

# Teaching SIMD-Computing with a Vector Forward-Mode AD Tool

28th EuroAD Workshop

*Johannes Schoder*, Martin Bückler, Torsten Bosse, Mia Ohlrogge

Friedrich Schiller University Jena, Department of Computer Science, Chair for Advanced Computing

CERN, 10th December 2025

# The Assignment - Motivation

*"Improve some given C++-Code by using SIMD instructions and (optionally) other techniques like loop unrolling etc.  
(btw: the code uses AD)"*

- *Not a goal:* teaching all of the basics of AD
- *Primary goal:* teaching SIMD processing *sneaking in* AD as an illustrative example

## Hidden Goal - Recruitment

1. Gain visibility for undergraduate students
2. Advertise follow-up lectures (e.g., a dedicated course on AD)
3. Spot interested and motivated students

# Setting

- Course on parallel processing for shared-memory architectures and GPUs (CUDA, OpenMP, SIMD)
- Undergraduate students, sophomore year
- Two-week assignment (2 · 120 minutes)

## Prerequisites

- C/C++ programming
- Fundamental understanding of computer architecture
- Inline Assembly / calling Assembly kernels in C code / intrinsics
- Students already used SIMD instructions earlier in the course

# (Learning) Goals

- Use a real-world example: optimization using gradient descent
- *Sneaking in a tiny operator overloading vector forward-mode AD tool*
- Optimize existing C code for performance (using SIMD)
- Train code understanding
- (Understand how compilers and compiler optimization impact efficiency)

# Course Material Available

## Public Git Repository

[https://git.uni-jena.de/parallel\\_assignments/simd-ad](https://git.uni-jena.de/parallel_assignments/simd-ad)

Includes [1]:

- task description
- example code
- example solution (RISC-V ISA)

In our course, we use the ARM ISA. The assignment's solution for ARM is available upon request: [johannes.schoder@uni-jena.de](mailto:johannes.schoder@uni-jena.de)

# Application: Optimization

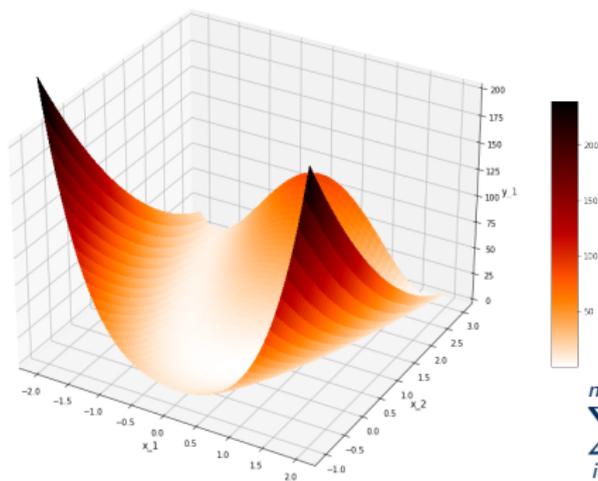


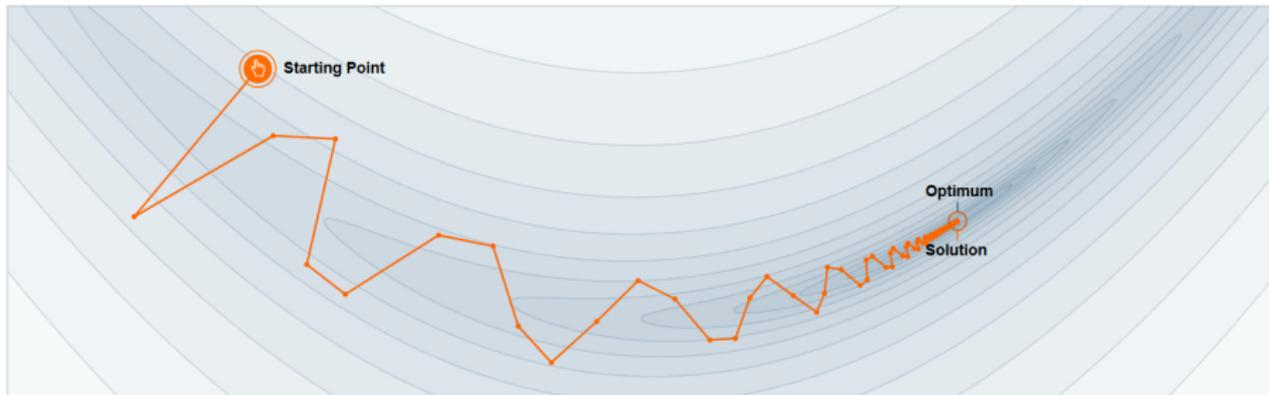
Figure 5 shows the 2-dimensional Rosenbrock function ( $f: \mathbb{R}^2 \rightarrow \mathbb{R}$ )

Rosenbrock( $\vec{x}$ ) =

$$\sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \quad (1)$$

Equation (1) gives the n-dimensional Rosenbrock function ( $f: \mathbb{R}^n \rightarrow \mathbb{R}$ )

# Gradient Descent with Momentum



Step-size  $\alpha = 0.0038$



Momentum  $\beta = 0.84$



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

(Credit: Gabriel Goh, UC Davis, "Why Momentum Really Works",  
<https://distill.pub/2017/momentum/> [2])

Source: [2]

# Tutorial

Students get a (very brief) introduction (approx. 60 minutes total) to:

- Gradient descent with momentum
- Vector forward-mode AD
- (Operator overloading)

# Material

Students get fully working code examples, which:

- Include an operator overloading AD class implementing
  - ADD
  - SUB
  - MUL
- The multi-dimensional Rosenbrock function using the custom AD class
- Gradient descent optimization for the Rosenbrock function
- Execution time measurement

Approximately 150-200 lines of code for each example.  
Slides that explain vector forward-mode AD are included  
→ *They are not expected to start from scratch!*

# Assignment Task

- Optimize the *already existing* vector forward-mode code by:
  - Adding and calling external Assembly kernels using SIMD instructions
  - (Optional) SIMD instructions as inline Assembly
  - (Optional) Intrinsics
  - (Optional) Loop unrolling, etc.
- Runtime tests for the different variants on a Raspberry Pi (ARM ISA)

# Vector Forward Mode

```
1 friend AD operator + (const AD& obj1, const AD& obj2) {  
2     AD temp;  
3     temp.val = obj1.val + obj2.val;  
4     for(int i = 0; i < MAX_DIR; i++){  
5         temp.adval[i] = obj1.adval[i] + obj2.adval[i];  
6     }  
7     return temp;  
8 }
```

Listing 1: Provided code example for addition in vector mode

# Vector Forward-Mode (Inline Assembly)

```
1 friend AD operator + (const AD& obj1, const AD& obj2) {
2     AD temp;
3     temp.val = obj1.val + obj2.val;
4     for(int i = 0; i < MAX_DIR; i+=SIMD_LANES ){
5         asm (
6             "ld1 {v1.4s}, [%0];"
7             "ld1 {v2.4s}, [%1];"
8             "fadd v0.4s, v1.4s, v2.4s;"
9             "st1 {v0.4s}, [%2];"
10            : //no output registers
11            : "r"((DTYPE*) &obj1.adval[i]),
12              "r"((DTYPE*) &obj2.adval[i]),
13              "r"((DTYPE*) &temp.adval[i]) //input registers
14            : "v2", "v1", "v0", "memory" //clobbers
15        );
16    }
17    return temp;
18 }
```

Listing 2: Possible solution using inline Assembly

# Vector Forward Mode (Intrinsics)

```
1 friend AD operator + (const AD& obj1, const AD& obj2) {  
2     AD temp;  
3     temp.val = obj1.val + obj2.val;  
4     for(int i = 0; i < MAX_DIR; i+=SIMD_LANES ){  
5         float32x4_t in1, in2, out;  
6         in1 = vld1q_f32(&obj1.adval[i]);  
7         in2 = vld1q_f32(&obj2.adval[i]);  
8         out = vaddq_f32(in1, in2);  
9         vst1q_f32(&temp.adval[i], out);  
10    }  
11    return temp;  
12 }
```

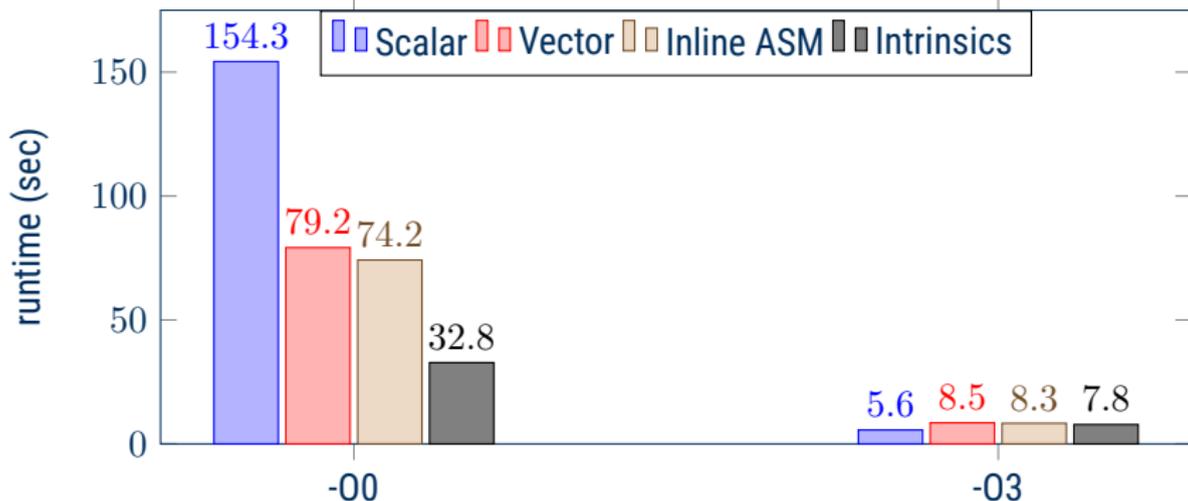
Listing 3: Possible solution using intrinsics

# Runtime

Here: Nvidia Jetson Nano in 5W mode (in class: Raspberry Pi)

(Run each experiment 3 times, selected fastest)

Note: always 31050 iterations to find the minimum, *except* in scalar mode with -03: 31063 iterations!



# Runtime

Careful: runtime results heavily depend on ISA/hardware architecture, compiler, and compiler options, our example works for ARM and RISC-V ISA

```
50 // overload the + operator
51 friend AD operator + (const AD& obj1, const AD& obj2) {
52     AD temp;
53     temp.val = obj1.val + obj2.val;
54     for(int i = 0; i < MAX_DIR; i+=SIMD_LANES){
55     simd t
56         "ldi {v1.4s}, [x0]; \n"
57         "ldi {v2.4s}, [x1]; \n"
58         "fadd v0.4s, v1.4s, v2.4s; \n"
59         "stl {v0.4s}, [x2]; \n"
60         //no output registers have to be rea
61         : "r"((DTYPE*) &obj1.adval[i]),
62           "r"((DTYPE*) &obj2.adval[i]),
63           "r"((DTYPE*) &temp.adval[i]) //input registers linked to var
64         : "v2", "v1", "v0", "memory" //clobbers = all used SIMD registers
65     };
66     return temp;
67 }
68 }
```



```
85 .L1:
87     ldsw  x0, [sp, 60]
88     lsl   x0, x0, 2
89     ldr   x1, [sp, 40]
90     add  x0, x1, x0
91     add  x0, x0, 4
92     ldsw  x1, [sp, 60]
93     lsl   x1, x1, 2
94     ldr   x2, [sp, 32]
95     add  x1, x2, x1
96     add  x1, x1, 4
97     ldsw  x2, [sp, 60]
98     lsl   x2, x2, 2
99     add  x2, x19, x2
100     add  x2, x2, 4
101     ldi  {v1.4s}, [x0];
102     ldi  {v2.4s}, [x1];
103     fadd v0.4s, v1.4s, v2.4s;
104     stl  {v0.4s}, [x2];
...
```

Credit: Compiler Explorer [3]:  
<https://godbolt.org/z/n7rMce15d>

# Student Feedback

(Selection of transcribed and translated written feedback)

## What *did* you like?

- Emphasis on performance testing
- Real-world application
- Deeper understanding of Assembly and processor design

## What *didn't* you like?

- Too easy
- Too complicated
- Introduction too short

## Sources:



J. Schoder, “Task git repository,” 2025. [Online]. Available: [https://git.uni-jena.de/parallel\\_assignments/simd-ad](https://git.uni-jena.de/parallel_assignments/simd-ad)



G. Goh, “Why momentum really works,” *Distill*, 2017. [Online]. Available: <http://distill.pub/2017/momentum>



M. Godbolt, “Compiler explorer.” [Online]. Available: <https://godbolt.org/>