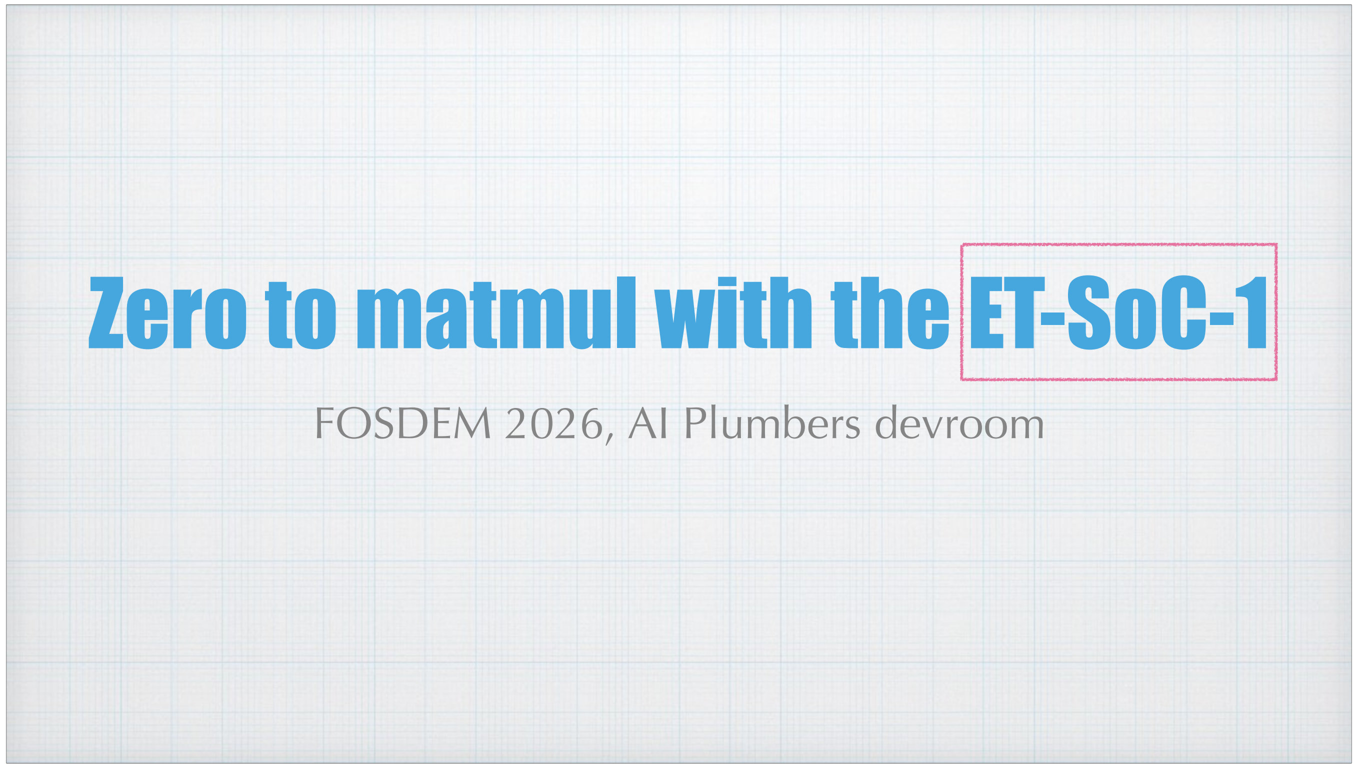
Zero to matmul with the ET-SoC-1

FOSDEM 2026, AI Plumbers devroom

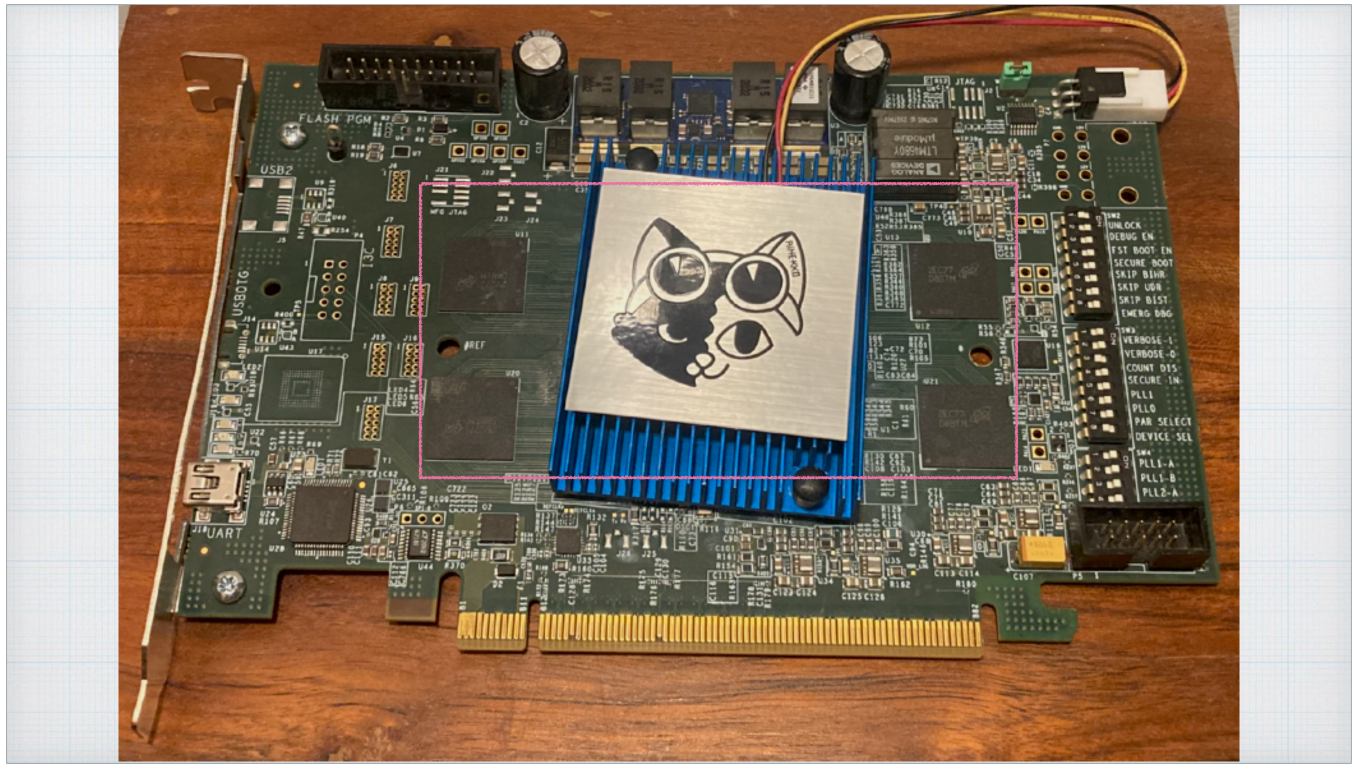


Zero to matmul with the $\boxed{\text{ET-SoC-1}}$

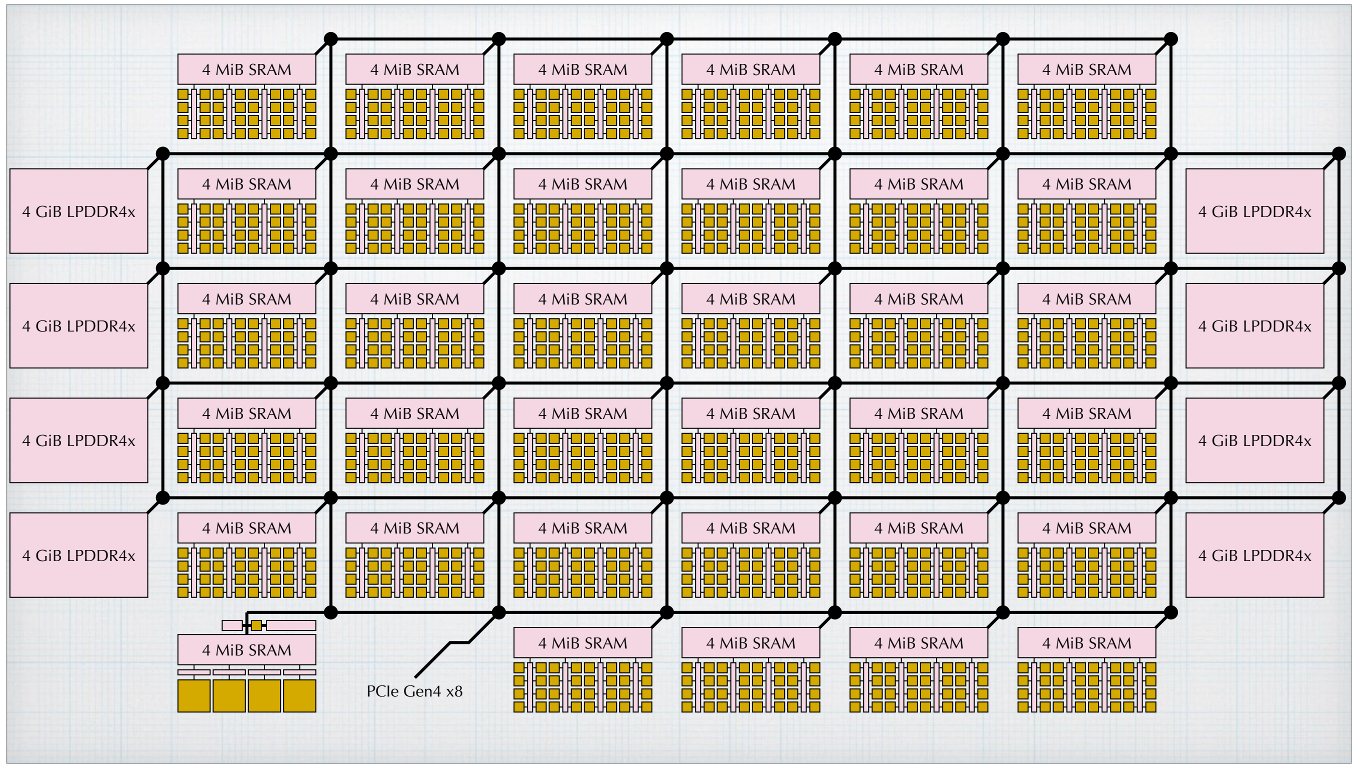
FOSDEM 2026, AI Plumbers devroom

Hopefully everyone in the room knows what zero is, and knows what matmuls are. If you were in the room for the previous talk from Gianluca, then you'll know what ET-SoC-1 is. For the benefit of anyone just joining us, ET-SoC-1 is a piece of hardware.

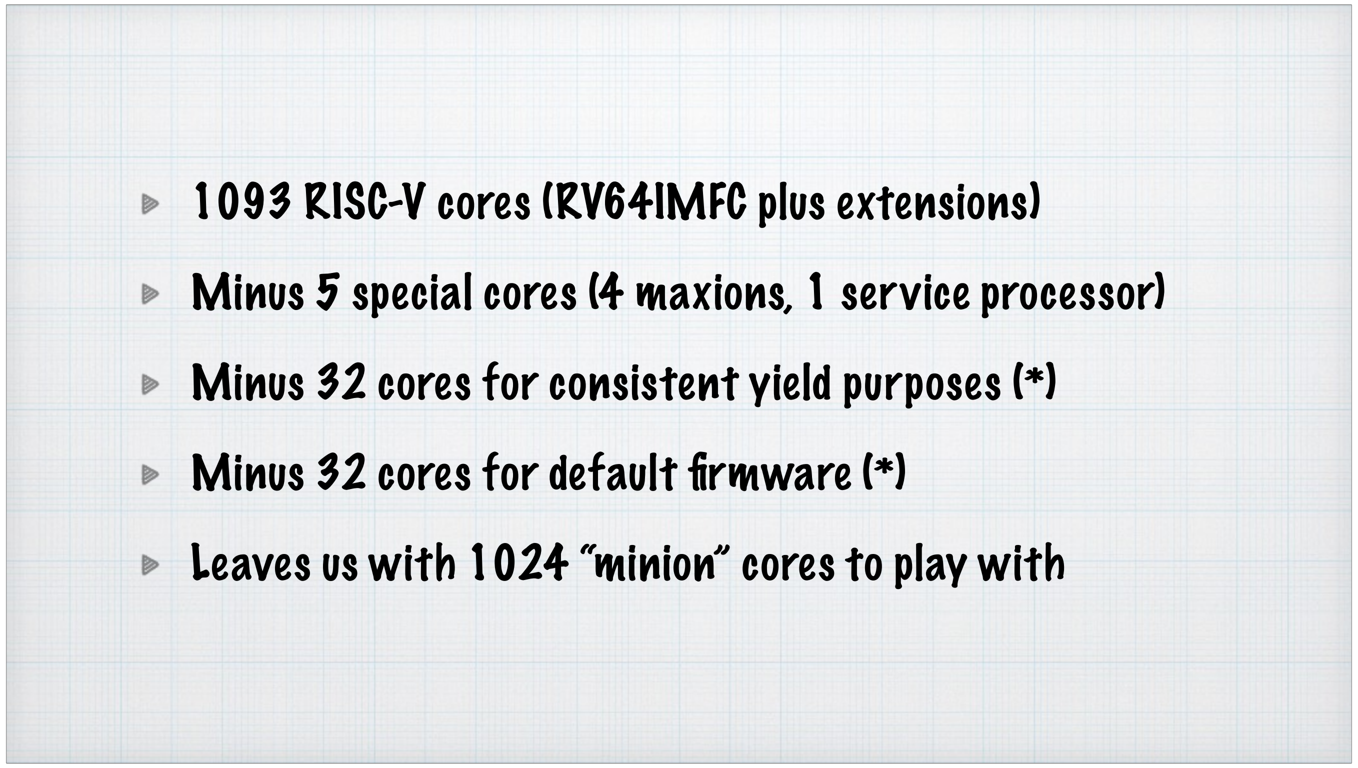
You can find this piece of hardware on cute little PCIe boards, such as this one.



All of the interesting bits are within this pink rectangle: four DRAM chips on the edges, and then the main ASIC under the blue heatsink.

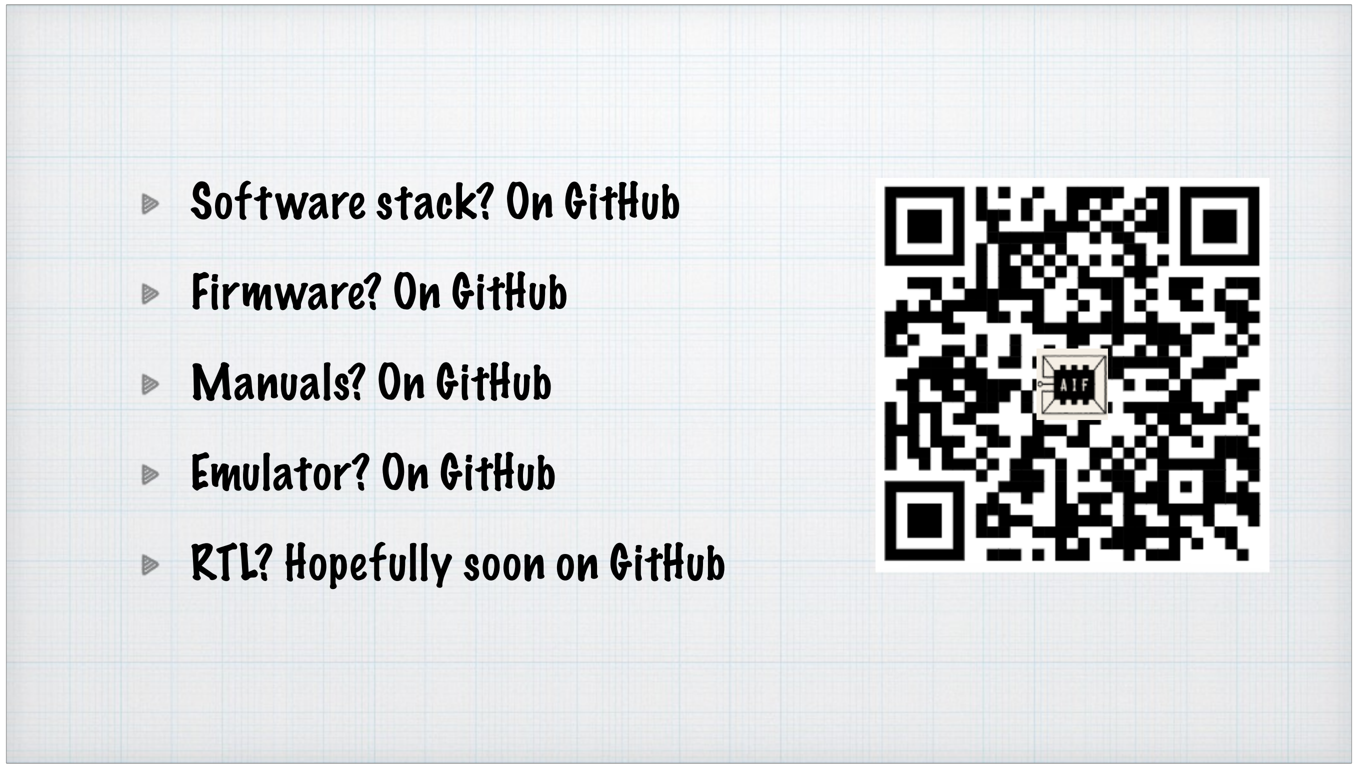


This is a block diagram of that highlighted area. Each pink rectangle is a piece of memory, and each gold square is a RISC-V CPU core. The black network then connects it all together.



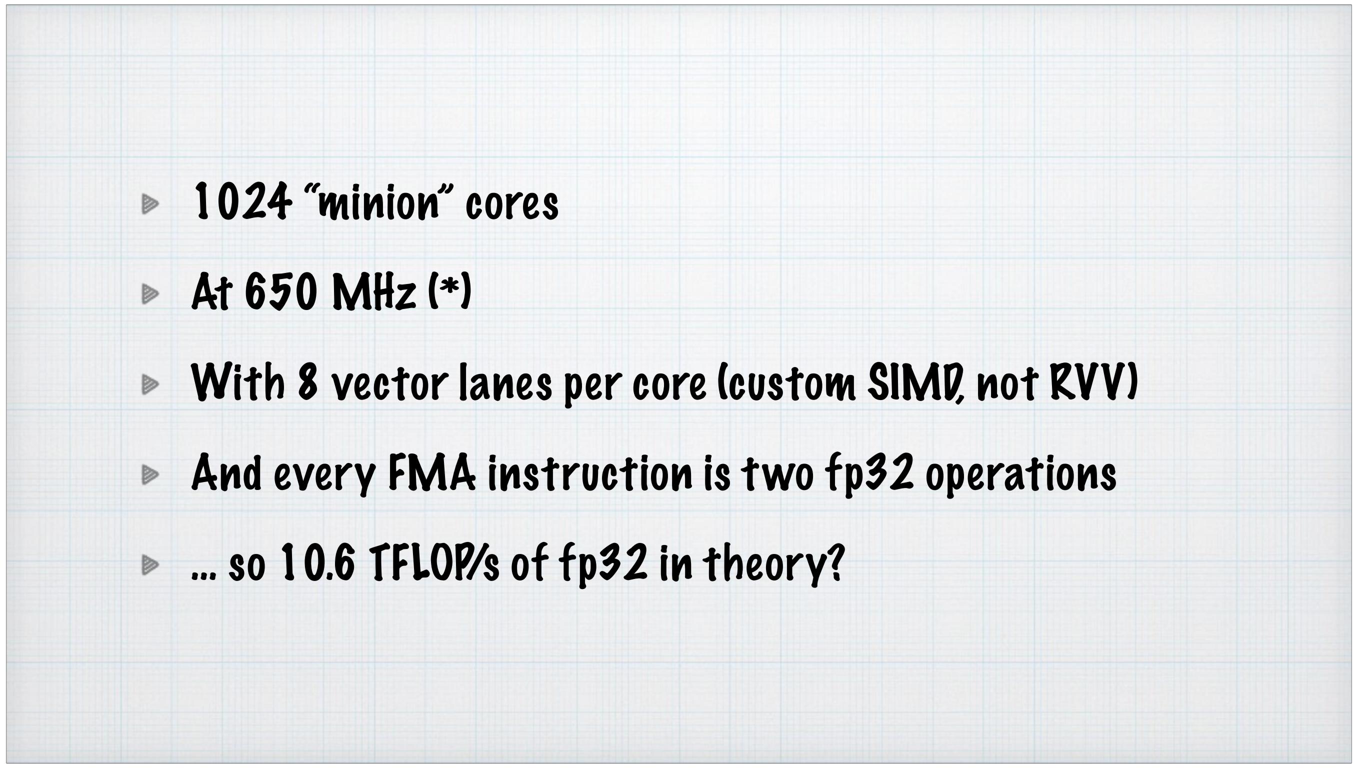
- 1093 RISC-V cores (RV64IMFC plus extensions)
- Minus 5 special cores (4 maxions, 1 service processor)
- Minus 32 cores for consistent yield purposes (*)
- Minus 32 cores for default firmware (*)
- Leaves us with 1024 "minion" cores to play with

If you were to count them all, you'd find 1093 gold squares, each of them a RISC-V core.
Five of them are special, so I'm ignoring those.
32 might be lost at the altar of silicon yield, so I'm ignoring those too.
Another handful are used for running the default firmware; to keep things simple, I'll say the firmware takes 32.



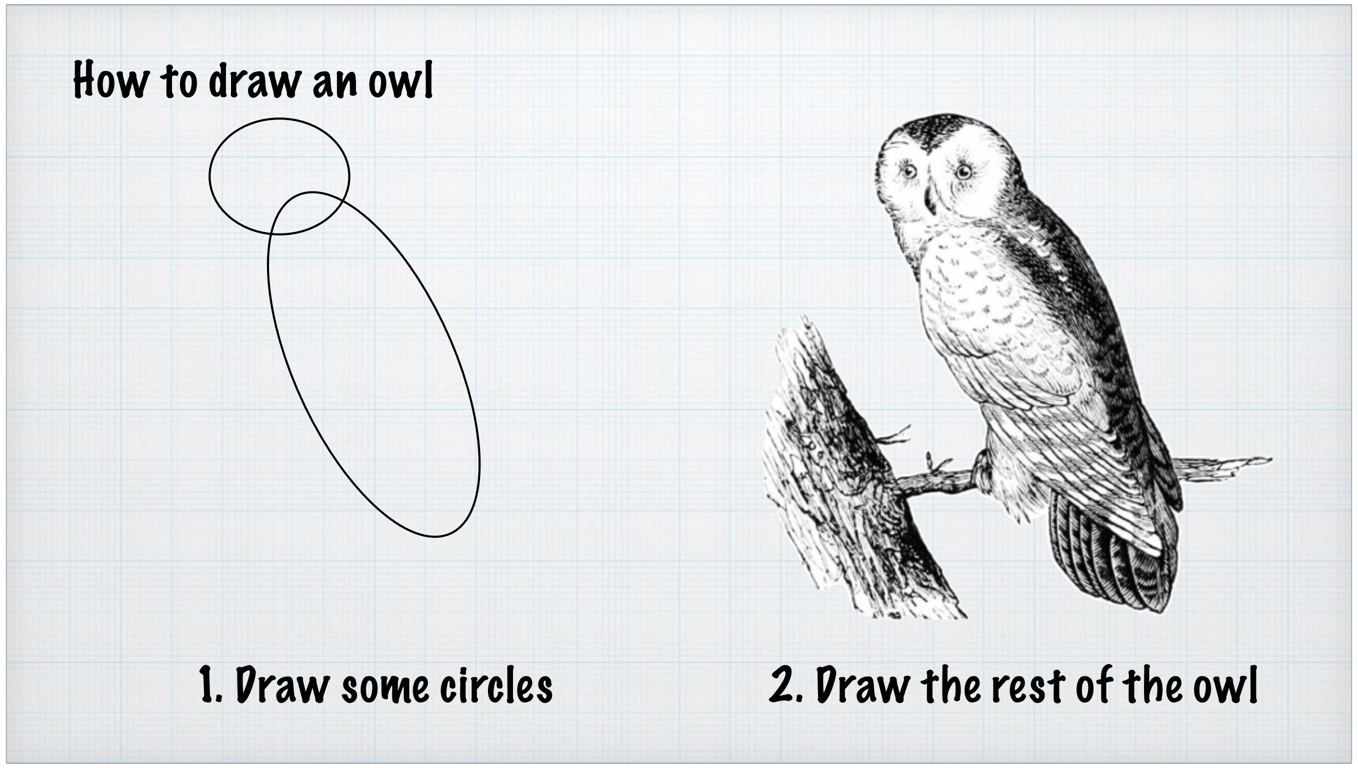
- Software stack? On GitHub
- Firmware? On GitHub
- Manuals? On GitHub
- Emulator? On GitHub
- RTL? Hopefully soon on GitHub

As this is FOSDEM, I want to call out the open source credentials here. This QR code takes you to the AIFoundry GitHub, where you'll find all the software, firmware, and manuals for this thing, along with a full emulator. I'm told that a bunch of the RTL will also be open-sourced soon, which I'm looking forward to.

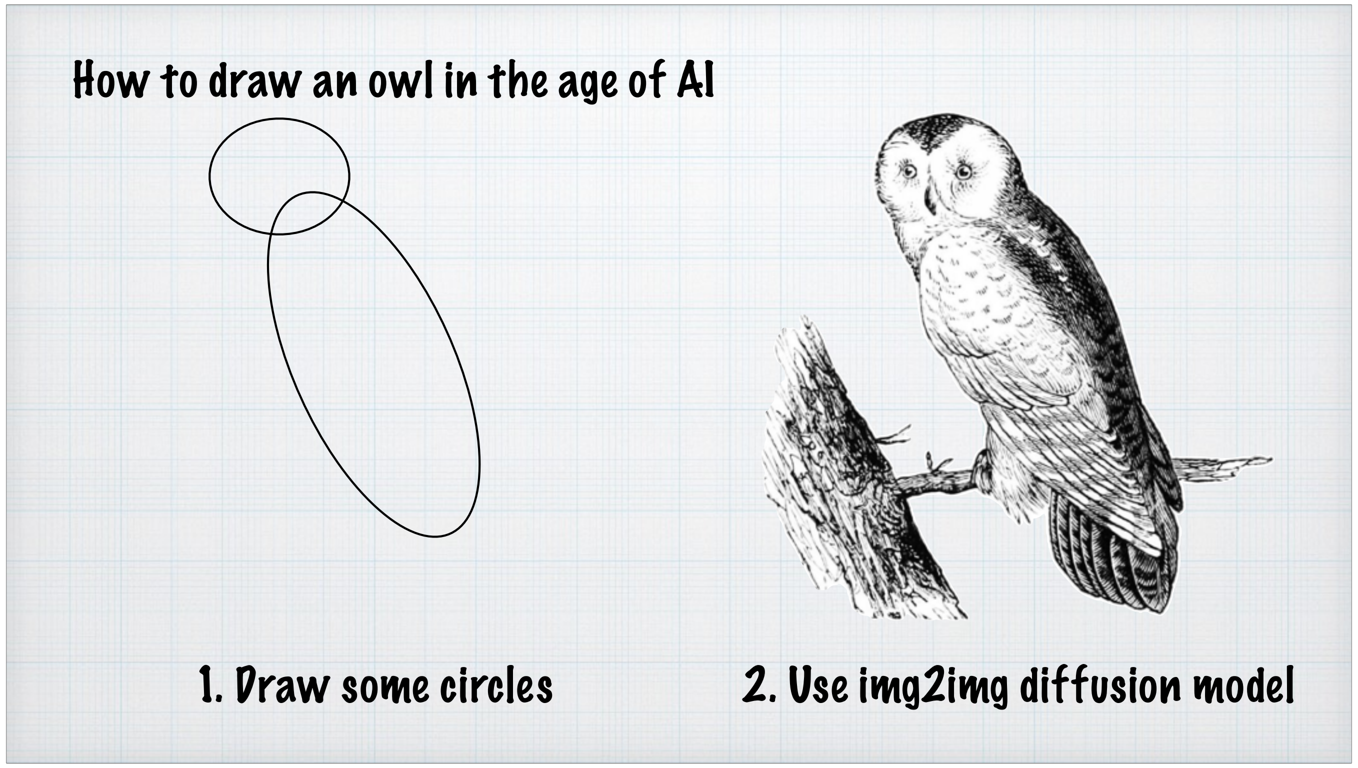


- 1024 “minion” cores
- At 650 MHz (*)
- With 8 vector lanes per core (custom SIMD, not RVV)
- And every FMA instruction is two fp32 operations
- ... so 10.6 TFLOP/s of fp32 in theory?

I’ve got these thousand cores, and I’m choosing to run them at a fixed clock rate for reproducibility. At my chosen fixed clock, the math says we should have just over 10 teraflops of fp32 compute.



Some of you will have seen this meme before, which is the how to draw the owl meme: first draw some circles, then draw the rest of the owl. The joke of course being that step 2 is a massively more work than step 1.

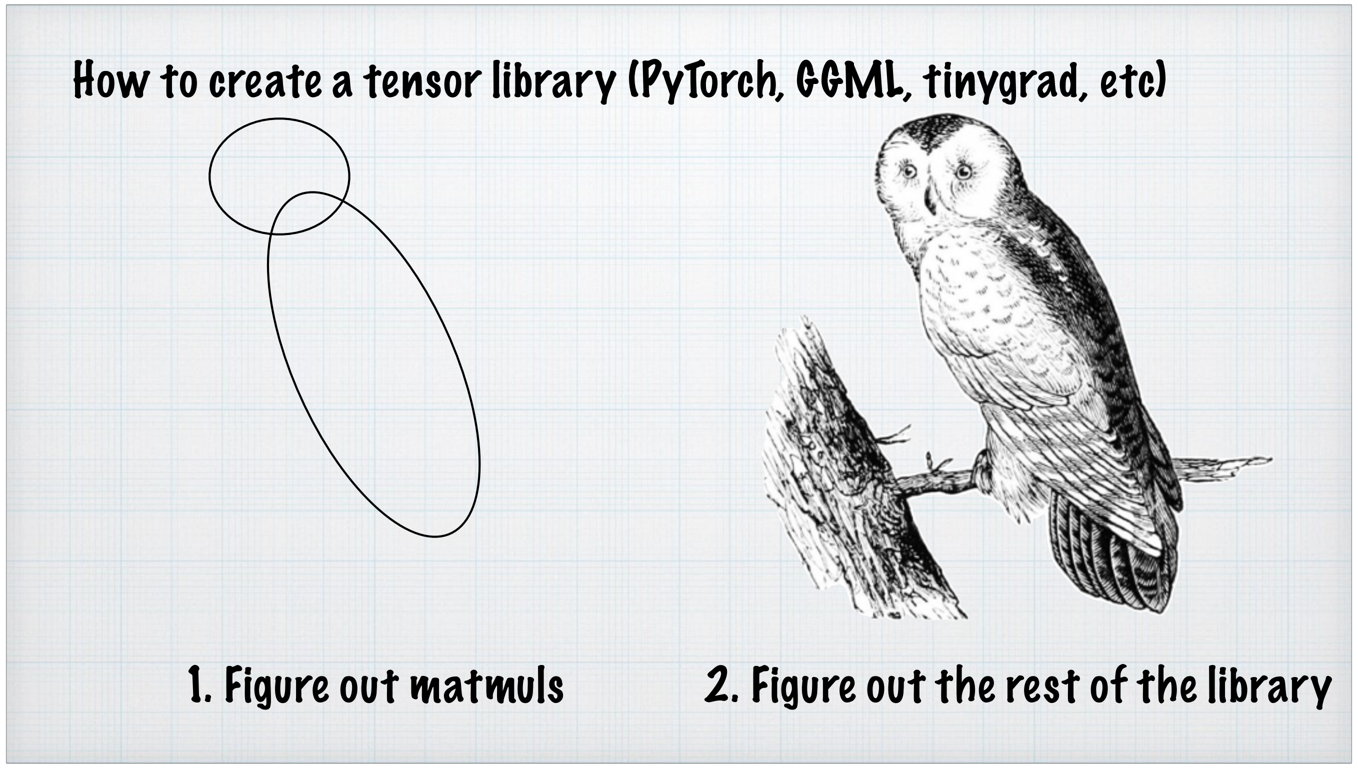


How to draw an owl in the age of AI

1. Draw some circles

2. Use img2img diffusion model

As we’re in the AI devroom, we can use AI for step 2.

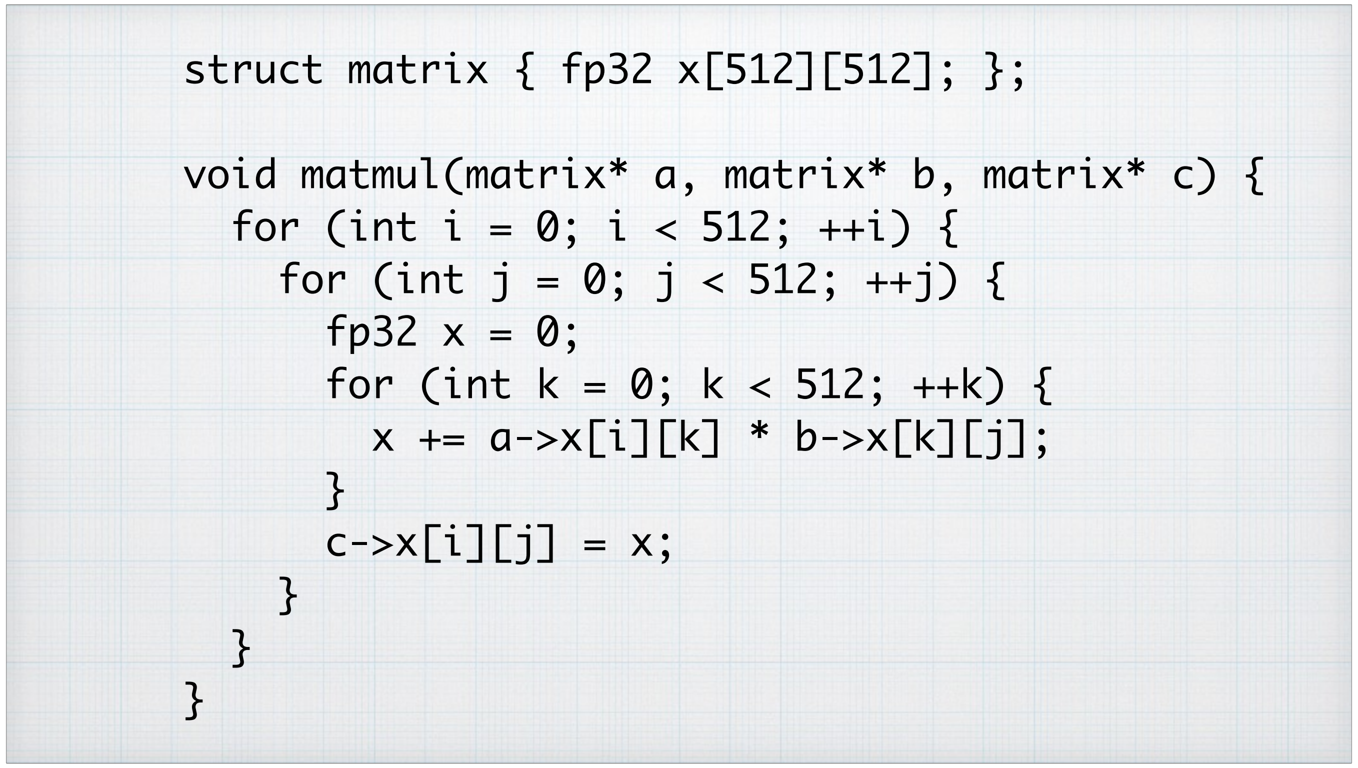


How to create a tensor library (PyTorch, GGML, tinygrad, etc)

1. Figure out matmuls

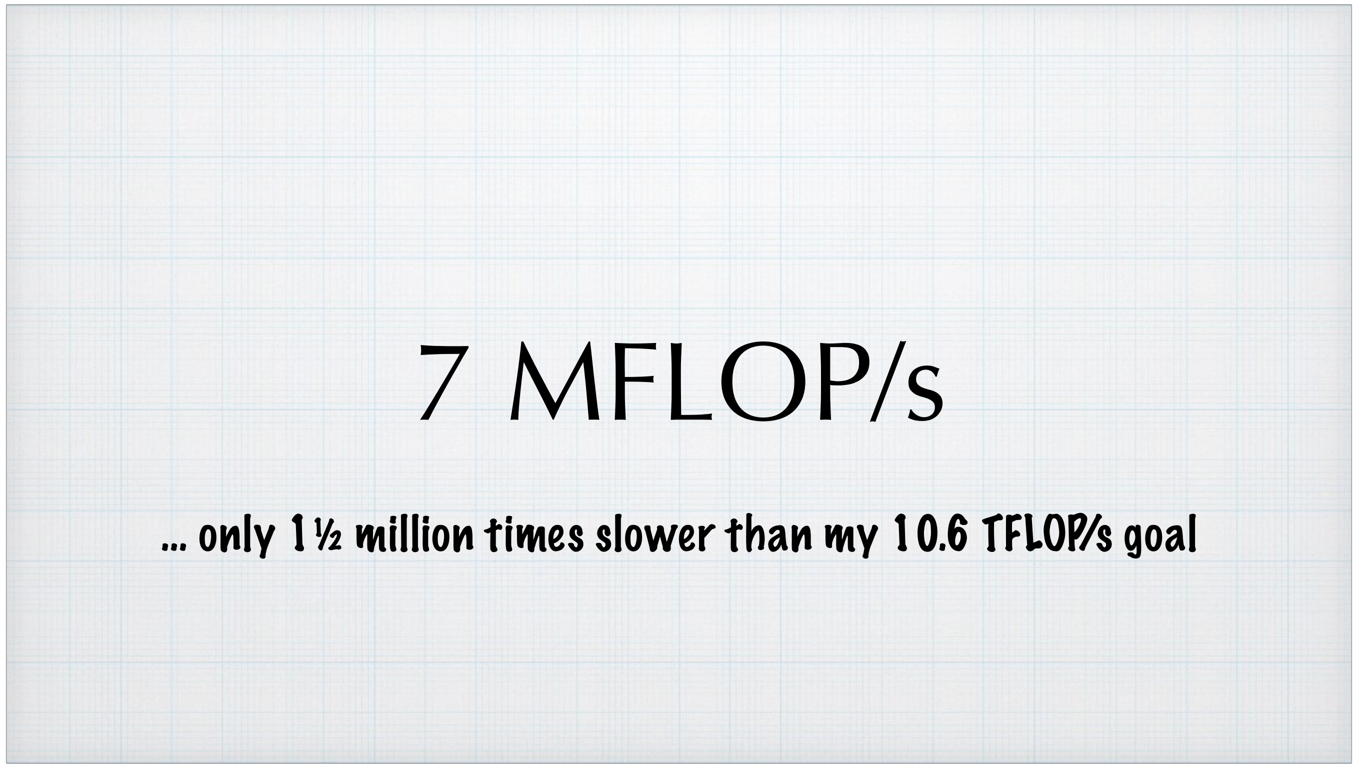
2. Figure out the rest of the library

But as this is the AI _plumbers_ devroom, we're implementing AI rather than just using it. Our equivalent of drawing an owl is creating a tensor library, and step 1 of that is figuring out matmuls. I've only got a 20 minute slot here, so I'm just drawing some circles; it's massively more work to create a proper library.



```
struct matrix { fp32 x[512][512]; };

void matmul(matrix* a, matrix* b, matrix* c) {
  for (int i = 0; i < 512; ++i) {
    for (int j = 0; j < 512; ++j) {
      fp32 x = 0;
      for (int k = 0; k < 512; ++k) {
        x += a->x[i][k] * b->x[k][j];
      }
      c->x[i][j] = x;
    }
  }
}
```

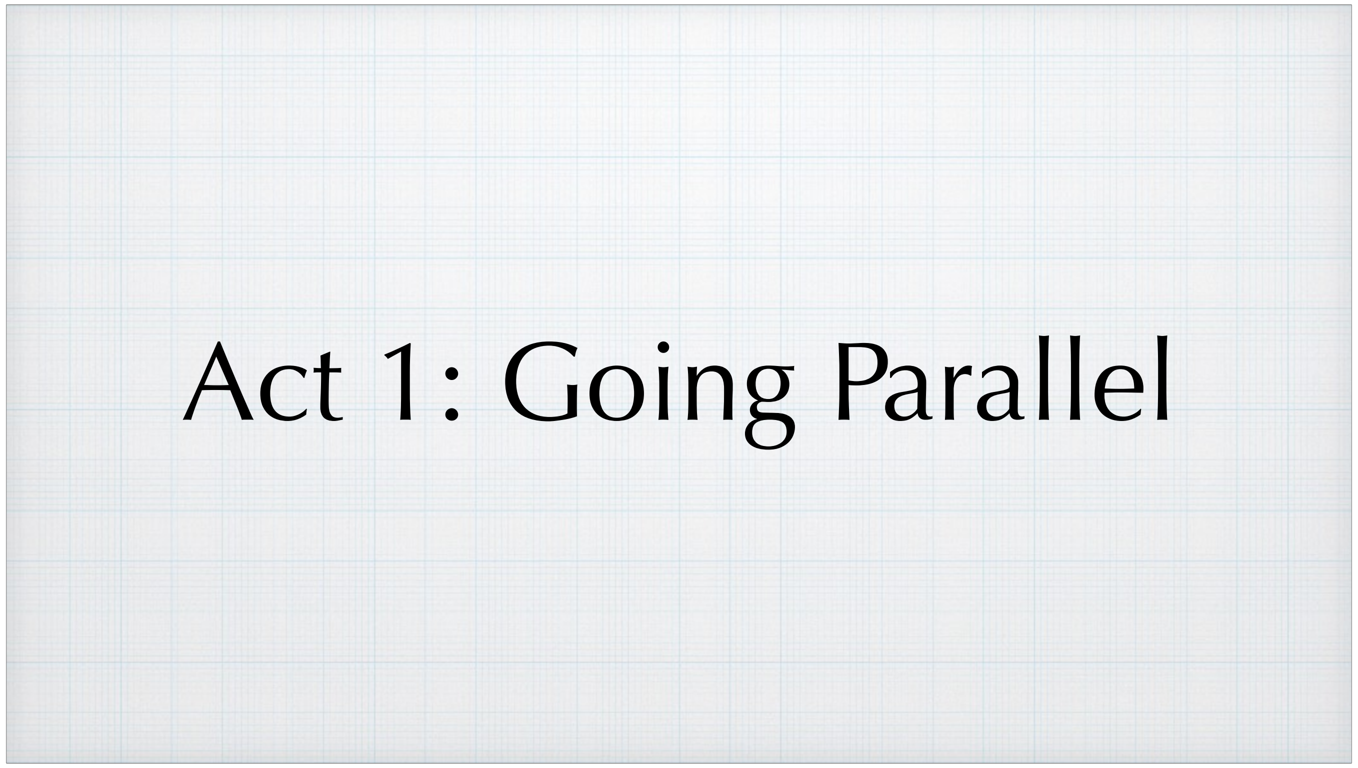
To stay within time, I'll assume that a matrix is always 512 by 512 at fp32 precision - anything else is drawing the rest of the owl. The code here will multiply matrix a by matrix b, and write the result to c, but is the worst performing matmul you'll ever see.



7 MFLOP/s

… only 1½ million times slower than my 10.6 TFLOP/s goal

If I run that code on my ET-SoC-1, it runs at an appalling 7 megaflops per second, which is more than a million times slower than my 10 teraflop goal. Like I said, worst performing matmul you'll ever see, but I've got 17 minutes to go, which is surely enough time to make it a million times faster.

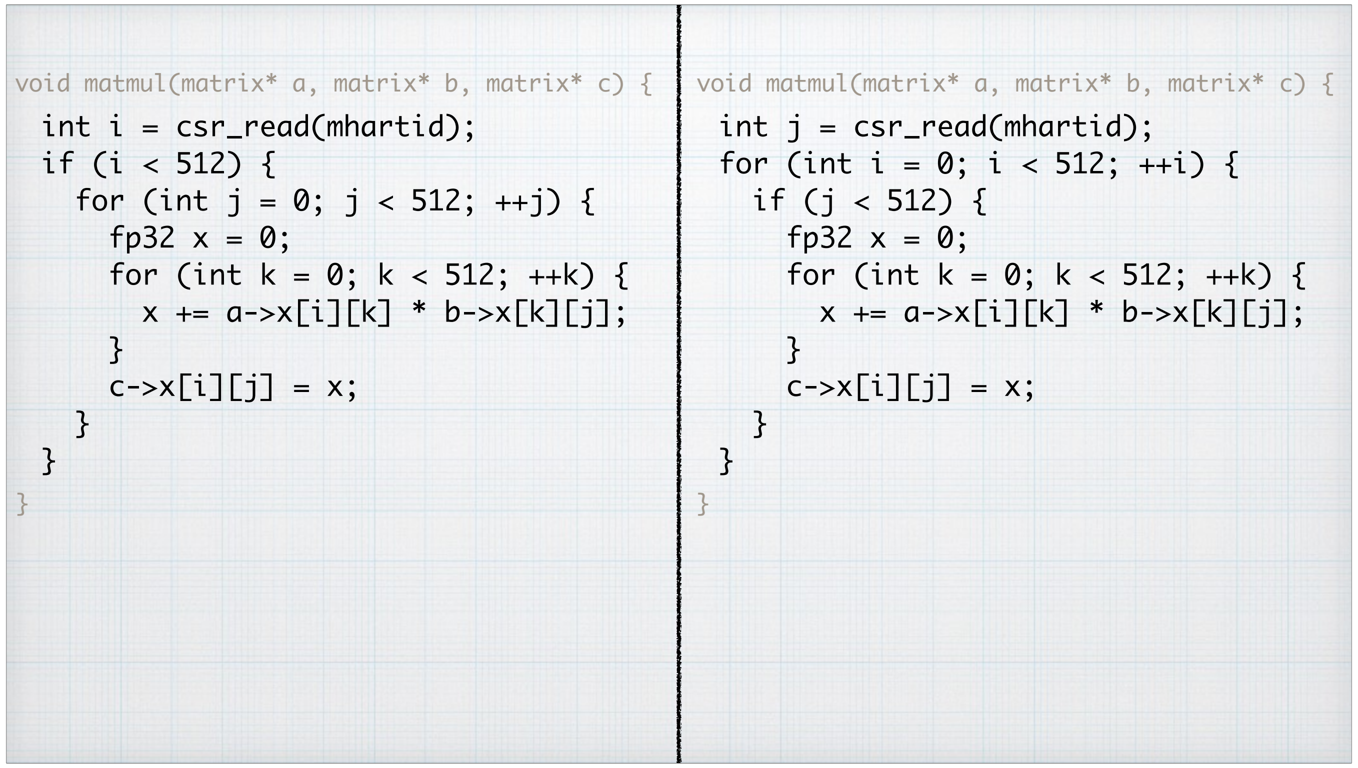


Act 1: Going Parallel

[Time guide: T+4m00s]
Like all good plays, I’m going to work toward my goal over three main acts. Act 1 I’ve titled “Going Parallel”.

mhartid is threadIdx and blockIdx (mostly)

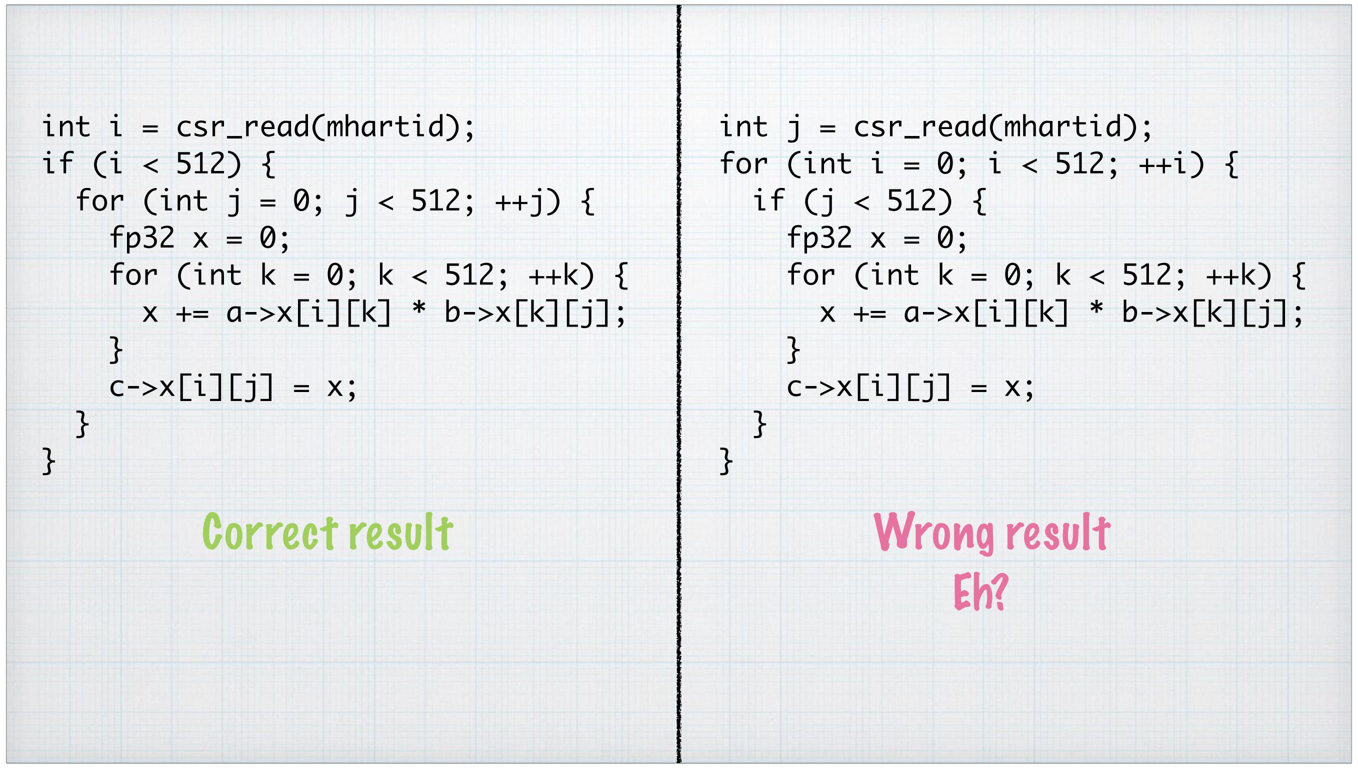
If you’ve written any CUDA code, you’ll have seen threadIdx and blockIdx: the CUDA model is that you run thousands of copies of your code in parallel, with each copy having a distinct threadIdx / blockIdx. I’m using a similar setup here: all the code I’m showing you gets run 2048 times in parallel, each with a distinct value of mhartid between 0 and 2047. If you’re paying attention, you’ll remember I said 1024 cores, but 2048 different mhartid values: the hardware effectively has two-way hyperthreading with two harts per core.



```
void matmul(matrix* a, matrix* b, matrix* c) {
  int i = csr_read(mhartid);
  if (i < 512) {
    for (int j = 0; j < 512; ++j) {
      fp32 x = 0;
      for (int k = 0; k < 512; ++k) {
        x += a->x[i][k] * b->x[k][j];
      }
      c->x[i][j] = x;
    }
  }
}
```

```
void matmul(matrix* a, matrix* b, matrix* c) {
  int j = csr_read(mhartid);
  for (int i = 0; i < 512; ++i) {
    if (j < 512) {
      fp32 x = 0;
      for (int k = 0; k < 512; ++k) {
        x += a->x[i][k] * b->x[k][j];
      }
      c->x[i][j] = x;
    }
  }
}
```

To start out, I can use mhartid to parallelise just one of the loops, either the outer loop (left) or the inner loop (right).



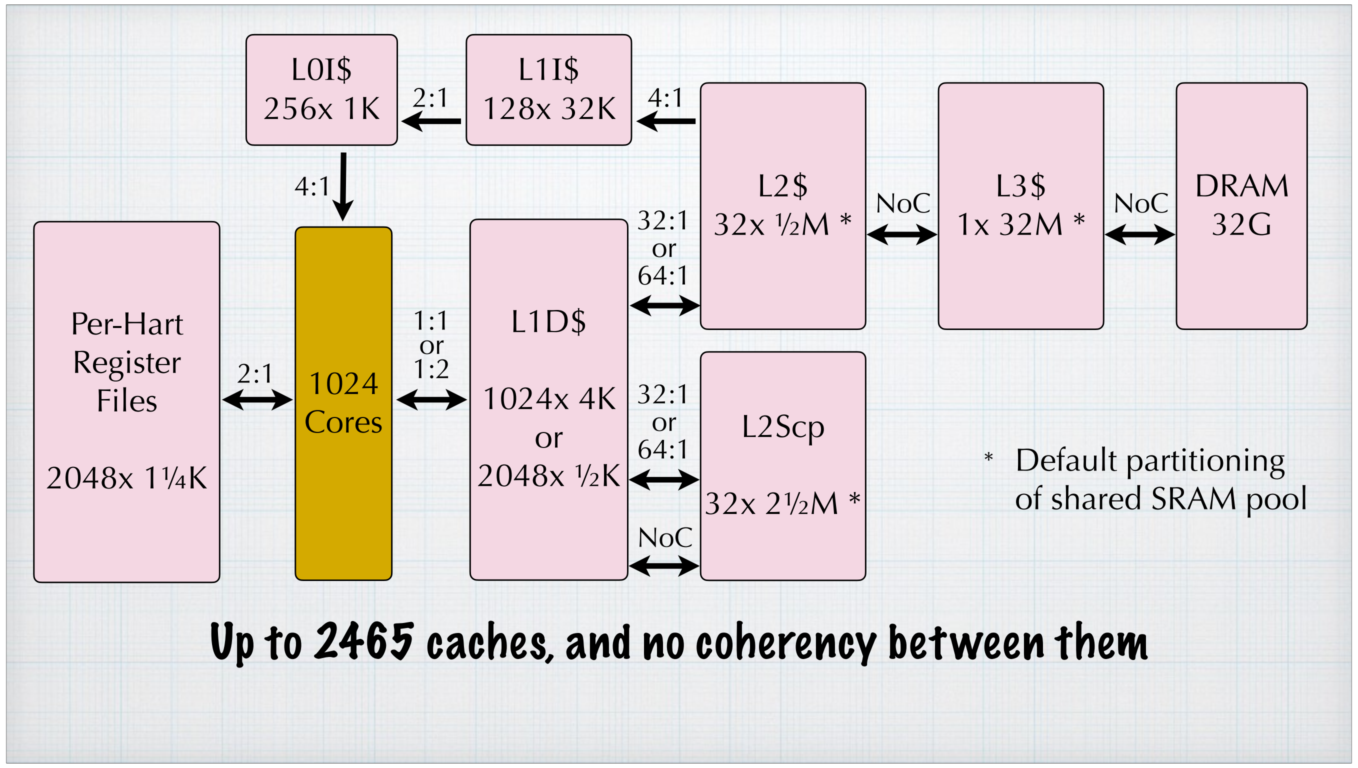
```
int i = csr_read(mhartid);
if (i < 512) {
  for (int j = 0; j < 512; ++j) {
    fp32 x = 0;
    for (int k = 0; k < 512; ++k) {
      x += a->x[i][k] * b->x[k][j];
    }
    c->x[i][j] = x;
  }
}
```

Correct result

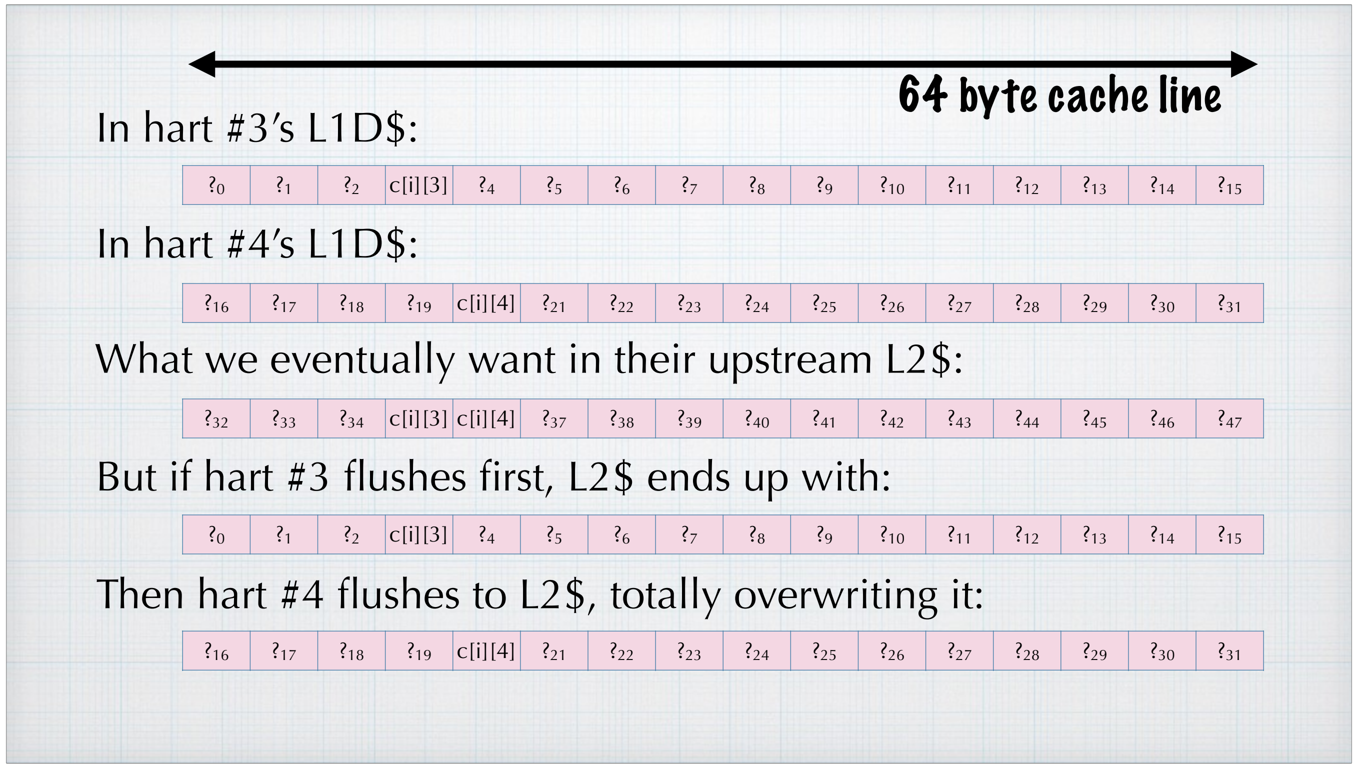
```
int j = csr_read(mhartid);
for (int i = 0; i < 512; ++i) {
  if (j < 512) {
    fp32 x = 0;
    for (int k = 0; k < 512; ++k) {
      x += a->x[i][k] * b->x[k][j];
    }
    c->x[i][j] = x;
  }
}
```

Wrong result
Eh?

Unfortunately, if I parallelise the inner loop, the code gives the wrong result.


L0I$ 256x 1K
L1I$ 128x 32K
2:1
4:1
4:1
L2$ 32x ½M *
NoC
L3$ 1x 32M *
NoC
DRAM 32G
Per-Hart Register Files 2048x 1¼K
2:1
1024 Cores
1:1 or 1:2
L1D$ 1024x 4K or 2048x ½K
32:1 or 64:1
32:1 or 64:1
NoC
L2Scp 32x 2½M *
* Default partitioning of shared SRAM pool
Up to 2465 caches, and no coherency between them

The reason for this lies in the memory hierarchy, which I've tried to show on this block diagram. We'll come back to this diagram later, so for now there's just one thing I want you to take from it: there are a _lot_ of caches on this chip, and there is no coherency between any of them. If you're coming from a GPU background, this'll be familiar to you, whereas if you're coming from a CPU background, it might be somewhat alarming.



64 byte cache line

In hart #3's L1D$:

$?_0$	$?_1$	$?_2$	c[i][3]	$?_4$	$?_5$	$?_6$	$?_7$	$?_8$	$?_9$	$?_{10}$	$?_{11}$	$?_{12}$	$?_{13}$	$?_{14}$	$?_{15}$

In hart #4's L1D$:

$?_{16}$	$?_{17}$	$?_{18}$	$?_{19}$	c[i][4]	$?_{21}$	$?_{22}$	$?_{23}$	$?_{24}$	$?_{25}$	$?_{26}$	$?_{27}$	$?_{28}$	$?_{29}$	$?_{30}$	$?_{31}$

What we eventually want in their upstream L2$:

$?_{32}$	$?_{33}$	$?_{34}$	c[i][3]	c[i][4]	$?_{37}$	$?_{38}$	$?_{39}$	$?_{40}$	$?_{41}$	$?_{42}$	$?_{43}$	$?_{44}$	$?_{45}$	$?_{46}$	$?_{47}$

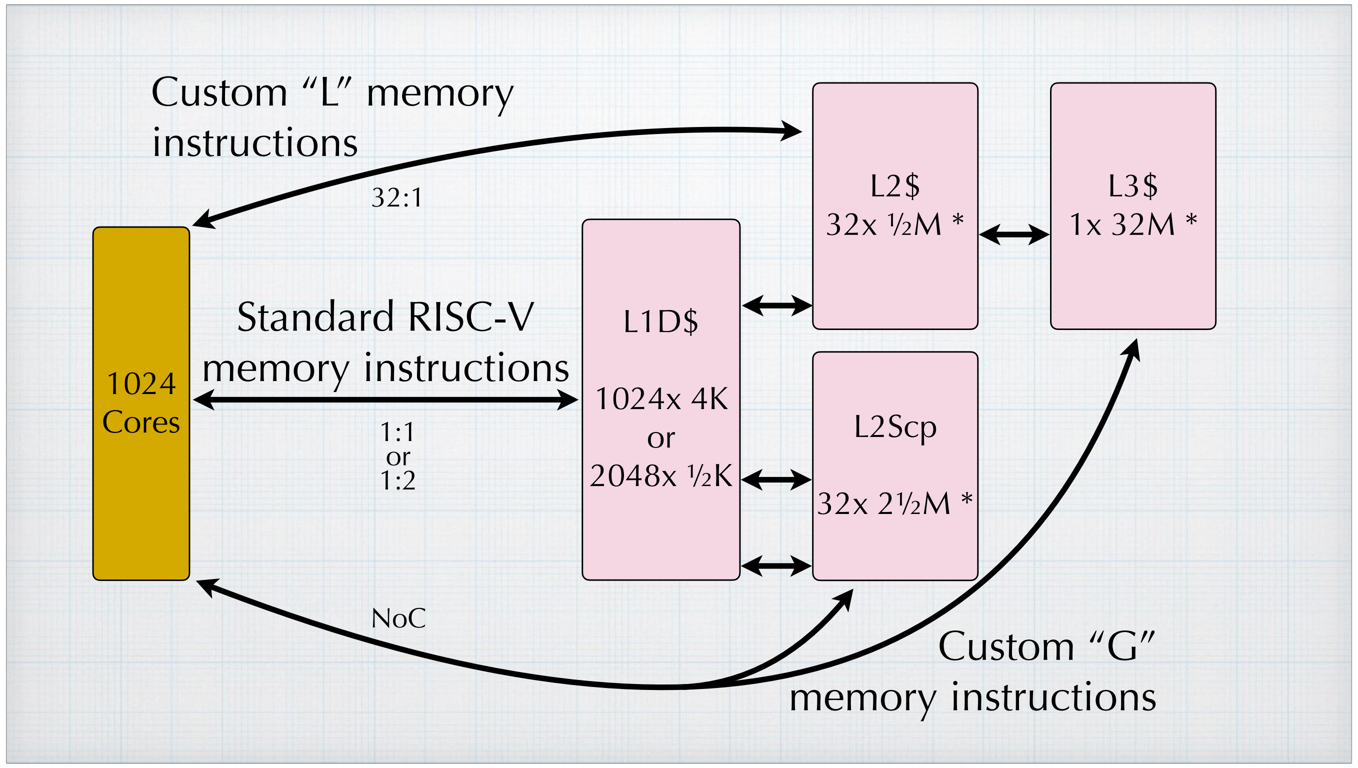
But if hart #3 flushes first, L2$ ends up with:

$?_0$	$?_1$	$?_2$	c[i][3]	$?_4$	$?_5$	$?_6$	$?_7$	$?_8$	$?_9$	$?_{10}$	$?_{11}$	$?_{12}$	$?_{13}$	$?_{14}$	$?_{15}$

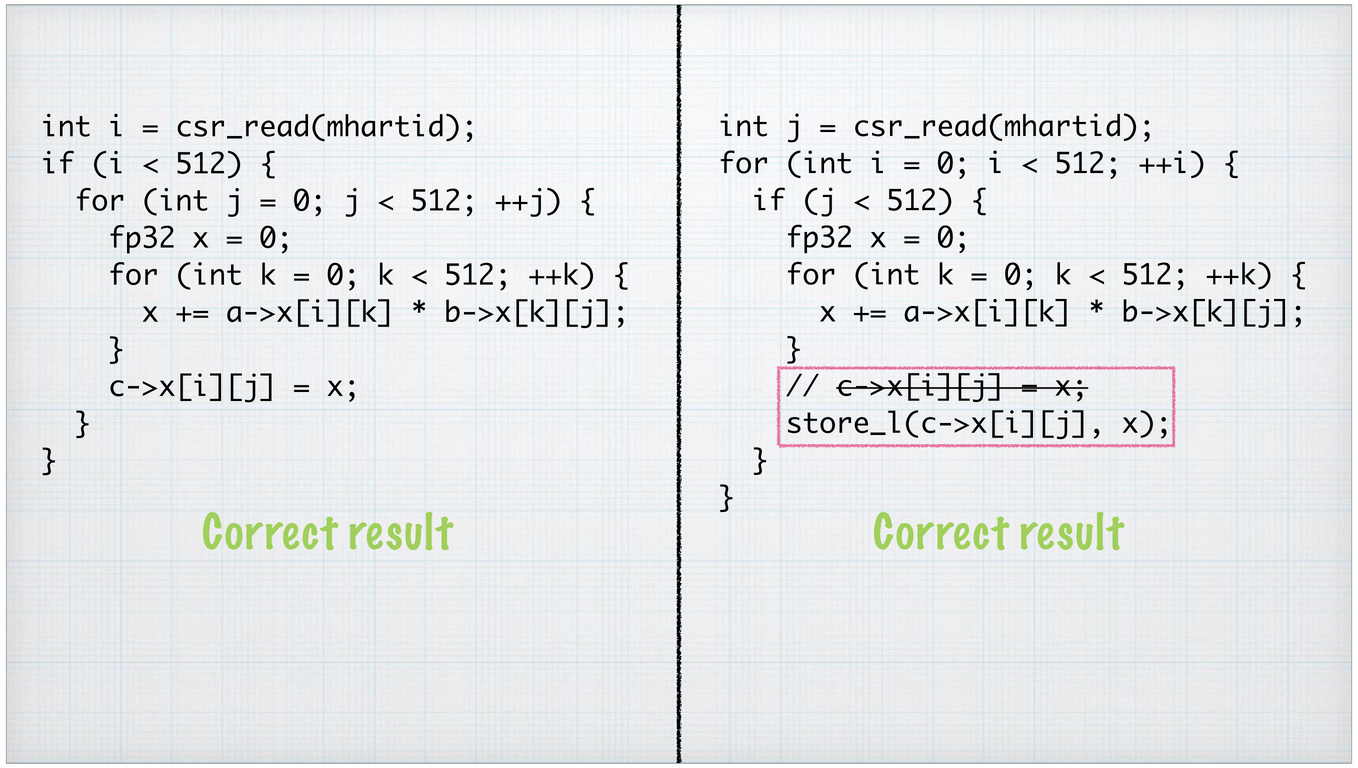
Then hart #4 flushes to L2$, totally overwriting it:

$?_{16}$	$?_{17}$	$?_{18}$	$?_{19}$	c[i][4]	$?_{21}$	$?_{22}$	$?_{23}$	$?_{24}$	$?_{25}$	$?_{26}$	$?_{27}$	$?_{28}$	$?_{29}$	$?_{30}$	$?_{31}$

To see why lack of coherency is a problem, we need to talk about cache lines. On this chip, cache lines are 64 bytes wide, meaning that when caches talk to each other, they do so in units of 64 bytes. If we think about just hart #3 and hart #4 when they write to the c matrix, they each perform a 4-byte write to their local L1D$, but the L1D$s have no awareness of each other, and then (at some point later) they'll each flush the entire 64 byte line to L2$ rather than only the modified part, so the 2nd flush completely overwrites the 1st flush.


Custom "L" memory instructions
32:1
L2$
32x ½M *
L3$
1x 32M *
Standard RISC-V memory instructions
1024 Cores
1:1 or 1:2
L1D$
1024x 4K or 2048x ½K
L2Scp
32x 2½M *
NoC
Custom "G" memory instructions

Thankfully, software can choose which cache to use on an instruction-by-instruction basis. Standard RISC-V memory instructions go to the core-local or hart-local L1D$, whereas custom "L" instructions to go the L2$ nearest the issuing core. Custom "G" instructions look more complex, but their effect is easy to understand: if you use them for everything, you'll have the appearance of coherency, as the cache will be chosen based on the memory address and not the issuing core, but this does come at a latency and bandwidth cost. I'm using a standard RISC-V compiler meant for CPUs, so memory accesses use standard RISC-V instructions by default, and thus go to L1D$.



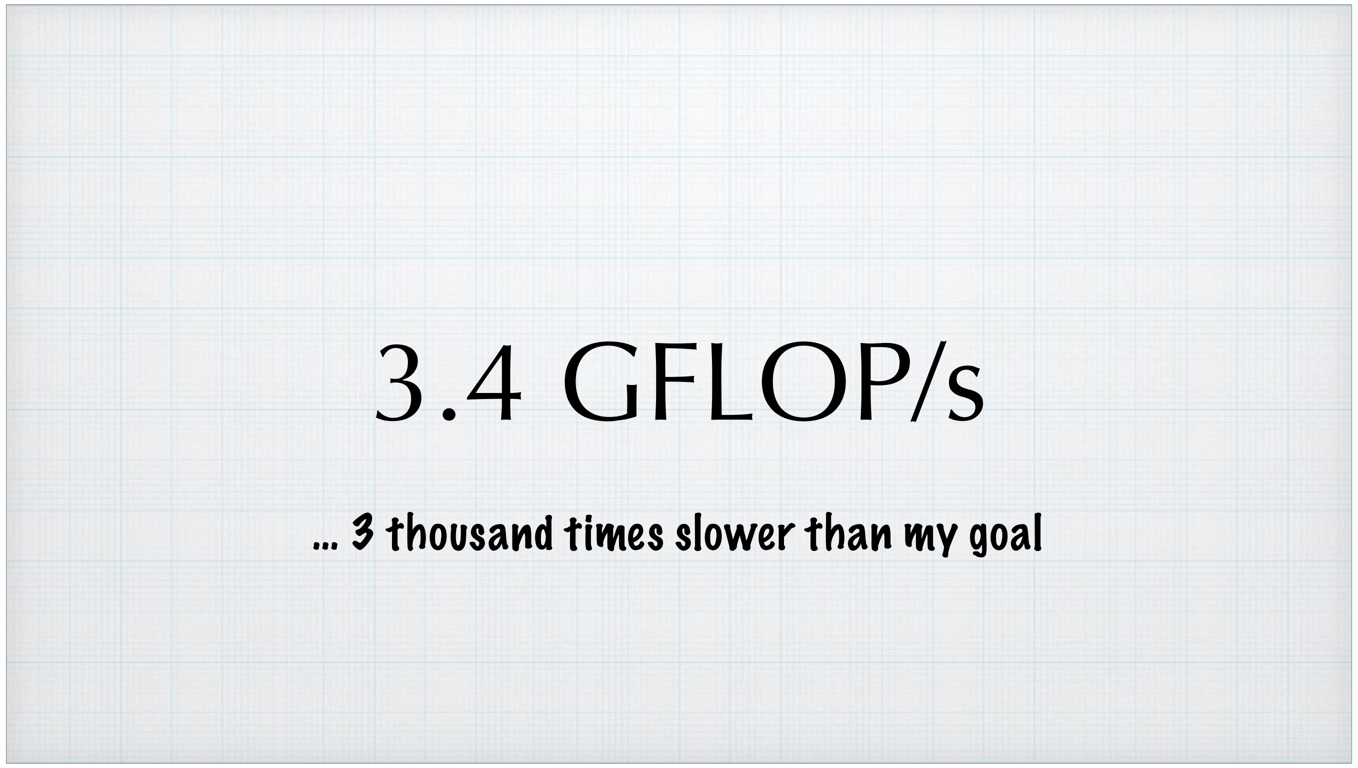
```
int i = csr_read(mhartid);
if (i < 512) {
  for (int j = 0; j < 512; ++j) {
    fp32 x = 0;
    for (int k = 0; k < 512; ++k) {
      x += a->x[i][k] * b->x[k][j];
    }
    c->x[i][j] = x;
  }
}
```

Correct result

```
int j = csr_read(mhartid);
for (int i = 0; i < 512; ++i) {
  if (j < 512) {
    fp32 x = 0;
    for (int k = 0; k < 512; ++k) {
      x += a->x[i][k] * b->x[k][j];
    }
    // ~~c->x[i][j] = x;~~
    store_l(c->x[i][j], x);
  }
}
```

Correct result

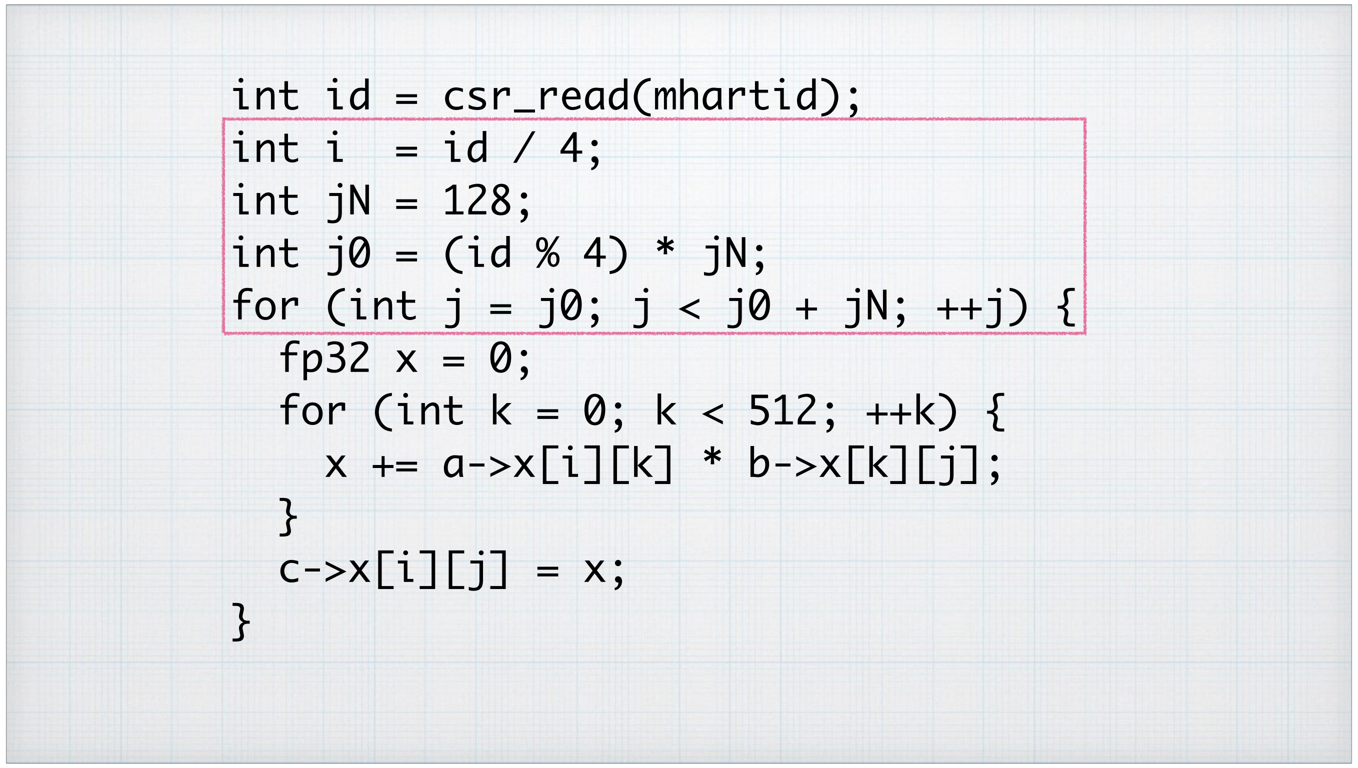
I could fix things by using custom “G” instructions, but the particular access pattern means that “L” instructions also work in this case, which is what I’ve gone with. If I was using a compiler which understood the hardware more, perhaps every pointer type would indicate what kind of instruction to use, but I’m not using a clever enough compiler.



3.4 GFLOP/s

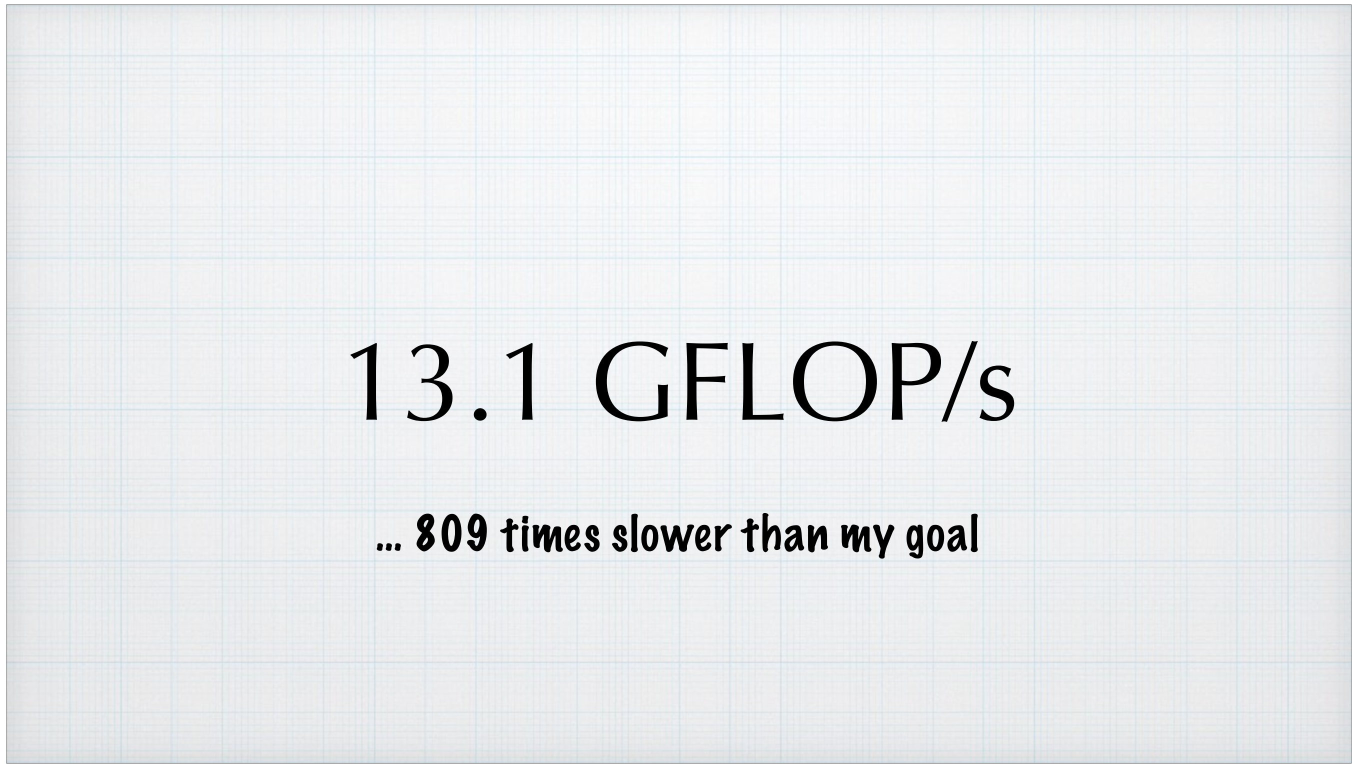
… 3 thousand times slower than my goal

Both versions of the code now give the correct result, and the faster of the two runs at 3.4 gigaflops.



```
int id = csr_read(mhartid);
int i  = id / 4;
int jN = 128;
int j0 = (id % 4) * jN;
for (int j = j0; j < j0 + jN; ++j) {
  fp32 x = 0;
  for (int k = 0; k < 512; ++k) {
    x += a->x[i][k] * b->x[k][j];
  }
  c->x[i][j] = x;
}
```

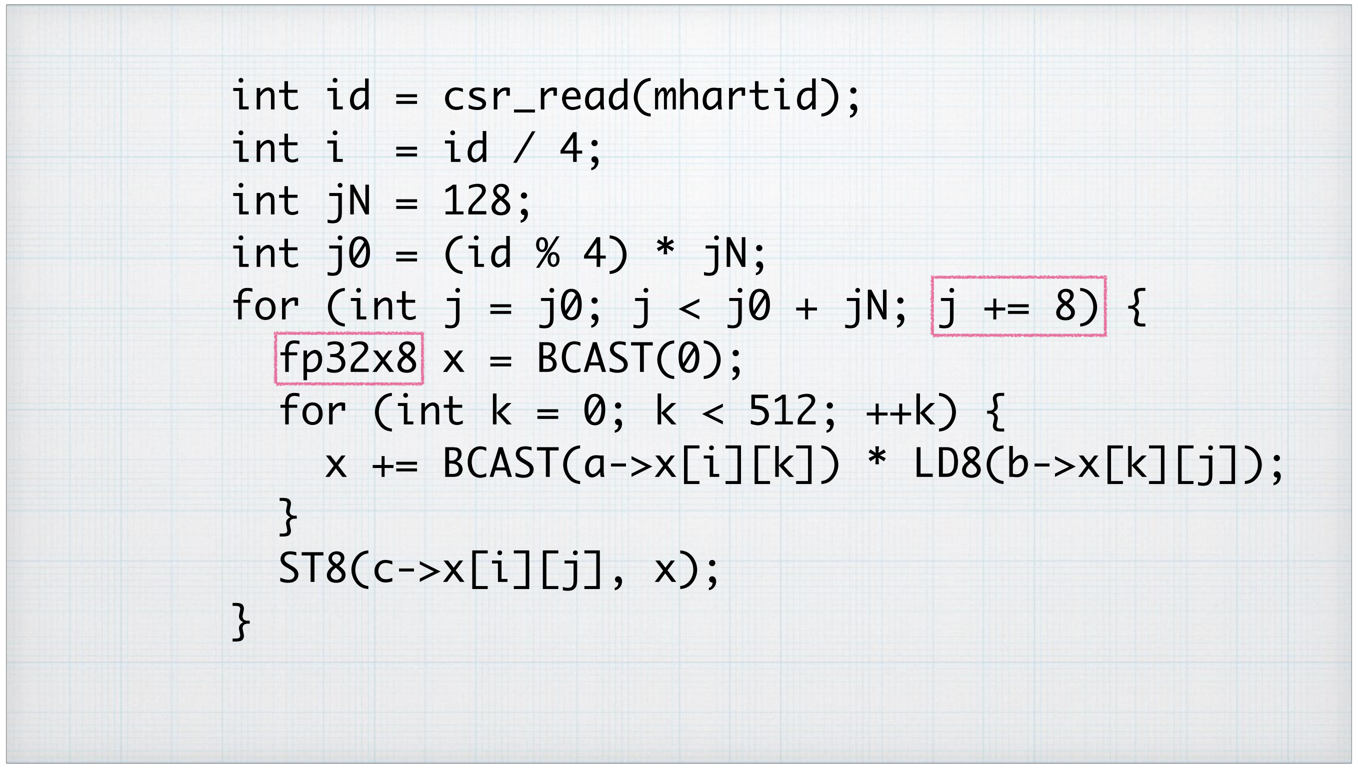
The next easy win is to use all 2048 harts rather than just the first 512. The outer i loop is fully parallelised, and then each hart runs one quarter of the inner j loop. The changed code is in the pink rectangle; everything else is as before.



13.1 GFLOP/s

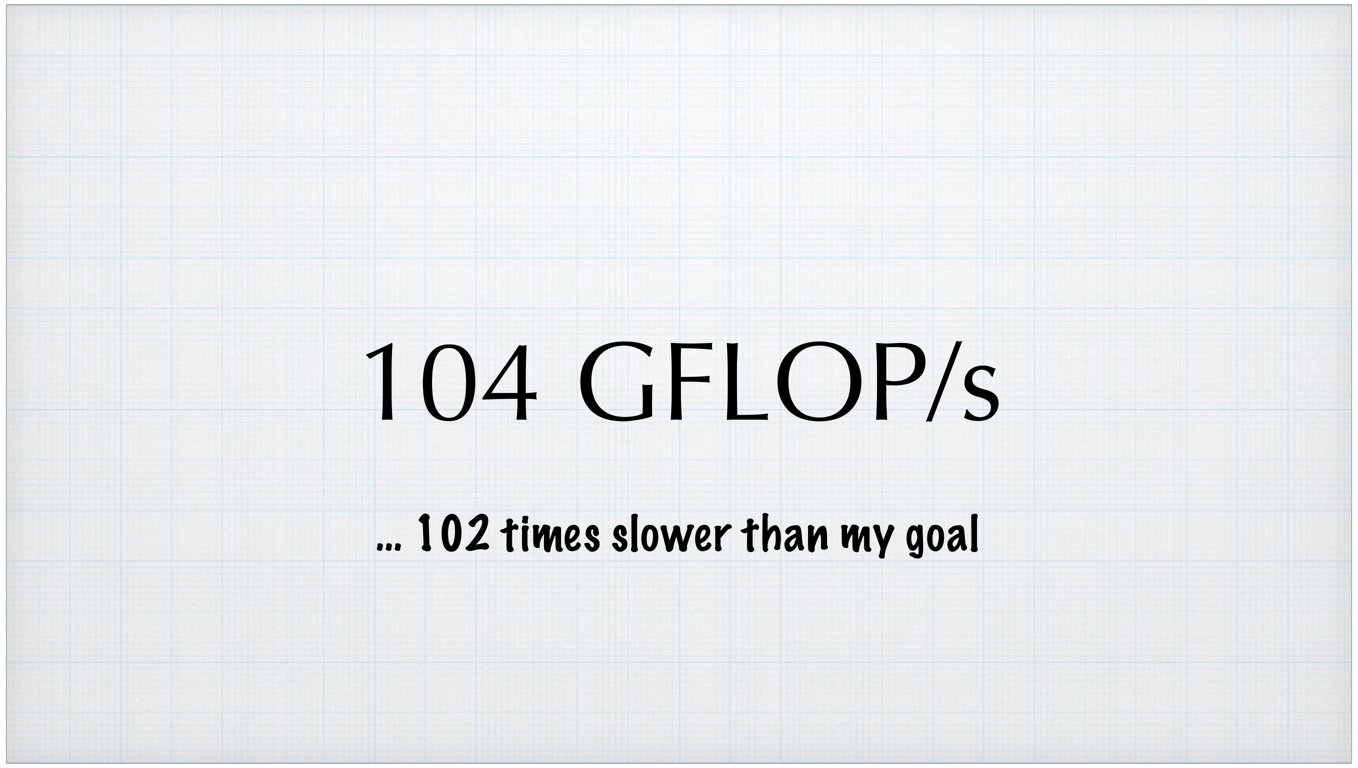
… 809 times slower than my goal

Using four times as many harts gives roughly a 4x speedup, putting me at 13.1 gigaflops.



```
int id = csr_read(mhartid);
int i  = id / 4;
int jN = 128;
int j0 = (id % 4) * jN;
for (int j = j0; j < j0 + jN; j += 8) {
  fp32x8 x = BCAST(0);
  for (int k = 0; k < 512; ++k) {
    x += BCAST(a->x[i][k]) * LD8(b->x[k][j]);
  }
  ST8(c->x[i][j], x);
}
```

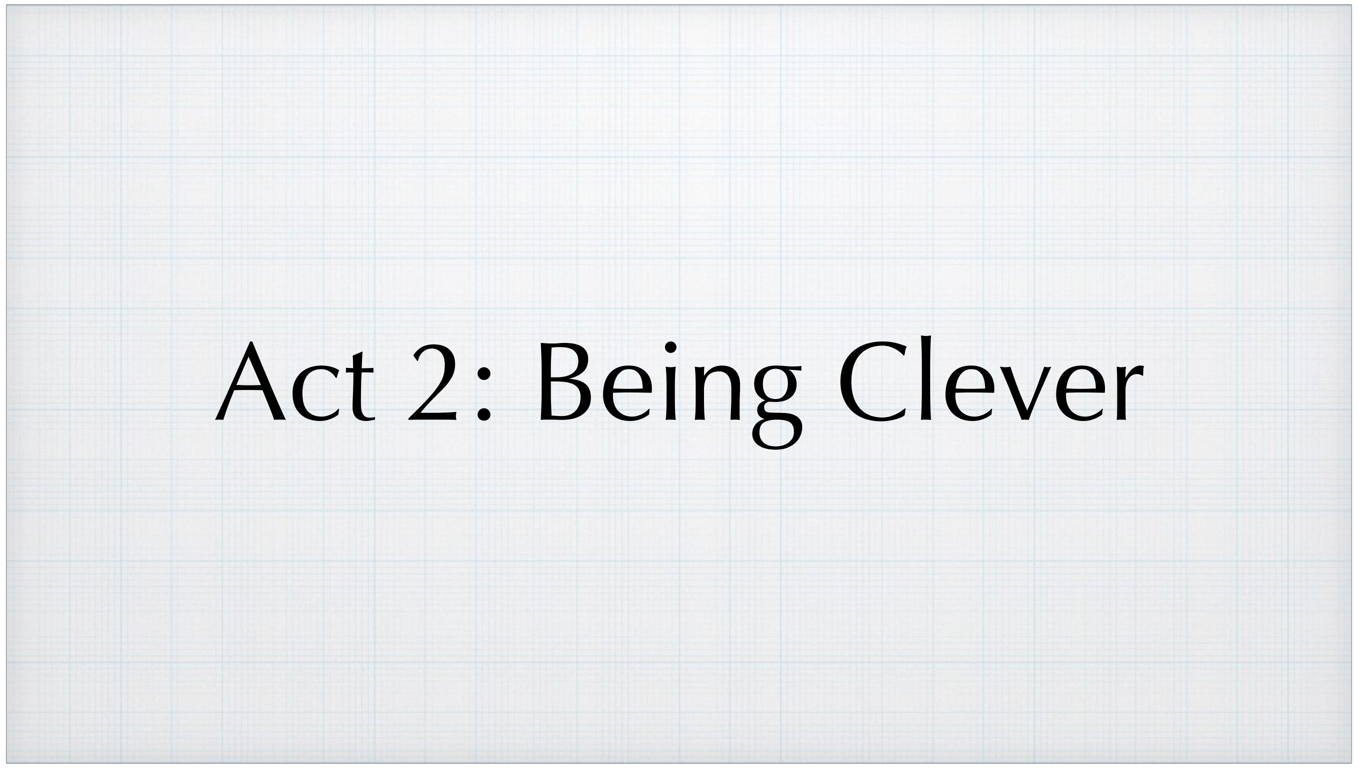
The final easy win is using all eight vector lanes. I’m going to pretend that the C compiler is a little bit better than it actually is, and that doing 8-wide SIMD just requires changing fp32 to fp32x8. Pending that compiler work, I’m assembling this code by hand (but if any LLVM hackers want a little project, please make this slide possible).



104 GFLOP/s

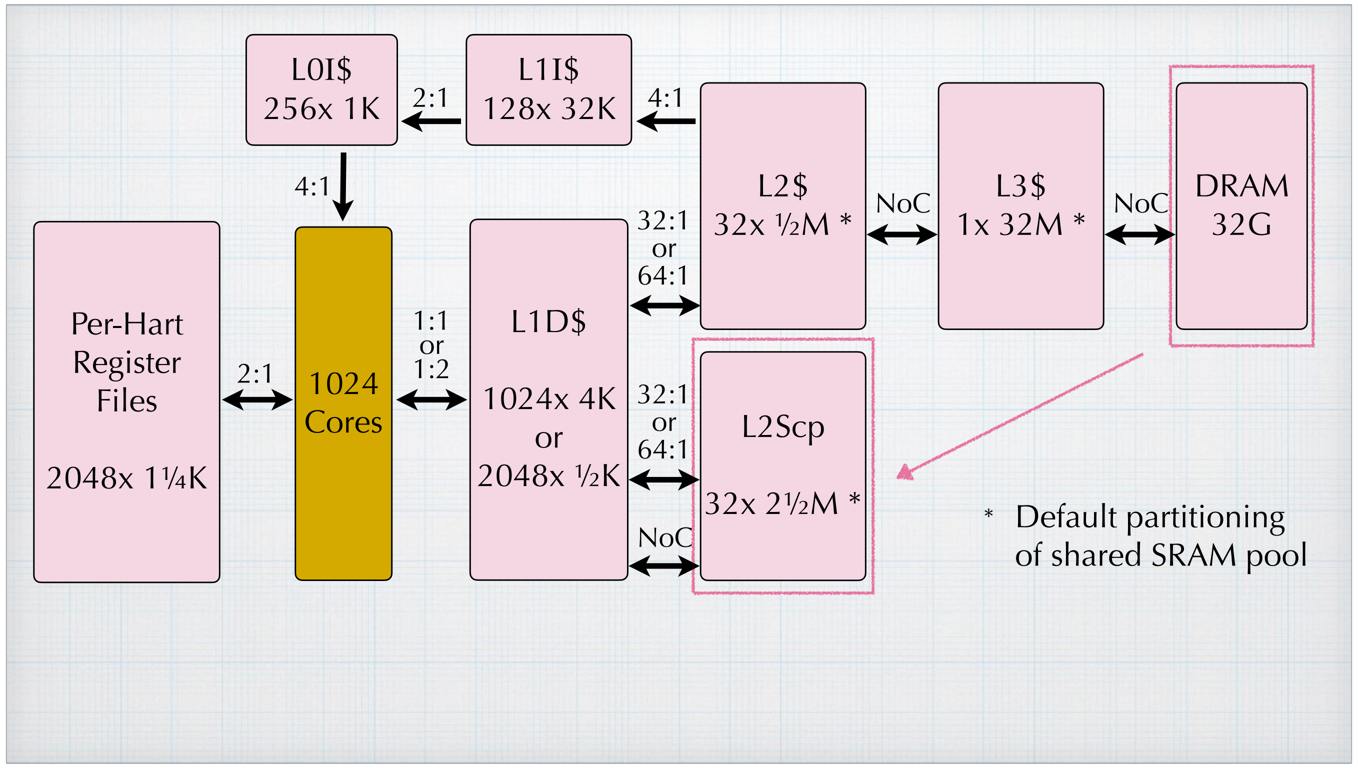
… 102 times slower than my goal

Using all eight vector lanes gives a factor 8 speedup, breaking through the 100 gigaflops barrier.



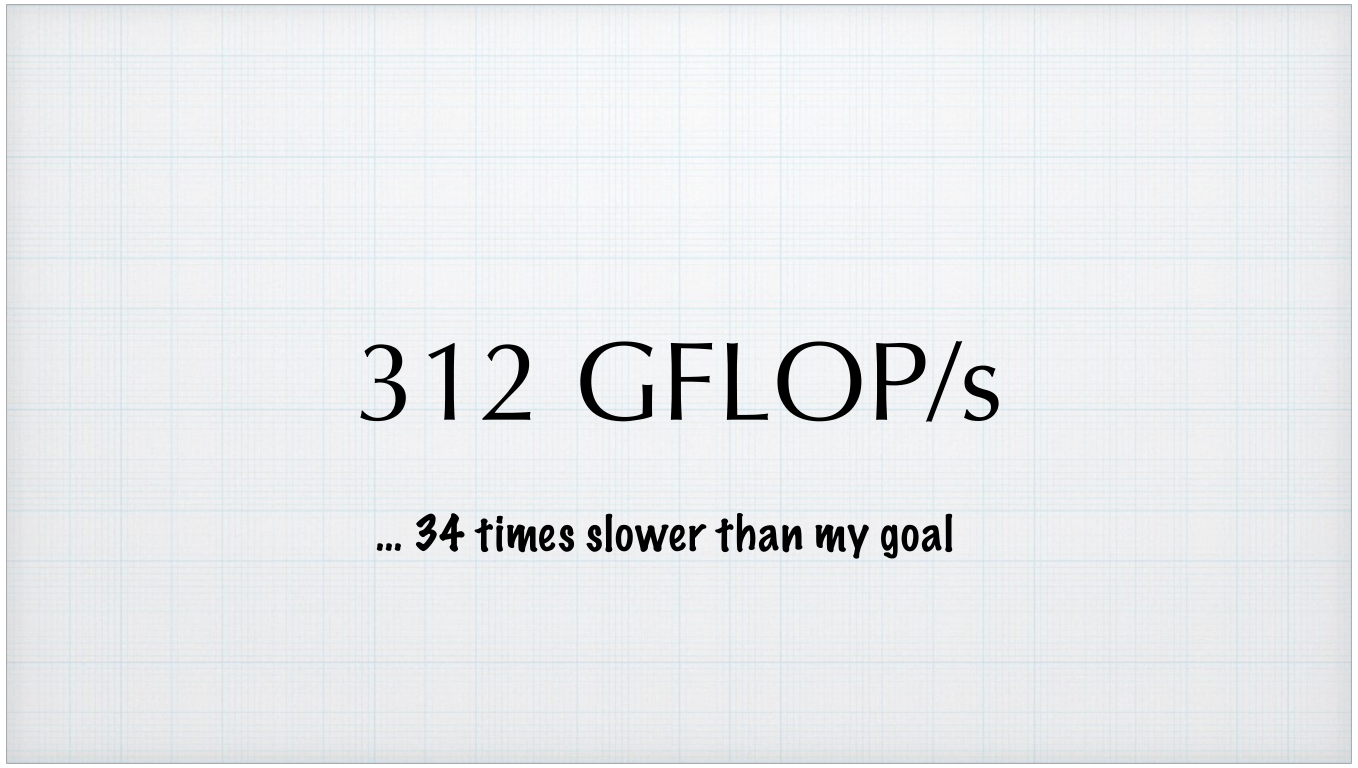
Act 2: Being Clever

[Time guide: T+8m30s]



This is the memory hierarchy diagram from before. There's a star on the capacities of L2$, L3$, and L2Scp, because there's a single pool of SRAM backing all three, with configurable partitioning, and the capacities shown here are the default split. Of the three, L2Scp is really interesting, as any SRAM assigned to it behaves like plain addressable memory rather than like cache, but with the same latency as L2$. Every core will have a nearest L2Scp just like it has a nearest L2$, but any core can access any L2Scp, it'll just be higher latency (I'm not making use of that particular functionality, but it is nice to have).

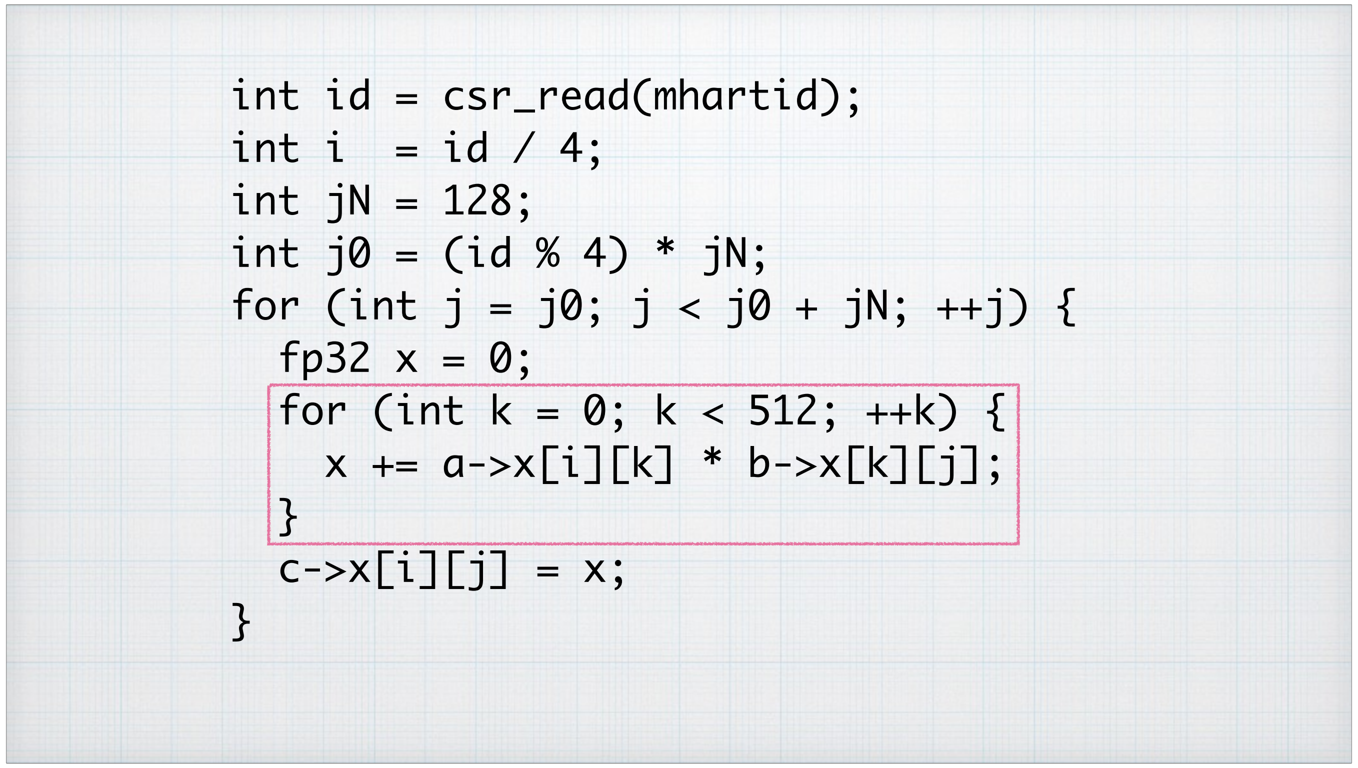
Up until now I've put the input a and b matrices in DRAM, but I can instead put them in every L2Scp. A proper implementation would need to intelligently stream data into L2Scp just in time, but I'm drawing circles rather than complete owls, so I can pre-position the data into every L2Scp and call it good.



312 GFLOP/s

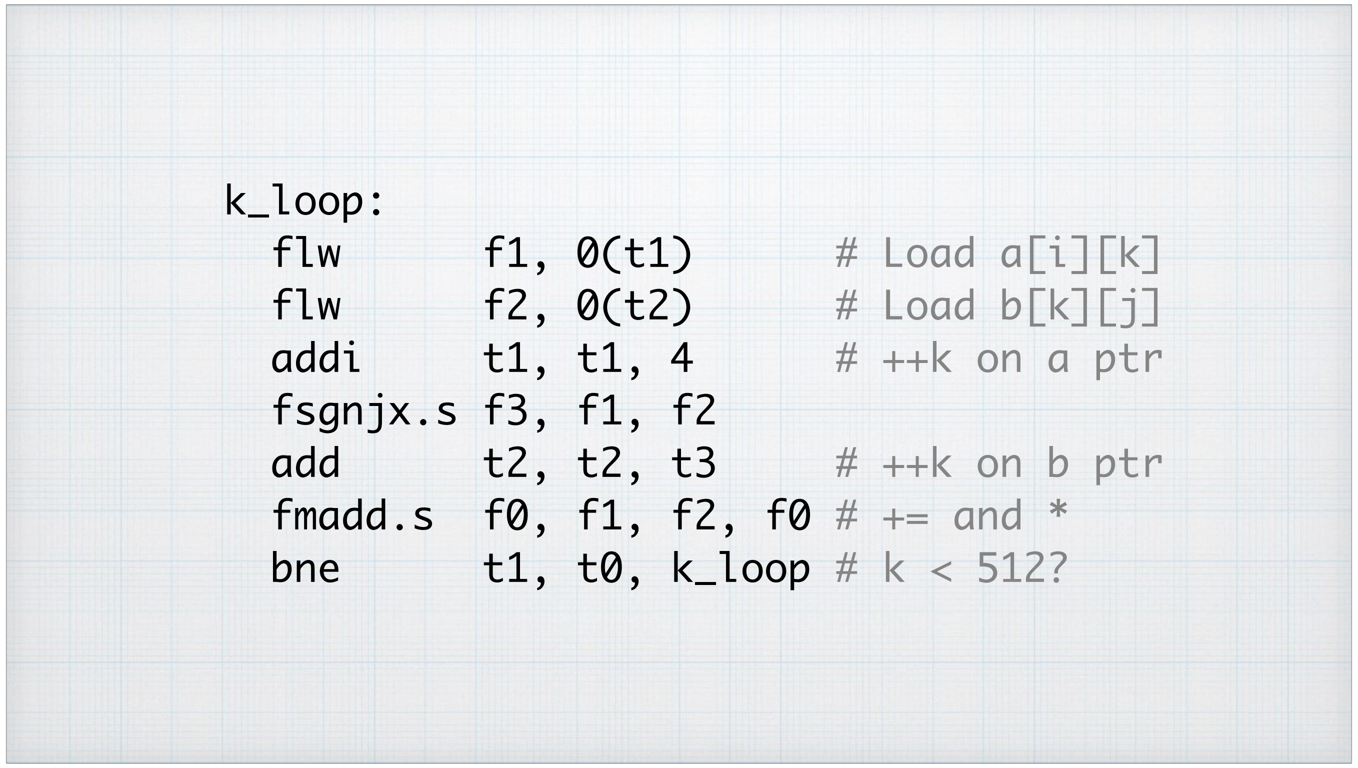
… 34 times slower than my goal

Reading from L2Scp gives a 3x speedup, so we’re now at 312 gigaflops.



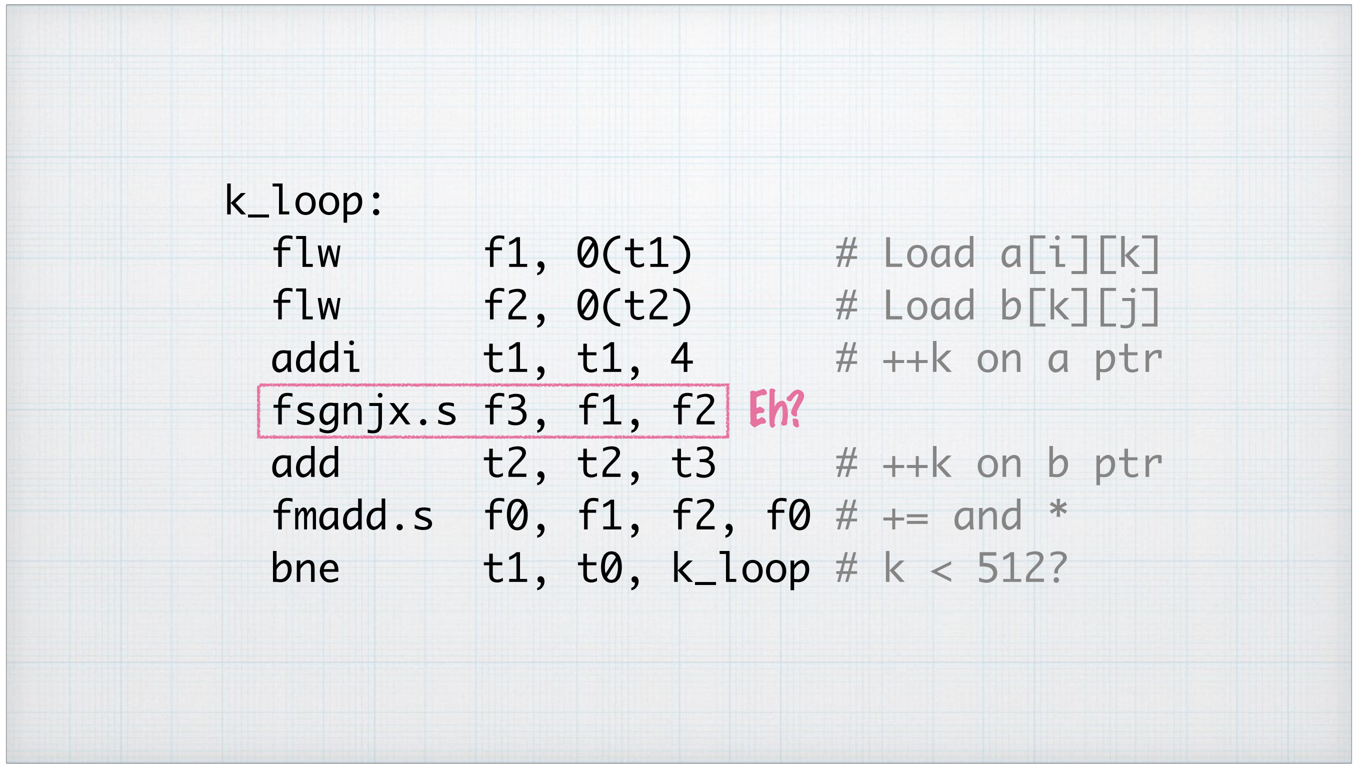
```
int id = csr_read(mhartid);
int i  = id / 4;
int jN = 128;
int j0 = (id % 4) * jN;
for (int j = j0; j < j0 + jN; ++j) {
  fp32 x = 0;
  for (int k = 0; k < 512; ++k) {
    x += a->x[i][k] * b->x[k][j];
  }
  c->x[i][j] = x;
}
```

At this point I want to look at the inner loop of the matmul in detail, and to keep things simple, I’m going backwards one step to the pre-SIMD code.



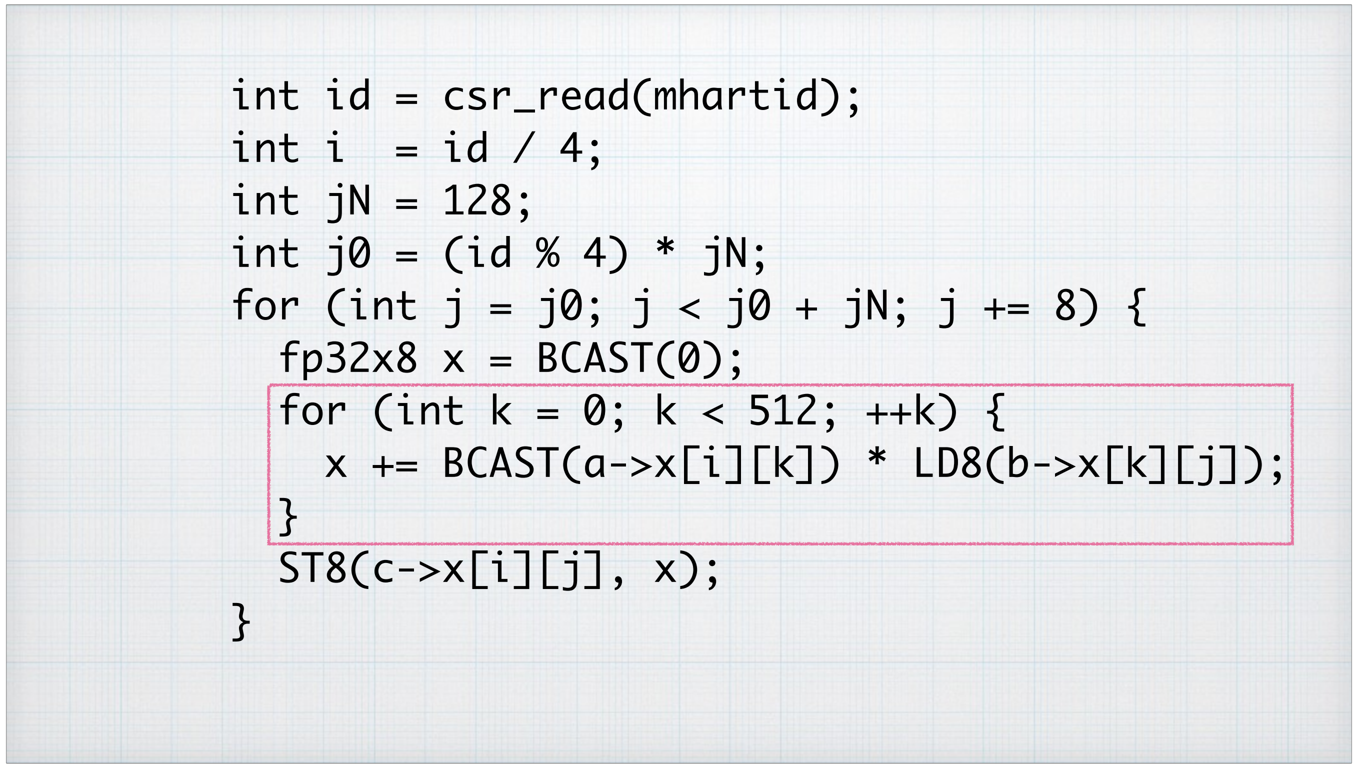
```
k_loop:
  flw      f1, 0(t1)      # Load a[i][k]
  flw      f2, 0(t2)      # Load b[k][j]
  addi     t1, t1, 4      # ++k on a ptr
  fsgnjx.s f3, f1, f2
  add      t2, t2, t3     # ++k on b ptr
  fmadd.s  f0, f1, f2, f0 # += and *
  bne      t1, t0, k_loop # k < 512?
```

This is the RISC-V assembly corresponding to that loop, with comments added for the benefit of anyone whose first language is not assembly.



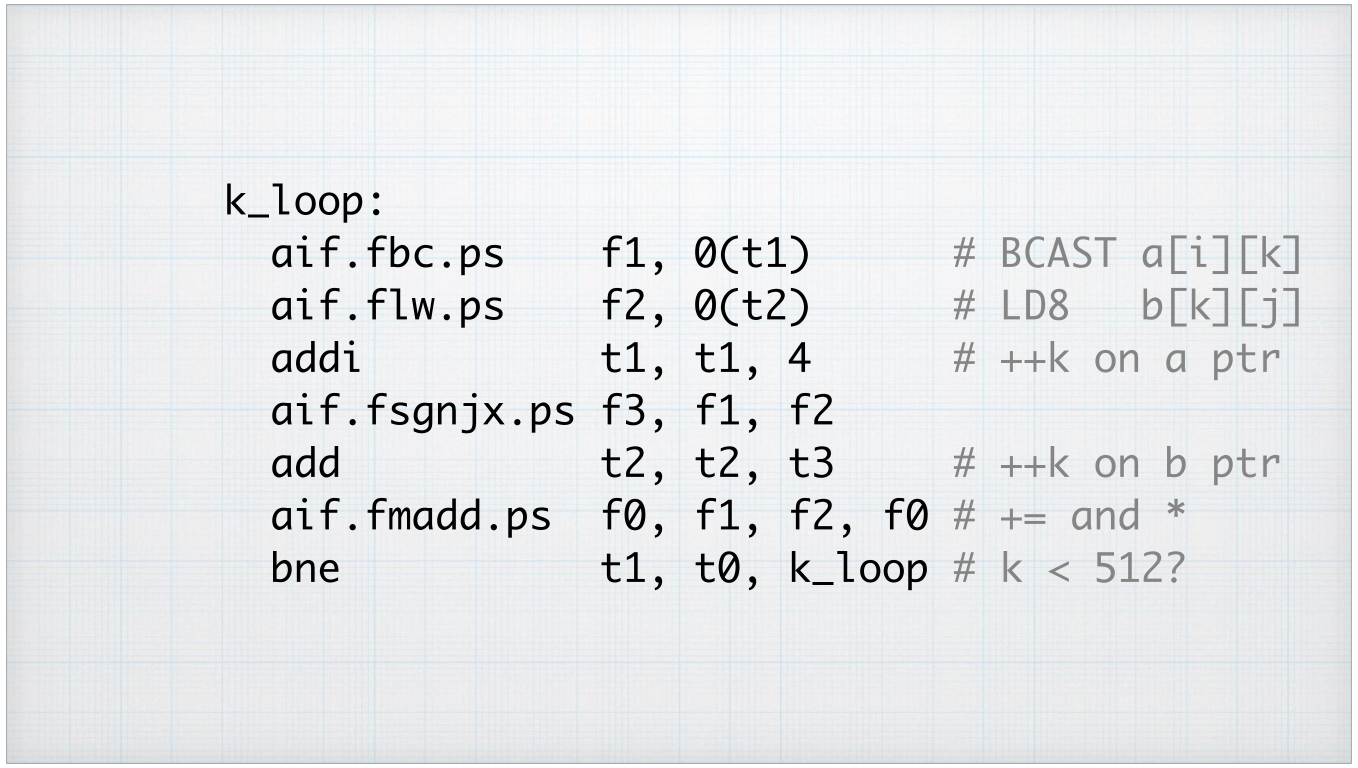
```
k_loop:
  flw      f1, 0(t1)      # Load a[i][k]
  flw      f2, 0(t2)      # Load b[k][j]
  addi     t1, t1, 4      # ++k on a ptr
  fsgnjx.s f3, f1, f2 Eh?
  add      t2, t2, t3     # ++k on b ptr
  fmadd.s  f0, f1, f2, f0 # += and *
  bne      t1, t0, k_loop # k < 512?
```

If you've got a keen eye for RISC-V assembly, the highlighted instruction will stand out as coming out of nowhere and being seemingly useless, but it is actually present to work around a hardware bug. The PCIe board I'm playing with is A0 silicon, and any hardware people in the room will likely acknowledge that A0 silicon always has some bugs in it, which can require software workarounds. For any software people in the room, A0 silicon is like version 0 of a piece of software. It's the first version you send to (say) TSMC for printing, and you hope it'll be bug free, but there are always surprises.



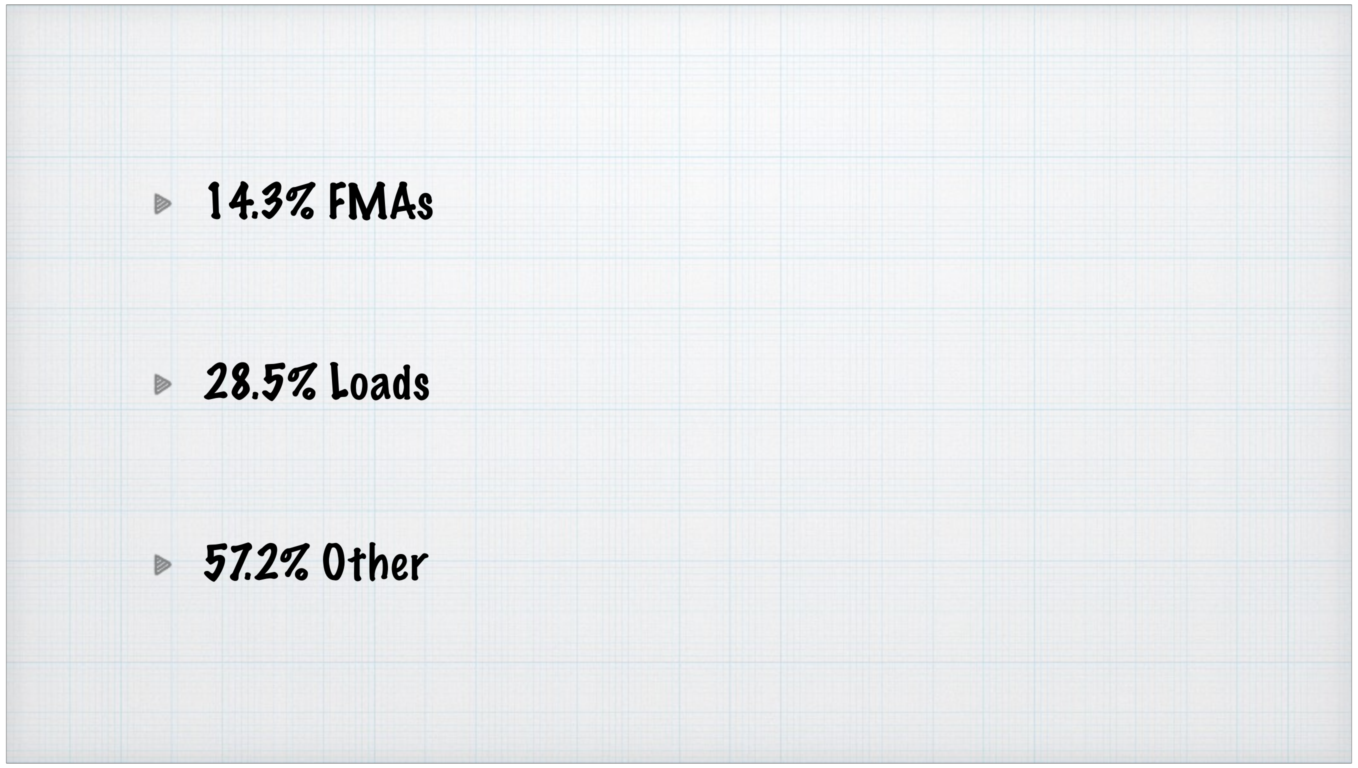
```
int id = csr_read(mhartid);
int i  = id / 4;
int jN = 128;
int j0 = (id % 4) * jN;
for (int j = j0; j < j0 + jN; j += 8) {
  fp32x8 x = BCAST(0);
  for (int k = 0; k < 512; ++k) {
    x += BCAST(a->x[i][k]) * LD8(b->x[k][j]);
  }
  ST8(c->x[i][j], x);
}
```

Now that you've got over the assembly shock, I can return to the SIMD version of the code. Again, this inner loop is the interesting part.



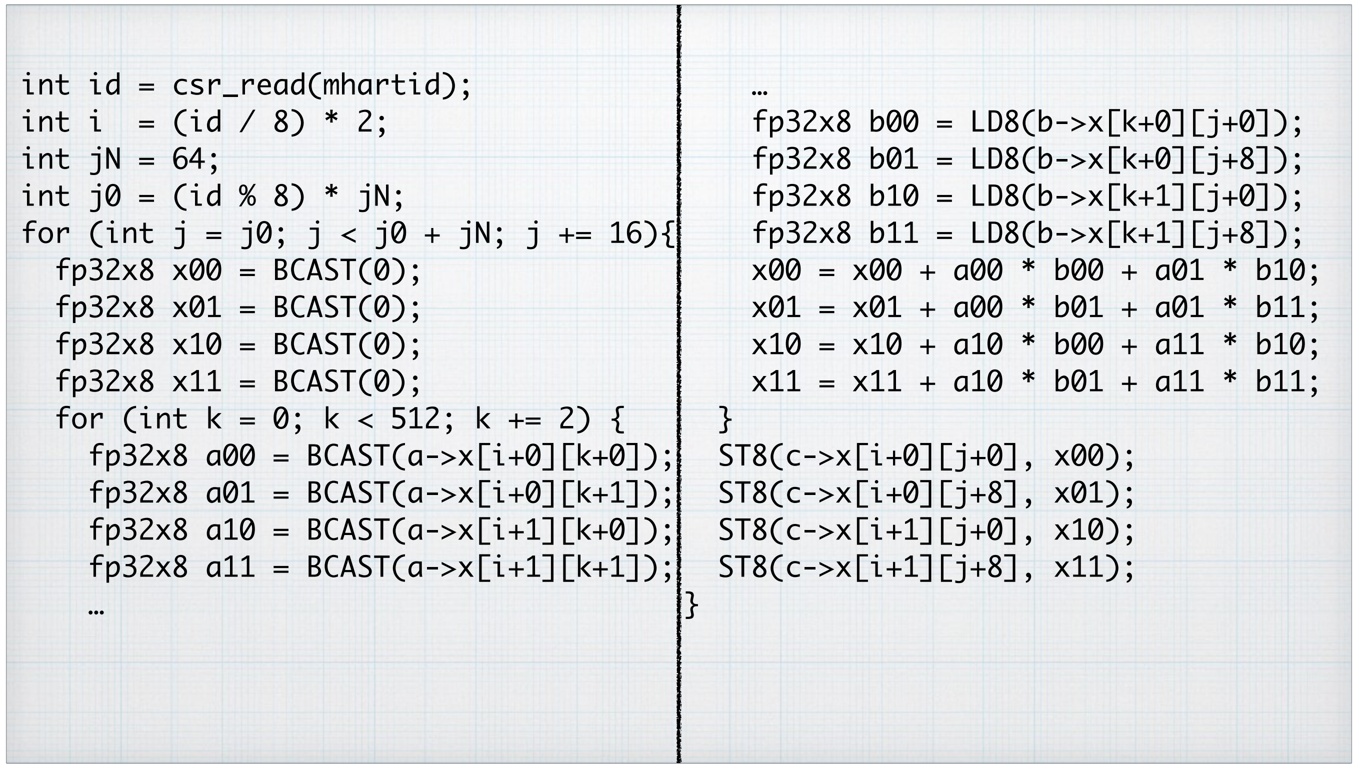
```
k_loop:
  aif.fbc.ps    f1, 0(t1)      # BCAST a[i][k]
  aif.flw.ps    f2, 0(t2)      # LD8   b[k][j]
  addi          t1, t1, 4      # ++k on a ptr
  aif.fsgnjx.ps f3, f1, f2
  add           t2, t2, t3     # ++k on b ptr
  aif.fmadd.ps  f0, f1, f2, f0 # += and *
  bne           t1, t0, k_loop # k < 512?
```

This is the assembly for the SIMD version. It’s very similar to the previous assembly, just using custom ps instructions rather than standard ones.



- 14.3% FMAs
- 28.5% Loads
- 57.2% Other

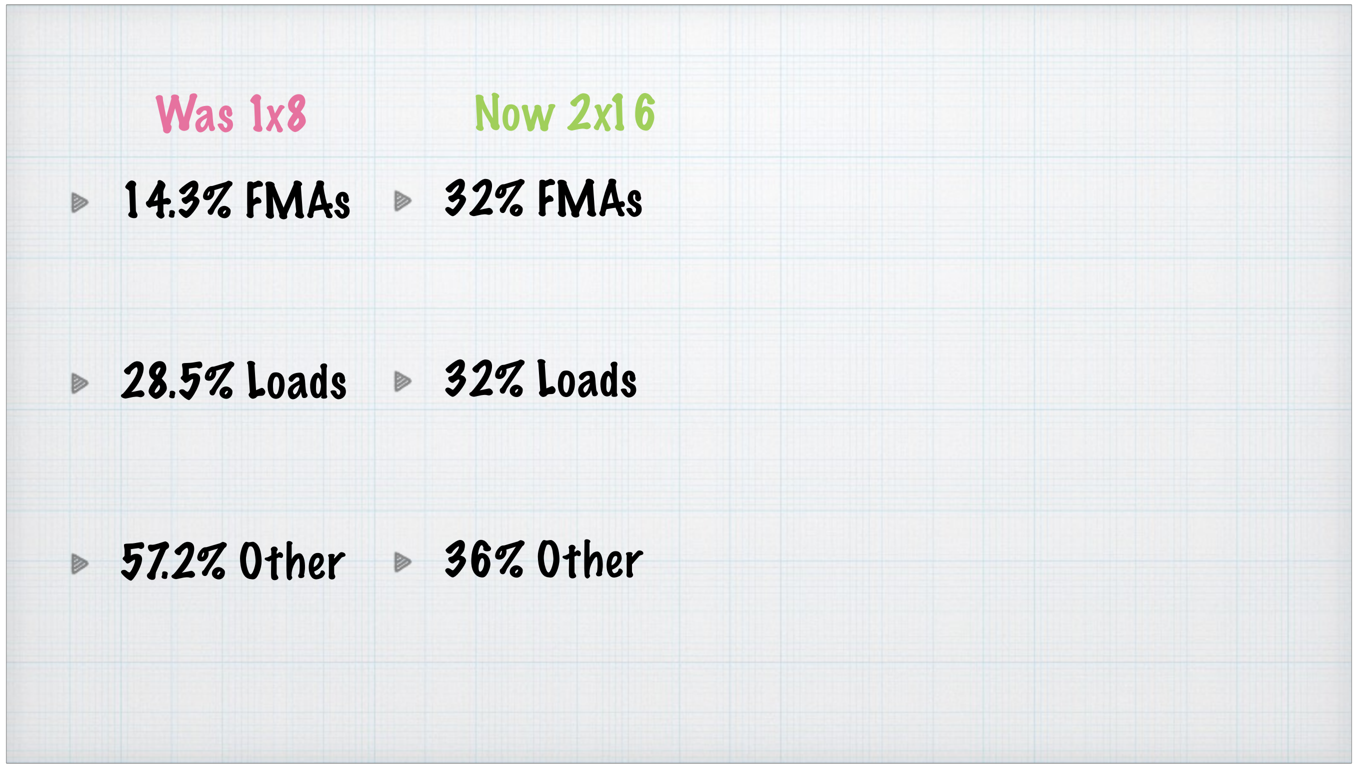
Regardless of which version we look at, only 14% of instructions are FMAs. Percentages don't tell the whole story, but it's fairly clear that the FMA percentage is too low and the other percentage is too high. One way of fixing this is to do more work per outer loop iteration. Four times as much work is a sweet spot: rather than computing a 1x8 piece of c, double up on both axes to compute a 2x16 piece of c.



```
int id = csr_read(mhartid);
int i  = (id / 8) * 2;
int jN = 64;
int j0 = (id % 8) * jN;
for (int j = j0; j < j0 + jN; j += 16){
  fp32x8 x00 = BCAST(0);
  fp32x8 x01 = BCAST(0);
  fp32x8 x10 = BCAST(0);
  fp32x8 x11 = BCAST(0);
  for (int k = 0; k < 512; k += 2) {
    fp32x8 a00 = BCAST(a->x[i+0][k+0]);
    fp32x8 a01 = BCAST(a->x[i+0][k+1]);
    fp32x8 a10 = BCAST(a->x[i+1][k+0]);
    fp32x8 a11 = BCAST(a->x[i+1][k+1]);
    …
```

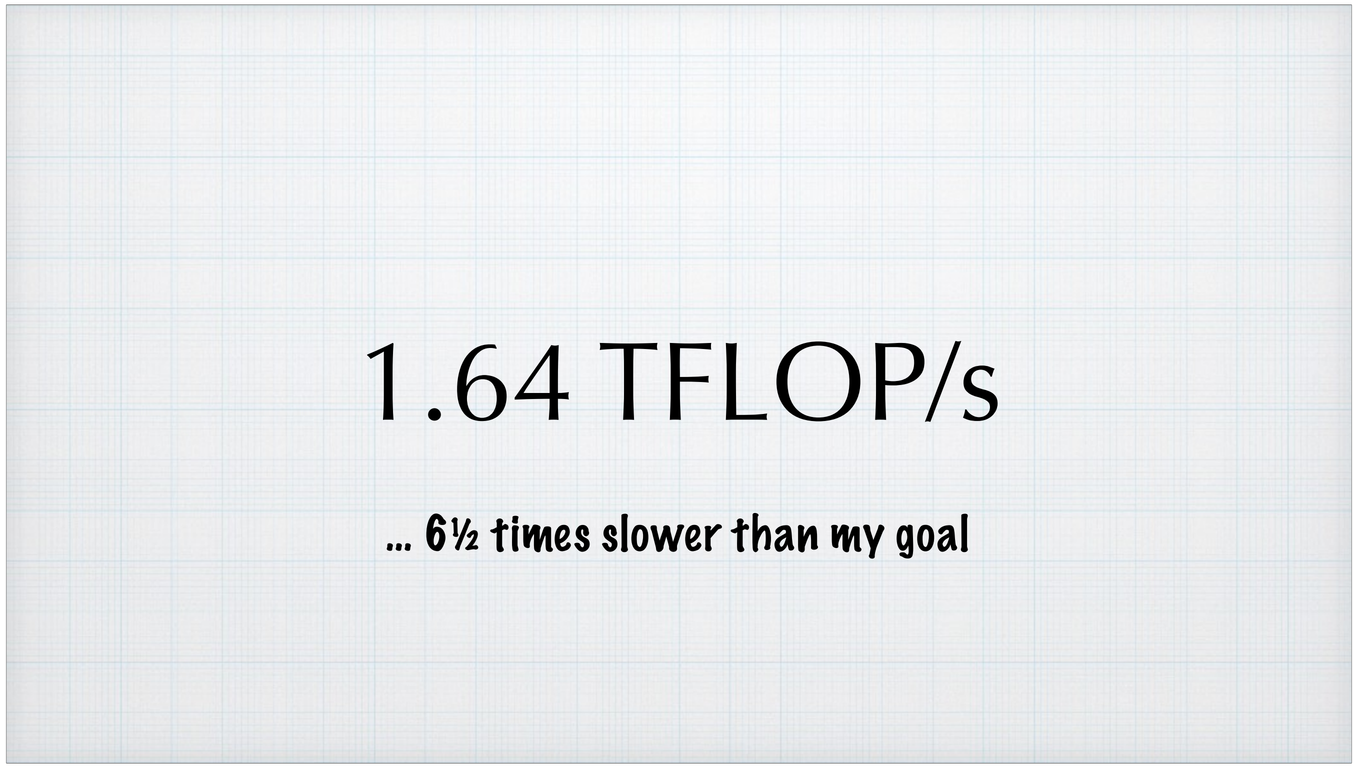
```
    …
    fp32x8 b00 = LD8(b->x[k+0][j+0]);
    fp32x8 b01 = LD8(b->x[k+0][j+8]);
    fp32x8 b10 = LD8(b->x[k+1][j+0]);
    fp32x8 b11 = LD8(b->x[k+1][j+8]);
    x00 = x00 + a00 * b00 + a01 * b10;
    x01 = x01 + a00 * b01 + a01 * b11;
    x10 = x10 + a10 * b00 + a11 * b10;
    x11 = x11 + a10 * b01 + a11 * b11;
  }
  ST8(c->x[i+0][j+0], x00);
  ST8(c->x[i+0][j+8], x01);
  ST8(c->x[i+1][j+0], x10);
  ST8(c->x[i+1][j+8], x11);
}
```

The code now looks like this. It is doing four times as much work per outer loop iteration, and you can clearly see that: four lines to initialise x variables, four BCAST loads from a, four LD8 loads from b, four lines of FMAs, and four ST8 stores to c.



Was 1x8	Now 2x16
14.3% FMAs	32% FMAs
28.5% Loads	32% Loads
57.2% Other	36% Other

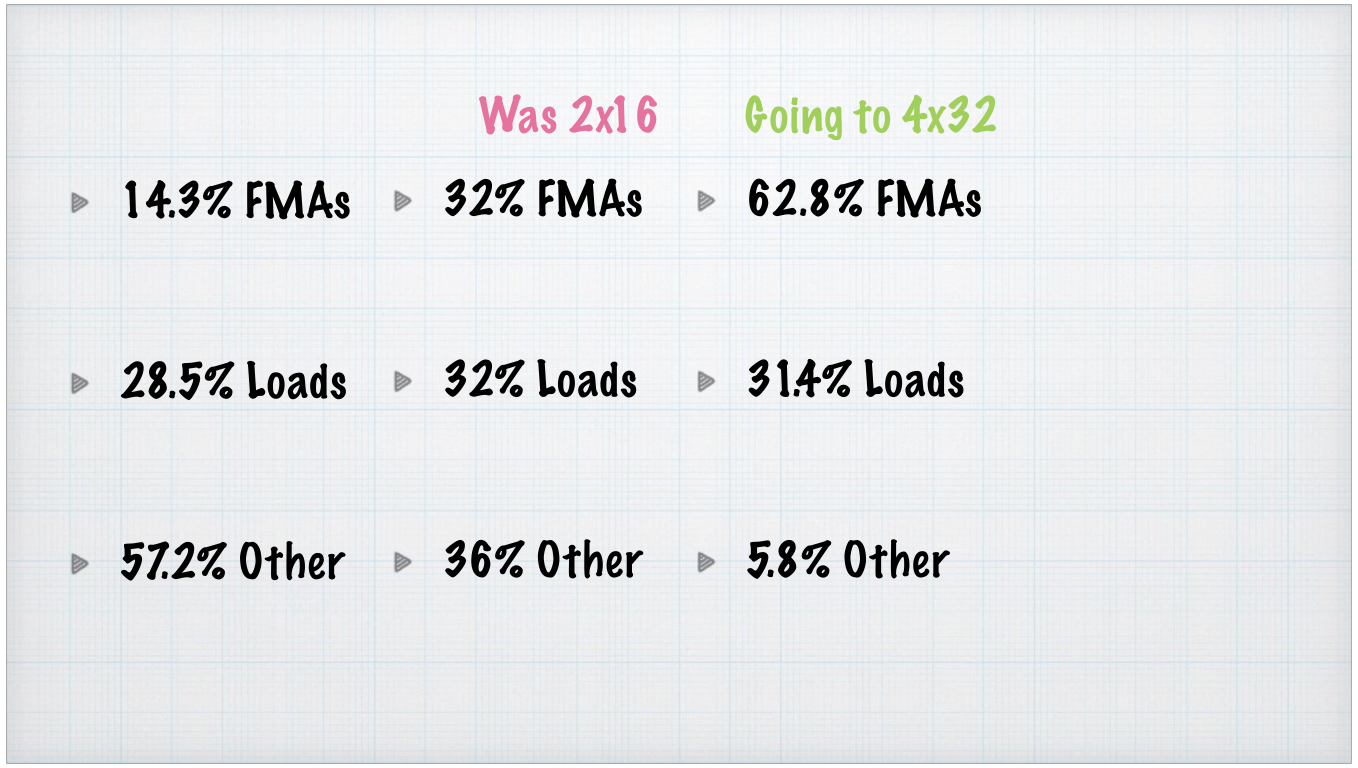
This change has the desired effect on the percentages: FMAs are up, and the “other” percentage is down.



1.64 TFLOP/s

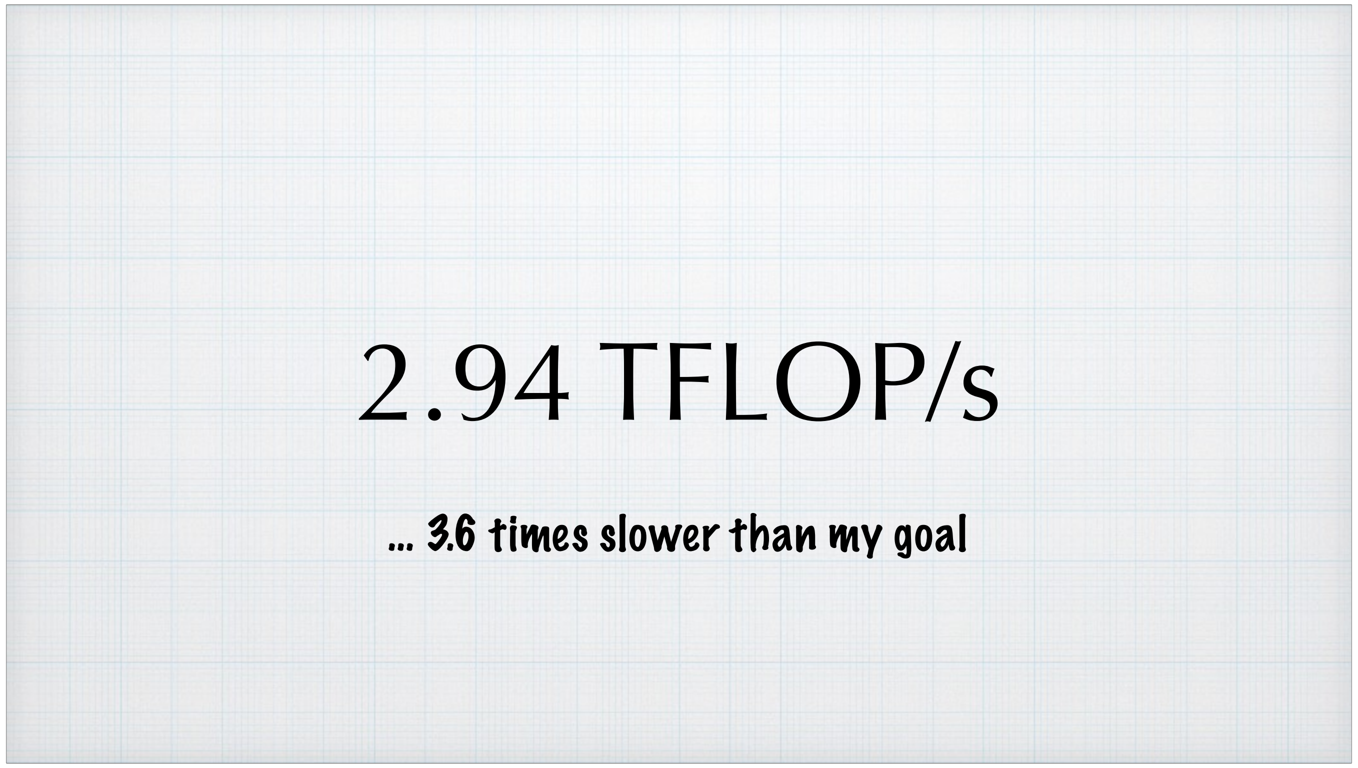
… 6½ times slower than my goal

Also a huge effect in performance, breaking through the teraflop barrier.



	Was 2x16	Going to 4x32
▶ 14.3% FMAs	▶ 32% FMAs	▶ 62.8% FMAs
▶ 28.5% Loads	▶ 32% Loads	▶ 31.4% Loads
▶ 57.2% Other	▶ 36% Other	▶ 5.8% Other

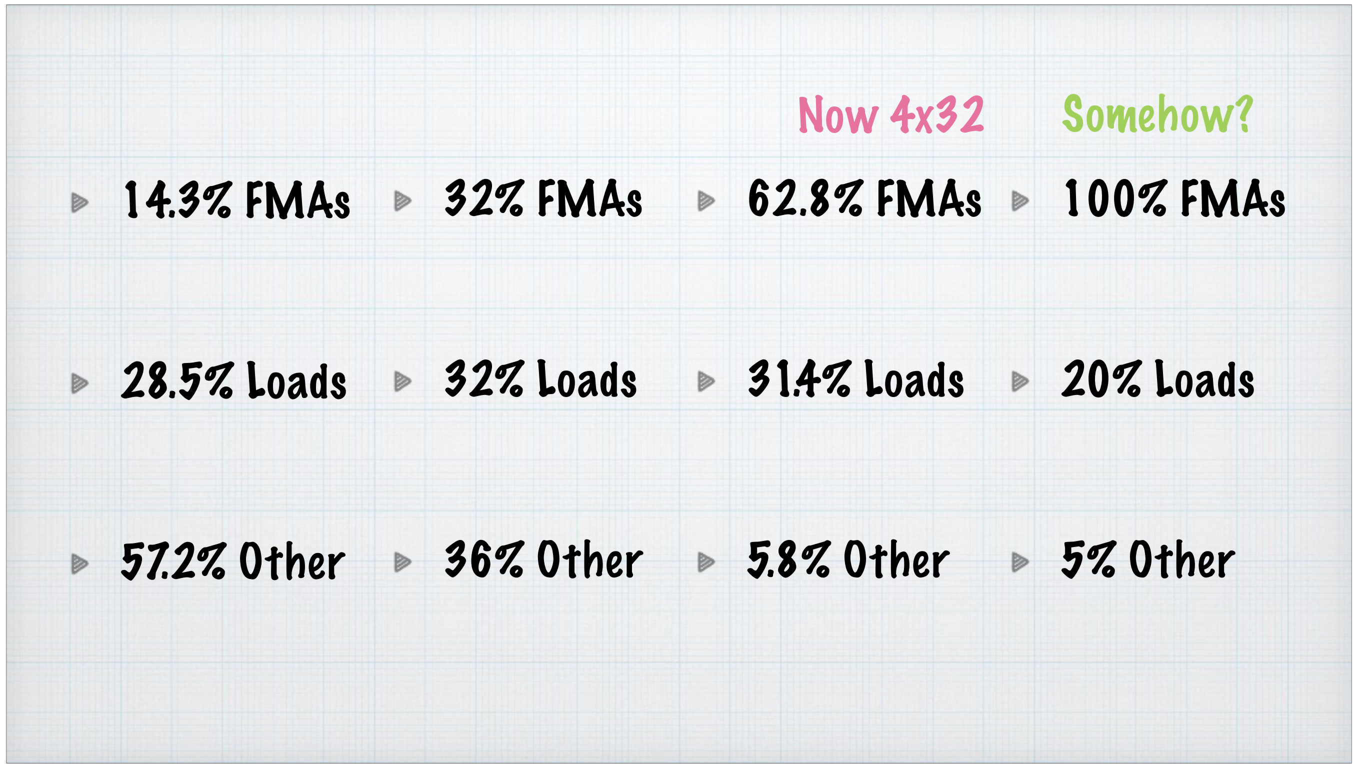
We can pull the same trick again, with another factor four work per outer loop iteration. I'm not going to show the code for this, as 2x16 only just fitted on one slide, and 4x32 is four times longer. I can show the percentages though; as before, FMAs are up, and the percentage of "other" is down.



2.94 TFLOP/s

… 3.6 times slower than my goal

At this point, performance is just shy of 3 teraflops, which is nice, but I’m still aiming for more than 10.

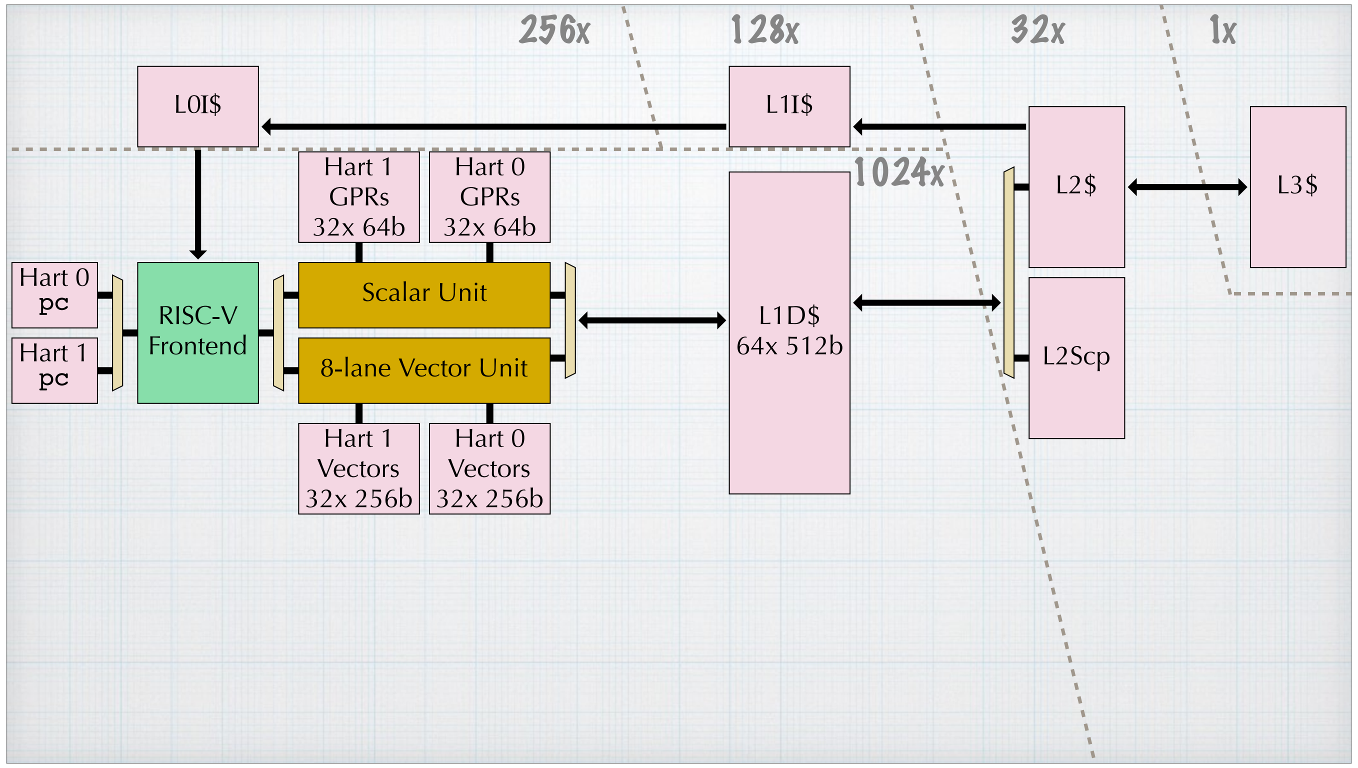


		Now 4x32	Somehow?
14.3% FMAs	32% FMAs	62.8% FMAs	100% FMAs
28.5% Loads	32% Loads	31.4% Loads	20% Loads
57.2% Other	36% Other	5.8% Other	5% Other

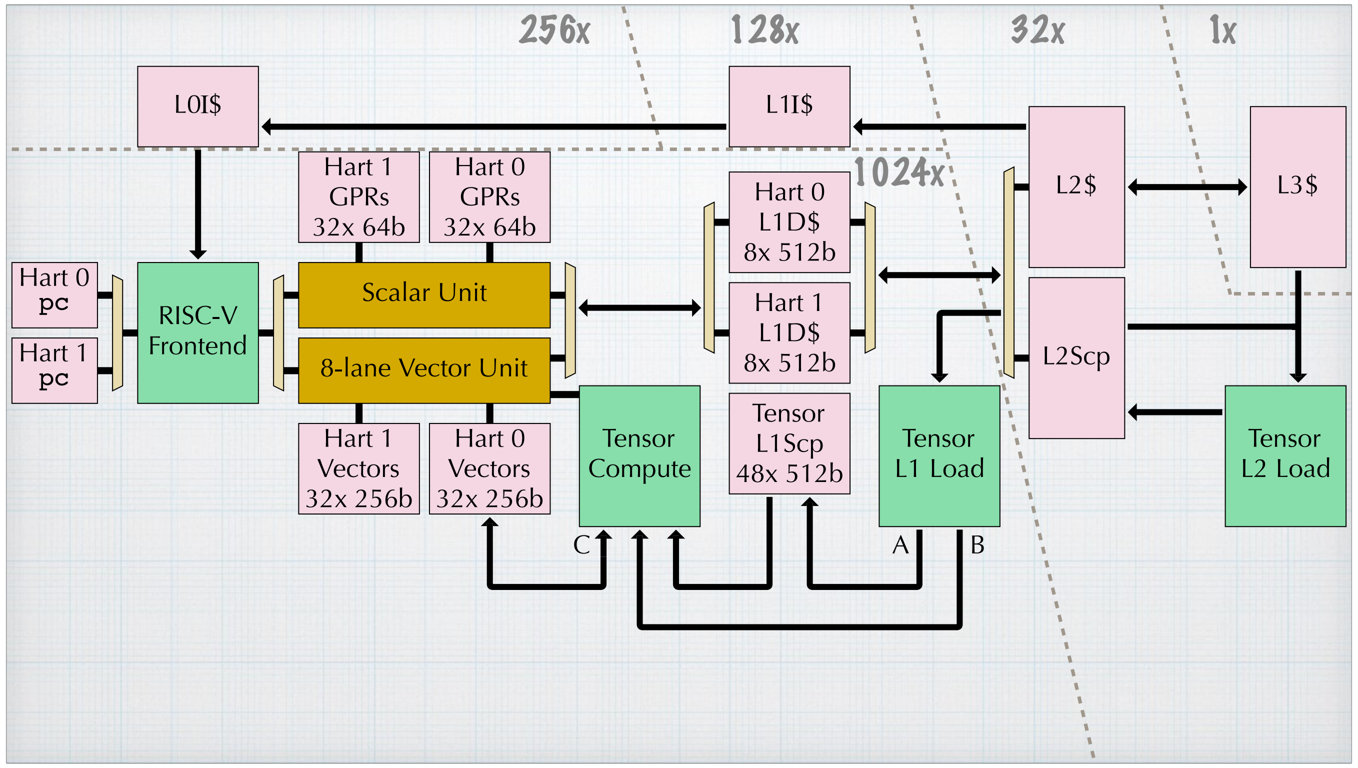
To get to 10 teraflops, I'd like to push the percentage of FMAs all the way to 100%. Of course, there still need to be some loads and some other instructions, which for sake of argument I'm going to ballpark at 20% and 5%. This adds up to 125%, which you might think is impossible, but it just requires creative thinking. To hit 125%, we need to execute 1.25 instructions per cycle on average, which means (at least sometimes) executing more than one instruction per cycle.

Act 3: Magic Hardware

[Time guide: T+13m30s]

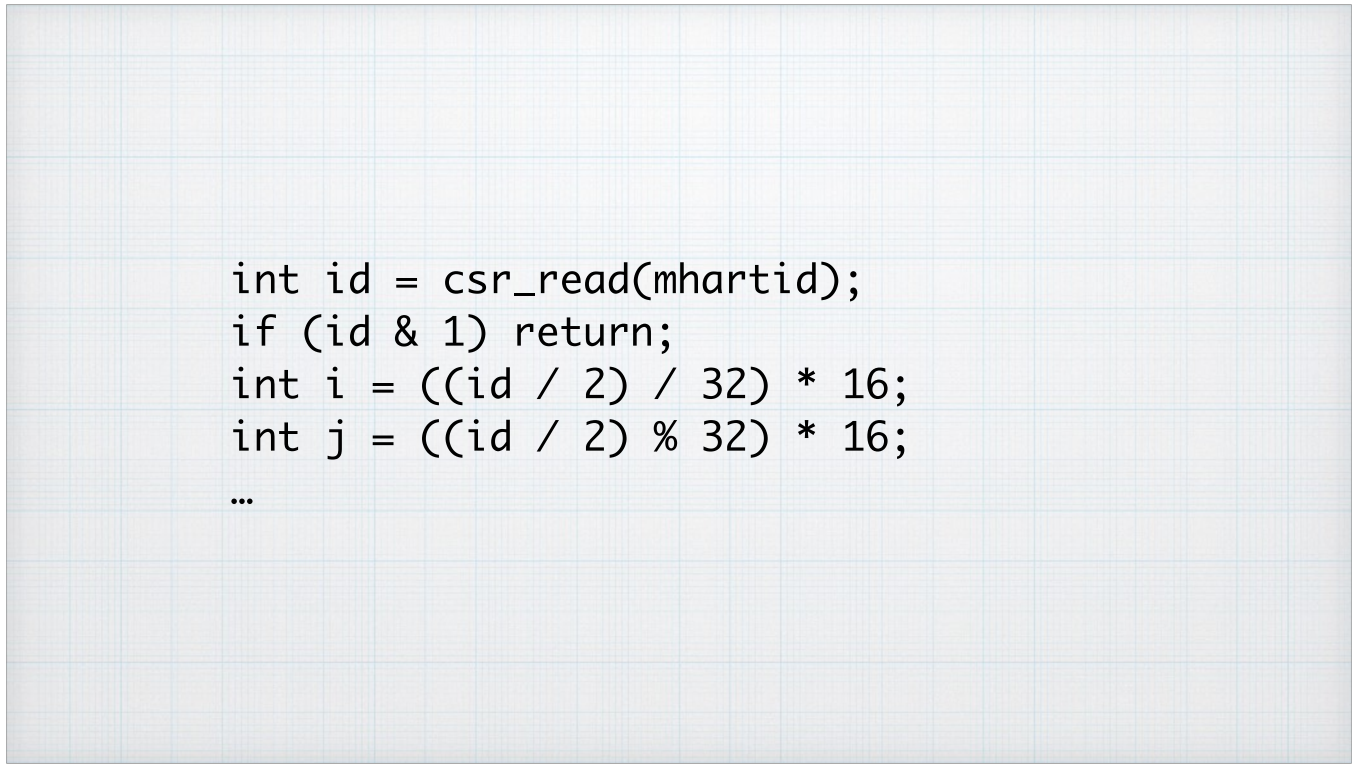


This is another diagram trying to show you the ET-SoC-1. The main region in the bottom left is showing a single minion CPU core, and as a reminder I'm playing with 1024 such cores. On this diagram I'm showing the scalar unit and the vector unit as separate pieces of hardware, so you might think that we can achieve two instructions per cycle by having one hart issue a scalar instruction and the other hart issue a vector instruction, but that doesn't actually work, as the frontend is only capable of one RISC-V instruction per cycle. There is however additional magic hardware which I've not yet used…



Same diagram, but now with the tensor hardware on it, shown as these additional green boxes. They're green because, like the RISC-V Frontend, they can each issue one instruction per cycle. Starting on the very right, all that Tensor L2 Load can do is load data into L2Scp. A proper implementation would use this to stream data into L2Scp, but I'm not drawing complete owls, so I'm not using it at all. Moving left, Tensor L1 Load is again only capable of loading data, but the destination of those loads is kind of interesting: most of the L1D$ is repurposed into a register file for holding Tensor L1 Load results, and then for loads which are used exactly once, there's a direct path from Tensor L1 Load to Tensor Compute. Those direct loads are are called B-type, whereas loads to L1Scp are called A-type, and I've put A and B labels on the diagram. Moving left again, the Tensor Compute box is what we're all really here for. Depending on how you count it, there are about ten different things this box can do, one of which is fp32 matmul accumulate with matrix shape 16x16. Relevant for me is that Tensor Compute can use the Vector Unit for doing the computation in parallel with the RISC-V Frontend using the Scalar Unit. You'll note that I've put a "C" label on the arrow linking Tensor Compute to hart 0's vector registers, and that all my matmul functions up until now have been computing C equals A times B, so hopefully you can guess where I'm going with this, which is that Tensor Compute can take an A-type load, multiply it with a B-type load, and accumulate the result C onto hart 0's vector registers.

These Tensor boxes don't have any pc or I$ wired up to them. Instead, the RISC-V frontend can spend a single scalar instruction to enqueue up to 512 instructions to Tensor Compute or up to 1 KiB of loads to either Tensor L1 or L2 Load. In any event, the enqueued instructions proceed asynchronously.

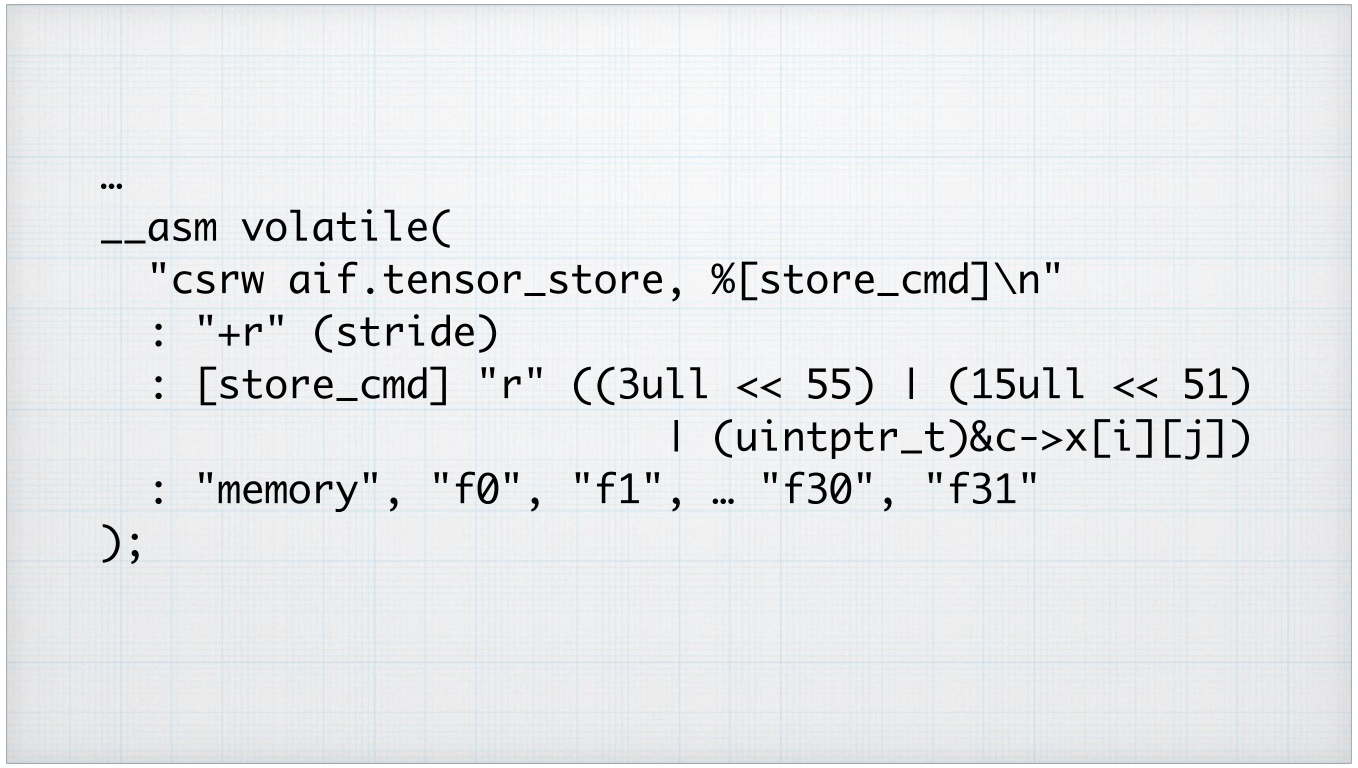


```
int id = csr_read(mhartid);
if (id & 1) return;
int i = ((id / 2) / 32) * 16;
int j = ((id / 2) % 32) * 16;
…
```

The code for using the tensor hardware isn't going to fit on one slide, so I'll show it over three slides. This is slide one of three, doing the usual mhartid stuff. The interesting part is that Tensor Compute and Tensor L1 Load can only be used by hart 0 of each core, so I'm making hart 1 return immediately. In a proper implementation, hart 1 would be prefetching data with Tensor L2 Load instructions, but I'm not drawing complete owls.

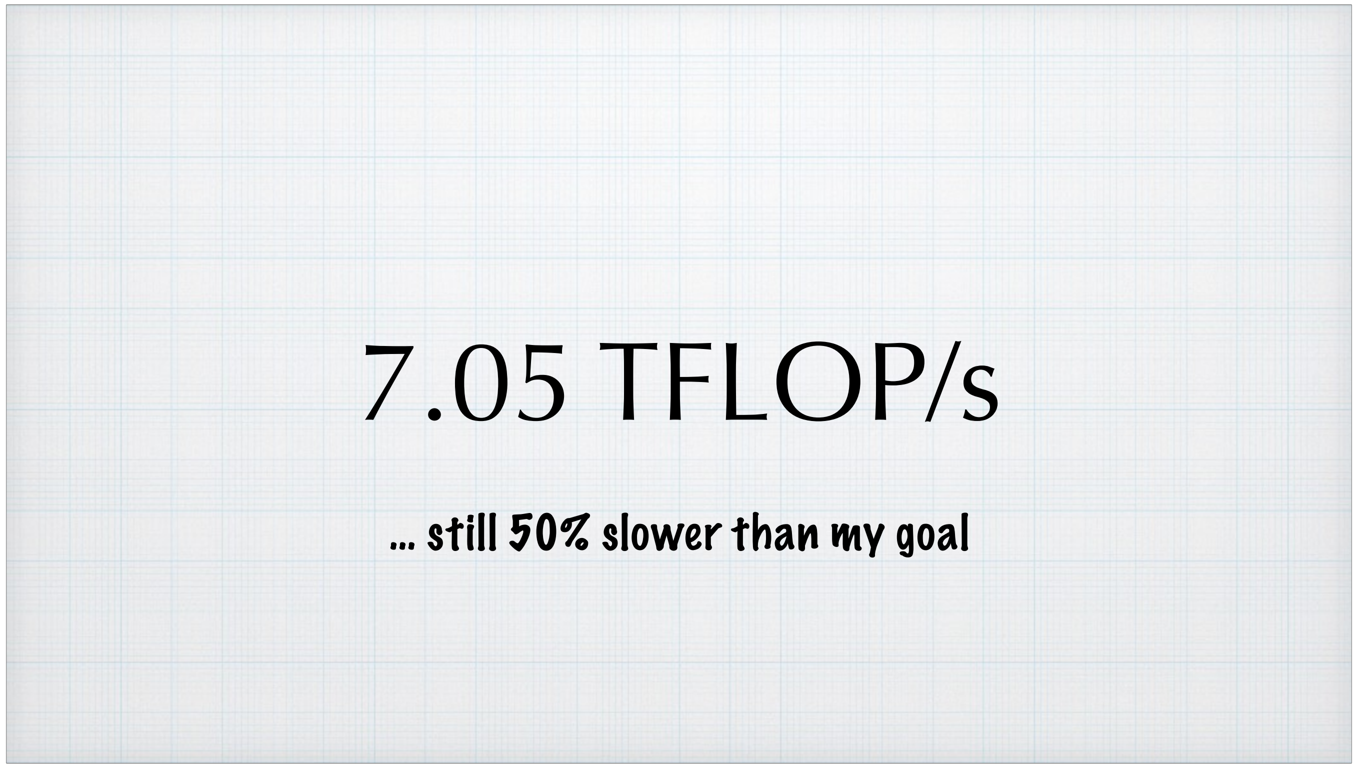
```
…
register u64 stride __asm("x31") = offsetof(matrix, x[1][0]);
for (int k = 0; k < 512; k += 16) {
  __asm volatile(
    "csrw  aif.tensor_load, %[load_a_cmd]\n"
    "csrw  aif.tensor_load, %[load_b_cmd]\n"
    "csrwi aif.tensor_wait, %[wait_a_cmd]\n"
    "csrw  aif.tensor_fma,  %[fma_cmd]\n"
    "csrwi aif.tensor_wait, %[wait_fma_cmd]\n"
    : "+r" (stride)
    : [load_a_cmd] "r" (               15 | (uintptr_t)&a->x[i][k])
    , [load_b_cmd] "r" ((1ull << 52) | 15 | (uintptr_t)&b->x[k][j])
    , [wait_a_cmd] "K" (0)
    , [fma_cmd]    "r" ((3ull << 55) | (15ull << 51) | (15ull << 47)
                                     | ( 1ull << 20) | (k == 0))
    , [wait_fma_cmd] "K" (7)
    : "memory", "f0", "f1", … "f30", "f31"
  );
}
…
```

Then we get the main loop. There isn't yet nice syntax for making use of the tensor hardware, so I've got inline RISC-V assembly in my C code, with the inline assembly containing all sorts of magic numbers to encode tensor instructions. The first two lines of assembly enqueue an A-type and a B-type load, the next line stalls the RISC-V frontend until the A-type load has completed, the next line enqueues 512 FMA instructions to Tensor Compute, and the final line again stalls the RISC-V frontend until those instructions are complete. If you want to understand this code, the hardware manual is on GitHub. For everyone else, just take my word that it works.



```
…
__asm volatile(
  "csrw aif.tensor_store, %[store_cmd]\n"
  : "+r" (stride)
  : [store_cmd] "r" ((3ull << 55) | (15ull << 51)
                         | (uintptr_t)&c->x[i][j])
  : "memory", "f0", "f1", … "f30", "f31"
);
```

The third slide of code is pretty short, just instructing the Tensor Compute unit to perform a store of C to memory.

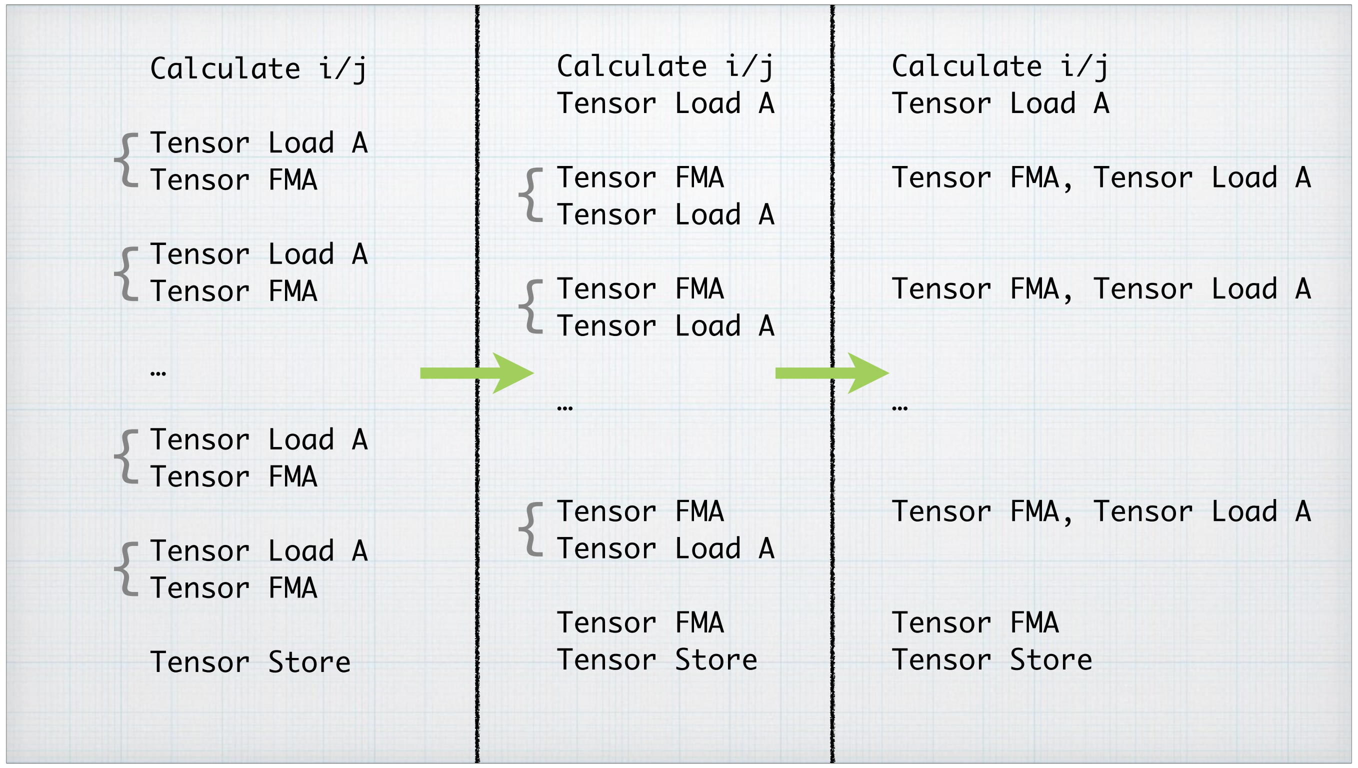


7.05 TFLOP/s

… still 50% slower than my goal

If I run the code from the previous three slides, it performs at just over 7 teraflops per second.

I’m not going to claim that the code is easy to read, but performance is not always pretty.

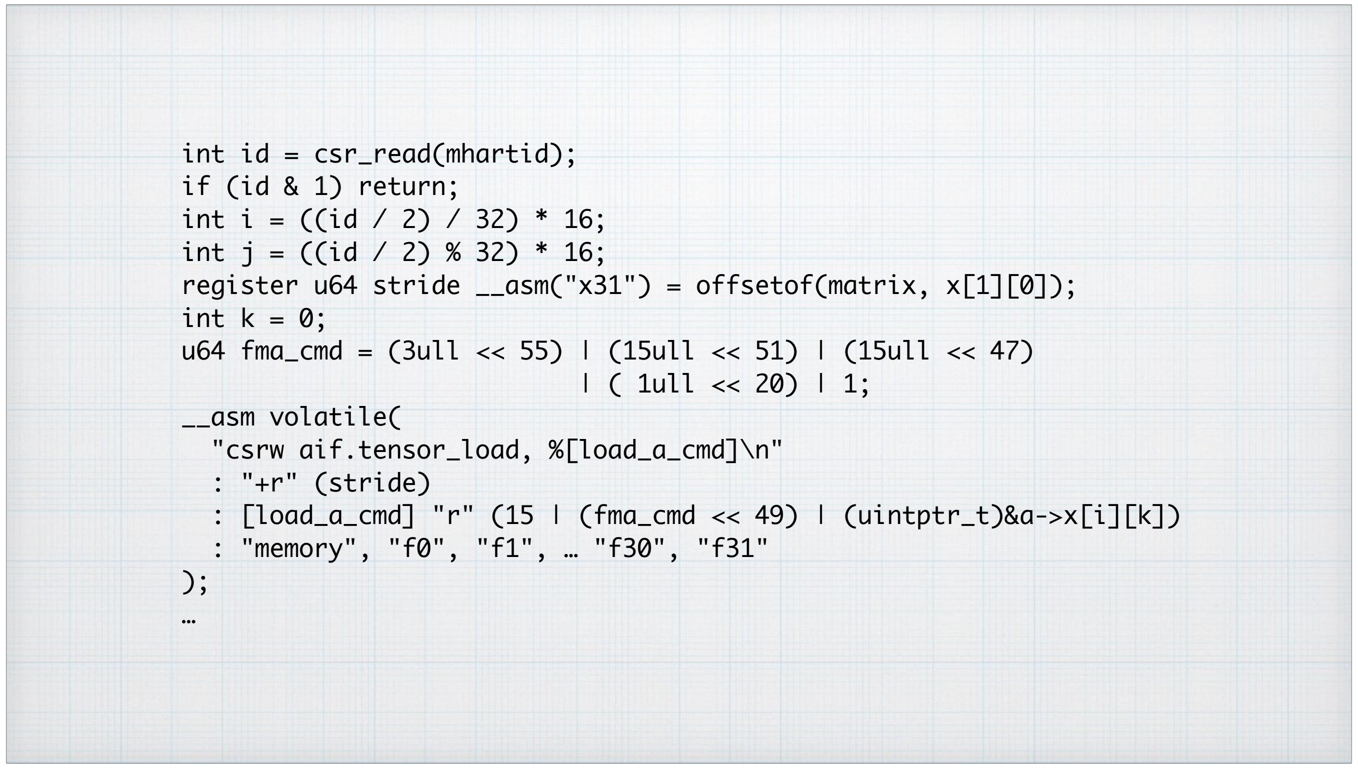


```
Calculate i/j

{ Tensor Load A
  Tensor FMA

{ Tensor Load A
  Tensor FMA

…

{ Tensor Load A
  Tensor FMA

{ Tensor Load A
  Tensor FMA

Tensor Store
```

```
Calculate i/j
Tensor Load A

{ Tensor FMA
  Tensor Load A

{ Tensor FMA
  Tensor Load A

…

{ Tensor FMA
  Tensor Load A

Tensor FMA
Tensor Store
```

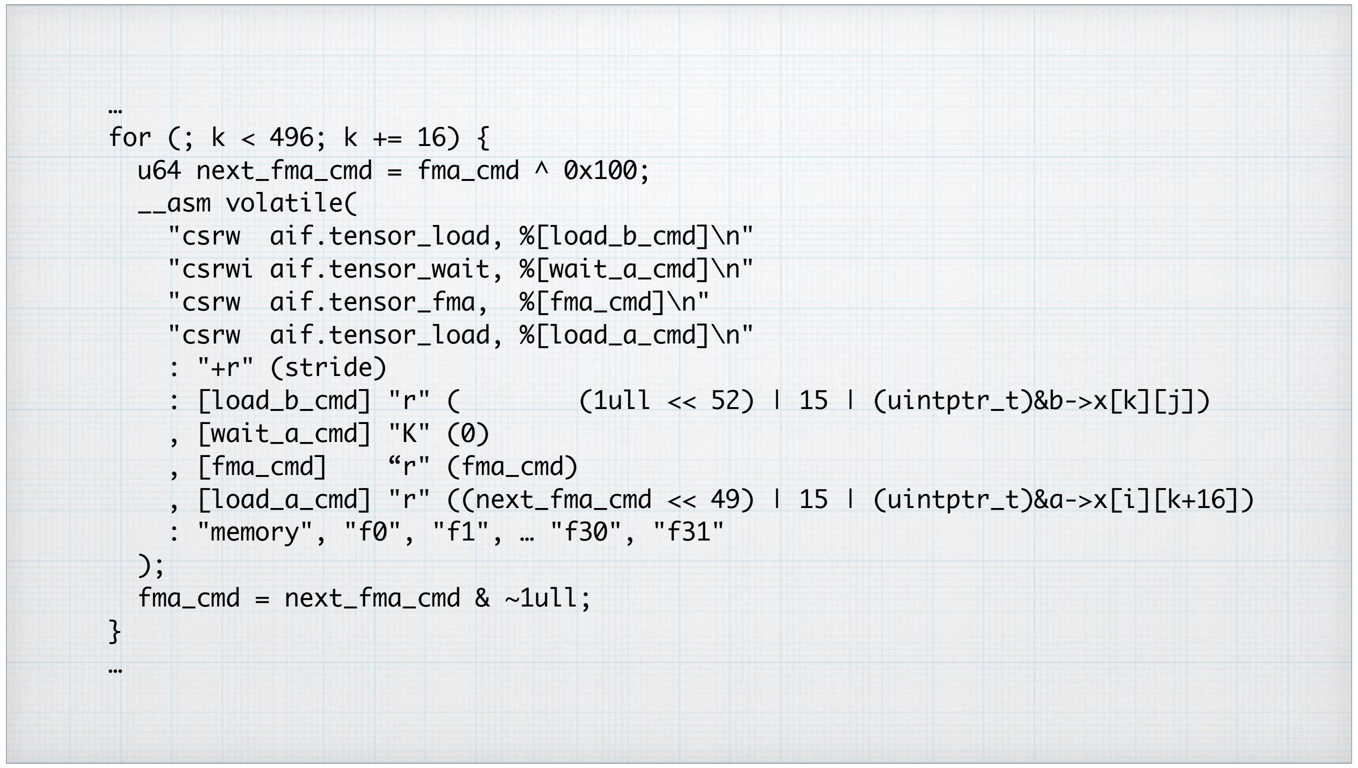
```
Calculate i/j
Tensor Load A

Tensor FMA, Tensor Load A

Tensor FMA, Tensor Load A

…

Tensor FMA, Tensor Load A

Tensor FMA
Tensor Store
```

I'm still a bit short of my goal, but I've got one more trick to go. The code I just showed you is the left-hand column here: calculate i/j, then a repeating loop of {A-type load, wait, FMA, wait}, then a store after the loop. I can re-arrange it a little bit as shown in the middle column: peel off the first A-type load and peel off the final FMA, and then the repeating loop changes to have the FMA before the A-type load. This is a very useful change, as now within each loop iteration, the FMA doesn't depend on the load nor does the load depend on the FMA, so they can be done in parallel, as shown in the right-hand column.



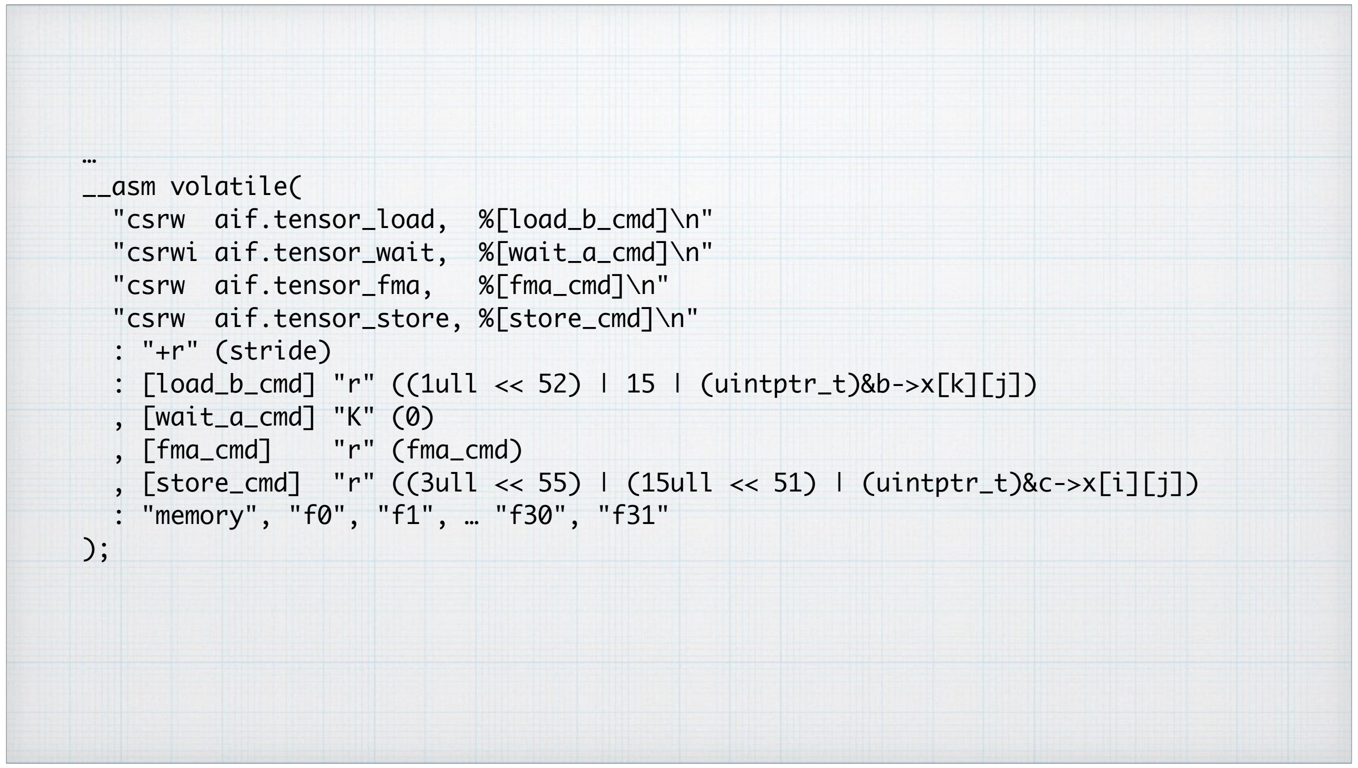
```
int id = csr_read(mhartid);
if (id & 1) return;
int i = ((id / 2) / 32) * 16;
int j = ((id / 2) % 32) * 16;
register u64 stride __asm("x31") = offsetof(matrix, x[1][0]);
int k = 0;
u64 fma_cmd = (3ull << 55) | (15ull << 51) | (15ull << 47)
                           | ( 1ull << 20) | 1;
__asm volatile(
  "csrw aif.tensor_load, %[load_a_cmd]\n"
  : "+r" (stride)
  : [load_a_cmd] "r" (15 | (fma_cmd << 49) | (uintptr_t)&a->x[i][k])
  : "memory", "f0", "f1", … "f30", "f31"
);
…
```

The code for the right-hand column is again three slides. We start with the same mhartid logic as before, and then the first A-type load.



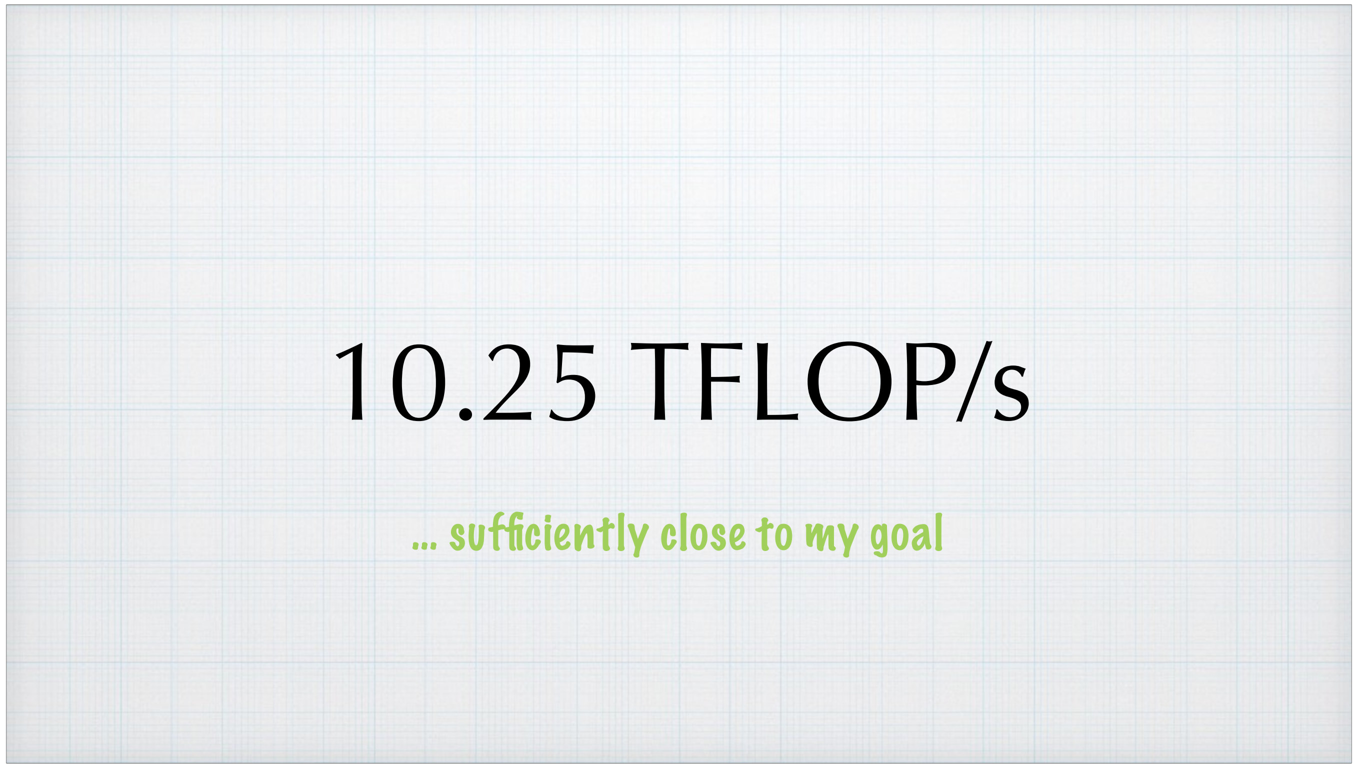
```
…
for (; k < 496; k += 16) {
  u64 next_fma_cmd = fma_cmd ^ 0x100;
  __asm volatile(
    "csrw  aif.tensor_load, %[load_b_cmd]\n"
    "csrwi aif.tensor_wait, %[wait_a_cmd]\n"
    "csrw  aif.tensor_fma,  %[fma_cmd]\n"
    "csrw  aif.tensor_load, %[load_a_cmd]\n"
    : "+r" (stride)
    : [load_b_cmd] "r" (        (1ull << 52) | 15 | (uintptr_t)&b->x[k][j])
    , [wait_a_cmd] "K" (0)
    , [fma_cmd]    “r" (fma_cmd)
    , [load_a_cmd] "r" ((next_fma_cmd << 49) | 15 | (uintptr_t)&a->x[i][k+16])
    : "memory", "f0", "f1", … "f30", "f31"
  );
  fma_cmd = next_fma_cmd & ~1ull;
}
…
```

The main loop now has the FMA before the A-type load, and as they can execute in parallel, there’s one less wait instruction.



```
…
__asm volatile(
  "csrw  aif.tensor_load,  %[load_b_cmd]\n"
  "csrwi aif.tensor_wait,  %[wait_a_cmd]\n"
  "csrw  aif.tensor_fma,   %[fma_cmd]\n"
  "csrw  aif.tensor_store, %[store_cmd]\n"
  : "+r" (stride)
  : [load_b_cmd] "r" ((1ull << 52) | 15 | (uintptr_t)&b->x[k][j])
  , [wait_a_cmd] "K" (0)
  , [fma_cmd]    "r" (fma_cmd)
  , [store_cmd]  "r" ((3ull << 55) | (15ull << 51) | (uintptr_t)&c->x[i][j])
  : "memory", "f0", "f1", … "f30", "f31"
);
```

The code after the loop contains the final FMA and then the store.



10.25 TFLOP/s

… sufficiently close to my goal

Again, I don’t honestly expect anyone in the room to fully comprehend three slides of fairly cryptic code, but if you are interested in the details you can download the slide deck and cross-reference with the hardware manual. What I would like you to take away is that the code works, and gets the hardware running at above 10 teraflops, which is within a few percent of the theoretical maximum. I’m going to declare that as goal accomplished. As I’m pretty much out of time, someone else can draw the rest of the owl.