Project 12, HPC4WCM 2024

Using AI language models to optimize the runtime of a Fortran based code in the context of weather and climate models

Jesse Connolly, Julien Delbeke, Nathanël Aidlin, Pavel Filippov

August 31, 2024

Abstract

This report explores the use of artificial intelligence in the context of code optimization for faster run-times on high performance computing systems. The free version of Gemini and ChatGPT are leveraged, to optimize computational efficiency in a Fortran code designed for simulating diffusion processes in weather and climate models using a stencil code. Various optimization strategies were applied, including loop fusion, inlining, blocking, memory allocation and parallelization techniques using OpenMP. While the results demonstrate improvements in performance, with the AI-optimized code exhibiting reduced runtime compared to the plain code, difficulties are also identified and discussed.

1 Introduction

The rapid growth in the popularity and capabilities of artificial intelligence (AI) has revolutionized much of the modern world by automating complex tasks and uncovering insights from vast datasets. Following the recent uptake in the use of AI, a question that comes up is how reasonably AIs can be used in scientific context to facilitate specific work, and if the results it produces can be trusted? In a field like high performance computing (HPC) one might be tempted to explore the use of AI as an aid to the coding process. As weather and climate systems are governed by intricate interactions between the atmosphere, oceans, land surfaces and many other factors, these models require immense computational power to solve the underlying mathematical equations that describe these dynamics. Weather and climate models are known for data intensity as these models process vast amounts of observational data, which are used to initialize, run and validate the simulations. Said models have to manage and analyze these large data sets efficiently to reduce computational costs and and runtime so that predictions become feasible, which is where HPC comes into play. Hence in this report we want to dive into if and how language based AI can be used to optimize the run time of a given specific Fortran code.

Subroutine	Description
`apply diffusion`	Performs the 4th-order diffusion calculation by iteratively updating the in_field and storing the result in out_field. This subroutine uses the allocation of temporary fields in the iterative process.
`laplacian`	Computes the laplacian of the in_field using the 2nd-order centered differences and stores the result in the lap_field. This subroutine is called upon in the `apply diffusion` subroutine.
`update halo`	Updates the halo zones, often referred to as ghost cells, of the field by copying values from opposite edges to ensure correct boundary conditions during the stencil operation. This subroutine is called upon in the `apply diffusion` subroutine.
`init`	Initializes the program by setting up the MPI environment and reading the command line arguments.
`setup`	Prepares the fields and allocates memory before the main computation. Initializes the input field with specific values.
`cleanup`	Deallocates the memory used by the fields after the main computation is done.
`finalize`	Finalizes the program by cleaning up the MPI environment.

Table 1: Description of the subroutines.

1.1 Aim and Scope

The aim of this project is to provide insights into workflow of optimizing a 4th-order diffusion calculation using a stencil operation with the help of the two language based AIs, ChatGPT-4o mini, developed by OpenAI, and Gemini 1.5 Flash, developed by Google DeepMind. Theses two AI have been used in their free trial version. The goal is less to systematically compare both AIs quantitatively, but rather to provide two qualitative first impressions of their use cases and to explore which prompt engineering techniques lead to good optimization results. A further aspect that was investigated was how the AIs fare when provided with the exercise prompts from the tutorial, as this type of prompt is more detailed (often restricted to a specific subroutine instead of the whole code) than prompts bluntly asking them how to optimize the code in its entirety.

Of relevance for this project are mainly three subroutines that can be found in the appendix: `apply diffusion`, `laplacian` and `update halo`. These are the parts of the code that have optimization potential. Some brief explanation of the individual subroutines in the code can be read in Table 1.1. An example of the initial state of the simulation, as well as the resulting field after 1024 time steps of diffusion have been applied can be seen in Figure 1.

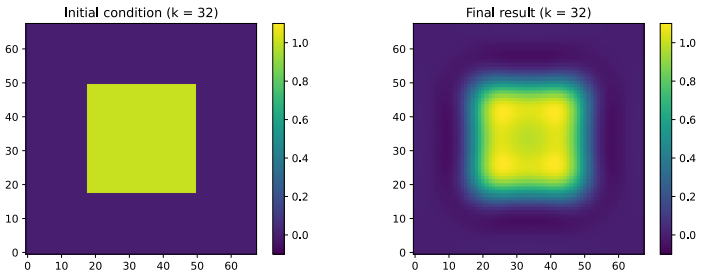

Figure 1: The diffusion process simulated with the stencil code.

2 Optimization approaches

In this section we will briefly outline the optimization approaches that we chose together with the AI to implement.

1. **Loop Fusion**
   Loop fusion is a compiler optimization technique that merges multiple nested loops into a single loop. This optimization can significantly improve program performance by reducing loop overhead, enhancing cache locality, and simplifying code.

2. **Loop Inlining**
   Inlining is a compiler optimization technique that replaces function calls with the actual code of the function. This can improve performance by reducing function call overhead, which includes saving and restoring registers, passing arguments, and returning values. However, inlining can also increase code size, which may have negative effects on performance in some cases.

3. **OpenMP Parallelization**
   OpenMP is an Application Programming Interface that allows programmers to write multi-threaded code for shared-memory parallel systems. By parallelizing computationally intensive sections of code, OpenMP can significantly reduce runtime on systems with multiple cores or processors. This is achieved by dividing the workload among multiple threads, allowing them to execute and potentially finish the task more quickly.

4. **Blocking**
   The data is divided into smaller, more manageable blocks that can fit into cache. By working on these smaller blocks, the algorithm can maximize data reuse within the cache, reducing the frequency of slower memory accesses. The effectiveness of blocking depends on the block size, which needs to be carefully chosen to match the architecture's cache characteristics.

5. **Memory Allocation Optimization**
   If possible, memory allocation should be done outside of frequently called routines, ideally during an initial setup phase. By allocating memory for temporary variables or buffers only once at the start, is avoided the need for repeated checks and allocations during subsequent calls. This approach minimizes overhead, streamlines memory management, and enhances overall program performance by ensuring that memory resources are efficiently utilized.

3 Prompt Engineering

The following is the typical interaction procedure with the AIs upon choosing an optimization technique. Deviations from this scheme usually occurred when trying to find new potential optimization techniques with help of the AI or implementing prompts directly from the tutorial.

1. **Prompt:**
   I'll provide you an original Fortran code that I'll want you to optimize. I want you to remember it and specifically the different subroutines where I'll want to perform different optimization techniques. **(*provide the whole code or specific subroutine to optimize*)**

   AI Assistant Response:
   I've processed the stencil2d code/the three Fortran subroutines: `apply_diffusion`, `laplacian`, and `update_halo`. I'll keep them in mind for future reference. If you have any more questions or tasks related to these code/subroutines, feel free to ask.

2. **Prompt:**
   I want to use **(*add optimization approaches*)** on the code and/or on this subroutine **(*add specific subroutine*)** can you give me an answer that include the whole subroutine so I can copy/paste in the code.

   AI Assistant Response:
   (*Provides optimized code/subroutine*)

3. **Prompt:**
   (If it's needed ask for correction/precision) The code you provided me is not working. **(*provide code*)** can you correct the possible issue

   AI Assistant Response:
   (*Provides correction for code/subroutine*)

4. **Prompt:**
   (If user wants to combine different technique) Here is the final code/subroutine **(*add code*)** that I already optimized. I now want to combine this with an other optimization approach **(*specify the approach*)**.

   AI Assistant Response:
   (*provides optimized code/subroutine*)

4 Results

Before going into the detail for a range of optimization techniques, this section will describe the general experience of prompt engineering, how it differed between both AIs as well as which approaches yielded helpful results.

A key difference between the two AIs that emerged is how they handle code in prompts. ChatGPT allows users to include entire blocks of code directly within the prompt. This feature is incredibly useful for providing context, testing, or troubleshooting, as users can easily reference the code they are working with. On the other hand, Gemini does not support the inclusion of large code blocks in the prompt, which can be a limitation when dealing with technical queries that require detailed examples. In general all the subroutines where given to Gemini individually.

When it comes to sourcing information, Gemini actively provides citations for the sources it references, often including direct links to the original content. ChatGPT, in contrast, does not typically cite sources directly, which can lead to questions about the origin and reliability of the information it generates.

Another issue with Gemini, was that it would often only include small parts of code instead of reproducing an entire functioning and running code, which would make for complication when inserting into the existing code. Further would Gemini often leave out small important signs, such as &, used for line breaks in Fortran, when pasting code, which would lead to errors, when trying to compile the optimized code.

Two main approaches at prompt engineering stood out as useful for ChatGPT. Either the whole stencil code or a specific subroutine was provided with the request to optimize it. The former approach was a good way to get a list of possible optimizations while optimizations that yielded good coding results were mostly achieved with the latter approach by asking the AI to apply a technique within one subroutine. A specific approach that was also briefly explored with both AIs was including a profiler report in the prompts in an attempt to give more context for optimization. In the case of ChatGPT the proposed optimization resulting from the prompt did not yield faster results than the original code version. For Gemini, the combination of code and profiling output appeared to confuse the AI and it was not able to produce useful output.

A general difficulty that was encountered is that both AIs seem to rather have a general understanding of the code and do not fully comprehend all the logical links. It was almost impossible to receive the entire updated and optimized code back as whole in a compilable form. Some essential logical links were usually lost in the process and snippets crucial to compilation were frequently omitted. Therefore most functioning improvements were constrained to one or two subroutines that were optimized internally with little changes needed in the rest of the code as a result of the optimization.

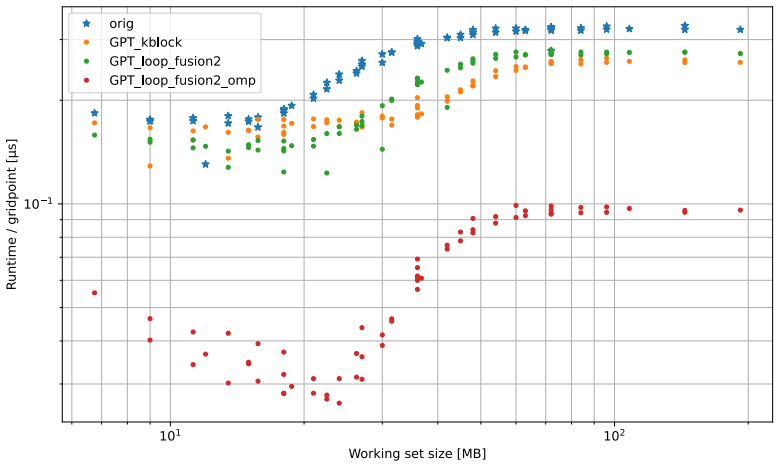

Figure 2: Optimization results using ChatGPT with orig referring to the original stencil computation code as benchmark.

4.1 K-Blocking Optimization

The blocking optimization of the ChatGPT and Gemini outputs has one of the faster runtime per grid point over all the working set. The way the AIs applied the technique is so : the original code utilized 3D arrays (`tmp1_field(:, :, :)`) to manage data, in the optimized version, these arrays have been reduced to 2D (`tmp1_field(:, :)`, `tmp2_field(:, :)`). This transition from 3D to 2D arrays leads to a reduction in size of the processed data blocks.

Moreover, the loop structure in the optimized code has been restructured to operate on these smaller blocks or tiles. The nested loops are now designed to iterate over specific subsets of the grid, corresponding to the smaller blocks. This restructuring enhances cache reuse by ensuring that the data within these blocks remains in the cache long enough to be fully processed.

Additionally, the removal of the `save` attribute in the optimized code suggests that persistent data storage across subroutine calls is no longer necessary. This change is due to the fact that each block of data is now fully processed within a single subroutine call before moving on to the next block.

ChatGPT made this change in the subroutine `apply diffusion` (A.6) and in the subroutine `laplacian` (A.3)

Gemini chose to only apply k-blocking to the `apply diffusion` subroutine, but applied inlining at the same time.

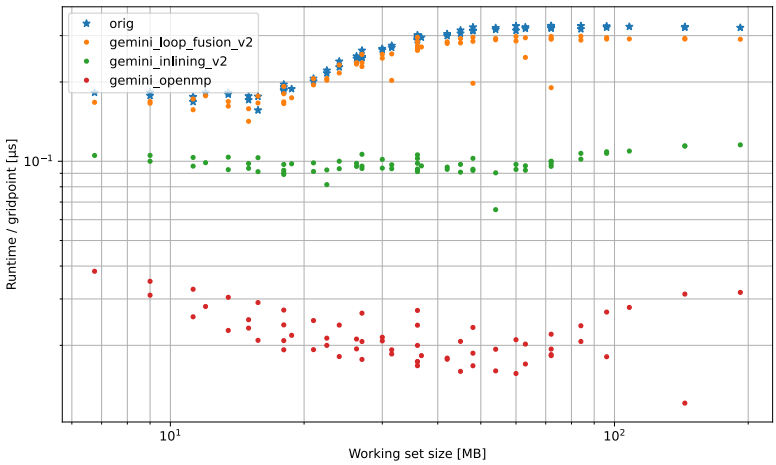

Figure 3: Optimization results using Gemini with orig referring to the original stencil computation code as benchmark.

4.2 Memory allocation

This optimization attempt (not plotted because it did not compile) showcases a situation that can happen where step 3 from the prompt engineering procedure is repeated several times but the code still does not compile. With time the code only gets further away from what its initial purpose was. Overall the changes are situated in the field allocation and initialization, the field update in `apply diffusion` when the code calls the `update halo` subroutine.

4.3 Loop Fusion

In the `apply diffusion` subroutine a loop fusion has been made to the `laplacian` computation. Originally, the computation was divided into two nested loops that ChatGPT and Gemini both easily and correctly fused. The detail are shown in the appendix (A.5). Here ChatGPT made the loop fusion efficiently and the result are what we expected from this optimization technique (faster computation).

4.4 Inlining

The inlining optimization for the Gemini output represents a significant speedup from the original computation time. The original code has a dedicated `laplacian` subroutine that is called within the `apply diffusion` routine. In the optimized

code, the Laplacian computation is performed directly (inlined) within the subroutine `apply diffusion` .

As the dataset size increases, cache misses and memory bandwidth become critical bottlenecks. Hence with large working sets, the time saved by this optimization becomes slightly smaller per working point.

4.5 OpenMP Parallelization

For OpenMP we ran the code on varying numbers of threads and compared the execution times. The results can be seen in Figure 4. What we see is that after around 12 threads the execution times start to increase again. This is due to overhead. Managing a large number of threads increases overhead as creating, scheduling, and synchronizing these threads is time consuming. As the number of threads increases, the overhead eventually will outweigh the benefits of parallelism, especially if the task size per thread becomes very small. Due to this we chose to run our OpenMP on 12 threads. The AIs correctly realized when to use private and shared variables.

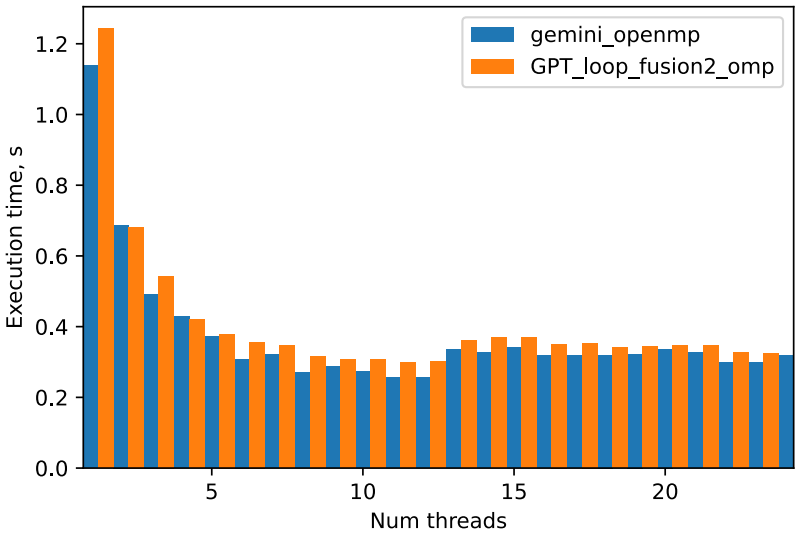

Figure 4: Execution times vs numbers of threads that the code was run on.

4.6 Combining Loop Fusion with Parallelization

The OpenMP and the loop fusion technique has been combined in "`GPT_loop_fusion2_omp`. In this case the code runs the fastest. This code was generated by providing the

AIs with the loop fusion code and asking it to further optimize it by parallelizing with OpenMP.

4.7 Quantitative Discussion

In Figure 3, we can nicely see how efficient the different optimizations are. It becomes clear that the loop and inlining optimizations do decrease runtime but the effect is most efficient for the OpenMP optimization.

When the working set size is small enough to fit within the CPU caches (L1, L2, or L3), the processor can access data quickly, which in turn leads to a reduced run time per data point. The optimizations such as loop fusion and inlining can help by reducing the overhead and improving cache locality, allowing more data to stay in the cache longer, thereby minimizing memory latency. We see that the loop fusion starts becoming less efficient at the same time as the original code, as we still loop over all k slices. In the inlining optimization we have changed our tmp_field to be 2-dim as, we got rid of our k dependency. This results in less data cycles between CPU cache and RAM, therefore moving the inefficiency bump to the right. The OpenMP optimization is utilizing more threads, thus the average runtime per point is significantly reduced. However, once we are moving above the L3 cache threshold, the improvement is not as significant, due to threads competing for space in L3 cache.

The AI's optimization codes have to be taken carefully specially concerning boundary conditions. This is happening when we try to make temporary field 2 dimensional. The first iteration over i and j has to go over smaller bounds, and LLMs oversee it. One has to pay attention because theses changes are hardly visible. They can still be seen by comparing the expected result from the original stencil code with the optimized version (figure 5). The change are visible in different subroutine optimized by AI (A.7)

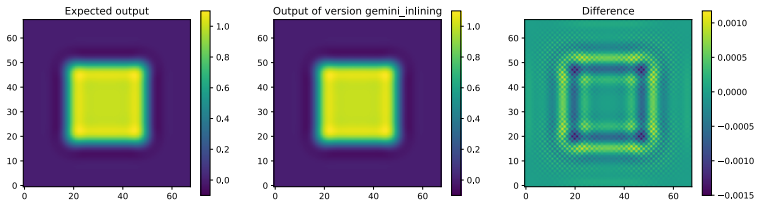

Figure 5: Comparaison original code with optimized one, highlighting boundary condition changes .

4.8 Qualitative Discussion

In this section we want to further discuss the pros & cons of replacing ourselves as performance engineers by AI.

Using AI assistants for optimizing the code is overall a powerful tool with potential. An example of this is that the AIs were capable of combining loop fusion with OpenMP parallelization correctly. On the other hand, only relying on the AI sometimes leads to endless optimization loops (e.g. in the memory allocation example). It also has to be mentioned that optimizations in a single subroutine worked significantly better than trying to change several parts of the code simultaneously which often yielded code that did not compile. For those reasons one needs to have some knowledge on code optimization to overcome some issues/errors commonly faced, critically interpret output and guide the AI in the right direction accordingly.

Only using the free version one can encounter some limitations. Having a paid version gives access to certain features not available on the free version, such as the ability to send code file directly without having to copy and paste it into the prompt, having certain pre-prompts included, or having faster, more stable responses.

Another potential annoyance is Gemini's tendency to "forget" code. Instead of having a long chat where one can point back to the same piece of code, it was required to copy paste the working routine roughly every 3 promts. This, combined with the lack of special formatting for code in promts, resulted in conversations that are very hard to trace back. Also, Gemini would very often replace parts of provided code with placeholders, making straight-forward copy-paste more annoying than it should be. This issue was also occasionally encountered with ChatGPT.

Furthermore, we tried to implement other optimizations not mentioned in this report before such as OpenACC, but ultimately failed. This can probably be lead back to the fact, that Fortran itself and OpenACC in particular doesn't have similar online community support, compared with languages like Python or C. This results in a less "creative" LLM, which results in it guessing the answers, rather than knowing them. This especially can be seen when debugging Cray compiler options and flags.

5 Conclusion

In conclusion, this project has demonstrated the potential of using AI language models, specifically ChatGPT and Gemini, to optimize the runtime of the stencil2d Fortran code used in weather and climate models. By applying various optimization techniques such as loop fusion, inlining, OpenMP parallelization, blocking, the AI-optimized code showed promising results in reducing computational time. However, the study also highlighted the limitations of current AI models in handling complex code optimizations.

The AI tool needs guidance to have meaningful result (i.e giving the opti-

mization technique and the subroutine where to apply the changes). AI models occasionally struggled with maintaining the logical integrity of the code, leading to non-functional outputs. Combining AI-driven optimizations with human expertise allowed for successful implementation of techniques like loop fusion combined with OpenMP parallelization.

A Fortran Subroutines

A.1 Subroutine: apply_diffusion

```
subroutine apply_diffusion( in_field, out_field, alpha, num_iter )
    implicit none

    ! arguments
    real (kind=wp), intent(inout) :: in_field(:, :, :)
    real (kind=wp), intent(inout) :: out_field(:, :, :)
    real (kind=wp), intent(in) :: alpha
    integer, intent(in) :: num_iter

    ! local
    real (kind=wp), save, allocatable :: tmp1_field(:, :, :)
    real (kind=wp), save, allocatable :: tmp2_field(:, :, :)
    integer :: iter, i, j, k

    ! this is only done the first time this subroutine is called (warmup)
    ! or when the dimensions of the fields change
    if ( allocated(tmp1_field) .and. &
        any( shape(tmp1_field) /= (/nx + 2 * num_halo, ny + 2 * num_halo, nz /) ) ) then
        deallocate( tmp1_field, tmp2_field )
    end if
    if ( .not. allocated(tmp1_field) ) then
        allocate( tmp1_field(nx + 2 * num_halo, ny + 2 * num_halo, nz) )
        allocate( tmp2_field(nx + 2 * num_halo, ny + 2 * num_halo, nz) )
        tmp1_field = 0.0_wp
        tmp2_field = 0.0_wp
    end if

    do iter = 1, num_iter

        call update_halo( in_field )

        call laplacian( in_field, tmp1_field, num_halo, extend=1 )
        call laplacian( tmp1_field, tmp2_field, num_halo, extend=0 )

        ! do forward in time step
        do k = 1, nz
        do j = 1 + num_halo, ny + num_halo
        do i = 1 + num_halo, nx + num_halo
```

```
39              out_field(i, j, k) = in_field(i, j, k) - alpha
                    * tmp2_field(i, j, k)
40          end do
41          end do
42          end do
43
44          ! copy out to in in case this is not the last
                iteration
45          if ( iter /= num_iter ) then
46              do k = 1, nz
47              do j = 1 + num_halo, ny + num_halo
48              do i = 1 + num_halo, nx + num_halo
49                  in_field(i, j, k) = out_field(i, j, k)
50              end do
51              end do
52              end do
53          end if
54
55      end do
56
57      call update_halo( out_field )
58
59  end subroutine apply_diffusion
```

Listing 1: Subroutine to Apply Diffusion

A.2 Subroutine: update_halo

```
1   subroutine update_halo( field )
2       implicit none
3
4       ! argument
5       real (kind=wp), intent(inout) :: field(:, :, :)
6
7       ! local
8       integer :: i, j, k
9
10      ! bottom edge (without corners)
11      do k = 1, nz
12      do j = 1, num_halo
13      do i = 1 + num_halo, nx + num_halo
14          field(i, j, k) = field(i, j + ny, k)
15      end do
16      end do
17      end do
18
19      ! top edge (without corners)
20      do k = 1, nz
21      do j = ny + num_halo + 1, ny + 2 * num_halo
```

```
    do i = 1 + num_halo, nx + num_halo
        field(i, j, k) = field(i, j - ny, k)
    end do
    end do
    end do

    ! left edge (including corners)
    do k = 1, nz
    do j = 1, ny + 2 * num_halo
    do i = 1, num_halo
        field(i, j, k) = field(i + nx, j, k)
    end do
    end do
    end do

    ! right edge (including corners)
    do k = 1, nz
    do j = 1, ny + 2 * num_halo
    do i = nx + num_halo + 1, nx + 2 * num_halo
        field(i, j, k) = field(i - nx, j, k)
    end do
    end do
    end do

end subroutine update_halo
```

Listing 2: Subroutine to Update Halo

A.3 Subroutine: `laplacian`

```
subroutine laplacian( field, lap, num_halo, extend )
    implicit none

    ! argument
    real (kind=wp), intent(in) :: field(:, :, :)
    real (kind=wp), intent(inout) :: lap(:, :, :)
    integer, intent(in) :: num_halo, extend

    ! local
    integer :: i, j, k

    do k = 1, nz
    do j = 1 + num_halo - extend, ny + num_halo + extend
    do i = 1 + num_halo - extend, nx + num_halo + extend
        lap(i, j, k) = -4._wp * field(i, j, k)      &
            + field(i - 1, j, k) + field(i + 1, j, k)  &
            + field(i, j - 1, k) + field(i, j + 1, k)
    end do
    end do
```

```
    end do

end subroutine laplacian
```

Listing 3: Subroutine to Compute Laplacian

A.4 Inlining optimized by Gemini

```
subroutine apply_diffusion( in_field, out_field, alpha,
    num_iter )
    implicit none

    ! arguments
    real (kind=wp), intent(inout) :: in_field(:, :, :)
    real (kind=wp), intent(inout) :: out_field(:, :, :)
    real (kind=wp), intent(in) :: alpha
    integer, intent(in) :: num_iter

    integer :: iter, i, j, k

    real (kind=wp), save, allocatable :: tmp_field(:, :)
    real (kind=wp) :: laplap

    if ( allocated(tmp_field) .and. &
        any( shape(tmp_field) /= (/nx + 2 * num_halo, ny +
            2 * num_halo/) ) ) then
        deallocate( tmp_field )
    end if

    if ( .not. allocated(tmp_field) ) then
        allocate( tmp_field(nx + 2 * num_halo, ny + 2 *
            num_halo) )
        tmp_field = 0.0_wp
    end if

    do iter = 1, num_iter
        call update_halo(in_field)

        do k = 1, nz
            do j = 1 + num_halo, ny + num_halo
                do i = 1 + num_halo, nx + num_halo
                    tmp_field(i, j) = -4._wp * in_field(i,
                        j, k) &
                                    + in_field(i - 1, j, k)
                                        + in_field(i + 1, j,
                                        k) &
                                    + in_field(i, j - 1, k)
                                        + in_field(i, j + 1,
                                        k)
```

```
                end do
            end do

            do j = 1 + num_halo, ny + num_halo
                do i = 1 + num_halo, nx + num_halo
                    laplap = -4._wp * tmp_field(i, j) &
                            + tmp_field(i - 1, j) +
                                tmp_field(i + 1, j) &
                            + tmp_field(i, j - 1) +
                                tmp_field(i, j + 1)
                    out_field(i, j, k) = in_field(i, j, k)
                        - alpha * laplap
                end do
            end do
        end do

        if (iter /= num_iter) then
            do k = 1, nz
                do j = 1 + num_halo, ny + num_halo
                    do i = 1 + num_halo, nx + num_halo
                        in_field(i, j, k) = out_field(i, j,
                            k)
                    end do
                end do
            end do
        end if
    end do

    out_field(:,:,:) = in_field(:,:,:)

end subroutine apply_diffusion
```

Listing 4: Inlining applied to apply_diffusion by Gemini

A.5 Loop fusion applied to apply_diffusion by Gemini

```
do k = 1, nz
    do j = 1 + num_halo, ny + num_halo
        do i = 1 + num_halo, nx + num_halo
            ! Compute Laplacian and update in a single pass
            tmp1_field(i, j, k) = -4._wp * in_field(i, j,
                k) &
                + in_field(i - 1, j, k) + in_field(i + 1,
                    j, k) &
                + in_field(i, j - 1, k) + in_field(i, j +
                    1, k)
            out_field(i, j, k) = in_field(i, j, k) - alpha
                * (-4._wp * tmp1_field(i, j, k) &
```

```
                + tmp1_field(i - 1, j, k) + tmp1_field(i +
                    1, j, k) &
                + tmp1_field(i, j - 1, k) + tmp1_field(i, j
                    + 1, k))
        end do
    end do
end do
```

Listing 5: Loop Fusion applied by ChatGPT

A.6 OpenMP optimized by Gemini

```
subroutine apply_diffusion( in_field, out_field, alpha,
    num_iter )
    implicit none

    ! arguments
    real (kind=wp), intent(inout) :: in_field(:, :, :)
    real (kind=wp), intent(inout) :: out_field(:, :, :)
    real (kind=wp), intent(in) :: alpha
    integer, intent(in) :: num_iter

    ! local
    real (kind=wp), save, allocatable :: tmp1_field(:, :)
    real (kind=wp) :: laplap
    integer :: iter, i, j, k

    ! this is only done the first time this subroutine is
        called (warmup)
    ! or when the dimensions of the fields change
    if ( allocated(tmp1_field) .and. &
        any( shape(tmp1_field) /= (/nx + 2 * num_halo, ny +
            2 * num_halo/) ) ) then
        deallocate( tmp1_field )
    end if
    if ( .not. allocated(tmp1_field) ) then
        allocate( tmp1_field(nx + 2 * num_halo, ny + 2 *
            num_halo) )
        tmp1_field = 0.0_wp
    end if

    do iter = 1, num_iter

        call update_halo( in_field )
        !$omp parallel do default(none) private(tmp1_field,
            laplap) shared(nz, num_halo, ny, nx, in_field,
            num_iter, out_field, alpha, iter)
        do k = 1, nz
            do j = 1 + num_halo - 1, ny + num_halo + 1
```

```
32              do i = 1 + num_halo - 1, nx + num_halo + 1
33                  tmp1_field(i, j) = -4._wp * in_field(i, j,
                        k)        &
34                      + in_field(i - 1, j, k) + in_field(i +
                            1, j, k)  &
35                      + in_field(i, j - 1, k) + in_field(i, j
                            + 1, k)
36              end do
37              end do
38
39              do j = 1 + num_halo, ny + num_halo
40              do i = 1 + num_halo, nx + num_halo
41
42                  laplap = -4._wp * tmp1_field(i, j)       &
43                      + tmp1_field(i - 1, j) + tmp1_field(i +
                            1, j)  &
44                      + tmp1_field(i, j - 1) + tmp1_field(i,
                            j + 1)
45
46                  if ( iter == num_iter ) then
47                      out_field(i, j, k) = in_field(i, j, k)
                            - alpha * laplap
48                  else
49                      in_field(i, j, k)  = in_field(i, j, k)
                            - alpha * laplap
50                  end if
51
52              end do
53              end do
54
55          end do
56          !$omp end parallel do
57
58      end do
59
60  end subroutine apply_diffusion
```

Listing 6: OpenMP applied to apply_diffusion by Gemini

A.7 Change in boundary condition made by the AI

```
1
2
3   ! Original code
4   do j = 1 + num_halo - 1, ny + num_halo + 1
5       do i = 1 + num_halo - 1, nx + num_halo + 1
6           tmp_field(i, j) = -4._wp * in_field(i, j, k) &
7                           + in_field(i - 1, j, k) +
                                in_field(i + 1, j, k)
```

```
8
9  ! AI optimized code
10 do j = 1 + num_halo, ny + num_halo
11     do i = 1 + num_halo, nx + num_halo
12         tmp_field(i, j) = -4._wp * in_field(i, j, k) &
13                           + in_field(i - 1, j, k) +
                                in_field(i + 1, j, k)
```

Listing 7: Change in boundary condition made by AI in different subroutine