High performance numerical validation with stochastic arithmetic

Pacôme Eberhart
Joint work with : Fabienne Jézéquel, Pierre Fortin

In collaboration with Julien Brajard from LOCEAN

RAIM2015
April 7, 2015
IRISA, Rennes

Estimation of rounding error propagation

Evaluating the accuracy of numerical results

- Accumulation of rounding errors $\Rightarrow$ numerical results different from mathematical results
- Measure of the reliability and reproducibility of the computation
- Particularly important in HPC environments and future exascale supercomputers
  - increased parallelism
  - higher amount of computation

Some methods

- Backward error analysis: low overhead, unfit for some types of code
- Interval arithmetic: 100% accurate but usually needs code rewriting
- Stochastic arithmetic: probabilistic approach easy to use in real-life applications
  - need to reduce overhead for high performance

Stochastic arithmetic and the CADNA library

CESTAC method

- Each arithmetic operation is performed N times
- Randomly rounded towards $+\infty$ or $-\infty$ with probability 0.5
- Number of exact significant digits estimated with statistical analysis
- First order approximation method : validity compromised if second order errors greater than first order

Implementation of the CADNA library

- Implementation of stochastic arithmetic in C/C++
- Classes and operator overloading for ease of use
- Contains $N = 3$ floating-point values and 1 integer

The CADNA library: self-validation and anomaly detection

Anomaly detection

- Self-validation to ensure validity of stochastic arithmetic
- Anomaly detection for numerical analysis of the code

Warning types

- **Self-validation**: both operands in a multiplication or a divisor not significant
- **Cancellation detection**: sudden loss in accuracy on addition or subtraction
- **Mathematical instability**: instability in a mathematical function
- **Branching instability**: undeterminism in a branching test

Overhead

Computation time

- Depends on the program and the level of detection
- Is usually one order of magnitude higher or more on real-life applications
- Even higher on highly optimised routines

Causes

- Cost of anomaly detection
- Cost of stochastic operations

Cost of anomaly detection

Detection types

- Self-validation and branching instability: relatively low cost test
- Mathematical instability: inexpensive compared to the cost of mathematical function calls
- Cancellation detection: computing the number of exact significant digits of both operands and the result

Calculating the number of exact significant digits

- Uses the mean value and the standard deviation of the set of samples
- Relies on a costly logarithmic evaluation

Cost of stochastic operations

FPU (Floating Point Unit) rounding modes

- Stochastic operations frequently change the rounding mode of the FPU
- Pipeline flushed when rounding mode changed, hence hindering performance
- Prevents vectorisation as rounding mode is the same for all lanes

Overloaded operators

- Operators replaced by functions, compiled in the library
- FPU instructions replaced by function calls, causing performance overhead, especially in arithmetic intensive code

Cancellation detection

Logarithm approximation

- Cancellation detection: number of exact significant digits computed with log_{10}
- Using the base 2 exponent (multiplied by $log_{10}(2)$) as a fast approximation for logarithm
- Easily obtained from binary representation of floating point numbers

Difference with the previous evaluation

- Estimated number of exact significant digits can vary
- However, since $log_{10}(2) < 0.31$, at most a 1 digit difference
- Approximation gives a more pessimistic estimation for number of digits

Stochastic operations

Removing the change of rounding mode during computation

- As $a \oplus_{+\infty} b = -(-a \oplus_{-\infty} -b)$ (likewise for subtraction),
- And $a \otimes_{+\infty} b = -(a \otimes_{-\infty} -b)$ (likewise for division),
- Obtain rounded up value from rounded down operations (or conversely) by changing signs
- Implemented through random flip of the bit sign of the IEEE binary representation

Inlining the functions

- Minimise the cost of function calls

Vectorising CADNA

Prerequisites

- FPU rounding mode changes not necessary anymore
- Random generator changed to ease vectorisation through replication

Vectorising CADNA

Prerequisites

- FPU rounding mode changes not necessary anymore
- Random generator changed to ease vectorisation through replication

Vectorising methods

- Using intrinsics: tedious and difficult to use due to data types

Vectorising CADNA

Prerequisites

- FPU rounding mode changes not necessary anymore
- Random generator changed to ease vectorisation through replication

Vectorising methods

- Using intrinsics: tedious and difficult to use due to data types
- Automatic vectorisation: impossible due to added dependency from random bit generation

Vectorising CADNA

Prerequisites

- FPU rounding mode changes not necessary anymore
- Random generator changed to ease vectorisation through replication

Vectorising methods

- Using intrinsics: tedious and difficult to use due to data types
- Automatic vectorisation: impossible due to added dependency from random bit generation
- Compilation directives based: problematic due to lack of lane identifier for random generator

Vectorising CADNA

Prerequisites

- FPU rounding mode changes not necessary anymore
- Random generator changed to ease vectorisation through replication

Vectorising methods

- Using intrinsics: tedious and difficult to use due to data types
- Automatic vectorisation: impossible due to added dependency from random bit generation
- Compilation directives based: problematic due to lack of lane identifier for random generator
- SPMD (*Single Program Multiple Data*) on SIMD
  - Scalar programming with simple C-like syntax, with lane identifier
  - Compiler generates SIMD instructions
  - ispc (Intel SPMD Program Compiler) supports operator overloading, chosen over OpenCL

Execution masks

Divergence in control flow when vectorising

- Vectorised code containing conditional branches
- Instructions executed even when they should not
- Changes not commited to memory, through the use of an execution mask
- Usually implemented through software and costly in terms of performance

Reducing the use of execution masks

- Tests on whether an instability is detected or not
- Replacing these tests with preprocessor directives evaluated at compile time
- Disables the possibility of changing the detection mode during execution

Performance setup

Hardware

- Intel Xeon E3-1275
- 3.5GHz, 1 core used only

Benchmarks

- Pure arithmetic benchmarks
  - Addition (multiplication) over long vector
- More realistic benchmarks
  - Mandelbrot set computation
  - Finite difference stencil computation
- Application code compiled with gcc -O3

Versions of the CADNA library

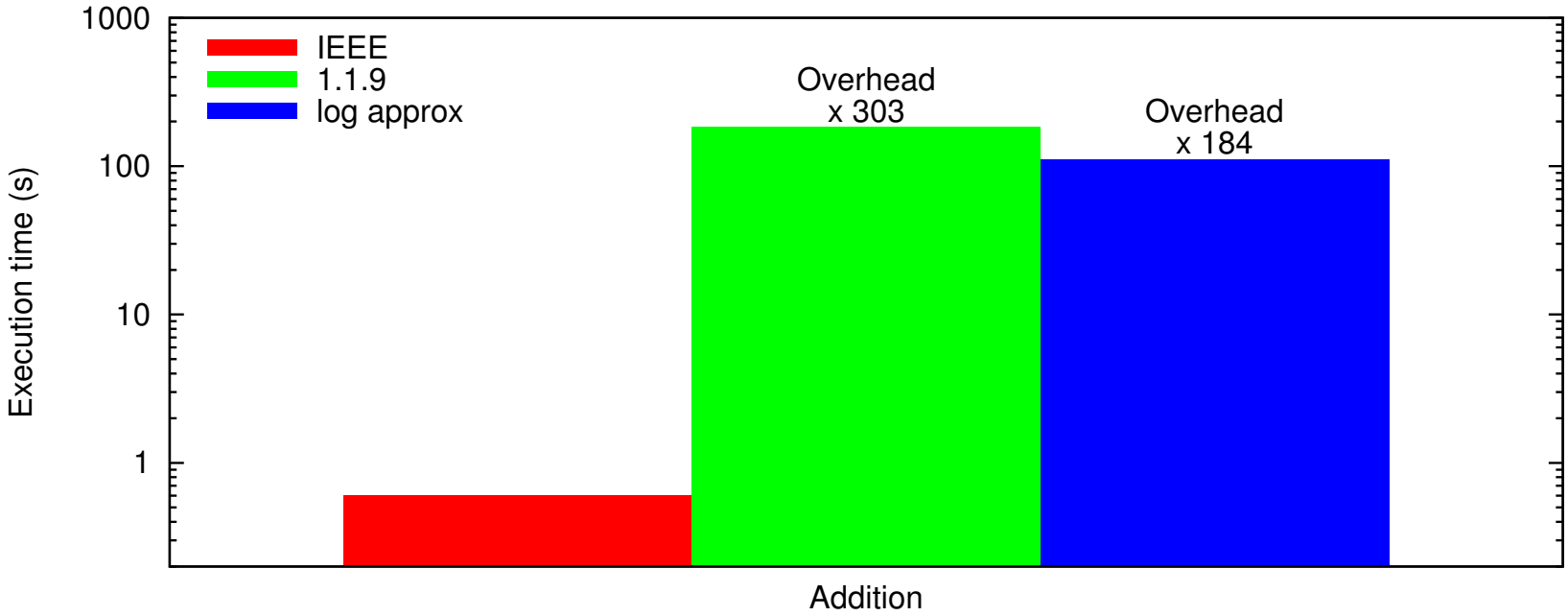
Compared versions of the benchmarks

- *ieee*, a IEEE version used as a baseline
- *1.1.9*, the previous version of CADNA
- *mask*, removing the FPU rouding mode change during operations and adding the change of sign through masks
- *inline*, using *mask* and inlining the operators
- *dyn*, using *inline* and changing the random generator to produce numbers dynamically

Compiling the libraries

- *1.1.9* compiled with gcc -O0 due to a known gcc bug
- *mask*, *inline* and *dyn* compiled with gcc -O3

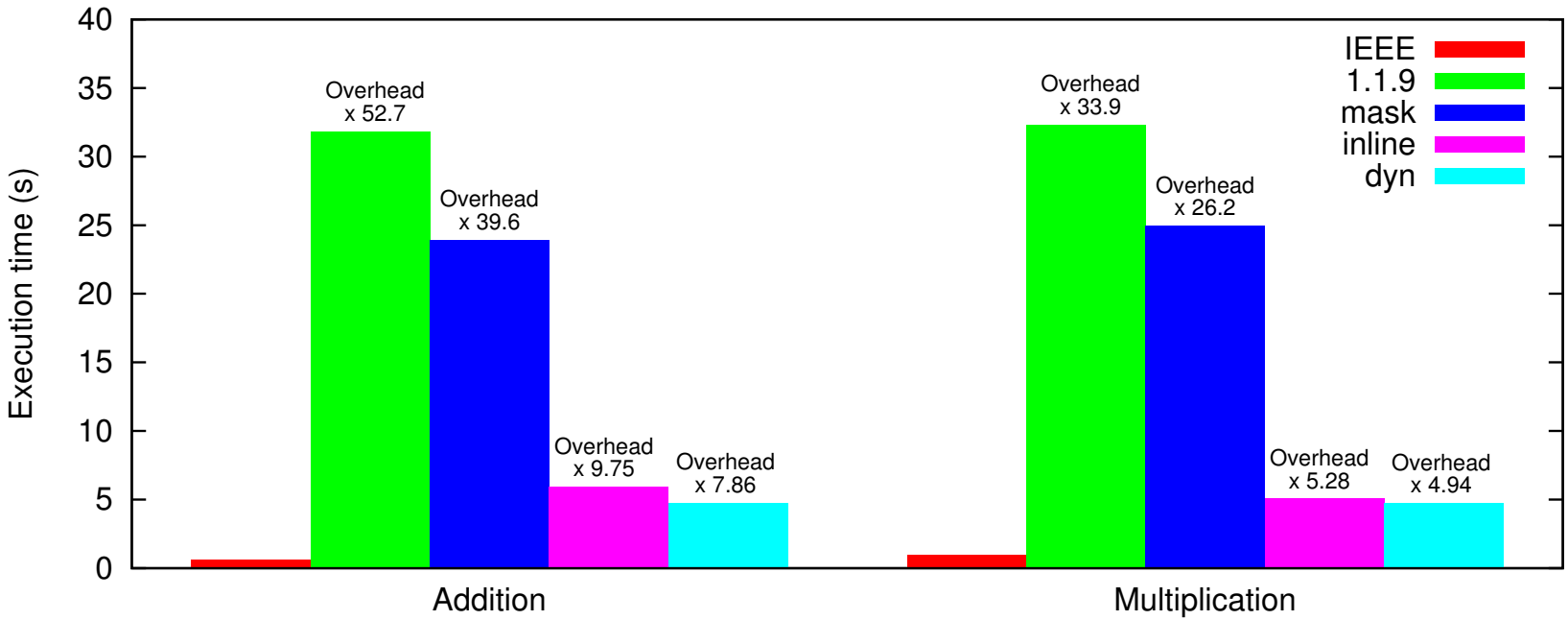
Cancellation detection



Analysis

- Addition only, cancellation only applies to addition
- All detections activated
- Overhead reduced by 40%

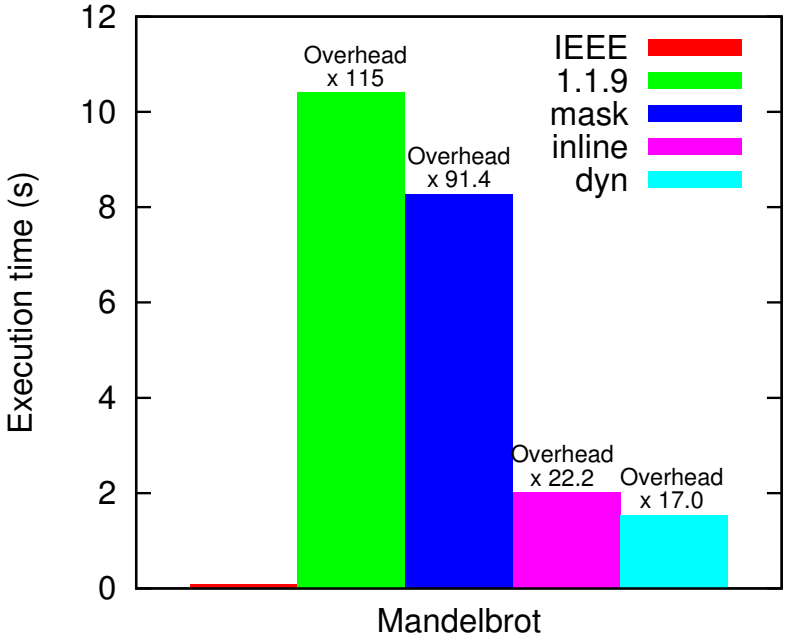
Arithmetic benchmarks

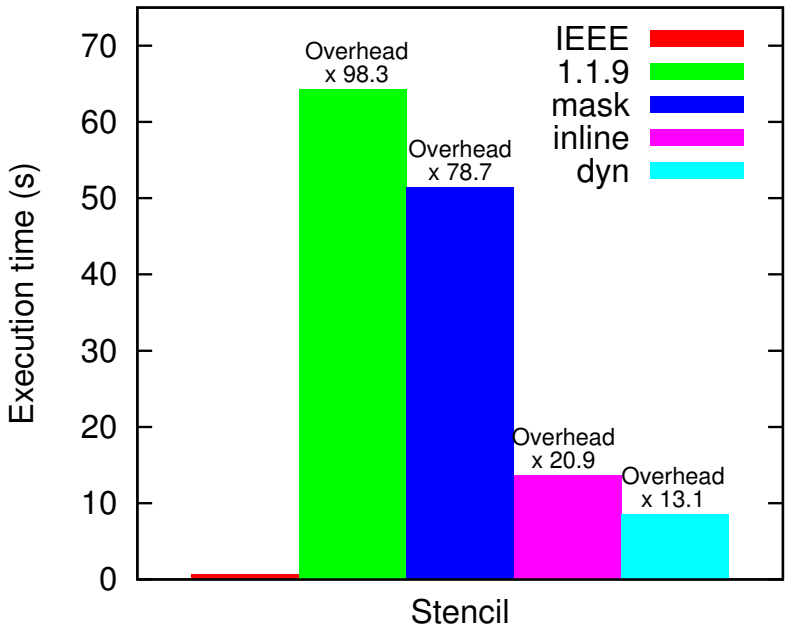


Analysis

- Overhead reduced by up to 84%

Realistic benchmarks





Analysis

- Initial overhead higher than for arithmetic benchmarks
- Due to nature of chosen applications
  - Mandelbrot more arithmetically intensive than arithmetic benchmarks
  - Stencil 3D memory access pattern lowers memory cache and prefetch efficiency
- Overhead reduced by up to 87%

Performance setup

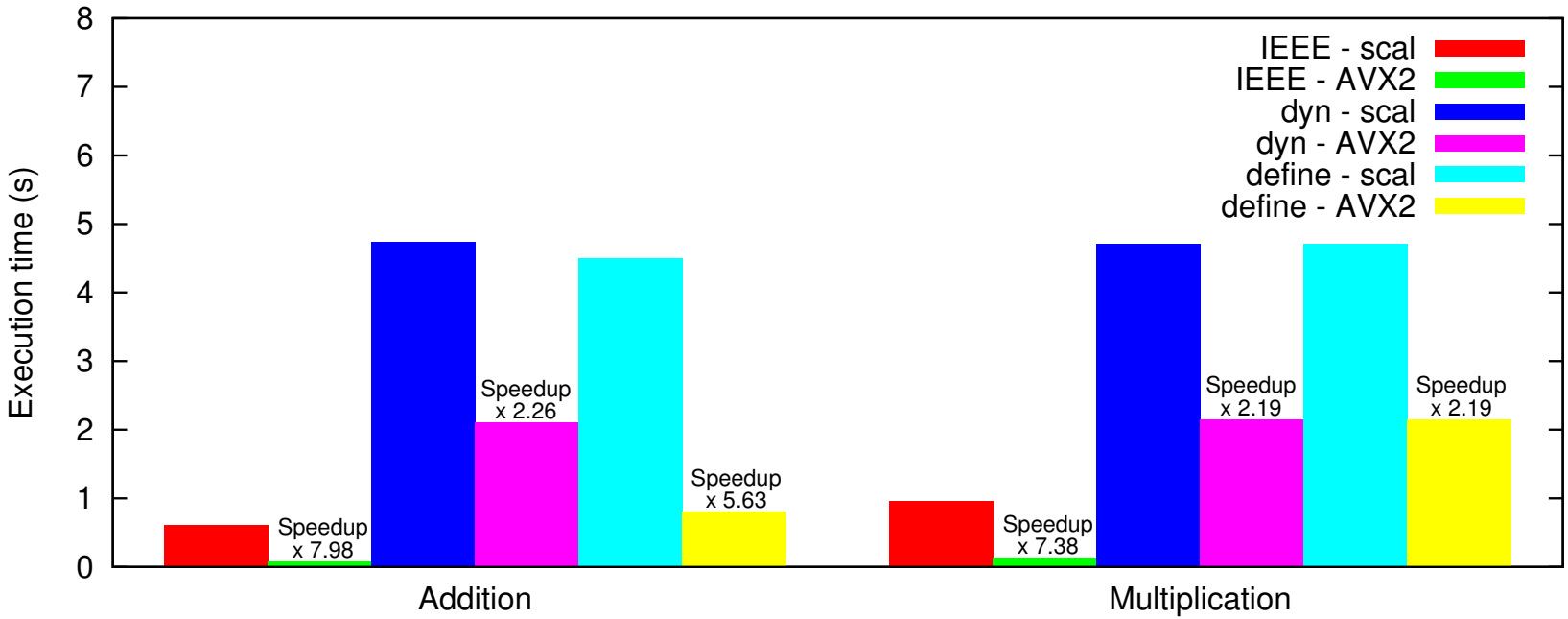
Hardware

- Same CPU as scalar, 1 core used only
- AVX2 instruction set

Compared versions of the benchmarks

- *ieee*, a IEEE version used as a baseline
- *dyn*, adapted from the scalar *dyn* version
- *define*, using *dyn* and replacing tests on anomaly detection flags by `#define` evaluated at compile time

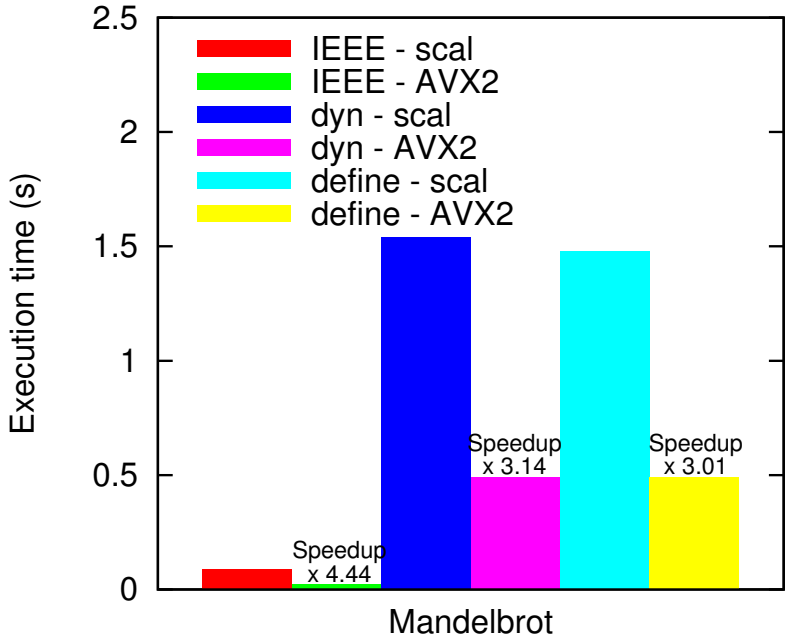
Arithmetic benchmarks on AVX2

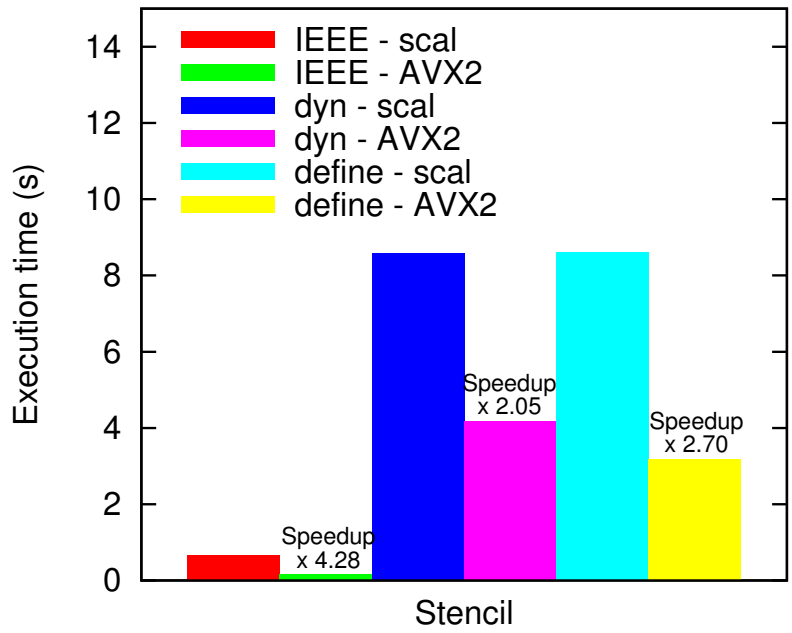


Analysis

- Vectorisation achieved (was impossible with original CADNA)
- CADNA speedup up to 5.63 times, lower than IEEE speedup
  - stochastic types are structures (AoS vs SoA)
- *define* version removing execution masks for addition

Realistic benchmarks on AVX2





Analysis

- Better performance of *dyn* on Mandelbrot confirms AoS vs SoA problem, as Mandelbrot has no memory accesses for stochastic types
- Improvement of speedup for stencil computation with *define* version due to removal of execution masks
- Speedup up to 3.14 times

Conclusion

Scalar improvements

- Improved performance on cancellation detection
  - Use of an approximation of the `log10` function
  - Overhead reduced by 40%
- Changes to CADNA enable large performance improvements on stochastic arithmetic operations
- Reduce in overhead between 81% and 87% on a variety of benchmarks

Vectorising

- Enabling SIMD computing with stochastic arithmetic
- Speedup of up to 5.63 on AVX2
- Lower speedup than IEEE due to AoS

Future prospects

New CADNA versions

- New CADNA version for shared memory (OpenMP)
- Extend current SPMD-on-SIMD version to target CPUs and GPUs
  - either with OpenCL,
  - or with ispc/CUDA.