# Toward Automatic HPC Code Optimization by Generative AI Based on LLMs

**Daichi Mukunoki[1], Shun-ichiro Hayashi[2], Koki Morita[2], Tetsuya Hoshino[1], Takahiro Katagiri[1]**
1 Information Technology Center, Nagoya University, 2: Graduate School of Informatics, Nagoya University
Contact: mukunoki@cc.nagoya-u.ac.jp

## Introduction

- The Information Technology Center at Nagoya University is conducting the HPC-GENIE project, which explores the use of generative AI for the automatic generation and optimization of HPC codes. (See separate poster for project overview.)
- This poster presents several ongoing research activities within the HPC-GENIE project


## BLAS code generation using general-purpose LLMs

- Evaluate the ability of general-purpose LLMs (OpenAI GPT-4.1 and o4-mini, April 2025) to generate C codes of BLAS routines
- Code correctness and numerical accuracy are validated across all parameter combinations (e.g., incx, trans, uplo, etc.)
- Working codes are generated from only the BLAS routine name, but their structure often differs from the Fortran reference implementations
- LLMs may rely on documentations and specifications available online, not only existing codes, leading to diverse coding styles and approaches
- Performance optimization remains a major challenge, especially for level-3 routines, which fall far short of OpenBLAS performance

- **NameToCcode:** Generate C code from routine name only
- **NameToOptCcode**: Generate optimized C code from routine name only
- **FrtcodeToOptCcode**: Generate optimized C code from reference Fortran code (including routine specifications)

*Note: due to randomness in LLM outputs, we report the count of correct results out of 10 attempts.

| Level | Routine | NameToCcode | | NameToOptCcode | | FrtcodeToOptCcode | |
|---|---|---|---|---|---|---|---|
| | | GPT-4.1 | o4-mini | GPT-4.1 | o4-mini | GPT-4.1 | o4-mini |
| 1 | dasum | 10 | 10 | 9 | 7 | 8 | 8 |
| | daxpy | 10 | 10 | 10 | 10 | 8 | 9 |
| | ddot | 10 | 10 | 8 | 9 | 6 | 10 |
| | idamax | 3 | 10 | 2 | 9 | 7 | 7 |
| | dnrm2 | 10 | 10 | 7 | 5 | 8 | 6 |
| | drot | 10 | 10 | 8 | 9 | 6 | 9 |
| | drotm | 8 | 5 | 3 | 6 | 8 | 8 |
| 2 | dgemv | 8 | 9 | 3 | 6 | 3 | 4 |
| | dger | 10 | 10 | 7 | 7 | 6 | 10 |
| | dsymv | 8 | 8 | 0 | 2 | 0 | 3 |
| | dsyr | 10 | 8 | 0 | 5 | 7 | 7 |
| | dsyr2 | 8 | 9 | 2 | 7 | 6 | 10 |
| | dtrmv | 3 | 4 | 0 | 5 | 2 | 1 |
| | dtrsv | 8 | 9 | 0 | 4 | 4 | 7 |
| 3 | dgemm | 10 | 10 | 3 | 0 | 5 | 7 |
| | dsymm | 4 | 8 | 3 | 3 | 1 | 4 |
| | dsyrk | 3 | 10 | 0 | 1 | 4 | 5 |
| | dsyr2k | 3 | 10 | 0 | 0 | 7 | 6 |
| | dtrmm | 0 | 1 | 0 | 0 | 1 | 0 |
| | dtrsm | 0 | 0 | 0 | 0 | 5 | 1 |

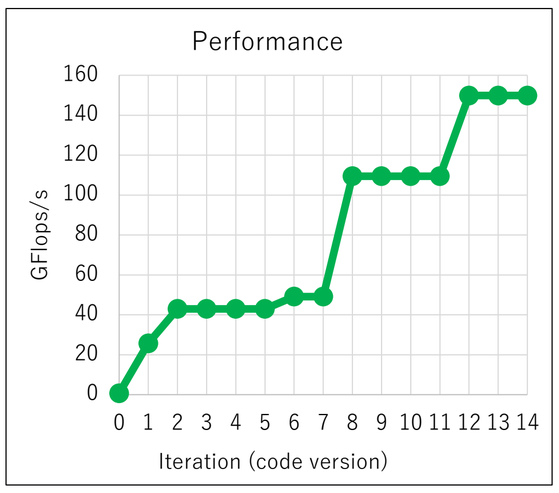
D. Mukunoki, S. Hayashi, T. Hoshino, T. Katagiri, “Performance Evaluation of General Purpose Large Language Models for Basic Linear Algebra Subprograms Code Generation”, arXiv:2507.04697, 2025.

## Proto-typing of multi-agent system for code optimization

- Developed a prototype multi-agent system for autonomous, non-stop code optimization as first step toward our own “Claude Code-like” solution
- Uses a local LLM (OpenAI’s gpt-oss-120b, equivalent to o4-mini)
- Runs on Ryzen AI Max+ 395 (128 GB), a ~$2,000 system
- Configured with 5 LLM agents: Project Manager (PM), 3 Programmers (PG), and Debugger
- Workflow: (1) PM assigns optimization tasks to PGs, (2) PGs implement code independently, (3) Debugger evaluates performance and detects errors, (4) Feedback is returned to PM - This loop continues until the time limit is reached

Input code
(naïve GEMM with triple loop)

```
void matmul_naive(int N, double *A, double *B, double *C) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int k = 0; k < N; k++) {
                sum += A[i * N + k] * B[k * N + j];
            }
            C[i * N + j] = sum;
        }
    }
}
```

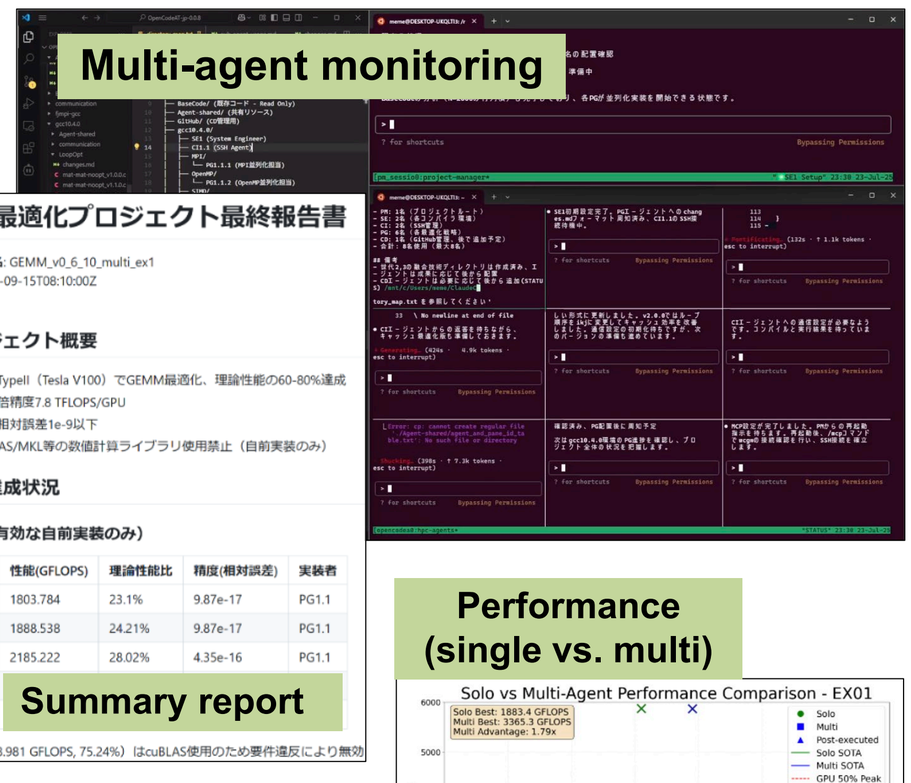



Optimized output code

```
void matmul_naive(int N, double *restrict A, double *restrict B, double *restrict C) {
    const int BS = 128;
    const int VEC = 4;
    A = (double *)__builtin_assume_aligned(A, 64);
    B = (double *)__builtin_assume_aligned(B, 64);
    C = (double *)__builtin_assume_aligned(C, 64);
    #pragma omp parallel
    {
        double *Bpack = (double *)_mm_malloc(BS * BS * sizeof(double), 64);
        double *Apack = (double *)_mm_malloc(BS * BS * sizeof(double), 64);
        Bpack = (double *)__builtin_assume_aligned(Bpack, 64);
        Apack = (double *)__builtin_assume_aligned(Apack, 64);
        #pragma omp for collapse(2) schedule(dynamic)
        for (int i0 = 0; i0 < N; i0 += BS) {
            for (int j0 = 0; j0 < N; j0 += BS) {
                int i_max = i0 + BS > N ? N : i0 + BS;
                int j_max = j0 + BS > N ? N : j0 + BS;
                for (int i = i0; i < i_max; ++i)
                    for (int j = j0; j < j_max; ++j)
                        C[i * N + j] = 0.0;
                for (int k0 = 0; k0 < N; k0 += BS) {
                    int k_max = k0 + BS > N ? N : k0 + BS;
                    int jb = j_max - j0;
                    int ib = i_max - i0;
                    for (int k = k0; k < k_max; ++k) {
                        const double *srcB = &B[k * N + j0];
                        double *dstB = &Bpack[(k - k0) * jb];
                        for (int j = 0; j < jb; ++j)
                            dstB[j] = srcB[j];
                    }
                    for (int i = i0; i < i_max; ++i) {
                        const double *srcArow = &A[i * N + k0];
                        double *dstA = &Apack[0];
                        for (int k = k0; k < k_max; ++k)
                            dstA[k - k0] = srcArow[k - k0];
                        for (int j = j0; j + VEC <= j_max; j += VEC) {
                            __m256d cvec = _mm256_loadu_pd(&C[i * N + j]);
                            for (int k = 0; k < k_max - k0; ++k) {
                                __m256d avec = _mm256_set1_pd(Apack[k]);
                                __m256d bvec = _mm256_load_pd(&Bpack[k * jb + (j - j0)]);
                                cvec = _mm256_fmadd_pd(avec, bvec, cvec);
                            }
                            _mm256_storeu_pd(&C[i * N + j], cvec);
                        }
                        for (int j = j0 + ((j_max - j0) / VEC) * VEC; j < j_max; ++j) {
                            double sum = C[i * N + j];
                            for (int k = 0; k < k_max - k0; ++k)
                                sum += Apack[k] * Bpack[k * jb + (j - j0)];
                            C[i * N + j] = sum;
                        }
                    }
                }
            }
        }
        _mm_free(Bpack);
        _mm_free(Apack);
    }
```

## VibeCodeHPC

- Multi-agent system for auto-tuning HPC code using CLI-based LLM agents (currently based on Anthropic Claude Code)
- Intended for vibe coding, but currently closer to agentic coding
- Multiple specialized agents collaborate, including Project Manager (PM), System Engineer (SE), Programmer (PG), and Continuous Deliverer (CD) (the configuration is customizable)
- Supports dynamic role reconfiguration and monitoring of activity, resource, budget, etc.
- Compared to a single-agent setup, it improves code generation quality per unit time and better detects requirement violations and other issues




S. Hayashi, K. Morita, D. Mukunoki, T. Hoshino, T. Katagiri, ”VibeCodeHPC: An Agent-Based Iterative Prompting Auto-Tuner for HPC Code Generation Using LLMs”, arXiv preprint arXiv:2510.00031, September 2025.
VibeCodeHPC - Multi Agentic Vibe Coding for HPC, https://github.com/Katagiri-Hoshino-Lab/VibeCodeHPC-jp

## Fortran to GPU

- GPU porting of GeoFEM/Cube - a fixed-length Fortran 90 finite volume Poisson solver (with ICCG method) - using Anthropic Claude Code
- The input package included OpenMP-enabled code plus Word/PDF documentations.
- The CG solver component was successfully ported to GPU, but the custom matrix generation code was not effectively implemented
- The LLM appears to have learned “CG solver optimization techniques” rather than general optimization strategies.

**Acknowledgement**
This research was supported by JSPS KAKENHI Grant Number JP23K11126 and JP24K02945. In addition, this work was supported by the Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (JHPCN) and the High-Performance Computing Infrastructure (HPCI) under project number jh250015.