Cache optimization for tape evaluations

M. Sagebaum, N.R. Gauger, J. Stegmann

Chair for Scientific Computing
University of Kaiserslautern-Landau (RPTU)

28th EuroAD workshop @ CERN

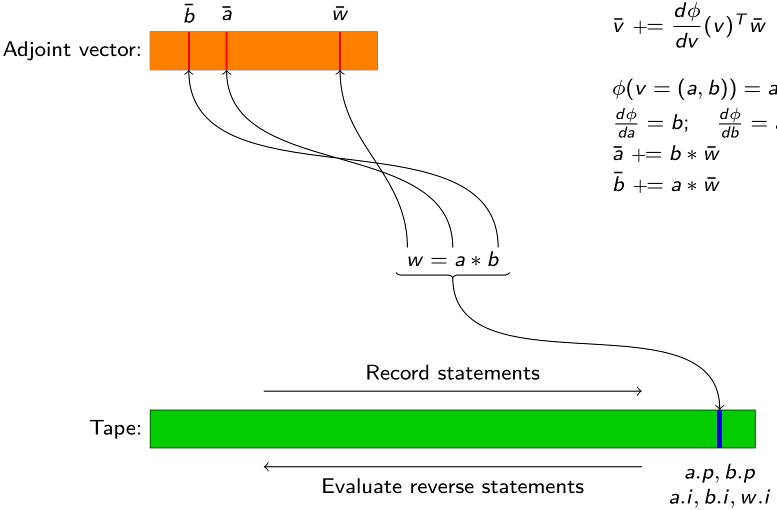
Overview

- Motivation
- Cache optimization heuristics
- Performance results

Preliminary: What is a CoDiPack ID/identifier/index

- Each variable gets a “unique” id
- This id is used to access the adjoint of the variable
- E.g. v is the variable, then $\bar{v}$ is the adjoint.
- In code: `v` is the variable, then `adjoint[v.id]` is the adjoint.

Preliminary: What is a CoDiPack ID/identifier/index



$\bar{b}$ $\bar{a}$ $\bar{w}$

Adjoint vector:

$w = a * b$

Record statements

Tape:

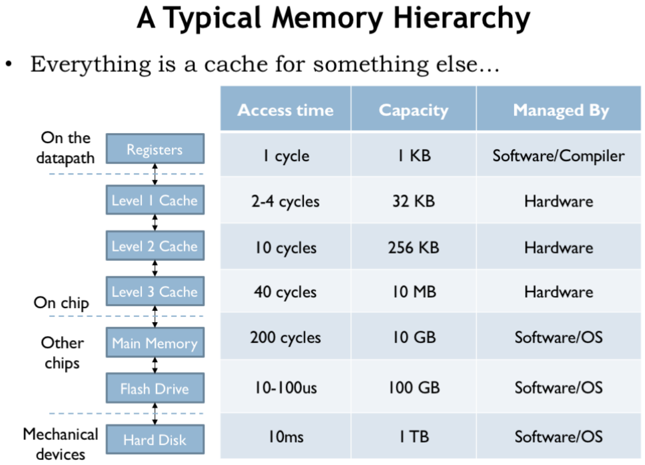
Evaluate reverse statements

$a.p, b.p$
$a.i, b.i, w.i$

$$\bar{v} \mathrel{+}= \frac{d\phi}{dv}(v)^T \bar{w}$$

$$\phi(v = (a, b)) = a * b$$
$$\frac{d\phi}{da} = b; \quad \frac{d\phi}{db} = a$$
$$\bar{a} \mathrel{+}= b * \bar{w}$$
$$\bar{b} \mathrel{+}= a * \bar{w}$$

Preliminary: Why cache optimization?



A Typical Memory Hierarchy

- Everything is a cache for something else…

	Access time	Capacity	Managed By
Registers	1 cycle	1 KB	Software/Compiler
Level 1 Cache	2-4 cycles	32 KB	Hardware
Level 2 Cache	10 cycles	256 KB	Hardware
Level 3 Cache	40 cycles	10 MB	Hardware
Main Memory	200 cycles	10 GB	Software/OS
Flash Drive	10-100us	100 GB	Software/OS
Hard Disk	10ms	1 TB	Software/OS

Picture from: https://computationstructures.org/lectures/caches/caches.html

Motivation: Cache optimization of a tape

How bad can it be?

Motivation: Cache optimization of a tape

How bad can it be?

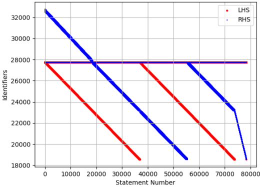
- Synthetic test with optimal distribution
- Randomize the id distribution of CoDiPack.

Motivation: Cache optimization of a tape

How bad can it be?

- Synthetic test with optimal distribution
- Randomize the id distribution of CoDiPack.

- Optimal distribution
- Runtime 2.44 s



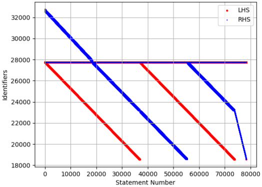
Motivation: Cache optimization of a tape

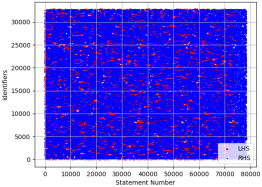
How bad can it be?

- Synthetic test with optimal distribution
- Randomize the id distribution of CoDiPack.

- Optimal distribution
- Runtime 2.44 s

- Randomized distribution
- Runtime 19.83 s





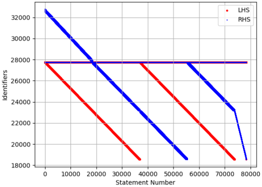
Motivation: Cache optimization of a tape

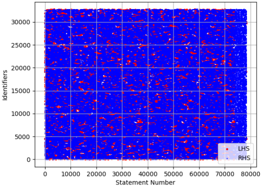
How bad can it be?

- Synthetic test with optimal distribution
- Randomize the id distribution of CoDiPack.

- Optimal distribution
- Runtime 2.44 s

- Randomized distribution
- Runtime 19.83 s





Difference of factor 8.

Motivation: Cache optimization of a tape

How good can it be?

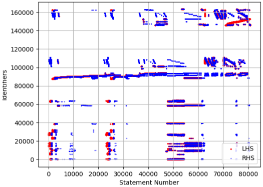
- Prototype implementation on a small test case in the TRACE code
    - TRACE: Turbomachinery simulation code developed by the DLR (German Aerospace Center)
    - Used by MTU Aero Engines (10.000 employees, €4.628 billion revenue)

Motivation: Cache optimization of a tape

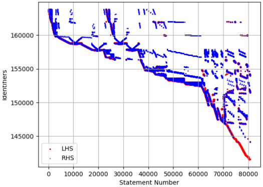
How good can it be?

- Prototype implementation on a small test case in the TRACE code
  - TRACE: Turbomachinery simulation code developed by the DLR (German Aerospace Center)
  - Used by MTU Aero Engines (10.000 employees, €4.628 billion revenue)

- Regular distribution
- Runtime 0.197 s



- Simple heuristic for redistribution
- Runtime 0.147 s



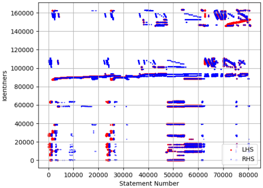
Motivation: Cache optimization of a tape

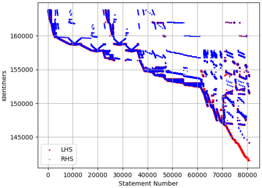
How good can it be?

- Prototype implementation on a small test case in the TRACE code
  - TRACE: Turbomachinery simulation code developed by the DLR (German Aerospace Center)
  - Used by MTU Aero Engines (10.000 employees, €4.628 billion revenue)

- Regular distribution
- Runtime 0.197 s

- Simple heuristic for redistribution
- Runtime 0.147 s





Improvement of 25 %

Tape evaluation cache optimization

Step 1: Analyze lifetime of ids

Step 2: Assign new ids based on life-time

- short life-time ⇒ assign hot id
- long life-time ⇒ assign cold id

Step 3: Remove gap between hot and cold ids

Step 1 - Analyze lifetime of ids

```
    a = x * y  // lifetime of a starts, e.g. a.id = 10
```

Step 1 - Analyze lifetime of ids

```
a = x * y  // lifetime of a starts, e.g. a.id = 10
b = a + x  // a is used
```

Step 1 - Analyze lifetime of ids

```
a = x * y  // lifetime of a starts, e.g. a.id = 10
b = a + x  // a is used
c = x - y
d = a * c  // a is used for the last time
```

Step 1 - Analyze lifetime of ids

```
a = x * y  // lifetime of a starts, e.g. a.id = 10
b = a + x  // a is used
c = x - y
d = a * c  // a is used for the last time
e = d / b
a = e / y  // a is overwritten, lifetime of a in line 1 ends
```

Step 1 - Analyze lifetime of ids

```
a = x * y  // lifetime of a starts, e.g. a.id = 10
b = a + x  // a is used
c = x - y
d = a * c  // a is used for the last time
e = d / b
a = e / y  // a is overwritten, lifetime of a in line 1 ends
```

- Original lifetime of a is 6 statements.

Step 1 - Analyze lifetime of ids

```
a = x * y  // lifetime of a starts, e.g. a.id = 10
b = a + x  // a is used
c = x - y
d = a * c  // a is used for the last time
e = d / b
a = e / y  // a is overwritten, lifetime of a in line 1 ends
```

- Original lifetime of a is 6 statements.
- Minimum lifetime of a is 4 statements.

Step 1 - Analyze lifetime of ids

```
a = x * y  // lifetime of a starts, e.g. a.id = 10
b = a + x  // a is used
c = x - y
d = a * c  // a is used for the last time
e = d / b
a = e / y  // a is overwritten, lifetime of a in line 1 ends
```

- Original lifetime of a is 6 statements.
- Minimum lifetime of a is 4 statements.
- If `d` and `e` are only used once then the id of a can be used for these.

Step 1 - Analyze lifetime of ids - Algorithm

1. Set lifetime start of all input variables to 0
2. For each statement:
    1. Set last use of all right hand side variables
    2. Update lifetime of the left hand side variable (It is overwritten)
    3. Set lifetime start of the left hand side variable
3. Set last use of all output variables
4. Update lifetime of all remaining variables

Result: Lifetime of variable for each statement

Step 2 - Assign new ids based on life-time - Hot and cold

Hot id area:

- Ids that are used for a short time.
- Can be reused quite often.
- Should always be available in the cache.

Cold id area:

- Ids that are used for a long time.
- Define the same variable for a long time.
- Need to be loaded from RAM most of the time.

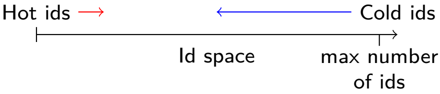
Step 2 - Assign new ids based on life-time - Hot and cold

Hot id area:

- Ids that are used for a short time.
- Can be reused quite often.
- Should always be available in the cache.

Cold id area:

- Ids that are used for a long time.
- Define the same variable for a long time.
- Need to be loaded from RAM most of the time.



- *max number of ids*: known from the recorded tape

Step 2 - Assign new ids based on life-time

```
lifetime    4: a (id: 102) = x (id: 100) * y (id: 101)
lifetime    4: b (id: 103) = a (id: 102) + x (id: 100)
lifetime    1: c (id: 104) = x (id: 100) - y (id: 101)
lifetime    1: d (id: 105) = a (id: 102) * c (id: 104)
lifetime    1: e (id: 106) = d (id: 105) / b (id: 103)
lifetime 1000: a (id: 102) = e (id: 106) / y (id: 101)
```

- Stmt 1: Nothing changes

Step 2 - Assign new ids based on life-time

```
lifetime    4: a (id: 102) = x (id: 100) * y (id: 101)
lifetime    4: b (id: 103) = a (id: 102) + x (id: 100)
lifetime    1: c (id: 104) = x (id: 100) - y (id: 101)
lifetime    1: d (id: 105) = a (id: 102) * c (id: 104)
lifetime    1: e (id: 106) = d (id: 105) / b (id: 103)
lifetime 1000: a (id: 102) = e (id: 106) / y (id: 101)
```

- Stmt 1: Nothing changes
- Stmt 2: Nothing changes

Step 2 - Assign new ids based on life-time

```
lifetime    4: a (id: 102) = x (id: 100) * y (id: 101)
lifetime    4: b (id: 103) = a (id: 102) + x (id: 100)
lifetime    1: c (id: 104) = x (id: 100) - y (id: 101)
lifetime    1: d (id: 105) = a (id: 102) * c (id: 104)
lifetime    1: e (id: 106) = d (id: 105) / b (id: 103)
lifetime 1000: a (id: 102) = e (id: 106) / y (id: 101)
```

- Stmt 1: Nothing changes
- Stmt 2: Nothing changes
- Stmt 3: Nothing changes

Step 2 - Assign new ids based on life-time

```
lifetime    4: a (id: 102) = x (id: 100) * y (id: 101)
lifetime    4: b (id: 103) = a (id: 102) + x (id: 100)
lifetime    1: c (id: 104) = x (id: 100) - y (id: 101)
lifetime    1: d (id: 104) = a (id: 102) * c (id: 104)
lifetime    1: e (id: 106) = d (id: 105) / b (id: 103)
lifetime 1000: a (id: 102) = e (id: 106) / y (id: 101)
```

- Stmt 1: Nothing changes
- Stmt 2: Nothing changes
- Stmt 3: Nothing changes
- Stmt 4: Lifetime of `c` ends. Id 104 is released and used for `d`.

Step 2 - Assign new ids based on life-time

```
lifetime    4: a (id: 102) = x (id: 100) * y (id: 101)
lifetime    4: b (id: 103) = a (id: 102) + x (id: 100)
lifetime    1: c (id: 104) = x (id: 100) - y (id: 101)
lifetime    1: d (id: 104) = a (id: 102) * c (id: 104)
lifetime    1: e (id: 104) = d (id: 104) / b (id: 103)
lifetime 1000: a (id: 102) = e (id: 106) / y (id: 101)
```

- Stmt 1: Nothing changes
- Stmt 2: Nothing changes
- Stmt 3: Nothing changes
- Stmt 4: Lifetime of `c` ends. Id 104 is released and used for `d`.
- Stmt 5: Id for `d` is updated. Lifetime of `d` and `a` ends. Id 104 and 102 are released. Id 104 is used for `e`.

Step 2 - Assign new ids based on life-time

```
lifetime    4: a (id: 102) = x (id: 100) * y (id: 101)
lifetime    4: b (id: 103) = a (id: 102) + x (id: 100)
lifetime    1: c (id: 104) = x (id: 100) - y (id: 101)
lifetime    1: d (id: 104) = a (id: 102) * c (id: 104)
lifetime    1: e (id: 104) = d (id: 104) / b (id: 103)
lifetime 1000: a (id: 999) = e (id: 104) / y (id: 101)
```

- Stmt 1: Nothing changes
- Stmt 2: Nothing changes
- Stmt 3: Nothing changes
- Stmt 4: Lifetime of `c` ends. Id 104 is released and used for `d`.
- Stmt 5: Id for `d` is updated. Lifetime of `d` and `a` ends. Id 104 and 102 are released. Id 104 is used for `e`.
- Stmt 6: Id for `e` is updated. Lifetime of `e` and `b` ends. Id 103 and 104 are released. The cold id 999 is used for `a`.

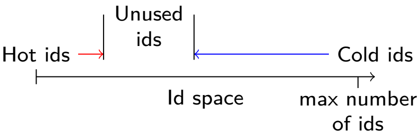
Step 2 - Assign new ids based on life-time - Algorithm

1. Assign new ids to all inputs
2. Add inputs ids to removal stack
3. For each statement:
    1. Update ids of right hand side variables
    2. Free all ids which lifetime ends
    3. Assign new id to left hand side variable
    4. Add id to removal stack

Result: Updated ids in statements

Step 3 - Remove gap between hot and cold ids

Before shift:



Unused ids

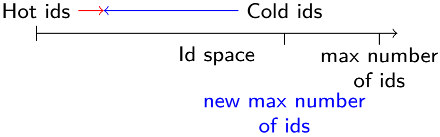
Hot ids

Cold ids

Id space

max number of ids

After shift:



Hot ids

Cold ids

Id space

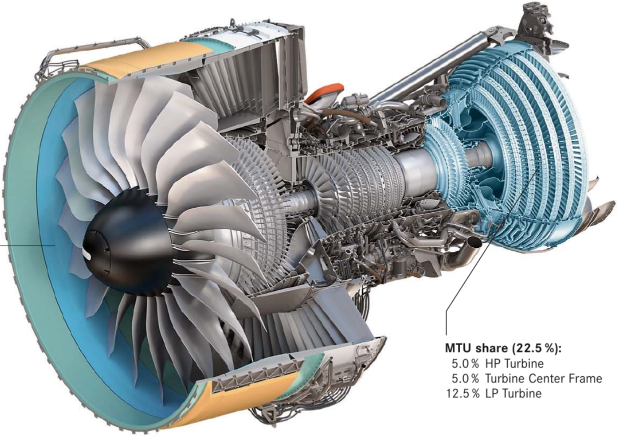
max number of ids

new max number of ids

Result: Smaller adjoint vector

Performance results - Turbine example

Example: A380 Turbine: GP7000



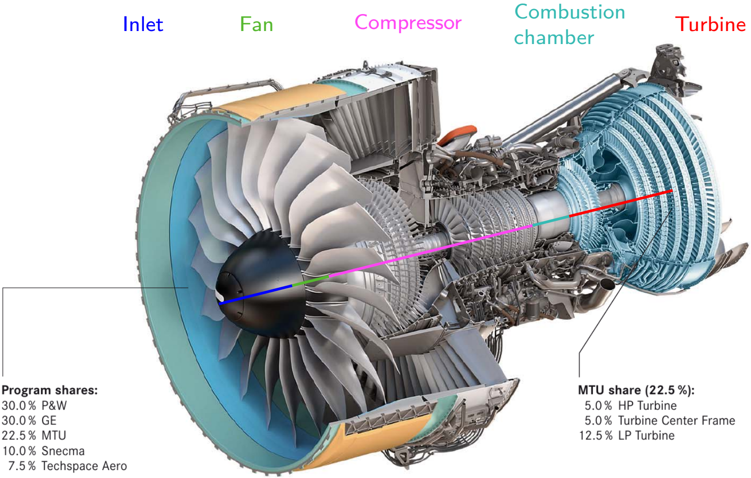
Program shares:
30.0 % P&W
30.0 % GE
22.5 % MTU
10.0 % Snecma
7.5 % Techspace Aero

MTU share (22.5 %):
5.0 % HP Turbine
5.0 % Turbine Center Frame
12.5 % LP Turbine

Pictures www.mtu.de

Performance results - Turbine example

Example: A380 Turbine: GP7000



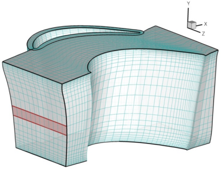
Program shares:
30.0 % P&W
30.0 % GE
22.5 % MTU
10.0 % Snecma
7.5 % Techspace Aero

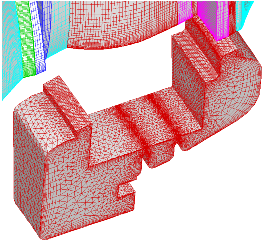
MTU share (22.5 %):
5.0 % HP Turbine
5.0 % Turbine Center Frame
12.5 % LP Turbine

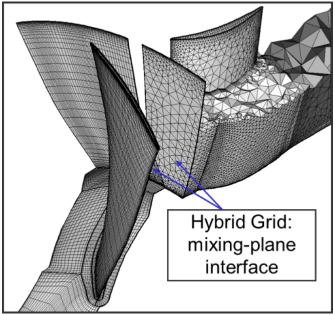
Pictures www.mtu.de

Performance Results - TRACE CFD Suite

- Finite volume solver
- (U)RANS equations and many more
- Structured and unstructured grids
- Multidisciplinary:
  - Aerodynamics, aeroelastics, aeroacoustics, combustion
- Multi stage computations
- Complex geometries
  - Fillets, inlets, cavities, bleeds, etc.



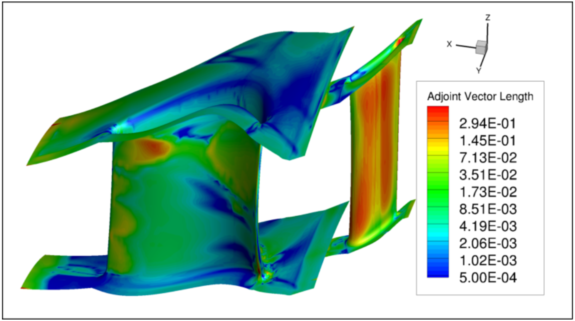




Pictures DLR-AT

Performance results

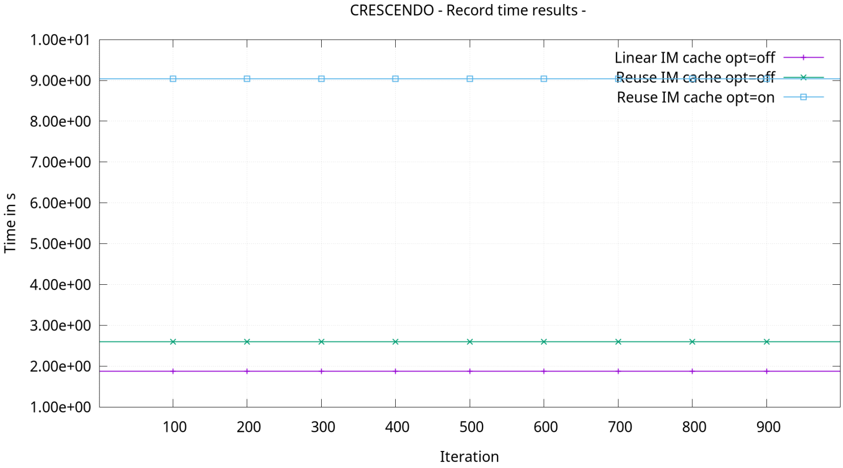
CRESENDO test case:



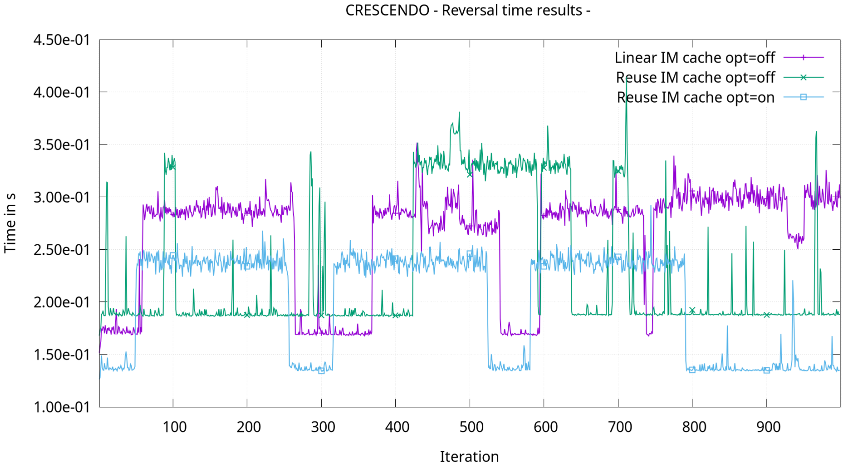
- Primal time: 0.15 sec. (1 G step)
- Primal memory: 12.85 GB
- Mach: 0.4
- *Re*: 800,000
- $\frac{\rho_{in}}{\rho_{out}}$: 1.27

- RPM: 4650
- Cells: 1.7 million
- TMTF with flow redirection of 40 degree
- Computed on 6 AMD EPYC 7262 Nodes (96 cores) on the Elwetritsch HPC cluster of the University Kaiserslautern-Landau (RPTU)

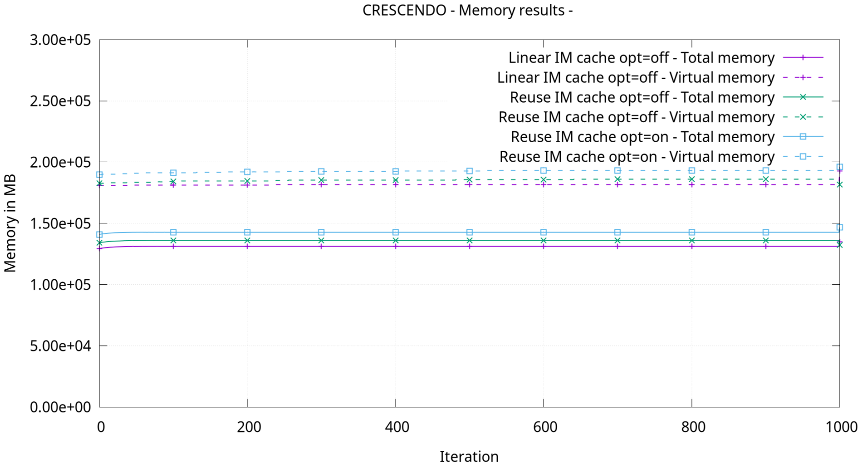
Performance results - recording time



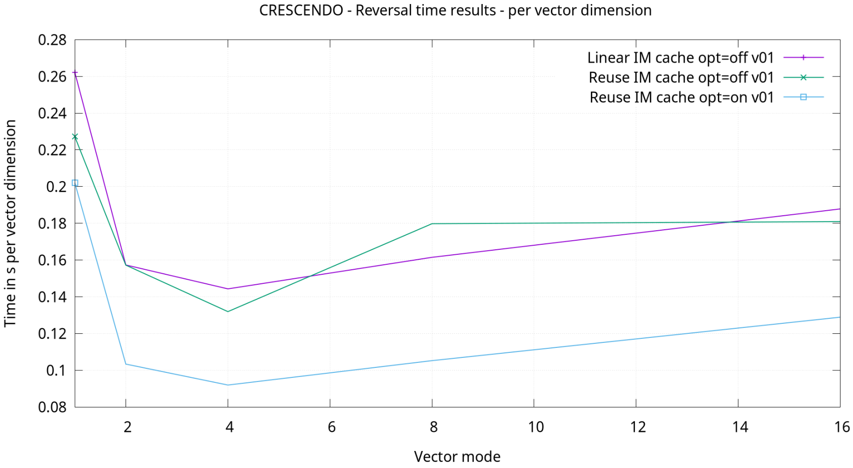
Performance results - reversal time



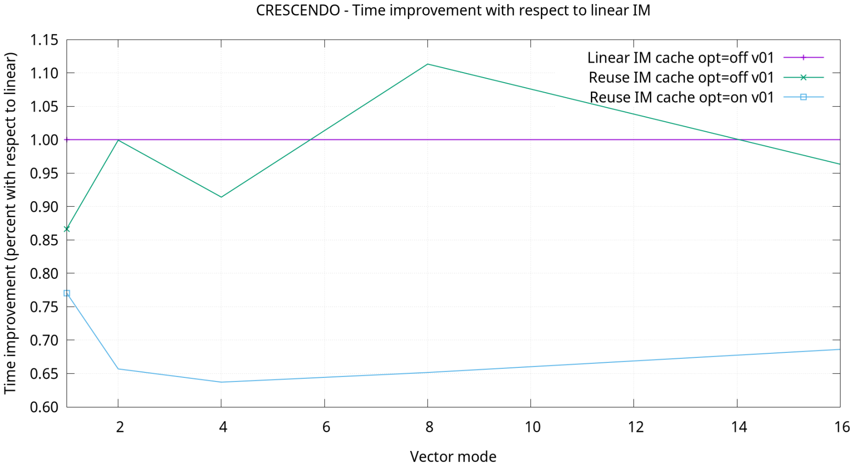
Performance results - memory



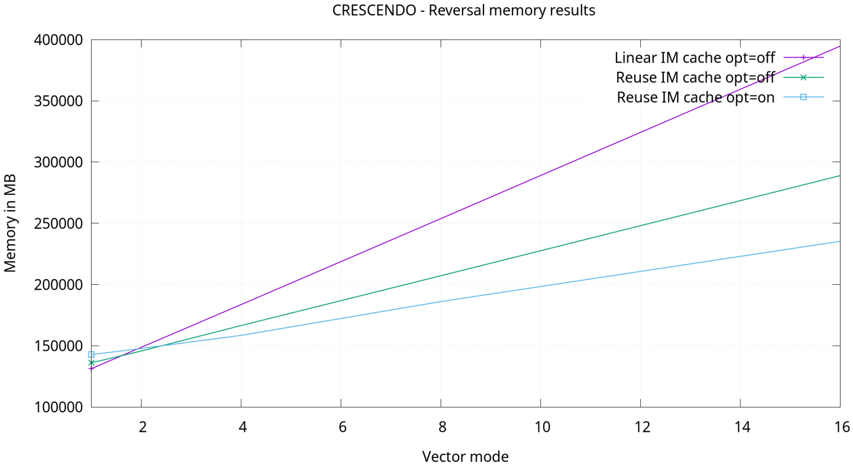
Performance results - vector mode reversal time



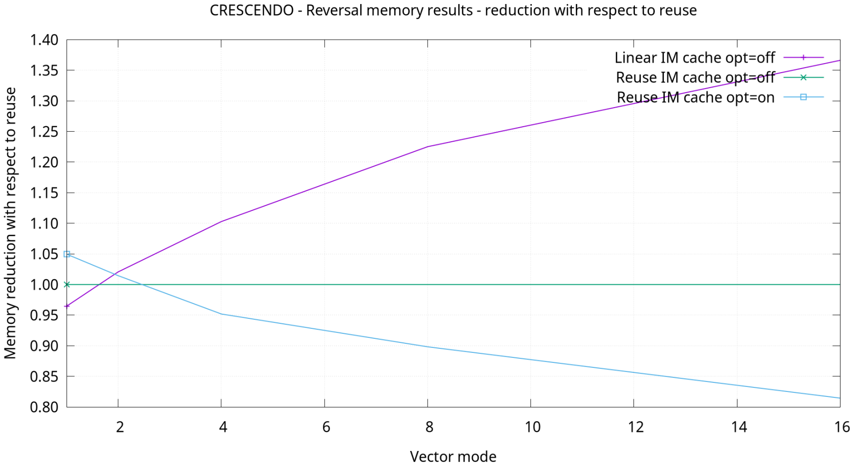
Performance results - vector mode reversal time



Performance results - vector mode memory



Performance results - vector mode memory



New CoDiPack features used

- Write and read tapes from disk
    - No support for external functions
- Full customization of the tape evaluation
    - Modify tape data on the fly
    - Access external function input and output identifiers
- New debug index manager
    - Automatically detects errors in tape recordings
- Tapes can now store arbitrary data in active types

Will be released soon$^{\text{TM}}$.

Conclusion & Outlook

Conclusion:

- New post processing feature for CoDiPack tapes
- Improves tape evaluation time
- Improves the memory requirement for the adjoint vector

Outlook:

- Improve speed for the post process
- Add additional heuristics for the analysis

Conclusion & Outlook

Conclusion:

- New post processing feature for CoDiPack tapes
- Improves tape evaluation time
- Improves the memory requirement for the adjoint vector

Outlook:

- Improve speed for the post process
- Add additional heuristics for the analysis

Thank you very much for your attention.