, Buffer 내 row\_data는 0~39에 존재하고 col\_data는 40~79에 존재하는 것을 알 수 있으며 결과값은 buffer[400]부터 저장하라는 것을 알 수 있다.**

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트, 흑백이(가) 표시된 사진

자동 생성된 설명

**Fig 19. 실제 Data Buffer내 data index**

스크린샷, 라인, 다채로움이(가) 표시된 사진

자동 생성된 설명

**Fig 20. ptr값에 따른 Buffer내 data 이동**

**해당 Fig 0을 보면 Ptr 값에 따라 buffer 내 존재하는 col, row 데이터가 이동하는 것을 확인할 수 있다.**

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명**Fig 21. 연산에 따른 mmu의 결과 데이터 및 Valid 신호**

스크린샷, 다채로움, 라인이(가) 표시된 사진

자동 생성된 설명**Fig 22. Act 결과물에 대한 Buffer 내 저장**

**위 Fig 22을 살펴보면 acc\_map 주소인 400부터 결과값이 저장된 것을 확인할 수 있다.**

**앞서 말했듯이 16x16 PE array에서는 16x16의 결과값이 나오게 되는데 피연산자 Matrix가 4x4 Matrix값이기에 실제 필요한 Matrix 결과값은 4x4에 해당하게 될 것이다.**

**그렇기에 4x4개의 데이터만 Buffer내 저장한 것을 확인할 수 있다.**

1. **CNN (Convolution) 연산**

**해당 구조를 통해 필터 연산을 진행한다.**

텍스트, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명

**Fig 23. CNN 연산의 예시**

**위는 1개의 입력 이미지(3x3x2)에 대해 2개의 필터(2x2x2)가 연산되는 구조이다.**

**해당 구조의 결과물은 1개의 2x2x2의 출력 데이터가 나오게 된다.**

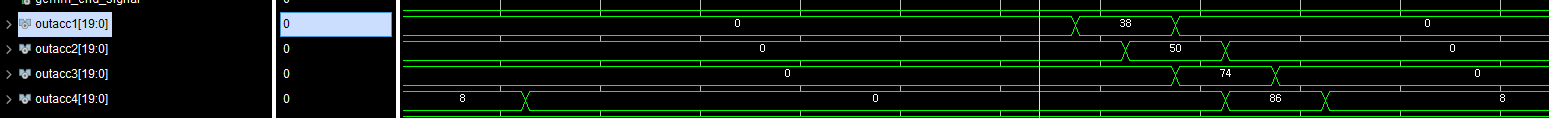
**Tensorflow 프레임워크에서 학습된 모델에서의 CNN 연산은 Matrix Transformation 과정을 거치게 된다. 입력 데이터와 파라미터간 연산이 수월하도록 하는 과정이라 할 수 있다. 그렇다면 결국 CNN 연산 또한 GEMM 연산 과정을 거친다고 할 수 있다.**

스크린샷, 다채로움, 그래픽 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**Fig 24. GEMM -> CNN 명령어가 입력된 모습**

**위 Fig 0을 통하여 GEMM 명령어에서 CNN 연산 명령어로 ptr값이 바뀜에 따라 act\_valid값들이 나오는 것을 확인할 수 있다.**



**Fig 25. 출력 데이터**

**위 Convolution 연산의 결과는 2개의 채널을 가진 2x2 데이터이다.**

**해당 채널들은 같은 데이터 값을 가지게 되며, 38, 50, 74, 86의 값을 가지게 된다.**

**Instruction내 acc\_map 데이터를 500으로 설정해 주었기에, 실제 데이터는 버퍼내 500번째 인덱스부터 저장되게 될 것이다. MMU내 col가 16개이기에, 첫번째 col에서 나오는 데이터가 500~515번까지 저장되고, 두번째 col에서 나오는 데이터는 516~530번 인덱스까지 저장되는 것을 확인할 수 있다.**

스크린샷, 다채로움, 라인이(가) 표시된 사진

자동 생성된 설명

**Fig 26. Result of Data WriteBack**

**Fig 26은 연산된 데이터가 다시 buffer로 돌아온 결과 시뮬레이션을 의미한다.**

**이렇듯 GEMM연산을 통하여 CNN 또한 연산이 가능함을 확인하였다.**

**참고 문헌**

**“In-Datacenter Performance Analysis of a Tensor Processing Unit”**

**“YouTube Channel of Prof. Dr.onur mutlu”**