Reduced precision in HPC

Why should AI folk have all the fun?

Felix LeClair

Relevant parties:

Standards
IEEE X54 series
ISO C, C++ and Fortran

Traditional Hardware Vendors
x86_64 procesor vendors (AMD, Intel)
ARM V8 and V9 processor vendors (Amazon, Ampere, Cavium, Nvidia etc.)
GPU/FPGA vendors (Intel, Nvidia, AMD, Amazon, GraphCore etc.)
RISC-V (Si-Five)

AI Hardware vendors that HPC can leverage
Graphcore, Groq, Cerebras, TensTorrent, AWS inferentia, Google TPU

Definitions 1:

Mixed precision: An operation (typically a compute kernel) that changes internal types based on the needs of the application. In AI: Convert to BF16 or TF32 from FP32

Reduced precision: Any type that uses less than 32 Bits

Mantissa: The fraction of a floating number, the other parts being exponent and sign

Heterogeneous Compute: the process of mixing compute tasks between the host CPU and other discrete devices; most commonly GPUs

Accelerator: A device that can access a hosts data, but contains hardware tailored to a specific task. Can take the form of fixed function blocks (Matrix Tiles), a general purpose parallel compute element (GPU, VPU) or reprogrammable device tailored to the task at hand (FPGA)

Definitions 2:

CFD: Computational Fluid Dynamics; the process of simulating how fluids interact. Typically used to optimize efficiency of vehicles, analyze interaction of fluids in the human body etc.

Downforce: The resultant effect of displacing air to create a downwards force in a motor vehicle. The opposite of “lift” in aircraft. Typically created via inverted wings or via virtual expansion champers, exploiting the ground effect (Bernoulli’s Principle). Used in motor racing to increase tyre grip

First up: **FP16**

But what ***is*** FP16?

Standard answer:
A floating point datatype defined by IEEE-754 since the 2008 revision.

The 2008 spec superseded IEEE-754 1985 and (effectively) deprecated IEEE-854-1987

~~Humorous answer: A huge pain~~

~~Standard answer:~~

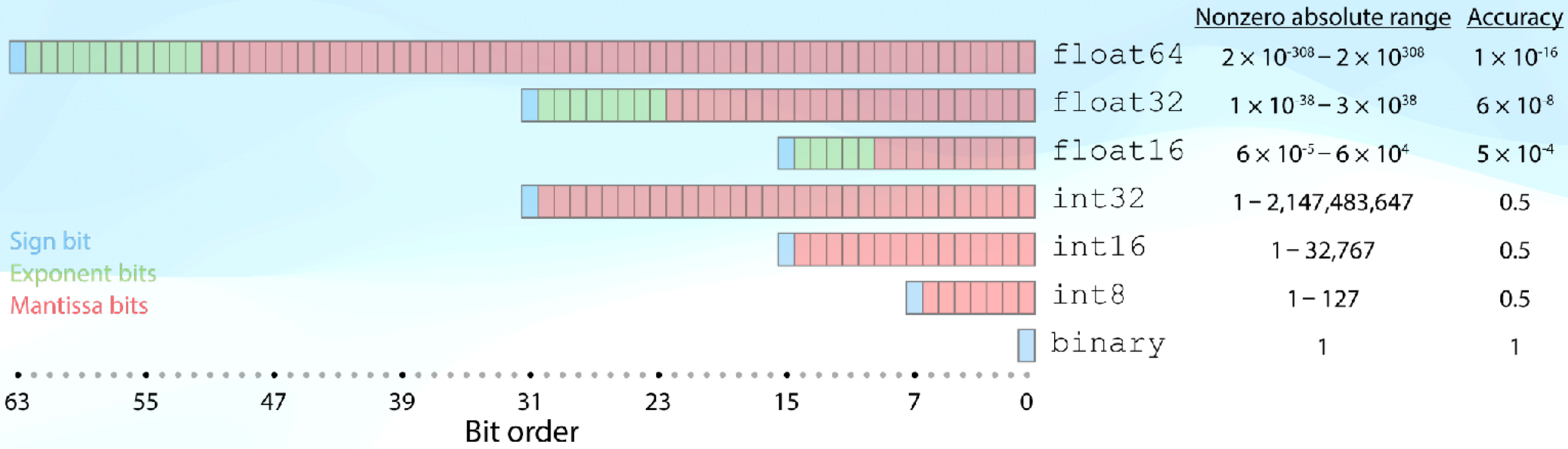
~~A floating point datatype defined by IEEE-754 since the 2008 revision.~~

~~The 2008 spec superseded IEEE-754-1985 and (effectively) deprecated IEEE-854-1987~~

Humorous answer:

Up until recently, a huge pain

What does it actually look like?



But why do we ***care*** about FP16?

Standard answer:

FP16 gets around 2 main problems that HPC has been facing for years;

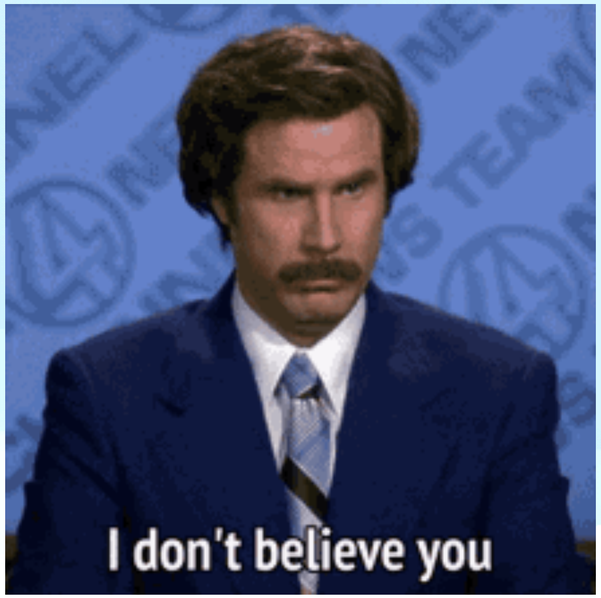
Being **memory bound or compute bound**

In ***principal***, we can get:

Direct **2x speedup over FP32** if we're compute bound.

We get a **2x effective boost in memory bandwidth** if we're memory bound.

“Ok great, but vendors have been talking about magic pixie dust that give 2x+ performance gains for years, does this actually do it?”



“Ok great, but vendors have been talking about magic pixie dust that give 2x+ performance gains for years, does this actually do it?”

Yes* but <u>nothing</u> is free.

“Convince me, I need data”

Case 1: Memory Bound

How does FP16 help us deal with this?

Fo most implementations, we get double the bang for our buck vs FP32, meaning that if we're at 100% overall memory bandwidth usage, and only 60% compute usage, we can shift the problem to being Compute bound and see a benefit

We have a choice:

Scenario 1: All computation in FP16.
No overhead, but loss of numerical accuracy.
Up to 2x increase in compute throughput

Scenario 2: Store in FP16, compute in FP32.
Adds overhead, better *overall* precision.
Since we were already memory bound, we're now compute bound. Information lost to rounding, not as much as Scenario1

Case 1: Memory Bound

Scenario 1: All FP16

This becomes a game of testing.

Does FP16 hold enough data for our computation? If so, we're golden and get ~2x for free*

Some AI applications (especially inference) are pure FP16 because of the relatively low need for precision.

Case 1: Memory Bound

Scenario 1: All FP16

What are we testing for?

Does FP16 hold enough data for our computation? If so, we're golden and get ~2x for free*

Problem for HPC or AI training? We're typically looking at large numbers which get small value updates. If the added value is too small AKA outside the range of precision FP16 allows, we have data loss.

Case 1: Memory Bound

Scenario 1: All FP16 - AI Training

The AI training problem:

The further into training you are, the smaller your updates to model weights can be.

If the weights are between $0.5 < x < 1$ FP16 can be incremented by values between $0 < y < 2^{-11}$

If the weights are between $0.25 < x < 0.5$ FP16 can increment by values between $0 < y < 2^{-12}$

If the weights are between $0.125 < x < 0.25$ FP16 can increment by values between $0 < y < 2^{-13}$

Case 1: Memory Bound

Scenario 1: All FP16 - AI Training

If we're at iteration ~10:
Updates tend to be larger, on the order of 2^-3 to 2^-9

But as we approach iteration ~10 000:
Updates tend to get small, on the order of 2^-9 to 2^-13

Assuming:
Weights clamped to $0 < x < 1$, value to be added is 2^-12

Weights between $0.125 < x < 0.25$, we can do the increment without data loss.

For that same iteration, if starting value is between 0.5 and 1, the increment is too small, data loss.

Case 1: Memory Bound

Scenario 1: All FP16 - AI Training

If we're at iteration ~10 M:
Updates tend to be minuscule, on the order of 2^-13 to 2^-19

But as we approach iteration ~1B:
Updates tend to get incredible small, on the order of 2^-20 to 2^-24

Assuming:
Weights clamped to $0 < x < 1$, value to be added is 2^-15

Any weight with value above 0.03125 will have data loss

You can forget about anything past the 10M mark getting updated if training in FP16

Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - AI Training

Here we have a different approach: x86, ARM and GPUs all have fast, in hardware capabilities for translating to and from FP16, with guaranteed levels of accuracy.

And since translating to and from takes less time than grabbing an additional set of data, we still come out ahead.

Moving from FP32 to FP16 store, FP32 math will increase performance.

Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - AI Training

Restated:

```
if ((Fetch FP16 from memory, FP16->FP32, *math in FP32*, FP32->FP16,
                                store FP16
                              Is faster than
          Fetch FP32 from memory, *math in FP32*, store FP32)
                                  &&
(error from rounding to and from FP16 < error accumulation threshold))
                            {use_conversion = true}
```

Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - AI Training

"How does this solve the problems with Scenario 1? We've added overhead for conversion, but you were saying that the difference between the large value with minor updates earlier was the problem"

Since we know FP32 was capable of properly dealing with our smaller updates, by having the underlying math happen in FP32, we're only rounding at the very end.

We're adding extra compute overhead, but the overhead is offset by the overall gain we get from not waiting on memory in the first place.

Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - AI Training

Simply put:

If we are already bound by memory, by switching to FP16->FP32, we gain a 2x in effective memory bandwidth.

Even if conversion was very computationally expensive, on the order of 50% increase in compute complexity (for very small kernels) we'd still come out ahead overall.

Case 1: Memory Bound

Scenario 1: All FP16 - HPC CFD

The CFD problem:

In CFD we're waiting for model convergence, where updates are beyond the rounding range of our datatype.

CFD by its nature deals with chaotic vorticity, meaning we can never "count" on values being clamped.

As in the AI training scenario, how much we can update by is dictated by the preexisting value at a specific tilmestep.

But Vortex generators don't like being consistent.

Case 1: Memory Bound

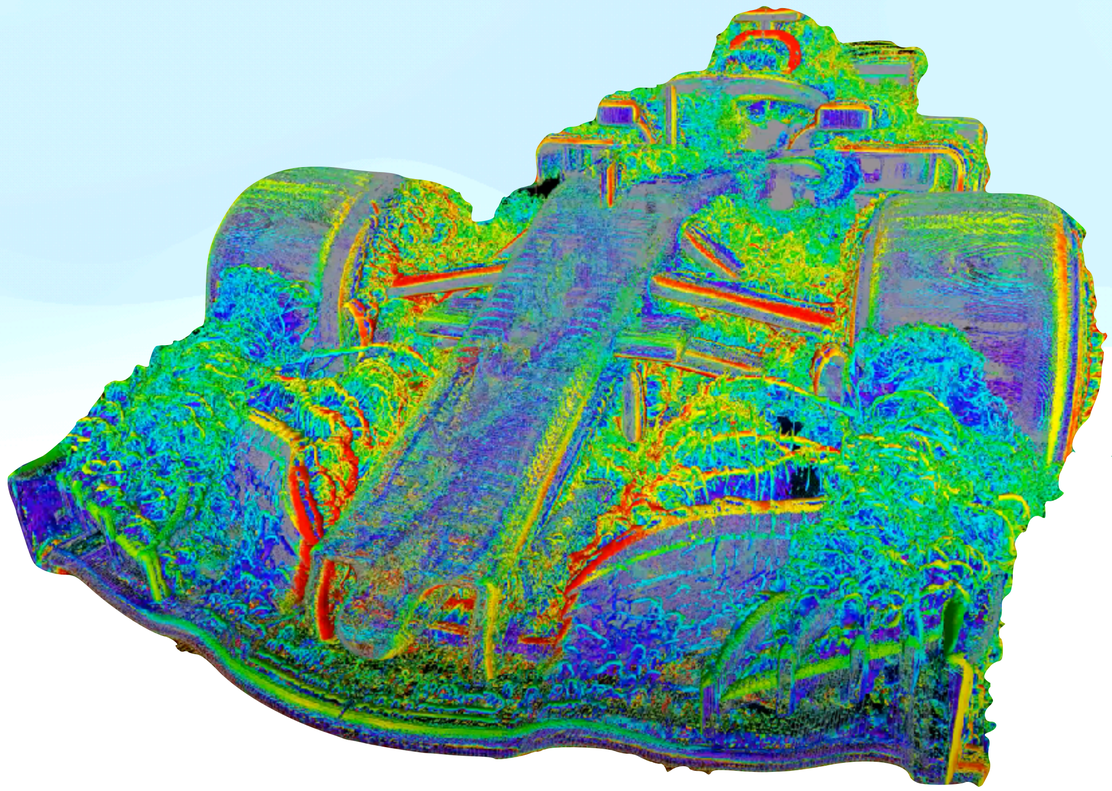
Scenario 1: All FP16 - HPC CFD

Practical example, Formula 1 race car.

Traveling from 0 KPH to 300 KPH in under 10 seconds, This vehicles experience up to 2.5 G of vertical downforce, with lateral downforce in the 4-6 G range

Traditional inverted wing “plows” the air out of the way, creating a downwards force vector

Air under car is accelerated, which by Bernoulli, we know creates a vacuum, pulling the car down.



Case 1: Memory Bound

Scenario 1: All FP16 - HPC CFD

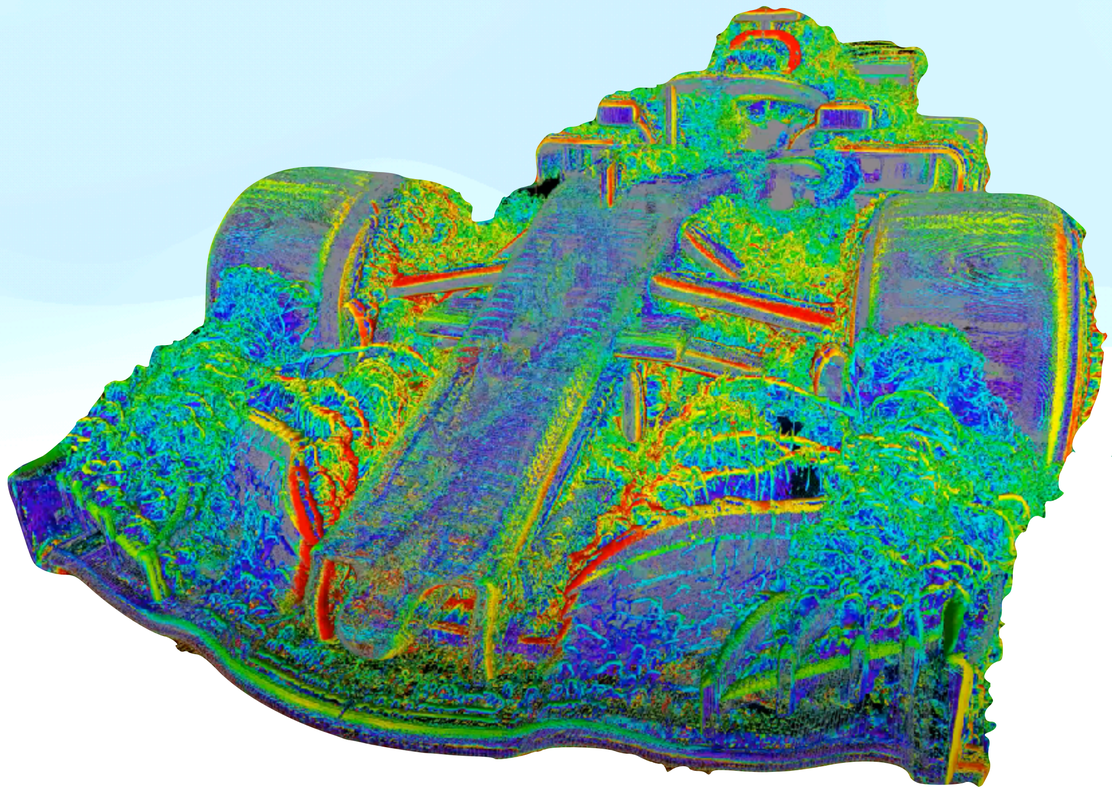
Practical example, Formula 1 race car.

The Bernoulli portion is where FP16 has problems.

Want a vacuum? We need to seal the air under the car.

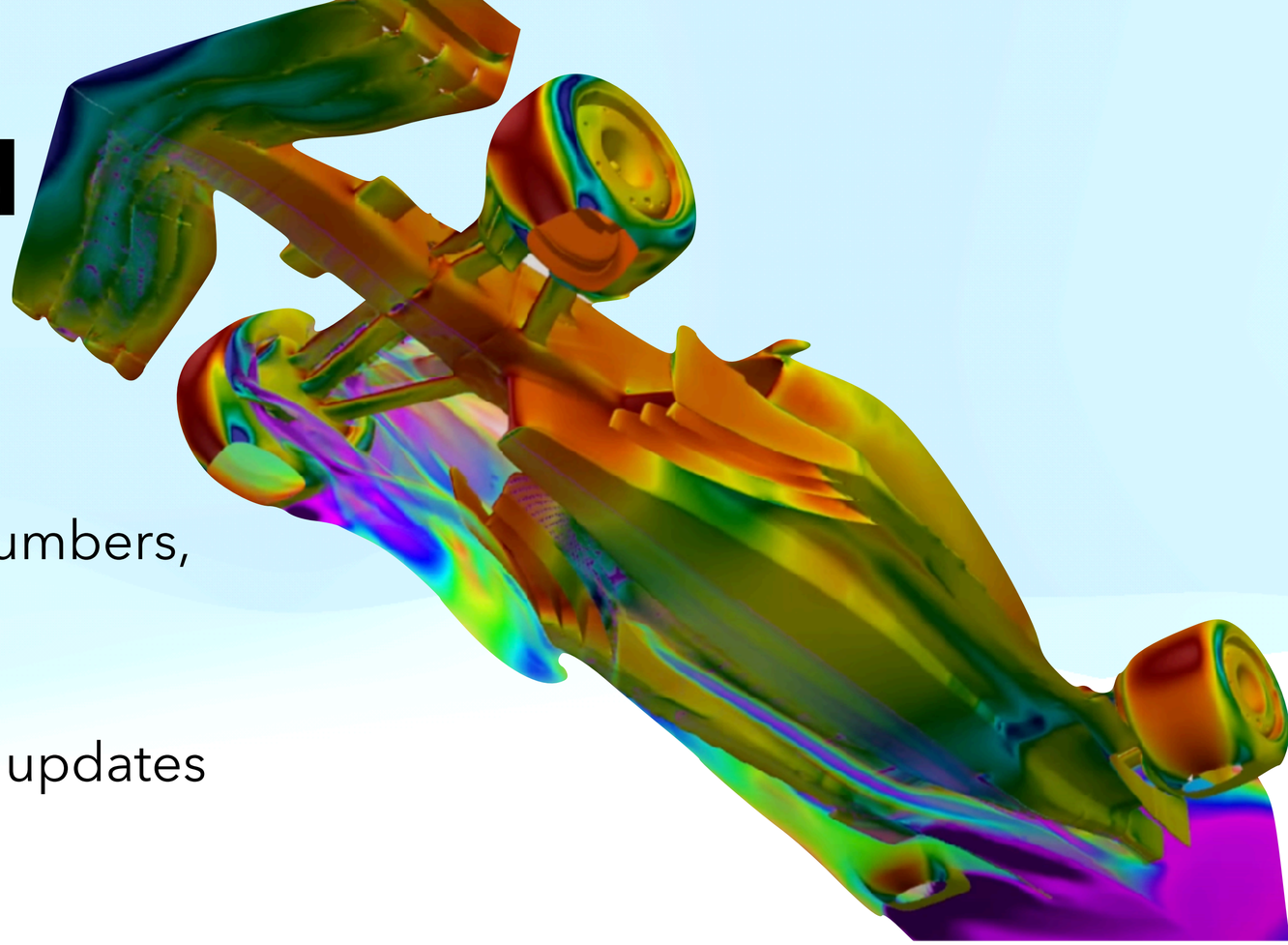
Rules say we can’t have skirts/physical barriers.

Solution? Create powerful vortices and “aim” them to the edge of the floor. The energized vortices act as barriers between the air under the floor and not under the floor.



Case 1: Memory Bound

Scenario 1: All FP16 - HPC CFD



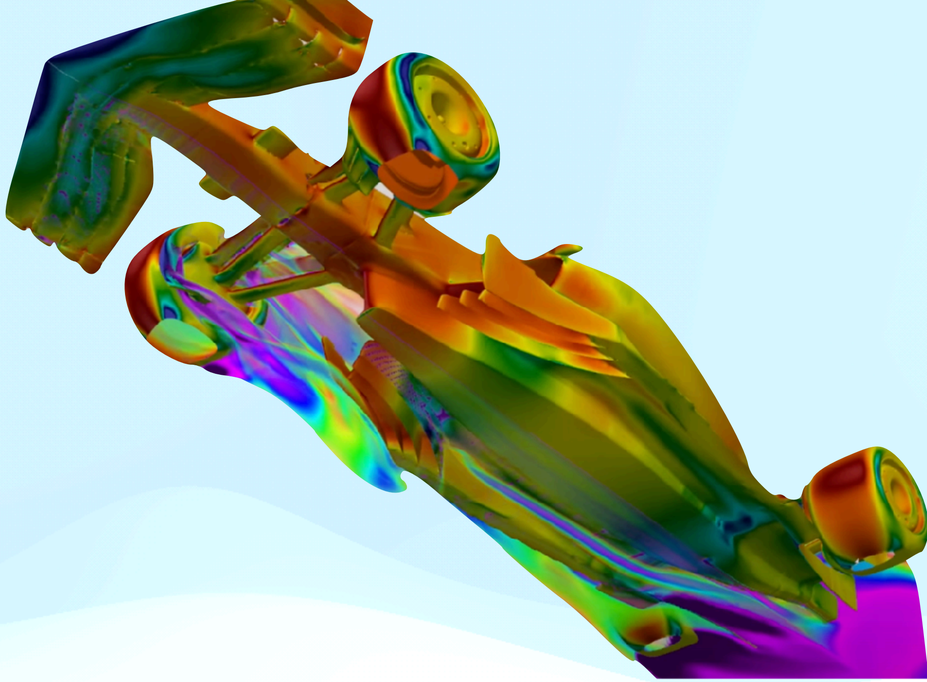
Recall from earlier:
When we want to do small updates to large numbers, we quickly run out of range

The longer the simulation runs, the smaller our updates are

Our simulation terminates when we reach steady state, and steady state is determined by going beyond the range of our datatypes.

Case 1: Memory Bound

Scenario 1: All FP16 - HPC CFD

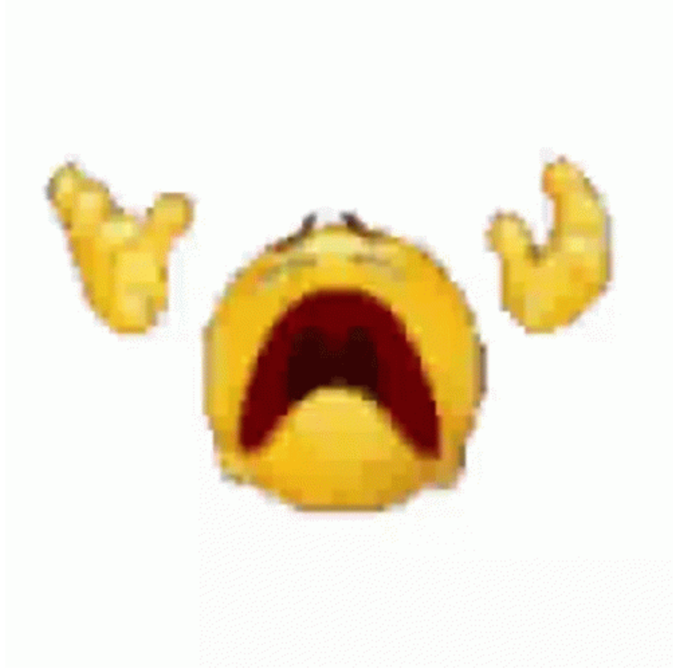


Well shucks, now we have an issue.

Even if we’re currently memory bound and we get a 2x speedup, the data is garbage.

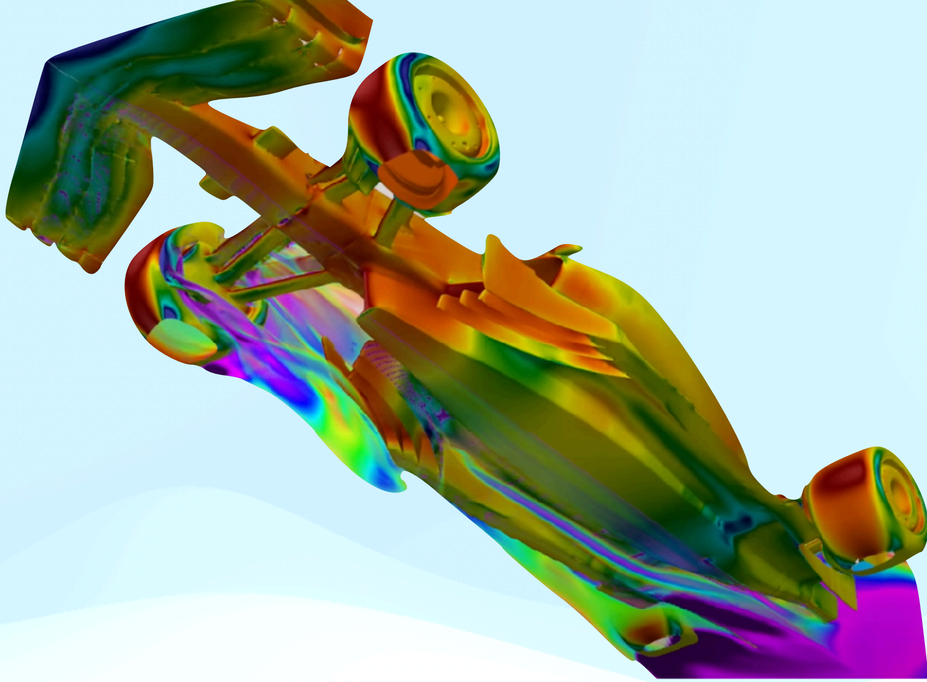
AND

Leading to case 2…



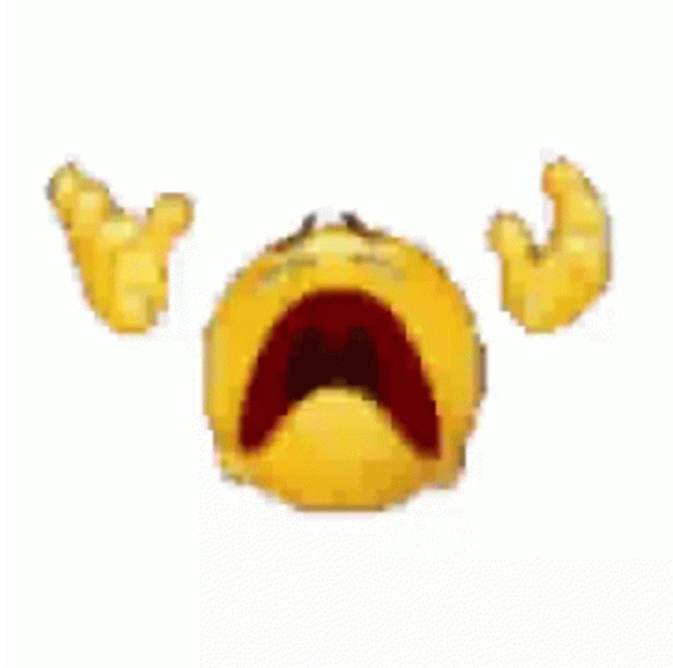
Case 1: Memory Bound

Scenario 1: All FP16 - HPC CFD



Well shucks, now we have an issue.

Even if we’re currently Memory bound and we get a 2x speedup, the data is garbage and unsuitable for use



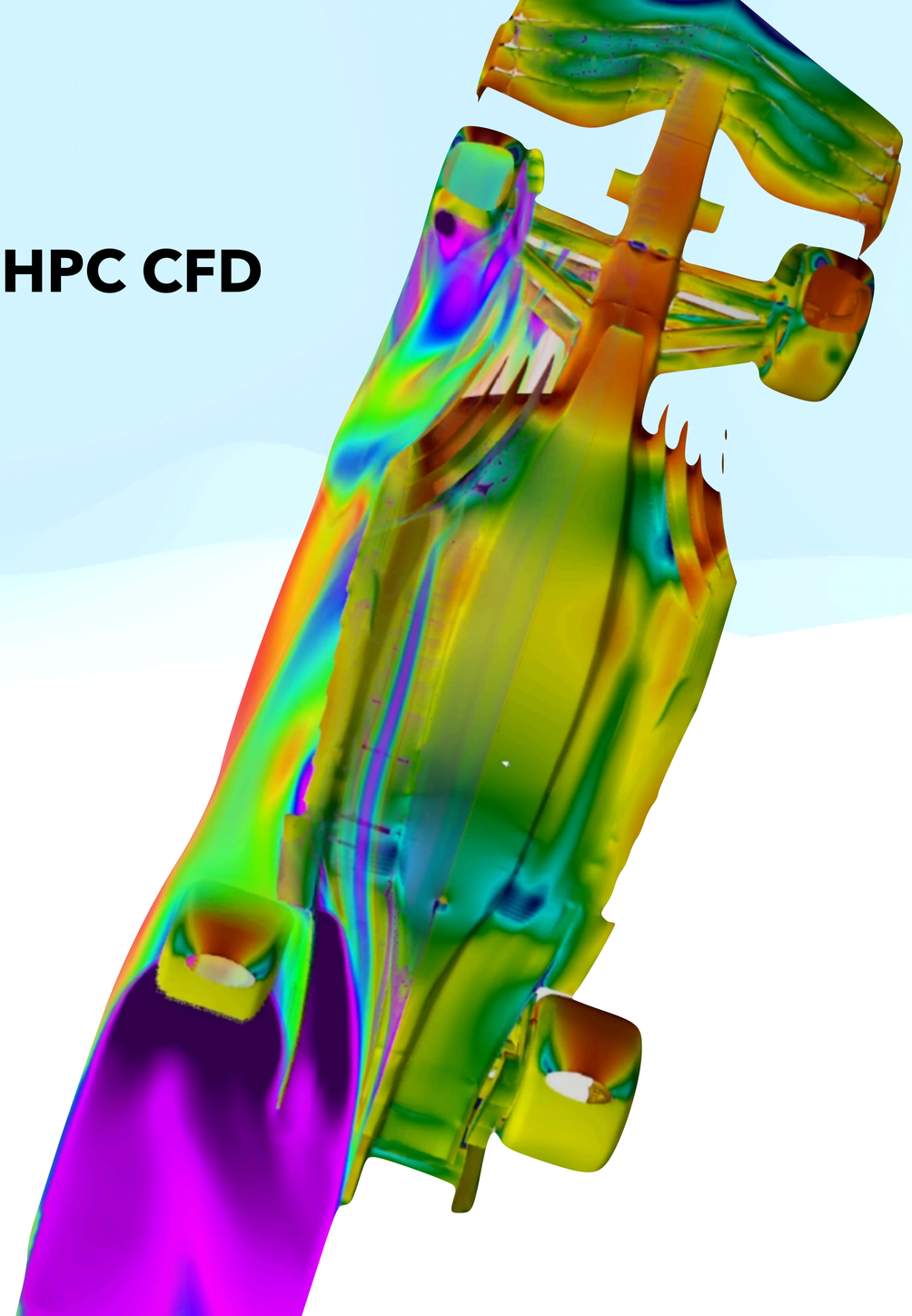
Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

Recall from earlier:
When we want to do small updates to large numbers, we quickly run out of range

The longer the simulation runs, the smaller our updates are

Our simulation terminates when we reach steady state, and steady state is determined by going beyond the range of our datatypes.

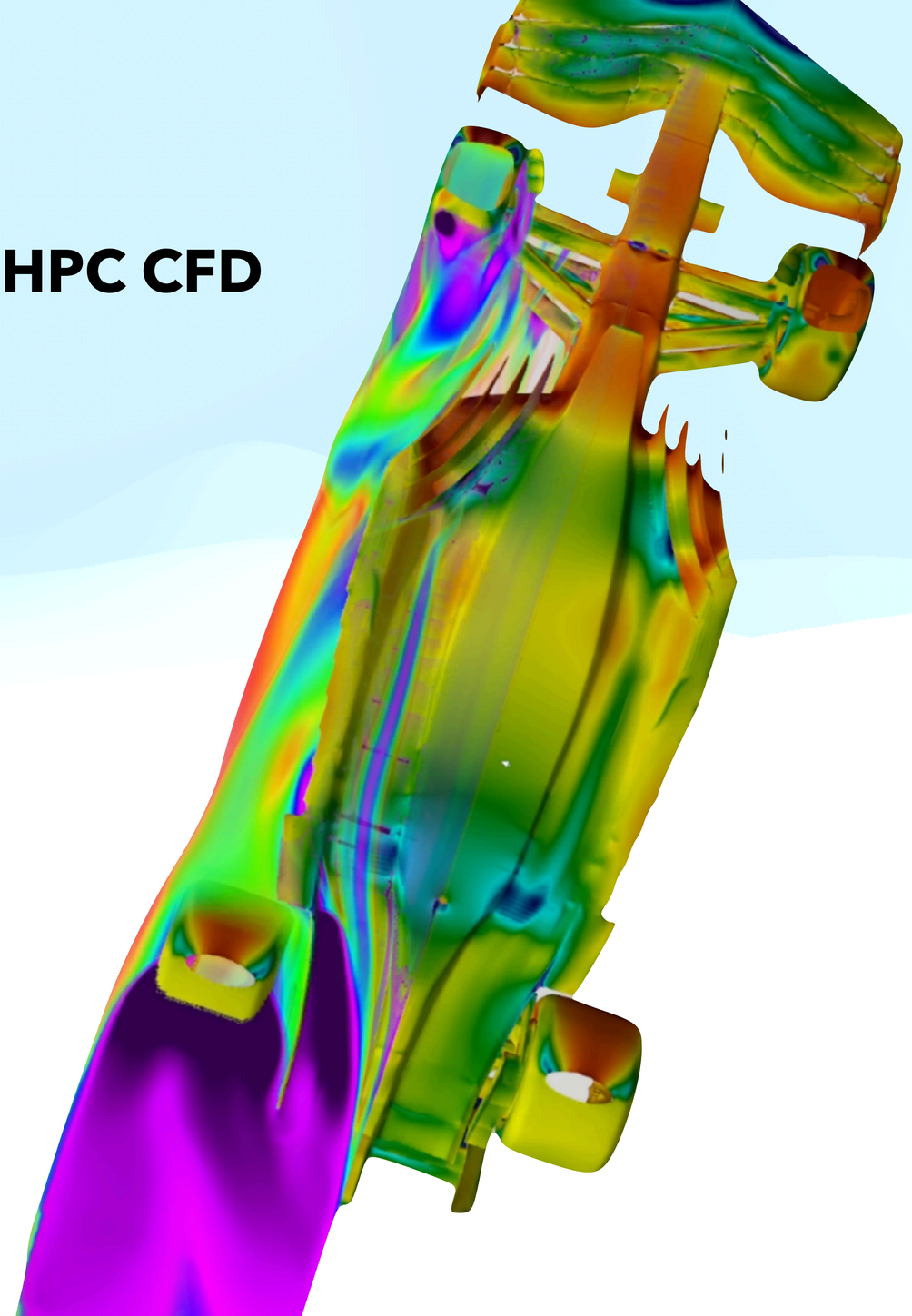


Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

We know our application in full FP32 was bound by memory

We know that FP16 wasn’t able to contain the data we needed, leading to pre-mature simulations termination and/or garbage data



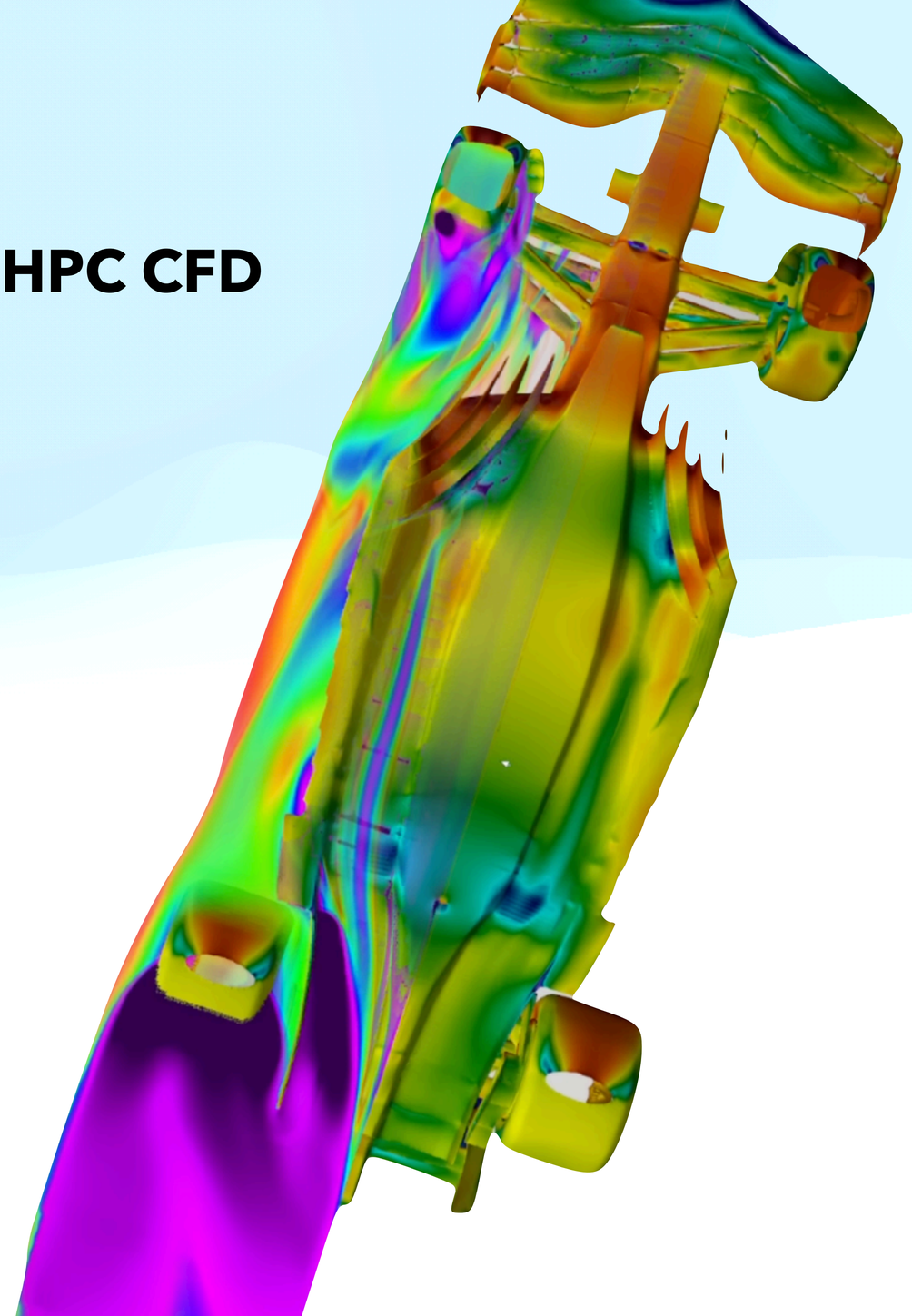
Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

But...

Much like the AI training example, we can split the difference, so long as storing in FP16 can be demonstrated as fit for purpose

If ***only*** such a thing existed



Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

Enter: FluidX3D

An Open-source, OpenCL based,
mixed precision solver

Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

Designed as part of a masters+PHD thesis, I'm collaborating with the author on a SYCL2020 port.

Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

The core concept is around what I suggested earlier:

We need to compute in FP32, but storing in FP32 is far too slow.

FluidX3D doesn’t use IEEE 754 FP16, but does use a custom 16 bit floating point type

Designed as part of a masters+PHD thesis, I’m collaborating with the author on a SYCL2020 port.

Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

"Great, so now you're telling me that I have to use a custom type? Doesn't that negate the speedup advantages since we can't use built in conversion hardware?"

No and Yes: turns out that a doubling of effective memory bandwidth is enough to offset even manual conversion and bit manipulation.

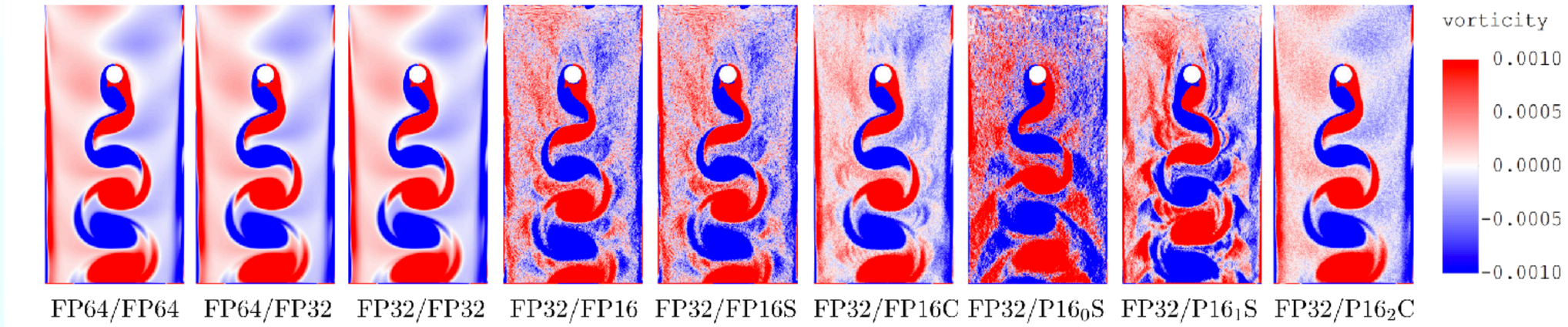
	Bits	n_m	b	Digits	ϵ	Range
IEEE FP64	64	52	1023	16.0	2.2×10^{-16}	$\pm1.797693\times10^{308}$
IEEE FP32	32	23	127	7.2	1.2×10^{-7}	$\pm3.402823\times10^{38}$
IEEE FP16	16	10	15	3.3	9.8×10^{-4}	$\pm6.550400\times10^{4}$
FP16S	16	10	30	3.3	9.8×10^{-4}	$\pm1.999023\times10^{0}$
FP16C	16	11	15	3.6	4.9×10^{-4}	$\pm1.999512\times10^{0}$
Posit $P16_0S$	16	0–13	–	$\leqslant$4.2	$\geqslant1.2\times10^{-4}$	$\pm1.280000\times10^{2}$
Posit $P16_1S$	16	0–12	0	$\leqslant$3.9	$\geqslant2.4\times10^{-4}$	$\pm2.097152\times10^{6}$
Posit $P16_2C$	16	0–12	0	$\leqslant$3.9	$\geqslant2.4\times10^{-4}$	$\pm1.999756\times10^{0}$

Case 1: Memory Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

"Ugh, fine but this data type better have been validated "

Sure has! And here's *real* data



Case 1: Memory Bound

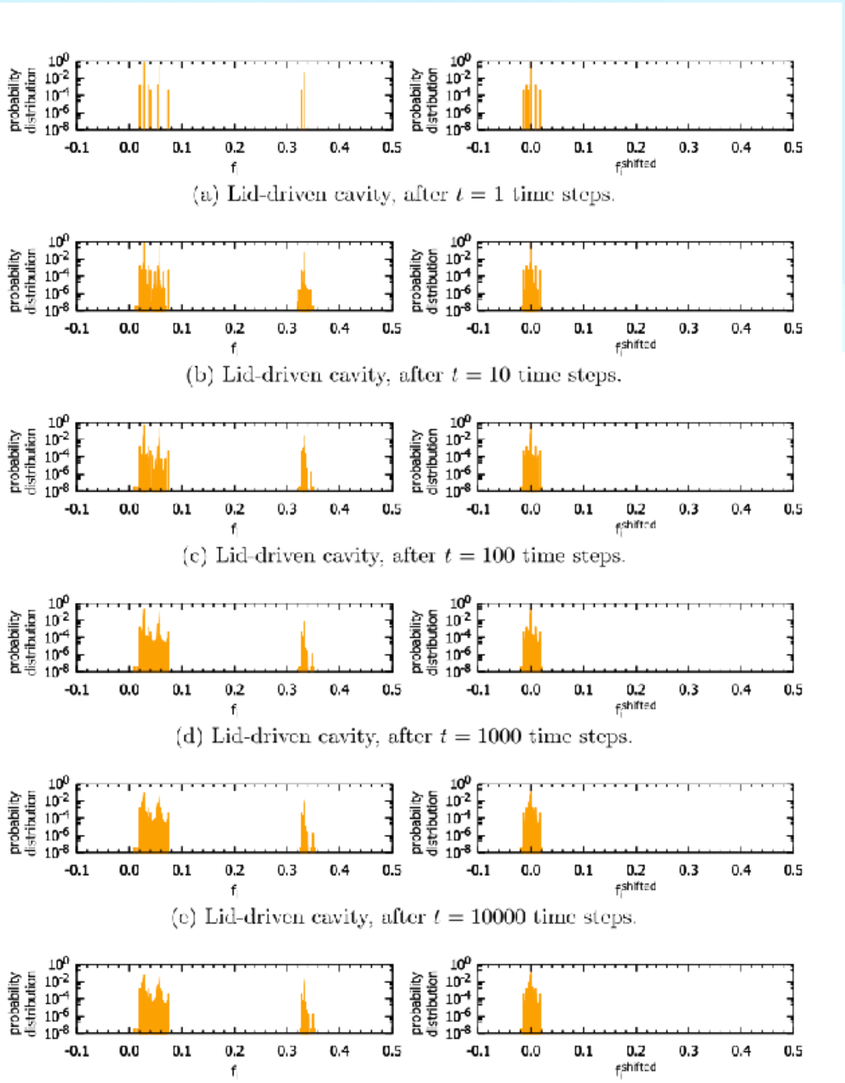
Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

"That looks like a mess "

Agreed, but some less than others.
And just like mixed FP32/FP64 or pure FP32, there are tricks we can play.

1. Density Distribution Functions
(DDF) shifting (specific to lattice Boltzmann method)
2. Adaptive precision based on temporal constraints
3. Adaptive precision based on positional constraints

Further readings and papers at the end



(a) Lid-driven cavity, after $t = 1$ time steps.

(b) Lid-driven cavity, after $t = 10$ time steps.

(c) Lid-driven cavity, after $t = 100$ time steps.

(d) Lid-driven cavity, after $t = 1000$ time steps.

(e) Lid-driven cavity, after $t = 10000$ time steps.

Case 2: Compute Bound

How does FP16 help us deal with this?

We get double the compute units vs FP32, meaning that if we're at 100% overall compute usage, and only 60% memory bandwidth, we can shift the problem to being bandwidth bound

We have a choice:

Scenario 1: All computation in FP16 No overhead, but loss of numerical accuracy. Up to 2x increase in compute throughput

Scenario 2: Store in FP16, compute in FP32. Adds overhead, better *overall* precision. Since we were already compute bound, we're not coming out ahead. Information lost to rounding, not as much as Scenario1

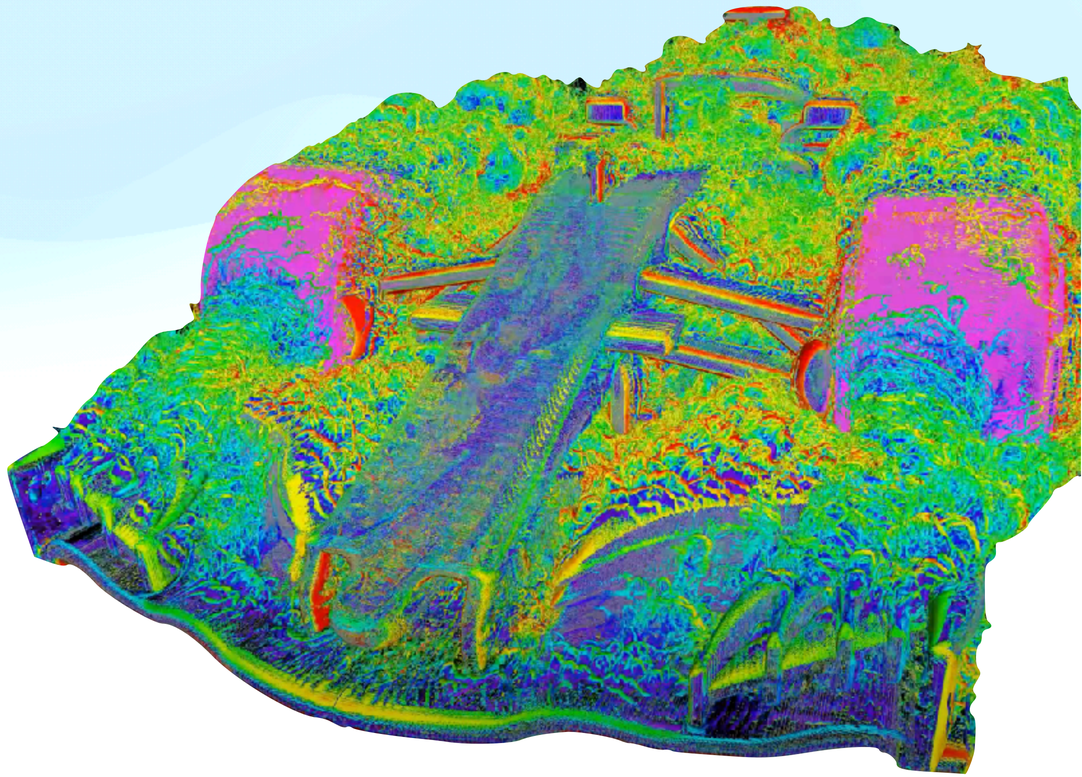
Case 2: Compute Bound

Scenario 1: All FP16 - HPC CFD

Unfortunately, as earlier, compute bound in FP16 for CFD isn't going to workout

Some accelerators (like GPUs) support 19 and 24 bit types that help with this. Typically these are AI types, which prefer range over precision

See Nvidia TF32 (19 bit type) or AMD FP24

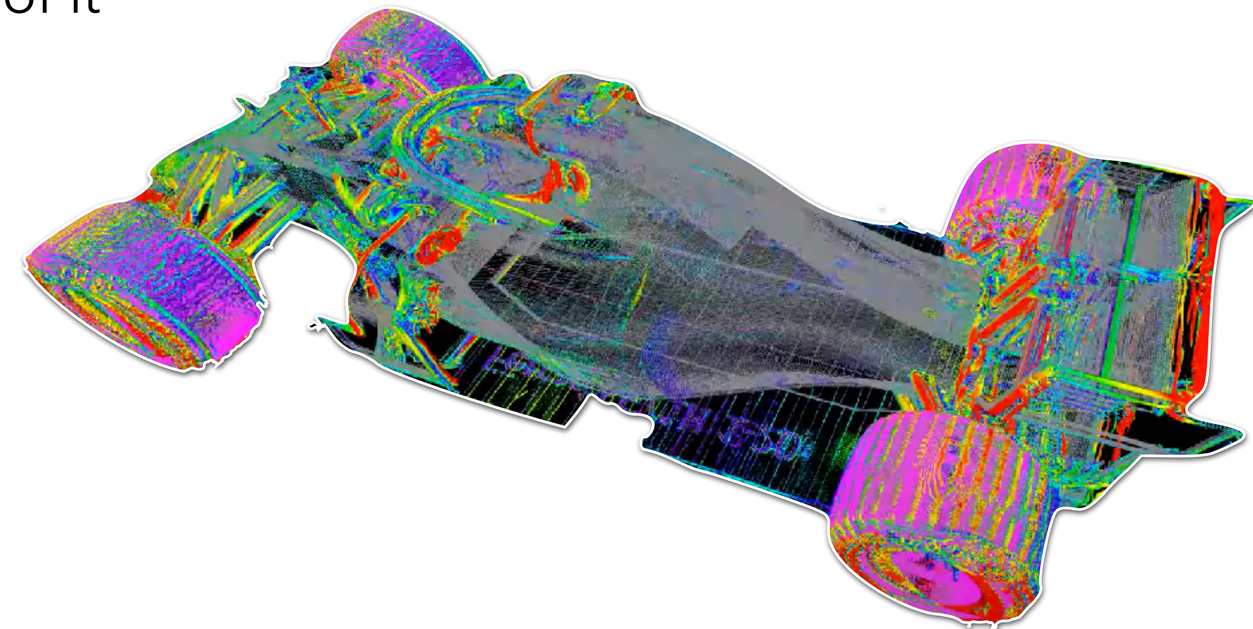


Case 2: Compute Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

Unfortunately, there isn't much point in *adding* compute overhead when we're already computationally bound.

Scenarios where we're compute bound are few and far between in the era, but still worth making a note of it



Case 2: Compute Bound

Scenario 2: Store in FP16-Compute in FP32 - HPC CFD

~~Unfortunately, there isn't much point in *adding* compute overhead when we're already computationally bound.~~

~~Scenarios where we're compute bound are few and far between in the era, but still worth making a note of it~~

However, due to the prevalence of reduced precision in AI inference, vendors have seen fit to add dedicated FP16 matrix accelerators to hardware

Case 2: Compute Bound

How does FP16 help us deal with this?

Let’s revisit the prior compute bound scenario, under the premise that FP16, instead of being twice is fast, is 4x faster in general compute, 8-64* times faster for HGEMM** operations.

We have a choice:

Scenario 1: All computation in FP16
No overhead, but loss of numerical accuracy.
Up to **64x** increase in compute throughput

Scenario 2: Store in FP16, compute in FP32.
Adds overhead, better *overall* precision.
Since we were already compute bound (**in FP32**), we’re not coming out ahead.
Information lost to rounding, not as much as Scenario1

*Nvidia V100 8:1, A100 16:1, H100 32-64:1
**HGEMM: Half precision General Matrix Multiply

Case 2: Compute Bound

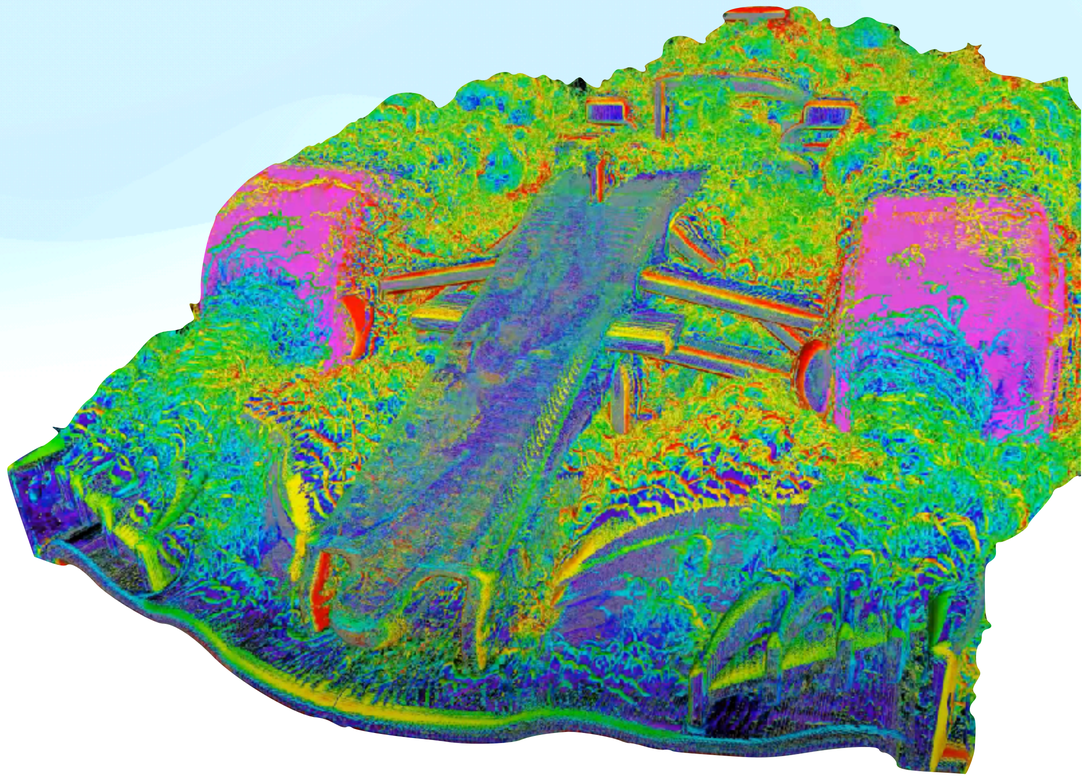
Scenario 1: All FP16 - HPC CFD

Unfortunately, as earlier, compute bound in FP16 for CFD isn't going to workout

We simply don't have enough precision to do the work we want to do

However, specific factors within CFD don't have issues related to precision in TF32/FP16.

If you have the engineering capital, you can make something out of it by splitting workloads



Case 2: Compute Bound

Scenario 1: All FP16 - HPC CFD

~~Unfortunately, as earlier, compute bound in FP16 for CFD isn't going to workout~~

~~We simply don't have enough precision to do the work we want to do~~

~~However, specific factors within CFD don't have issues related to precision in TF32/FP16.~~

~~If you have the engineering capital, you can make something out of it by splitting workloads~~

Here be dragons

So FP16: ***What's the takeaway?***

We're *almost* always memory bound

If your simulation can handle the slight loss in precision inherent to jumping between types, FP16 storage with FP32 compute is a game changer

Classical example:

DGEMM vs DGEMV: x86_64 cache line: 64 Bytes, AVX512 FP64 entries: 8

Level 2 BLAS DGEMV 5x5 1x5
25+5 64 bit numbers, 32 ops
Dot product=FMA

Instructions: packed FMA

```
vfmadd213pd zmm, zmm, zmm
vfmadd231pd zmm, zmm, zmm
vfmadd213pd zmm, zmm, zmm
vfmadd231pd zmm, zmm, zmm
```

Can solve in ~4 instruction, each uses 4 Cycles, memory takes ~100 Cycles

Level 3 BLAS DGEMM 5x5 5x5
25+25 64 bit numbers, 96 ops
Dot product=FMA

Instructions: packed FMA

```
vfmadd213pd zmm, zmm, zmm
vfmadd231pd zmm, zmm, zmm
…
vfmadd213pd zmm, zmm, zmm
vfmadd231pd zmm, zmm, zmm
```

Can solve in ~12 instructions, each takes 4 Cycles, memory takes ~100 Cycles

Relatable example:

DGEMM vs DGEMV: x86_64 cache line: 64 Bytes, AVX512 FP64 entries: 8

Level 2 BLAS DGEMV 5x5 1x5

Takes up 5 entire cache lines (25 entries in 64 Byte lines we need 4+1)

Takes ~16 cycles

Level 3 BLAS DGEMM 5x5 5x5

Takes up 8 entire cache lines (25 entries in 64 Byte lines we need 4+4)

Takes ~48 cycles

We've done much harder work, more of it, and we're still waiting on memory

“So there’s a place for it, but how do I (and my users) make use of this?”

Standard answer:
What's in your cluster?

The standards:

FP16 was only ratified as a specification in 2008 (IEEE 754-2008, supersedes 754-1985 and 854-1987)

The Software:

Fortran added support in 2018

C/C++ are getting support when the C(++) 23 spec is fully adopted

Python does not have support in its standard libraries, some AI libraries add support

MatLab: only via Cleve_lab, an experimental support library

Accelerator/GPU languages all have support since being added to hardware (ROCm, CUDA, OpenCL, SYCL)

The Hardware:

All x86 CPU's since 2014 (`x86_64 f16c`) but not math

The newest chips from Intel support it via AVX512FP16 (Sapphire Rapids, some Alder Lake) with math

ARM CPUs `ARMv8.2-FP16`

GPU's since Nvidia Pascal (2016), AMD Vega (2017), Intel (2022)

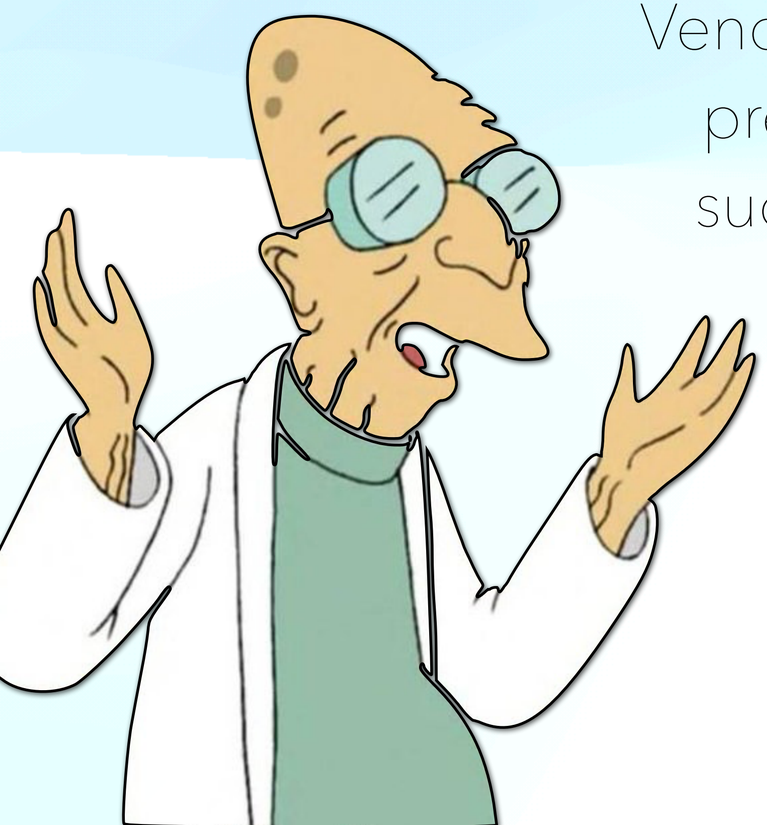
All FPGA's

AI accelerators from Graphcore, Groq, Cerebras, TensTorrent

“So if I have the hardware, I now need to also port my software?”

Yes, but we already port/redevelop whenever a new system is delivered by vendors. These sort of compute kernels aren't terribly complicated, no more so than standard uArch targeting.

But wait! Good news everyone!



Because the AI folks wanted this already,
Vendors have already created auto *mixed*
precision libraries that we can leverage,
such as cuSOLVER, cuTLAS from Nvidia
and oneDNN from Intel.

Case study: **Mixed precision Quantum Chromodynamics on Nvidia Tensor cores**

Joint academia, NL and vendor FOSS effort.

QUDA: QCD on CUDA

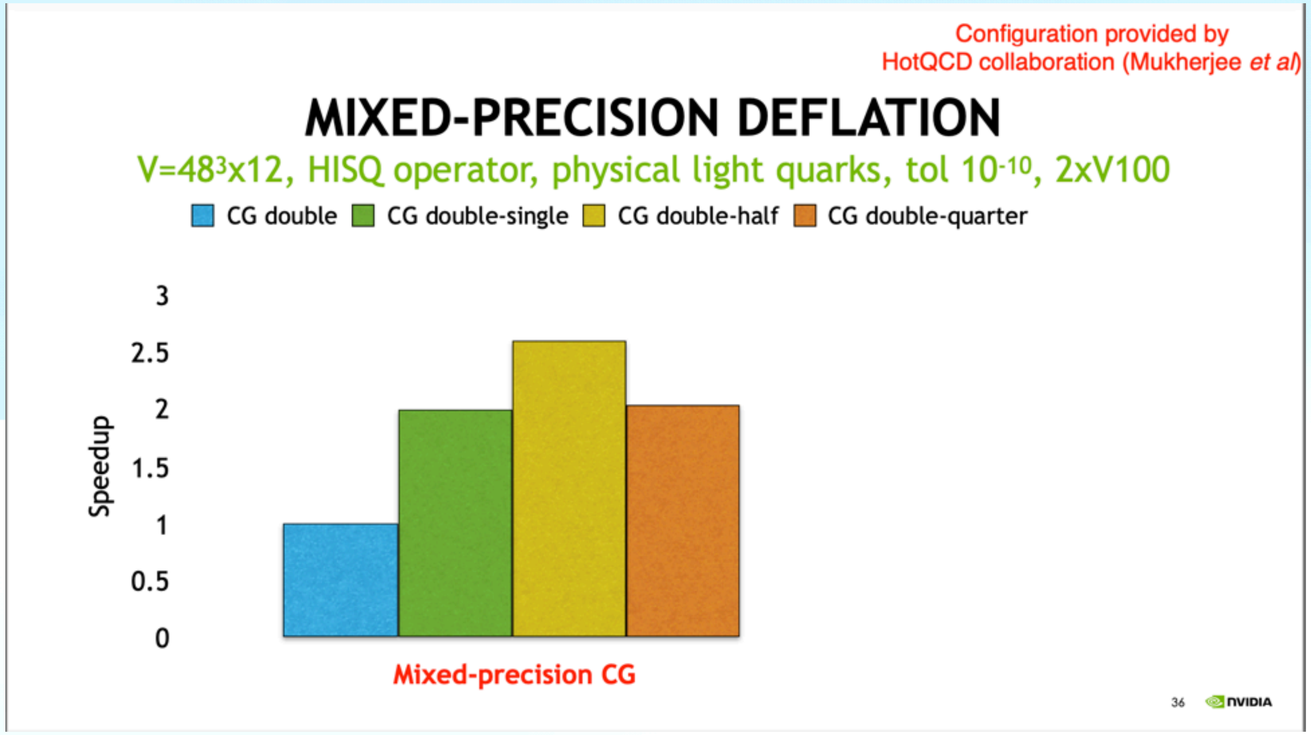
Motivation?

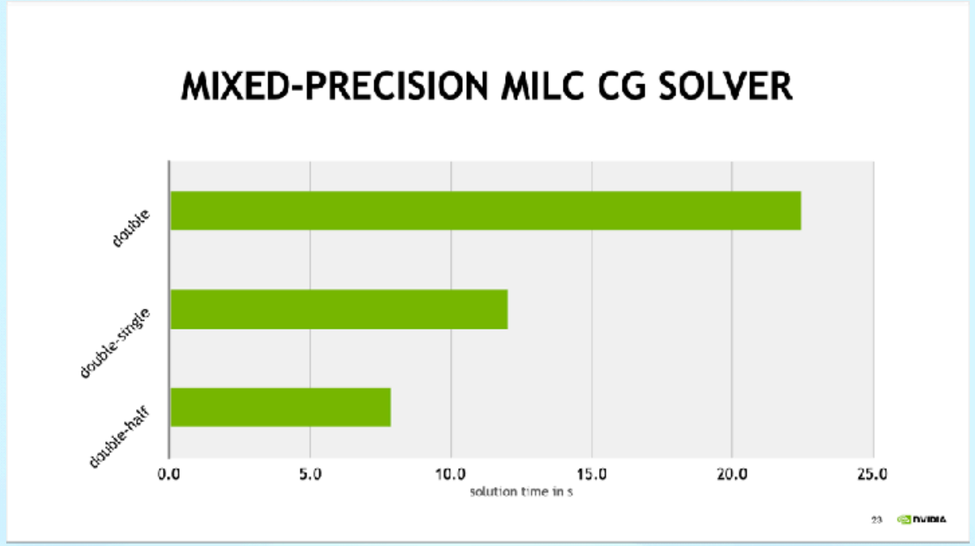
Leverage new techniques (reduced and mixed precision) to speed up extremely common jobs

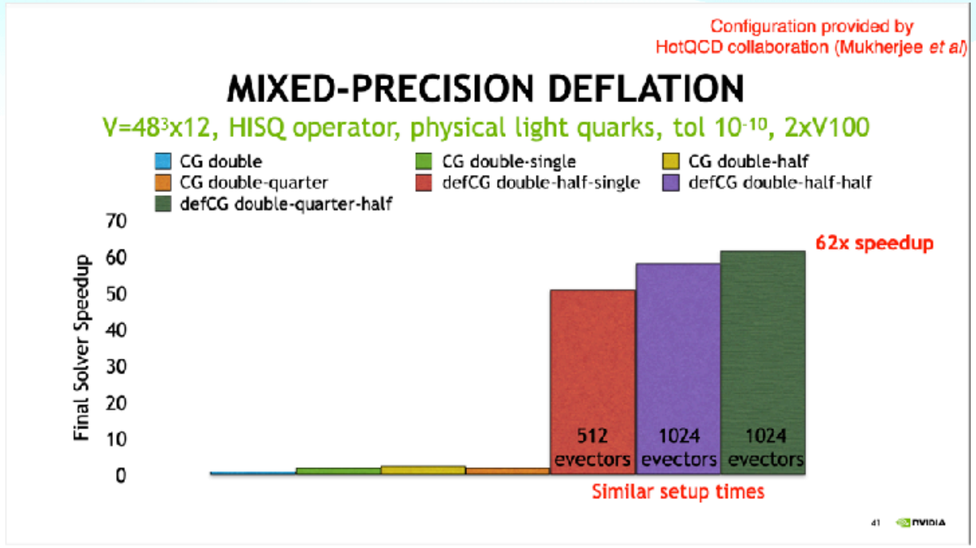
Make use of the beyond 2:1 performance in FP16 as compared to FP32 on recent Nvidia accelerators.

Minimize researcher/centre staff effort to make use of the above

Results?







https://developer.nvidia.com/gtc/2020/slides/s21725-effective-use-of-mixed-precision-for-hpc.pdf

“And what exactly are you doing about it?”

Building it in Free and Open Source Software

Hardware:

The quest for FP16 on AlderLake

- No hardware currently exists in market that officially supports AVX512FP16
- If you go through a ton of work, build custom Linux Kernels, mess with custom microcode, have specific CPUs launched within a specific set of dates, then and only then does it work

- I wrote about it here: https://gist.github.com/FCLC/56e4b3f4a4d98cfd274d1430fabb9458 and ended up talking to Intels engineering team about it.

Software:

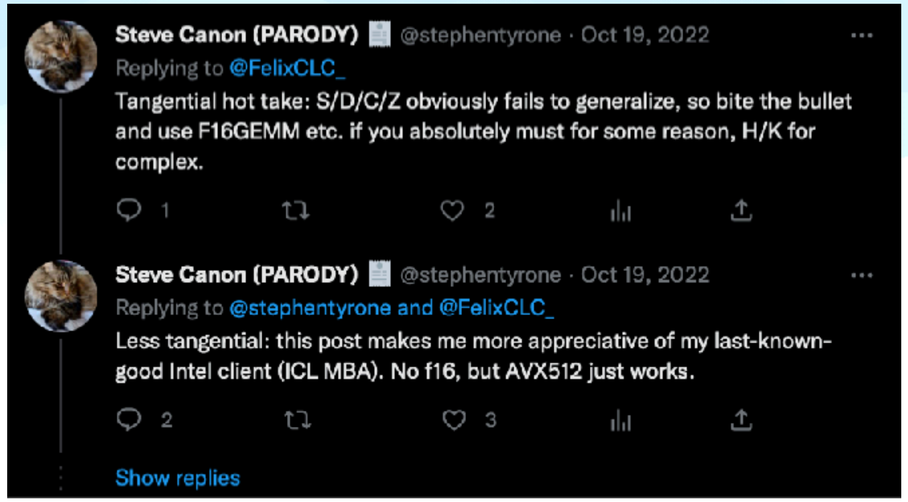
The quest for FP16 on AlderLake

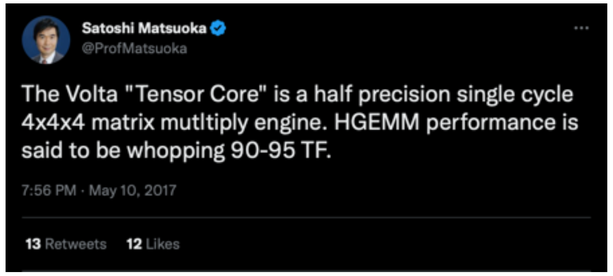
- With the extremely hacked together setup, it works well enough that I can develop HGEMM and HGEMV kernels as part of OpenBLAS
- It’s WIP, currently waiting on Intel for hardware that officially supports the AVX512FP16 ISA to be able to finalize the work/research
- Unfortunately, there’s now “standard” way to call FP16 kernels

Standards

The quest for FP16 on ~~AlderLake~~ Anything

- There is no official name for half precision kernels, be they real or complex
- Most have unofficially adopted HGEMM
- For now I'm working to YGEMM for complex FP16 kernels





Demo/PoC

Switch presenter to Linux Machine

Questions?

What else is there?

Temporal precision shifting

- Dynamically changing precision within a simulation dependent on complexity
- Mixed Classical HPC and ML
- Quick turn around simulations without the need for ML inference

Where to reach me?

Mastodon: FCLC@mast.hpc.social (best)
Twitter: @felixCLC_

FOSS email: felix.leclair123@hotmail.com (best)
Academic: 14fl11@queensu.ca

January 23 2023
ISO format: 2023/01/23