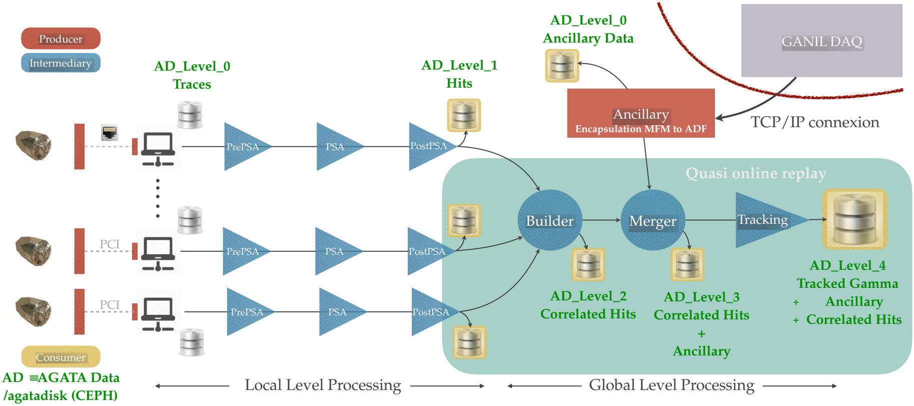
Speeding PSA
with half-precision and GPU

Roméo Molina Vincent Lafage

IJCLab, CNRS/IN2P3 & Université Paris-Saclay, Orsay, France

11th September 2024

AGATA Data flow[1]



[1] O. Stézowski, AGATA Meeting 2022

Using low precisions is promising

		Number of bits Signif. (t)	Exp.	Range	$u = 2^{-t}$
fp128	quadruple	113	15	$10^{\pm 4932}$	1×10^{-34}
fp80	long double	64	15	$10^{\pm 4932}$	5×10^{-20}
fp64	double	53	11	$10^{\pm 308}$	1×10^{-16}
fp32	single	24	8	$10^{\pm 38}$	6×10^{-8}
fp16	half	11	5	$10^{\pm 5}$	5×10^{-4}
bfloat16	half	8	8	$10^{\pm 38}$	4×10^{-3}
fp8 (e4m3)	quarter	4	4	$10^{\pm 2}$	6×10^{-2}
fp8 (e5m2)	quarter	3	5	$10^{\pm 5}$	1×10^{-1}

- Low precision increasingly supported by hardware
- **Great benefits:**
  - Reduced **storage**, data movement, and communications
  - Reduced **energy** consumption ($5\times$ with fp16, $9\times$ with bfloat16)
  - Increased **speed** ($16\times$ on A100 from fp32 to fp16/bfloat16)

Floating-point arithmetic

Floating-point computation $\neq$ mathematical evaluation

- rounding $a \oplus b \neq a + b$
- no more associativity $(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$

Consequences:

- invalid results
- non reproducibility
- performance issue (useless iterations)

Some limitations to the low precisions: (= low resolution)

- Low accuracy
- Narrow range

$\Rightarrow$

multiplication: good ; substraction : bad

Discrete Stochastic Arithmetic (DSA)

Classic arithmetic

$$A \oplus B \longrightarrow R$$

$R =$3.14237654356891

DSA

Random rounding

$$A_1 \oplus B_1 \longrightarrow R_1$$

$$A_2 \oplus B_2 \longrightarrow R_2$$

$$A_3 \oplus B_3 \longrightarrow R_3$$

$R_1 =$**3.14**1354786390989
$R_2 =$**3.14**3689456834534
$R_3 =$**3.14**2579087356598

- each operation executed 3 times with a random rounding mode
- number of correct digits in the results estimated using Student's test with the confidence level $95\,\%$
- operations executed synchronously
  - ⇒ detection of numerical instabilities (ex: `if (A>B)` with `A-B` numerical noise)
  - ⇒ optimization of stopping criteria to avoid useless iterations

Assess the accuracy

- implements stochastic arithmetic for C/C++ or Fortran codes
- all operators and mathematical functions overloaded $\Rightarrow$ little code rewriting
- support for MPI, OpenMP, GPU, vectorised codes
- supports emulated ou native half precision
- one CADNA execution: accuracy of any result, complete list of instabilities

CADNA cost

- memory: $\times 4$
- run time $\approx$ $\times 10$

CADNA **validates** `fp32` **results**

- PSA performed natively in `fp32`
- minimum search in a 504-dimensional space

  … as in 56 time steps times 9 segments
- risk to accumulate catastrophic cancellations

$$\chi = \sum_{s,t} \left| S^{\text{mes}}_{s,t} - S^{\text{ref}}_{s,t} \right|^{p=0.3}$$

- requires instrumentation to assess the accuracy results

⇒ code sensitive to perturbations?

- but $0.02\,\%$ of points matched differently between `fp64` and original `fp32` version
- only $0.02\,\%$ between CADNA version and original version

⇒ Satisfactory original fullgrid PSA results!

CADNA **validates** `fp32` **results**

- PSA performed natively in `fp32`
- minimum search in a 504-dimensional space

  … as in 56 time steps times 9 segments
- risk to accumulate catastrophic cancellations

$$\chi = \sum_{s,t} \left| S^{\text{mes}}_{s,t} - S^{\text{ref}}_{s,t} \right|^{p=0.3}$$

- requires instrumentation to assess the accuracy results

⇒ code sensitive to perturbations?

- but $0.02\,\%$ of points matched differently between `fp64` and original `fp32` version
- only $0.02\,\%$ between CADNA version and original version

⇒ Satisfactory original fullgrid PSA results!

CADNA **validates** `fp32` **results**

- PSA performed natively in `fp32`
- minimum search in a 504-dimensional space

  … as in 56 time steps times 9 segments
- risk to accumulate catastrophic cancellations

$$\chi = \sum_{s,t} \left| S_{s,t}^{\text{mes}} - S_{s,t}^{\text{ref}} \right|^{p=0.3}$$

- requires instrumentation to assess the accuracy results

⇒ code sensitive to perturbations?

- but $0.02\,\%$ of points matched differently between `fp64` and original `fp32` version
- only $0.02\,\%$ between CADNA version and original version

⇒ Satisfactory original fullgrid PSA results!

Turn it into half computation (CPU)

- emulated `fp16`
- $7.76\,\%$ differences between original `fp32` and `fp16` version
- too much?
- need to find another way to exploit low precision

Mixed precision algorithms

Mix several precisions in the same code with the goal of

- Getting the performance benefits of low precisions
- While preserving the accuracy and stability of high precision

⇒ **Why does it make sense to make the precision vary?**

- Because not all computations are equally "important"!
  Example:

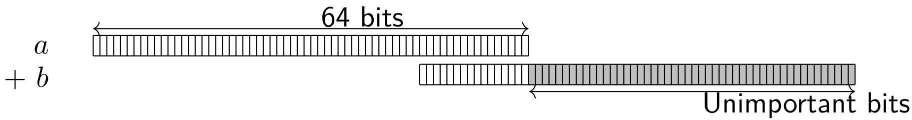
Mixed precision algorithms

Mix several precisions in the same code with the goal of

- Getting the performance benefits of low precisions
- While preserving the accuracy and stability of high precision

⇒ **Why does it make sense to make the precision vary?**

- Because not all computations are equally "important"!
  Example:



Coarse in half, fine in float

- first step in half
- second step in float
- $8.55\,\%$ differences with fullgrid `fp32` version
- under the same conditions, half-half produces $14.04\,\%$ differences!

Results $'23$

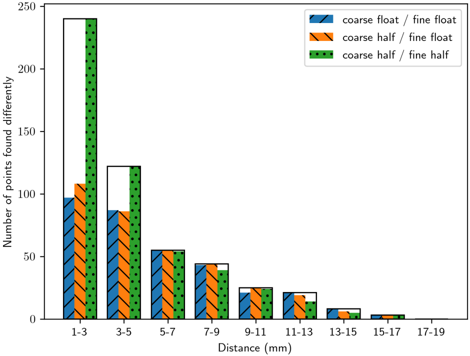


Figure: Distances between points found by the full grid `fp32` algorithm and alternative methods

preGPU

- we already saw vectorised version on CPU
- we also tried emulated `fp16` on CPU
- ⇒ first, extract a minimum, standalone version of PSA on CPU
  `https://gitlab.com/romeomolina/psa-test-env.git`
- ⇒ then, turn to modern `C++` conventions
  - `const`
  - `auto`
  - `constexpr`

GPU

- code should bind neatly to GPU
  as concurrency is clearly expressed
- moving it on GPUs to exploit `fp16` half-precision hardware
  we will show our `CUDA` implementation, to keep using CADNA
- `CUDA` *vs* `OpenACC` / `OpenMP` : better performance,
- ...less portability (NVidia only),
- ...more coding effort

GPU with CUDA

```
__global__ void gpu_samp_loop(float* hitSegAmp, float* corSegAmp, float* baseAmp, int* baseGrid
     , float* chi2, int numPts){
  //constexpr auto baseScale = PF.baseScale*RESCALE;  // scaling signals to data, including
       expansion factor for mapped metric
        const float baseScale = 0.457844;
        const int iCore = 36;
        const int netChSeg = 34;
        const int jPts = blockIdx.x * blockDim.x + threadIdx.x;
        if(jPts < numPts){
                const float *baseTrace1 = baseAmp + jPts*LOOP_SAMP*TCHAN + netChSeg*LOOP_SAMP;
                const float *baseTrace2 = baseAmp + jPts*LOOP_SAMP*TCHAN + iCore*LOOP_SAMP;
                float chi2_local = 0.0f;
                for(auto nn = 0U; nn < LOOP_SAMP; nn++) {
                        f_type fdiff = hitSegAmp[nn] -  baseScale * baseTrace1[nn];
                        chi2_local += exp2f(log2f(fabs(fdiff))*chiExponent);
                }
                for(auto nn = 0U; nn < LOOP_SAMP; nn++) {
                        f_type fdiff = corSegAmp[nn] -  baseScale * baseTrace2[nn];
                        chi2_local += exp2f(log2f(fabs(fdiff))*chiExponent);
                }
                chi2[jPts] = chi2_local;
        }
}
```

Results $'24$

Execution time for the different configurations on CPU and GPU (ticks)

	CPU-FP32	GPU-FP32	GPU-FP16
FGS-NOLUT	624	55	52
FGS-LUT	97	51	–
CFGS-NOLUT	102	–	–
CFGS-LUT	17	–	–

Points identified within 5mm of those found by reference (FGS-FP32 without the LUT executed in CPU %)

	CPU-FP32	GPU-FP32	GPU-FP16
FGS-NOLUT	100	100	94
FGS-LUT	90	90	–
CFGS-NOLUT	72	–	–
CFGS-LUT	68	–	–

sample of 5342 events with energies ranging from 15 keV to 5 MeV

Room for improvement

- CPU experiments on an Intel® Core™ i9-11950H Processor with 8 cores at 2.6GHz with 24MB cache
- GPU experiments on a NVIDIA RTX A2000 with 3328 CUDA cores and 4GB memory.

⇒ increase the occupancy of the GPU cores, suggesting a possible acceleration up to a factor $\times 15$ on larger GPUs

- GPU `fp16` really bear fruits with tensor cores...Can we express the computation as a matrix product?

Conclusion

- low precision is beneficial (speed, energy, storage)
- accuracy control is mandatory
- CADNA is well designed to do so
- mixed-precision is a way to benefit from low precision while keeping good accuracy
- PSA on GPU (CUDA)
- similar results between uniform precision and mixed precision for PSA

To improve optimisation of PSA:

- more events should be put simultaneously on the GPU to really benefit of GPU
- have the coarse/fine grid size vary
- have a hierarchy of intermediate grids

... address PrePSA