Matrix Multiply: Writing and Refining FSMs

Nirav Dave
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

2007 MEMOCode HW/SW Design Contest

- Matrix Multiply engine
  - Square matrices $(64n)^2$ for n = 1..16
  - Fixed point arithmetic
  - Can do either in HW/SW
- Time Constraint – 4 weeks
- MIT Team won contest
  - BSV on XUPv2
  - 10x faster than next entry
  - 9 man weeks – mostly spent on FPGA platform, not design

The Algorithm

```
for(int i=0; i < N; i++)
  for(int j=0; j < N; j++)
    for(int k=0; k < N; k++)
      c[i][j] += a[i][k] * b[k][j];
```

- Can we implement it in terms of smaller matrix operations?
  - (KxK) matrix multiply and (KxK) matrix addition
  - A (KxK) Multiply-Add-Accumulate (MAC) unit: C = C + A*B

Expressing a MM using a (KxK) MAC – step 1

- Turn each loop into a doubly nested loop

```
for(int ib = 0; ib < N; ib += K)
 for(int io = 0; io < K; io++)
  for(int jb = 0; jb < N; jb += K)
   for(int jo = 0; jo < K; jo++)
    for(int kb = 0; kb < N; kb += K)
     for(int ko = 0; ko < K; ko++)
      c[ib+io][jb+jo]+=
                  (a[ib+io][kb+ko]*
                   b[kb+ko][jb+jo]);
```

Expressing a MM using a (KxK) MAC – step 2

- Reorder loops (addition is associative)

```
for(int ib = 0; ib < N; ib += K)
 for(int io = 0; io < K; io++)
  for(int jb = 0; jb < N; jb += K)
   for(int jo = 0; jo < K; jo++)
    for(int kb = 0; kb < N; kb += K)
     for(int ko = 0; ko < K; ko++)
      c[ib+io][jb+jo]+=
                  (a[ib+io][kb+ko]*
                   b[kb+ko][jb+jo]);
```

KxK MAC Unit

Expressing a MM using a (KxK) MAC – step 3

```
for(int ib = 0; ib < N; ib += K)
 for(int jb = 0; jb < N; jb += K)
  for(int kb = 0; kb < N; kb += K)
   kernelOperation(ib,jb,kb)
```

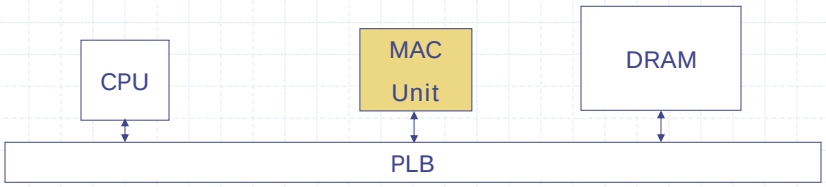
Outer Loops: SW Driver

KxK MAC Kernel to be implemented in HW

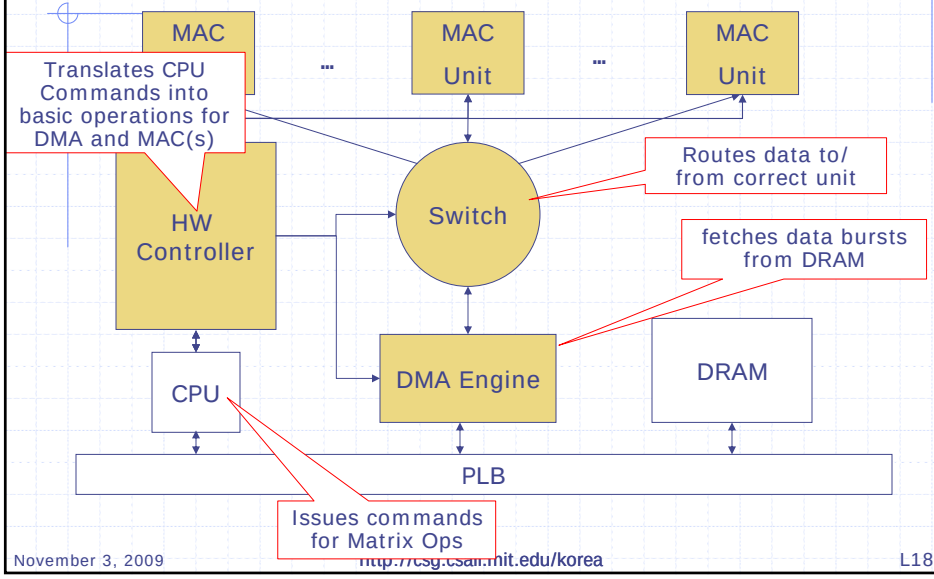
Architecture with a HW MAC

- Add a MAC to bus
  - *Assume* MAC has enough memory to hold 3 KxK matrices
  - Memory-mapped IFC
  - CPU explicitly writes and reads to the MAC

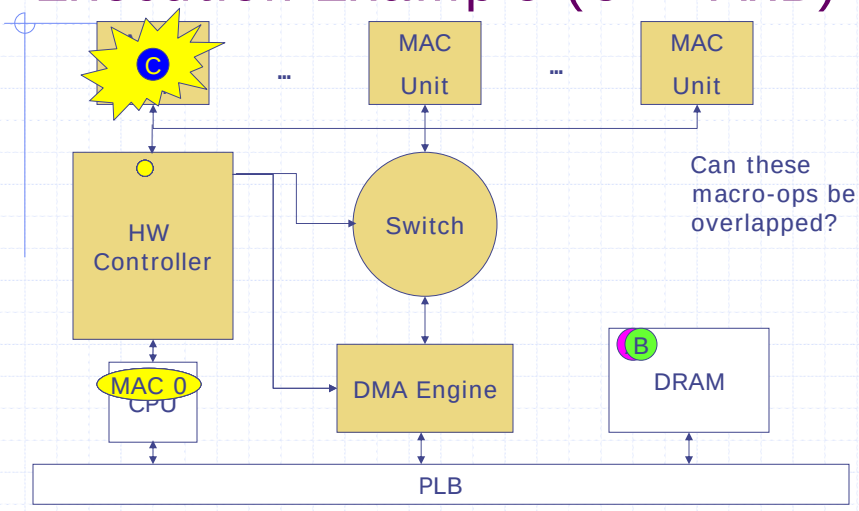
Slow way of moving data to/from MAC. Why?



A Better Architecture



Execution Example (C+=AxB)

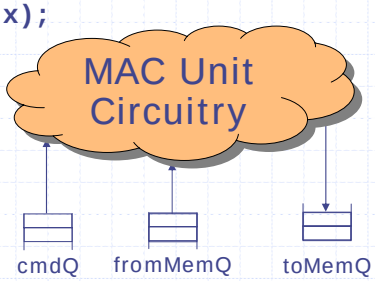


The MAC Interface

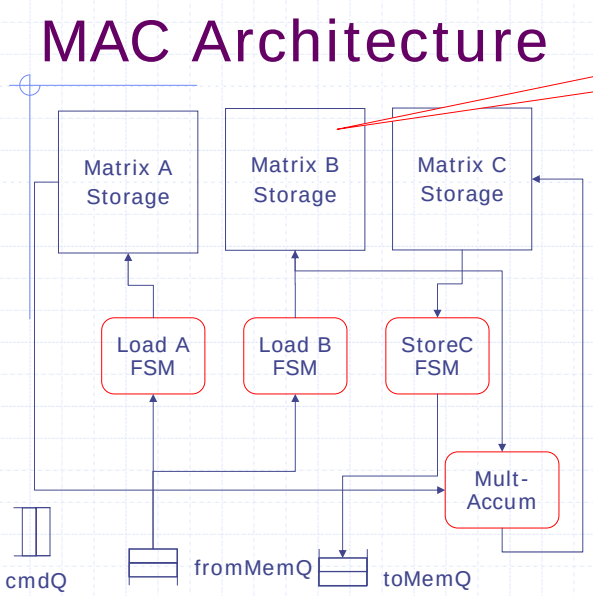
A fairly typical accelerator interface

```
interface MAC;
  method Action command(Command x);
  method ActionValue#(Value) getMem();
  method Action putMem(Value x);
endinterface

typedef enum{
   MultAccum,
   LoadA,
   LoadB,
   StoreC
} Command deriving(Eq, Bits);
```



MAC Architecture



- One FSM for each command
- Only allow one FSM to operate at a time
- Matrices are sent one word at a time through the memory FIFOs

Reminder: The FSM Interface

- FSM interface:

```
interface FSM;
    method Action  start();
    method Bool    done();
endinterface
```

done = "FSM is idle"

- Creating an FSM:

```
module     mkFSM#(Stmt s)(FSM);

module mkAutoFSM#(Stmt s)(Empty);
```

Implementing the MAC IFC

```
method Action command(Command x);
  commandQ.enq(x);
Endmethod

method ActionValue#(Value) getMem();
  toMemQ.deq();
  return toMemQ.first();
endmethod

method Action putMem(Value x);
  fromMemQ.enq(x);
endmethod
```


Starting FSMs to Process Commands

```
rule startCommand(
     loadAfsm.done && loadBfsm.done &&
    storeCfsm.done &&  multfsm.done);
  commandQ.deq();
  case(commandQ.first())
    loadA:    loadAfsm.start();
    loadB:    loadBfsm.start();
    storeC:  storeCfsm.start();
    multAccum: multfsm.start();
  endcase
endrule
```

Implementing the FSMs

Each FSM carries out K^2 operations. Let's use StmtFSM to generate the necessary rules

```
Stmt loadA =
  for(i = 0; i < K; i <= i + 1)
    for(j = 0; j < K; j <= j +1)
      action
        a.write(i,j,fromMem.first());
        fromMem.deq();
      endaction
FSM loadAfsm <- mkFSM(loadA);

Stmt loadB =
  for(i = 0; i < K; i <= i + 1)
    for(j = 0; j < K; j <= j +1)
      action
        b.write(i,j,fromMem.first());
        fromMem.deq();
      endaction
FSM loadBfsm <- mkFSM(loadB);
```

Remember: B is transposed (we are storing things in column-major order)

Implementing the FSMs

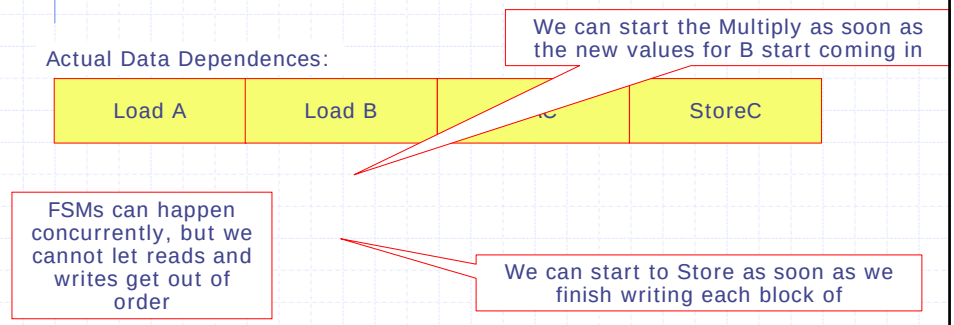
```
Stmt mulAccum =
 for(i <= 0;i < K; i <= i + 1)
  for(j <= 0;j < K; j <= j + 1)
   for(k <= 0;k < K; k <= k + 1)
     action let av <- a.read(i,k);
            let bv <- b.read(j,k);
            let cv = c.read(i,j);
            c.write(i,j,cv+ab*bv); endaction
FSM multfsm <- mkFSM(multAccum);

Stmt storeC =
  for(i <= 0; i < K; i <= i + 1)
    for(j <= 0; j < K; j <= j + 1)
      action let x <- c.read(i,j);
             toMem.enq(x);
      endaction;
FSM storeCfsm <- mkFSM(storeC);
```

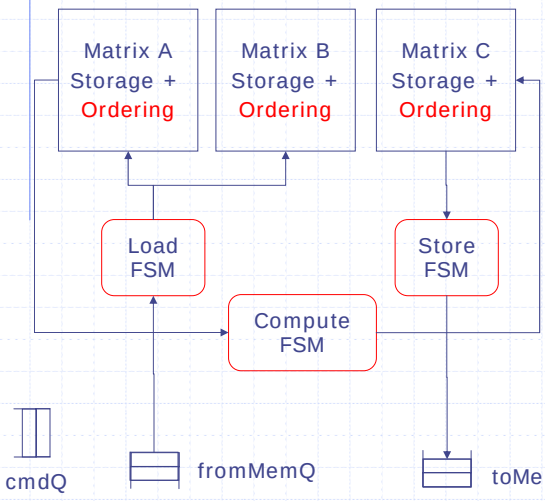
Remember B matrix is transposed

We have a Working Version!

- Matches high-level algorithms
- Can we be more parallel?
  - Consider: Ld A, Ld B, MAC, St C



Revising the MAC Unit



- FSMs are tied to resources
  - No structural hazards
  - FSMs can run concurrently
  - addresses computed in the storage
- Data dependences?
  - Add logic to storage to keep reads and writes in the correct order