Energy Efficient Matrix

A Proposal for RISC-V Matrix Extension(s)

Claim

- AI may fuel significant increases in energy use
- AI currently depends on matrix multiplication
  - (Unless matmul-free preprints pan out)
- Without energy efficiency, solutions will be discarded and replaced with efficient ones
- RISC-V should bake in energy efficiency from the start

Preliminaries

- Problem Size N, Tile size T
  - Matrixes and Tiles can be rectangular, but complicates presentation
  - Data is N^2, number of mul/adds is N^3, Tile is T^2 computed with T^3 mul/adds
  - Parallelism N^2, full parallelism latency $N\Delta$, tile parallelism T^2 latency $T\Delta$
  - Data usage: each element of A used N times, same for B, T for tiles
- Matrix Multiply

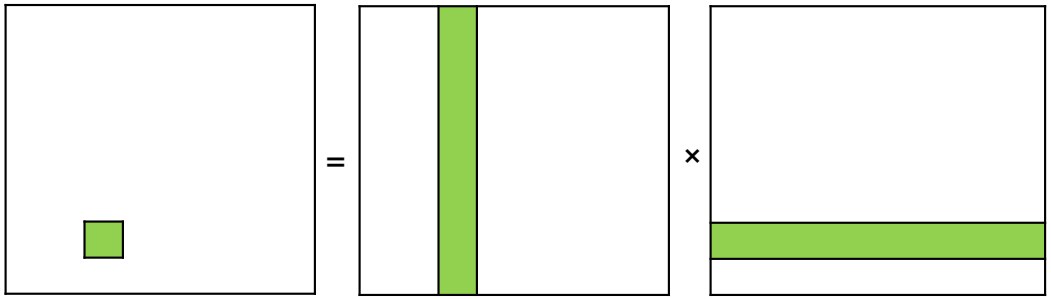
```
for i ← 0 to m-1
  for j ← 0 to p-1
    for k ← 0 to n-1
      c[i,j] ← c[i,j] + a[i,k] * b[k,j]
```

- Matrix Multiply with accumulator

```
for i ← 0 to m-1
 for j ← 0 to p-1
   acc ← c[i,j]
   for k ← 0 to n-1
     acc ← acc + a[i,k] * b[k,j]
   c[i,j] ← acc
```

- Tiled Matrix Multiply

```
for ti ← 0 to m-1 step TR.  // tile i
  for tj ← 0 to p-1 step TC // tile j
    for i ← 0 to TR-1
      for j ← 0 to TC-1
        for k ← 0 to n-1
          c[ti+i,tj+j] ← c[ti+i,tj+j] + a[ti+i,k] * b[k,tj+j]
```



Goals

- Energy efficiency
  - Use load data as many times as reasonable per load
  - Use register file reads as many times as reasonable per access
  - Make storage local and small
- Balancing load bandwidth and computation
  - Make them equally limiting when reasonable
- Scalability
  - Expose parallelism so that implementations may exploit as much as they are willing to pay for
- Cost
  - Exploit pipelining to save area
  - Keep storage proportional to benefit
- Complexity
  - Prefer simple instructions to complex ones when other things equal
    - Avoid iteration in hardware when iteration in software works just as well
  - Provide flexibility

Balance Load and Compute

- If load bandwidth is V elements/cycle $T = V$ and computing V^2 mul/adds in two cycles balances load and compute time
  - $V \times (\frac{V}{2})$ array of mul/add units accomplishes this if $\Delta \leq 2$
  - Compute $\Delta/2$ tiles of C in parallel if $\Delta > 2$
- Outer product method:
  - 1 cycle load V element vector from A to VRF
  - 1 cycle load V element vector from B to VRF
  - 2 cycle outer product accumulating to C tile in ???
  - iterate V times in software
- Matrix method
  - V cycle load $V \times V$ tile from A to MRF
  - V cycle load $V \times V$ tile from B to MRF
  - $2V$ cycle tile matmul accumulating to C tile in MRF, iterating in hardware

Outer Product (OP) vs Matrix Multiply (MM)

- Both have computational intensity $\frac{V}{2}$ when load balanced ($T = V$)
- Both have V element reuse when load balanced
- OP has $2V$ VRF storage, and V^2 for C tile in ???
- MM has $3V^2$ MRF storage, much more than OP
- MM has greater complexity, OP is simpler
- Question is where to store C tile for accumulation for OP
  - Storing C file in VRF is far from compute, high-power, high-area
    - VRF bits have minimum 4 read ports, 2 write ports
    - VRF size increases often required to store V^2 C tile—not a good trade-off
  - Distributed accumulators in the mul/add array are local, low-power, low-area
    - Local accumulator bits have 1 read, 1 write port

Using Memory Data Multiple Times

- An accumulation tile of T×T use each source element T times
- Energy efficiency therefore wants to maximize T (fewer reloads)
- Maximizing T with matrixes means $3T^2$ register file storage (C, A, B)
- Maximizing T with outer product means T^2 for C only
  A and B are only T element vectors
- For the same number of register file bits, outer product supports larger T
- Examples for 512 b/cycle load bandwidth:
  64×64 int8 with int32 accumulation = 128 Kib (16 KiB)
  64×64 fp8 with TF32 accumulation = 78 Kib (9.7 KiB)

Less Compute Than Load Bandwith

$\Delta \leq 2$ mul/add array sizes

Load bits	8b	16b	32b	64b
256	32×16	16×8	8×4	4×2
512	64×32	32×16	16×8	8×4

$\Delta = 4$ mul/add array sizes

Load bits	8b	16b	32b	64b
256	64×16	32×8	16×4	8×2
512	128×32	64×16	32×8	16×4

If these mul/add arrays are too large, compute will be slower than loads.
Example: load 512b/cycle, TF32 + BF16×BF16, $\Delta = 2$

- Balanced needs 512 mul/adds to compute one 32×32 tile every 2 cycles
- If only 128 mul/add possible? Clearly 4× slower, but the method matters:
  - OP: load 32-element vectors, use 16×8 mul/add array 8 times into 32×32 accumulators each element used 32 times ⇒ high energy efficiency
  - MM: load 16×16 tiles (16 cycles), use 16×8 mul/add array 32 times into 16×16 MRF accumulation tile, each element used 16 times times ⇒ lower energy efficiency
- Comparison: OP is simpler, better energy efficiency
- OP wins in balanced, wins more when sub-bandwidth compute

Proposed Instructions

- `msetcli rd, rs1, vtypei` sets `VL` based on the number of columns specified in `rs1` that can be processed by the `v*outer.vv` instructions and returns this value to `rd`.
- `msetrli rd, rs1, vtypei` sets `VL2` based on the number of rows specified in `rs1` that can be processed by the `v*outer.vv` instructions and returns this value to `rd`.
- `vwouter.vv vs1, vs2` performs an integer widening outer product (`acc ← acc + outerproduct(vs1, vs2)`) for the current settings of `SEW`, `VL`, and `VL2`. Only `VL` columns and `VL2` rows of the accumulators are modified.
- `vfouter.vv vs1, vs2` performs a floating-pointer outer product (`acc ← acc + outerproduct(vs1, vs2)`) for the current settings of `SEW`, `VL`, and `VL2`. Only `VL` columns and `VL2` rows of the accumulators are modified.
- `vracc vd, rs1` reads the row of `acc` selected by `rs1` and writes this to `vd` in the integer format selected by `SEW` using the length specified by `VL`. This transfer may involve some format conversion from the internal accumulator format to the specified integer format.
- `vfracc vd, rs1` reads the row of `acc` selected by `rs1` and writes this to `vd` in the floating-point format selected by `SEW` using the length specified by `VL`. This transfer may involve some format conversion from the internal accumulator format to the specified integer format.
- `vrracc vd, rs1` reads the raw format row of `acc` selected by `rs1` and writes this to `vd` as bytes as determined by `acccolb`.
- `vwacc rs1, vs2` writes the row of `acc` selected by by `rs1` from `vs2` in the integer format selected by `SEW` using the length specified by `VL`. This transfer may involve some format conversion to the internal accumulator format from the specified integer format.
- `vfwacc rs1, vs2` writes the row of `acc` selected by `rs1` from `vd` in the floating-point format selected by `SEW` using the length specified by `VL`. This transfer may involve some format conversion to the internal accumulator format from the specified integer format.
- `vrwacc rs1, vs2` writes the raw format row of `acc` selected by `rs1` from `vs2` as bytes as determined by `acccolb`.

Conclusions

- This analysis suggests
  - Don’t store matrixes in VRF or MRF
  - Accumulate into local mul/add array registers, not a distant register file
  - Matrix extensions should optimize for
    - Loading vectors
    - Computing the outer product of these vectors
    - Accumulate into local storage for energy savings
  - Target a balance between load bandwidth and compute when possible
    - Maintain element reuse when compute < load bandwidth for energy efficiency
  - Other additions possible but this should be the core functionality