SCA/HPCAsia2026

Evaluation of the Capability of Coding AI in Generating SYCL-Based Numerical Computation Codes for Intel GPUs

Koki Morita[1], Daichi Mukunoki[2], Tetsuya Hoshino[2], Takahiro Katagiri[2]

1. Graduate School of Informatics, Nagoya University 2. Information Technology Center, Nagoya University

Introduction

- **Gap:** While GPUs are standard in modern supercomputers, many numerical codes still lack GPU support.
- **Traditional Approach:** Portability frameworks (e.g., OpenACC, Kokkos [1]) facilitate GPU acceleration but rely on abstraction layers.
- **New Approach:** Generative AI agents can **direct code porting and tuning** for specific devices, offering a potential alternative to these traditional frameworks.
- **Problem:** Existing studies focus on NVIDIA GPUs (CUDA), while **Intel GPUs (SYCL)** remain underexplored despite their distinct syntax and semantics.
- **Goal:** Evaluate Claude Code (an AI Agent by Anthropic) for generating SYCL kernels targeting Intel GPUs.
- **Novelty:** Unlike existing studies focusing on CPUs[2] or simple BLAS routines[3][4], we target Intel Arc GPUs and BLAS Level 2 & 3 routines.

Method

AI Agent & Workflow

- **Agent**: **Claude Code**(v2.0.27, model: claude-sonnet-4-5-20250929)
  - Perform code optimization autonomously according to the initial instructions (prompt).
- **Input: BLAS++(1)** reference code (C++, no parallelization)
- **Task Definition:** Describe the optimization workflow in CLAUDE.md to instruct Claude Code.
  - Defined a step-by-step workflow for development, testing, and benchmarking.
  - Enforced standard C++ and SYCL (vendor libraries are prohibited).
- **Prompting Steps:** Provide the following three prompts:
  1. Start the workflow.
  2. Optimize it further.
  3. Optimize it more aggressively.

Target Benchmarks

- **BLAS Routines:** representative single-precision BLAS routines
  - Level 2: SGEMV, SSYMV, STRSV, STRMV
  - Level 3: SGEMM SSYMM, SSYRK, STRSM
- **Configuration:**
  - Problem Sizes: Fixed at 32768 (Level 2) and 8192 (Level 3).
  - Parameters: Restricted to specific settings (incx=incy=1, trans=transA=transB='N', uplo=side='L', diag='N', Layout='C')
- **Verification:**
  - Correctness verified using BLAS++ testing framework.
  - Conducted 4 independent experiments (EX1–EX4) to evaluate the stability of stochastic code generation.

Result

Evaluation Environment

- **Hardware:** Intel Arc B580 (SPARKLE SB580T-12GOC)
  - GPU boost clock: 2740 MHz
  - Theoretical FP32 peak: 14028.8 GFlops/s
  - Memory Bandwidth: 456.0 GBytes/s
- **Software:**
  - Intel oneAPI 2025.2.1 with the Math Kernel Library (MKL).

Generated Codes

- **STRMV – EX1:**

```
for (int64_t j = n-1; j >= 0; --j) {
    TX tmp = x[j];
    for (int64_t i = n-1; i >= j+1; --i) {
        x[i] += tmp * A(i, j);
    }
    if (nonunit) {
        x[j] *= A(j, j);
    }
}
```

Porting & Optimization

```
queue.stream().submit([&](sycl::handler& h) {
    h.parallel_for(sycl::range<1>(n), [=](sycl::id<1> i_id) {
        int64_t i = i_id[0];

        // Multiple independent accumulators for better ILP
        scalar_t sum0 = scalar_t(0);
        scalar_t sum1 = scalar_t(0);
        scalar_t sum2 = scalar_t(0);
        scalar_t sum3 = scalar_t(0);
        scalar_t sum4 = scalar_t(0);
        scalar_t sum5 = scalar_t(0);
        scalar_t sum6 = scalar_t(0);
        scalar_t sum7 = scalar_t(0);

        // 8-way unrolling with 8 independent accumulators
        int64_t j = 0;
        for (; j + 7 < i; j += 8) {
            sum0 += A[i + j * lda] * x[j * incx];
            sum1 += A[i + (j+1) * lda] * x[(j+1) * incx];
            sum2 += A[i + (j+2) * lda] * x[(j+2) * incx];
            sum3 += A[i + (j+3) * lda] * x[(j+3) * incx];
            sum4 += A[i + (j+4) * lda] * x[(j+4) * incx];
            sum5 += A[i + (j+5) * lda] * x[(j+5) * incx];
            sum6 += A[i + (j+6) * lda] * x[(j+6) * incx];
            sum7 += A[i + (j+7) * lda] * x[(j+7) * incx];
        }

        // Combine with tree reduction for minimal latency
        scalar_t tmp = ((sum0 + sum1) + (sum2 + sum3)) + ((sum4
+ sum5) + (sum6 + sum7));

        // Handle remainder
        for (; j < i; ++j) {
            tmp += A[i + j * lda] * x[j * incx];
        }

        // Diagonal
        if (nonunit) {
            tmp += A[i + i * lda] * x[i * incx];
        } else {
            tmp += x[i * incx];
        }

        x_temp[i] = tmp;
    });
});
```

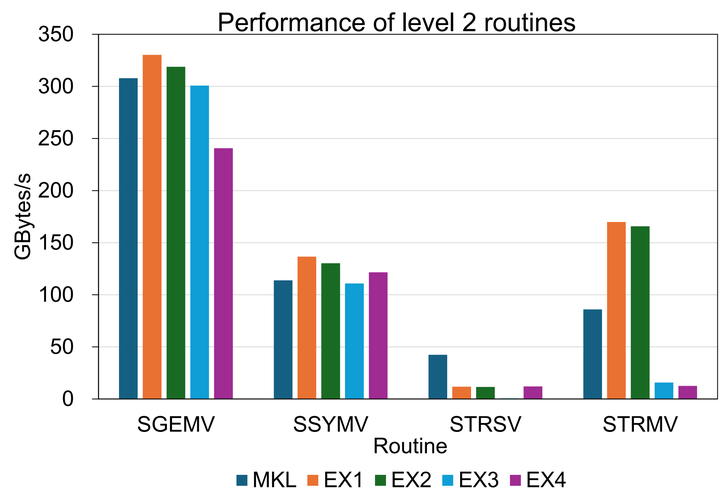
- **SGEMM – EX3:**

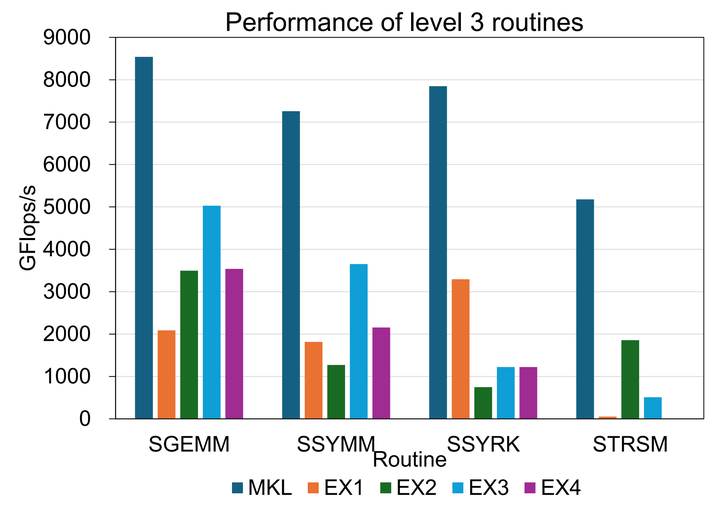
```
for (int64_t j = 0; j < n; ++j) {
    for (int64_t l = 0; l < k; ++l) {
        scalar_t alpha_Blj = alpha*B(l, j);
        for (int64_t i = 0; i < m; ++i)
            C(i, j) += A(i, l)*alpha_Blj;
    }
}
```

Porting & Optimization

```
void gemm_kernel_nn_opt(...)
{
    (omission)

    // Register arrays for accumulation
    scalar_t reg_c[REG_M][REG_N];

    // Initialize accumulators
    #pragma unroll
    for (int i = 0; i < REG_M; ++i) {
        #pragma unroll
        for (int j = 0; j < REG_N; ++j) {
            reg_c[i][j] = 0;
        }
    }

    // Precompute constants
    const int num_threads = WORK_GROUP_M * WORK_GROUP_N;
    const int tileA_size = TILE_M * TILE_K;
    const int tileB_size = TILE_K * TILE_N;

    // Loop over k-dimension in tiles
    for (int64_t kt = 0; kt < k; kt += TILE_K) {
        // Optimized cooperative loading - coalesced and vectorized pattern
        // Load tile A using flat indexing for better coalescing
        #pragma unroll 2
        for (int idx = tid; idx < tileA_size; idx += num_threads) {
            int i = idx / TILE_K;
            int j = idx % TILE_K;
            int row = block_m + i;
            int col = kt + j;
            scalar_t val = 0;
            if (row < m && col < k) {
                val = A[row + col * lda];
            }
            tileA_ptr[idx] = val;
        }

        // Load tile B using flat indexing for better coalescing
        #pragma unroll 2
        for (int idx = tid; idx < tileB_size; idx += num_threads) {
            int i = idx / TILE_N;
            int j = idx % TILE_N;
            int row = kt + i;
            int col = block_n + j;
            scalar_t val = 0;
            if (row < k && col < n) {
                val = B[row + col * ldb];
            }
            tileB_ptr[idx] = val;
        }

        item.barrier(sycl::access::fence_space::local_space);

        // Compute using register blocking
        #pragma unroll
        for (int kk = 0; kk < TILE_K; ++kk) {
            scalar_t reg_a[REG_M];
            scalar_t reg_b[REG_N];

            // Load from local memory to registers
            #pragma unroll
            for (int i = 0; i < REG_M; ++i) {
                reg_a[i] = tileA_ptr[(tid_m + i * WORK_GROUP_M) * TILE_K + kk];
            }

            #pragma unroll
            for (int j = 0; j < REG_N; ++j) {
                reg_b[j] = tileB_ptr[kk * TILE_N + (tid_n + j * WORK_GROUP_N)];
            }
            // Compute outer product
            #pragma unroll
            for (int i = 0; i < REG_M; ++i) {
                #pragma unroll
                for (int j = 0; j < REG_N; ++j) {
                    reg_c[i][j] += reg_a[i] * reg_b[j];
                }
            }
        }

        item.barrier(sycl::access::fence_space::local_space);
    }

    // Write results back to global memory with optimized pattern
    scalar_t alpha_val = alpha;
    scalar_t beta_val = beta;

    #pragma unroll
    for (int i = 0; i < REG_M; ++i) {
        #pragma unroll
        for (int j = 0; j < REG_N; ++j) {
            int row = block_m + tid_m + i * WORK_GROUP_M;
            int col = block_n + tid_n + j * WORK_GROUP_N;
            if (row < m && col < n) {
                int idx = row + col * ldc;
                if (beta_val == scalar_t(0)) {
                    C[idx] = alpha_val * reg_c[i][j];
                } else {
                    C[idx] = alpha_val * reg_c[i][j] + beta_val * C[idx];
                }
            }
        }
    }
}

void gemm(...)
{
    (omission)
    sycl_queue.submit([&](sycl::handler& h) {
        (omission)
        h.parallel_for(
            sycl::nd_range<2>(global_size, local_size),
            [=](sycl::nd_item<2> item) {
                scalar_t* tileA_ptr = tileA.template
get_multi_ptr<sycl::access::decorated::no>().get();
                scalar_t* tileB_ptr = tileB.template
get_multi_ptr<sycl::access::decorated::no>().get();
                gemm_kernel_nn_opt(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc,
                                   item, tileA_ptr, tileB_ptr);
            });
    });
    (omission)
}
```

Benchmark Scores





- **Level 2:**
  - **Achieved performance comparable to or exceeding MKL** in multiple cases.
  - Likely due to MKL under-optimization or the generated code being specialized for specific options.
- **Level 3:**
  - Performance fell short of MKL.
  - **High Variance:** Significant performance fluctuation observed across trials (EX1-EX4), indicating stochasticity in code generation.

Conclusion

- **Feasibility:** Validated that Claude Code can generate functional SYCL kernels for Intel GPUs from simple BLAS codes for CPUs
- **Performance Variance:** Performance varied across trials, ranging from MKL-equivalent speeds to significantly lower efficiency.
- **Future Outlook:** Highlights the potential for LLMs to reduce vendor lock-in and support hardware diversity. Future work will investigate capabilities for generating more practical applications.

References

[1] C. Trott et al. "Kokkos 3: Programming Model Extensions for the Exascale Era." IEEE Transactions on Parallel and Distributed Systems. 2022
[2] D. Mukunoki et al. "Performance Evaluation of General Purpose Large Language Models for Basic Linear Algebra Subprograms Code Generation." arXiv:2507.04697. 2025
[3] P. Valero-Lara et al. "ChatBLAS: The First AI-Generated and Portable BLAS Library. " SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2024
[4] Q. Zhou et al. "QiMeng-GEMM: Automatically Generating High-Performance Matrix Multiplication Code by Exploiting Large Language Models." AAAI-25 39m 21. 2025
(1) https://icl.bitbucket.io/blaspp/