ACCELERATING THE FLOWSIMULATOR

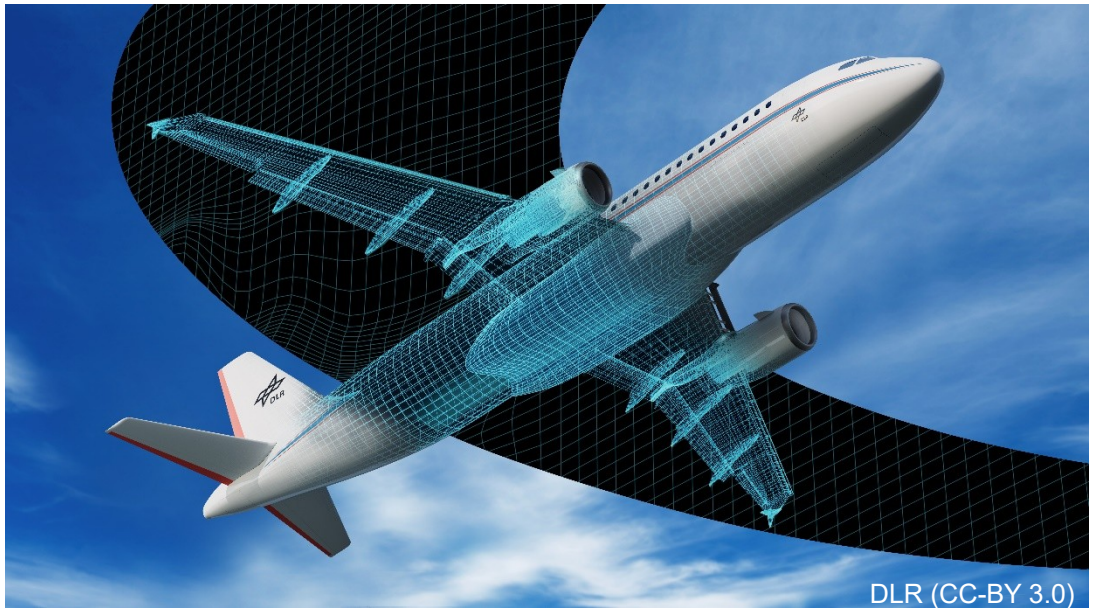
Low-level Analysis and Optimization of CODA Kernels

Johannes Wendler, DLR-SP, 07.11.2025

Motivation

- Computational Fluid Dynamics (CFD)
  - Accurate prediction of fluid flows
  - High demands on computational resources
  - Continuous performance analysis and optimization is key to efficient utilization of HPC resources

- CODA is shown to scale well, but what about node-level performance?



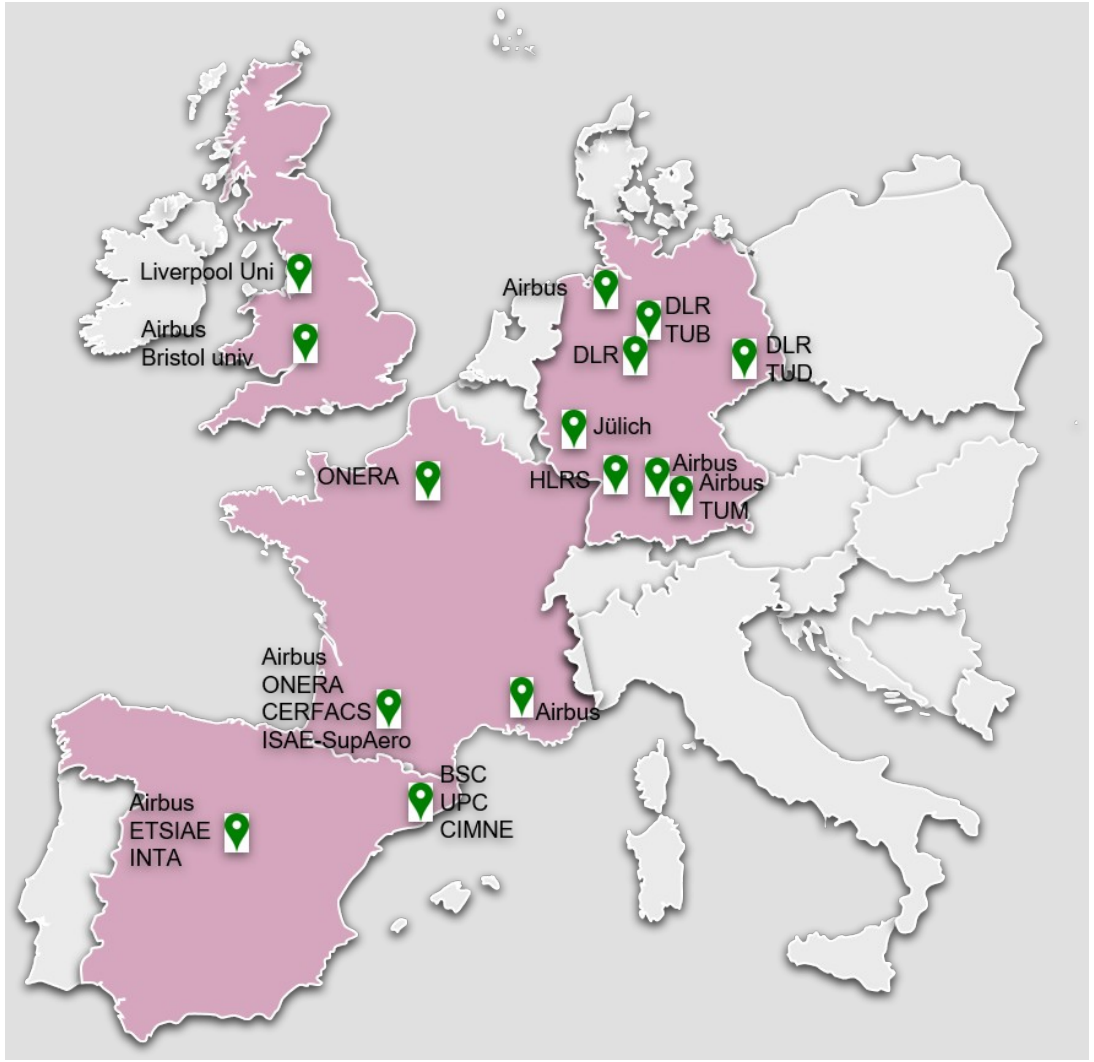
Motivation

CODA

- Flow solver written by
  - ONERA
  - DLR
  - Airbus
  - Many other European partners

Coding

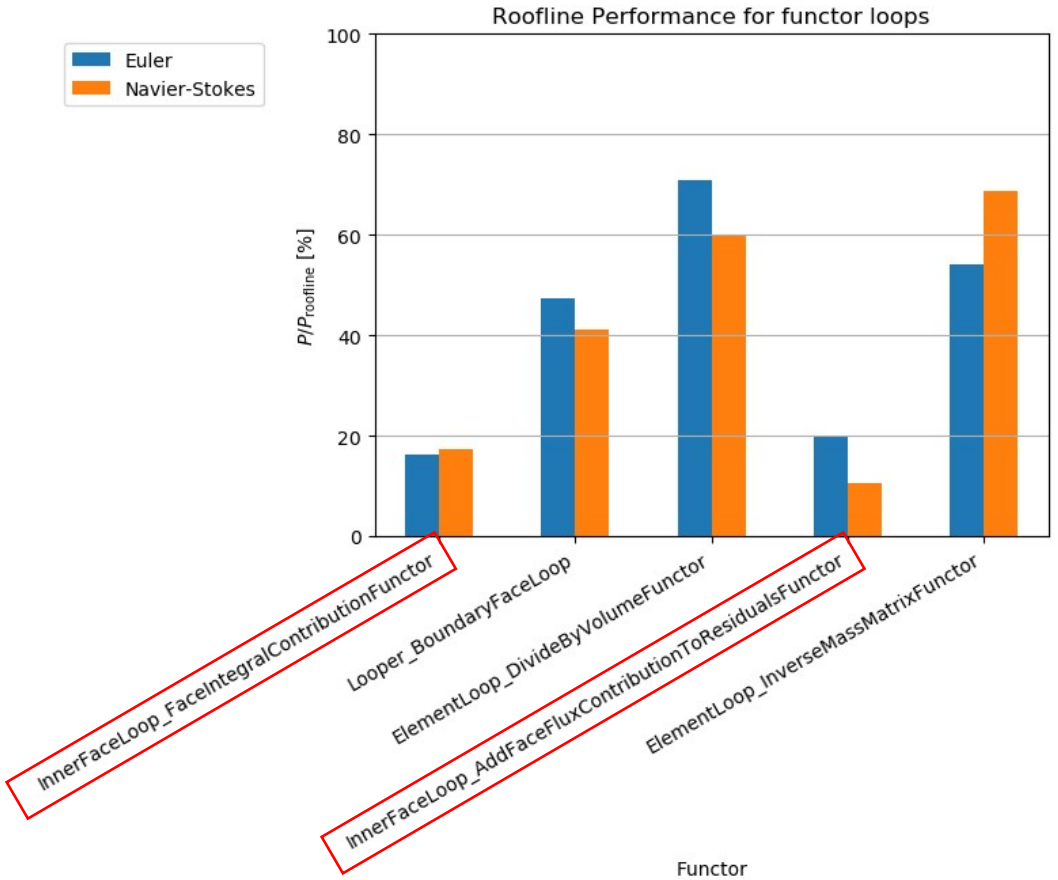
- User scenarios written in Python, solver core written in C++
- Hybrid parallelization, computation & communication overlap
- Spliss as linear solver



Recap of IPTW 2023

- Both inner face loops together account for ~90% of runtime
- Performance of them is < 20% of Roofline
- Bottleneck unclear

- Outlook:
  - A deeper look into the assembly code necessary



MICROBENCHMARKS

Microbenchmarks

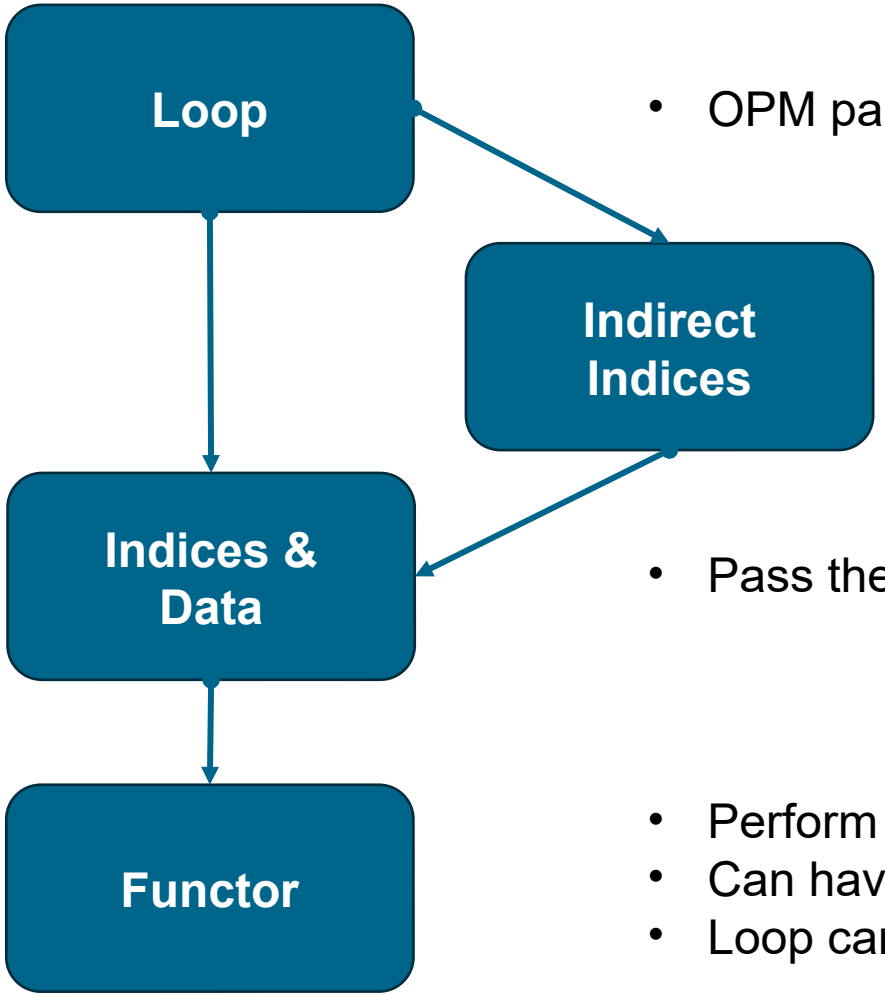
Why?

- Loops are large and contain many different sub parts from different Classes/Functions
- Many similar symbols with long names (10-15 lines +) in one compilation unit
- Preprocessing of actual physical meshes/data takes some time

How?

- Create a minimal (non physical) dataset
- mimick the CODA access pattern (just OMP parallel)
- Create functors with just a subset of the operations of the actual functors, increasing the complexity and completeness step by step

Microbenchmarks - Loops



- OPM parallel
- Optionally loop over index array
  - Has variable blocks of stride-one pattern. Then jumping to a different part of the data indices
  - Emulate indirect access pattern of unstructured grids
- Pass the Data and the index of the “Grid Cell“ to the functor
- Perform a computation on the “Grid Cell“
- Can have variable in- and outputs
- Loop can be called with different functors

Microbenchmarks - Functors

Augmentation

- Compute intermediate state variables from state variables, physical constants and parameters (i.e. compute pressure from the perfect gas law)

PDE flux

- Compute the (convection-) flux of the **P**artial **D**ifferential **E**quation (from the intermediate state)

Roe flux

- Compute the Roe approximate flux between two cells (from the fluxes of two neighboring cells)

TOOLS

Dynamic Analysis

LIKWID

- Manual instrumentation of code
- Measure hardware performance counters (dependent on CPU)
- Access to many different metrics of what happens on the CPU
- May be unreliable data (always sanity check)

Perf Tools

- Samples the program execution
- Can also measure hardware performance counters
- Annotates assembly code with time spent at each binary block / instruction

Static Analysis

OSACA

- Manual marker insertion into assembly code
- Analyzes the loop between those markers
- Gives information about execution port pressure and loop carried dependencies (assuming data is in cache)

MAQAO

- Automatically find loops inside binary, can be analyzed seperately by loop ID
- Gives information about several bottlenecks (assuming data is in cache)
  - i.e. vectorization ratio of instructions
  - Execution port or front end pressure
  - Function calls
  - Register usage
  - …

RESULTS

Results

Measure Environment

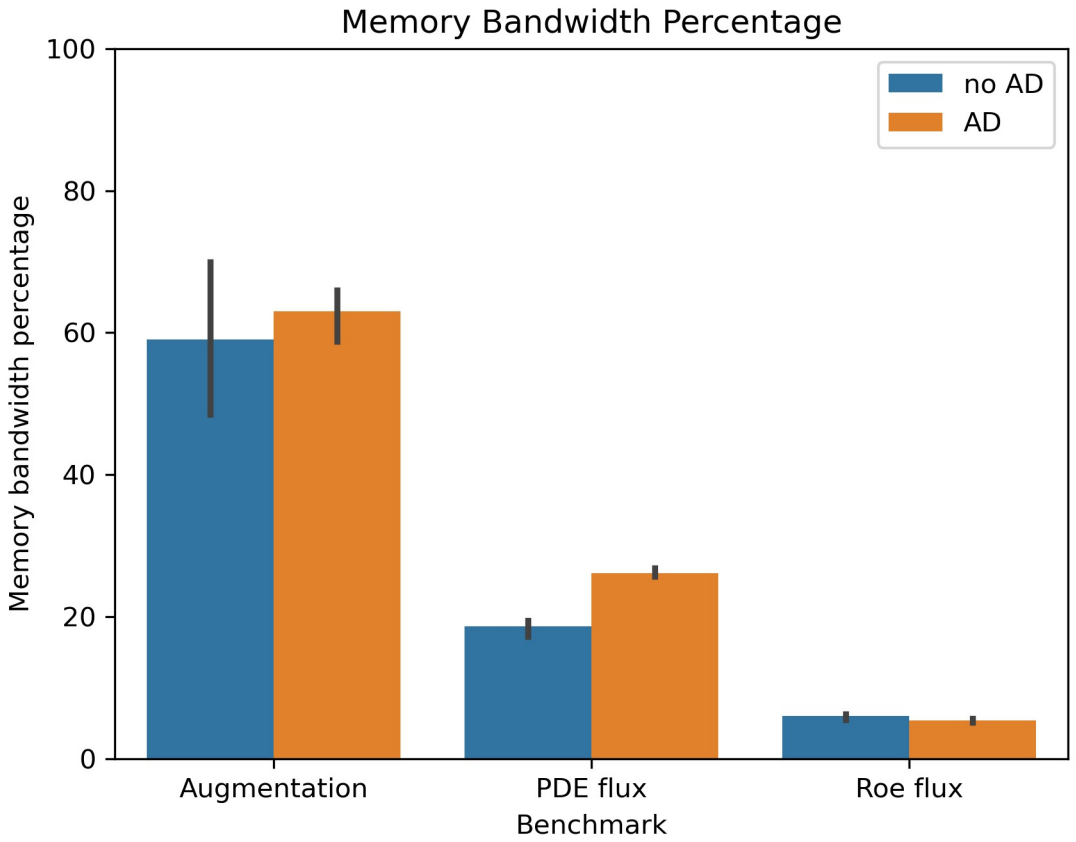
- IntelXeonW-2295 (18 phys. Cores)
  - Has more counters than our AMD Zen2 Cluster
  - Disabled boost @ 2.25 GHz
  - P_{max} = 2.25 GHz x 18 cores x 2 ports x 4 AVX x 2 fma = 648 Gflops/s
  - Measured peak memory bandwidth: 72.7 GB/s

Compilers

- GCC 11.4.0
- Clang 17.04
- Flags:
  - -fverbose-asm -Ofast -DNDEBUG -mtune=native -march=native -fno-finite-math-only

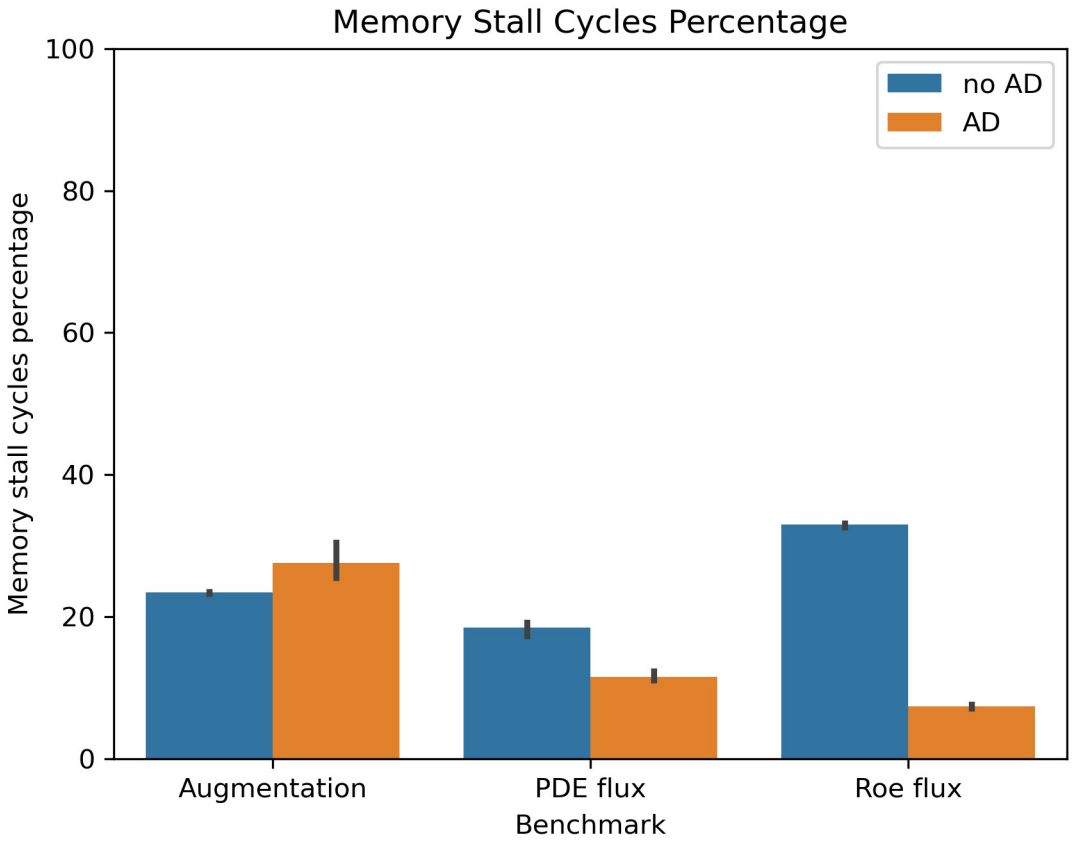
LIKWID

- Functors in order have rising complexity
- AD has not as much impact on the PDE flux (no complicated sqrt op?)
- Roofline performance sub par
  - AD less impacted: maybe it hides a bottleneck
- The two flux Functors in particular seem to also have a different bottleneck than memory



LIKWID

- Clang seems to be worse in some cases
- For AD Clang is consistently worse
- New hint: Memory Stall cycles
  - Was not possible on AMD Zen2
  - We are waiting on memory without the bandwidth being exhausted
  - Could be a latency problem

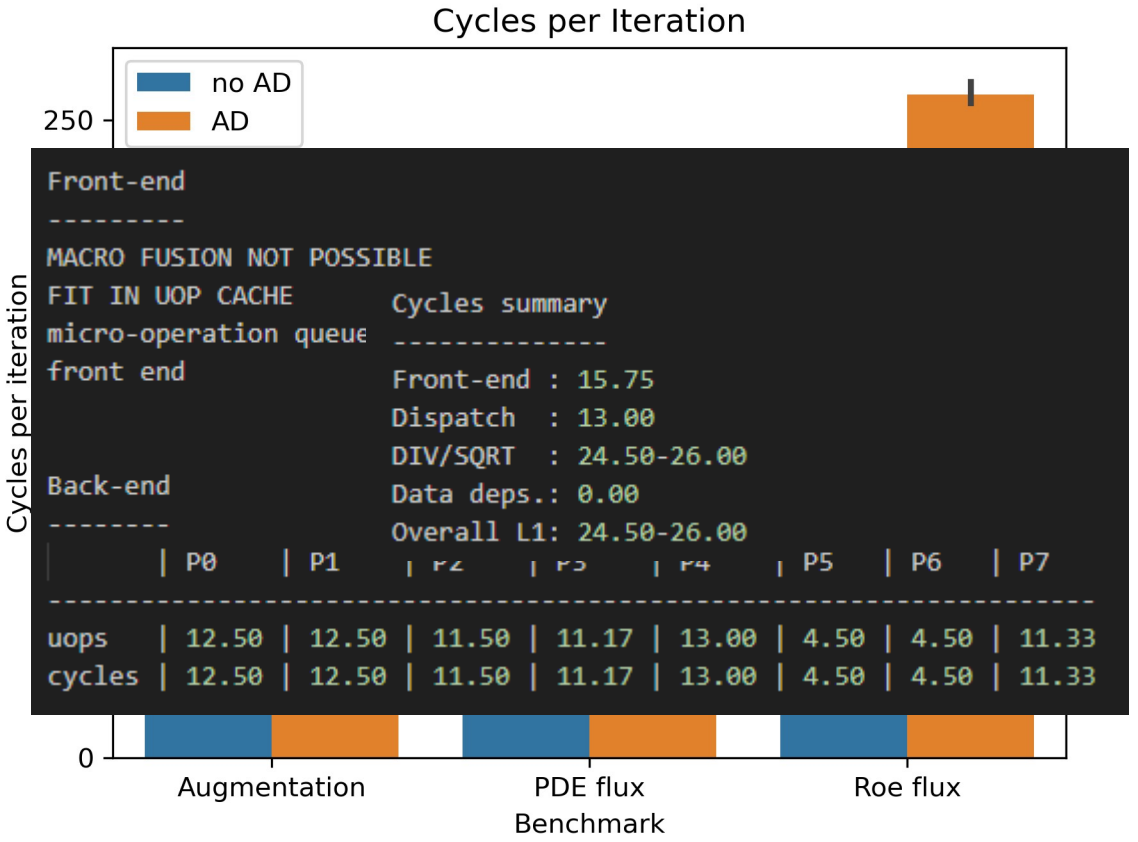


OSACA + MAQAO

Port Pressure / Loop Cycles

- Busy cycles of front- and back-end
- Final cycle estimation per iteration
- Comparison of static L1 prediction and LIKWID measurements

Functor	MAQAO / OSACA No AD / AD	LIKWID No AD / AD
Augmentation	24.5 - 26 / 64.5-66	25.6 - 27.9 / 50.7 - 51.9
PDE Flux	43.25 / 67.5	28.9 - 30.8 / 39.1 - 40.5
Roe Flux	26.75 / 46.25	~181.0 / 261.0



- Static analysis maybe overestimates div ops
- Roe flux estimation does not match measurements => completely different bottle neck

MAQAO + Perf Tools

MAQAO code quality analysis

- Op queue for DIV / SQRT
  - For Augementation / PDE flux
  - Hint: remove those ops if possible
- No vectorization
  - For all functors
  - Hint: stride-1 access => reorganize arrays of structures to structures of arrays
- Function calls
  - For PDE flux / Roe flux
  - Hint: inlining
- Used register count => for Roe flux the amount of registers is exhausted

Perf Tools

- Hottest instructions are div instructions
- Very few MOV ops were slightly hotter than usual => maybe memory latency issues

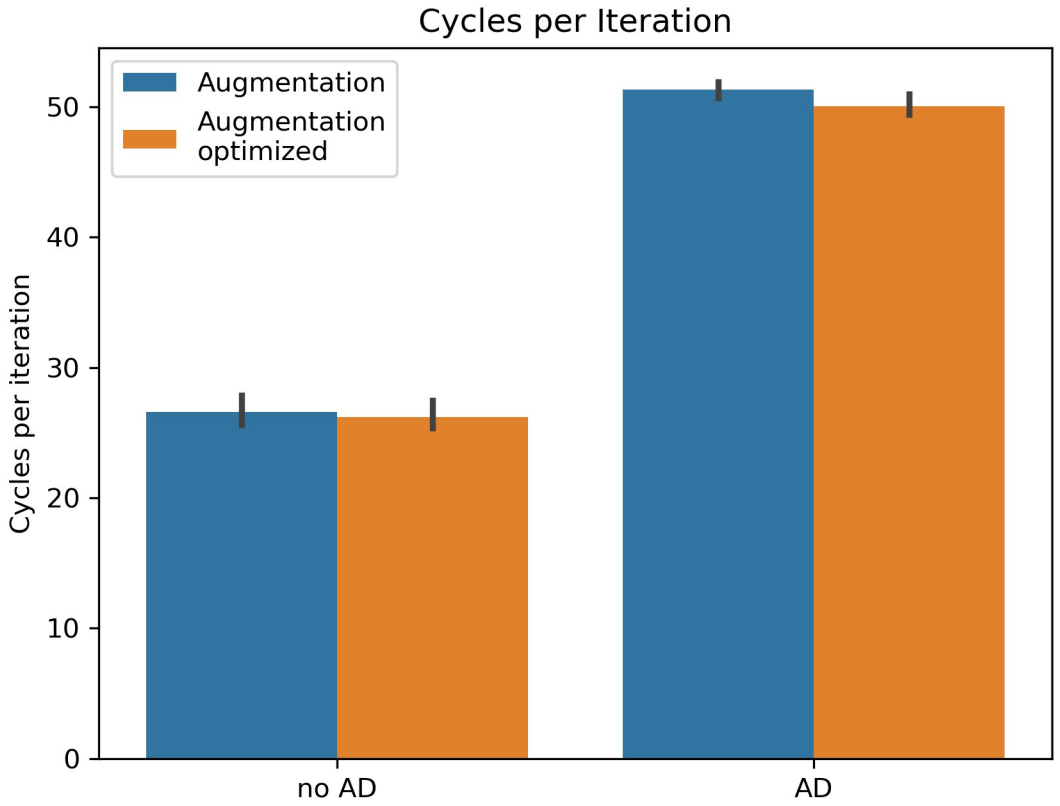
Optimization Strategies

Low hanging fruits

- Remove redundant DIV ops
  - Augmentation has 1 easy to remove DIV
- Manual prefetching to alleviate memory latency problems
  - Can be worked into all functors with a few lines of code
- Inlining
  - Experiment with compiler flags
  - Add compiler hints into code

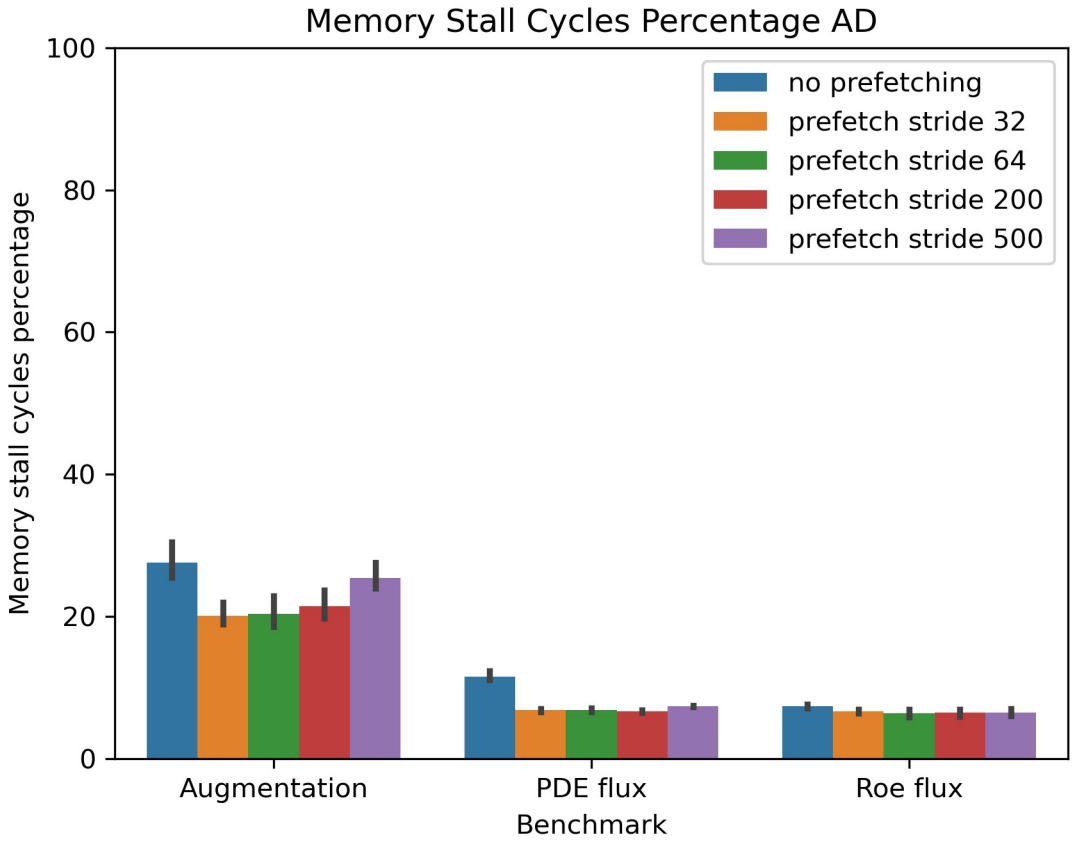
Results div opt

- Speedup
  - No AD: 1.144
  - AD: 1.053
  - Inside uncertainty intervals
- MAQAO loop cycles prediction:
  - No AD: 4 cycles less
  - AD: 8 cycles less
  - Does not match LIKWID measurements
  - Inside uncertainty intervals
- Speedup inconclusive with positive bias



Result prefetching

- Prefetching with many different strides (more than shown here)
- Small gains, but within uncertainty intervals
- Looks worse for AD
- Memory stall cycles
  - show that we should be waiting on memory less
  - More so for AD
- Hints that this is not the dominant bottleneck or stall counters are unreliable



Result Inlining

- Experiments with compiler flags
- Stay with GCC here, as clang has different flags
  - Inline flags:

    --param inline-unit-growth=200 --param inline-min-speedup=1

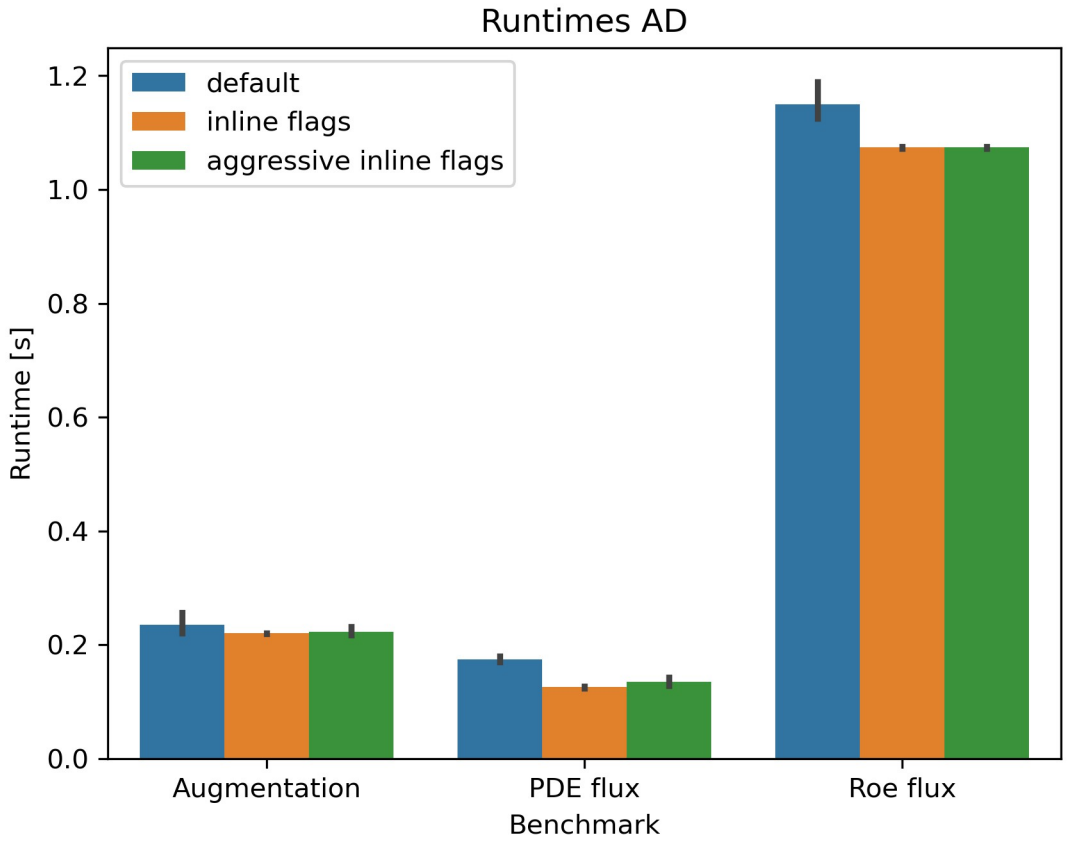
  - Aggressive inline flags:

    --param inline-unit-growth=1000 --param inline-min-speedup=1

Result Inlining

- No AD
  - Only the Roe flux benefits here
  - Many functions still not inlined
- AD
  - PDE flux and Roe flux just slightly better
  - Many functions still not inlined
- MAQAO: function calls inside loop

Functor	default no AD / AD	inline no AD / AD	aggressive no AD / AD
Augmentation	0 / 1	0 / 0	0 / 0
PDE flux	10 / 9	10 / 8	10 / 8
Roe flux	5 / 5	20 / 19	20 / 19



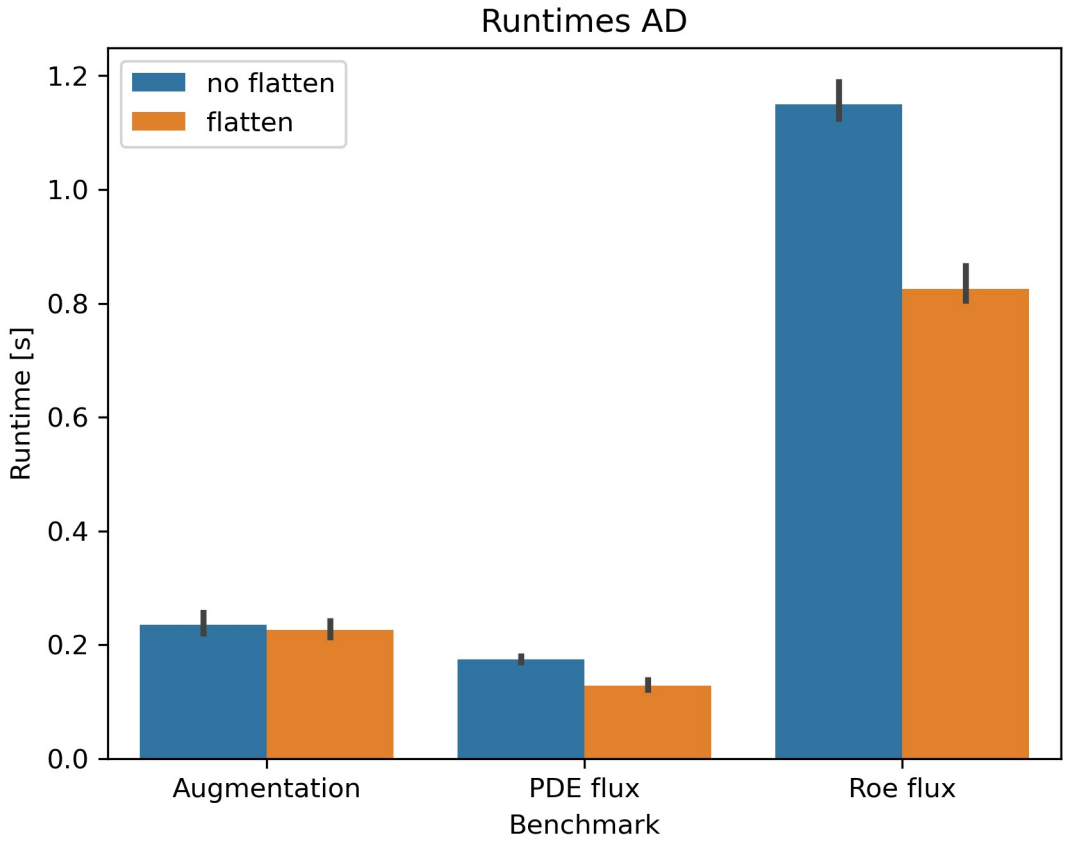
Result Inlining

- Other strategy for inlining:
  - __attribute__((flatten))
  - Inlines all calls inside a function
- Speedups

Functor	no AD	AD
PDE flux	1.55	1.36
Roe flux	1.62	1.39

- MAQAO: function calls inside loop

Functor	default no AD / AD	flatten no AD / AD
Augmentation	0 / 1	0 / 0
PDE flux	10 / 9	8 / 8
Roe flux	5 / 5	18 / 18



- All calls in flattened version are to “unknown“ functions
  => misinterpretation by MAQAO?

CONCLUSION

Conclusion

LIKWID

- Pro:
  - Very usefull for all kinds of performance metrics
  - Even without HPM counters: benchmark suite, pinning capabilities
- Con:
  - Limited by capabilities of used hardware
  - Clunky to use with everything that has to be paid attention to, to not mess up
  - Manual instrumentation

Perf Tools

- Pro:
  - Easy to get going with sampling approach
  - Insights into “hotness“ of assembly instructions
- Con:
  - Does not tell you why an instruction may be hot

Conclusion

OSACA

- Pro:
  - Small tool, easy to get going with
- Con:
  - Limited capabilities (just port pressure and loop carried dependencies)
  - Manual instrumentation inside assembly code
  - Hard to interpret results

MAQAO

- Pro:
  - Very powerful tool, with almost too many features
  - Static analysis gives even hints to possible optimizations
  - No other tool found so far that can give you info about register usage
- Con:
  - Not well documented (lots of try and error)
  - Much manual input and reading needed to filter and analyse the important parts

Conclusion

Achievements

- Know-How with tools
- Bottlenecks
  - Memory latency seems not to be a problem so far
  - Removing DIV ops maybe helpful where it is easy to do
  - Inlining is the game changer

Next Steps

- Port the inlining strategies to the actual production code
- See if then register spilling is a problem
- Extend the gained knowledge to HyperCODA

THANK YOU

Imprint

Topic: **Accelerating the FlowSimulator**
Low-levelAnalysis and Optimization of CODA Kernels

Date: 2025-11-07

Author: Johannes Wendler

Institute: DLR - SP