MLIR-based compiler backend for SimpliC

Albin Nyström
D21, Lund University, Sweden
al4140ny-s@student.lu.se

Aron Somi
C21, Lund University, Sweden
ar6781so-s@student.lu.se

Abstract

LLVM is a common compiler framework that provides a low-level intermediate representation (IR) and supports many hardware instructions sets. LLVM is effective for traditional compiler workloads, but as LLVM IR operates at a low level of abstraction expressing domain specific optimizations becomes difficult. MLIR (Multi-Level Intermediate Representation) is an evolution of LLVM with a higher level IR enabling representation of programs at multiple levels of abstraction. This preserves semantic information necessary for optimizing common workloads used in machine learning and linear algebra.

In this paper we present our experience of adding a MLIR-based backend to a C-like language. The compiler lowers programs through several MLIR dialects before targeting LLVM. Our results show that a simple MLIR based backend can leverage the optimizations of MLIR to produce a matrix multiplication program that is faster than code produced by GCC or Clang.

1 Introduction

The x86 instruction set has long dominated compute-intensive programs, as a result it is supported by a mature ecosystem of programming languages, compilers and hardware. A central component of this ecosystem is the LLVM compiler framework, which uses its own intermediate representation (IR) as a common ground between compilers and hardware. Compilers can target LLVM IR, and LLVM can in turn lower this IR to x86 or to other architectures such as ARM, MIPS, or RISC-V. LLVM's backend is also modular in design to make it possible for developers to add new backends for their specific hardware.

Despite its flexibility, it is proving difficult for LLVM to support modern deep learning workloads and hardware. The hardware and its instruction set often relies on compiler optimizations on a higher abstraction level than what LLVM provides, so when code is lowered to LLVM's IR information that is important for efficient computation on this hardware is lost. For example, the ability of detecting and replacing matrix multiplication operations with pre-written kernels when targeting GPUs. This is one of the limitations of LLVM that MLIR aims to solve. All LLVM code is a subset of MLIR, but MLIR importantly extends LLVM with high-level concepts such as regions, dialects and tensors, among others. These high-level concepts also make MLIR far more modular and adaptable than LLVM and allows for a wider support of hardware. For example, *E. Tiotto et al* use MLIR in a compiler for a higher-level language (SYCL), used in heterogeneous computing, to achieve 4.3x better performance than an equivalent LLVM implementation. [5]

MLIR is still relatively young and does not yet have an as established position as LLVM. The barrier to entry is quite high, as there is a general lack of tutorials and introductory material. With this paper we aim to contribute to that space by sharing our experience of adding a MLIR backend to a compiler for a simple C-like language, called SimpliC, used in the compilers course at Lund University. The original SimpliC compiler generated only x86, we extend it to also generate MLIR code. Our compiler lowers SimpliC through increasingly lower-level MLIR forms until reaching a LLVM-equivalent representation, which is then translated to LLVM and compiled with Clang to generate an executable.

As part of the evaluation we show that the MLIR-based backend can execute a wide range of test programs. We also explore a targeted optimization for matrix multiplication, where leveraging MLIR's built in loop tiling achieves a 60% performance increase over GCC compiled with `-O3` and `-ffast-math`, the second best performer.

2 Background

2.1 Multi-Level Intermediate Representation - MLIR

Software fragmentation is an ongoing problem in machine learning frameworks and compiler development. There are many different compilers and intermediate representations with different infrastructures and standards [3]. This complicates running heterogeneous systems, where different compilers and IRs then have to interact, which is a common occurrence in machine learning where training and inference pipelines rely on specialized accelerators.

MLIR is a reusable and extensible compiler framework designed to support multiple abstraction levels within a single infrastructure. More concretely this means that MLIR can represent a program at different stages of detail allowing a program to move from a high-level to a low-level representation while staying inside the same framework. MLIR provides tools for a multi-level approach where developers can package domain or language specific concepts in "dialects", making it easier to keep a common IR with common solutions where possible. MLIR also supports universal and

Course paper, EDAN70, Lund University, Sweden
January 14, 2026.

target specific optimization within the same compiler making it easy to deploy across a heterogeneous set of hardware.

2.2 Important MLIR concepts

In this section we introduce some important concepts in MLIR, such as dialects, rewrite rules, and static single assignment (SSA).

Dialects - A MLIR dialect is a group of operations and rewrite rules that essentially represent a layer in MLIR. Using dialects developers can define their own domain specific constructs, types and rewrite rules. Dialects allow MLIR to represent a program at different abstraction levels within the same framework. An example is the `linalg` dialect that describes high level linear algebra semantics, such as `matmul`. Code written using `linalg` can be transformed to lower level dialects that describe the same computation using loops, memory access or LLVM instructions.

SSA - MLIR uses Static Single Assignment (SSA) form for defining values ensuring that each value is defined exactly once. SSA eliminates variable mutation, allowing for easier analysis and optimization during compilation.

Operations - Operations are statements that can contain blocks or structured control flow concepts like loops. An operation defines its operands and results on SSA form.

Rewrite rules - Lowering between abstraction levels in MLIR happens with rewrite rules which define how one operation is expressed in another. Defining a rewrite rule is done declaratively where a pattern and a replacement for that pattern is defined. For example, MLIR can represent a matrix multiplication at a high level with the matmul operation in the `linalg` dialect using `linalg.matmul`. Using rewrite rules the `linalg` dialect can be lowered into a dialect where loop based representation of the same matrix multiplication is expressed.

Progresive Lowering - For each level of abstraction used in the compiler there exists a corresponding transformation leading from that level to a lower level of abstraction. Each pass will apply rewrite rules, perform optimizations or convert code between dialects. More abstract language concepts will gradually be removed in favour of generic instructions. In this project, the final step is lowering to the LLVM dialect, translating to LLVM IR and compiling with Clang to generate an executable.

2.3 SimpliC

During the course EDAN65 at Lunds University the students create a compiler for a subset[1] of C called SimpliC. SimpliC supports many common language constructs like variable declarations, variable assignment, functions and function calls, `if-` and `while` statements and arithmetic expressions. All variables in SimpliC are integers, but expressions also support booleans.

[1]SimpliC is not quite a subset, as the language also includes the predefined functions `print` and `read`

2.4 Optimizations concepts relevant to MLIR

In this section we introduce optimization concepts relevant to our solution and MLIR. This includes loop interchange, memory locality, cache utilization, cache lines, vectorization and loop tiling.

Loop interchange is the process of swapping the order of nested loops, preserving the meaning of the computation to improve **memory locality** and **data access patterns** [2]. In C, two-dimensional arrays are stored in **row-major order**. Meaning that the whole row `a[0][0]` → `a[0][n]` is stored consecutively in memory before the first element in the second row `a[1][0]` is found. This layout means that accessing memory by increasing the column index instead of the row index fetches data that is stored sequentially in memory. A demonstration of loop interchange is seen in Listing 1. The CPU fetches data from memory in blocks called **cache lines** typically of 64 bytes in size. As a float is 4 bytes meaning 16 floats can be loaded at each memory access. Using sequentially stored and accessed memory results in much fewer cache misses as each consecutive element loaded will be used. In contrast if memory is not sequentially used, only 4 out of 64 bytes in each cache line is used, and around 94% of the bandwidth is wasted. Loop interchange is a common compiler optimization used to improve memory bound computations, like matrix multiplications. Some compilers (GCC) can automatically apply this transformations while other cannot (Clang).

Listing 1. An example of loop interchange. Changing loop order improves memory access and increases performance for matrix multiplication

```
//A is accessed row-by-row, efficient
//B is accessed column-by-column, inefficient
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
           C[i][j] += A[i][k] * B[k][j];
        }
    }
}

//A and B are both accessed row-by-row, efficient
for (int i = 0; i < N; i++) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
          C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Loop tiling (also known as blocking) is a transformation that restructures a loop so that it processes data in smaller and more cache optimized ways, rather than operating on entire rows and columns at once. The goal is that once data has been loaded into cache, it is reused as much as possible before being swapped out [2]. Processors have a hierarchy of caches, each with limited capacity. In many computations, the innermost loop often exceeds the capacity of the cache that cause eviction and reloading of data from slow memory. Loop tiling breaks the large inner loops into smaller iteration spaces called **tiles** that fit into cache. An example of loop tiling is shown in Listing 2. The naive loop may require data from separate memory regions and if the arrays are large they do not fit into cache causing frequent cache misses. In contrast the tiled version splits the arrays into portions that fit into cache and reused many times before being evicted. Each tile computes partial sums for the output matrix, and the final value accumulates over all tiles along the k-dimension.

Listing 2. An example of loop tiling a matmul to increase cache reuse and performance

```
//Naive matmul
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         C[i][j] += A[i][k] * B[k][j];


//Loop tiling matmul
for (int ii = 0; ii < N; ii += Ti)
   for (int jj = 0; jj < N; jj += Tj)
      for (int kk = 0; kk < N; kk += Tk)
         for (int i = ii; i < ii + Ti; i++)
            for (int j = jj; j < jj + Tj; j++)
               for (int k = kk; k < kk + Tk; k++)
                  C[i][j] += A[i][k] * B[k][j];
```

3 Solution

3.1 Vectors, malloc free and types

To be able to perform relevant machine learning operations SimpliC had to be extended to support vectors. The vectors were made to be heap allocated. The type system of SimpliC has also been extended to support floats in addition to integers, and corresponding pointer-types to these types.

3.2 SimpliC to MLIR

Next, we implemented the translation between SimpliC and MLIR. Working with and keeping track of SSAs can be quite complicated. We instead made use of the `memref` dialect for variable handling, mainly using `load` and `store` to represent accessing and modifying a variable stored on the stack.

Expressions generally had a one-to-one MLIR equivalent in the `arith` dialect and were simple to implement. Since the operations in `arith` only take SSAs we had to implement some overhead handling of our `memrefs`, like `memref.load` into SSAs before the operations, and store them with `memref.store` afterwards.

To represent `if`, `while` and `return` statements we used the `cf` (control flow) dialect, which support jumping and branching between different blocks. Because early returns are not allowed in MLIR we always generate a return block at the end of the function, as a substitute for an early return we instead jump to this block.

3.3 Running passes on generated MLIR

We translate our sidestepping of the SSAs into actual SSAs by using a pass called `mem2reg` that translates `memref` into SSAs where possible. We also run `canonicalize` to merge the unnecessary branching, which our if, while, and return statements often generate.

3.4 Optimization Using MLIR

Once the core SimpliC → MLIR translation was complete, we explored what kinds of optimizations MLIR could provide. As matrix multiplication is a common benchmark for evaluating optimizations, we developed several matmul variants from naive implementations to loop interchange versions to use as test programs.

Because our translation targets relatively low-level MLIR (primarily `memref`, `arith`, and `cf`), many of MLIR's higher level transformation passes cannot be used. To take advantage of MLIR's passes we extended SimpliC with a built in `matmul` function that lowers directly to the high level `linalg.matmul` operation. With `linalg.matmul` available we could use the affine transformation pipeline and take advantage of the built in loop tiling pass, `-affine-loop-tile`. This experiment shows how MLIR's higher level abstractions enable powerful optimizations.

3.5 Basic SimpliC to MLIR example

This section shows the steps we take to generate MLIR code from a SimpliC input program. Listing 3 shows a SimpliC program with two functions. Listing 4 shows the generated MLIR code for Listing 3.

Listing 3. An example of an input SimpliC program that the compiler can translate to MLIR

```
int add(int a, int b) {
   return a + b;
}
int main() {
   int a = 3;
   int b = 4;
   int sum = add(a, b);
   return sum;
}
```

Listing 4. Generated MLIR code from the input program in Listing 3

```
func.func @add(%a0 : i32, %b0 : i32) -> i32{
   %0 = memref.alloca() : memref<i32>
   memref.store %a0, %0[] : memref<i32>
   %1 = memref.alloca() : memref<i32>
   memref.store %b0, %1[] : memref<i32>
   %2 = memref.load %0[] : memref<i32>
   %3 = memref.load %1[] : memref<i32>
   %4 = arith.addi %2, %3 : i32
   cf.br ^return(%4 : i32)
   ^return(%ret : i32):
   func.return %ret : i32
}

func.func @main() -> i32{
   %a10 = memref.alloca() : memref<i32>
   %0 = arith.constant 3 : i32
   memref.store %0, %a10[] : memref<i32>

   %b11 = memref.alloca() : memref<i32>
   %1 = arith.constant 4 : i32
   memref.store %1, %b11[] : memref<i32>

   %sum12 = memref.alloca() : memref<i32>
   %2 = memref.load %a10[] : memref<i32>
   %3 = memref.load %b11[] : memref<i32>
   %4 = func.call @add(%2,%3) : (i32, i32) -> i32
   memref.store %4, %sum12[] : memref<i32>

   %5 = memref.load %sum12[] : memref<i32>
   cf.br ^return(%5 : i32)
   ^return(%ret : i32):
   func.return %ret : i32
}
```

The generated code might be hard to understand at first glance, it is made to be easy to generate and not easy to read. To get a more human readable version we run the passes `-mem2reg`[2] (converting loads/stores to SSAs) and `-canonicalize`[3] (folds trivial branching etc.). The result of running these passes can be seen in Listing 5.

Listing 5. MLIR code after running passes mem2reg and canonicalize

```
module {
  func.func @add(%arg0: i32, %arg1: i32) -> i32{
    %0 = arith.addi %arg0, %arg1 : i32
    return %0 : i32
  }
  func.func @main() -> i32{
    %c3_i32 = arith.constant 3 : i32
    %c4_i32 = arith.constant 4 : i32
    %0 = call @add(%c3_i32, %c4_i32) : (i32, i32) ->
         i32
    return %0 : i32
  }
}
```

4 Evaluation

The purpose of the evaluation is to gather knowledge about the *technical performance* and *capabilities* of our SimpliC -> MLIR backend. As one of the goals of this projects is to understand if a simple MLIR solution can produce competitive code, we benchmark our pipeline against GCC and Clang. These experiments show in what situations MLIR performs well, where it falls short and whether MLIR can serve as a compilation target without extensive knowledge of optimizing compilers. With this we aim to answer if a simple MLIR backend can generate code that is competitive with a traditional compiler.

4.1 Performance evaluation

As SimpliC is not a strict subset of C, the functions `print`, `read`, `malloc`, `matmul`, and `free` must be provided by the runtime. This means that the source code used for the benchmarking is not identical across compilers. Polygeist, a C to MLIR compiler and optimization tool [4], GCC and Clang compiles source code that begins with `#include <stdlib.h>` and our compiler first translates to MLIR and then lowers it to LLVM IR. The commands for compiling the programs with Polygeist, GCC and Clang are shown in Listing 6.

Listing 6. Compilation using GCC, Clang, and Polygeist

```
# GCC
gcc -O3 -march=native -ffast-math matmul.c -o
    matmul_gcc

# Clang
clang -O3 -march=native -ffast-math matmul.c -o
    matmul_clang

# Polygeist
cgeist matmul.c --function=* -O3 -emit-llvm -S \
   | clang -x ir - -O3 -o matmul_polygeist
```

Compilation of SimpliC→MLIR consist of a few additional steps shown bellow and done using the code in Listing 7.

[2]https://mlir.llvm.org/docs/Passes/#-mem2reg
[3]https://mlir.llvm.org/docs/Passes/#-canonicalize

1. Compile SimpliC to MLIR
2. Apply MLIR transformations passes
3. Lower MLIR to LLVM IR
4. Use Clang to generate executable

Listing 7. SimpliC → MLIR → LLVM compilation pipeline

```
# Generate MLIR from SimpliC and lower to LLVM IR
java -jar simplc.jar source.simplic \
  | mlir-opt --convert-scf-to-cf --mem2reg \
           --expand-strided-metadata \
           --convert-math-to-llvm --convert-arith-to-
               llvm \
           --convert-func-to-llvm --convert-cf-to-
               llvm \
           --finalize-memref-to-llvm --reconcile-
               unrealized-casts \
  | mlir-translate --mlir-to-llvmir \

# Use clang to produce final executable
   clang -x ir - -O3 -o program_simplic.elf
```

Once the executable program is produced the performance evaluation can begin. Runtime variability due factors like OS scheduling, cache effects and background processes is not uncommon. To only report a single value of a measurement would be to hide this variability so we report a 95% confidence interval computed from repeated executions of the same benchmark. Another benefit of using confidence intervals is that two experiments with similar means and overlapping confidence intervals likely don't represent a performance difference.

4.1.1 Benchmarking setup

The benchmarks were performed on a dedicated machine with the hardware and software shown in Table 1. The benchmarks were executed without any user involvements or background tasks running.

Source	Description
CPU	Intel(R) i7-1065G7 CPU @ 1.30GHz
Cores / Threads	8 / 16
Memory	16 GiB
OS	Ubuntu 24.04.3 LTS
GCC version	13.3.0
Clang version	20.0.0
MLIR version	20.0.0

Table 1. Benchmarking setup

Each benchmarking program was compiled with the commands shown in Listing 6 and 7. The clock-time was measured from process start to finish using a python script. To mitigate processor cold start effects, one warm-up run was executed before starting the 20 measurement runs. The results of the benchmarking can be seen in Table 2. First a naive 2048X2048 matmul was benchmarked implemented as three nestled while loops. To leverage MLIR's optimization the built in `matmul` using the dialect and operation `linalg.matmul` is instead used, these results can also be seen in Table 2. Each program can be found in Appendix A.

Program	Compiler	Avg. time (s)	95% CI (s)
matmul_naive	GCC	6.48	[6.32, 6.64]
	Clang	35.20	[34.89, 35.51]
	Polygeist	35.80	[35.54, 36.04]
	Our MLIR	35.45	[35.20, 35.71]
matmul_better	GCC	1.19	[1.17, 1.21]
	Clang	1.70	[1.64, 1.75]
	Polygeist	0.86	[0.85, 0.87]
	Our MLIR	0.73	[0.72, 0.74]

Table 2. Comparison between compilers on 2048 `float matmul`

4.1.2 Performance Results and Discussion

Table 2 summarizes the execution times for two matrix multiplication benchmarks. `Matmul_naive` uses a triple nested loop whereas `matmul_better` leverages loop interchange for GCC, Clang and Polygeist, and in the case of MLIR it uses the built in loop-tiling optimization.

For `matmul_naive`, GCC is the best performer with an average runtime of 6.48 s, while the MLIR based pipelines all run at around 35 s. This large difference can perhaps be explained by GCC's use of loop interchange on nestled C loops. The loops being while loops is likely also a contributing factor. Clang's default pipeline is more conservative in this matter causing worse cache locality as explained in Section 1.

When we rewrite the program to use a loop-interchanged implementation in `matmul_better` the performance relations change. GCC and Clang both benefit from the improved loop order. GCC improves to 1.19 s and Clang to 1.70 s. Polygeist performs better at 0.86 s, likely due to its use of polyhedral optimizations on the Clang-generated MLIR. Our MLIR backend achieves the fastest execution time at 0.73 s. It is 1.6× speed-up over GCC and a 2.3× speed-up over Clang on this benchmark. Compared to Polygeist, our solution is 1.2× faster using only MLIR's built in passes and no external tools.

The main reason for this improvement is that SimpliC built in `matmul` lowers directly to `linalg.matmul`. This enables the affine transformation pipeline with the `-affine-loop-tiles` pass to use cache friendly tiles without manually restructuring the program. Expressing the operation as a high level MLIR operation allows the compiler to automatically apply optimizations similar to hand tuned optimizations in performance libraries. This ability to combine domain specific operations with generic optimization passes to produce optimized code is one of MLIR's strengths.

4.2 Proof of concept evaluation

To demonstrate that SimpliC → MLIR works we evaluate it on a set of programs that cover different language constructs. The compilers course at Lund University required a large set of test programs, all these are covered, with additions for floats, pointers, heap allocation and memory freeing. More complex programs that the new language features are also tested. One such program is matrix multiplication that contains nestled loops, pointers, dynamic memory allocation, control-flow and floating point arithmetic.

5 Related work

5.1 High Performance Code Generation in MLIR

Bondhugula presents an early case study using MLIR for general matrix multiplication [1]. A comparison between C code using GCC or clang is used as a starting point. The author leverages some MLIR built affine and memref infrastructure in combination with deep domain knowledge of optimization defining custom tiling, packing, unrolling and vectorixation passes, resulting in a compiler that achieves 61 % of the theoretical computation limit on his hardware while GCC -O3 and clang -O3 only achieved 0,6% and 6% respectively. The work showed that MLIR's infrastrucure can be used to develop optimization similar to highly tuned BLIS libraries. In contrast to Bondhugula we restrict ourself to MLIR's off the shelf standard passes, but use a similar approach as the author trying to replicate the optimization passes they use.

5.2 Polygeist

Polygeist is a project with two aims: (1) it traverses the Clang AST to emit MLIR code translated from C/C++, and (2) it supports polyhedral optimizations via external tools, benefiting from MLIR's high-level affine and structured dialects [4]. Polygeist can generate code that is optimized.

While Polygeist is a C/C++ to MLIR translator, it uses the Clang AST for translation into MLIR. In contrast, our approach relies on our own AST. Furthermore, whereas Polygeist leverages external tools like Pluto and OpenScop to support polyhedral optimization, we use only MLIR's built in optimization passes.

6 Conclusion

In this report we presented our implementation of an MLIR-based backend for SimpliC. The backend translates SimpliC programs to MLIR and then lowers them through multiple MLIR dialects before targeting LLVM. Our results show that even a relatively simple MLIR pipeline can generate correct and competitive code. Our benchmarks demonstrate that MLIR's multi-level abstractions can be particularly effective for optimizing cache-sensitive computations, like a matrix multiplication code. By lowering matrix multiplication directly to `linalg.matmul`, we were able to apply built-in loop tiling and achieve better performance than GCC or Clang on an equivalent benchmark. This demonstrates, albeit in a very specific example, the advantage MLIR has in preserving high-level semantic information during compilation. Overall, our experience suggests that MLIR is a viable and powerful backend for new or experimental languages. While there is a learning curve, the ability to leverage MLIR's modular design and optimization infrastructure makes it an appealing compilation target.

Acknowledgments

We would like to thank our supervisor Anton Risberg Alaküla for his guidance, support and patience throughout this project. We finally did remove the half-baked memory allocator!

References

[1] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. *CoRR* abs/2003.00532 (2020). arXiv:2003.00532 https://arxiv.org/abs/2003.00532

[2] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. 2017. Chapter 5 - Source code transformations and optimizations. In *Embedded Computing for High Performance*, João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz (Eds.). Morgan Kaufmann, Boston, 137–183. https://doi.org/10.1016/B978-0-12-804189-5.00005-3

[3] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[4] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 45–59. https://doi.org/10.1109/PACT52795.2021.00011

[5] Ettore Tiotto, Víctor Pérez, Whitney Tsang, Lukas Sommer, Julian Oppermann, Victor Lomüller, Mehdi Goli, and James Brodman. 2024. Experiences Building an MLIR-Based SYCL Compiler. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 399–410. https://doi.org/10.1109/CGO57630.2024.10444866

A Matmul programs

These are the programs used for benchmarking. Listing 8 shows the first test program performing a naive matmul using while loops and no loop interchange in C. Listing 9 shows the equivalent program but in SimpliC. Listing 10 shows the loop interchanged C implementation of matmul using for loops instead of while loops. Listing 11 Shows the SimpliC optimized matmul, that directly uses a built in `matmul` function.

Listing 8. Naive C matmul

```c
#include <stdlib.h>
int idx(int row, int col) {
    int N = 2048;
    return row * N + col;
}
int main() {
    int N = 2048;
    int totalSize = N * N;

    float *A = malloc(totalSize * sizeof(float));
    float *B = malloc(totalSize * sizeof(float));
    float *C = malloc(totalSize * sizeof(float));

    int i = 0;
    while (i < totalSize){
        A[i] = 2.0f;
        B[i] = 2.0f;
        C[i] = 0.0;

        i = i + 1;
    }
    int row = 0;
    while (row < N){
        int col = 0;
        while (col < N){
            float sum = 0.0;
            int k = 0;
            while (k < N){
                sum = sum + A[idx(row, k)] * B[idx(k, col)];
                k = k + 1;
            }
            C[idx(row, col)] = sum;
            col = col + 1;
        }
        row = row + 1;
    }
    float check = 0.0;
    check = C[idx(100, 100)];

    int exitcode;
    if(check == 0.0) {
        exitcode = 42;
    } else {
        exitcode = 41;
    }
    free(A);
    free(B);
    free(C);
    return exitcode;
}
```

Listing 9. Naive simpliC matmul

```

int idx(int row, int col) {
    int N = 2048;
    return row * N + col;
}

int main() {
    int N = 2048;
    int totalSize = N * N;

    float* A = malloc(totalSize * 4);
    float* B = malloc(totalSize * 4);
    float* C = malloc(totalSize * 4);

    int i = 0;
    while (i < totalSize) {
        A[i] = 2.0;
        B[i] = 2.0;
        C[i] = 0.0;

        i = i + 1;
    }
    int row = 0;
    while (row < N) {
        int col = 0;
        while (col < N) {
            float sum = 0.0;
            int k = 0;
            while (k < N) {
                sum = sum + A[idx(row, k)] * B[idx(k, col)];
                k = k + 1;
            }
            C[idx(row, col)] = sum;
            col = col + 1;
        }
        row = row + 1;
    }
    float check = 0.0;
    check = C[idx(100, 100)];

    int exitcode;
    if(check == 0.0){
        exitcode = 42;
    } else {
        exitcode = 41;
    }
    free(A);
    free(B);
    free(C);
    return exitcode;
}
```

Listing 10. Better C matmul

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    int N = 2048;
    float EXPECTED = N * 4.0f;

    int totalSize = N * N;

    float *Abuf = malloc(totalSize * sizeof(float));
    float *Bbuf = malloc(totalSize * sizeof(float));
    float *Cbuf = malloc(totalSize * sizeof(float));

    float (*A)[2048] = (float (*)[2048])Abuf;
    float (*B)[2048] = (float (*)[2048])Bbuf;
    float (*C)[2048] = (float (*)[2048])Cbuf;

    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            A[i][j] = 2.0f;
            B[i][j] = 2.0f;
            C[i][j] = 0.0f;
        }
    }

    // Loop-interchanged matmul: i-k-j instead of i-j-k
    for (int i = 0; i < N; i++){
        for (int k = 0; k < N; k++){
            float aik = A[i][k];
            for (int j = 0; j < N; j++){
                C[i][j] += aik * B[k][j];
            }
        }
    }

    float check = C[100][100];
    int exitcode = (check == EXPECTED) ? 42 : 41;

    free(Abuf);
    free(Bbuf);
    free(Cbuf);

    return exitcode;
}
```

Listing 11. Optimized SimpliC matmul

```
int main()
{
    int N = 2048;
    float EXPECTED = 2048.0 * 4.0;
    float* A = malloc(N*N*4);
    float* B = malloc(N*N*4);
    float* C = malloc(N*N*4);

    int totalSize = N * N;
    int i = 0;

    while (i < totalSize) {
        A[i] = 2.0;
        B[i] = 2.0;
        C[i] = 0.0;
        i = i + 1;
    }

    int exitcode = 0;
    matmul(N,A,B,C);

    float check = 0.0;
    check = C[100];


    if(check == EXPECTED){
        exitcode = 42;
    } else {
        exitcode = 41;
    }

    free(A);
    free(B);
    free(C);

    return exitcode;
}
```