High level overview of the architecture

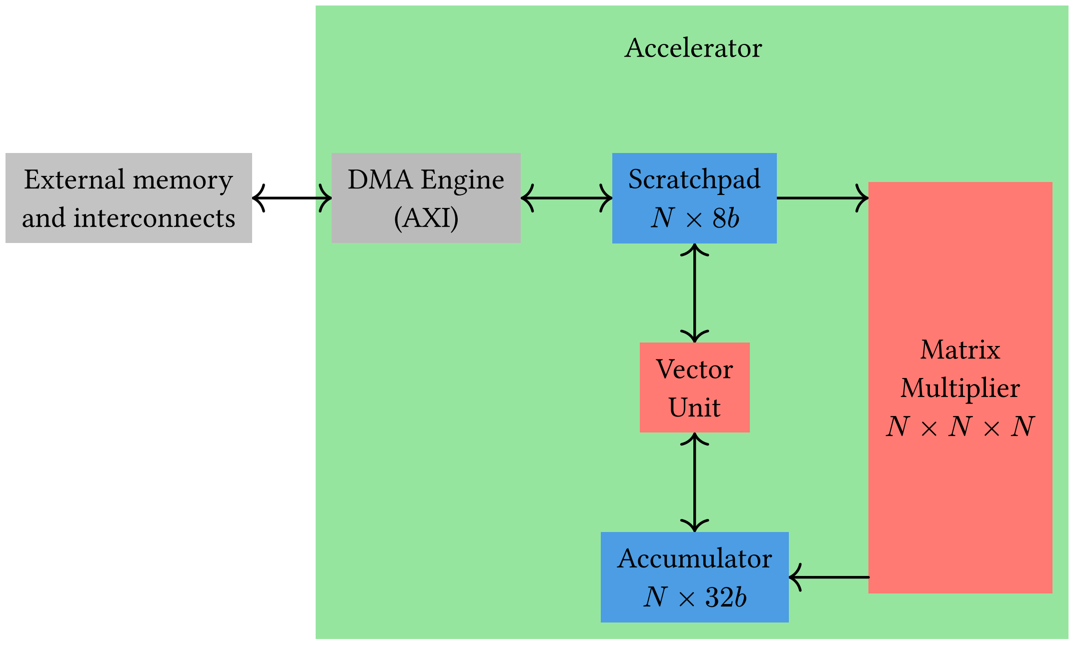
Accelerator
External memory and interconnects
DMA Engine (AXI)
Scratchpad $N \times 8b$
Vector Unit
Matrix Multiplier $N \times N \times N$
Accumulator $N \times 32b$

Figure 1: architectural diagram of the accelerator

The accelerator is build over two memories:

- The "scratchpad", this memory contains vectors of `N` elements of 8-bits, it is used to load the weights and the inputs from the external memory to the matrix multiplication unit. The registers coresponding to this memory are represented using the letters `x0, x1, x2...`
- The "accumulator", this memory contains vectors of `N` elements of 32-bits, it is used to accumulate the results from the matrix multiplier, and as an intermediate memory for the some of activation functions.The registers coresponding to this memory are represented using the letters `y0, y1, y2...`

Then the vector unit can load data from the scratchpad and the accumulator to perform intermediate computations. As example, in a convolutional neural network, it can load the value `X` from the accumulator, and write `clip(-128, 127, relu(X) >> log_scale)` back to the scratchpad to perform a `relu` layer and convert the outputs to 8-bit values.

Instruction set architecture

Register group

Vectors in the scratchpad and accumulator are named registers, written x_0, x_1, ... for the registers in the scratchpad, and y_0, y_1, ... for the ones in the accumulator. As the ISA is CISC oriented, some operations may use contiguous set of registers (named register groups), as example $x_0..x_3$ represent the list of registers [x_0, x_1, x_2, x_3], and $y_4..x_7$ represent the set of registers [y_4, y_5, y_6, y_7]. Each register group must start at an index less or equal than the one it finish at (e.e. $x_2..x_1$ is not a valid register group).

In addition x_k can be used to represent the register group $x_k..x_k$ (and y_l represent $y_l..y_l$).

Memory operations

Description

- `load` $x_a..x_b$`, (addr)`: moves `b+1-a` vectors from the main memory to the scratchpad. The inputs address must be aligned. Each loaded vector is of size $N \times 8b$.

- `store` x_a`..`x_b`, (addr)`: moves `b+1-a` vectors from the scratchpad to the main memory. The inputs address must be aligned. Each vector is of size $N \times 8b$.

Pesudo code

```
def load(destinations, address):
  for reg in destinations:
    scratchpad[reg] = memory[address]
    address += N

def store(destinations, address):
  for reg in destinations:
    memory[address] = scratchpad[reg]
    address += N
```

Matrix operations

Description

- `weights.set` x_a`..`x_b: moves `N` vectors of size $N \times 8b$ from the scratchpad to the matrix multiplication unit. Assume that `b-a+1` is equal to `N`.
- `multiply.set` y_c`..`y_d`,` x_a`..`x_b : multiply each vectors x_a`..`x_b by the matrix in the weights in the multiplication unit, and store the results in y_c`..`y_d. $d - c$ must be equal to $b - a$.
- `multiply.acc` y_c`..`y_d`,` x_a`..`x_b : multiply each vectors x_a`..`x_b by the matrix in the weights in the multiplication unit, and accumulate the results in y_c`..`y_d. $d - c$ must be equal to $b - a$.
- `multiply_reduce.set` y_c`,` x_a`..`x_b : multiply each vectors x_a`..`x_b by the matrix in the weights in the multiplication unit, sum of the results, and write it in y_c.
- `multiply_reduce.acc` y_c`,` x_a`..`x_b : multiply each vectors x_a`..`x_b by the matrix in the weights in the multiplication unit, sum of the results, and accumulate it in y_c.

Pesudo code

```
def weights.set(sources):
  for row in range(N):
    weights[row] = scratchpad[sources[i]]
    address += N

def multiply_and_set(destinations, sources):
  for (dst, src) in zip(destinations, sources):
    accumulator[dst] = weights @ scratchpad[src]

def multiply_and_accumulate(destinations, sources):
  for (dst, src) in zip(destinations, sources):
    accumulator[dst] += weights @ scratchpad[src]

def multiply_reduce_and_set(dst, sources):
  acc = zeros
  for src in sources:
    acc += weights @ scratchpad[src]
  accumulator[dst] = acc

def multiply_reduce_and_accumulate(dst, sources):
  acc = accumulator[dst]
  for src in sources:
    acc += weights @ scratchpad[src]
  accumulator[dst] = acc
```

with `@` the matrix multiplication operator.

Vector operations

Description

- `li` x_a`..`x_b`, imm` : load the constant value `imm` into all the elements of the vectors x_a`..`x_b. Really usefull to write zeros in a set of vectors.
- `li` y_a`..`y_b`, imm` : load the constant value `imm` into all the elements of the vectors y_a`..`y_b. Really usefull to write zeros in a set of vectors.
- `move` x_c`..`x_d`,` x_a`..`x_b : move vectors from x_c`..`x_d to x_a`..`x_b. Assume `d-c` is equal to `a-b`. The destinations must not intersect the sources.
- `move` y_c`..`y_d`,` y_a`..`y_b : move vectors from y_c`..`y_d to y_a`..`y_b. Assume `d-c` is equal to `a-b`. The destinations must not intersect the sources.
- `broadcast` x_a`..`x_b`,` x_c : move vector from x_c to x_a`..`x_b.
- `broadcast` y_a`..`y_b`,` y_c : move vector from y_c to y_a`..`y_b.
- `scale` x_c`..`x_d`,` y_a`..`y_b`, imm` : scale and clip values at y_a`..`y_b from 32-bits to 8-bits using a scale of `2 ** imm`, and store the results in x_c`..`x_d. Assume `d-c` is equal to `a-b`.
- `scale.relu` x_c`..`x_d`,` y_a`..`y_b`, imm` : scale, clip values at y_a`..`y_b from 32-bits to 8-bits using a scale of `2 ** imm`, and perform a relu operation. Store the results in x_c`..`x_d. Assume `d-c` is equal to `a-b`.

Pseudo code

```
def li_scratchpad(destinations, imm):
  for dst in destinations:
    for i in range(N):
      scratchpad[dst][i] = imm

def li_accumulator(destinations, imm):
  for dst in destinations:
    for i in range(N):
      accumulator[dst][i] = imm

def move_scratchpad(destinations, sources):
  for (dst, src) in zip(destinations, sources):
    scratchpad[dst] = scratchpad[src]

def move_accumulator(destinations, sources):
  for (dst, src) in zip(destinations, sources):
    accumulator[dst] = accumulator[src]

def broadcast_scratchpad(destinations, src):
  val = scratchpad[src]
  for dst in destinations:
    scratchpad[dst] = val

def broadcast_accumulator(destinations, src):
  val = accumulator[src]
  for dst in destinations:
    accumulator[dst] = val
```

```
def scale(destinations, sources, imm):
  for (dst, src) in zip(destinations, sources):
    scratchpad[dst] = clip(-127, 128, accumulator[src] >> imm)

def scale_and relu(destinations, sources, imm):
  for (dst, src) in zip(destinations, sources):
    scratchpad[dst] = clip(-127, 128, relu(accumulator[src]) >> imm)
```

Example: fully connected layers

Let's assume that we have `N=8`, and we want to perform a two layer neural network, made of two fully connected layers with a relu operator in between. The first layer is of the form $Y = \text{relu}(A \times X)$, and the second layer is of the form $Z = \text{relu}(B \times Y)$. With A of size 16×16, and B of size 8×16.

We can decompose the matrix A and B in matrices of size 8×8 using

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, B = \begin{pmatrix} B_0 & B_1 \end{pmatrix}$$

Now we have:

$$\begin{pmatrix} Y_0 \\ Y_1 \end{pmatrix} = \text{relu}\left(\begin{pmatrix} A_{00} \times X_0 + A_{01} \times X_1 \\ A_{10} \times X_0 + A_{11} \times X_1 \end{pmatrix}\right)$$

$$Z = \text{relu}(B_0 \times Y_0 + B_1 \times Y_1)$$

And we can store the those data:

- X is at address `x: *i8[16]`
- Z is at address `z: *i8[8]`
- A_{00} is at address `a00 : *i8[64]`
- A_{01} is at address `a01 : *i8[64]`
- A_{10} is at address `a10 : *i8[64]`
- A_{11} is at address `a11 : *i8[64]`
- B_0 is at address `b0 : *i8[64]`
- B_1 is at address `b1 : *i8[64]`

```
; Load the weights
load x2..x9, (a00)
load x10..x17, (a01)
load x18..x25, (a10)
load x26..x33, (a11)
load x34..x41, (b0)
load x42..x49, (b1)

; Load X
load x0..x1, (x)

; Compute Y in x0..x1
weights.set x2..x9
multiply.set y0, x0
weight.set x10..x17
multiply.acc y0, x1
weight.set x18..x25
```

```
multiply.set y1, x0
weight.load x26..x33
multiply.set y1, x1

scale.relu x0..x1, y0..y1, scale0

; Compute Z in x0
weights.set x34..x41
multiply.set y0, x0
weights.load x42..x49
multiply.acc y0, x1

scale.relu x0, y0, scale1

; Store the result
store x0, (z)
```

Off course we don't use the accelerator at it's full potential here because we use batch size of 1, we can also increase the size of the batchs to improve the usage of each sub-matrix instead of re-loading the matrix into the array each times we use it.

How to choose the scales?