Automatic Generation of Numerical Codes for GPUs Using LLMs[1)]

Daichi Mukunoki [1] Shunichiro Hayashi [2] Tetsuya Hoshino [1] Ryo Mikasa [3]
Koki Morita [2] Takahiro Katagiri [1]

[1]Information Technology Center, Nagoya University, [2]Graduate School of Informatics, Nagoya University,
[3]School of Informatics, Nagoya University

JHPCN Field Workshop
State-of-the-Art in Code Generative AI for High-Performance Computing
Dec. 5, 2025.

[1)]This research was supported by the Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (JHPCN) and the High Performance Computing Infrastructure (HPCI) (Project ID: jh250015). It was also supported by JSPS KAKENHI Grant Numbers JP23K11126 and JP24K02945.

HPC-GENIE[a] – High-Performance Computing with GEnerative Neural Intelligence for Execution

- A project for HPC code generation using LLMs at the Information Technology Center, Nagoya University
  - Project leader: Takahiro Katagiri
  - Sub-leader: Daichi Mukunoki

Key Missions

- Automatic HPC code optimization
- Fortran to GPU for Fugaku NEXT
- Technologies for local LLMs
- And more ...

[a] https://www.hpc.itc.nagoya-u.ac.jp/menu/hpc_genie.html

BLAS Code Generation Using o4-mini & GPT-4.1[2)]

Number of Correct Codes (in 10 Generations)

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

- OpenAI GPT-4.1 & o4-mini – entry-level general-purpose LLMs
- **Generation with one simple prompt** (for optimized code): "*Implement #ROUTINE# routine in BLAS in C language. Thread parallelization, SIMD vectorization, and cache blocking should be considered for speedup.*"
- Given just a routine name, working code can often be generated.

[2)] D. Mukunoki, S. Hayashi, T. Hoshino, T. Katagiri, "Performance Evaluation of General Purpose Large Language Models for Basic Linear Algebra Subprograms Code Generation", arXiv preprint arXiv:2507.04697, 2025.

DGEMM

Reference Fortran code (part)

```
      IF (NOTB) THEN
          IF (NOTA) THEN
*           Form  C := alpha*A*B + beta*C.
              DO 90 J = 1,N (略)


                  DO 80 L = 1,K
                      TEMP = ALPHA*B(L,J)
                      DO 70 I = 1,M
                          C(I,J) = C(I,J) + TEMP*A(I,L
                                   )
   70                     CONTINUE
   80                 CONTINUE
   90             CONTINUE
          ELSE
*           Form  C := alpha*A**T*B + beta*C
              DO 120 J = 1,N
                  DO 110 I = 1,M
                      TEMP = ZERO
                      DO 100 L = 1,K
                          TEMP = TEMP + A(L,I)*B(L,J)
  100                 CONTINUE
                      IF (BETA.EQ.ZERO) THEN
                          C(I,J) = ALPHA*TEMP
                      ELSE
                          C(I,J) = ALPHA*TEMP + BETA*C
                                   (I,J)
                      END IF
  110             CONTINUE
  120         CONTINUE
          END IF
      ELSE
          IF (NOTA) THEN
```

Generated C code (an example)

```
#include <stdio.h>

void GPTBLAS_dgemm(const char *transa, const char *
     transb, const int *m, const int *n, const int *k,
     const double *alpha, const double *a, const int *
     lda, const double *b, const int *ldb, const double
      *beta, double *c, const int *ldc)
{
    printf("[gptblas]");
    int i, j, l;
    int nota = (*transa == 'N' || *transa == 'n');
    int notb = (*transb == 'N' || *transb == 'n');
    for (j = 0; j < *n; j++) {
        for (i = 0; i < *m; i++) {
            double cij = (*beta) * c[i + j * (*ldc)];
            double sum = 0.0;
            for (l = 0; l < *k; l++) {
                double a_val = nota ? a[i + l * (*lda)
                     ] : a[l + i * (*lda)];
                double b_val = notb ? b[l + j * (*ldb)
                     ] : b[j + l * (*ldb)];
                sum += a_val * b_val;
            }
            c[i + j * (*ldc)] = cij + (*alpha) * sum;
        }
    }
}
```

Code structure is not necessarily the same as the reference Fortran code

Level-1 BLAS Performance (GB/s, n=16777216)

		Optimized C code from routine name only				Optimized C code from reference Fortran code			
Routine	Ref	GPT-4.1		o4-mini		GPT-4.1		o4-mini	
dasum	6.0	68.0	(11.4x)	63.5	(10.7x)	61.5	(10.3x)	63.6	(10.7x)
daxpy	17.2	77.6	(4.5x)	68.8	(4.0x)	71.9	(4.2x)	67.8	(3.9x)
ddot	11.7	65.8	(5.6x)	63.1	(5.4x)	61.0	(5.2x)	60.8	(5.2x)
idamax	7.1	63.8	(9.0x)	65.1	(9.2x)	60.3	(8.6x)	62.8	(8.9x)
dnrm2	5.1	66.5	(12.9x)	64.0	(12.5x)	60.6	(11.8x)	63.5	(12.4x)
drot	10.6	40.6	(3.8x)	34.1	(3.2x)	34.9	(3.3x)	34.4	(3.3x)
drotm	11.1	39.6	(3.6x)	36.6	(3.3x)	34.7	(3.1x)	34.1	(3.1x)

- "Ref": reference Fortran code (non-parallelized, non-optimized)
- Xeon Gold 6230 (20 cores, Cascade Lake) $\times$ 2, 40 threads, gcc/gfortran 11.3.0 -march=native
- Best result among the 10 generated codes
- Blank entries indicate that no working code was generated

Level-2 BLAS Performance (GB/s, m=n=8129)

Routine	Parameters	Ref	Optimized C code from routine name only				Optimized C code from reference Fortran code			
			GPT-4.1		o4-mini		GPT-4.1		o4-mini	
dgemv	trans=N	9.1	5.4	(0.6x)	30.0	(3.3x)	32.3	(3.5x)	6.6	(0.7x)
	trans=T	6.7	68.9	(10.4x)	66.3	(10.0x)	67.6	(10.2x)	65.9	(9.9x)
dger		18.5	46.3	(2.5x)	64.9	(3.5x)	66.4	(3.6x)	64.2	(3.5x)
dsymv	uplo=L	6.1			4.6	(0.8x)			4.3	(0.7x)
	uplo=U	6.1			5.3	(0.9x)			4.7	(0.8x)
dsyr	uplo=L	17.4	32.4	(1.9x)	64.0	(3.7x)	64.5	(3.7x)	60.9	(3.5x)
	uplo=U	17.5			61.5	(3.5x)	64.4	(3.7x)	58.9	(3.4x)
dsyr2	uplo=L	7.6	26.4	(3.5x)	63.7	(8.3x)	61.0	(8.0x)	60.2	(7.9x)
	uplo=U	15.5	29.0	(1.9x)	59.6	(3.9x)	62.1	(4.0x)	58.6	(3.8x)
dtrmv	uplo=L, trans=N, diag=N	7.6			7.3	(1.0x)	3.0	(0.4x)	3.0	(0.4x)
	uplo=L, trans=N, diag=U	7.5			7.7	(1.0x)	2.9	(0.4x)	2.9	(0.4x)
	uplo=L, trans=T, diag=N	6.5			56.9	(8.8x)	3.3	(0.5x)	3.2	(0.5x)
	uplo=L, trans=T, diag=U	6.5			56.3	(8.7x)	3.3	(0.5x)	3.2	(0.5x)
	uplo=U, trans=N, diag=N	9.0			7.6	(0.8x)	3.1	(0.3x)	3.1	(0.3x)
	uplo=U, trans=N, diag=U	8.8			7.6	(0.9x)	3.1	(0.3x)	3.1	(0.4x)
	uplo=U, trans=T, diag=N	6.1			56.7	(9.3x)	3.2	(0.5x)	3.2	(0.5x)
	uplo=U, trans=T, diag=U	6.1			56.4	(9.3x)	3.2	(0.5x)	3.2	(0.5x)

- "Ref": reference Fortran code (non-parallelized, non-optimized)
- Xeon Gold 6230 (20 cores, Cascade Lake) $\times$ 2, 40 threads, gcc/gfortran 11.3.0 -march=native
- Best result among the 10 generated codes
- Blank entries indicate that no working code was generated

Level-3 BLAS Performance (GFlops/s, m=n=k=2048)

Routine	Parameters	Ref	Optimized C code from routine name only				Optimized C code from reference Fortran code			
			GPT-4.1		o4-mini		GPT-4.1		o4-mini	
dgemm	transa=N, transb=N	2.7	17.4	(6.5x)	20.5	(7.7x)	18.0	(6.8x)	20.8	(7.8x)
	transa=N, transb=T	2.5	16.5	(6.5x)			16.1	(6.3x)	20.7	(8.1x)
	transa=T, transb=N	1.8	16.8	(9.3x)	0.8	(0.5x)	29.2	(16.1x)	23.8	(13.1x)
	transa=T, transb=T	0.3	15.7	(46.3x)			3.5	(10.4x)	20.3	(59.8x)
dsymm	side=L, uplo=L	3.6	13.6	(3.7x)	17.2	(4.7x)	19.9	(5.5x)	22.3	(6.2x)
	side=L, uplo=U	3.5	13.9	(4.0x)	17.3	(4.9x)	19.9	(5.7x)	22.4	(6.4x)
	side=R, uplo=L	2.7	13.4	(4.9x)	16.4	(6.0x)	19.0	(7.0x)	21.2	(7.8x)
	side=R, uplo=U	2.7	13.4	(4.9x)	21.3	(7.8x)	18.7	(6.8x)	21.1	(7.7x)
dsyrk	uplo=L, trans=N	1.0			2.9	(2.9x)	18.2	(18.2x)	16.6	(16.6x)
	uplo=L, trans=T	1.9	20.1	(10.6x)	21.8	(11.5x)	27.7	(14.7x)	23.0	(12.2x)
	uplo=U, trans=N	2.3			3.8	(1.6x)	16.5	(7.1x)	16.5	(7.1x)
	uplo=U, trans=T	1.9	36.5	(19.2x)	21.0	(11.0x)	34.7	(18.2x)	22.5	(11.9x)
dsyr2k	uplo=L, trans=N	1.8					31.0	(16.8x)	18.4	(10.0x)
	uplo=L, trans=T	3.1	42.9	(13.8x)			29.6	(9.5x)	23.1	(7.4x)
	uplo=U, trans=N	3.1					30.2	(9.9x)	18.1	(5.9x)
	uplo=U, trans=T	3.1	43.2	(13.7x)			27.4	(8.7x)	23.0	(7.3x)

- "Ref": reference Fortran code (non-parallelized, non-optimized)
- Xeon Gold 6230 (20 cores, Cascade Lake) × 2, 40 threads, gcc/gfortran 11.3.0 -march=native
- Best result among the 10 generated codes
- Blank entries indicate that no working code was generated

Optimized Code / Summary

- Performance optimization is very poor...
- OpenMP is applied, but not always with the optimal strategy
- SIMD is often applied with #pragma omp simd or AVX2/AVX-512 intrinsics
  - Code branching is implemented with macros (__AVX__, __AVX512F__)
- Block size is often arbitrarily set to 64 (some cases use 128 or 256)

- Difficult to generate sufficiently optimized code in one shot from a single simple prompt
- GEMM is not necessarily easy

Auto-Tuning System Using Local LLMs on Consumer-Grade PCs

Why Local LLMs? – Issues with Commercial Services

- High costs: Cloud-based API usage fees or subscription charges
- Black box: Closed source, making custom improvements and verification difficult for R&D
- Security concerns: Code and research data processed on external servers

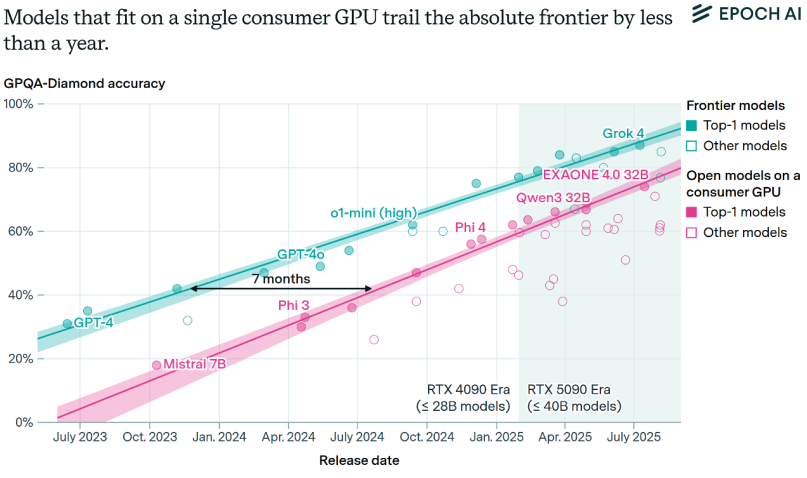
Objective 1: Building a Lightweight System Runnable on Consumer-Grade Local PCs

- High-end open-source models are available but require large-scale computing resources
- For research purpose: to examine techniques to compensate for limited model performance

Objective 2: Specializing in Performance Optimization of HPC Code

- HPC code development is more challenging than general code development
- Performance optimization is not just about "making it work"
- An iterative code improvement process (implement $\rightarrow$ evaluate $\rightarrow$ modify) is required

Expectations for Local LLMs Runnable on Consumer-Grade PCs



3)

[3] Source: https://epoch.ai/data-insights/consumer-gpu-model-gap

Our Prototype System

Simple Automatic Code Optimization System for Local PCs

- This is more like a toy, still just practice ...
- Multi-agent system to compensate for limited model performance
- Automatic iterative prompt generation for autonomous optimization (non-stop without human intervention)
- Using gpt-oss-120b (for now)

OpenAI gpt-oss-120b[4] (August 2025)

- Open model with approximately 120 billion parameters (about 80 GB needed)
- Comparable performance to o4-mini (recall the results of BLAS generation)
- Can run on AMD Ryzen AI Max+ 395 with 128GB unified memory[5]
- PC price: approximately $2,000 → Can be considered "runnable on consumer-grade PCs"

[4] https://github.com/openai/gpt-oss

[5] https://www.amd.com/en/products/processors/laptop/ryzen/ai-300-series/amd-ryzen-ai-max-plus-395.html

System Workflow: Iterative Optimization Phase

- Initial prompt: "Optimize this code for the target processor" (actually more complicated)

1. **Programmer (PG) 1 – PG 5: Code generation**
   - Each PG independently creates code
2. **Testing and benchmarking each code**
   - Generated code is compiled and verified/evaluated using test programs
3. **Debugger (DG): Debugging**
   - DG analyzes the cause when errors occur and PG modifies the code (up to 2 retries)
4. **PG 1 – PG 5: Code analysis**
   - Analyzes which optimizations contributed to performance improvement or degradation
5. **Project Manager (PM): Optimization strategy planning for next iteration**
   - Collects analysis results from PGs and generates different prompts for each PG
   - Adopts the fastest code as the base for the next generation

- Repeat from step 1 until maximum iterations or time limit is reached

System Specification

- The user does not give any target-code-specific instructions to the system – any prompts are given; the system automatically generates iterative prompts from the initial prompt to proceed with optimization
- Optimization target is one file (computation kernel in our evaluation)
- Makefile, compiler options, execution way cannot be changed
- Benchmark and verification program is given by user (system does not edit this)
- Using gpt-oss-120b (temperature = 1.5 for high randomness) via Cerebras inference API

Performance Evaluation Environment

Evaluation Environment

- Xeon Gold 6230 (20 cores) $\times$ 2 (2688 GFlops/s in FP64, 281.5 GB/s)
- gcc 11.3.0, `-O3 -march=native -fopenmp -lm -lpthread` (system cannot change)
- MKL links and paths configured for verification and performance comparison (practically MKL can be used in the optimization code)

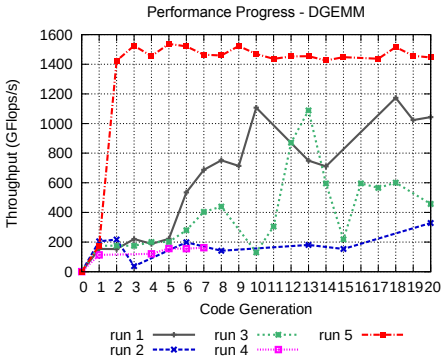
Conditions

- Strings that could reveal code content are not passed to the LLM
- Use of MKL is prohibited in optimization (but violated sometimes...)

Test 1: DGEMM

Input Code

```
1  void exp1_opt(int N,
2                double *A,
3                double *B,
4                double *C) {
5    for (int i = 0; i < N; i++) {
6      for (int j = 0; j < N; j++) {
7        double sum = 0.0;
8        for (int k = 0; k < N; k++) {
9          sum += A[i*N+k] * B[k*N+j];
10       }
11       C[i*N+j] = sum;
12     }
13   }
14 }
```



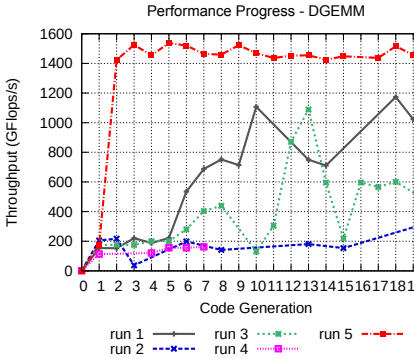
Attention!: how to read the figure

- 5 experiments (run 1 – 5) were performed with the same input and under the same conditions, due to random nature of LLM outputs
- The horizontal axis represents the number of iterations (code generations)
- Each iteration generates 5 codes, and the performance of the fastest code is plotted

Test 1: DGEMM

Input Code

```
1  void exp1_opt(int N,
2                double *A,
3                double *B,
4                double *C) {
5    for (int i = 0; i < N; i++) {
6      for (int j = 0; j < N; j++) {
7        double sum = 0.0;
8        for (int k = 0; k < N; k++) {
9          sum += A[i*N+k] * B[k*N+j];
10       }
11       C[i*N+j] = sum;
12     }
13   }
14 }
```



- Performance evaluated with $n = 1024$ square matrices (small for target)
- Fastest: 1537.06 GFlops/s in run 5, but this called MKL's dgemm
  → LLM recognized this code as a mat-mul (but use of BLAS was prohibited in prompt)
- Full-scratch fastest: 1173.42 GFlops/s at Gen. 18 of run 1
- Large variation in results across runs

Fastest Generated DGEMM Code (1/3)

```
static inline void pack_B_panel(const double *B_src,
    double *B_pack, int N, int kp, int jp, int k_max,
    int blk_width) {
  for (int k = kp; k < k_max; ++k) {
    const double *src = &B_src[k * N + jp];
    double *dst = &B_pack[(k - kp) * blk_width];
    int j = 0;
    _mm_prefetch((const char *)src, _MM_HINT_T0);
    for (; j + 8 <= blk_width; j += 8) {
      __m512d v = _mm512_loadu_pd(&src[j]);
      _mm512_storeu_pd(&dst[j], v);
    }
    for (; j < blk_width; ++j) { dst[j] = src[j]; }
  }
  _mm_sfence();
}
void exp1_opt(int N, double *restrict A, double *
    restrict B, double *restrict C) {
  const int BLOCK = 64;
  omp_set_num_threads(omp_get_max_threads());
  #pragma omp parallel for collapse(2) schedule(static)
  for (int ii = 0; ii < N; ii += BLOCK) {
    for (int jj = 0; jj < N; jj += BLOCK) {
      int i_max = (ii + BLOCK > N) ? N : ii + BLOCK;
      int j_max = (jj + BLOCK > N) ? N : jj + BLOCK;
      int blk_width = j_max - jj;
      int vec_width = blk_width & ~31;
      double C_local[BLOCK * BLOCK] __attribute__((
          aligned(64)));
      double *C_local_aligned = (double *)
          __builtin_assume_aligned(C_local, 64);
      for (int i = 0; i < BLOCK * BLOCK; ++i)
        C_local_aligned[i] = 0.0;
      for (int kk = 0; kk < N; kk += BLOCK) {
        int k_max = (kk + BLOCK > N) ? N : kk +
            BLOCK;
        double B_pack_local[BLOCK * BLOCK]
            __attribute__((aligned(64)));
        pack_B_panel(B, B_pack_local, N, kk, jj,
            k_max, blk_width);
        for (int i = ii; i + 3 < i_max; i += 4) {
          const double *a_row0 = &A[(i + 0) * N +
              kk];
          const double *a_row1 = &A[(i + 1) * N +
              kk];
          const double *a_row2 = &A[(i + 2) * N +
              kk];
          const double *a_row3 = &A[(i + 3) * N +
              kk];
          double *c_loc0 = &C_local_aligned[(i - ii
              ) * BLOCK];
          double *c_loc1 = &C_local_aligned[(i - ii
              + 1) * BLOCK];
          double *c_loc2 = &C_local_aligned[(i - ii
              + 2) * BLOCK];
          double *c_loc3 = &C_local_aligned[(i - ii
              + 3) * BLOCK];
          for (int j = 0; j < vec_width; j += 16) {
            __m512d c00 = _mm512_loadu_pd(&c_loc0[
                j + 0]);
            __m512d c01 = _mm512_loadu_pd(&c_loc0[
                j + 8]);
            __m512d c10 = _mm512_loadu_pd(&c_loc1[
                j + 0]);
            __m512d c11 = _mm512_loadu_pd(&c_loc1[
                j + 8]);
            __m512d c20 = _mm512_loadu_pd(&c_loc2[
                j + 0]);
            __m512d c21 = _mm512_loadu_pd(&c_loc2[
                j + 8]);
            __m512d c30 = _mm512_loadu_pd(&c_loc3[
                j + 0]);
            __m512d c31 = _mm512_loadu_pd(&c_loc3[
                j + 8]);
            for (int k = kk; k < k_max; ++k) {
              const int k_off = k - kk;
              __m512d b0 = _mm512_load_pd(&
                  B_pack_local[k_off * blk_width
                  + j + 0]);
              __m512d b1 = _mm512_load_pd(&
                  B_pack_local[k_off * blk_width
                  + j + 8]);
              __m512d a0 = _mm512_set1_pd(a_row0[
                  k_off]);
              __m512d a1 = _mm512_set1_pd(a_row1[
                  k_off]);
```

Fastest Generated DGEMM Code (2/3)

```
                    __m512d a2 = _mm512_set1_pd(a_row2[
                        k_off]);
                    __m512d a3 = _mm512_set1_pd(a_row3[
                        k_off]);
                    c00 = _mm512_fmadd_pd(a0, b0, c00);
                    c01 = _mm512_fmadd_pd(a0, b1,
                        c01);
                    c10 = _mm512_fmadd_pd(a1, b0, c10);
                    c11 = _mm512_fmadd_pd(a1, b1,
                        c11);
                    c20 = _mm512_fmadd_pd(a2, b0, c20);
                    c21 = _mm512_fmadd_pd(a2, b1,
                        c21);
                    c30 = _mm512_fmadd_pd(a3, b0, c30);
                    c31 = _mm512_fmadd_pd(a3, b1,
                        c31);
                }
                _mm512_storeu_pd(&c_loc0[j + 0], c00);
                    _mm512_storeu_pd(&c_loc0[j + 8],
                    c01);
                _mm512_storeu_pd(&c_loc1[j + 0], c10);
                    _mm512_storeu_pd(&c_loc1[j + 8],
                    c11);
                _mm512_storeu_pd(&c_loc2[j + 0], c20);
                    _mm512_storeu_pd(&c_loc2[j + 8],
                    c21);
                _mm512_storeu_pd(&c_loc3[j + 0], c30);
                    _mm512_storeu_pd(&c_loc3[j + 8],
                    c31);
            }
            for (int j = vec_width; j < blk_width; ++
                j) {
                double s0 = c_loc0[j]; double s1 =
                    c_loc1[j];
                double s2 = c_loc2[j]; double s3 =
                    c_loc3[j];
                for (int k = kk; k < k_max; ++k) {
                    const int k_off = k - kk;
                    double b = B_pack_local[k_off *
                        blk_width + j];
                    s0 += a_row0[k_off] * b; s1 +=
                        a_row1[k_off] * b;
                    s2 += a_row2[k_off] * b; s3 +=
                        a_row3[k_off] * b;
                }
                c_loc0[j] = s0; c_loc1[j] = s1; c_loc2
                    [j] = s2; c_loc3[j] = s3;
            }
        }
        for (int i = i_max - ((i_max - ii) % 4); i <
            i_max; ++i) {
            const double *a_ptr = &A[i * N + kk];
            double *c_loc = &C_local_aligned[(i - ii)
                * BLOCK];
            for (int j = 0; j < vec_width; j += 8) {
                __m512d c_vec = _mm512_loadu_pd(&c_loc
                    [j]);
                for (int k = kk; k < k_max; ++k) {
                    const int k_off = k - kk;
                    __m512d b_vec = _mm512_load_pd(&
                        B_pack_local[k_off * blk_width
                        + j]);
                    __m512d a_vec = _mm512_set1_pd(
                        a_ptr[k_off]);
                    c_vec = _mm512_fmadd_pd(a_vec,
                        b_vec, c_vec);
                }
                _mm512_storeu_pd(&c_loc[j], c_vec);
            }
            for (int j = vec_width; j < blk_width; ++
                j) {
                double sum = c_loc[j];
                for (int k = kk; k < k_max; ++k) {
                    const int k_off = k - kk;
                    sum += a_ptr[k_off] * B_pack_local[
                        k_off * blk_width + j];
                }
                c_loc[j] = sum;
            }
        }
    }
    for (int i = ii; i < i_max; ++i) {
        double *c_ptr = &C[i * N + jj];
        double *c_loc = &C_local_aligned[(i - ii) *
            BLOCK];
```

Fastest Generated DGEMM Code (3/3)

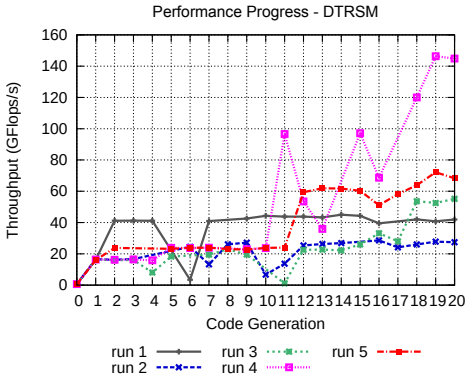
```
106             c_ptr = (double *)__builtin_assume_aligned(
                    c_ptr, 64);
107             c_loc = (double *)__builtin_assume_aligned(
                    c_loc, 64);
108             int j = 0;
109             for (; j < vec_width; j += 8) {
110                 __m512d tmp = _mm512_loadu_pd(&c_loc[j]);
111                 _mm512_storeu_pd(&c_ptr[j], tmp);
112             }
113             for (; j < blk_width; ++j) {
114                 c_ptr[j] = c_loc[j];
115             }
116         }
117     }
118 }
119
```

*Due to space limitations, auto-inserted comments, blank lines, and header includes were removed from the actual generated code, and indentation/line breaks were partially modified.

Test 2: DTRSM

Input Code

```
1  void exp2_opt(int N,
2      double *A, double *B,
3      double *X, double alpha) {
4    for (int i = 0; i < N; i++) {
5      for (int j = 0; j < N; j++) {
6        X[i*N+j] = alpha * B[i*N+j];
7      }
8    }
9    for (int j = 0; j < N; j++) {
10     for (int i = 0; i < N; i++) {
11       for (int k = 0; k < i; k++) {
12         X[i*N+j] -= A[i*N+k]*X[k*N+j];
13       }
14       X[i*N+j] /= A[i*N+i];
15     }
16   }
17 }
```



- Solving equations with triangular matrix coefficients (lower triangular, $n = 1024$); more complex data dependencies than DGEMM; high optimization difficulty
- Fastest: 146.33 GFlops/s at Gen.19 of run 4 (MKL: 305.74 GFlops/s, 48% of MKL)
- Not observed in this experiment, but cases using MKL were seen during test runs

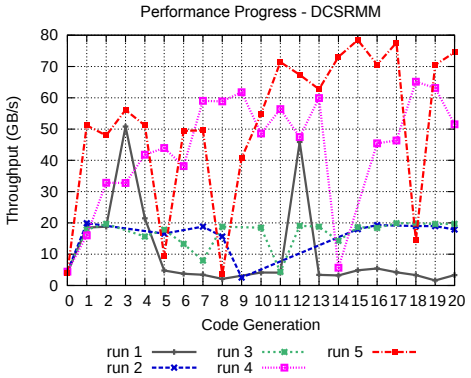
Fastest Generated DTRSM Code (part)

```
#define COL_BLOCK 8
void exp2_opt(int N, double * restrict A, double *
     restrict B,
              double * restrict X, double alpha) {
    double inv_diag[N] __attribute__((aligned(64)));
    double *trans_X = (double*)aligned_alloc(64, (size_t)
         N * N * sizeof(double));
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < N; ++i) {
       double *xrow = X + i * N;
       double *brow = B + i * N;
       int j = 0;
       __m512d valpha = _mm512_set1_pd(alpha);
       for (; j <= N-8; j += 8) {
          __m512d v = _mm512_loadu_pd(brow + j);
          v = _mm512_mul_pd(v, valpha);
          _mm512_storeu_pd(xrow + j, v);
       }
       for (; j < N; ++j) xrow[j] = alpha * brow[j];
       inv_diag[i] = 1.0 / A[i * N + i];
    }
    const int BS = 64;
    #pragma omp parallel for collapse(2) schedule(static)
    for (int i0 = 0; i0 < N; i0 += BS) {
       for (int j0 = 0; j0 < N; j0 += BS) {
          int i_max = i0 + BS < N ? i0 + BS : N;
          int j_max = j0 + BS < N ? j0 + BS : N;
          for (int i = i0; i < i_max; ++i) {
             double *xrow = X + i * N;
             for (int j = j0; j < j_max; ++j) {
                trans_X[j * N + i] = xrow[j];
             }
          }
       }
    }
    #pragma omp parallel
    {
       #pragma omp for schedule(runtime) nowait
       for (int jb = 0; jb < N; jb += COL_BLOCK) {
          int j_end = (jb + COL_BLOCK < N) ? jb +
               COL_BLOCK : N;
          #pragma omp task depend(inout: trans_X[jb * N])
          {
             for (int j = jb; j < j_end; ++j) {
                double *xcol = trans_X + j * N;
                #pragma omp simd
                for (int i = 0; i < N; ++i) {
                   const double *arow = A + i * N;
                   __m512d vsum = _mm512_setzero_pd();
                   int limit = i & ~7;
                   if (limit >= 8) {
                      _mm_prefetch(arow + 8, _MM_HINT_T0)
                           ;
                      _mm_prefetch(xcol + 8, _MM_HINT_T0)
                           ;
                      __m512d a_prev = _mm512_loadu_pd(
                           arow);
                      __m512d x_prev = _mm512_loadu_pd(
                           xcol);
                      int k = 8;
                      for (; k < limit; k += 8) {
                         _mm_prefetch(arow + k + 8,
                              _MM_HINT_T0);
                         _mm_prefetch(xcol + k + 8,
                              _MM_HINT_T0);
                         __m512d a_cur = _mm512_loadu_pd(
                              arow + k);
                         __m512d x_cur = _mm512_loadu_pd(
                              xcol + k);
                         vsum = _mm512_fmadd_pd(a_prev,
                              x_prev, vsum);
                         a_prev = a_cur; x_prev = x_cur;
                      }
                      vsum = _mm512_fmadd_pd(a_prev,
                           x_prev, vsum);
                   }
                   int rem = i - limit;
                   if (rem) {
                      _mm_prefetch(arow + limit + 8,
                           _MM_HINT_T0);
                      _mm_prefetch(xcol + limit + 8,
                           _MM_HINT_T0);
                      __mmask8 m = (1U << rem) - 1U;
                      __m512d a = _mm512_maskz_loadu_pd(m
                           , arow + limit);
                      __m512d x = _mm512_maskz_loadu_pd(m
                           , xcol + limit);
                      vsum = _mm512_fmadd_pd(a, x, vsum);
```

Test 3: DCSRMM

Input Code

```
1  void exp3_opt(int m, int k, int n,
2      const int *row_ptr,
3      const int *col_indices,
4      const double *values,
5      const double *B, double *C) {
6    memset(C, 0, m*n*sizeof(double));
7    for (int i = 0; i < m; i++) {
8      int row_start = row_ptr[i];
9      int row_end = row_ptr[i + 1];
10     for (int idx=row_start;
11          idx<row_end; idx++) {
12       int col = col_indices[idx];
13       double val = values[idx];
14       for (int j = 0; j < n; j++) {
15         C[i*n+j] += val*B[col*n+j];
16       }
17     }
18   }
19 }
```



- Sparse-dense mat-mul in CSR format (gridgena.mtx: $n = 48962$, $nnz = 512084$, RHS width $m = 128$); irregular memory access; depends on sparse structure
- Fastest: 78.45 GB/s at Gen. 15 of run 5 (MKL: 86.88 GB/s, 90% of MKL)
- Attempted FP16 (failed), CSC and ELLPACK format changes (failed) $\rightarrow$ sparse matrix structure was recognized

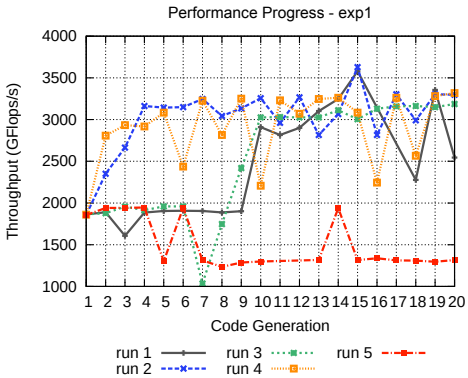
Fastest Generated DCSRMM Code (part)

```
#define UNROLL_FACTOR 4
#define VEC_WIDTH 8
#define UNROLL_STEP (UNROLL_FACTOR * VEC_WIDTH)
void exp3_opt(int m, int k, int n,
              const int *__restrict row_ptr,
              const int *__restrict col_indices,
              const double *__restrict values,
              const double *__restrict B,
              double *__restrict C) {
    int block_n = 64;
    const char *env_n = getenv("BLOCK_N");
    if (env_n) { int v = atoi(env_n); if (v > 0) block_n
         = v; }
    int block_m = 128;
    const char *env_m = getenv("BLOCK_M");
    if (env_m) { int v = atoi(env_m); if (v > 0) block_m
         = v; }
    int block_shift = 0;
    while ((1 << block_shift) < block_n) ++block_shift;
    #pragma omp parallel for schedule(static)
    for (int i0 = 0; i0 < m; i0 += block_m) {
        int i1 = i0 + block_m;
        if (i1 > m) i1 = m;
        for (int i = i0; i < i1; ++i) {
            double *c_row = C + (size_t)i * n;
            int row_start = row_ptr[i];
            int row_end   = row_ptr[i + 1];
            for (int idx = row_start; idx < row_end; ++idx)
                 {
                int col = col_indices[idx];
                double val = values[idx];
                __m512d val_vec = _mm512_set1_pd(val);
                int block_id = col >> block_shift;
                int col_off  = col & (block_n - 1);
                const double *b_ptr = B +
                    ((size_t)block_id * block_n + (size_t)
                         col_off) * n;
                int j = 0;
                int unroll_limit = n - (n % UNROLL_STEP);
                for (; j < unroll_limit; j += UNROLL_STEP) {
                    __m512d c0 = _mm512_loadu_pd(c_row + j);
                    __m512d b0 = _mm512_loadu_pd(b_ptr + j);
                    __m512d c1 = _mm512_loadu_pd(c_row + j +
                         VEC_WIDTH);
                    __m512d b1 = _mm512_loadu_pd(b_ptr + j +
                         VEC_WIDTH);
                    __m512d c2 = _mm512_loadu_pd(c_row + j +
                         2 * VEC_WIDTH);
                    __m512d b2 = _mm512_loadu_pd(b_ptr + j +
                         2 * VEC_WIDTH);
                    __m512d c3 = _mm512_loadu_pd(c_row + j +
                         3 * VEC_WIDTH);
                    __m512d b3 = _mm512_loadu_pd(b_ptr + j +
                         3 * VEC_WIDTH);
                    c0 = _mm512_fmadd_pd(val_vec, b0, c0);
                    c1 = _mm512_fmadd_pd(val_vec, b1, c1);
                    c2 = _mm512_fmadd_pd(val_vec, b2, c2);
                    c3 = _mm512_fmadd_pd(val_vec, b3, c3);
                    _mm512_storeu_pd(c_row + j, c0);
                    _mm512_storeu_pd(c_row + j + VEC_WIDTH,
                         c1);
                    _mm512_storeu_pd(c_row + j + 2 *
                         VEC_WIDTH, c2);
                    _mm512_storeu_pd(c_row + j + 3 *
                         VEC_WIDTH, c3);
                }
                int tail_rem = n - j;
                int tail_vec = tail_rem & ~7;
                for (int t = 0; t < tail_vec; t += VEC_WIDTH
                     ) {
                    __m512d c_vec = _mm512_loadu_pd(c_row + j
                         + t);
                    __m512d b_vec = _mm512_loadu_pd(b_ptr + j
                         + t);
                    c_vec = _mm512_fmadd_pd(val_vec, b_vec,
                         c_vec);
                    _mm512_storeu_pd(c_row + j + t, c_vec);
                }
                int tail_elem = tail_rem - tail_vec;
                if (tail_elem) {
                    __mmask8 mask = (1U << tail_elem) - 1;
                    __m512d c_mask = _mm512_maskz_loadu_pd(
                         mask, c_row + j + tail_vec);
                    __m512d b_mask = _mm512_maskz_loadu_pd(
                         mask, b_ptr + j + tail_vec);
                    c_mask = _mm512_fmadd_pd(val_vec, b_mask,
                         c_mask);
```

Test 1: DGEMM (CUDA)

Input Code

```
1  __global__ void exp1_kernel(int N, const
       double *A, const double *B, double *C) {
2      int row = blockIdx.y * blockDim.y +
           threadIdx.y;
3      int col = blockIdx.x * blockDim.x +
           threadIdx.x;
4      if (row < N && col < N) {
5          double sum = 0.0;
6          for (int k = 0; k < N; k++) {
7              sum += A[row * N + k] * B[k * N +
                   col];
8          }
9          C[row * N + col] = sum;
10     }
11 }
12 extern "C" void exp1_opt_kernel(int N, const
       double *d_A, const double *d_B, double *
       d_C) {
13     dim3 blockDim(16, 16);
14     dim3 gridDim((N + blockDim.x - 1) /
           blockDim.x, (N + blockDim.y - 1) /
           blockDim.y);
15     exp1_kernel<<<gridDim, blockDim>>>(N, d_A
           , d_B, d_C);
16     cudaDeviceSynchronize();
17 }
```



Target: Tesla V100 (one GPU, 7000 GFlops/s in FP64)
Best code: 3628.29 GFlops/s
cuBLAS: 6168.09 GFlops/s

Fastest Generated DGEMM Code (CUDA) (1/3)

```
#include <cuda_runtime.h>
#include <stdlib.h>

extern "C" cudaError_t cudaGetDeviceProperties_v2(
    cudaDeviceProp *prop, int device) {
    return cudaGetDeviceProperties(prop, device);
}

extern "C" long __isoc23_strtol(const char *nptr, char
    **endptr, int base) {
    return strtol(nptr, endptr, base);
}

#define TILE 32                          // ×3232 タイル
#define K_UNROLL 32

// 手動アンロールマクロ
#define UNROLL_K(iter)                                  \
    do {                                                \
        double a0 = As[calc][threadIdx.y*2 + 0][iter];  \
        double a1 = As[calc][threadIdx.y*2 + 1][iter];  \
        double b0 = Bs[calc][iter][threadIdx.x*2 + 0];  \
        double b1 = Bs[calc][iter][threadIdx.x*2 + 1];  \
        regC[0][0] += a0 * b0;                          \
        regC[0][1] += a0 * b1;                          \
        regC[1][0] += a1 * b0;                          \
        regC[1][1] += a1 * b1;                          \
    } while (0)

__global__ void exp1_kernel(int N, const double *
    __restrict__ A,
                            const double *__restrict__ B,
                            double *__restrict__ C) {
    double regC[2][2] = {{0.0, 0.0},
                         {0.0, 0.0}};                   //
                                                  ×22 出力

    __shared__ double As[2][TILE][TILE];
    __shared__ double Bs[2][TILE][TILE];

    int numTiles = (N + TILE - 1) / TILE;

    int baseRow = blockIdx.y * TILE + threadIdx.y * 2;
                  // 2 行担当
    int baseCol = blockIdx.x * TILE + threadIdx.x * 2;
                  // 2 列担当

    // ---- 1st タイルを load (ping = 0) ----
    int ping = 0;
    for (int dy = 0; dy < 2; ++dy) {
        int aRow = baseRow + dy;
        int aCol = 0 * TILE + threadIdx.x * 2;
        double4 a_pair = *((double4*)(A + aRow * N +
            aCol));
        double a0 = a_pair.x;
        double a1 = a_pair.y;
        As[ping][threadIdx.y * 2 + dy][threadIdx.x * 2]
            = a0;
        As[ping][threadIdx.y * 2 + dy][threadIdx.x * 2 +
            1] = a1;
    }
    for (int dy = 0; dy < 2; ++dy) {
        int bRow = 0 * TILE + threadIdx.y * 2 + dy;
        int bCol = baseCol;
        double4 b_pair = *((double4*)(B + bRow * N +
            bCol));
        double b0 = b_pair.x;
        double b1 = b_pair.y;
        Bs[ping][threadIdx.y * 2 + dy][threadIdx.x * 2]
            = b0;
        Bs[ping][threadIdx.y * 2 + dy][threadIdx.x * 2 +
            1] = b1;
    }
    __syncthreads();
                                                  // 初期タイルロード
          待ち

    for (int t = 0; t < numTiles; ++t) {
```

Fastest Generated DGEMM Code (CUDA) (2/3)

```
65          int calc = ping;
                                                    // 現在計算に使用するバッファ
66
67          // ---- 計算 (calc バッファ) ----
68          UNROLL_K(0);  UNROLL_K(1);  UNROLL_K(2);  UNROLL_K(3);
69          UNROLL_K(4);  UNROLL_K(5);  UNROLL_K(6);  UNROLL_K(7);
70          UNROLL_K(8);  UNROLL_K(9);  UNROLL_K(10); UNROLL_K(11);
71          UNROLL_K(12); UNROLL_K(13); UNROLL_K(14); UNROLL_K(15);
72          UNROLL_K(16); UNROLL_K(17); UNROLL_K(18); UNROLL_K(19);
73          UNROLL_K(20); UNROLL_K(21); UNROLL_K(22); UNROLL_K(23);
74          UNROLL_K(24); UNROLL_K(25); UNROLL_K(26); UNROLL_K(27);
75          UNROLL_K(28); UNROLL_K(29); UNROLL_K(30); UNROLL_K(31);
76
77          // 次タイルを prefetch (ping バッファ) —すでに最後の場合はスキップ
78          ping = 1 - ping;
                                                    // バッファを反転
79          if (t + 1 < numTiles) {
80              for (int dy = 0; dy < 2; ++dy) {
81                  int aRow = baseRow + dy;
82                  int aCol = (t + 1) * TILE + threadIdx.x * 2;
83                  double4 a_pair = *((double4*)(A + aRow * N + aCol));
84                  double a0 = a_pair.x;
85                  double a1 = a_pair.y;
86                  As[ping][threadIdx.y * 2 + dy][threadIdx.x * 2]     = a0;
87                  As[ping][threadIdx.y * 2 + dy][threadIdx.x * 2 + 1] = a1;
88              }
89              for (int dy = 0; dy < 2; ++dy) {
90                  int bRow = (t + 1) * TILE + threadIdx.y * 2 + dy;
91                  int bCol = baseCol;
```

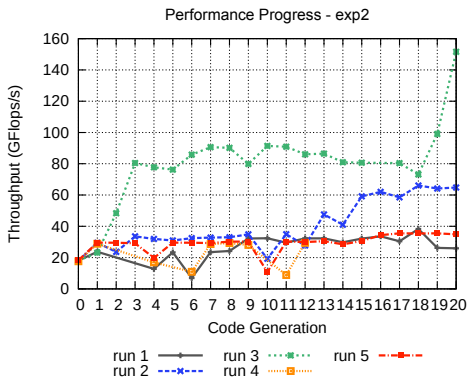
```
92                  double4 b_pair = *((double4*)(B + bRow * N + bCol));
93                  double b0 = b_pair.x;
94                  double b1 = b_pair.y;
95                  Bs[ping][threadIdx.y * 2 + dy][threadIdx.x * 2]     = b0;
96                  Bs[ping][threadIdx.y * 2 + dy][threadIdx.x * 2 + 1] = b1;
97              }
98          }
99          __syncthreads();
                                                    // 次計算に備える
100     }
101
102     // ---- 書き戻し ----
103     C[baseRow * N + baseCol]                    = regC[0][0];
104     C[baseRow * N + baseCol + 1]                = regC[0][1];
105     C[(baseRow + 1) * N + baseCol]              = regC[1][0];
106     C[(baseRow + 1) * N + baseCol + 1]          = regC[1][1];
107 }
108
109 extern "C" void exp1_opt_kernel(int N, const double *d_A,
110                                 const double *d_B,
                                    double *d_C) {
111     if (N % 8 != 0) {
112         int Npad = ((N + 7) / 8) * 8;
113         size_t pitch_pad = (size_t)Npad * sizeof(double);
114         size_t pitch_orig = (size_t)N * sizeof(double);
115         double *d_Apad = nullptr, *d_Bpad = nullptr, *d_Cpad = nullptr;
116         cudaMalloc(&d_Apad, pitch_pad * Npad);
117         cudaMalloc(&d_Bpad, pitch_pad * Npad);
118         cudaMalloc(&d_Cpad, pitch_pad * Npad);
119         cudaMemset(d_Apad, 0, pitch_pad * Npad);
120         cudaMemset(d_Bpad, 0, pitch_pad * Npad);
121         cudaMemset(d_Cpad, 0, pitch_pad * Npad);
```

Fastest Generated DGEMM Code (CUDA) (2/3)

```
122         cudaMemcpy2D(d_Apad, pitch_pad, d_A, pitch_orig,
123                      pitch_orig, N,
                             cudaMemcpyDeviceToDevice);
124         cudaMemcpy2D(d_Bpad, pitch_pad, d_B, pitch_orig,
125                      pitch_orig, N,
                                  cudaMemcpyDeviceToDevice);
126
127         dim3 blockDim(TILE / 2, TILE / 2);
128         dim3 gridDim((Npad + TILE - 1) / TILE, (Npad +
                TILE - 1) / TILE);
129         exp1_kernel<<<gridDim, blockDim>>>(Npad, d_Apad,
                d_Bpad, d_Cpad);
130         cudaDeviceSynchronize();
131
132         cudaMemcpy2D(d_C, pitch_orig, d_Cpad, pitch_pad,
133                      pitch_orig, N,
                             cudaMemcpyDeviceToDevice);
134
135         cudaFree(d_Apad);
136         cudaFree(d_Bpad);
137         cudaFree(d_Cpad);
138         return;
139     }
140
141     dim3 blockDim(TILE / 2, TILE / 2);
                                    // ×1616 スレッド
142     dim3 gridDim((N + TILE - 1) / TILE, (N + TILE - 1) /
            TILE);
143     exp1_kernel<<<gridDim, blockDim>>>(N, d_A, d_B, d_C)
            ;
144     cudaDeviceSynchronize();
145 }
146
147 #undef K_UNROLL
148 #undef UNROLL_K
```

Test 2: DTRSM (CUDA)

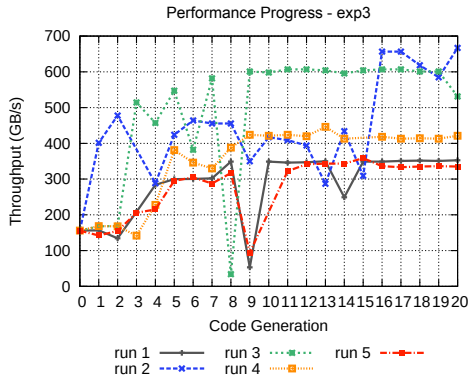
```
1  __global__ void kernel1(int N, const double *B,
       double *X, double alpha) {
2      int idx = blockIdx.x * blockDim.x + threadIdx.x;
3      if (idx < N * N)
4          X[idx] = alpha * B[idx];
5  }
6  __global__ void kernel2(int N, int row, const double
       *A, double *X) {
7      int col = blockIdx.x * blockDim.x + threadIdx.x;
8      if (col < N) {
9          double sum = 0.0;
10         for (int k = 0; k < row; k++) {
11             sum += A[row * N + k] * X[k * N + col];
12         }
13         X[row * N + col] = (X[row * N + col] - sum)
               / A[row * N + row];
14     }
15 }
16 extern "C" void exp2_opt_kernel(int N, const double
       *d_A, const double *d_B, double *d_X, double
       alpha) {
17     int total = N * N;
18     int threads = 256;
19     int blocks = (total + threads - 1) / threads;
20     kernel1<<<blocks, threads>>>(N, d_B, d_X, alpha)
           ;
21     cudaDeviceSynchronize();
22     threads = 256;
23     blocks = (N + threads - 1) / threads;
24     for (int i = 0; i < N; i++) {
25         kernel2<<<blocks, threads>>>(N, i, d_A, d_X)
               ;
26         cudaDeviceSynchronize();
27     }
28 }
```



Best code: 151.59 GFlops/s
cuBLAS: 1278.75 GFlops/s

Test 3: DCSRMM (CUDA)

```
1  __global__ void exp3_kernel(int m, int n, const
       int *row_ptr, const int *col_indices, const
       double *values, const double *B, double *C) {
2      int row = blockIdx.x * blockDim.x + threadIdx
           .x;
3      int col = blockIdx.y * blockDim.y + threadIdx
           .y;
4      if (row < m && col < n) {
5          double sum = 0.0;
6          int row_start = row_ptr[row];
7          int row_end = row_ptr[row + 1];
8          for (int idx = row_start; idx < row_end;
               idx++) {
9              int sparse_col = col_indices[idx];
10             double val = values[idx];
11             sum += val * B[sparse_col * n + col];
12         }
13         C[row * n + col] = sum;
14     }
15 }
16 extern "C" void exp3_opt_kernel(int m, int k, int
       n, const int *d_row_ptr, const int *
       d_col_indices, const double *d_values, const
       double *d_B, double *d_C) {
17     cudaMemset(d_C, 0, m * n * sizeof(double));
18     dim3 blockDim(16, 16);
19     dim3 gridDim((m + blockDim.x - 1) / blockDim.
           x, (n + blockDim.y - 1) / blockDim.y);
20     exp3_kernel<<<gridDim, blockDim>>>(m, n,
           d_row_ptr, d_col_indices, d_values, d_B,
           d_C);
21     cudaDeviceSynchronize();
22 }
```



Best code: 666.78 GFlops/s
cuSparse: 397.58 GFlops/s
→ This is true (but possible as it is specialized for the given problem)

Conclusion

Summary (our sysytem)

- Our toy system with gpt-oss-120b is pretty impressive
- Matrix multiplication is easy, right? Yes, but still difficult.
- Actual more complicated code? – will be more difficult
- There are tons of ideas for improvement

Generatie AI

- A Must-Have Technology for Fugaku NEXT
- From an academic research perspective, there is concern about whether we can catch up with the rapid pace of technological evolution (and $ cost for GPUs)
- Model performance and commercial services are evolving so rapidly – our modest efforts will soon be wasted ??? (though still meaningful as research)