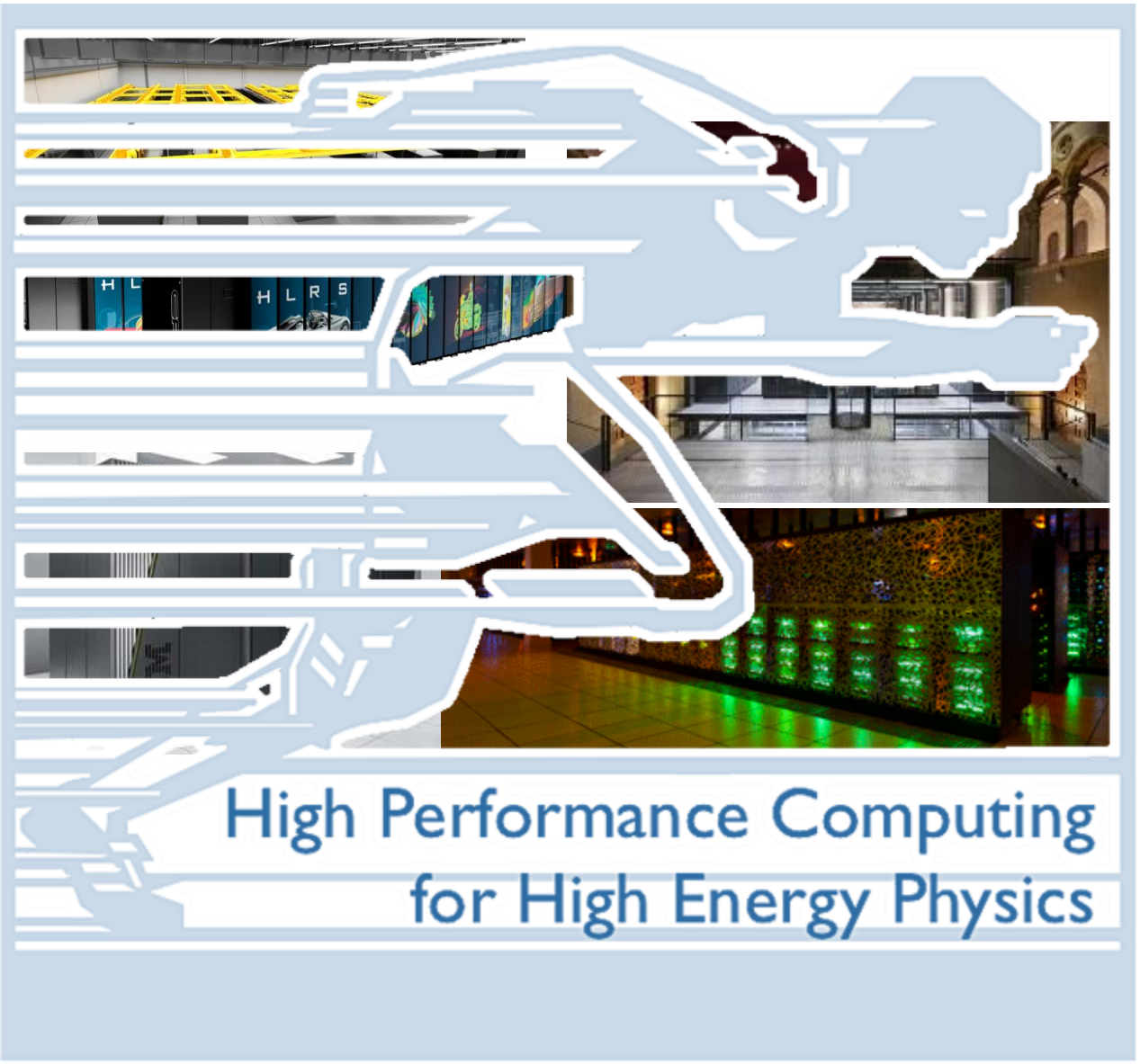
Using “float-pair” for precise and fast arithmetic on GPU

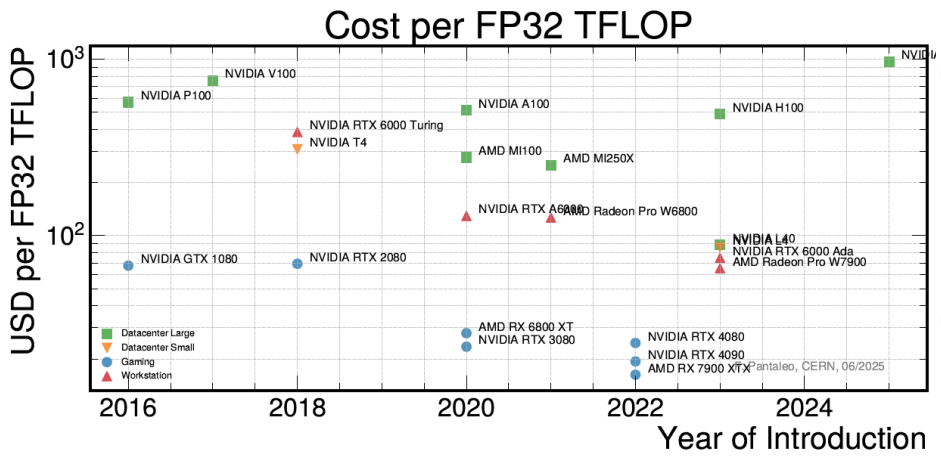


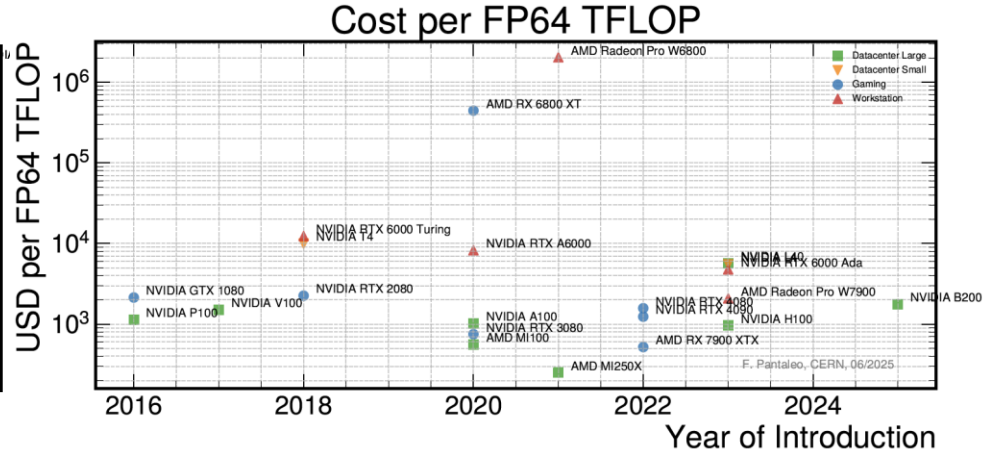
Vincenzo Innocente
CMS Experiment

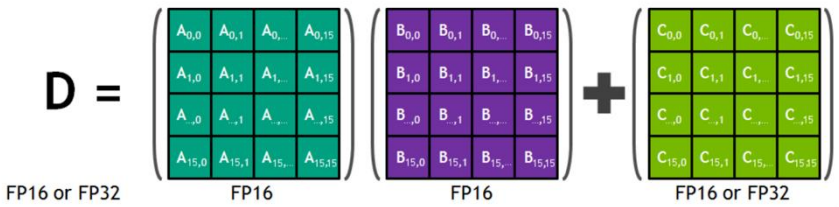
CERN, September 2025

Motivation:

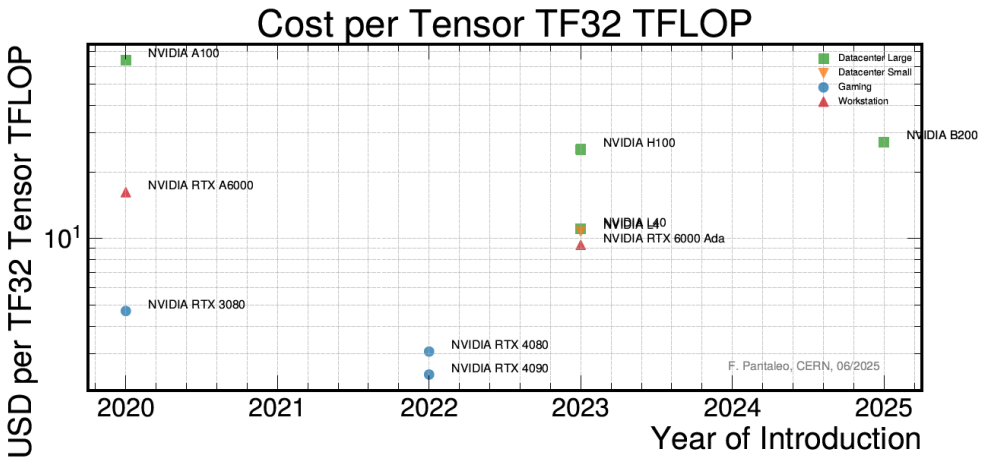
FP64 is 32 times slower than FP32 on affordable GPU







min rank for tensor is 16 (not 4)!



Methodology

- “double-word arithmetic” is a well known technique to improve precision beyond what available from HW
- Restricted to double-double in the past
- The ”evolution” of GPUs with the increased relevance of low-precision HW instructions (FP16, FP32 even int8) has spurred interest and research on how to “emulate” high precision arithmetic using them
  - See recent NGT workshop https://indico.cern.ch/event/1538409/timetable/ - all
- I coded a “TwoFloat” (float pair) C++ class that implement generic “double-word arithmetic” following the algorithms in the refences listed in next slide
  - TwoFloat is in principle a direct replacement of float or double in any already coded algorithm
  - (I’ve since then discovered a similar library by ESA https://github.com/esa-tu-darmstadt/twofloat)

References (EYAWK about double-word arith....)

- Mioara Maria Joldes, Jean-Michel Muller, Valentina Popescu. Tight and rigorous error bounds for basic building blocks of double-word arithmetic. ACM Transactions on Mathematical Software, 2017, 44 (2), pp.1 - 27. .10.1145/3121432.. https://hal.science/hal-01351529v3
- Vincent Lefèvre, Nicolas Louvet, Jean-Michel Muller, Joris Picot, Laurence Rideau. Accurate Calculation of Euclidean Norms using Double-Word Arithmetic. ACM Transactions on Mathematical Software, 2023, 49. https://hal.science/hal-03482567
- S. M. Rump and M. Lange. Faithfully Rounded Floating-Point Computation. ACM Transactions on Mathematical Software, 46(3):1–20, jul 2020. https://www.tuhh.de/ti3/paper/rump/LaRu2017b.pdf
- https://gitlab.inria.fr/core-math/core-math

Table 1. Summary of the Results Presented in This Paper

Operation	Algorithm	Previously known bound	Our bound	Largest relative error observed in experiments	♯ of FP ops
DW + FP	Algorithm 4	?	$2u^2$	$2u^2 - 6u^3$	10
DW + DW	Algorithm 5	N/A	N/A	1	11
	Algorithm 6	$2u^2$ (incorrect)	$3u^2 + 13u^3$	$2.25u^2$	20
DW × FP	Algorithm 7	$4u^2$	$\frac{3}{2}u^2 + 4u^3$	$1.5u^2$	10
	Algorithm 8	?	$3u^2$	$2.517u^2$	7
	Algorithm 9	N/A	$2u^2$	$1.984u^2$	6
DW × DW	Algorithm 10	$11u^2$	$7u^2$	$4.9916u^2$	9
	Algorithm 11	N/A	$6u^2$	$4.9433u^2$	8
	Algorithm 12	N/A	$5u^2$	$3.936u^2$	9
DW ÷ FP	Algorithm 13	$4u^2$	$3.5u^2$	$2.95u^2$	16
	Algorithm 14	N/A	$3.5u^2$	$2.95u^2$	10
	Algorithm 15	N/A	$3u^2$	$2.95u^2$	10
DW ÷ DW	Algorithm 16	?	$15u^2 + 56u^3$	$8.465u^2$	24
	Algorithm 17	N/A	$15u^2 + 56u^3$	$8.465u^2$	18
	Algorithm 18	N/A	$9.8u^2$	$5.922u^2$	31

For each algorithm, we give the previously known bound (when we are aware of it, and when the algorithm already existed), the bound we have proved, the largest relative error observed in our fairly intensive tests, and the number of floating-point operations required by the algorithm.

Sum is sensitive to cancellations
Summing two positive DW can cost as low as 8 FP ops.
Algorithm 5 is knows as “SloppySum”: it is correct “most” of the time.
Algorithm 6 is precise: still twice as expensive

From this table we can estimate a speed up of a factor ~2 even 3 w/r/t double precision when using float-pair on an affordable GPU

What can go wrong?

1. float-pair may introduce additional overhead (reduced optimization)
2. Cost of a kernel in double may be way less than 32*float-cost due to memory access, register pressure etc that overwhelm the cost of arithmetic

Tools

- Measuring latency on GPU
  - clock64: returns the value of a per-multiprocessor counter that is incremented every clock cycle
    - Very reproducible
  - %%globaltimer: 64-bit global nanosecond timer: essentially wall-clock
    - Used in cuda::std::chrono::system_clock
    - Gives identical results as the above
- Counting instructions
  - perf on AMD Bergamo provides detailed counters for all Arith instructions
  - on INTEL separates single and double precision
- Counting arith operations in code
  - tr –cd '*' | wc etc

A first Benchmark

Brute-force inversion of 5x5 Pos-Def Matrix using Cholesky decomposition

- no pivoting, no rescaling, no regularization
- https://github.com/cms-sw/cmssw/blob/master/DataFormats/Math/interface/choleskyInversion.h
- https://github.com/cms-sw/cmssw/blob/master/DataFormats/Math/test/CholeskyInvert_t.cpp

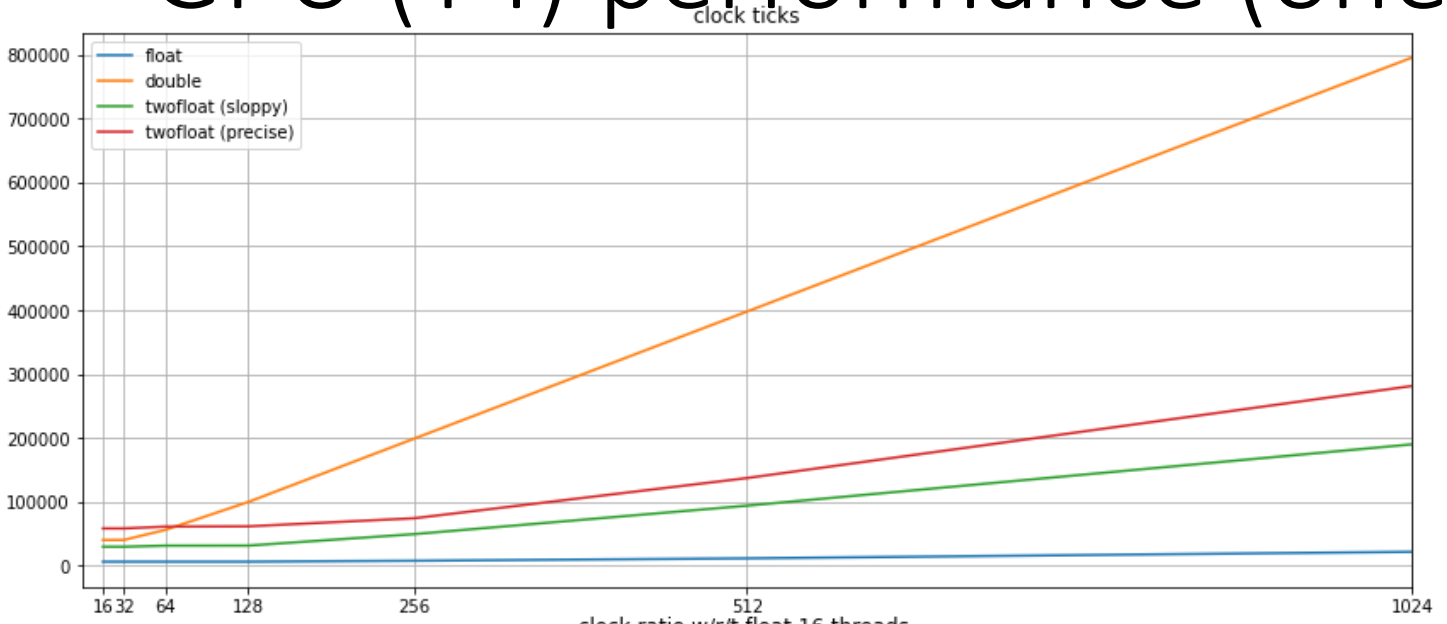
- 33 + , 29 - , 126 * , 5 / , no sqrt
  - Compiler optimization and fma may reduce these values
- On CPU: 5M random generated matrices and recursively inverted twice
- ON GPU: one random generated matrix per thread and recursively inverted 10M times
  - Weak scaling
- Precision test:
  - Max difference between elements of original matrix and the resulting one

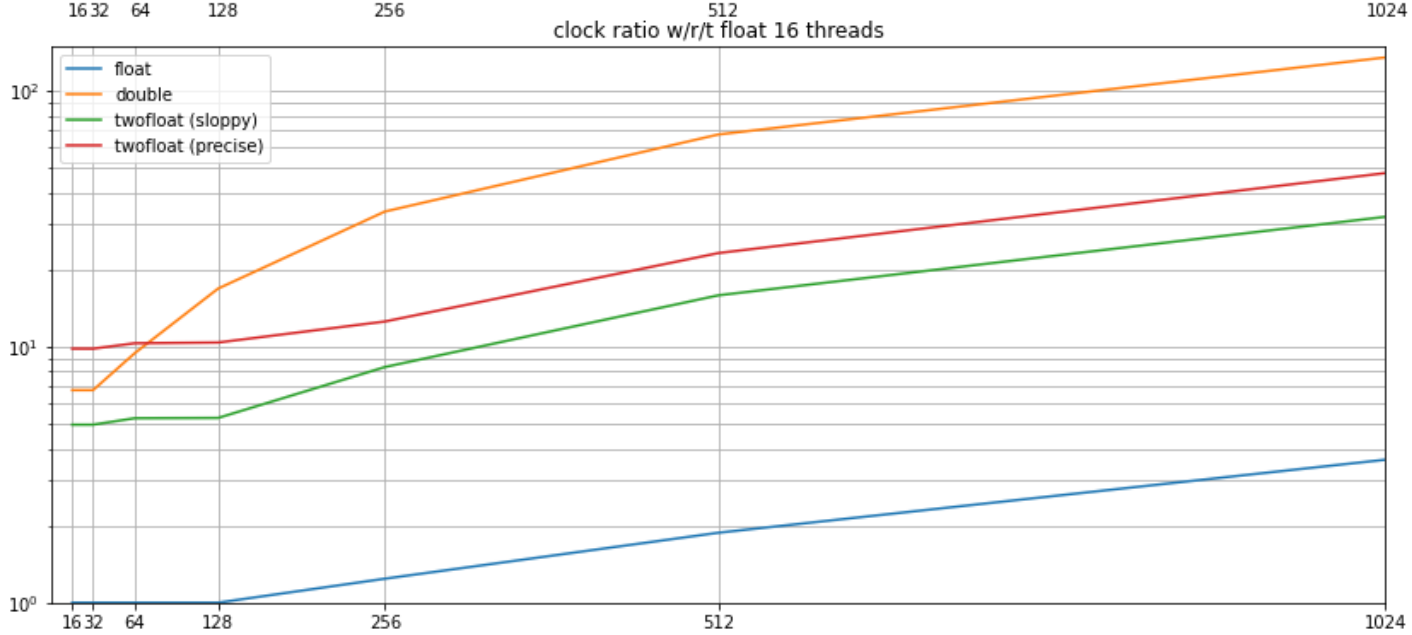
CPU Performance (AMD Bergamo)

	no-inv (float)	float	2float sloppy	2float precise	double	2double sloppy	2double precise
precision	N/A	$5.8e^{-5}$ 0.3	$9.6e^{-12}$ $2.3e^{-8}$	$7.0e^{-12}$ $2.0e^{-8}$	$1.0e^{-13}$ $5.4e^{-10}$	$8.9e^{-30}$ $6.8e^{-26}$	$1.0e^{-29}$ $7.4e^{-26}$
cycles	317.2	436.0	1612.2	2317.0	458.5	1487.1	2071.2
instr	822.0	927.5	3143.7	4025.3	938.0	2983.5	3785.7
Arith instr	164.0	287.5	1427.1	2005.7	285.6	1425.9	2003.2
mul	32.5	95.6	159.0	160.5			
fma	2.5	46.5	371.6	376.0			
add	32.5	38.5	344.3	565.6			
sub	7.5	13.5	446.6	797.0			
div	7.5	12.5	25.0	25.0			

(1427-164)/(287.5-164) = 10 (2006-164)/(287.5-164) = 15

GPU (T4) performance (one block, N threads)

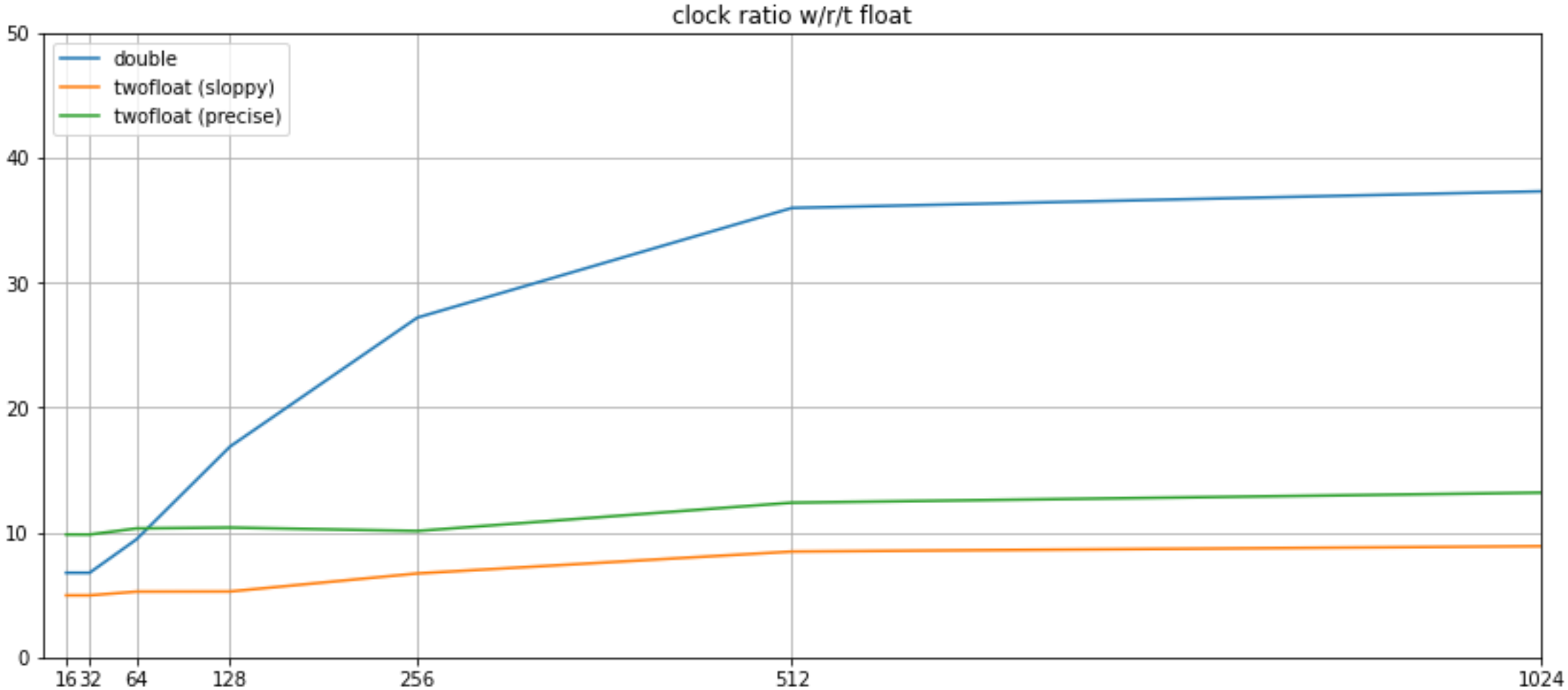




Ideally flat till 64 than linear growth

Float: flat till 128: 512 only ~2 times slower
Apparently (for this benchmark) a SM can fit 8 times more work (thread time-slicing)
Double: not surprising is only ~9 time slower @64
Is 32 times slower than float only at 512
DW-Arith follows float with a plateat at x8
for sloppy-sum and x12 for precise-sum
(see next slide)

GPU performance



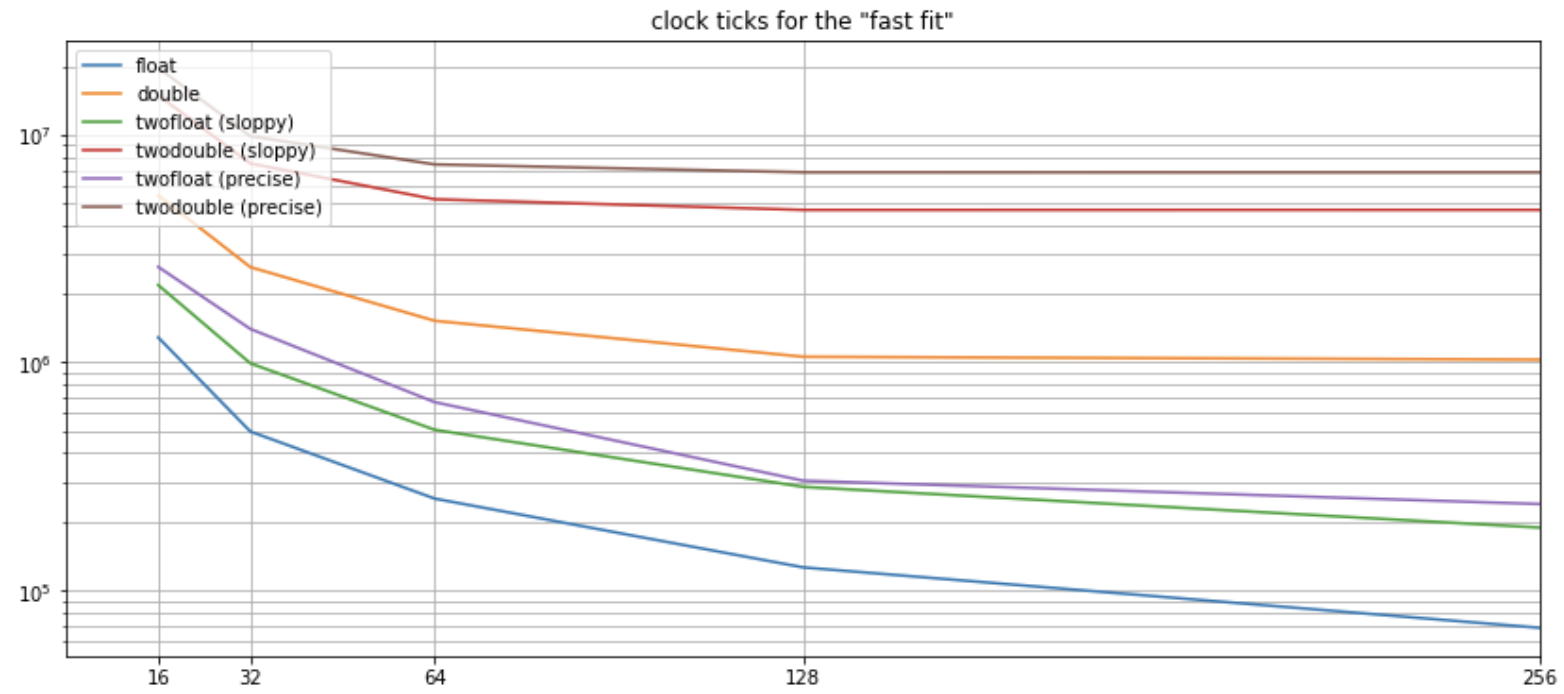
A more realistic benchmark

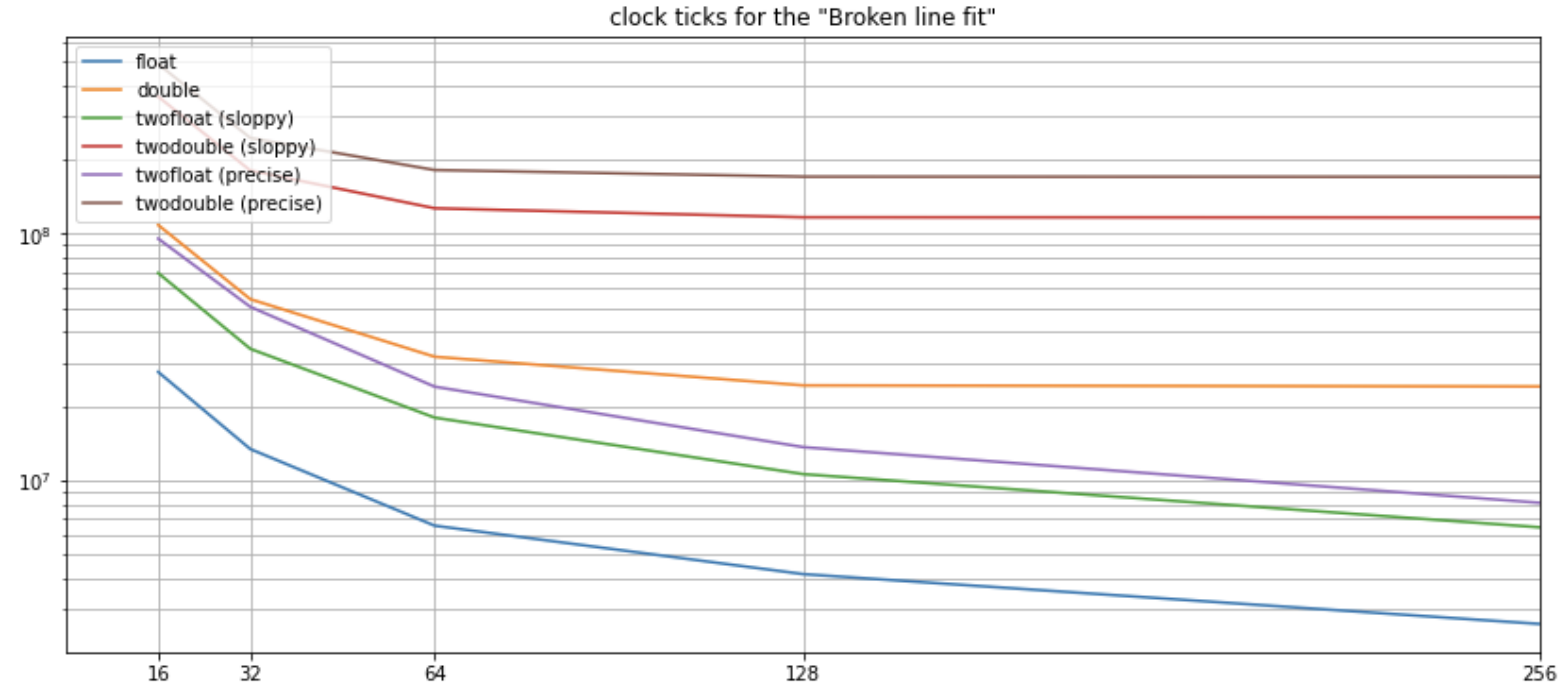
- BrokenLine fit
  - Modified version of
    - https://github.com/cms-patatrack/pixeltrack-standalone/blob/master/src/cuda/plugin-PixelTriplets/BrokenLineFitOnGPU.h
    - https://github.com/cms-patatrack/pixeltrack-standalone/blob/master/src/cuda/test/testEigenGPU.cu
  - Eigen works out of the box once implicit conversion is implemented
    - Eigen does not support mixed precision
  - Need to make sure NO (double precision) literals are left in the code as the compiler will silently promote everything to *double*
    - Not sure about eigen itself
- Benchmark: fit 16K 4-hit-tracks in N threads (one block)
  - Strong scaling

Precision

- FastFit (circle x0,y0,R from 3 points)
  - ~correct even in *float* up to 1 TeV
- BrokenLine fit
  - *double:* ok up to 1000 TeV
  - *float:* breaks down between 100 and 500 GeV
  - DW-*float:* ok up to ~500 TeV
- GPU results differ form CPU when computation starts to break down.
  - difference in compiler generated fma? Should not happen for DW arith

single-precision is most probably sufficient for a fit of 4 hits once p_t is limited





Stop at 256 as it does not run at 512

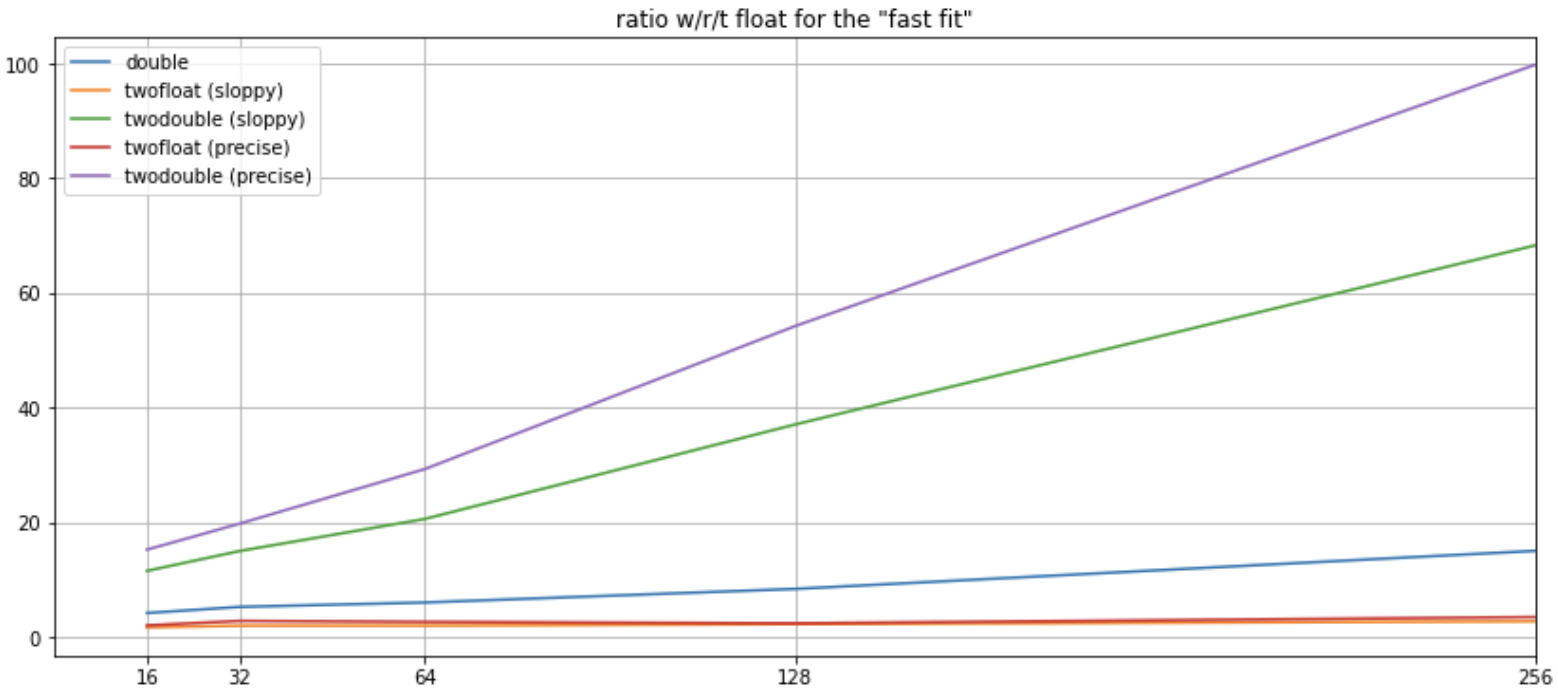
Ideally expect 4x speedup from 16 to 64 then flat

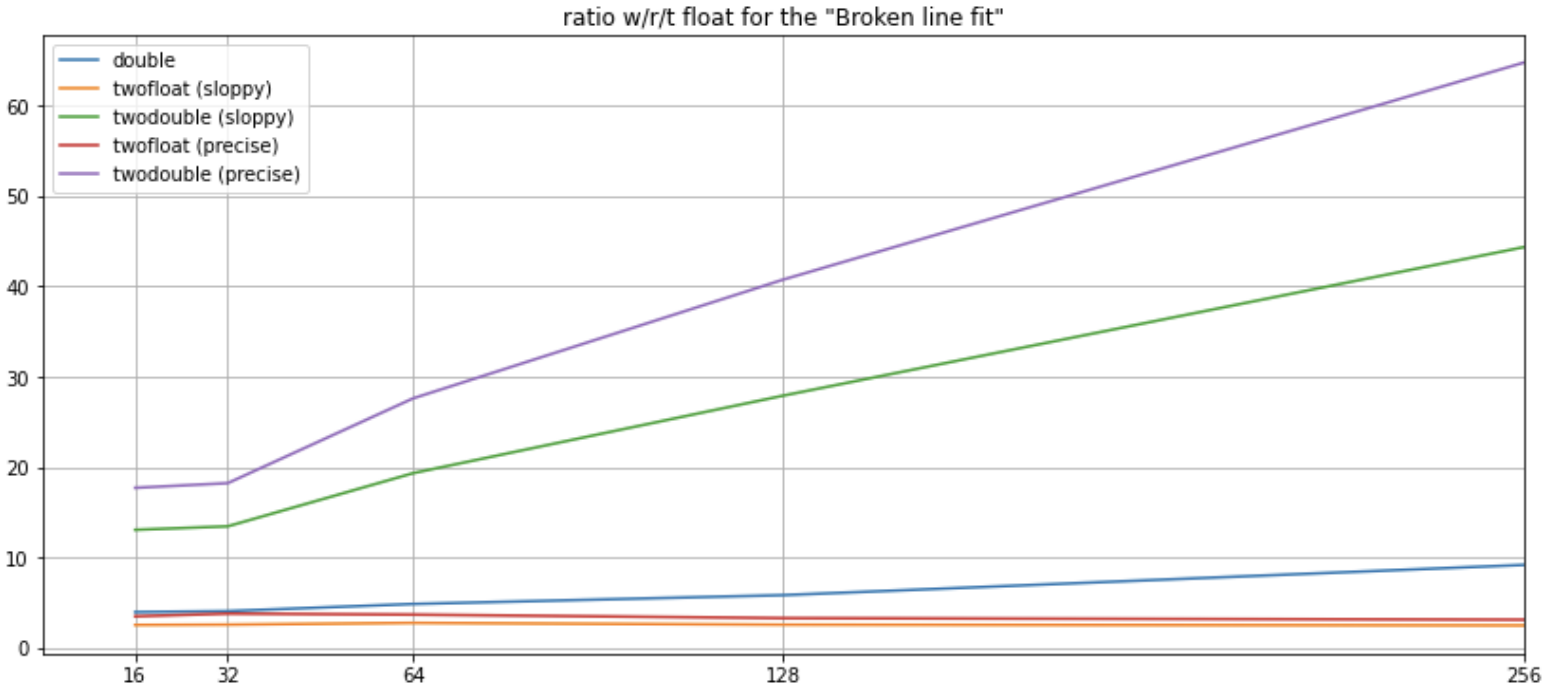
Observe for float 2x 16->32, 10x 16->256

Double precision do saturate at 128 (even 64) for the heaviest load

There is a factor x20 between double and float for the heaviest load (precise DW-Arith @256)

DW float is ~3.5 faster than double precision


ratio w/r/t float for the "fast fit"
double
twofloat (sloppy)
twodouble (sloppy)
twofloat (precise)
twodouble (precise)
0
20
40
60
80
100
16
32
64
128
256


ratio w/r/t float for the "Broken line fit"
double
twofloat (sloppy)
twodouble (sloppy)
twofloat (precise)
twodouble (precise)
0
10
20
30
40
50
60
16
32
64
128
256

Summary and conclusions

- Double precision arithmetic is indeed up to 32 (even more) slower than single precision on affordable GPUs
  - At least for my benchmarks
- Double-Word “float” arithmetic provide higher precision than float and is >3 times faster than double precision with full SM loads
- More tests with other workloads and in a production environment are of course required for a final assessment