A Case Study: Using a Conversational LLM to Build a High Performance Physics Engine for Gas Diffusion*

Andrew J. Pounds
Departments of Chemistry and Computer Science
Mercer University, Macon, GA, 31207
pounds_aj@mercer.edu

Abstract

Herein is described an initial set of tests to utilize a generative artificial intelligence tool to build high performance code to simulate the physical process of gas diffusion. This was done to see how a large language model (LLM) would perform in producing code to accurately describe the diffusion process, and also see how it would respond to requests to optimize the code. Optimizations were attempted for single-threaded code, shared memory symmetric multiprocessing code using *OpenMP*, and for GPUs using *OpenACC*. Superfluous data was also provided to the LLM to see the effects on the code generation process. Explicit prompts are shown, the characteristics of the generated code are described, and benchmarks are tabulated. The tests used to verify the correctness of the results are also described.

1 Introduction

The last few years have seen the use of artificial intelligence (AI) greatly expand across every area of human endeavor.[1] One area where this is particularly the case is in software development.[2] The tools for AI software development have advanced to the point that some institutions have chosen to limit AI use or

*Copyright ©2025 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

even return to handwritten code assignments.[3] An alternative approach is to embrace AI and yoke its strengths in the development process. This strategy can be particularly beneficial when developing portions of code that rely on discipline specific content knowledge, assumptions, and coding strategies that are unknown to the programmer.

One of the common tasks in developing realistic 3D computer games is the development of the physics engine. The physics engines are generally numerically intensive section of code built on principles unknown to the programmer and that require significant optimization to limit the effects of the engine on game play. While Large Language Model generative AI (LLMs) are somewhat effective at optimizing high performance computing (HPC) code [4] others have found the results lacking to the point that new LLMs needed to be developed for this specific purpose.[5]. Similarly, LLM systems have been applied to the code-building process of physics engines [6], and some outstanding libraries and tools can handle these sorts of tasks.[7] Because the physics engines need to be both accurate and fast, and since not all games are necessarily written using the same languages, libraries, or tools, there are proposed methods that "bridge the gap"[8]. However, as is often the case, the person building the physics engine often finds themselves in the position of writing some or all the code *a priori*. If the programmer does not understand the underlying physics and math, then this can be a daunting task, and precisely Where AI may be of some help. This, however, can also become an issue because the AI engine may solve the problem in a way that is optimal, but does not result in a physically correct solution. As has been noted by others, the workflow of AI-based coding for physics involves not only prompting and coding, but also extensive testing and verification.[9]

While instructors are generally being encouraged to adopt generative AI in their courses, there are still concerns about how it could or should be implemented. This paper provides a concrete example, using the physical process of gas diffusion, to show how current freely available generative AI tools and compilers can be used, through trial and error, to build HPC code that gives accurate results using modern, commonly available hardware. As such, we demonstrate a desired workflow to use LLMs to develop physically correct physics engines for computer games in a general manner.

2 Environment

The Google *Gemini* (ver. 2.5)[10] LLM conversational engine was used to produce all of the AI-generated code in this paper. For the sake of comparison with existing code, the programs were generated in modern Fortran and then slightly modified to ensure that the same type of problem was being solved across all the programs for benchmarking purposes. All code was compiled with the freely available Nvidia HPC SDK (ver. 25.3) using -O2 optimization and run on a 3.6 GHz Intel i7-7700 CPU with 48 GB of DDR 4 memory. Any GPU-accelerated code was run on a Nvidia RTX 4060 Ti graphics card with 8 GB of memory using driver version 575.51.03 with CUDA version 12.9.

3 Methods

Fick's first law of diffusion states that particles move from regions of higher concentration to those of lower concentration. The change in the concentration between the regions of space occurs according to Fick's second law that is shown in Eqn. 1

$$\frac{dC}{dt} = D\left(\frac{\partial^2 C}{\partial x^2} + \frac{\partial^2 C}{\partial y^2} + \frac{\partial^2 C}{\partial z^2}\right) \tag{1}$$

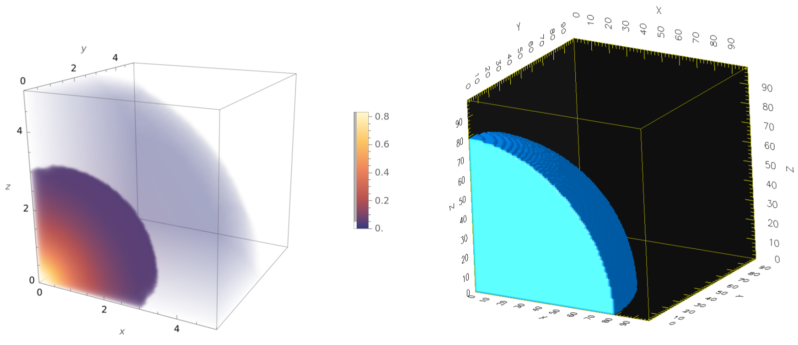
where C is the concentration of the diffusing species and D is the coefficient of diffusion. In this paper it is assumed that D is isotropic across the solution space. A common way to solve the diffusion problem is to take a finite element approach and break the space up into several equal volume cubes [11] and then propagate the concentration of the species as a function of time via Eqn. 1.

In all cases, *Gemini* was asked to construct code that solved the problem for a room that was 125 cubic meters in volume (5 meters per side). *Gemini* initially proposed a finite element approach using 1 cubic meter volume elements that would run for a specified amount of simulated time (1 hour). To better simulate HPC codes, the memory requirements and run times were increased by reducing the volume elements to 0.125 L. This resulted in one million finite volume elements using 64-bit precision.

To test the ability of the code to produce physically realistic results, the code was modified to run until the room was completely equilibrated. At this equilibration point, the concentration values of all the volume elements must be within 1% of one another. By simply comparing a selection of volume elements from the final equilibrated room to those of our stock diffusion code, it was easy to verify the correctness of the results. *Mathematica*[12] was also used to solve Eqn. 1 with zero flux allowed at the room boundaries. A timeslice from these calculations is show in Figure 1a. Using *Mathematica* it was determined that the room should completely equilibrate in approximately 92 hours.

To see what assumptions the conversation LLM would make, no statements were initially given to *Gemini* about room mass conservation, flux across the boundary, or Neumann boundary conditions.

Finally, a misleading assumption was given to *Gemini*; the average speed of the diffusing gas molecule was provided. It is often thought that gas diffusion, and thus the coefficient of diffusion (D), depends on the speed of the gas molecules. Diffusion is a temperature and pressure dependent transport phenomenon that results from the collisions of molecules with one another.[13] The *relative velocity*, not *average velocity*, of the molecules is a function of the *mean free path*, (λ) which is a function of their *collision cross section* (σ). The collision cross section is related to the *size* and *shape* of the molecule – so it is correlated to the mass of the molecule, but also depends on other factors. So while the average speed of a gas molecule is related to the temperature and mass of the molecule, the diffusion coefficient is dependent on several other transport properties. None of this background or additional data was provided to *Gemini* to see what it would do with the superfluous gas speed information.



(a) *Mathematica* Timeslice (b) Finite Element Timeslice

Figure 1: 3D plots showing the evolution of the diffusion process at an early, but not identical, timestep. In the *Mathematica* plot the axes denote the room size of 5 cubic meters. The plot on the left was constructed with the data-flow program *OpenDX* using data produced from the finite element programs. The axes denote the indices of the finite elements in the room. At t=0 the initial concentration was 1.0 in element [0,0,0]

4 Results

Several prompts were conversationally provided to *Gemini*. The results were generally a few pages describing what assumptions and techniques were going to be used for code generation followed by a working program, and instructions on how to compile and run the program.

4.1 General Observations

With respect to code generation the LLM avoided many of the array access issues common in these types of programs. For example, all of the concentration updates to the arrays involved nested loops. In many textbook and online examples these updates are done using the same array. The updates produced by *Gemini* were done by reading from the array holding the current concentrations and writing the updated concentrations to another array and then, after the update is complete, the updated array is copied to the original array. This is a common technique in HPC work because it allows for cache optimization, vectorization, and parallelization in the compiler; students often do not think to do this because it requires more memory and supposedly superfluous code.

All of the code generated by *Gemini* contained parameters so that the code would run for one hour of simulated time. As such all of the code had to be modified so that it would run to meet the "equilibration condition" previously described.

4.2 Individual Prompts

What follows are the prompts provided to *Gemini* and a brief description of the textual responses and code produced. The code was also run and the simulated time and run time tabulated.

First prompt: Using modern Fortran, write code to use Fick's first and second laws to simulate gas diffusion in a room that is 5 meters long 5 meters high and five meters wide you may assume that the average speed of the gas molecules is 250 meters/sec

- *Gemini* first flagged the problem of using gas velocity in diffusion problems and proposed a value for the diffusion coefficient that was reasonable based on literature values (5x10^{-5} m^2/s)
- An initial value for the concentration in the room was needed. *Gemini* proposed an initial value at one corner of the room.
- There were no given boundary conditions, *Gemini* assumed that there was no-flux and that particles could not pass through the wall
- A forward-Euler finite element element numerical method approach was adopted and the room discretized into a 3D grid and the process stepped through time. For the sake of stability, *Gemini* set the step size so that $\Delta t \leq \frac{(\Delta x)^2}{2D}$
- *Gemini* claimed it was using Neumann conditions to conserve mass. *The solution did not conserve mass.*

- Generated code used row-major order multidimensional array indexing (Fortran should be column major for performance)
- Simulated equilibration time of 97.7 hours, actual run time was 126 seconds

Second prompt: Repeat the previous task but switch to the Jacobi tiled algorithm to improve cache performance

- *Gemini* applauded the idea of moving to a tiled Jacobi algorithm[14] to improve performance.
- *Gemini* switched to the Backward-Euler approximation for implementation into the Jacobi loops. It correctly constructed the three outer tiling loops and the inner Jacobi loops.
- The generated code switched to column-major indexing for Fortran performance
- *Gemini* proposed a tilesize of 8, but stated that this must be optimized through testing. How tiling improves performance was also described.
- *Gemini* warned that the value of the timestep might need to be hand-adjusted to give correct values for the simulation time
- *Gemini* suggested switching to *OpenMP*[15] parallelism for greater performance
- Mass was, to a reasonable approximation, conserved.
- Simulated equilibration time was 291 hours, actual run time was 362 seconds

Third prompt: In the code you produced in the first prompt the room loses mass because the sum of all the finite elements at the end of the run did not equal the sum of all the finite elements at the beginning of the run. Can you generate a logical mask to create code that conserves mass?

- Logical masks are oftentimes used in diffusion code to specify regions of space that are not accessible to the gas.

  Gemini indicated that a more robust procedure was needed to account for the conservation laws. To enforce zero concentration gradient at the boundary a stencil based approach was adopted so that the value of the concentration at the boundary was set equal to its nearest neighbor inside the boundary at the end of each calculation. How this was implemented is shown in Listing 1 and how these boundary conditions were implemented in the update equations are show in Listing 2.

Listing 1: Implementation of a region computation mask to handle barriers and impenetrable regions with a Mask equal to 1. This code was replicated for the Y and Z dimensions.

```
! X-direction neighbors
IF (i == 1 .OR. Mask(i-1,j,k) == 1) THEN
    C_val_im1 = Concentration_old(i,j,k)
    ! No flux from left (domain boundary or mask)
ELSE
    C_val_im1 = Concentration_old(i-1,j,k)
END IF

IF (i == NX .OR. Mask(i+1,j,k) == 1) THEN
    C_val_ip1 = Concentration_old(i,j,k)
    ! No flux from right (domain boundary or mask)
ELSE
    C_val_ip1 = Concentration_old(i+1,j,k)
END IF
```

Listing 2: Demonstrating how the computational mask was used in the stencil calculation.

```
! Apply finite difference stencil for diffusion to
! this OPEN cell
Concentration_new(i, j, k) = Concentration_old(i, j, k) + &
    D * DT * ( &
    (C_val_ip1 - 2.0_8*Concentration_old(i, j, k)
            + C_val_im1) / DX**2 + &
    (C_val_jp1 - 2.0_8*Concentration_old(i, j, k)
            + C_val_jm1) / DY**2 + &
    (C_val_kp1 - 2.0_8*Concentration_old(i, j, k)
            + C_val_km1) / DZ**2 )
```

- The generated code switched back to row-major array indexing
- Mass was conserved.
- Simulated equilibration time was 101 hours, actual run time was 150 seconds

Fourth prompt: Repeat what you did in the previous prompt but now switch to OpenMP parallelism to enhance the performance.

- *Gemini* correctly placed the `!$OMP PARALLEL DO COLLAPSE(3)` constructs at the proper locations and also added the needed directives to indicate which variables were shared and private to each thread.
- Inside the code *Gemini* include comments describing all of the OpenMP directives
- The generated code returned to proper column-major array indexing for Fortran
- Mass was conserved.
- Simulated equilibration time was 101 hours, actual run time was 168 seconds using the optimal recommended number of threads (4) for the i7 processor to minimize cache conflicts

Fifth prompt: Can you now replace the OpenMP code with OpenACC code so those pieces can be offloaded to a GPU to enhance the performance?

- *Gemini* used the `!$ACC DATA COPY` construct from *OpenACC*[16] to move the arrays to the GPU and then did all of the equilibration processing on the GPU
- Similarly to the generated *OpenMP* code, *Gemini* collapsed the loops, `!$ACC PARALLEL LOOP COLLAPSE(3)` and included to instructions to tell the GPU that the needed arrays were already in GPU memory
- The `COLLAPSE(3)` also instructs the GPU that it can optimally rearrange the array indexing for its memory architecture
- Inside the code *Gemini* included comments describing all of the *OpenACC* directives
- Mass was conserved.
- Simulated equilibration time was 101 hours, actual run time was 15 seconds

4.3 Evaluation

The correctness of the computed results was based on two things: the ability of the code to conserve mass and to produce a realistic simulated time. The value of the simulated time computed with *Mathematica* (approximately 92 hours) was taken as the correct time and a lower limit. Referring to Table 1, only the last three prompts reasonably met these criteria. It should be noted that *Gemini* did warn that the simulated times produced from the code of prompt 2 would probably be incorrect and the tile sizes needed to be optimized to improve the execution times.

Table 1: Summary of Results for Code Produced by *Gemini*

Prompt	Finite Element Method	Parallel Method	Mass Conserved	Simulated Time (Hours)	Run Time (Seconds)
1	Forward Euler	N/A	N	98	126
2	Tiled Jacobi	N/A	Y	291	362
3	Stencil	N/A	Y	101	150
4	Stencil	*OpenMP*	Y	101	168
5	Stencil	*OpenACC*	Y	101	15

Considering the last three "correct" prompts, the benchmarking results show one of the weaknesses of LLM HPC code. Prompt 3 produced a serial code. Prompt 4 asked the LLM to implement *OpenMP* parallelism on four threads. It was verified that the code was running on four threads concurrently; if there was sufficient work to do in the triply nested loops, then the run time should have dropped significantly compared to the serial time. However, since this was not the case the overhead of setting up the parallel region during each update pass must have increased the overall runtime. The LLM had no way to detect this increase in runtime and proposed no methods to guard against it. In the *OpenACC* code the entire update process, including the calculation of the variable used to check if equilibration had been achieved, was moved onto the GPU. This resulted in a ten-fold speedup over the single-threaded stencil code.

5 Discussion and Conclusions

One of the primary questions addressed in this paper centers on how well a conversation LLM can write accurate code for physical simulations. We have clearly shown that, given minimal, or even incorrect input that the *Gemini* LLM can, with a few exceptions, do this. As such, these AI systems should quickly find themselves as part of the arsenal of those building code for 3D simulations or computer games..

As stated, the *Gemini* LLM generally did a good job of writing code to solve the problem at hand, but there were a few issues when it came to generating code that produced physically correct results. Based on our tests, all of these could have been avoided with more specific prompting. Our tests using the LLM to generate HPC code are very tentative but do reveal a drawback. In progressing from a serial code to a shared memory parallel code with *OpenMP*, one must carefully examine the problem at hand to determine if there is enough computational work to benefit from this modification. Because *OpenMP* does incur some overhead to set up the loops in these calculations, one must often test extensively and decide limits for when to switch from single-threaded to multi-threaded execution. Surprisingly the *Gemini* LLM recognized this issue when asked to build single-threaded code with tiles to optimize cache, but ignored it when building *OpenMP* solutions. In our opinion there is a lot of room for improvement in LLM generated *OpenMP* code. In these experiments, the LLM did an outstanding job of adding the appropriate keywords for *OpenACC* GPU acceleration. It was also verified while the code was running that the GPU was being used optimally. The primary difference between the location of the parallelization commands in the *OpenMP* and *OpenACC* codes was that in the *OpenACC* case all of the arrays and control variables were moved to the GPU and the entire calculation took place on the GPU while in the case of *OpenMP* the parallelization constructs had to be issued for each timestep. While initial testing has shown that other LLMs give similar solutions to *Gemini* for the problems described in this paper, space limitations prevent even a cursory discussion of those results here.

The disparities in runtimes between different coding strategies are issues that have to be dealt with routinely in high performance computing. The experiments in this research show that the LLM can easily generate working code using any number of HPC techniques: tiling, stencils, shared memory parallelization, GPU acceleration, etc. Thus for those teaching HPC courses the LLM could produce a multitude of examples to demonstrate how to build HPC programs and thus provide starting points upon which students could improve. With the wide availability of conversational LLMs it is hoped that the examples provided herein entice others to incorporate AI into their HPC workflows and encourage their students to do the same.

References

[1] Adib Bin Rashid and MD Ashfakul Karim Kausik. "AI revolutionizing industries worldwide: A comprehensive overview of its diverse applica-

tions". In: *Hybrid Advances* 7 (2024), p. 100277. ISSN: 2773-207X. DOI: https://doi.org/10.1016/j.hybadv.2024.100277. URL: https://www.sciencedirect.com/science/article/pii/S2773207X24001386.

[2] Pisut Manorat, Suppawong Tuarob, and Siripen Pongpaichet. "Artificial intelligence in computer programming education: A systematic literature review". In: *Computers and Education: Artificial Intelligence* 8 (2025), p. 100403. ISSN: 2666-920X. DOI: https://doi.org/10.1016/j.caeai.2025.100403. URL: https://www.sciencedirect.com/science/article/pii/S2666920X25000438.

[3] Sam Lau and Philip Guo. "From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot". In: *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1.* ICER '23. Chicago, IL, USA: Association for Computing Machinery, 2023, pp. 106–121. ISBN: 9781450399760. DOI: 10.1145/3568813.3600138. URL: https://doi.org/10.1145/3568813.3600138.

[4] Marko Mišić and Matija Dodović. "An assessment of large language models for OpenMP-based code parallelization: a user perspective". In: *Journal of Big Data* 11.1 (Nov. 2024), p. 161. ISSN: 2196-1115. DOI: 10.1186/s40537-024-01019-z. URL: https://doi.org/10.1186/s40537-024-01019-z.

[5] Xianzhong Ding et al. "Hpc-gpt: Integrating large language model for high-performance computing". In: *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis.* 2023, pp. 951–960.

[6] Mohamad Ali-Dib and Kristen Menou. "Physics simulation capabilities of LLMs". In: *Physica Scripta* 99.11 (Oct. 2024), p. 116003. DOI: 10.1088/1402-4896/ad7a27. URL: https://dx.doi.org/10.1088/1402-4896/ad7a27.

[7] Michael Kaup et al. *A Review of Nine Physics Engines for Reinforcement Learning Research.* 2024. arXiv: 2407.08590 [cs.AI]. URL: https://arxiv.org/abs/2407.08590.

[8] Luiza Martins de Freitas Cintra et al. "An LLM-Enhanced Framework for Bridging Simulators and Game Engines towards Realistic 3D Simulations". In: *Proceedings of the 2025 ACM International Conference on Interactive Media Experiences.* 2025, pp. 402–405.

[9] Licheng Jiao et al. "AI meets physics: a comprehensive survey". In: *Artificial Intelligence Review* 57.9 (Aug. 2024), p. 256. ISSN: 1573-7462. DOI: 10.1007/s10462-024-10874-4. URL: https://doi.org/10.1007/s10462-024-10874-4.

[10] Google AI. *Gemini conversational AI model, Version 2.5 Flash.* Chat with Gemini. Accessed June 11, 2025. 2025. URL: https://gemini.google.com/.

[11] Christopher Batchelor-McAuley and Richard G. Compton. "Diffusion to a cube: A 3D implicit finite difference method". In: *Journal of Electroanalytical Chemistry* 877 (2020), p. 114607. ISSN: 1572-6657. DOI: https://doi.org/10.1016/j.jelechem.2020.114607. URL: https://www.sciencedirect.com/science/article/pii/S1572665720308353.

[12] Wolfram Research Inc. *Mathematica, Version 14.2.* Champaign, IL, 2024. URL: https://www.wolfram.com/mathematica.

[13] Ignacio Tinoco et al. *Physical Chemistry: Principles and Applications in Biological Sciences.* 5th. Pearson Education, 2014.

[14] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* 1st. CRC Press. DOI: https://doi.org/10.1201/EBK1439811924.

[15] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming.* MIT press, 2007.

[16] OpenACC. *OpenACC, Directives for Accelerators.* URL: http://www.openacc.org/.