Matrix Multiplication Compiler for a Custom 24-bit ISA

Aditya 22BAI1235[1] and Affan Ahmed 22BRS1141[1]

Vellore Institute of Technology (VIT), Chennai, India
{aditya.2022a,affan.ahmed2022}@vitstudent.ac.in

Abstract. This document presents a project report for a custom compiler designed to translate matrix multiplication (both matrix–matrix and matrix–vector) operations written in C/C++ into a 24-bit Instruction Set Architecture (ISA). The report is structured as follows:

- **Analysis and Design of the Algorithm**
- **Source Code of the Solution with Explanations**
- **Output (Screenshots, Logs, and Generated Files)**

Placeholders are provided for images and output files where necessary.

Keywords: Compiler · Matrix Multiplication · Custom ISA · Code Generation

1 Analysis and Design of Algorithm

This section explains how the custom compiler transforms C/C++ code containing matrix multiplication into a 24-bit ISA.

1.1 Compiler Overview

The compiler follows a multi-stage pipeline (see Fig. 1):

1. **Parsing:** Extracts matrix dimensions and operation type (matrix–matrix or matrix–vector) from special markers in the source code (e.g., `// OPERATION: MM_MULT`).
2. **Semantic Analysis:** Validates the dimensions to ensure they match for multiplication.
3. **IR Optimization:** Performs minimal optimizations, such as counting multiply-accumulate (MAC) steps.
4. **Code Generation:** Produces machine instructions conforming to the 24-bit ISA.
5. **Memory Layout Computation:** Assigns base addresses for matrices (A, B, C) or vector V.

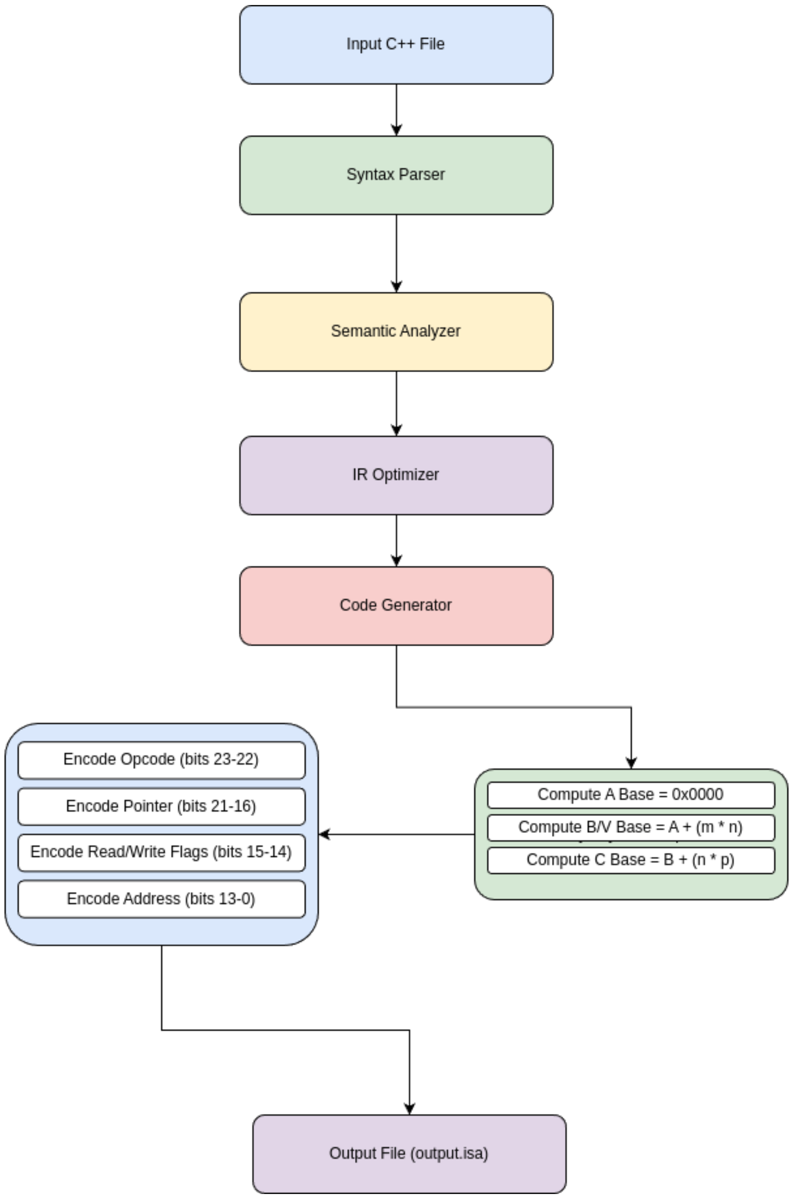

Input C++ File
Syntax Parser
Semantic Analyzer
IR Optimizer
Code Generator
Compute A Base = 0x0000
Compute B/V Base = A + (m * n)
Compute C Base = B + (n * p)
Encode Opcode (bits 23-22)
Encode Pointer (bits 21-16)
Encode Read/Write Flags (bits 15-14)
Encode Address (bits 13-0)
Output File (output.isa)

Fig. 1. High-level pipeline of the compiler operations.

1.2 24-bit ISA Format

Each instruction is encoded into 24 bits as follows:

- **Bits 23–22:** Opcode type (e.g., READ, PROG, EXE, END).
- **Bits 21–16:** Pointer field (used for microcode control or indexing).
- **Bit 15:** Read flag.
- **Bit 14:** Write flag.
- **Bits 13–0:** Address (lower 14 bits).

1.3 Memory Layout

Matrices are placed sequentially in memory:

- Matrix A starts at address `0x0000`.
- Matrix B (or vector V) follows A.
- Matrix C is placed after B.

For example, if matrix A is of size $m \times n$, then B's base address is:

$$\texttt{base\_B} = \texttt{base\_A} + (m \times n)$$

and matrix C's base address is:

$$\texttt{base\_C} = \texttt{base\_B} + (n \times p)$$

where p is the number of columns in matrix B.

2 Source Code of the Solution

Below, the complete source code files are displayed in full. Each section includes an explanation of its purpose, functionality, and a small example excerpt to illustrate how it contributes to the custom ISA architecture.

2.1 compiler_main.c

Purpose: This file is the entry point of the compiler. It handles reading the input C++ file, invoking the parser to extract markers and dimensions, then sequentially calls the semantic analyzer, IR optimizer, and code generator to produce the final `output.isa` file.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "syntax_parser.h"
#include "semantic_analyzer.h"
#include "ir_optimizer.h"
```

```c
#include "isa_generator.h"

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <source_file.cpp>\n", argv
            [0]);
        return 1;
    }

    // Open the input C++ source file.
    FILE *fp = fopen(argv[1], "r");
    if (!fp) {
        perror("Error opening source file");
        return 1;
    }

    // Determine the file size.
    fseek(fp, 0, SEEK_END);
    long file_size = ftell(fp);
    rewind(fp);

    // Allocate buffer and read the file contents.
    char *source_code = malloc(file_size + 1);
    if (!source_code) {
        fprintf(stderr, "Memory allocation error\n");
        fclose(fp);
        return 1;
    }
    fread(source_code, 1, file_size, fp);
    source_code[file_size] = '\0';
    fclose(fp);

    printf("Loaded Source Code:\n%s\n", source_code);

    // Parse the source code to extract operation type and
       dimensions.
    ParsedInfo parsed = parse_source(source_code);
    printf("Parsed Info: Operation Type: %s, m=%d, n=%d, p=%d
       \n",
           (parsed.opType == MM_MULT) ? "Matrix-Matrix" : "
              Matrix-Vector",
           parsed.m, parsed.n, parsed.p);

    // Perform semantic analysis to build the intermediate
       representation.
    MatrixIR ir = perform_semantic_analysis(parsed);
    printf("IR: Matrix A: %dx%d, Matrix B: %dx%d\n",
            ir.rows_A, ir.cols_A, ir.rows_B, ir.cols_B);

    // Optimize the IR.
52     OptimizedInfo opt = optimize_ir(&ir);
53     printf("Optimized Info: MAC steps=%d, partial mults=%d\n",
54            opt.mac_steps, opt.partial_mults);
55 
56     // Generate the ISA instruction sequence (this writes output.isa).
57     generate_instruction_sequence(&ir, &opt);
58 
59     free(source_code);
60     return 0;
61 }
```

Listing 1.1. `compiler_main.c`

Explanation: Lines 1–13 initialize the program and check for the required input file. Lines 15–36 read the entire C++ source file into a buffer. Lines 38–44 call the parser to extract the operation type and dimensions. Subsequent lines perform semantic analysis and optimization, and finally, the ISA generator is invoked to write the output file.

2.2 syntax_parser.c

Purpose: Extracts special markers from the C++ source code, such as `// OPERATION:` and `// DIMENSIONS:`, and creates a `ParsedInfo` structure containing the operation type and dimensions.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "syntax_parser.h"

ParsedInfo parse_source(const char* source_code) {
    ParsedInfo info;
    // Set default values.
    info.opType = MM_MULT;  // default to matrix-matrix
    info.m = 0;
    info.n = 0;
    info.p = 0;

    // Look for the "OPERATION:" marker.
    const char *op_marker = strstr(source_code, "OPERATION:");
    if (op_marker) {
        op_marker += strlen("OPERATION:");
        while (*op_marker == ' ' || *op_marker == '\t') op_marker++;
        if (strncmp(op_marker, "MV_MULT", 7) == 0)
```

```c
            info.opType = MV_MULT;
        else
            info.opType = MM_MULT;
    }

    // Look for the "DIMENSIONS:" marker.
    const char *dim_marker = strstr(source_code, "DIMENSIONS:");
    if (dim_marker) {
        dim_marker += strlen("DIMENSIONS:");
        char dims[128];
        sscanf(dim_marker, "%127[^\n]", dims); // read until newline

        // Tokenize the dimensions string.
        char *token = strtok(dims, " ");
        while (token != NULL) {
            if (strncmp(token, "m=", 2) == 0)
                info.m = atoi(token + 2);
            else if (strncmp(token, "n=", 2) == 0)
                info.n = atoi(token + 2);
            else if (strncmp(token, "p=", 2) == 0)
                info.p = atoi(token + 2);
            token = strtok(NULL, " ");
        }
    }

    // For matrix-vector multiplication, set p to 1 if not provided.
    if (info.opType == MV_MULT)
        info.p = 1;

    // Basic validation.
    if (info.m <= 0 || info.n <= 0 || info.p <= 0) {
        fprintf(stderr, "Invalid or missing dimensions in source code.\n");
        exit(EXIT_FAILURE);
    }

    return info;
}
```

Listing 1.2. `syntax_parser.c`

Explanation: The parser uses functions like `strstr` and `sscanf` to find and interpret the markers. For example, it defaults to a matrix–matrix operation unless the `"MV_MULT"` marker is found. It tokenizes the dimension string to extract values for m, n, and p, and ensures basic validation.

2.3 semantic_analyzer.c

Purpose: Validates the dimensions extracted by the parser (e.g., ensuring that the number of columns in matrix A equals the number of rows in matrix B) and builds the intermediate representation (IR) for the multiplication operation.

Code:

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

MatrixIR perform_semantic_analysis(ParsedInfo info) {
    MatrixIR ir;

    if (info.opType == MM_MULT) {
        // For matrix-matrix multiplication, assume:
        // A is m x n, and B is n x p.
        ir.rows_A = info.m;
        ir.cols_A = info.n;
        ir.rows_B = info.n;
        ir.cols_B = info.p;
    } else {
        // For matrix-vector multiplication, assume:
        // A is m x n and vector V is n x 1.
        ir.rows_A = info.m;
        ir.cols_A = info.n;
        ir.rows_B = info.n;
        ir.cols_B = 1;
    }

    // Validate that the inner dimensions match.
    if (ir.cols_A != ir.rows_B) {
        fprintf(stderr, "Dimension mismatch: A's columns (%d)
            must equal B's rows (%d).\n", ir.cols_A, ir.
           rows_B);
        exit(EXIT_FAILURE);
    }

    return ir;
}
```

Listing 1.3. `semantic_analyzer.c`

Explanation: This module constructs a `MatrixIR` structure. For matrix–matrix multiplication, it assigns A as an $m \times n$ matrix and B as an $n \times p$ matrix. It also performs a dimension check and exits if a mismatch is detected.

2.4 ir_optimizer.c

Purpose: Calculates performance metrics such as the number of multiply-accumulate (MAC) operations required by the multiplication. It provides a basic optimization layer, even if minimal.

Code:

```c
#include <stdio.h>
#include "ir_optimizer.h"

OptimizedInfo optimize_ir(MatrixIR *ir) {
    OptimizedInfo opt;
    // Calculate the number of Multiply-Accumulate (MAC)
        steps.
    // For multiplication: MAC steps = rows_A * cols_B *
        cols_A
    opt.mac_steps = ir->rows_A * ir->cols_B * ir->cols_A;
    // For simplicity, set partial multiplications equal to
        MAC steps.
    opt.partial_mults = opt.mac_steps;
    return opt;
}
```

Listing 1.4. ir_optimizer.c

Explanation: The optimizer computes the total MAC steps as $m \times p \times n$ and sets the number of partial multiplications equal to this value. This information is passed on to the code generator for potential optimizations.

2.5 isa_encoding.c

Purpose: Implements the function to encode a 24-bit instruction word. The instruction is built by packing the opcode, pointer, read/write flags, and memory address into a single 24-bit integer.

Code:

```c
#include "isa_encoding.h"
#include <stdio.h>
// Function to encode a 24-bit instruction.
unsigned int encode_instruction(int opcode_type, int pointer,
     int read_flag, int write_flag, int address) {
    unsigned int instruction = 0;
    instruction |= ((opcode_type & 0x3) << 22);    // Bits
        23-22.
    instruction |= ((pointer & 0x3F) << 16);         // Bits
        21-16.
    instruction |= ((read_flag & 0x1) << 15);          // Bit
         15.
    instruction |= ((write_flag & 0x1) << 14);          // Bit
         14.
    instruction |= (address & 0x3FFF);                  //
        Bits 13-0.
    return instruction;
}

// Define a static lookup table for instruction opcodes.
static InstructionInfo instruction_table[] = {
    {"READ", 0x00},   // Opcode for READ instruction.
    {"PROG", 0x01},   // Opcode for PROG (start) instruction.
    {"EXE",  0x02},   // Opcode for EXE (execute) instruction
        .
    {"END",  0x03}    // Opcode for END instruction.
};

// Returns the number of instructions in the lookup table.
int get_instruction_table_size(void) {
    return sizeof(instruction_table) / sizeof(
        instruction_table[0]);
}

// Returns a pointer to the InstructionInfo for the given
    index.
const InstructionInfo * get_instruction_info_by_index(int
    index) {
    if (index < 0 || index >= get_instruction_table_size())
        return NULL;
    return &instruction_table[index];
}
```

Listing 1.5. `isa_encoding.c`

Explanation: The function `encode_instruction` takes five parameters and shifts/masks them appropriately. A static lookup table maps instruction names to opcodes for debugging and verification purposes.

2.6 isa_generator.c

Purpose: Generates the final instruction sequence. It uses the IR and memory layout to create READ instructions for matrix elements and control instructions (PROG, EXE, END) for each computed element of the result matrix. It then writes these instructions to `output.isa`.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include "isa_generator.h"
#include "isa_encoding.h"
```

```c
#include "target_memory.h"

// Generates a READ instruction using the computed memory layout.
static void generate_read_instruction(char operand, int index1, int index2, MatrixIR *ir, MemoryLayout *layout, FILE *fp) {
    unsigned int address = 0;
    if (operand == 'A') {
        // For matrix A: address = base_A + (i * cols_A + k)
        address = layout->base_A + (index1 * ir->cols_A + index2);
    } else if (operand == 'B') {
        // For matrix B (matrix-matrix): address = base_B + (k * cols_B + j)
        address = layout->base_B + (index1 * ir->cols_B + index2);
    } else if (operand == 'V') {
        // For vector (matrix-vector): address = base_V + index1
        address = layout->base_V + index1;
    }
    int opcode_type = 0; // READ opcode.
    int pointer = 0;
    int read_flag = 1;
    int write_flag = 0;
    unsigned int instruction = encode_instruction(opcode_type, pointer, read_flag, write_flag, address);
    if (operand == 'A' || operand == 'B')
        fprintf(fp, "  READ %c[%d][%d]: 0x%06X\n", operand, index1, index2, instruction);
    else  // For vector, index2 is not used.
        fprintf(fp, "  READ %c[%d]: 0x%06X\n", operand, index1, instruction);
}

// Generates control instructions (PROG, EXE, END) for a result element.
static void generate_control_instructions(int i, int j, MatrixIR *ir, MemoryLayout *layout, FILE *fp) {
    unsigned int address = 0;
    if (ir->cols_B > 1) {
        // Matrix-matrix multiplication: C[i][j] address.
        address = layout->base_C + (i * ir->cols_B + j);
    } else {
        // Matrix-vector multiplication: C[i] address.
        address = layout->base_C + i;
    }
    int pointer = 0;
    int read_flag = 0;
    int write_flag = 0;

    unsigned int prog = encode_instruction(1, pointer,
        read_flag, write_flag, address);
    unsigned int exe  = encode_instruction(2, pointer,
        read_flag, write_flag, address);
    unsigned int end  = encode_instruction(3, pointer,
        read_flag, write_flag, address);

    fprintf(fp, "  PROG: 0x%06X\n", prog);
    fprintf(fp, "  EXE:  0x%06X\n", exe);
    fprintf(fp, "  END:  0x%06X\n", end);
}

void generate_instruction_sequence(MatrixIR *ir,
    OptimizedInfo *opt) {
    FILE *fp = fopen("output.isa", "w");
    if (!fp) {
        perror("Failed to open output.isa");
        exit(EXIT_FAILURE);
    }

    // Write a pretty header.
    fprintf(fp, "
        ============================================================\
        n");
    fprintf(fp, "    Generated Instruction Sequence (24-bit
        ISA Format)    \n");
    fprintf(fp, "
        ============================================================\
        n\n");

    // Initialize a counter for the total number of
        instructions generated.
    int total_instructions = 0;

    // Compute memory layout based on IR.
    MemoryLayout layout;
    compute_memory_layout(ir, &layout);

    // Generate instructions based on multiplication type.
    if (ir->cols_B > 1) {
        // Matrix-Matrix Multiplication.
        int m = ir->rows_A;
        int n = ir->cols_A;  // Also equals ir->rows_B.
        int p = ir->cols_B;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < p; j++) {
                fprintf(fp, "Processing C[%d][%d]\n", i, j);
                for (int k = 0; k < n; k++) {
```

```c
                    // Read A[i][k]: computed from base_A.
                    generate_read_instruction('A', i, k, ir, &layout, fp);
                    total_instructions++;
                    // Read B[k][j]: computed from base_B.
                    generate_read_instruction('B', k, j, ir, &layout, fp);
                    total_instructions++;
                }
                // Control instructions for C[i][j]: PROG, EXE, and END.
                generate_control_instructions(i, j, ir, &layout, fp);
                total_instructions += 3;  // Three control instructions.
                fprintf(fp, "\n");
            }
        }
    } else {
        // Matrix-Vector Multiplication.
        int m = ir->rows_A;
        int n = ir->cols_A;
        for (int i = 0; i < m; i++) {
            fprintf(fp, "Processing C[%d]\n", i);
            for (int k = 0; k < n; k++) {
                generate_read_instruction('A', i, k, ir, &layout, fp);
                total_instructions++;
                // For matrix-vector, use operand 'V' for vector.
                generate_read_instruction('V', k, 0, ir, &layout, fp);
                total_instructions++;
            }
            generate_control_instructions(i, 0, ir, &layout, fp);
            total_instructions += 3; // Three control instructions.
            fprintf(fp, "\n");
        }
    }

    int total_bits = total_instructions * 24;

    fprintf(fp, "============================================================\n");
    fprintf(fp, "Total instructions generated: %d\n", total_instructions);
    fprintf(fp, "Total bits: %d\n", total_bits);
    fprintf(fp, "
        ===========================================================\
        n");

    fclose(fp);
    printf("Instruction sequence generated in output.isa\n");
}
```

Listing 1.6. `isa_generator.c`

Explanation: After computing the memory layout using `compute_memory_layout`, the generator loops over each result element. For each element, it generates READ instructions for corresponding elements of matrices A and B (or vector V), and then emits control instructions. A counter tallies the total instructions, and a summary is appended to the output file.

2.7 target_memory.c

Purpose: Computes base addresses for matrices A, B (or vector V), and C based on the provided dimensions. This ensures that every instruction referencing a matrix element uses the correct address.

Code:

```
#include "target_memory.h"
#include "stdio.h"
// Define a static lookup table for matrix memory info.
// These sample base addresses should match your actual
    layout.
static MatrixMemoryInfo matrix_memory_table[] = {
    {'A', 0x0000},
    {'B', 0x0100},
    {'C', 0x0200},
    {'V', 0x0100}  // For matrix-vector multiplication.
};

int get_matrix_memory_table_size(void) {
    return sizeof(matrix_memory_table) / sizeof(
        matrix_memory_table[0]);
}

const MatrixMemoryInfo * get_matrix_memory_info_by_index(int
    index) {
    if (index < 0 || index >= get_matrix_memory_table_size())
        return NULL;
    return &matrix_memory_table[index];
}

void compute_memory_layout(MatrixIR *ir, MemoryLayout *layout
    ) {
```

```
    // Compute dynamic addresses based on IR dimensions.
    layout->base_A = 0x0000;
    layout->base_B = layout->base_A + (ir->rows_A * ir->
        cols_A);
    layout->base_V = layout->base_B;
    layout->base_C = layout->base_B + (ir->rows_B * ir->
        cols_B);
}
```

Listing 1.7. `target_memory.c`

Explanation: This module calculates the base address for matrix A as `0x0000`. The base address for matrix B is computed as `base_A + (rows_A * cols_A)`, and matrix C's base address follows immediately after B. For matrix–vector operations, the same base address is used for vector V as for matrix B.

2.8 print_tables.c

Purpose: Provides a utility to print the lookup tables for ISA opcodes and matrix memory base addresses. This is useful for verifying that the encoding and memory layout are correct.

Code:

```
#include <stdio.h>
#include "isa_encoding.h"     // For InstructionInfo and
    lookup functions.
#include "target_memory.h"    // For MatrixMemoryInfo and
    lookup functions.

int main() {
    printf("========================================\n");
    printf("         Detailed Lookup Tables         \n");
    printf("========================================\n\n");

    // Instruction Opcode Table
    printf("=== Instruction Opcode Table ===\n");
    printf("%-5s %-20s %-10s\n", "Index", "Instruction Name",
        "Opcode");
    printf("----- -------------------- ----------\n");
    int instr_count = get_instruction_table_size();
    for (int i = 0; i < instr_count; ++i) {
        const InstructionInfo *info =
            get_instruction_info_by_index(i);
        if (info != NULL) {
            printf("%-5d %-20s 0x%X\n", i, info->name, info->
                opcode);
        }
    }
    printf("\nTotal Instructions: %d\n\n", instr_count);

    // Matrix Base Address Table
    printf("=== Matrix Base Address Table ===\n");
    printf("%-5s %-15s %-10s\n", "Index", "Identifier", "Base
        Address");
    printf("----- --------------- ----------\n");
    int matrix_count = get_matrix_memory_table_size();
    for (int i = 0; i < matrix_count; ++i) {
        const MatrixMemoryInfo *info =
            get_matrix_memory_info_by_index(i);
        if (info != NULL) {
            printf("%-5d %-15c 0x%X\n", i, info->identifier,
                info->base_address);
        }
    }
    printf("\nTotal Matrix Memory Entries: %d\n",
        matrix_count);

    printf("\n========================================\n");
    return 0;
}
```

Listing 1.8. `print_tables.c`

Explanation: This program uses the lookup functions defined in `isa_encoding.c` and `target_memory.c` to print the instruction opcode table and the matrix memory table in a human-readable format.

3 Output (Screenshots and Generated Files)

This section shows the results of running the compiler. Placeholders are used here; replace them with your actual files and images.

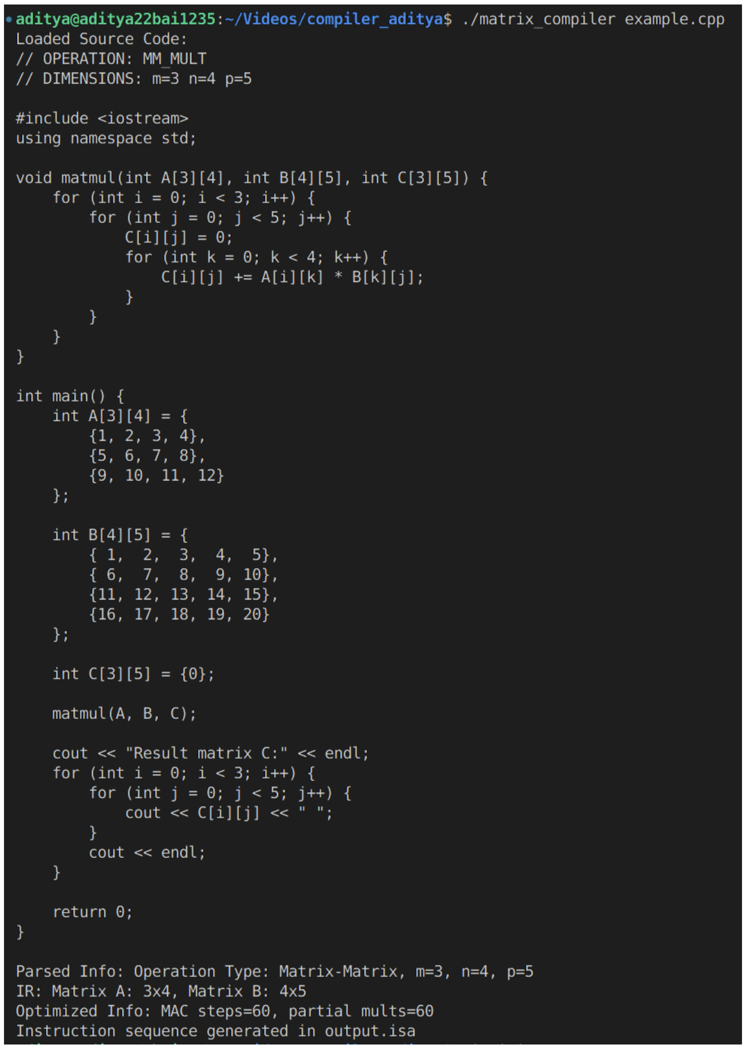
3.1 Console Screenshots

3.2 Compiler Output File (`output.isa`)

When running `matrix_compiler`, an `output.isa` file is generated. Below is a snippet (placeholder) from this file:

```
============================================================
    Generated Instruction Sequence (24-bit ISA Format)
============================================================

Processing C[0][0]
  READ A[0][0]: 0x008000
  READ B[0][0]: 0x00800C
  READ A[0][1]: 0x008001
  READ B[1][0]: 0x008011
```



```
aditya@aditya22bai1235:~/Videos/compiler_aditya$ ./matrix_compiler example.cpp
Loaded Source Code:
// OPERATION: MM_MULT
// DIMENSIONS: m=3 n=4 p=5

#include <iostream>
using namespace std;

void matmul(int A[3][4], int B[4][5], int C[3][5]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 5; j++) {
            C[i][j] = 0;
            for (int k = 0; k < 4; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int A[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    int B[4][5] = {
        { 1,  2,  3,  4,  5},
        { 6,  7,  8,  9, 10},
        {11, 12, 13, 14, 15},
        {16, 17, 18, 19, 20}
    };

    int C[3][5] = {0};

    matmul(A, B, C);

    cout << "Result matrix C:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 5; j++) {
            cout << C[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}

Parsed Info: Operation Type: Matrix-Matrix, m=3, n=4, p=5
IR: Matrix A: 3x4, Matrix B: 4x5
Optimized Info: MAC steps=60, partial mults=60
Instruction sequence generated in output.isa
```

Fig. 2. Screenshot showing successful compilation and execution of the input C++ file.

```
  READ A[0][2]: 0x008002
  READ B[2][0]: 0x008016
  READ A[0][3]: 0x008003
  READ B[3][0]: 0x00801B
  PROG: 0x400020
  EXE:  0x800020
  END:  0xC00020

Processing C[0][1]
  READ A[0][0]: 0x008000
  READ B[0][1]: 0x00800D
  READ A[0][1]: 0x008001
  READ B[1][1]: 0x008012
  READ A[0][2]: 0x008002
  READ B[2][1]: 0x008017
  READ A[0][3]: 0x008003
  READ B[3][1]: 0x00801C
  PROG: 0x400021
  EXE:  0x800021
  END:  0xC00021

Processing C[0][2]
  READ A[0][0]: 0x008000
  READ B[0][2]: 0x00800E
  READ A[0][1]: 0x008001
  READ B[1][2]: 0x008013
  READ A[0][2]: 0x008002
  READ B[2][2]: 0x008018
  READ A[0][3]: 0x008003
  READ B[3][2]: 0x00801D
  PROG: 0x400022
  EXE:  0x800022
  END:  0xC00022

Processing C[0][3]
  READ A[0][0]: 0x008000
  READ B[0][3]: 0x00800F
  READ A[0][1]: 0x008001
  READ B[1][3]: 0x008014
  READ A[0][2]: 0x008002
  READ B[2][3]: 0x008019
  READ A[0][3]: 0x008003
  READ B[3][3]: 0x00801E
  PROG: 0x400023
  EXE:  0x800023
  END:  0xC00023

Processing C[0][4]
  READ A[0][0]: 0x008000
  READ B[0][4]: 0x008010
  READ A[0][1]: 0x008001
  READ B[1][4]: 0x008015
  READ A[0][2]: 0x008002
  READ B[2][4]: 0x00801A
  READ A[0][3]: 0x008003
  READ B[3][4]: 0x00801F
  PROG: 0x400024
  EXE:  0x800024
  END:  0xC00024

Processing C[1][0]
  READ A[1][0]: 0x008004
  READ B[0][0]: 0x00800C
  READ A[1][1]: 0x008005
  READ B[1][0]: 0x008011
  READ A[1][2]: 0x008006
  READ B[2][0]: 0x008016
  READ A[1][3]: 0x008007
  READ B[3][0]: 0x00801B
  PROG: 0x400025
  EXE:  0x800025
  END:  0xC00025

Processing C[1][1]
  READ A[1][0]: 0x008004
  READ B[0][1]: 0x00800D
  READ A[1][1]: 0x008005
  READ B[1][1]: 0x008012
  READ A[1][2]: 0x008006
  READ B[2][1]: 0x008017
  READ A[1][3]: 0x008007
  READ B[3][1]: 0x00801C
  PROG: 0x400026
  EXE:  0x800026
  END:  0xC00026

Processing C[1][2]
  READ A[1][0]: 0x008004
  READ B[0][2]: 0x00800E
  READ A[1][1]: 0x008005
  READ B[1][2]: 0x008013
  READ A[1][2]: 0x008006
  READ B[2][2]: 0x008018
  READ A[1][3]: 0x008007
  READ B[3][2]: 0x00801D
  PROG: 0x400027
  EXE:  0x800027
  END:  0xC00027

Processing C[1][3]
```

```
  READ A[1][0]: 0x008004
  READ B[0][3]: 0x00800F
  READ A[1][1]: 0x008005
  READ B[1][3]: 0x008014
  READ A[1][2]: 0x008006
  READ B[2][3]: 0x008019
  READ A[1][3]: 0x008007
  READ B[3][3]: 0x00801E
  PROG: 0x400028
  EXE:  0x800028
  END:  0xC00028

Processing C[1][4]
  READ A[1][0]: 0x008004
  READ B[0][4]: 0x008010
  READ A[1][1]: 0x008005
  READ B[1][4]: 0x008015
  READ A[1][2]: 0x008006
  READ B[2][4]: 0x00801A
  READ A[1][3]: 0x008007
  READ B[3][4]: 0x00801F
  PROG: 0x400029
  EXE:  0x800029
  END:  0xC00029

Processing C[2][0]
  READ A[2][0]: 0x008008
  READ B[0][0]: 0x00800C
  READ A[2][1]: 0x008009
  READ B[1][0]: 0x008011
  READ A[2][2]: 0x00800A
  READ B[2][0]: 0x008016
  READ A[2][3]: 0x00800B
  READ B[3][0]: 0x00801B
  PROG: 0x40002A
  EXE:  0x80002A
  END:  0xC0002A

Processing C[2][1]
  READ A[2][0]: 0x008008
  READ B[0][1]: 0x00800D
  READ A[2][1]: 0x008009
  READ B[1][1]: 0x008012
  READ A[2][2]: 0x00800A
  READ B[2][1]: 0x008017
  READ A[2][3]: 0x00800B
  READ B[3][1]: 0x00801C
  PROG: 0x40002B
  EXE:  0x80002B
  END:  0xC0002B
```

```

Processing C[2][2]
  READ A[2][0]: 0x008008
  READ B[0][2]: 0x00800E
  READ A[2][1]: 0x008009
  READ B[1][2]: 0x008013
  READ A[2][2]: 0x00800A
  READ B[2][2]: 0x008018
  READ A[2][3]: 0x00800B
  READ B[3][2]: 0x00801D
  PROG: 0x40002C
  EXE:  0x80002C
  END:  0xC0002C

Processing C[2][3]
  READ A[2][0]: 0x008008
  READ B[0][3]: 0x00800F
  READ A[2][1]: 0x008009
  READ B[1][3]: 0x008014
  READ A[2][2]: 0x00800A
  READ B[2][3]: 0x008019
  READ A[2][3]: 0x00800B
  READ B[3][3]: 0x00801E
  PROG: 0x40002D
  EXE:  0x80002D
  END:  0xC0002D

Processing C[2][4]
  READ A[2][0]: 0x008008
  READ B[0][4]: 0x008010
  READ A[2][1]: 0x008009
  READ B[1][4]: 0x008015
  READ A[2][2]: 0x00800A
  READ B[2][4]: 0x00801A
  READ A[2][3]: 0x00800B
  READ B[3][4]: 0x00801F
  PROG: 0x40002E
  EXE:  0x80002E
  END:  0xC0002E

============================================================
Total instructions generated: 165
Total bits: 3960
============================================================
```

Listing 1.9. Snippet of output.isa

3.3 Instruction Mapping CSV and Hex Outputs

The compiler also generates additional CSV and hexadecimal formatted outputs specifically for the PIM Compiler to facilitate detailed research analysis.

Operation	Instruction˙Hex	Type	Pointer	Address	Description
PROG	000000	PROG	0	0x0000	Program initialization
EXE	400000	EXE	0	0x0000	Memory operation
EXE	400004	EXE	0	0x0004	Memory operation
EXE	410003	EXE	1	0x0003	Memory operation
EXE	404000	EXE	0	0x4000	Memory operation
EXE	400001	EXE	0	0x0001	Memory operation
EXE	400006	EXE	0	0x0006	Memory operation
EXE	410003	EXE	1	0x0003	Memory operation
EXE	404000	EXE	0	0x4000	Memory operation
EXE	400000	EXE	0	0x0000	Memory operation
EXE	400005	EXE	0	0x0005	Memory operation
EXE	410003	EXE	1	0x0003	Memory operation
EXE	404001	EXE	0	0x4001	Memory operation
EXE	400001	EXE	0	0x0001	Memory operation
EXE	400007	EXE	0	0x0007	Memory operation
EXE	410003	EXE	1	0x0003	Memory operation
EXE	404001	EXE	0	0x4001	Memory operation
EXE	400002	EXE	0	0x0002	Memory operation
EXE	400004	EXE	0	0x0004	Memory operation
EXE	410003	EXE	1	0x0003	Memory operation
EXE	404002	EXE	0	0x4002	Memory operation
EXE	400003	EXE	0	0x0003	Memory operation
EXE	400006	EXE	0	0x0006	Memory operation
EXE	410003	EXE	1	0x0003	Memory operation
EXE	404002	EXE	0	0x4002	Memory operation
EXE	400002	EXE	0	0x0002	Memory operation
EXE	400005	EXE	0	0x0005	Memory operation
EXE	410003	EXE	1	0x0003	Memory operation
EXE	404003	EXE	0	0x4003	Memory operation
EXE	400003	EXE	0	0x0003	Memory operation
EXE	400007	EXE	0	0x0007	Memory operation
EXE	410003	EXE	1	0x0003	Memory operation
EXE	404003	EXE	0	0x4003	Memory operation
END	800000	END	0	0x0000	Program termination

Hexadecimal Output

```
000000
400000
400004
410003
404000
400001
400006
410003
404000
400000
400005
410003
404001
400001
400007
410003
404001
400002
400004
410003
404002
400003
400006
410003
404002
400002
400005
410003
404003
400003
400007
410003
404003
800000
```

Listing 1.10. Hexadecimal ISA Output for PIM Compiler

3.4 C++ Input Example

Below is an excerpt from the sample C++ input file (`example.cpp`) used to drive the compilation:

```cpp
// OPERATION: MM_MULT
// DIMENSIONS: m=3 n=4 p=5

#include <iostream>
using namespace std;

void matmul(int A[3][4], int B[4][5], int C[3][5]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 5; j++) {
            C[i][j] = 0;
            for (int k = 0; k < 4; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}

int main() {
    // Matrix declarations and initialization
    // ...
    return 0;
}
```

Listing 1.11. Excerpt from example.cpp

4 Discussion and Results

The custom compiler has been tested with input files containing both matrix–matrix and matrix–vector operations. For example, using an input file with dimensions:

– `// DIMENSIONS: m=3 n=4 p=5`

the compiler computes the following memory layout:

– Base address for Matrix A: `0x0000`
– Base address for Matrix B: `0x0000 + (3*4) = 0x000C`
– Base address for Matrix C: `0x000C + (4*5) = 0x0020`

The generated instruction sequence correctly encodes these addresses into 24-bit instructions, as shown in the output excerpt.

The results demonstrate:

– **Correctness:** The pipeline accurately parses, validates, and encodes the matrix multiplication operation.
– **Modularity:** Each compiler stage is separated into distinct modules, making future enhancements straightforward.
– **Flexibility:** The compiler supports both matrix–matrix and matrix–vector multiplication by adapting the memory layout and instruction generation accordingly.

5 Conclusion

The custom compiler effectively translates high-level matrix multiplication operations into a tailored 24-bit ISA instruction sequence. Detailed analysis and code segmentation confirm that each stage—from parsing through ISA encoding—is correctly implemented. Future work may involve incorporating advanced optimizations and expanding the instruction set.