# Infusing Computational Science into Computer Science: A Nifty SIMD Assignment

Johannes Schoder[1 [0000-0002-0771-3738]], Torsten F. Bosse[1 [0000-0002-4894-0742]], and H. Martin Bücker[1,2 [0000-0002-5210-0789]]

[1] Institute for Computer Science, Friedrich Schiller University Jena, Jena, Germany
[2] Michael Stifel Center Jena for Data-Driven and Simulation Science, Friedrich Schiller University Jena, Jena, Germany
{johannes.schoder,torsten.bosse,martin.buecker}@uni-jena.de

**Abstract.** Successfully solving challenging problems arising from real-world applications in computational science and engineering frequently requires a concerted effort from different scientific disciplines. Besides leveraging knowledge of science, technology, and mathematics, it is often crucial to also harness practical skills from computer science. Therefore, it is no surprise that elements from computer science are taught in education for computational scientists. However, it is rare to turn this process around and integrate aspects of computational science in courses for computer scientists. We describe a programming assignment that has been successfully taught in classroom to provide inspiration for computational science to computer science undergraduates. The assignment is used in a course on performance engineering and consists of programming SIMD instructions to implement automatic differentiation. We describe its tasks, share corresponding solutions, and discuss student feedback.

**Keywords:** Education · Computational Science · Single Instruction, Multiple Data (SIMD) · Automatic Differentiation · Software Performance Engineering.

## 1 Addressing Performance on the Level of Programs

The ultimate goal of computational science is to increase human understanding of complex phenomena in science, technology, and society by means of algorithms using computational resources. These resources vary widely and depend on a variety of problem-specific factors. Prominent examples of resources include running time, storage requirement, network utilization, parallel scalability, or energy consumption. Despite taking a large variety of different forms, computational resources share a common feature: scarcity. In most situations of practical interest, computational scientists need to work within severe constraints on computational resources. To take an evident example, consider a large organization that distributes a substantial part of its workload to cloud computing services. Here, the pay-as-you-go model is currently popular in which computational resources are invoiced for their actual usage [11]. Therefore, an

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

important question of a computational scientist is, how can the improvement in human understanding of a concrete problem of interest be maximized for a given limit on computational resources?

More specific examples of the previous general question are as follows.

- By up to which factor can the grid resolution be increased to obtain a more accurate numerical prediction of a groundwater flow in porous media given a running time of no more than $x$ hours on a compute cluster $y$?
- To what extent is it possible to refine a mathematical model of wheel-rail contact by adding more friction factors such as surface properties without exceeding the time limit of $x$ hours using up to $p$ processors in batch mode?

The key insight from these and related questions is that, in addition to understanding the characteristics of the problem, asking the right questions, and using sophisticated numerical techniques, computational science education also needs to pay attention to performance engineering. Any increase in performance can be traded off against other metrics of interest to computational science such as refining spatial and temporal discretizations, integrating over longer time periods, and improving mathematical models.

Performance engineering is a multifaceted subdiscipline of computer science which is increasingly getting more complex. It comes in different flavors, including exploitation of deep memory hierarchies, designing efficient algorithms on parallel systems, and understanding compilation techniques, to name a few. Rather than concentrating on the “bottom” of the computing-technology stack, this article addresses performance at the “top” of this stack [10]. The reason is that current trends in computer architecture strongly suggest to educate the next generation of computational scientists to address performance not by relying on black box compilation systems, but developing software explicitly tailored for modern hardware architectures by themselves.

Since most universities cannot afford to offer multiple courses on a single topic, there is every reason to teach a course on performance-oriented programming to students enrolled in computer science together with those enrolled in computational science. For such a course primarily attended by computer science undergraduates, this article introduces a programming assignment as an instrument to thoroughly explore the individual topic of explicitly programming Single Instruction, Multiple Data (SIMD) instructions. These vector instructions that operate on multiple data elements concurrently are taught using automatic differentiation, a powerful technique frequently used in computational and data science as sketched in Sect. 2. The university setting and the course in which the assignment is offered are outlined in Sect. 3, followed by the design goals summarized in Sect. 4. The programming assignment is detailed in Sect. 5 and teaching experiences are reported in Sect. 6. Related work is sketched in Sect. 7.

## 2 Automatic Differentiation in Computational Science

The goal of designing a novel programming assignment for a course primarily taught for undergraduates in computer science is not only to teach the fun-

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

damental principles of vector instructions, but also to provide inspiration for computational science. Automatic differentiation (AD) [5] is a prime candidate to establish an intimate connection between vector instructions and computational science. The reason is twofold. Firstly, derivatives play a crucial role in computational science. For instance, they arise when linearizing nonlinear phenomena and are widely used in a variety of different numerical techniques. They also appear in the solution of systems of nonlinear equations and of differential equations. They lay the foundations of sensitivity analysis and are a cornerstone of local optimization methods for continuous objectives with applications ranging from inverse problems, to experimental design, and to machine learning.

Secondly, vector instructions are naturally present in AD because AD techniques take as input a program executing a potentially long sequence of scalar operations $\phi$ of the form

$$w \leftarrow \phi(u, v), \quad \text{where} \quad u, v, w \in \mathbb{R}, \tag{1}$$

and generate as output a program that transforms each of the previous scalar statements into the corresponding vector statement

$$\mathbf{w} \leftarrow \alpha \cdot \mathbf{u} + \beta \cdot \mathbf{v}, \quad \text{where} \quad \alpha, \beta \in \mathbb{R} \quad \text{and} \quad \mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^p. \tag{2}$$

Here, the scalars are given by $\alpha = \partial\phi/\partial u$ and $\beta = \partial\phi/\partial v$ and the vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ denote the gradients of $u, v$ and $w$ with respect to $p$ scalar parameters.

As an example consider the problem of finding the minimum of the Rosenbrock function $f : \mathbb{R}^n \to \mathbb{R}$ given by

$$f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2, \tag{3}$$

with $n \geq 2$. Using AD to solve this nonlinear optimization problem via a gradient-based algorithm, we set $p = n$ and any intermediate scalar operation to evaluate the function (3) such as the binary addition $w \leftarrow u + v$ is transformed into the corresponding vector operation, in this case $\mathbf{w} \leftarrow \mathbf{u} + \mathbf{v}$. Then, applying this mechanical process to the complete sequence of scalar operations provides the gradient of $f$ with respect to $x_1, x_2, \ldots, x_n$. The cases $p = 1$ and $p > 1$ are referred to as *scalar* and *vector mode*, respectively.

## 3 Course "Efficient Computing" at University Jena

With around 17,000 students across 10 academic divisions, Friedrich Schiller University Jena, which was founded in 1558, is among the medium-sized schools in Germany. Since 2014, its Faculty of Mathematics and Computer Science offers a two-year master's degree program in "Computational and Data Science" to join the two disciplines computational science and data science. This program is open to students with a bachelor's degree not only in computer science and

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

mathematics, but also in natural sciences and engineering. These diverse scientific backgrounds require a considerable effort for course design. However, at the same time this situation opens the door to implement teaching activities that bring together students on the master and bachelor level in a single course.

In 2024, the bachelor's degree program in "Computer Science" was redesigned. One of its new elements is the course "Efficient Computing," giving an introduction to programming modern computers with a particular emphasis on high performance. Though the majority of its participants is enrolled in the second year of that program, the course is also mandatory for master's in "Computational and Data Science" in their first year and is co-designed for both audiences. This heterogeneity provides the opportunity to expose computer science undergraduates to carefully selected, engaging problems from computational science.

The course is organized in the format of one lecture and one exercise per week, 90 minutes each, and spans about 15 weeks. It gives an introduction to the basic principles of performance engineering. It covers serial processing on systems with deep memory hierarchies and then concentrates on parallel processing for shared-memory architectures (OpenMP) and graphics processing units (CUDA).

The central objective of this article is to describe the individual topic of a programming assignment asking students to increase the performance of an existing code by manually adding SIMD instructions. Optionally, students are encouraged to use intrinsics and further performance optimization strategies such as loop unrolling. The proposed assignment is solved by the students within two consecutive weeks. Its solution requires the following prerequisites. First and foremost, practical programming skills in C/C++ are necessary. Additionally, students need a basic understanding of computer architecture, in particular the von Neumann model. Furthermore, the assignment expects students to write assembly code for the ARMv8a instruction set architecture (ISA). Hence, students should either be familiar with this ISA or should be capable of working with the official ARM ISA guides and datasheets. Students also require an understanding of compile chains and linking. Specifically, they are supposed to compile C++ code, disassemble, assemble, and link using the GNU Compiler Collection (gcc).

The assignment is preceded by other assignments and introductory lectures on SIMD instructions. Hence, students already have practice in the related concepts of loop unrolling, pipelining, inline assembly, and calling assembly kernels from C code. They are also familiar with the terms *performance* and *peak performance* both measured in floating-point operations per second (FLOP/s) and, for a given hardware, they are capable of calculating and measuring both.

Other topics needed to solve the assignment are briefly presented in a tutorial session before handing out the assignment to the students. This tutorial takes approximately 30–60 minutes in total. Students get a brief introduction to nonlinear optimization by gradient descent. Special attention is given to a short introduction to the forward mode of AD which is represented by (1) and (2). Slides explaining this technique are included in the course material. Optionally, for students unfamiliar with operator overloading, a concise introduction directly linked to the code handed out to the students is offered.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

## 4 Instructional Goals

The interest in designing the assignment is primarily caused by our motivation to inspire and enable young students to realize the full potential of interdisciplinary thinking. While it is clear to computational scientists that collaborations between scientific disciplines can unlock new insight, an interdisciplinary spirit is typically not well established in a computer science curriculum. So, we wanted to enrich a course mainly attended by computer scientists with a concrete example from computational science that illustrates the high relevance of AD and its excellent opportunities in application areas.

Moreover, undergraduates near the end of their second year as well as students beginning their master's program have already developed and refined their skills in programming, computer architecture, and problem solving. This progress allows for tackling more complex and intricate tasks. However, within the short period of a one-semester class covering a wide range of different topics, it is often difficult to justify addressing a specific topic with scrupulous attention to detail. So, an assignment on a single topic designed to grasp the concept behind that topic is often demonstrated by a small example. We wanted to go beyond toy examples and showcase "cool stuff" and "hot topics" in somewhat greater depth.

Finding interesting and challenging assignments is not only of pedagogic importance, but is also a vital part of recruiting the next generation of academics. An intense competitive pressure with industry forces academic teachers to spot and approach motivated students early throughout their curriculum. Since we also faced difficulties when trying to attract qualified candidates for bachelor's theses, another goal was to make known the AD technology as a powerful and versatile tool for solving problems of practical interest. Our group is constantly offering theses on research-oriented topics in AD which is typically not introduced in undergraduate programs, but appears only at the master's level.

## 5 Programming Assignment: Tasks and their Solutions

As previously mentioned, the assignment starts with a tutorial session. Here, the problem of finding the location of a future geothermal power plant based on measurements of the temperature in deep boreholes is taken as an illustrative example to motivate the need for solving optimization problems. While this particular problem instance is certainly relevant for computational science it leads to excessively large running times in a programming assignment for students. However, if this optimization were to be solved in the assignment, the programming tasks and their corresponding solutions would be identical to an assignment minimizing the Rosenbrock function. Since it does not take much imagination to replace the objective of the geothermal optimization problem by the objective (3) we prefer the latter with $n = 100$. This choice reduces the students' time for software development, enabling debugging in small and practicable steps.

Before introducing the actual assignment, we provide an overview on three variants of AD implementations and their notations used throughout this article.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

All variants are implemented via operator overloading using a class `AD` with data members `val` and `adval` representing values and their derivatives, respectively.

- ***Scalar***: This variant implements the scalar mode corresponding to $p = 1$. Here, all objects of the class `AD` are instantiated with the data member `adval` which is a scalar used to compute a single derivative.
- ***Vector***: This variant implements the vector mode corresponding to $p > 1$. Here, all objects of the class `AD` are instantiated with the data member `adval` which is a vector used to compute $p$ derivatives simultaneously.
- ***Kernels***: This SIMD variant also implements the vector mode. However, in contrast to the previous variant ***Vector***, the implementation is now based on calling assembly kernels with SIMD instructions.

In summary, the three variants ***Scalar***, ***Vector***, and ***Kernels*** compute the gradient of the function $f$ with respect to $x_1, x_2, \dots, x_n$ but differ in how they compute the $n$ gradient entries. The implementations of ***Scalar*** and ***Vector*** are considered to be lightweight, each comprising approximately 150–200 lines of code, and are given to the students as part of the assignment. Students are asked to implement the variant ***Kernels*** by the following two mandatory tasks.

1) Implement the SIMD variant ***Kernels*** by overloading the operators `+`, `-`, and `*` of the class `AD`. The corresponding three routines have to be written in assembly, using SIMD instructions with 4 variables of type `float` per SIMD register, and should perform the same computations as in the non-SIMD variant ***Vector***.
2) Run the three variants ***Scalar***, ***Vector***, and ***Kernels*** on a Raspberry Pi 4B and report their performance in FLOP/s.

Besides the mandatory tasks, there are two optional bonus tasks.

- ***Inline***: Implement a (SIMD) variant of ***Kernels*** using inline assembly.
- ***Intrinsics***: Implement a (SIMD) variant of ***Kernels*** using intrinsics [1].

The precise instructions describing these programming tasks as well as a set of corresponding source files handed out to the students are publicly available at [15]. Part of that repository is an implementation of a gradient-based optimization algorithm using an approach with momentum [18,4]. All variants use the same values for the step size, the momentum, and identical starting vectors for the gradient-descent algorithm. Note that the hyperparameters are not tweaked to find the minimum particularly fast. Instead, they are set to values that enable accurate and reliable time measurements on the Raspberry Pi.

To give the reader a more detailed perspective on the assignment, the following discussion summarizes the different variants.

The variant ***Scalar*** implements the scalar mode and serves as a baseline implementation. An example of the overloaded operator `+` is shown in Listing 1.1. Here, the scalar mode propagates values stored in the scalar data member `val` and, in addition, it is capable of computing a single partial derivative at a time

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

represented by the scalar data member `adval` alongside evaluating the original function $f$. In each iteration of the algorithm to minimize the Rosenbrock function $f$ with $n = 100$, the gradient of $f$ consisting of 100 entries is required. Thus, in each iteration, this variant needs 100 evaluations of $f$ to compute the gradient, resulting in 99 redundant evaluations of $f$.

```
friend AD operator + (const AD& obj1, const AD& obj2) {
  AD temp;
  temp.val = obj1.val + obj2.val;
  temp.adval = obj1.adval + obj2.adval;
  return temp;
}
```

**Listing 1.1.** Variant ***Scalar*** without any code optimizations.

By implementing the vector mode, the variant ***Vector*** removes the aforementioned unnecessary overhead of redundantly evaluating $f$. Since `adval` is now a vector, the resulting overloaded operator `+` sketched in Listing 1.2 uses a `for` loop over all partial derivatives. The constant `MAX_DIR` defines the number of these derivatives. Therefore, while executing the original function once, the gradient of length $n = 100$ is computed alongside. A comparison of this variant with the previous variant ***Scalar*** both of which are handed out to the students shows that algorithmic choices can significantly change the number of operations, potentially leading to improved performance. Notice that this particular implementation of ***Vector*** does not contain any performance optimizations.

```
  friend AD operator + (const AD& obj1, const AD& obj2) {
    AD temp;
    temp.val = obj1.val + obj2.val;
    for(int i = 0; i < MAX_DIR; i++){
      temp.adval[i] = obj1.adval[i] + obj2.adval[i];
    }
    return temp;
}
```

**Listing 1.2.** Variant ***Vector*** without any code optimizations.

The third variant, ***Kernels***, expects the students to modify the `for` loops within the overloaded operators of the ***Vector*** variant in an attempt to increase the performance. More precisely, the `for` loops contained in the overloaded operators such as in Listing 1.2 are replaced with kernels that call assembly code. An example is illustrated in Listing 1.3.

```
friend AD operator + (const AD& obj1, const AD& obj2) {
  AD temp;
  temp.val = obj1.val + obj2.val;
  kernel_add(obj1.adval, obj2.adval, temp.adval, MAX_DIR);
  return temp;
}
```

**Listing 1.3.** Variant ***Kernels*** calling the assembly kernel shown in Listing 1.4.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

Students are expected to program these kernels such as `kernel_add` themselves using assembly and SIMD instructions. An example illustrating again the overloaded operator `+` is depicted in Listing 1.4. In this simple implementation, the instruction

```
ld1 {v1.4s}, [x0], #16
```

loads four floating-point values in single precision from the memory address stored in register `x0`, writes it to the vector register `v1`, and post-increments the value of `x0` by 16. The second vector register `v2` is then similarly filled with data. After executing the SIMD instruction `fadd` the data is finally stored in memory. The `while` loop takes care of iterating $n/4$ times over the data so that all $n$ entries of the gradient are computed. This solution could be improved further by, e.g., adding loop unrolling while simultaneously reducing read-after-write dependencies in the kernel code.

```
1      .text
2      .align   4
3      .type    kernel_add, %function
4      .global kernel_add
5  kernel_add:
6  while:
7      cmp x3, xzr
8      b.eq finish
9      ld1 {v1.4s}, [x0], #16
10     ld1 {v2.4s}, [x1], #16
11     fadd v0.4s, v1.4s, v2.4s
12     st1 {v0.4s}, [x2], #16
13     sub x3, x3, #4
14     b while
15 finish:
16     ret
17     .size    kernel_add, (. - kernel_add)
```

**Listing 1.4.** Assembly kernel called by variant ***Kernels*** given in Listing 1.3.

The next variant, ***Inline***, is based on a technique called inline assembly, offering a comfortable way to embed assembly code directly into C/C++ code. Thus, it provides a compelling alternative to the kernel calls of the variant ***Kernels*** in Listing 1.3. An example of the overloaded `+` operator is shown in Listing 1.5. Here, the constant `SIMD_LANES` specifies the length of the vectors supported by the SIMD instructions. The assembly statements are written in strings and inlined using the extension `asm()`.

There is a plethora of possible solutions to ***Inline***. For instance, the `for` loop of the solution given in Listing 1.5 is left as C/C++ code. An alternative is to include that loop within the assembly code. As a result, overhead arising by executing the inline assembly such as securing register contents could potentially be reduced.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

```
friend AD operator + (const AD& obj1, const AD& obj2) {
  AD temp;
  temp.val = obj1.val + obj2.val;
  for(int i = 0; i < MAX_DIR; i+=SIMD_LANES){
    asm (
    "ld1 {v1.4s}, [%0];"
    "ld1 {v2.4s}, [%1];"
    "fadd v0.4s, v1.4s, v2.4s;"
    "st1 {v0.4s}, [%2];"
    :   //no output registers
    :   "r"((DTYPE*) &obj1.adval[i]),
        "r"((DTYPE*) &obj2.adval[i]),
        "r"((DTYPE*) &temp.adval[i]) //input registers
    :   "v2", "v1", "v0","memory" //clobbers
    );
  }
  return temp;
}
```

**Listing 1.5.** Variant ***Inline***.

The last variant addressed in this assignment is called ***Intrinsics***. It offers an alternative to calling external assembly kernels as carried out by ***Kernels*** given in Listing 1.3 by using intrinsics. An example is demonstrated in Listing 1.6. From a functional point of view, ***Intrinsics*** performs exactly the same operations as ***Inline***. However, the assembly instructions are wrapped inside function calls. While a compiler such as gcc is not capable of optimizing the inline assembly directly, it might consider performance optimization for intrinsics.

```
friend AD operator + (const AD& obj1, const AD& obj2) {
  AD temp;
  temp.val = obj1.val + obj2.val;
  for(int i = 0; i < MAX_DIR; i+=SIMD_LANES){
    float32x4_t in1, in2, out;
    in1 = vld1q_f32(&obj1.adval[i]);
    in2 = vld1q_f32(&obj2.adval[i]);
    out = vaddq_f32(in1, in2);
    vst1q_f32(&temp.adval[i], out);
  }
  return temp;
}
```

**Listing 1.6.** Variant ***Intrinsics***.

After implementing the different variants, the students should use them to perform runtime measurements and calculate their performance in FLOP/s.

To conduct these runtime and performance experiments, the students had access to a Raspberry Pi 4B. However, the experiments do not necessarily have to be carried out on that machine, but can also be carried out on various other suitable hardware platforms. Depending on the chosen hardware platform, there are certain settings that should be taken into account to obtain reliable results.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

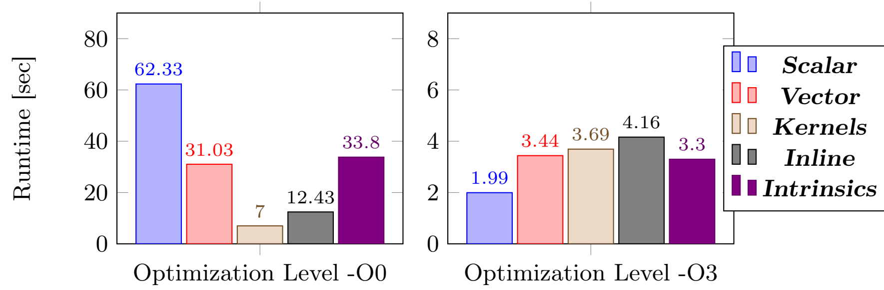


**Fig. 1.** Running times of the non-SIMD variants ***Scalar*** and ***Vector*** compared with the SIMD variants ***Kernels***, ***Inline***, and ***Intrinsics*** executed on Raspberry Pi 4B.

For example, for the Raspberry Pi 4B, it is recommended to set the CPU governor to *performance mode* enabling a maximal clocking rate of 1.6 GHz.

Figure 1 shows exemplary results obtained during these experiments. The results represent the average runtime of the variants for different compiler options. From the results without compiler optimization (`Optimization Level -O0`), it is obvious that manual optimization strategies can be beneficial and improve the runtime of an application by an order of magnitude. For this experiment in particular, the variants ***Kernels*** and ***Inline*** are typically 3–4 times faster than the baseline variant ***Vector*** and almost 5–9 times faster than the naive and algorithmically sub-optimal variant ***Scalar***.

Once the advantages of specialized SIMD kernels have been observed and understood, further and more in-depth questions can be addressed with the students. For example, investigating why the variant ***Intrinsics*** is slightly slower than the baseline variant ***Vector***. Another question that sparks a fruitful discussion is the investigation of the runtime results when the compiler optimization is enabled (`Optimization Level -O3`). To interpret and explain the runtime results, students have to understand concepts such as function inlining, function calls, and stack operations. Also, they should be able to analyze disassembled code and set code optimization into context of the application.

Another result from these experiments showing how important it is to set code optimization into context of the application is given in Table 1. These numbers indicate how code optimization affects the number of iterations to find the minimum of the Rosenbrock function, namely, how changes on the lowest compiler code level can lead to unexpected numerical results on application level.

**Table 1.** Required iterations of all variants with and without compiler optimization.

| Level | ***Scalar*** | ***Vector*** | ***Kernels*** | ***Inline*** | ***Intrinsics*** |
|---|---|---|---|---|---|
| `-O0` | 24880 | 24880 | 24962 | 24962 | 24962 |
| `-O3` | 24849 | 25161 | 24962 | 24962 | 24962 |

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

## 6 Teaching Experiences

Infusing computational science into the computer science curriculum is a delicate balancing act. Rigid learning objectives defined by the curriculum often leave little room for an in-depth introduction to computational science problems. At the same time, incorporating real-world problems rather than purely academic examples is essential for improving student motivation and learning outcomes. To assess whether the presented SIMD assignment achieves this balance, we conducted both quantitative and qualitative analyses. The quantitative evaluation examines the correlation between students' performance on the assignment and their final examination results, while the qualitative analysis focuses on free-text responses from student feedback.

**Quantitative Analyses** To qualify for the final exam, students were required to attain a minimum cumulative score (14 out of 28 points) on all their assignments. The SIMD assignment was worth a maximum of 9 points. The points were distributed equally across three categories reflecting its primary learning goals: correctness of code, concise and accessible commenting, and performance measurements. The students were encouraged to solve the assignment working in teams of up to three members without any collaboration policies.

Out of the 94 students initially enrolled in the course, we consider in Fig. 2 the results of the subset of 53 students who handed in the SIMD assignment and who completed the final exam. Students who were ill or failed in the first final exam were allowed a resit in the final exam. Figure 2 illustrates the correlation between the percentage of points earned on the assignment and the corresponding exam results for each of these participants. The exam is failed if less than 50% of points are achieved (gray shading). The data reveals a slight positive correlation, indicated by the red regression line. On the one hand, almost no student achieved a high exam score without also performing well on the assignment. Namely, with only two exceptions, scoring above 80% on the exam was almost exclusively limited to students who also performed similarly well on the SIMD assignment. On the other hand, all but one of the students who scored above 80% on the assignment passed the exam. When analyzing this figure, keep in mind that (i) roughly a quarter of all points of the exam were related to SIMD and three quarters of all points dealt with other topics and (ii) students were allowed to complete the assignment collaboratively, leaving them the individual responsibility to actively take part in the learning process.

**Qualitative Analyses** Friedrich Schiller University offers an optional survey for assessment of the individual courses. Within these surveys, there is an option for educators to add individual questions. In summer term 2025, we took the opportunity to evaluate the success of the SIMD assignment. More specifically, we added two free-form questions and two questions with a rating scale. The questionnaire was answered by 25 students, 20 of which were enrolled as undergraduates, 5 as master's students and 1 person was enrolled in a distinct diploma

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

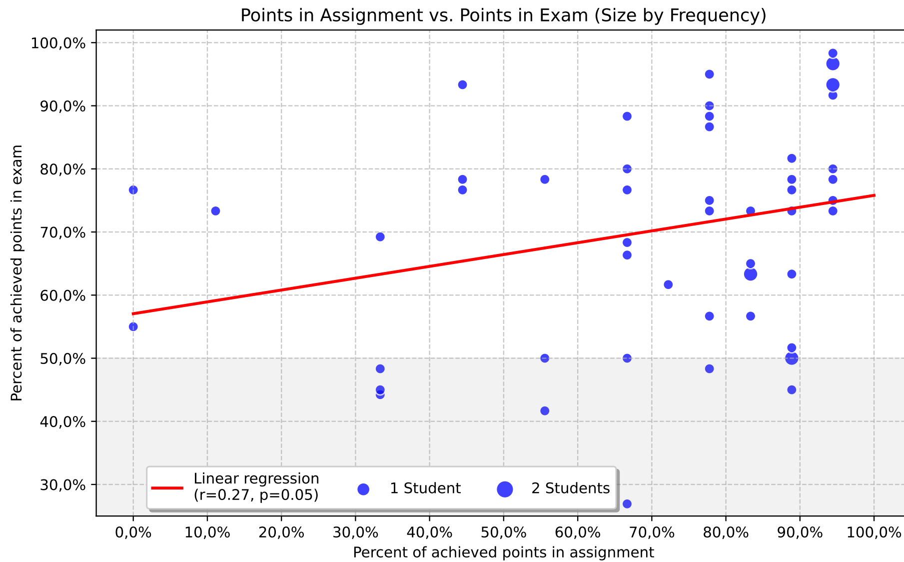


**Fig. 2.** Correlation between assignment and exam points.

in Germany for future secondary school teachers. We assume that one person was simultaneously enrolled in two distinct courses of study. While the number of responses is certainly too small to draw general conclusions, we believe this feedback to be interesting. The transcribed and translated written feedback of the free-form questions about the assignment is summarized in the following.

*Q1: What* did *you like about the assignment?*

- Adequate time frame to solve the assignment through fill-in style of existing code, enabling a deeper understanding of vectorized vs. SIMD.
- Emphasis on performance measurements and paying attention to efficiency.
- In-depth introduction to SIMD registers and illustration of an application.
- Real-world application instead of a purely academic example.
- Step-by-step introduction and extensive tutorial.
- Deeper understanding of assembler and internal processes within the processor.
- Content is well taught. Obtaining information directly within the classroom and the possibility to ask questions.
- Distinguishing between the variants ***Scalar***, ***Vector***, and ***Kernels*** helped in understanding the differences.

*Q2: What* didn't *you like about the assignment in particular?*

- Somewhat repetitive workload.
- I would suggest more complicated tasks on assembly programming.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

- To some extent being "just thrown in at the deep end", without clear instructions on the assignment.
- The extensive set of codes initially given to the students partly reduced the motivation to look into further methods, for example, methods for time measurement.
- Unclear how to remotely access the computing systems.
- Lacking documentation on this topic in general, the course material gives some information but hardly provides an overview of the "big picture". Unfortunately, the number of grading points of this assignment is comparatively large making it difficult to catch up with points if the solution of the assignment is missing.

In total, eight students answered the first free-form question (Q1), while six students answered the second one (Q2).

Furthermore, the two additional questions Q3 and Q4 use a rating scale ranging from 1 (disagree) to 5 (agree). Question Q3 assesses the perceived difficulty of the assignment by stating

*Q3: The assignment was adequately difficult.*

The other question (Q4) aims to evaluate whether or not the illustrative example aided in understanding the concepts of performance by stating

*Q4: The assignment helped in understanding of performance.*

Both questions answered by 22 students resulted in the following responses shown in Fig. 3. We conclude that students were mostly satisfied with the assignment. Free-form responses indicate that the critical learning goals were indeed met for some of the students by enabling a deeper understanding of SIMD instructions and code optimization to improve the performance of a given code describing a real-world problem from computational sciences.

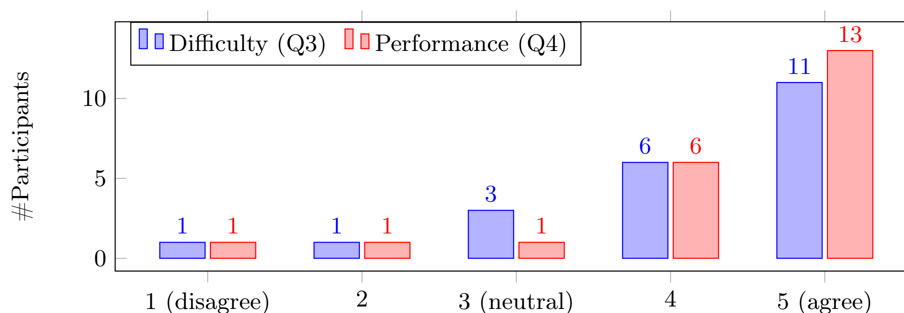


**Fig. 3.** Additional questions with a rating scale ranging from 1 (disagree) to 5 (agree).

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

## 7 Related Work

Guidelines for computer science education such as [7] acknowledge the relevance of SIMD and suggest to give room for this topic in a curriculum whereas the topic of AD is not mentioned in [7] at all. The combination of SIMD and AD is not new as a research topic [6,16]. However, to the best of the authors' knowledge, this combination has not been discussed in any publication on teaching. The literature on introducing the principles of SIMD programming in forms of textbooks [9] or course descriptions [8,12] is vast, none of which contains any aspects of AD. Some textbooks on programming [2], optimization [13,18], or machine learning [17,3] include short introductions to AD, but without any connection to SIMD. One of the aims of the recent textbook [14] is to bridge the gap between the areas of computer architecture and machine learning and contains both topics, SIMD and AD.

## 8 Concluding Remarks

The proposed assignment connecting SIMD with AD is an individual teaching activity addressing current trends in computer architecture. These trends suggest that performance engineering is becoming increasingly important on the level of programming. Therefore, any scientific discipline with a strong emphasis in computing needs to rebalance its subjects within the curriculum. An education preparing students for the modern society requires a thorough concentration on programming that pays attention to performance, not only for degree programs in computer science, but also in computational science. The key idea of the new programming assignment is to bring together performance-oriented SIMD programming with automatic differentiation, a technology with a growing relevance in computational science. Student feedback shows that the intended interdisciplinary spirit of the assignment is appreciated.

**Acknowledgments.** This work is funded in part by the German Federal Ministry of Education and Research (BMBF), within the project THInKI, project number 16DHBKI084, and by the Carl Zeiss Foundation within the project P2021-02-005 "Interactive Inference."

## References

1. ARM Limited: NEON Intrinsics, chapter of the eletronic book: NEON™ Programmer's Guide: Version 1.0. ARM, Cambridge, England (2013), https://developer.arm.com/documentation/den0018/a/NEON-Intrinsics
2. Barton, J.J., Nackman, L.R.: Scientific and Engineering C++: An Introduction with Advanced Techniques. Addison-Wesley, Reading (1994)
3. Bishop, C.M., Bishop, H.: Deep Learning: Foundations and Concepts. Springer International Publishing, Cham, Switzerland (2024). https://doi.org/10.1007/978-3-031-45468-4

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34

4. Goh, G.: Why momentum really works. Distill (2017). https://doi.org/10.23915/distill.00006
5. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 105 in Other Titles in Applied Mathematics, SIAM, Philadelphia, PA, 2nd edn. (2008). https://doi.org/10.1137/1.9780898717761
6. Hückelheim, J., Schanen, M., Narayanan, S.H.K., Hovland, P.: Vector forward mode automatic differentiation on SIMD/SIMT architectures. In: 49th International Conference on Parallel Processing – ICPP. ICPP'20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3404397.3404470
7. Kumar, A.N., Raj, R.K., Aly, S.G., Anderson, M.D., Becker, B.A., Blumenthal, R.L., Eaton, E., Epstein, S.L., Goldweber, M., Jalote, P., Lea, D., Oudshoorn, M., Pias, M., Reiser, S., Servin, C., Simha, R., Winters, T., Xiang, Q.: Computer Science Curricula 2023. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3664191
8. Kumar, V.: Teaching task-based parallel programming with a runtime systems-aware perspective. In: Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 376–383. SC Workshops '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3731599.3767387
9. Kusswurm, D.: Modern Parallel Programming with C++ and Assembly Language: X86 SIMD Development Using AVX, AVX2, and AVX-512. Apress (2022). https://doi.org/10.1007/978-1-4842-7918-2
10. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's plenty of room at the Top: What will drive computer performance after Moore's law? Science **368**(6495), eaam9744 (2020). https://doi.org/10.1126/science.aam9744
11. Lin, C., Shahrad, M.: Bridging the sustainability gap in serverless through observability and carbon-aware pricing. SIGENERGY Energy Informatics Review **4**(5), 120–126 (2025). https://doi.org/10.1145/3727200.3727218
12. Maurer, W.D.: Integrating SIMD into the undergraduate curriculum. Journal of Computing Sciences in Colleges **16**(4), 162–173 (2001)
13. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer, New York, 2nd edn. (2006). https://doi.org/10.1007/978-0-387-40065-5
14. Reddi, V.J.: Introduction to Machine Learning Systems. MIT Press, Cambridge, MA, USA (2026), https://mlsysbook.ai/book
15. Schoder, J., Bosse, T.F., Bücker, H.M.: Single instruction, multiple data (SIMD) illustrated by automatic differentiation. Git Repository (2026), https://git.uni-jena.de/parallel_assignments/simd-ad
16. Shin, S., Anitescu, M., Pacaud, F.: Accelerating optimal power flow with GPUs: SIMD abstraction of nonlinear programs and condensed-space interior-point methods. Electric Power Systems Research **236**, 110651 (2024). https://doi.org/10.1016/j.epsr.2024.110651
17. Watt, J., Borhani, R., Katsaggelos, A.K.: Machine Learning Refined: Foundations, Algorithms, and Applications. Cambridge University Press, Cambridge, UK, 2nd edn. (2020). https://doi.org/10.1017/9781108690935
18. Wright, S.J., Recht, B.: Optimization for Data Analysis. Cambridge University Press, Cambridge, UK (2022). https://doi.org/10.1017/9781009004282

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_34