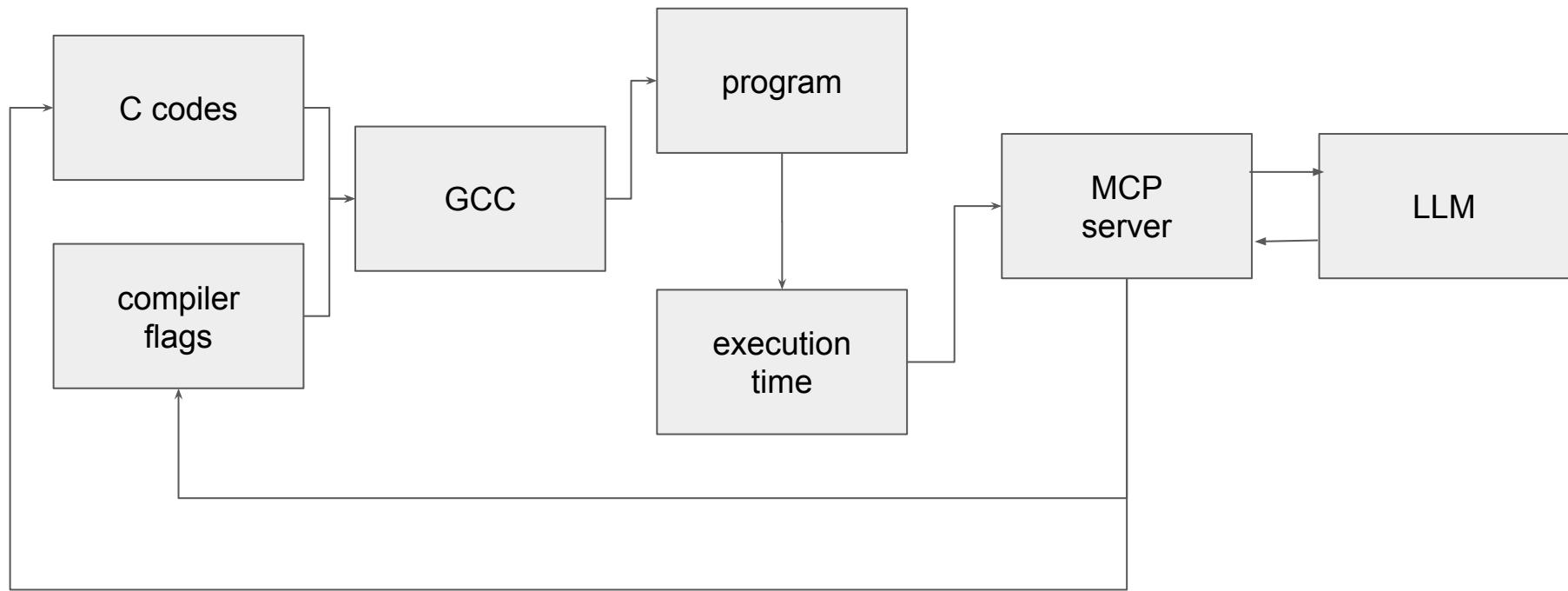


# MCP for Tuning C Benchmark

# Motivation

- Naively written C program usually cannot achieve the peak performance on CPUs
- Need to tune the codes and compiler flags!
- Coding assistants can handle this task with enough directions, but an MCP server allows us to encapsulate the compilation and measurement process.
  - Potentially run on a remote server.
- Test case: STREAM Benchmark
  - Measures the memory bandwidth of a given machine.
  - Requires very careful code optimization to get good performance.

# Workflow



# Tools



**make\_stream\_benchmark** -  
Compile STREAM benchmark with  
specified compiler and flags.



**test\_correctness** - Verify  
benchmark produces correct  
results (returns JSON with  
PASS/FAIL).



**test\_speed** - Measure memory  
bandwidth for each kernel (returns  
JSON with GB/s for copy, scale,  
add, triad).



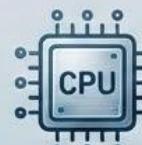
**make\_clean** - Remove compiled  
artifacts for fresh build.



**get\_source\_code** - Retrieve  
default STREAM benchmark C  
source code.



**make\_custom\_benchmark** -  
Compile with custom kernel  
implementations (optional:  
allocation\_code, copy\_code,  
scale\_code, add\_code,  
triad\_code).



**list\_cpu\_info** - Get CPU and OS  
information (architecture, cores,  
processor model, etc.).

# Demo Result - M4 Mac Pro

- Ask claude desktop to optimize the code.

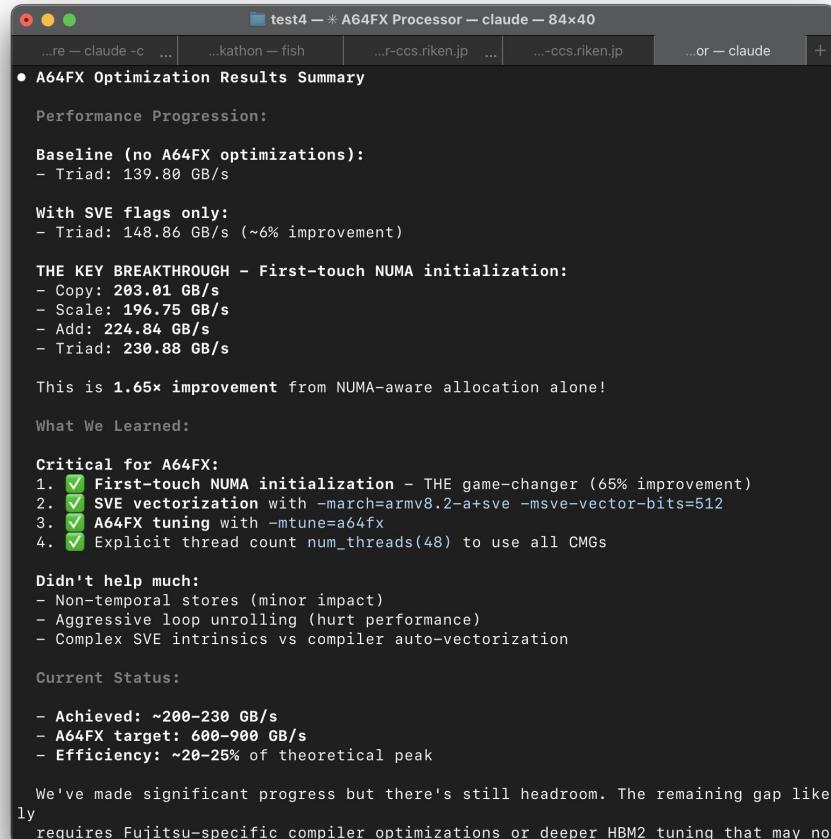
Optimization Results		
Baseline (Single-threaded)		
Kernel	Bandwidth (GB/s)	
Copy	104.09	
Scale	103.42	
Add	114.68	
Triad	113.98	

Optimized (Multi-threaded + SIMD)		
Kernel	Bandwidth (GB/s)	Improvement
Copy	231.31	2.22x
Scale	223.69	2.16x
Add	247.39	2.16x
Triad	245.05	2.15x

# Demo Result - R-CCS Cloud

- Run a singularity container of our MCP server on the remote server.
  - R-CCS cloud, which through slurm job manager can give us access to an A64FX node (just like Fugaku)
- MCP Client running locally.
- If you can reserve a machine, and have a narrow focus of optimization, this can be a good approach.



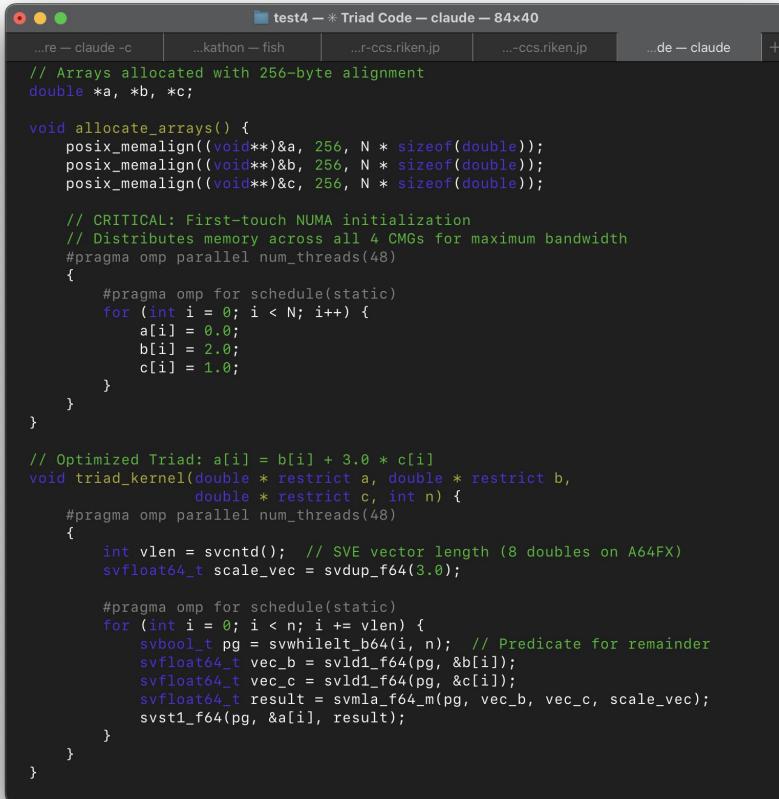
The screenshot shows a terminal window titled "test4 — \* A64FX Processor — claude — 84x40". The window displays performance results for an A64FX processor. Key sections include:

- Performance Progression:**
  - Baseline (no A64FX optimizations):**
    - Triad: 139.80 GB/s
  - With SVE flags only:**
    - Triad: 148.86 GB/s (~6% improvement)
  - THE KEY BREAKTHROUGH – First-touch NUMA initialization:**
    - Copy: 203.01 GB/s
    - Scale: 196.75 GB/s
    - Add: 224.84 GB/s
    - Triad: 230.88 GB/s
- This is 1.65x improvement from NUMA-aware allocation alone!**
- What We Learned:**
  - Critical for A64FX:**
    - ✓ First-touch NUMA initialization – THE game-changer (65% improvement)
    - ✓ SVE vectorization with `-march=armv8.2-a+sve -msve-vector-bits=512`
    - ✓ A64FX tuning with `-mtune=a64fx`
    - ✓ Explicit thread count `num_threads(48)` to use all CMGs
- Didn't help much:**
  - Non-temporal stores (minor impact)
  - Aggressive loop unrolling (hurt performance)
  - Complex SVE intrinsics vs compiler auto-vectorization

- Current Status:**
- Achieved: ~200-230 GB/s
- A64FX target: 600-900 GB/s
- Efficiency: ~20-25% of theoretical peak
- We've made significant progress but there's still headroom. The remaining gap likely requires Fujitsu-specific compiler optimizations or deeper HBM2 tuning that may no**

# Discussion

- The model could not get peak performance out of A64FX despite trying many things (something like 25% of peak). Telling it to think about the CPU it was using kept it motivated.
  - Much better than I could do.
- Sometimes the LLM thinks “Ok I will just write the code and compile it myself”, not knowing that it is only allowed to work with the MCP server.
  - Need make it clear it doesn’t have direct access to the machine.



The screenshot shows a terminal window titled "test4 -- \* Triad Code -- claude -- 84x40". The window contains C code for a triad computation. The code includes comments indicating memory alignment (256-byte) and NUMA initialization. It defines arrays a, b, and c, initializes them with values 0.0, 2.0, and 1.0 respectively, and then performs an optimized triad operation where each element of array a is updated to be the sum of its original value and three times its corresponding value from array c.

```
// Arrays allocated with 256-byte alignment
double *a, *b, *c;

void allocate_arrays() {
    posix_memalign((void**)&a, 256, N * sizeof(double));
    posix_memalign((void**)&b, 256, N * sizeof(double));
    posix_memalign((void**)&c, 256, N * sizeof(double));
}

// CRITICAL: First-touch NUMA initialization
// Distributes memory across all 4 CMGs for maximum bandwidth
#pragma omp parallel num_threads(48)
{
    #pragma omp for schedule(static)
    for (int i = 0; i < N; i++) {
        a[i] = 0.0;
        b[i] = 2.0;
        c[i] = 1.0;
    }
}

// Optimized Triad: a[i] = b[i] + 3.0 * c[i]
void triad_kernel(double * restrict a, double * restrict b,
                  double * restrict c, int n) {
    #pragma omp parallel num_threads(48)
    {
        int vlen = svrndt(); // SVE vector length (8 doubles on A64FX)
        svfloat64_t scale_vec = svdup_f64(3.0);

        #pragma omp for schedule(static)
        for (int i = 0; i < n; i += vlen) { // Predicate for remainder
            svbool_t pg = svwhilet_b64(i, n);
            svfloat64_t vec_b = svld1_f64(pg, &b[i]);
            svfloat64_t vec_c = svld1_f64(pg, &c[i]);
            svfloat64_t result = svmla_f64_m(pg, vec_b, vec_c, scale_vec);
            svst1_f64(pg, &a[i], result);
        }
    }
}
```