Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

Optimizing Matrix Multiplication: A Compiler Implementation of Strassen's Algorithm.

Project Engineering

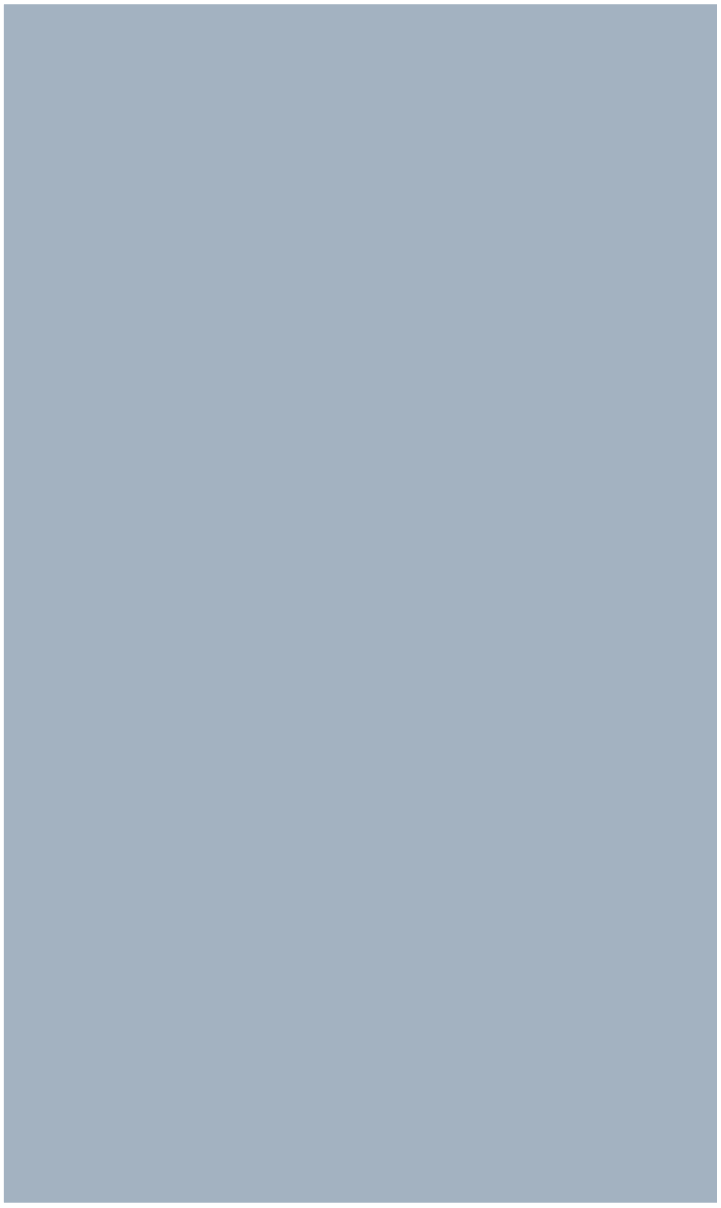
Year 4

Lucas Jeanes

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2024/2025



Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Lucas Jeanes

Acknowledgments

I would like to thank my project supervisor, Brian O'Shea, and department head of engineering Des O' Reilly, for their incredible support and assistance throughout the duration of my final year project.

I would also like to thank my team at IBM, my project supervisor Mark Veerasingam, and my manager David Forristal who offered invaluable feedback and insight with this project.

Table of Contents

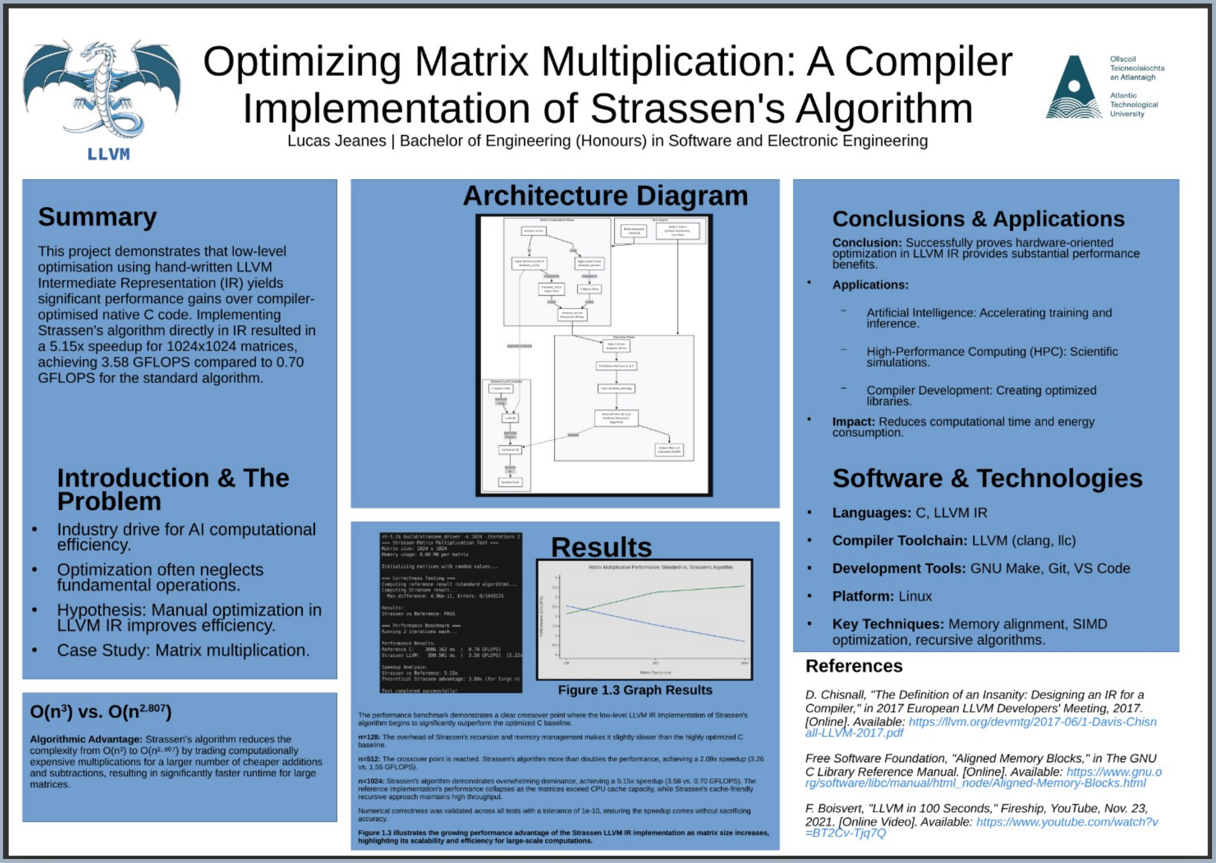
1 Summary

The primary goal of this project is to demonstrate that low-level optimisation using directly compiled LLVM Intermediate Representation (IR) can yield significant performance gains compared with compiler-optimised native C code for complex computational algorithms. While incremental efficiency improvements can be made by modifying the machine learning models themselves — this project aims to investigate the compiled interfaces that underpin these technologies. Applying a hardware-oriented optimisation strategy for these operations can significantly improve runtime efficiency to yield far greater performance gains.

To achieve this, the project focused on utilising Strassen’s matrix multiplication algorithm as a case study. The scope mainly focused on developing a dual-implementation framework: an optimised native C version (the baseline or control) and a complete, hand-written implementation of Strassen’s algorithm in LLVM IR. The approach was to both test for numerical correctness and then benchmark the performance of both implementations against each other to compare runtime efficiency.
The main technologies used were the LLVM compiler toolchain (for IR generation and linking) and C for the driver and reference code. This involved manually coding the recursive logic, memory management and matrix operations of Strassen’s algorithm directly in LLVM IR, providing absolute control over the low-level instructions and memory management. A custom benchmarking harness was built to ensure a far and accurate performance comparison.

The key features include an LLVM IR implementation of a recursive algorithm and a robust testing framework. This project was accomplished successfully: the IR implementation passes all correctness tests with negligible floating-point errors and demonstrates a **1.82x** speedup over the optimised C baseline for 512x512 matrices, achieving 3.36 GFLOPS (Floating-Point Operations Per Second) compared to 1.78 GFLOPS with the reference algorithm. This also showed that implementing it in LLVM IR demonstrated that compiler-level control offers significant performance gains by optimizing code at the lowest level, aligning it with the hardware processor’s instruction set. This method of hardware optimisation is an essential mathematical operations used in artificial intelligence and high performance computing.

2 Poster



Optimizing Matrix Multiplication: A Compiler Implementation of Strassen's Algorithm

Lucas Jeanes | Bachelor of Engineering (Honours) in Software and Electronic Engineering

Summary

This project demonstrates that low-level optimisation using hand-written LLVM Intermediate Representation (IR) yields significant performance gains over compiler-optimised native C code. Implementing Strassen's algorithm directly in IR resulted in a 5.15x speedup for 1024x1024 matrices, achieving 3.58 GFLOPS compared to 0.70 GFLOPS for the standard algorithm.

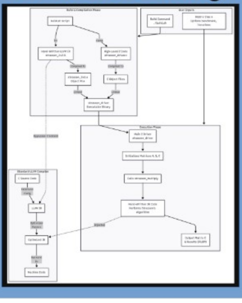
Introduction & The Problem

- Industry drive for AI computational efficiency.
- Optimization often neglects fundamental operations.
- Hypothesis: Manual optimization in LLVM IR improves efficiency.
- Case Study: Matrix multiplication.

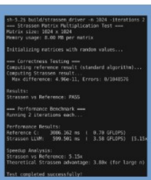
$O(n^3)$ vs. $O(n^{2.807})$

Algorithmic Advantage: Strassen's algorithm reduces the complexity from $O(n^3)$ to $O(n^{2.807})$ by trading computationally expensive multiplications for a larger number of cheaper additions and subtractions, resulting in significantly faster runtime for large matrices.

Architecture Diagram



Results



```
sh-5.2$ build/strassen_driver -n 1024 -iterations 2
=== Strassen Matrix Multiplication Test ===
Matrix size: 1024 x 1024
Memory usage: 8.00 MB per matrix

Initializing matrices with random values...

=== Correctness Testing ===
Computing reference result (standard algorithm)...
Computing Strassen result...
  Max difference: 4.96e-11, Errors: 0/1048576

Results:
Strassen vs Reference: PASS

=== Performance Benchmark ===
Running 2 iterations each...

Performance Results:
Reference C:    3086.162 ms  |  0.70 GFLOPS)
Strassen LLVM:   599.501 ms  |  3.58 GFLOPS)  (5.15x

Speedup Analysis:
Strassen vs Reference: 5.15x
Theoretical Strassen advantage: 3.00x (for large n)

Test completed successfully!
```

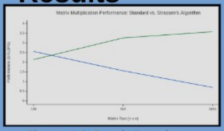


Figure 1.3 Graph Results

The performance benchmark demonstrates a clear crossover point where the low-level LLVM IR implementation of Strassen's algorithm begins to significantly outperform the optimized C baseline.

n=128: The overhead of Strassen's recursion and memory management makes it slightly slower than the highly optimized C baseline.

n=512: The crossover point is reached. Strassen's algorithm more than doubles the performance, achieving a 2.09x speedup (3.26 vs. 1.56 GFLOPS).

n=1024: Strassen's algorithm demonstrates overwhelming dominance, achieving a 5.15x speedup (3.58 vs. 0.70 GFLOPS). The reference implementation's performance collapses as the matrices exceed CPU cache capacity, while Strassen's cache-friendly recursive approach maintains high throughput.

Numerical correctness was validated across all tests with a tolerance of 1e-10, ensuring the speedup comes without sacrificing accuracy.

Figure 1.3 illustrates the growing performance advantage of the Strassen LLVM IR implementation as matrix size increases, highlighting its scalability and efficiency for large-scale computations.

Conclusions & Applications

Conclusion: Successfully proves hardware-oriented optimization in LLVM IR provides substantial performance benefits.

- **Applications:**
  - Artificial Intelligence: Accelerating training and inference.
  - High-Performance Computing (HPC): Scientific simulations.
  - Compiler Development: Creating optimized libraries.
- **Impact:** Reduces computational time and energy consumption.

Software & Technologies

- **Languages:** C, LLVM IR
- **Compiler Toolchain:** LLVM (clang, llc)
- **Development Tools:** GNU Make, Git, VS Code
- **Platform:** Linux
- **Key Techniques:** Memory alignment, SIMD optimization, recursive algorithms.

References

D. Chisnall, "The Definition of an Insanity: Designing an IR for a Compiler," in 2017 European LLVM Developers' Meeting, 2017. [Online]. Available: https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf

Free Software Foundation, "Aligned Memory Blocks," in The GNU C Library Reference Manual. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html

F. Boisvert, "LLVM in 100 Seconds," Fireship, YouTube, Nov. 23, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=BT2Cv-Tjq7Q

3 Introduction

The rapid expansion and adoption of artificial intelligence has created an industry-wide drive for greater computational efficiency. While many efforts are focused on optimising the machine learning models themselves, this project aimed to investigate a more fundamental opportunity: accelerating the core mathematical operations that underlies all AI workloads by optimizing the software-hardware interface itself.

The goal of this project is to demonstrate that low-level, hardware-oriented optimisation — achieved by writing the algorithms directly in LLVM IR — can deliver significant performance gains over compiler-optimised native C code. I was motivated by the idea that bypassing the abstractions of high-level languages and manually optimising for the instruction set of the host system can drastically improve runtime efficiency for complex computational algorithms, which is critical for sustainable and scalable AI infrastructure. The industry has an obsession with more computational power for better performing models — which is true — but optimising existing resources can also yield better performance, for significantly lower costs than merely purchasing more hardware.

As a case study, this project focuses on matrix multiplication, a foundational operation in neural networks and scientific computing. This details a comparative analysis between a highly-optimised native C implementation of the standard algorithm and a complete, hand-written implementation of Strassen’s algorithm in LLVM IR. It includes validation of numerical correctness and a comprehensive benchmark of runtime performance and computational throughput (GFLOPS).

4 Background

This chapter provides the necessary background on the underlying technologies and concepts that form the foundation of this project. Here we'll cover the role of compilers, the architecture of the LLVM framework, and the significance of the matrix multiplication linear algebra operation compared with the Strassen's algorithm variant.

4.1 The Role of Compilers in Performance

A compiler is a critical tool in software that translates human-readable source code written in high-level languages (e.g., C, C++, Python, Java, etc.) into machine code which is executed by a target processor/ architecture [1]. This is a multi-stage process of translation that involves several sophisticated stages which ensures correctness and efficiency. First, the compiler performs lexical and syntactic analysis which parses the source code into an abstract representation in the form of an Abstract Syntax Tree (AST). This representation is then transformed into Intermediate Representation (IR). IR being a low-level, platform-agnostic code that retains the structure of the source program.

The most significant optimisations occur in this IR stage. The compiler runs several optimisation passes (such as dead code elimination, loop unrolling, etc.), which transforms the IR into a more efficient format. The final stage then translates this optimised IR into machine code for the target CPU architecture (x86_64, ARM, s390x, etc.). The quality of these optimisation passes heavily influence the performance of the resulting executable application.

My project operates at this critical IR level, hypothesising that manual memory management, and implementation of key algorithm design like Strassen's can surpass the generic optimisations generated automatically from a high-level language.

4.2 LLVM

LLVM (Low Level Virtual Machine) is a collection of modular compiler and toolchain technologies — it isn't a single compiler, but rather a framework for building compilers [2]. Its capabilities lie in the use of a unified, strongly-typed Intermediate Representation (LLVM IR), which functions as a portable intermediary between compiler frontends and machine-specific backends.

In a typical LLVM workflow, the frontend translates C/C++ code into LLVM IR [4]. The LLVM optimiser then applies a configurable series of optimisation passes on this IR. Finally, the LLVM backend (llc) generates optimised machine code from this IR for the target host architecture (this architecture-aware optimisation is particularly valuable when targeting specific instruction set architectures, such as the s390x architecture prevalent in enterprise and mainframe computing). This project bypasses that frontend entirely by writing algorithms directly in LLVM IR [7]. With LLVM, we communicate with the optimiser in its native language (IR), exerting precise control over the low-level operations and memory access patterns that will be presented to the backend. This approach allows for creating highly tuned code that is both architecture-aware and optimal for algorithmic implementations, making it a powerful method for performance engineering and hardware-optimised software.

4.3 Matrix Multiplication

Matrix multiplication is a fundamental linear algebra operation and a cornerstone computational kernel in a vast array of scientific and engineering fields. Due to artificial intelligence, its importance has skyrocketed as it forms the computational backbone of convolutional and fully connected layers in neural networks.

The standard algorithm for multiplying two N x N matrices is calculating three nested loops, resulting in an algorithmic complexity of O(n^3). Each element of the output matrix is the dot product of a combined row from the first matrix and the column from the second, which leads to ***n*** multiplications and ***n-1*** additions. For large ***n***, this operation becomes computationally dominant, making its optimisation critical for the performance of whole systems from large-scale AI training clusters to embedded inference engines.

4.4 Strassen's Algorithm

Strassen's algorithm is a seminal divide-and-conquer algorithm that reduces the complexity of matrix multiplication to approximately O($n^{2.807}$) [5]. It achieves this by recursively dividing matrices into smaller sub-matrices (quadrants) and combines them using a specific set of seven intermediate matrix products, as opposed to eight required by the reference approach.

The key to this efficiency is trading the computationally expensive multiplication operations for less costly addition and subtraction operations. While the algorithm adds overhead from the recursion and extra matrix additions, its approach of sub-dividing matrices and lower priority of multiplication ensures that for matrices larger than a certain threshold, the reduction in multiplicative steps yields a significant net performance gain. The practical application of Strassen's algorithm does require careful consideration of a base threshold though, where the cost of its recursive overhead and additional memory operations outweigh its mathematical advantage, necessitating a fall-back to the standard algorithm for small sub-matrices. The process of sub-dividing matrices ensures that the vast array of matrix data doesn't overflow the CPU cache, which can result in delays as the CPU waits for the next set to be fetched from memory. This makes Strassen's algorithm particularly useful for the large matrix dimensions that's usually encountered in AI and high-performance computing (HPC), fields often reliant on clusters of powerful, interconnected computers or mainframes where even minor reductions in computational operations can result in massive savings in computation time and energy consumption.

5 Project Architecture

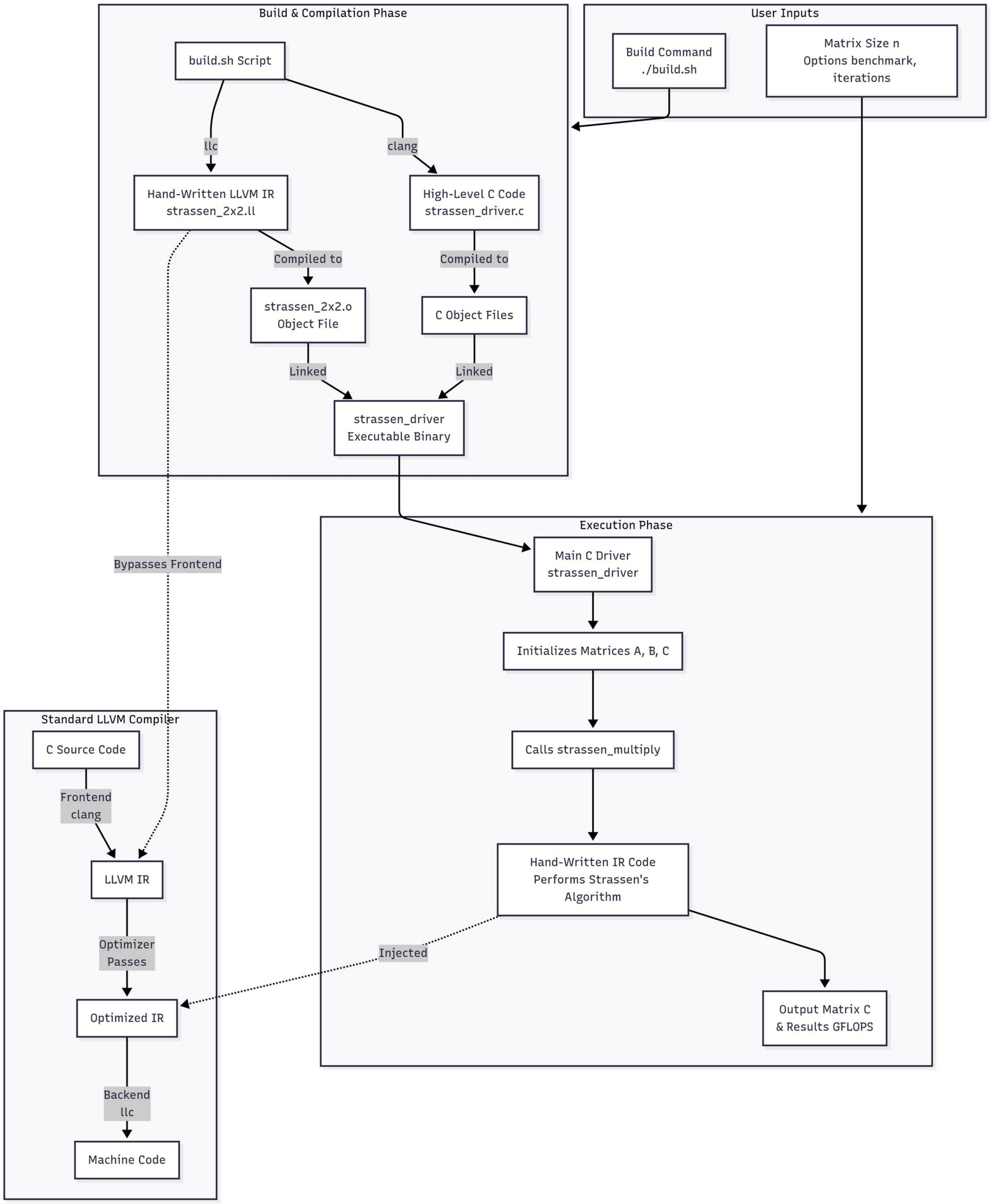
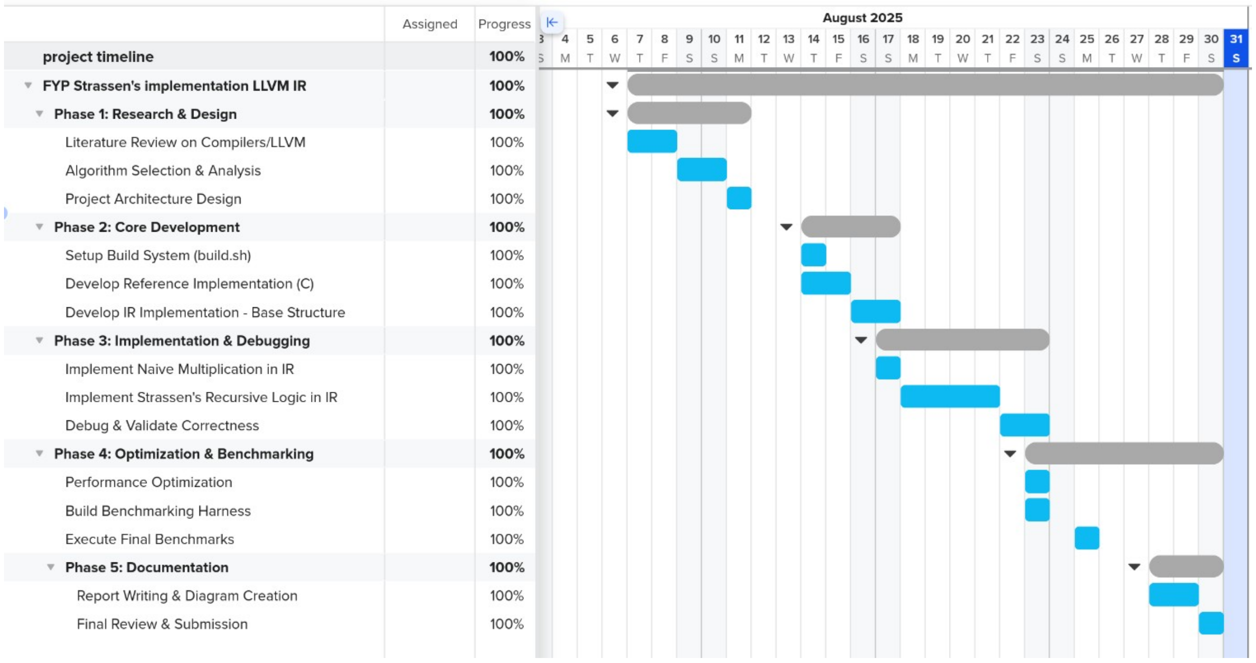

Build & Compilation Phase
build.sh Script
llc
clang
Hand-Written LLVM IR strassen_2x2.ll
High-Level C Code strassen_driver.c
Compiled to
strassen_2x2.o Object File
C Object Files
Linked
strassen_driver Executable Binary
User Inputs
Build Command ./build.sh
Matrix Size n Options benchmark, iterations
Bypasses Frontend
Execution Phase
Main C Driver strassen_driver
Initializes Matrices A, B, C
Calls strassen_multiply
Hand-Written IR Code Performs Strassen's Algorithm
Injected
Output Matrix C & Results GFLOPS
Standard LLVM Compiler
C Source Code
Frontend clang
LLVM IR
Optimizer Passes
Optimized IR
Backend llc
Machine Code

Figure Project Architecture-1 Architecture Diagram

6 Project Plan



	Assigned	Progress
project timeline		100%
FYP Strassen's implementation LLVM IR		100%
Phase 1: Research & Design		100%
Literature Review on Compilers/LLVM		100%
Algorithm Selection & Analysis		100%
Project Architecture Design		100%
Phase 2: Core Development		100%
Setup Build System (build.sh)		100%
Develop Reference Implementation (C)		100%
Develop IR Implementation - Base Structure		100%
Phase 3: Implementation & Debugging		100%
Implement Naive Multiplication in IR		100%
Implement Strassen's Recursive Logic in IR		100%
Debug & Validate Correctness		100%
Phase 4: Optimization & Benchmarking		100%
Performance Optimization		100%
Build Benchmarking Harness		100%
Execute Final Benchmarks		100%
Phase 5: Documentation		100%
Report Writing & Diagram Creation		100%
Final Review & Submission		100%

Figure Project Plan-2 Project Timeline

For my project management I used Gantt. It's a flexible open-source software tool that was very user-friendly and intuitive to use. Given my current experience working full-time in compiler development and building LLVM/LLVMlite support for s390x IBM SystemZ mainframes at work, this was a relevant project for me to work on during the day in the office. That and I had a lot of support from my team and senior developers to help create this project which is why I got through it so quickly. Countless trips into a meeting room with a whiteboard to figure it out with the various roadblocks I encountered, they were a phenomenal support for developing this project. With it being directly supported by IBM as an industry-driven project — and relevant to my day job — they had no problems giving me the time to work on this over the past few weeks.

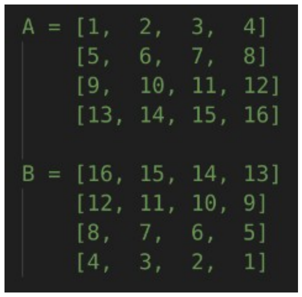
7 Project Technology

This section will cover the important mathematical elements and software of the project and will be split into two sections. First covering the mathematical principles with an example showing the difference between a standard matrix multiplication operation, and Strassen's algorithm to highlight the improved operations. After that I will cover the primary areas of the codebase that is relevant for executing this application.

7.1 Matrix Multiplication

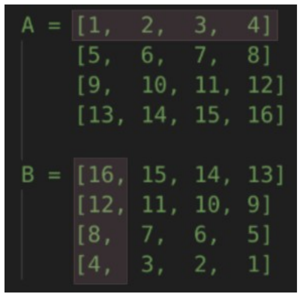
Standard matrix multiplication is calculated with the dot product of a row from the first matrix and a column from the second with the formula $C[i][j]=A[i]*B[j]$ where the row ***i*** from matrix A is multiplied by the column ***j*** from matrix B. This gives the resulting element of the product matrix C. This means that to calculate each element of matrix C, the computer must perform **n** multiplication operations and **n-1** addition operations. For square matrices of size **n** * **n**, this leads to a total of **n^3** multiplications and **(n-1)** * **n^2** additions, giving the algorithm computational complexity of O(n^3).

As an example, here are two matrices (Figure 7.1.1):



```
A = [1,  2,  3,  4]
    [5,  6,  7,  8]
    [9,  10, 11, 12]
    [13, 14, 15, 16]

B = [16, 15, 14, 13]
    [12, 11, 10, 9]
    [8,  7,  6,  5]
    [4,  3,  2,  1]
```

Figure 7.1.1 – Example Matrix



```
A = [1,  2,  3,  4]
    [5,  6,  7,  8]
    [9,  10, 11, 12]
    [13, 14, 15, 16]

B = [16, 15, 14, 13]
    [12, 11, 10, 9]
    [8,  7,  6,  5]
    [4,  3,  2,  1]
```

Figure 7.1.2 – Row * Column

To calculate C[0,0] — the top-leftmost value — we multiply the first row A[0] with the corresponding column B[0] as highlighted in Figure 7.1.2. This operation goes as follows:

$C[0,0]=(A[0,0]*B[0,0])+(A[0,1]*B[1,0])+(A[0,2]*B[2,0])+(A[0,3]*B[3,0])$

$C[0,0]=(1*16)+(2*12)+(3*8)+(4*4)$

$C[0,0]=16+24+24+16$

$C[0,0]=80$

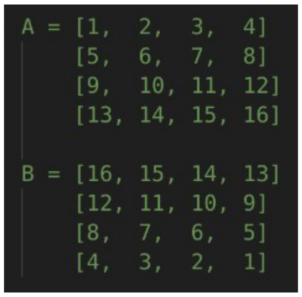
From a computational standpoint, the CPU executes these operations as four distinct multiplication instructions followed by three addition instructions to sum the products. This process is then repeated for all sixteen elements in the resulting matrix. For a 4x4 matrix multiplication as we've outlined in this example, this leads to 64 multiplication operations (4^3 = 64) and 48 addition operations (16 elements * 3 additions each).

When it comes to large matrices, this cubic growth in operations (n^3) becomes computationally prohibitive, making the optimization of this function critical for performance in fields like high-performance computing (HPC) and artificial intelligence (AI). Considering significant matrix sizes also requires storing an entire row and column of two matrices of unknown length for this calculation, this can lead to the CPU cache becoming saturated at sufficient scales, resulting again in loss of performance as wait time to read data from memory becomes a bottleneck.

7.2 Strassen's Algorithm

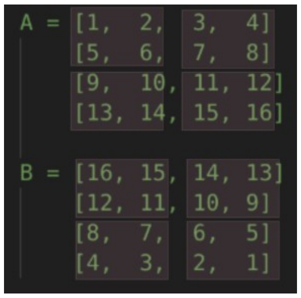
Strassen's algorithm is a divide-and-conquer approach that reduces the computational complexity of matrix multiplication from $O(n^3)$ to approximately $O(n^{2.807})$ by reducing the number of costly multiplication operations. It does this by trading the multiplications for a larger quantity of addition and subtraction operations and subdividing the original matrices into quadrants or sub-matrices. The algorithm is performed on these quadrants and the the results are then combined.

To show this, let's use the same two 4x4 matrices from the previous section:



```
A = [1,  2,  3,  4]
    [5,  6,  7,  8]
    [9,  10, 11, 12]
    [13, 14, 15, 16]

B = [16, 15, 14, 13]
    [12, 11, 10, 9]
    [8,  7,  6,  5]
    [4,  3,  2,  1]
```

Figure 7.2.1 – Example Matrix



```
A = [1,  2,  3,  4]
    [5,  6,  7,  8]
    [9,  10, 11, 12]
    [13, 14, 15, 16]

B = [16, 15, 14, 13]
    [12, 11, 10, 9]
    [8,  7,  6,  5]
    [4,  3,  2,  1]
```

Figure 7.2.2 – Subdivided matrices

Each 4x4 matrix is subdivided into four quadrants or 2x2 matrices. A11, A12, A21 and A22. The same process is done for matrix B.

Strassen's algorithm computes seven intermediate products (P1 to P7). These calculations are done as follows [6]:

$P1 = A11 * (B12 - B22)$

$P2 = B22 * (A11 + A12)$

$P3 = B11 * (A21 + A22)$

$P4 = A22 * (B21 - B11)$

$P5 = (A11 + A22) * (B11 + B22)$

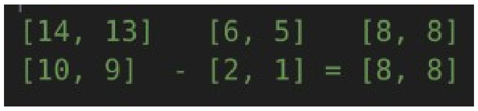
$P6 = (A12 - A22) * (B21 + B22)$

$P7 = (A11 - A21) * (B11 + B12)$

To calculate the quadrant C[0,0], we use a specific combination here:

$C11 = P5 + P4 - P2 + P6$

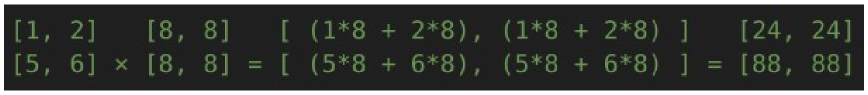
Due to the volume of calculations, I'll show how P1 is calculated and we can extrapolate from there. Below (Figure 7.2.3) is the matrix subtraction for (B12-B22) as the first step for calculating P1.



```
[14, 13]   [6, 5]   [8, 8]
[10, 9]  - [2, 1] = [8, 8]
```

Figure 7.2.3 – Matrix subtraction

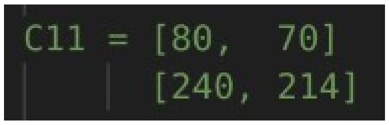
After the matrix subtraction of element-by-element we then multiply this resulting matrix by A11 as seen in Figure 7.2.4 below. Note: the equation form for my word document doesn't seem to do multi-line equations so I'm opting for images instead to give better visual clarity of the matrix.



```
[1, 2]   [8, 8]   [ (1*8 + 2*8), (1*8 + 2*8) ]   [24, 24]
[5, 6] × [8, 8] = [ (5*8 + 6*8), (5*8 + 6*8) ] = [88, 88]
```

Figure 7.2.4 – Matrix Multiplication

This process is then repeated for all seven P matrices. To find the resulting C11 matrix, we use the equation we outlined above which equates to this as seen in Figure 7.2.5:



```
C11 = [80,  70]
      [240, 214]
```

Figure 7.2.1 – Example Matrix

As we can see, C[0,0] is equal to 80 which matches the result from the standard matrix multiplication algorithm. While this process is more complex, the computational advantage is significant. Strassen’s algorithm treats the 4x4 matrix as separate 2x2 matrix multiplications. Each 2x2 matrix requires 8 multiplications but with only 7 multiplications for the entire matrix when combined, this works out to 56 total multiplication operations for the 4x4 matrix (7 * 8 = 56). Compared with the standard approach of $O(n^3)$ requiring 64 multiplications for the same matrix, this advantage begins to compound exponentially as the matrix size increases, leading to the difference of $O(n^{2.807})$ vs $O(n^3)$.

To add to this, by recursively working on smaller sub-matrices, the algorithm operates on blocks of data that can easily fit into the CPU’s fast cache memory. This avoids the cache saturation bottleneck, a critical performance limitation for the standard algorithm when calculating large matrices. The combination of fewer operations and optimised cache efficiency is a significant factor for accelerating large-scale computations in HPC and AI.

7.3 Project Structure

This project has several files to run the matrix multiplications and benchmark which can be broken up into 5 key areas:

1. The Build Script (**build.sh**): The shell script that automates the compiling and linking process. It uses the LLVM toolchain to assemble to project by compiling the written LLVM IR code into an object file then links it with the C driver code.
2. LLVM IR Implementation (**strassen_2x2.ll**): The core of the project, this file contains the hand-written Strassen's algorithm implemented in LLVM IR. This provides the low-level control over the operations and memory access.
3. Reference matrix multiplication (**reference_driver.c / reference_driver.h**): The optimized native C implementation of the standard O(n^3) matrix multiplication. This is used as the baseline comparison and to validate the correctness of the Strassen algorithm.
4. Main application (**strassen_driver.c**): This acts as the controller or main application of the project. It handles the command-line parsing for inputs, matrix allocation and initialization, validation testing, timing and the benchmarking functionality. It handles the calls to both the reference function and the external LLVM IR function.
5. Build output (**build/** directory): This directory contains the generated object file (**strassen_2x2.o**) and the final executable binary (**strassen_driver**) produced by the build script.

The workflow of this project operates like this: The main application (**strassen_driver.c**) initializes the matrices, runs correctness tests if not disabled, then calls the function **strassen_multiply** (functionality defined in the LLVM IR file). If benchmarking is enabled it iteratively runs the multiplication and records the time taken + calculates the GFLOPS. The build script (**build.sh**) is responsible for compiling the IR module and links it with the driver to create a single executable, combining the high-level C control flow with the low-level IR I wrote.

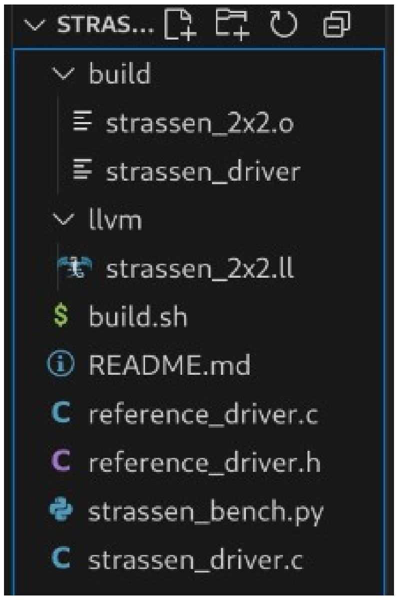
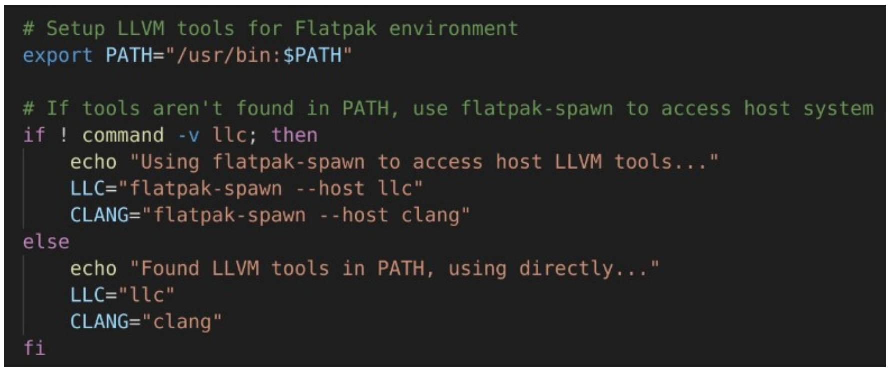


Figure 7.3.1 – Project Structure

7.4 Build.sh

The build.sh script automates the process of compiling and linking the project's C code and hand-written LLVM IR into a single executable binary file. As I've hand-written the LLVM IR code, this is a step I need to specify. It performs several key functions:

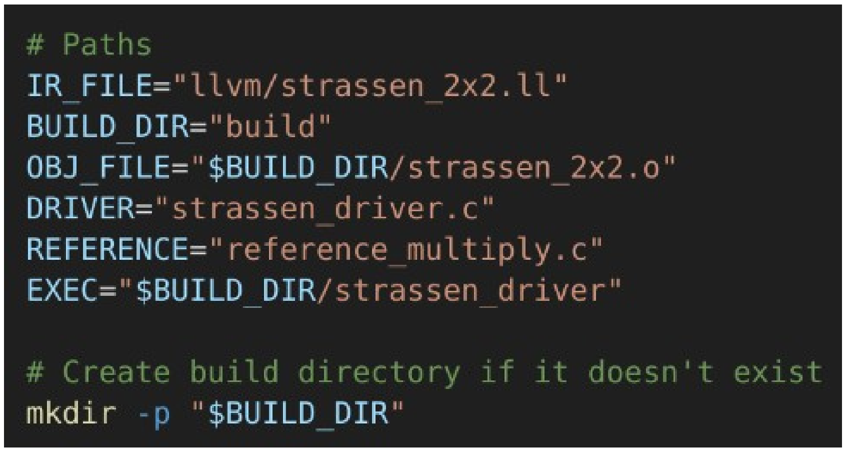
The script first ensures that it can access the necessary LLVM compiler tools like **llc** and **clang**. It's designed to be flexible, working in both a standard development environment and in a constrained environment like Flatpak (VS Code on my laptop is a flatpak install on Linux which is primarily why I had to include this). This can be seen in Figure 7.4.1 below:



```
# Setup LLVM tools for Flatpak environment
export PATH="/usr/bin:$PATH"

# If tools aren't found in PATH, use flatpak-spawn to access host system
if ! command -v llc; then
    echo "Using flatpak-spawn to access host LLVM tools..."
    LLC="flatpak-spawn --host llc"
    CLANG="flatpak-spawn --host clang"
else
    echo "Found LLVM tools in PATH, using directly..."
    LLC="llc"
    CLANG="clang"
fi
```

Figure 7.4.1 – Flatpak check

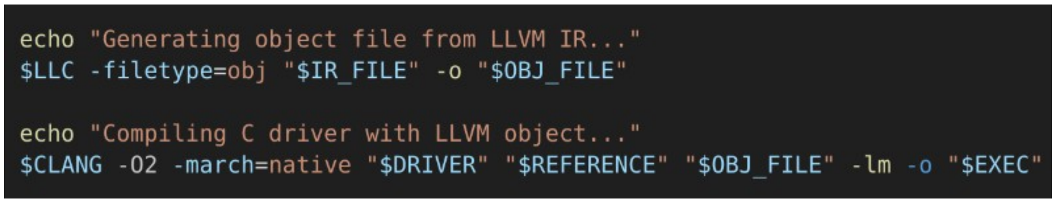
This conditional check makes the build process portable (can work in either standard or constrained environment). It first tries to use the tools available in the current **PATH**. If they're not found (i.e. in a sandboxed Flatpak environment), it uses **flatpak-spawn** to execute the tools from the host system.



```
# Paths
IR_FILE="llvm/strassen_2x2.ll"
BUILD_DIR="build"
OBJ_FILE="$BUILD_DIR/strassen_2x2.o"
DRIVER="strassen_driver.c"
REFERENCE="reference_multiply.c"
EXEC="$BUILD_DIR/strassen_driver"

# Create build directory if it doesn't exist
mkdir -p "$BUILD_DIR"
```

Figure 7.4.2 – Defined paths

The script also defines the necessary file paths and ensures the **build/** output directory exists as seen in above Figure 7.4.2. This just makes it easier to maintain and modify if every file is in the right place and has a consistent naming scheme.



```
echo "Generating object file from LLVM IR..."
$LLC -filetype=obj "$IR_FILE" -o "$OBJ_FILE"

echo "Compiling C driver with LLVM object..."
$CLANG -O2 -march=native "$DRIVER" "$REFERENCE" "$OBJ_FILE" -lm -o "$EXEC"
```

Figure 7.4.3 – Generating object file/binary

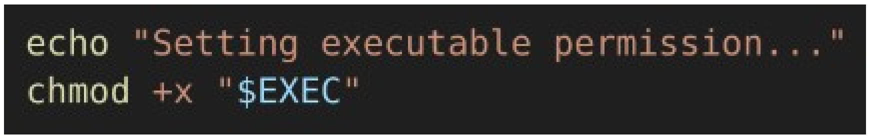
The next two steps are the most important. As seen in Figure 7.4.3 above, this is where the LLVM IR code is compiled, and then linked with the C driver. "**llc**" is the LLVM static compiler, the **-filetype=obj** flag compiles the LLVM IR file (**strassen_2x2.ll**) into a native object file (**strassen_2x2.o**) which contains the machine code for the **strassen_multiply** function. This step is what translates the platform-agnostic IR into the architecture-specific machine code like x86_64 or ARM to run on the CPU.

The next line uses **clang** to compile the C source file (**strassen_driver.c** and **reference_multiply.c**) and then links them with the previously generated object file **(strassen_2x2.o)**.

The (**-O2**) flag enables a high level of compiler optimisations for the C code, which ensures a fair performance comparison (given that I have manually optimised the LLVM IR code myself).

The (**-march=native**) flag just specifies the compiler to generate machine code optimized for the host CPU of the system, to enable maximum performance from the hardware.

The (**-lm**) flag links the standard math library which is needed for the matrix multiplications we're doing.



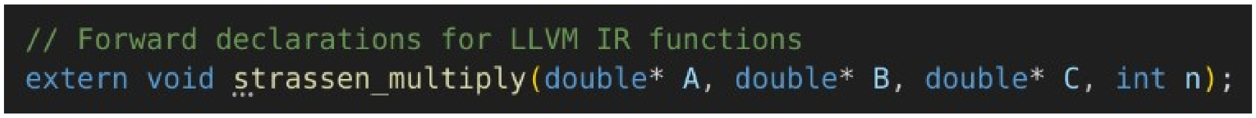
```
echo "Setting executable permission..."
chmod +x "$EXEC"
```

Figure 7.4.4 – Generating object file/binary

As seen in above Figure 7.4.4, I have to manually set the permissions for the executable as I'm running this on a Linux machine.

7.5 Strassen_driver.c

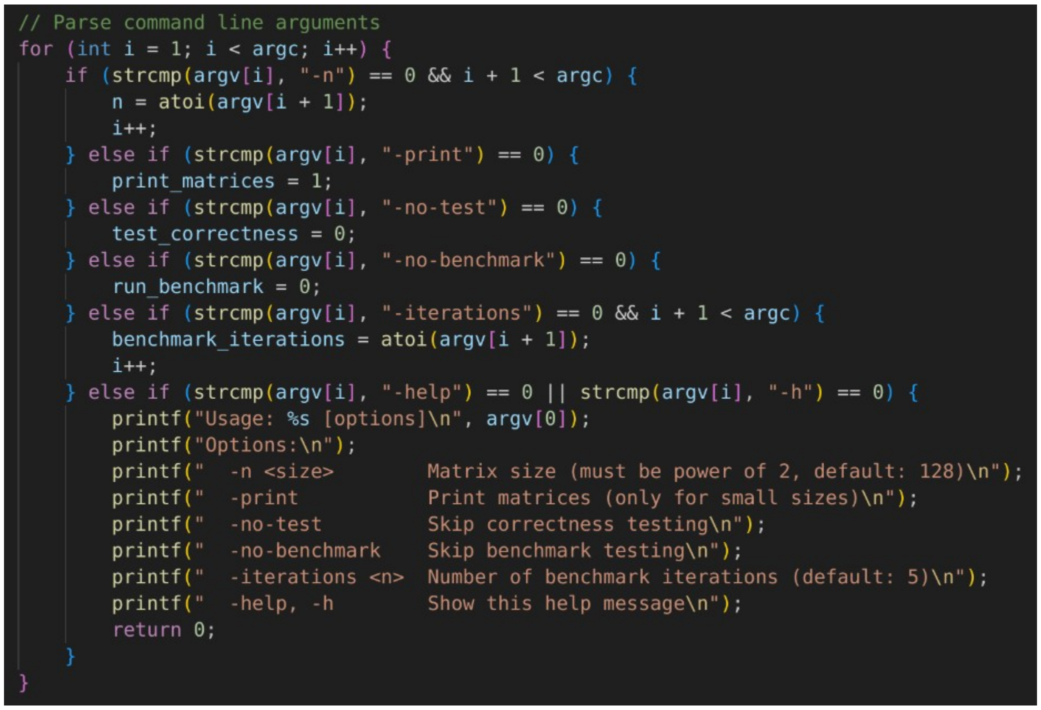
The **strassen_driver.c** file functions as the main controller application for this project. It defines the benchmarking process, handles user input, manages the memory, coordinates between the C matrix multiplication and the hand-written LLVM IR of Strassen's algorithm, validation testing, and reporting the results.



```
// Forward declarations for LLVM IR functions
extern void strassen_multiply(double* A, double* B, double* C, int n);
```

Figure 7.5.1 – Strassen_multiply declaration

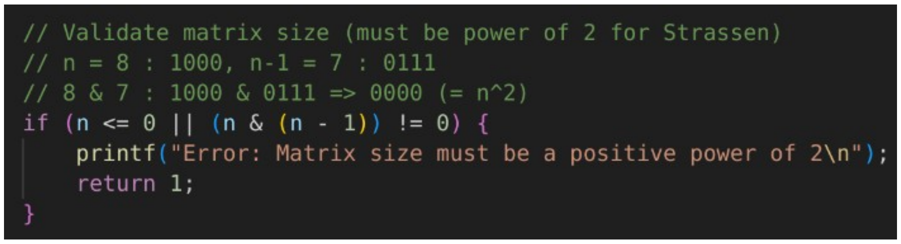
The declaration of strassen_multiply in Figure 7.5.1 is critical as it declares the interface to the function implemented in the LLVM IR file (**strassen_2x2.ll**). This allows the C code to call the hand-written IR implementation seamlessly after linking in the build script. The input arguments [A, B, C, and n] correspond with input matrix A, input matrix B, output matrix C, and the matrix size. The **extern** keyword tells the C compiler that this function will be resolved in the linking phase, which enables to be easily called later.



```c
// Parse command line arguments
for (int i = 1; i < argc; i++) {
    if (strcmp(argv[i], "-n") == 0 && i + 1 < argc) {
        n = atoi(argv[i + 1]);
        i++;
    } else if (strcmp(argv[i], "-print") == 0) {
        print_matrices = 1;
    } else if (strcmp(argv[i], "-no-test") == 0) {
        test_correctness = 0;
    } else if (strcmp(argv[i], "-no-benchmark") == 0) {
        run_benchmark = 0;
    } else if (strcmp(argv[i], "-iterations") == 0 && i + 1 < argc) {
        benchmark_iterations = atoi(argv[i + 1]);
        i++;
    } else if (strcmp(argv[i], "-help") == 0 || strcmp(argv[i], "-h") == 0) {
        printf("Usage: %s [options]\n", argv[0]);
        printf("Options:\n");
        printf("  -n <size>        Matrix size (must be power of 2, default: 128)\n");
        printf("  -print           Print matrices (only for small sizes)\n");
        printf("  -no-test         Skip correctness testing\n");
        printf("  -no-benchmark    Skip benchmark testing\n");
        printf("  -iterations <n>  Number of benchmark iterations (default: 5)\n");
        printf("  -help, -h        Show this help message\n");
        return 0;
    }
}
```

Figure 7.5.2 – User input

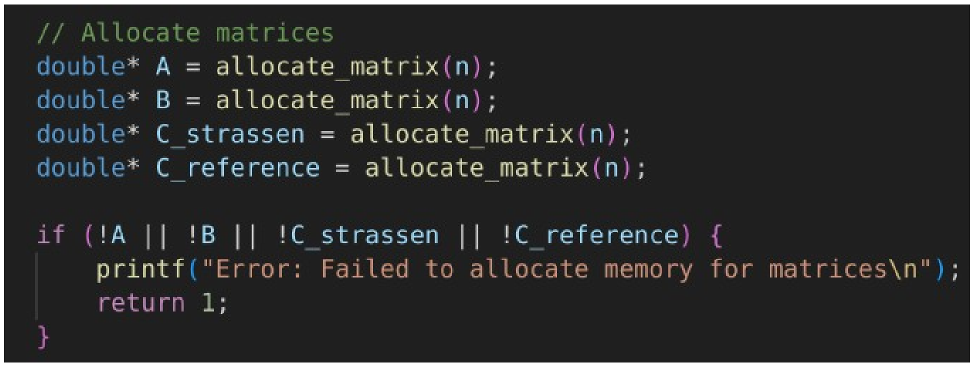
In Figure 7.5.2 above, this is how the program handles several command-line options to control the application. It allows various options like skipping benchmark tests, defining matrix sizes, iterations for repeated runs with the benchmark etc. C code isn’t pretty when it comes to reading strings, in future I’d like to rewrite this in Python to be more user-friendly.



```c
// Validate matrix size (must be power of 2 for Strassen)
// n = 8 : 1000, n-1 = 7 : 0111
// 8 & 7 : 1000 & 0111 => 0000 (= n^2)
if (n <= 0 || (n & (n - 1)) != 0) {
    printf("Error: Matrix size must be a positive power of 2\n");
    return 1;
}
```

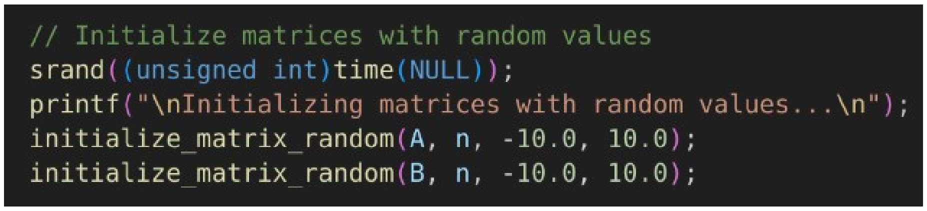
Figure 7.5.3 – Matrix size validation

Due to how Strassen’s algorithm subdivides matrices, the input matrix size needs to be a power of 2 in order to work correctly. In Figure 7.5.3 above I handle this error check. I had considered a number of ways to do this but ultimately the internet is a wonderful tool and this method was by far the most optimal. It basically does a byte subtraction to flip the bits (every power of 2 seems to have only one bit active), then bitwise adds those together. If the resulting byte value is 0, then it's a power of 2! It's quite ingenious, and far more efficient than other workarounds I tried to implement.



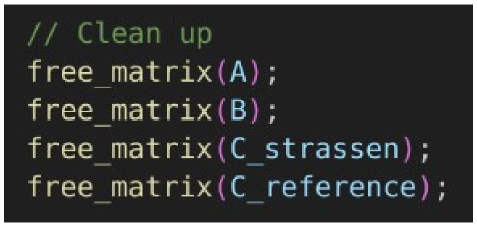
```c
// Allocate matrices
double* A = allocate_matrix(n);
double* B = allocate_matrix(n);
double* C_strassen = allocate_matrix(n);
double* C_reference = allocate_matrix(n);

if (!A || !B || !C_strassen || !C_reference) {
    printf("Error: Failed to allocate memory for matrices\n");
    return 1;
}
```

Figure 7.5.4 – Allocate matrix



```c
// Initialize matrices with random values
srand((unsigned int)time(NULL));
printf("\nInitializing matrices with random values...\n");
initialize_matrix_random(A, n, -10.0, 10.0);
initialize_matrix_random(B, n, -10.0, 10.0);
```

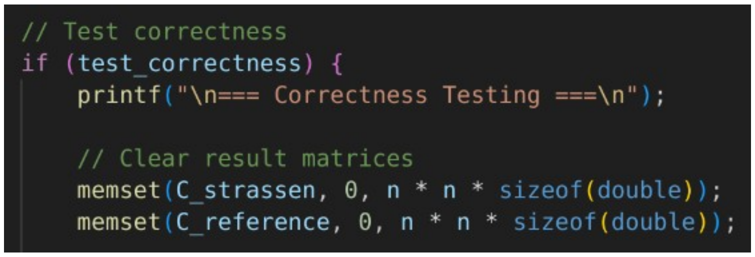
Figure 7.5.5 – Initialise matrix values



```c
// Clean up
free_matrix(A);
free_matrix(B);
free_matrix(C_strassen);
free_matrix(C_reference);
```

Figure 7.5.6 – Free matrices from memory

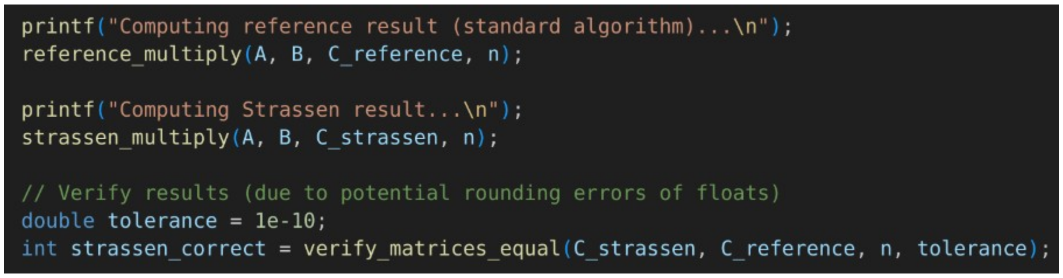
The C driver uses functions from **reference_multiply.h** to handle the memory allocation (Figure 7.5.4), matrix data initialisation (Figure 7.5.5) and memory de-allocation (Figure 7.5.6). I'll go into detail on these later when we look at this file, but for now just to point out that the memory management is called here.



```c
// Test correctness
if (test_correctness) {
    printf("\n=== Correctness Testing ===\n");

    // Clear result matrices
    memset(C_strassen, 0, n * n * sizeof(double));
    memset(C_reference, 0, n * n * sizeof(double));
```

Figure 7.5.7 – Test correctness, memset

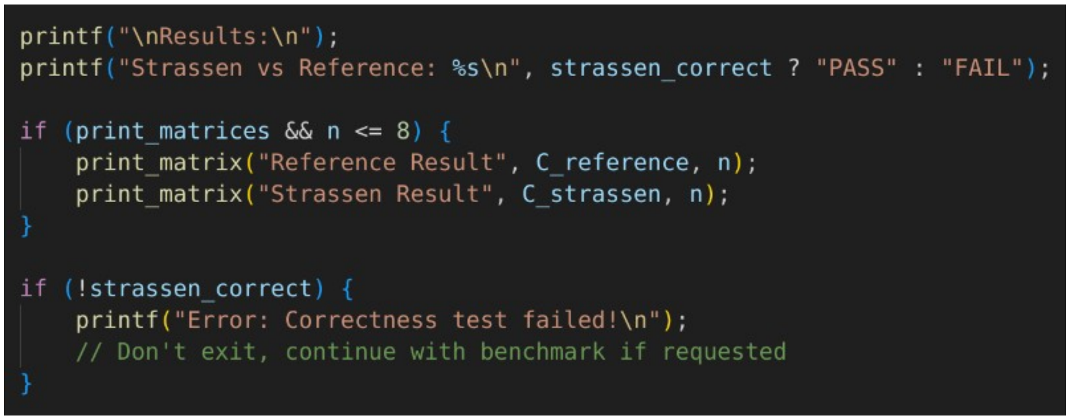
The correctness test is an important function. This essentially validates that the low-level LLVM IR implementation of Strassen's algorithm produces the same results as the straightforward standard algorithm. In Figure 7.5.7 this sets both matrices to a clean state (all zeroes). This is important because both **strassen_multiply** and **reference_multiply** functions will add their results to these output matrices. If there was any existing data in this memory, it would corrupt the result.



```c
printf("Computing reference result (standard algorithm)...\n");
reference_multiply(A, B, C_reference, n);

printf("Computing Strassen result...\n");
strassen_multiply(A, B, C_strassen, n);

// Verify results (due to potential rounding errors of floats)
double tolerance = 1e-10;
int strassen_correct = verify_matrices_equal(C_strassen, C_reference, n, tolerance);
```

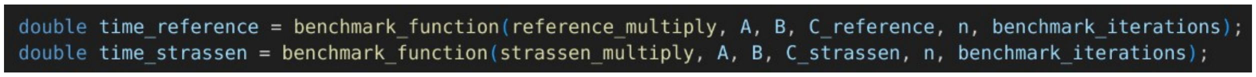
Figure 7.5.8 – Test correctness, operation

The execution of the multiplication with Strassen's and the reference is the core part of the test. They take the same inputs and the resulting matrices **C_reference** and **C_strassen** are then compared together in the **verify_matrices_equal** function defined in **reference_driver.c**. The tolerance factor included here is also important as the order of Strassen's algorithm can lead to minute differences in the very small decimals compared to the standard algorithm. The tolerance value defined here (1e-10 or 0.0000000001) defines the maximum allowable difference between any corresponding matrix elements for them to be considered "equal". Floating point rounding can be weird at small values so it was important to specify this. The test correctness was mostly an aid for debugging to compare values and see potential errors in the calculation with the prints in Figure 7.5.9 below.



```c
printf("\nResults:\n");
printf("Strassen vs Reference: %s\n", strassen_correct ? "PASS" : "FAIL");

if (print_matrices && n <= 8) {
    print_matrix("Reference Result", C_reference, n);
    print_matrix("Strassen Result", C_strassen, n);
}

if (!strassen_correct) {
    printf("Error: Correctness test failed!\n");
    // Don't exit, continue with benchmark if requested
}
```

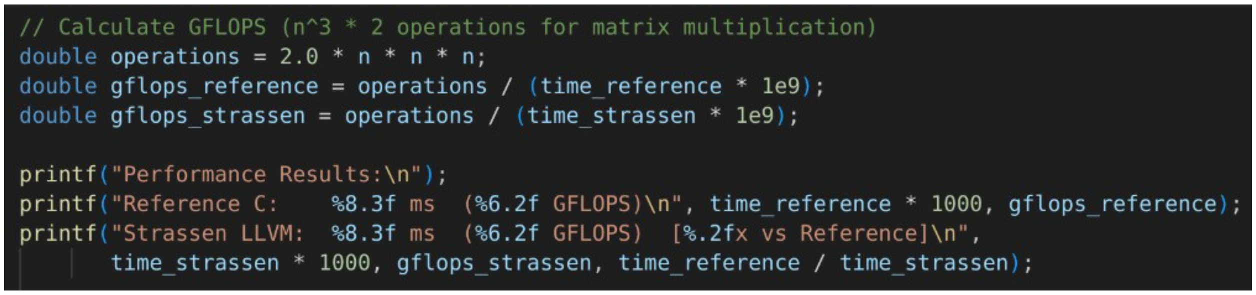
Figure 7.5.9 – Test correctness, print results



```c
double time_reference = benchmark_function(reference_multiply, A, B, C_reference, n, benchmark_iterations);
double time_strassen = benchmark_function(strassen_multiply, A, B, C_strassen, n, benchmark_iterations);
```

Figure 7.5.10 – Benchmark, input

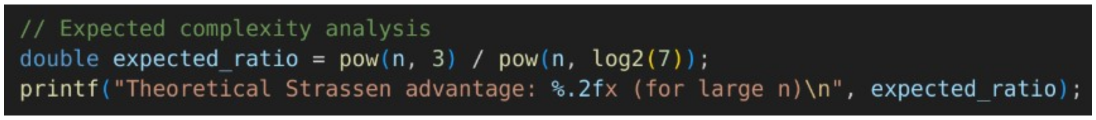
The benchmarking function as seen in Figure 7.5.10 above executes both algorithms multiple times (this is where the iterations argument comes in) for accurate time measurement.



```c
// Calculate GFLOPS (n^3 * 2 operations for matrix multiplication)
double operations = 2.0 * n * n * n;
double gflops_reference = operations / (time_reference * 1e9);
double gflops_strassen = operations / (time_strassen * 1e9);

printf("Performance Results:\n");
printf("Reference C:    %8.3f ms  (%6.2f GFLOPS)\n", time_reference * 1000, gflops_reference);
printf("Strassen LLVM:  %8.3f ms  (%6.2f GFLOPS)  [%.2fx vs Reference]\n",
       time_strassen * 1000, gflops_strassen, time_reference / time_strassen);
```

Figure 7.5.11 – Benchmark, GFLOPS

The benchmarking function also calculates the GFLOPS to determine the computational throughput using the formula above in Figure 7.5.11. The operations are defined as n^3 for matrix multiplication, and 1e9 is used for Giga-FLOPS given modern CPUs operate in the GHz range for calculations per second.



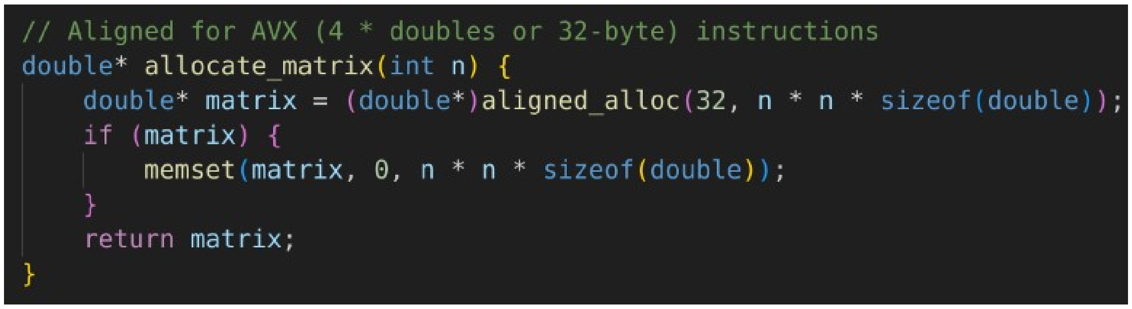
```c
// Expected complexity analysis
double expected_ratio = pow(n, 3) / pow(n, log2(7));
printf("Theoretical Strassen advantage: %.2fx (for large n)\n", expected_ratio);
```

Figure 7.5.11 – Benchmark, GFLOPS

Given that Strassen's algorithm operates with $O(n^{2.807})$ instead of $O(n^3)$, I used this equation to calculate the theoretical advantage of Strassen by dividing these together with the input matrix size. This essentially represents how many times more work the standard algorithm has to do to match Strassen's algorithm for this matrix size. It's a purely theoretical value that ignores all real-world constraints but it's a nice visual for the big matrices.

7.6 reference_driver.c

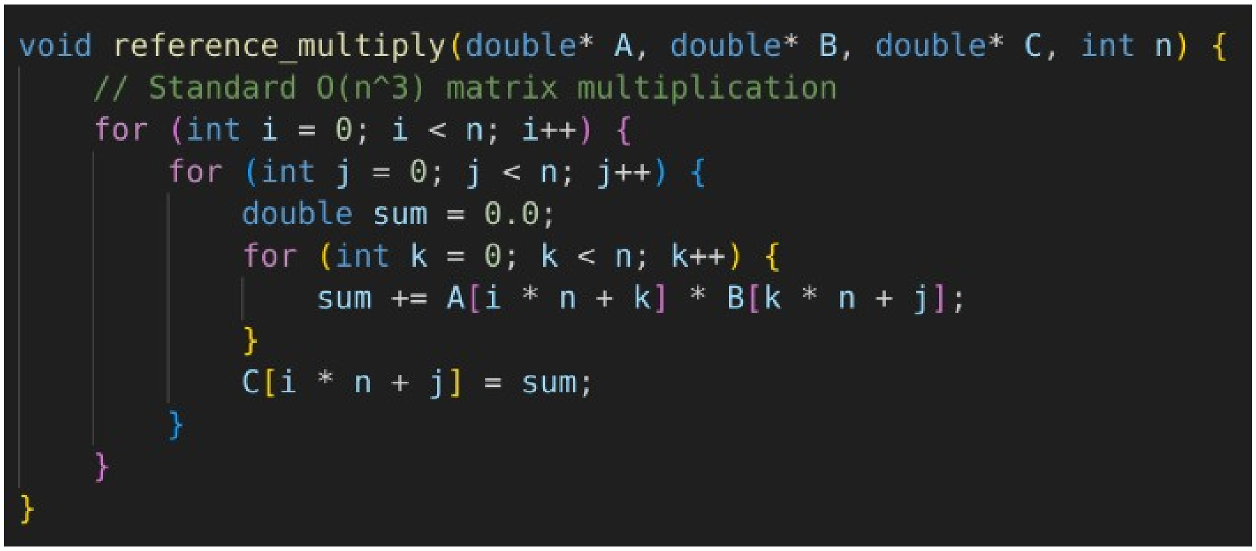
The **reference_driver.c** file holds many of the functions used in **strassen_driver.c** and the baseline implementation of the matrix multiplication that is compared against Strassen's algorithm.



```c
// Aligned for AVX (4 * doubles or 32-byte) instructions
double* allocate_matrix(int n) {
    double* matrix = (double*)aligned_alloc(32, n * n * sizeof(double));
    if (matrix) {
        memset(matrix, 0, n * n * sizeof(double));
    }
    return matrix;
}
```

Figure 7.6.1 – Matrix allocation

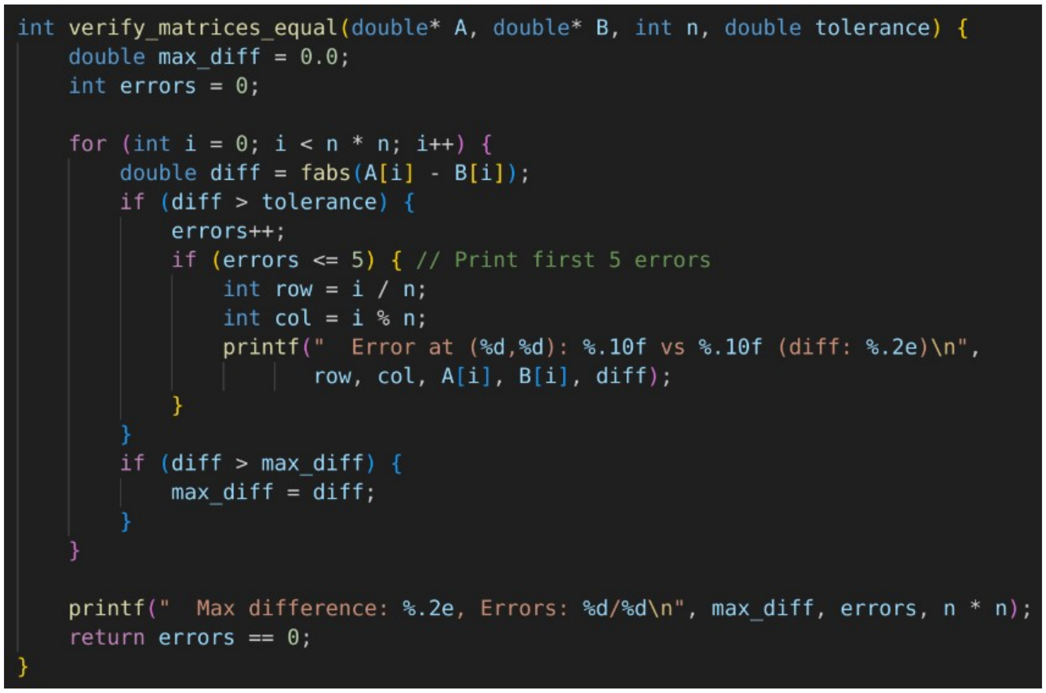
This **allocate_matrix** function uses **aligned_alloc**() instead of **malloc()** to ensure the matrices are 32-byte aligned, enabling AVX 256-bit (Single Instruction Multiple Destination) SIMD optimisations [8][10]. This basically means that it allows the registers to perform a single instruction on multiple pieces of data. Each register can hold 8x 32-bit single-precision floating-point numbers which is well optimised for our matrix multiplications. The more data elements we can squeeze into the registers, the more optimally the matrix multiplication can run as it lowers the chance of the CPU waiting to read data from memory (not cache). Memset is back again to immediately zero all values in the newly allocated matrix to ensure we're working with a clean slate.



```
void reference_multiply(double* A, double* B, double* C, int n) {
    // Standard O(n^3) matrix multiplication
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            double sum = 0.0;
            for (int k = 0; k < n; k++) {
                sum += A[i * n + k] * B[k * n + j];
            }
            C[i * n + j] = sum;
        }
    }
}
```

Figure 7.6.2 – Matrix multiplication standard

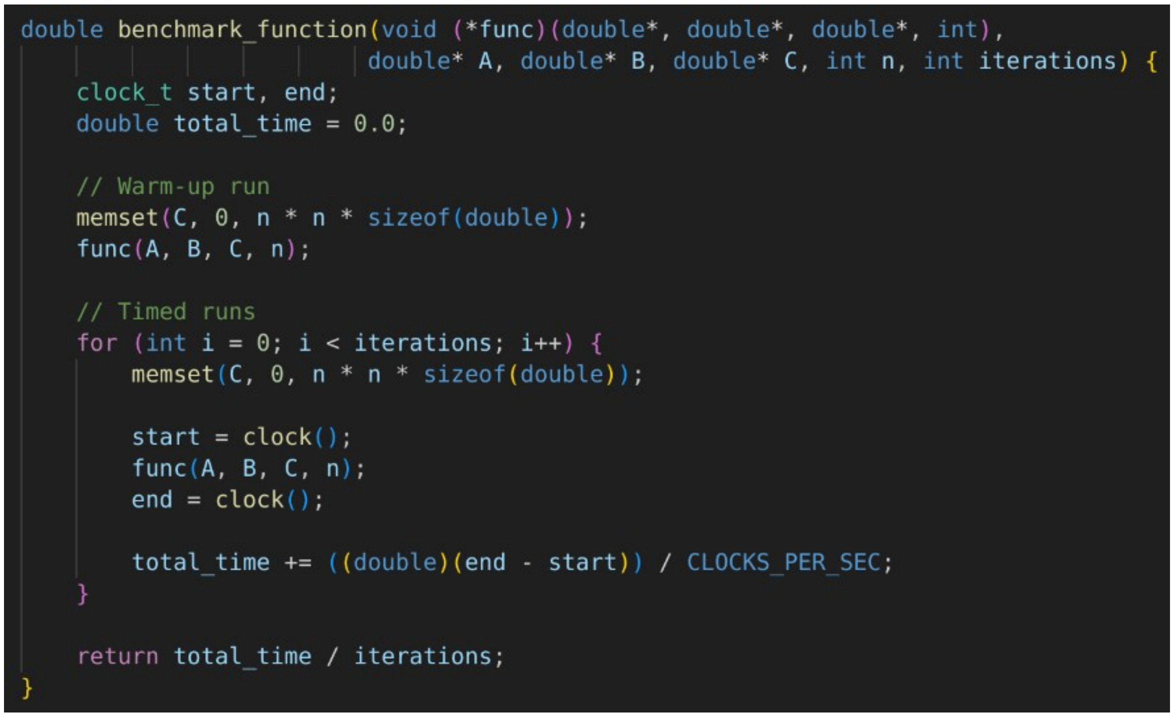
As seen in above Figure 7.6.2, this is the standard reference matrix multiplication that we use to benchmark against the Strassen's implementation. It's the classic triple-nested loop, using row-major ordering (i * n + k and k * n + j) and serves as the optimised baseline for performance comparison.



```c
int verify_matrices_equal(double* A, double* B, int n, double tolerance) {
    double max_diff = 0.0;
    int errors = 0;

    for (int i = 0; i < n * n; i++) {
        double diff = fabs(A[i] - B[i]);
        if (diff > tolerance) {
            errors++;
            if (errors <= 5) { // Print first 5 errors
                int row = i / n;
                int col = i % n;
                printf("  Error at (%d,%d): %.10f vs %.10f (diff: %.2e)\n",
                       row, col, A[i], B[i], diff);
            }
        }
        if (diff > max_diff) {
            max_diff = diff;
        }
    }

    printf("  Max difference: %.2e, Errors: %d/%d\n", max_diff, errors, n * n);
    return errors == 0;
}
```

Figure 7.6.3 – Verify matrices are equal

Pointing to Figure 7.6.3 above, this is the **verify_matrices_equal()** function we saw earlier. Floating-point calculations are prone to rounding errors and due to how Strassen's algorithm compounds addition and subtraction operations, those values can get quite large (and small!). Due to this, an exact comparison will nearly always fail so the solution was to include a level of tolerance that we defined in the **strassen_driver.c** file. If the absolute difference of the elements are above the tolerance, we increment the error count and print just the first 5 errors (so we don't get drowned by matrix values). This way we get some clean debugging in case something goes awry in the calculations. Also given that our tolerance is extremely small, we need to print to this decimal place in order to see the potential issue. The return value only returns 1 if the error count is zero, otherwise we get a simple boolean check to print PASS/FAIL at the other end.



```c
double benchmark_function(void (*func)(double*, double*, double*, int),
                          double* A, double* B, double* C, int n, int iterations) {
    clock_t start, end;
    double total_time = 0.0;

    // Warm-up run
    memset(C, 0, n * n * sizeof(double));
    func(A, B, C, n);

    // Timed runs
    for (int i = 0; i < iterations; i++) {
        memset(C, 0, n * n * sizeof(double));

        start = clock();
        func(A, B, C, n);
        end = clock();

        total_time += ((double)(end - start)) / CLOCKS_PER_SEC;
    }

    return total_time / iterations;
}
```

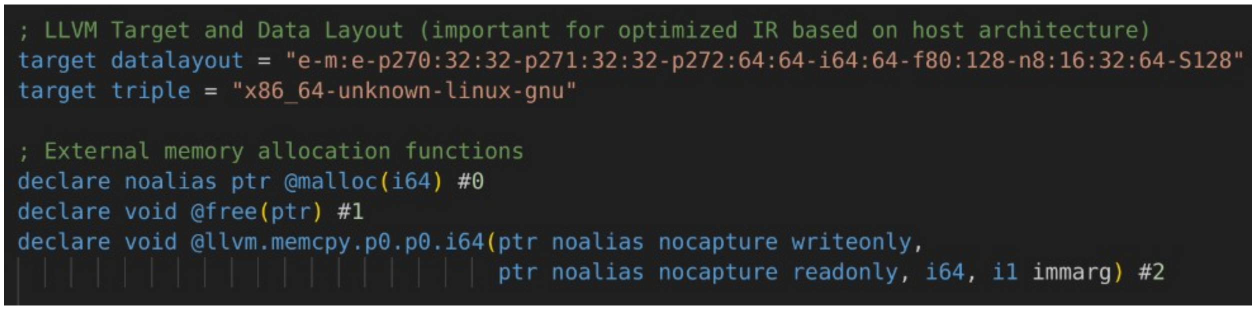
Figure 7.6.4– Benchmark function

The benchmark function as seen above in Figure 7.6.4 handles the benchmarking or time recording for the matrix multiplication. I wanted the same benchmark for both Strassen's and the reference matrix functions which is why I used the **void (*func)**. This is a function pointer and allows the benchmark to time any function that matches the corresponding signature (**double***, **double***, **double* int**).

The warmup run is important to initially execute the matrix multiplication algorithm once before timing it. This pre-loads the matrices A, B and C into the CPU's cache memory from RAM, and cleans out the C matrix of any garbage residual data. After the initial run, the clock time is saved at the start and end of each matrix multiplication run and the total time is saved. This time is then averaged out over the number of iterations to give a smoothed estimate of the time taken.

7.7 strassen_2x2.ll – Initialisation and Naive matrix

This is the most important aspect of the project, the hand-written LLVM IR code to handle the recursive operations of Strassen's algorithm, as well as defining the memory management, and target architecture this application runs on.



```
; LLVM Target and Data Layout (important for optimized IR based on host architecture)
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; External memory allocation functions
declare noalias ptr @malloc(i64) #0
declare void @free(ptr) #1
declare void @llvm.memcpy.p0.p0.i64(ptr noalias nocapture writeonly,
                                    ptr noalias nocapture readonly, i64, i1 immarg) #2
```

Figure 7.7.1– Target layout and optimisation attributes

The target datalayout string is an encoded description of the fundamental data layout for the target architecture (in my case, it's x86_64) [9]. This tells LLVM how to size, align and place the data types into memory.

"**e**" stands for Little-endian: This represents the least significant byte of a multi-byte value is stored at the lowest memory address. This is standard for x86 and x86_64 architectures.

"**m:e**": This is the mangling scheme for symbols, defining the ELF (Executable and Linkable Format), which is how the build and linking process from build.sh should operate. This is the standard for Linux systems.

"**p270-272**": These specify the alignment for pointers. For the specified address spaces (270, 271 and 272), these pointers are 32 bits, with a preferred alignment of 32 bits, where p272 is a 64-bit pointer with a preferred alignment of 64 bits. The alignment in this case means that the data within these memory registers are aligned accordingly to be read in a single step. For example if the CPU reads memory in 4-byte chunks (0-3), we want to store the pointer data at address 0. If it's stored in address 2, the pointer data may be spread across memory bytes that the CPU may need multiple reads to process and that can slow it down. This is a critical definition for the **aligned_alloc()** we've detailed earlier.

"**i64:64**": This specifies the alignment for 64-bit integers.

"**f80:128**": This is the alignment for 80-bit floats, that they should be aligned on a 128-bit boundary for optimal SIMD performance.

"**n8:16:32:64**": This defines the native integer widths that the target supports (8, 16, 32, 64 bits). This helps the optimiser with defined integer sizes.

"**S128**": This is the stack alignment with 128 bits (16 bytes). It's critical for ensuring that values on the stack are properly aligned (with above data type alignments defined above), especially for AVX instructions that require 16-byte alignment. (AVX 256 we're optimizing our memory for requires these byte values match for optimisation).

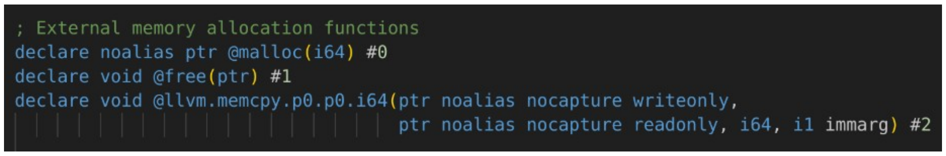
"**Target triple**": This is a simple identifier that describes the target machine (architecture, vendor, OS, ABI) format.

"**x86_64**": The CPU architecture of the host machine is AMD's 64-bit x86 (my laptop CPU).

"**unknown**": This is the vendor which is unspecified, this is moreso a brand label and not strictly required.

"**linux**": The target operating system is Linux.

"**gnu**": The application binary interface (ABI) is the GNU ABI, which defines the calling conventions, symbol naming, and library linking standards on Linux. Different OS uses different ABI's and this is again a critical definition.



```
; External memory allocation functions
declare noalias ptr @malloc(i64) #0
declare void @free(ptr) #1
declare void @llvm.memcpy.p0.p0.i64(ptr noalias nocapture writeonly,
                                    ptr noalias nocapture readonly, i64, i1 immarg) #2
```

Figure 7.7.2– Eternal memory allocation functions

These lines declare the functions that aren't defined in this IR file but will be available externally (from the standard C library) during the linking phase.

"**declare**": This informs LLVM that the function exists, but the implementation is elsewhere.

"**noalias**": This is strong optimisation hint, it promises that the pointer returned by the **malloc** doesn't alias or overlap in memory with any other pointers. This allows the optimiser to make assumptions it otherwise wouldn't, leading to more aggressive reordering and caching of memory (this helps alongside the alignment of our data types we specified earlier).

"**ptr @malloc(i64)**": The function named **malloc** takes one i64 (64-bit integer) argument and returns a pointer to the allocated memory.

"**void @free(ptr)**": The function named **free** takes a single pointer argument and returns void.

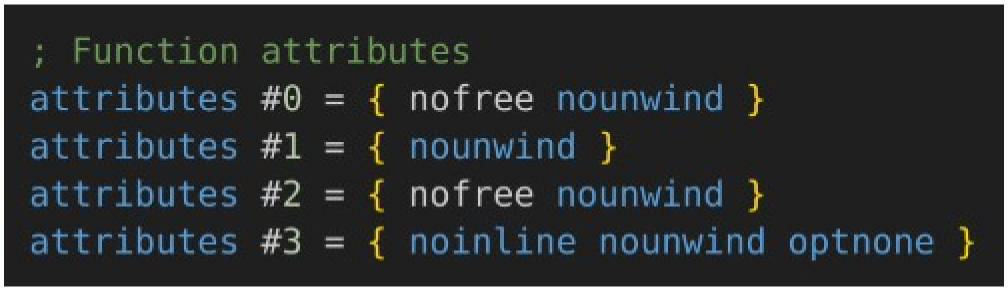
The last declare is **@llvm.memcpy**.

"**.p0.p0.i64**": This is the signature, it copies from p0 (pointer in address space 0) to a p0, using length of type i64.

"**nocapture**": The function doesn't "capture" the pointer and store it somewhere that outlives the function call.

"**writeonly/readonly**": The first pointer is only written to, and the second pointer is only read from. This helps the optimiser to eliminate redundant reads or writes.

"**i1 immarg**": The last argument is an i1 (1-bit integer or a boolean) that must be an immediate or constant value. It generally specifies whether the copy is volatile (which is usually false).



```
; Function attributes
attributes #0 = { nofree nounwind }
attributes #1 = { nounwind }
attributes #2 = { nofree nounwind }
attributes #3 = { noinline nounwind optnone }
```

Figure 7.7.3– Function Attributes

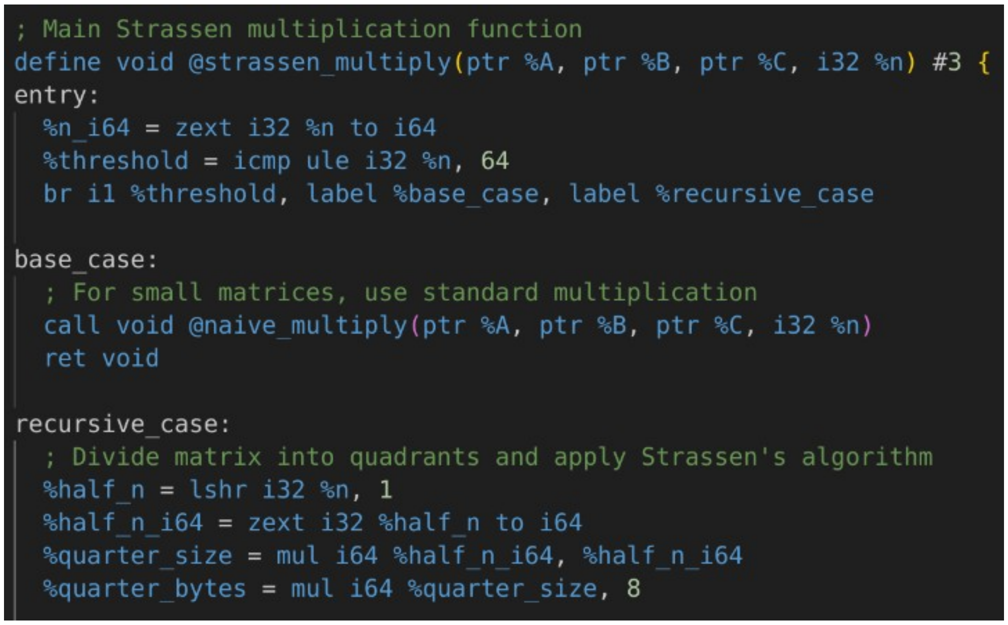
These attach specific properties to the declared and defined functions we highlighted above, which help with optimisation and code generation.

"**nofree**": The function doesn't free (deallocate) any memory. This isn't true for **free** function itself, so it doesn't have this attribute.

"**nounwind**": This function is guaranteed not to unwind the stack or throw an exception, this is pretty standard for C functions.

"**noinline**": This tells the optimize that it shouldn't inline this function. Due to the recursive calls of Strassen's algorithm, if the LLVM optimiser inlines a recursive call, it tries to copy the entire body of the function inside itself. This leads to the inlined body containing the recursive call which infinitely propagates until the compiler runs out of memory or reaches an internal limit. Even when it is stopped, it would create a massive unrolled version of the function for just one level of recursion which breaks the algorithm's logic and triggers a stack overflow.

"**optnone**": This disables all optimizations on this function. As I've specified all the memory and branching operations myself, any automated optimisations the compiler may attempt to do will end up obscuring the performance measurements I'm trying to take when comparing this to the reference matrix multiplication. If I don't set this, the compiler may alter the algorithms and memory management I'm trying to do and that would heavily skew the results.



```
; Main Strassen multiplication function
define void @strassen_multiply(ptr %A, ptr %B, ptr %C, i32 %n) #3 {
entry:
  %n_i64 = zext i32 %n to i64
  %threshold = icmp ule i32 %n, 64
  br i1 %threshold, label %base_case, label %recursive_case

base_case:
  ; For small matrices, use standard multiplication
  call void @naive_multiply(ptr %A, ptr %B, ptr %C, i32 %n)
  ret void

recursive_case:
  ; Divide matrix into quadrants and apply Strassen's algorithm
  %half_n = lshr i32 %n, 1
  %half_n_i64 = zext i32 %half_n to i64
  %quarter_size = mul i64 %half_n_i64, %half_n_i64
  %quarter_bytes = mul i64 %quarter_size, 8
```

Figure 7.7.4– Entry point of Strassen

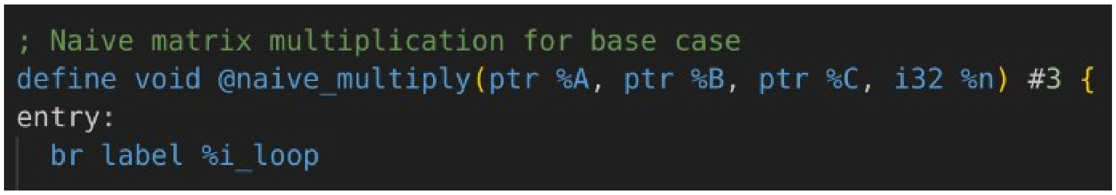
In Figure 7.7.4 above, this is the start of the strassen_multiply() function as defined by the "**entry**" tag. This is an important part of the code as it specifies whether to apply Strassen's algorithm or standard multiplication (for small matrices). The first line is a type conversion on the input variable **%n**, this **zext** (zero extends) the 32-bit integer to a 64-bit integer (integer data type denoted with "i"). This function is called from C with a 32-bit integer but memory allocation and address calculations in LLVM IR for a 64-bit system uses 64-bit integers (when using malloc as an example). This just prepares the input so the data types match.

"**%threshold = icmp ule i32 %n, 64**": This is an unsigned integer compare on the matrix size variable n, if it's less than or equal to 64 which is our matrix threshold for standard or Strassen's multiplication.

The following line "**br i1 %threshold**" is a conditional branch instruction, like an if else statement. If **n** is less than or equal to 64, it jumps to the base_case, otherwise it goes to recursive_case.

base_case is the standard multiplication path for small matrices (<= 64) that uses the simple multiplication. It calls the **@naive_multiply** function and returns back to the caller of the

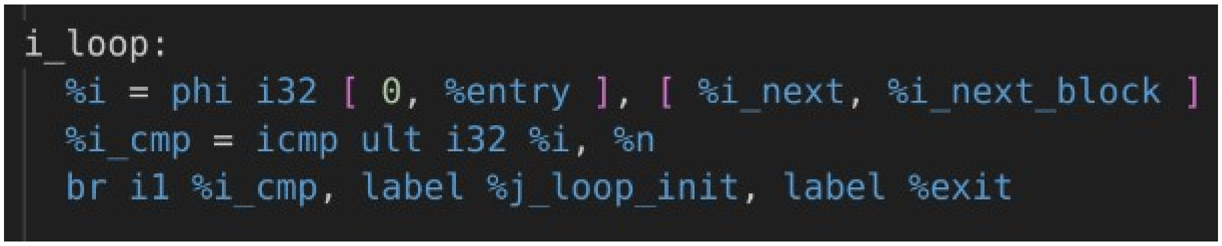
strassen_multiply function which would be the main application **strassen_driver.c**. The inputs are the same input matrix A, input matrix B, output matrix C, and matrix size n that we've covered in the **strassen_driver.c** file.



```
; Naive matrix multiplication for base case
define void @naive_multiply(ptr %A, ptr %B, ptr %C, i32 %n) #3 {
entry:
  br label %i_loop
```

Figure 7.7.5– Entry point of naive matrix multiplication

As seen in above Figure 7.7.5, this is the entry point of the reference matrix multiplication using O(n^3) to calculate C = A[i] * B[j].

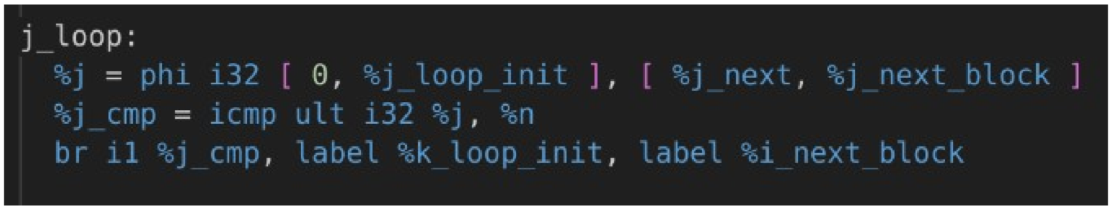


```
i_loop:
  %i = phi i32 [ 0, %entry ], [ %i_next, %i_next_block ]
  %i_cmp = icmp ult i32 %i, %n
  br i1 %i_cmp, label %j_loop_init, label %exit
```

Figure 7.7.6– Outer loop

This function implements the three nested loops we saw in **reference_driver.c** using LLVM's basic blocks (think of it as a mini function) and phi nodes. The phi node helps keep track of where we are in the loop (which iteration). It initialises **%i** to zero on the first iteration when coming from the entry block, then assigns **%i** to the value of **%i_next** when coming from **%i_next_block**. This is the loop variable for the outer loop of the matrix multiplication (the rows).

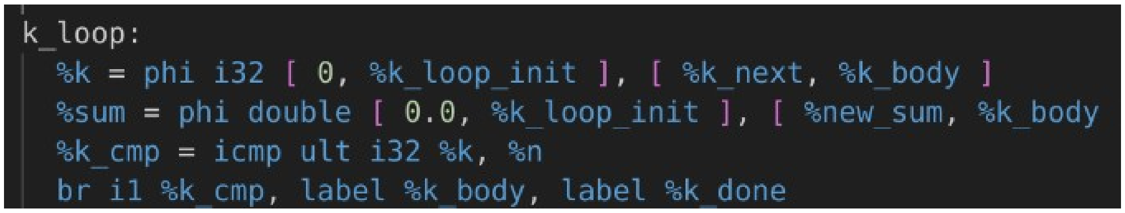
It then does an integer compare on **%i** to check if it's unsigned less than the matrix size, and breaks to either **%j_loop_init** or **%exit** if it has hit the limit of the matrix size (it's finished calculating).



```
j_loop:
  %j = phi i32 [ 0, %j_loop_init ], [ %j_next, %j_next_block ]
  %j_cmp = icmp ult i32 %j, %n
  br i1 %j_cmp, label %k_loop_init, label %i_next_block
```

Figure 7.7.6– Middle loop

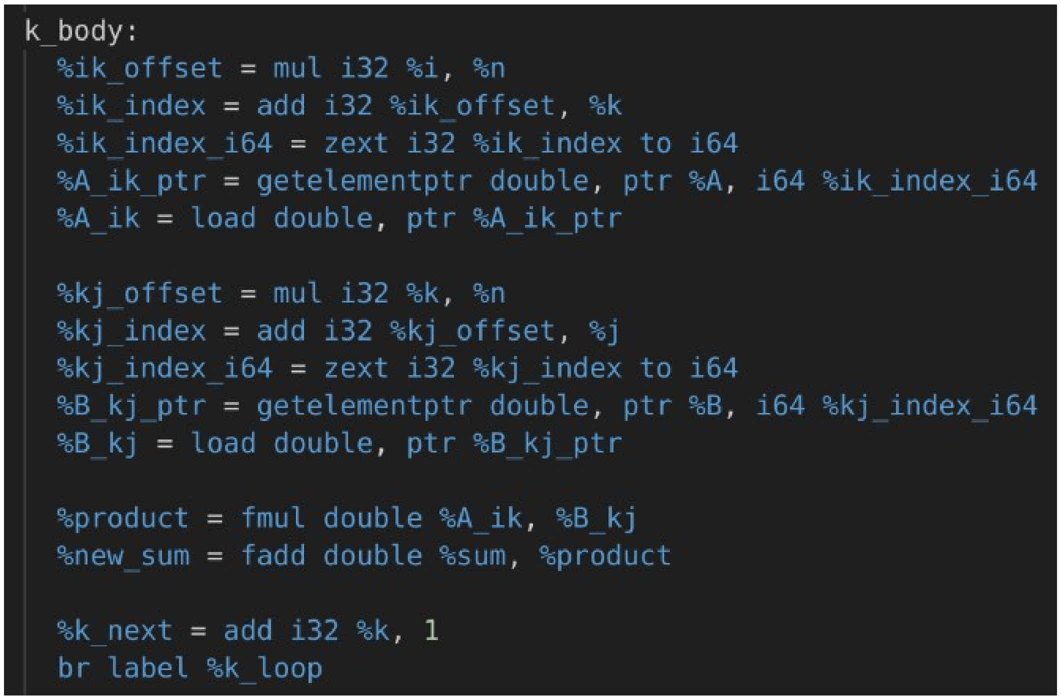
j_loop is the middle loop of the matrix multiplication, it does the same thing as the outer loop, initializing **%j** to zero if coming from the init block, or using the incremented value if the loop has iterated. This then compares again to either go to the inner loop or exits back.



```
k_loop:
  %k = phi i32 [ 0, %k_loop_init ], [ %k_next, %k_body ]
  %sum = phi double [ 0.0, %k_loop_init ], [ %new_sum, %k_body
  %k_cmp = icmp ult i32 %k, %n
  br i1 %k_cmp, label %k_body, label %k_done
```

Figure 7.7.7– Inner loop

k_loop is the inner loop which is the core computation loop stage. It uses two phi nodes for **%k** (the count variable for the inner loop) and **%sum** which accumulates the sum of products for the dot product of the matrix element being calculated. **%sum** is initialised to 0.0 at the start and updates it's value from **%new_sum** in the loop body.



```
k_body:
  %ik_offset = mul i32 %i, %n
  %ik_index = add i32 %ik_offset, %k
  %ik_index_i64 = zext i32 %ik_index to i64
  %A_ik_ptr = getelementptr double, ptr %A, i64 %ik_index_i64
  %A_ik = load double, ptr %A_ik_ptr

  %kj_offset = mul i32 %k, %n
  %kj_index = add i32 %kj_offset, %j
  %kj_index_i64 = zext i32 %kj_index to i64
  %B_kj_ptr = getelementptr double, ptr %B, i64 %kj_index_i64
  %B_kj = load double, ptr %B_kj_ptr

  %product = fmul double %A_ik, %B_kj
  %new_sum = fadd double %sum, %product

  %k_next = add i32 %k, 1
  br label %k_loop
```

Figure 7.7.8– k_body, the calculations

The inner loop body **k_body** is where the actual multiplication and accumulation is happening for the matrix multiplication. First we have to multiply the row index **%i** by the total number of columns **%n**. In row-major order, the entire row of **n** elements is stored consecutively so to get to the start of row **%i**, we have to skip **i** rows so to get to the first element in row 2, we have to do ***I = I * n***.

Now that we're on the correct row for our loop iteration, we can add **%k** elements to get to the correct column within that row. The next line just does a type conversion with **zext** again from i32 to i64. Memory addresses are 64 bits wide in 64-bit systems so we need to account for that with our data types.

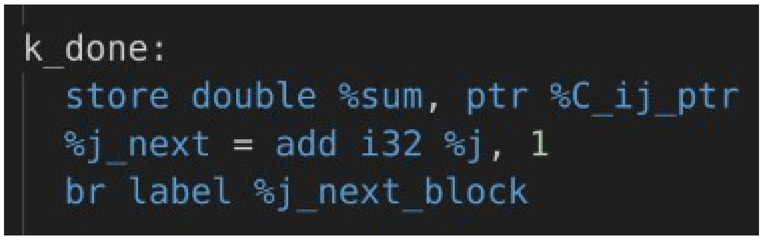
"**getelementptr**": This is the get element pointer instruction in LLVM. It's the safe way to perform pointer arithmetic, calculating the memory address of the element **%A_ik_ptr** from the base array pointer **%A** and the offset **%ik_index_i64** [3]. It essentially scales the index by the size of the data type (in this case we've provided **double** which is 8 bytes). We could do this manually with a **mul** operation then add it to **%A** but this can cause errors. LLVM can do this automatically and correctly with **getelementptr** so I'll use it's built-in functions where possible. This is the IR equivalent of creating a pointer reference to the location of the element's memory. Note, this isn't accessing the memory (such as read, load or store), this is just calculating the location of the memory of the element we want to use **for** the calculation.

"**%A_ik = load double**": This is a load instruction that reads the 8-byte double value from the memory address **%A_ik_ptr** and stores it in the register **%A_ik**. This is essentially the end goal for navigating through the matrix and finding the element we want to manipulate (perform the matrix multiplication with). The value A[i][k] is now in a register, ready to be used for multiplying in the next part of the algorithm. For the sake of not repeating information, the exact same process happens with **%kj_offset** with matrix B.

Now that the matrix values have been collected and loaded into memory we can directly use, we can use **fmul** to float multiply the **%A_ik** and **%B_kj** values together, saving them in the variable **%product**. This product is then added to **%new_sum** with the existing **%sum** variable.

Finally we increment **%k,** our loop count by 1 with the **add** function and loop back around again.

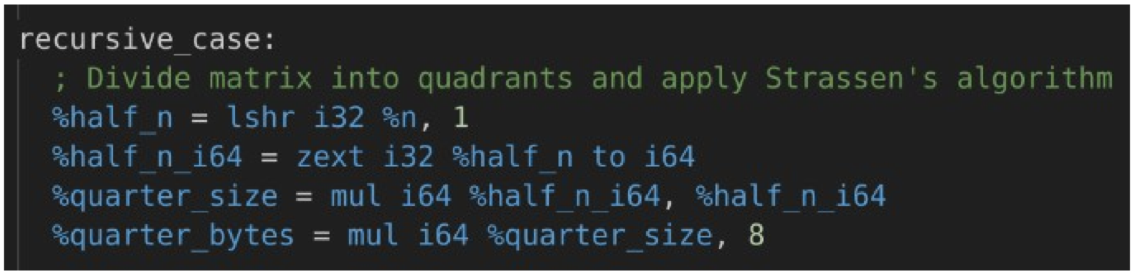
When the inner loop is completed, the accumulated sum is stored in the resulting matrix to the pre-calculated address **%C_ij_ptr** as seen below in Figure 7.7.9:



```
k_done:
  store double %sum, ptr %C_ij_ptr
  %j_next = add i32 %j, 1
  br label %j_next_block
```

Figure 7.7.9– k_done, store the value

7.8 strassen_2x2.ll – Strassen's recursion



```
recursive_case:
  ; Divide matrix into quadrants and apply Strassen's algorithm
  %half_n = lshr i32 %n, 1
  %half_n_i64 = zext i32 %half_n to i64
  %quarter_size = mul i64 %half_n_i64, %half_n_i64
  %quarter_bytes = mul i64 %quarter_size, 8
```

Figure 7.8.1– recursive case – prep

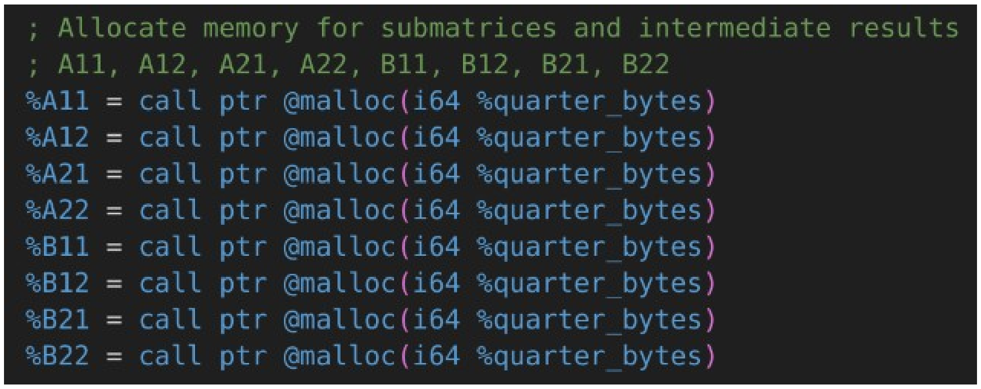
Now that the basic matrix multiplication process is complete, let's go over Strassen's recursion.

Given that Strassen predominately subdivides matrices into smaller quadrants, we have to start by dividing matrix size **n** into smaller pieces. We can do a very fast division by executing a bit-shift right (**lshr**) on the **%n** value, which divides it by 2 as seen in Figure 7.8.1 above. This specifies that each quadrant with be n/2 by n/2 size. (Taking a 4x4 matrix and allocating a 2x2 instead).

Next we zero extend (**zext**) the i32 **%half_n** to i64 as again, any memory management we do (we're going to start doing malloc's), we want to make sure the data types are aligned with the working memory addresses for our 64-bit system.

From there we calculate the number of elements in a single quadrant. We can do this by squaring (multiplying by itself) the **%half_n_64** we zero extended to get the total number of data elements each sub-matrix will contain.

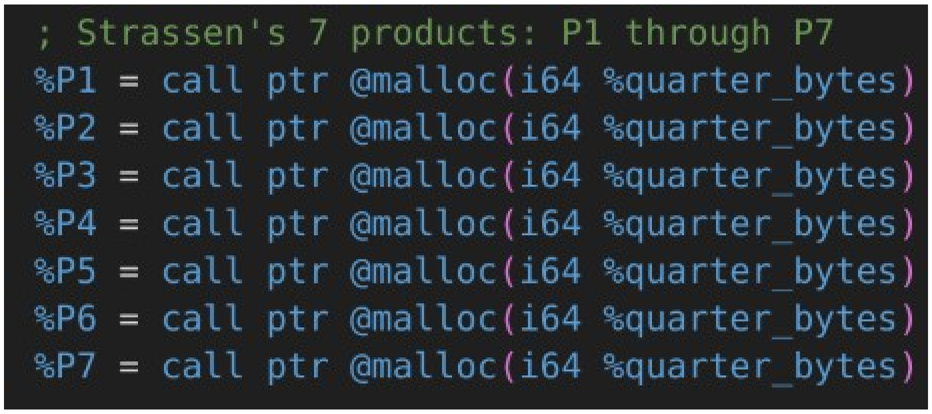
Finally we can convert the element count into the actual memory requirement by multiplying the int 64 **%quarter_size** by 8 to get the byte value, giving us the total bytes needed for each quadrant.



```
; Allocate memory for submatrices and intermediate results
; A11, A12, A21, A22, B11, B12, B21, B22
%A11 = call ptr @malloc(i64 %quarter_bytes)
%A12 = call ptr @malloc(i64 %quarter_bytes)
%A21 = call ptr @malloc(i64 %quarter_bytes)
%A22 = call ptr @malloc(i64 %quarter_bytes)
%B11 = call ptr @malloc(i64 %quarter_bytes)
%B12 = call ptr @malloc(i64 %quarter_bytes)
%B21 = call ptr @malloc(i64 %quarter_bytes)
%B22 = call ptr @malloc(i64 %quarter_bytes)
```

Figure 7.8.2– recursive case – prep, allocate submatrix memory

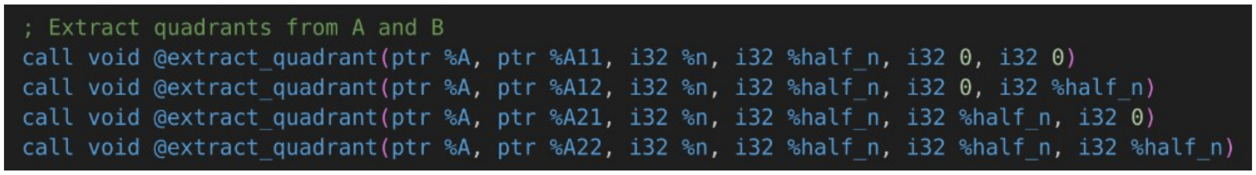
As we can see in Figure 7.8.2, now that we've defined the memory required for each quadrant, we can now allocate the memory for these using **@malloc**. This requires an input of the i64 data type we prepared earlier, as well as the memory for each quarter we've also defined. We'll store these as the labels for each quadrant later on. The **call** argument tells the program to pause its current execution, jump to the code for **@malloc**, run it, then return here with the result.



```
; Strassen's 7 products: P1 through P7
%P1 = call ptr @malloc(i64 %quarter_bytes)
%P2 = call ptr @malloc(i64 %quarter_bytes)
%P3 = call ptr @malloc(i64 %quarter_bytes)
%P4 = call ptr @malloc(i64 %quarter_bytes)
%P5 = call ptr @malloc(i64 %quarter_bytes)
%P6 = call ptr @malloc(i64 %quarter_bytes)
%P7 = call ptr @malloc(i64 %quarter_bytes)
```

Figure 7.8.3– recursive case – prep, allocate products memory

As we did with allocating the submatrix memory, we'll do the same with the product memory which stores the result of the seven operations that Strassen's algorithm will do as can be seen in above Figure 7.8.3.



```
; Extract quadrants from A and B
call void @extract_quadrant(ptr %A, ptr %A11, i32 %n, i32 %half_n, i32 0, i32 0)
call void @extract_quadrant(ptr %A, ptr %A12, i32 %n, i32 %half_n, i32 0, i32 %half_n)
call void @extract_quadrant(ptr %A, ptr %A21, i32 %n, i32 %half_n, i32 %half_n, i32 0)
call void @extract_quadrant(ptr %A, ptr %A22, i32 %n, i32 %half_n, i32 %half_n, i32 %half_n)
```

Figure 7.8.4– recursive case – prep, allocate products memory

The **extract_quadrant** (Figure 7.8.4 above) function copies the submatrices from the source to the destination. This functions in the same way as the setup from the reference matrix multiplication by iterating through the loops, finding the data elements we want and copies it.

extract_quadrant takes in the following inputs:

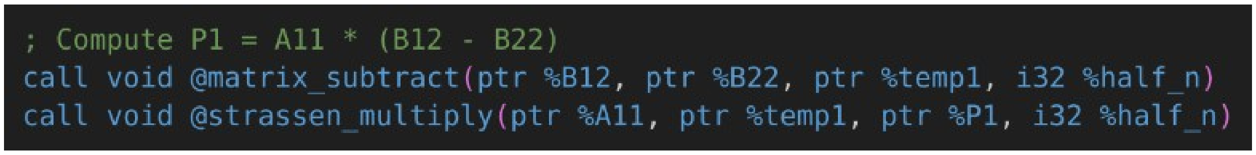
%src: The source matrix pointer

%dest: The destination matrix pointer

%src_n: Size of the source matrix

%dest_n: Size of the destination matrix (quadrant size)

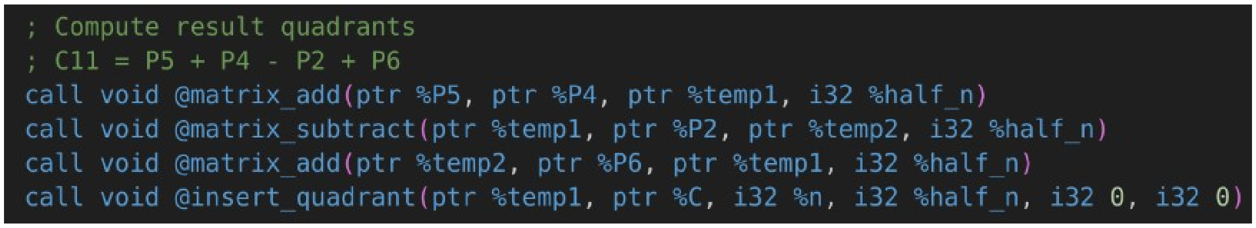
%row_offset, %col_offset: Position of quadrant within the source matrix



```
; Compute P1 = A11 * (B12 - B22)
call void @matrix_subtract(ptr %B12, ptr %B22, ptr %temp1, i32 %half_n)
call void @strassen_multiply(ptr %A11, ptr %temp1, ptr %P1, i32 %half_n)
```

Figure 7.8.5– recursive case – Calculating P1

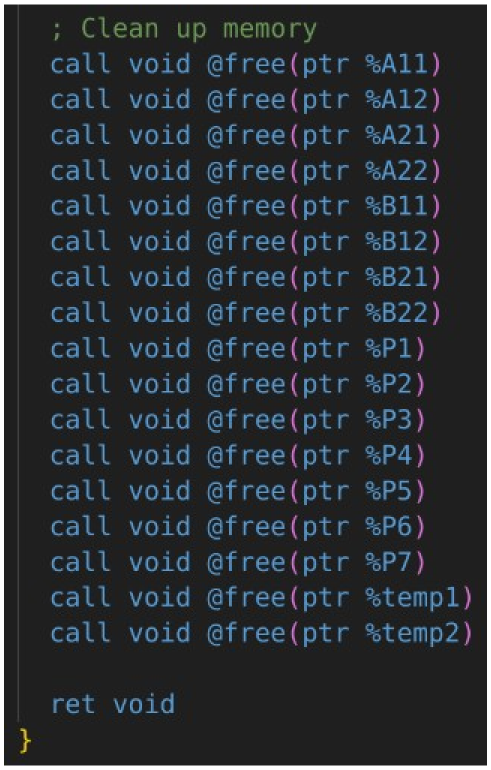
The seven products follow the same patten as P1 (Figure 7.8.5 above). First matrix_subtract is called and executed, subtracting B12 and B22, stores the result into **%temp1**, then when finished, the **call** function returns here and immediately multiplies %**A11** with it, storing the result into **%P1.**



```
; Compute result quadrants
; C11 = P5 + P4 - P2 + P6
call void @matrix_add(ptr %P5, ptr %P4, ptr %temp1, i32 %half_n)
call void @matrix_subtract(ptr %temp1, ptr %P2, ptr %temp2, i32 %half_n)
call void @matrix_add(ptr %temp2, ptr %P6, ptr %temp1, i32 %half_n)
call void @insert_quadrant(ptr %temp1, ptr %C, i32 %n, i32 %half_n, i32 0, i32 0)
```

Figure 7.8.6– recursive case – Calculating Submatrix 1

Once all the products have been calculated, then the submatrix quadrants are calculated together using the same **@matrix_add** and **@matrix_subtract**. The recursion with calculating the products runs again with the use of **call**, jumping in and out and recursively adding and subtracting itself, before finally running **@insert_quadrant** which performs the inverse of **@extract_quadrant**, copying the results back to the appropriate position in the output matrix C.



```
  ; Clean up memory
  call void @free(ptr %A11)
  call void @free(ptr %A12)
  call void @free(ptr %A21)
  call void @free(ptr %A22)
  call void @free(ptr %B11)
  call void @free(ptr %B12)
  call void @free(ptr %B21)
  call void @free(ptr %B22)
  call void @free(ptr %P1)
  call void @free(ptr %P2)
  call void @free(ptr %P3)
  call void @free(ptr %P4)
  call void @free(ptr %P5)
  call void @free(ptr %P6)
  call void @free(ptr %P7)
  call void @free(ptr %temp1)
  call void @free(ptr %temp2)

  ret void
}
```

Figure 7.8.7– recursive case – Wipe memory

Now that all the calculations are finished, we deallocate the pointer memory we used for this matrix to clean up and return back to the high-level C application as seen in above Figure 7.8.7.

There are a few key characteristics with Strassen's algorithm and how it was implemented. Firstly, compared with the reference matrix multiplication, there is deep recursion with the algorithm recursively dividing the problem until reaching the base case, where it creates a call tree that grows exponentially with matrix size.

Secondly it is very memory intensive. The implementation requires $O(n^2)$ additional memory for temporary matrices, which is a significant trade-off for the reduced computational complexity (of significantly more multiplication operations with the base matrix multiplication by comparison).

Overall though, having such strict control over the memory allocation, recursion and logical branching of the SCF (Structured Control Flow) with the LLVM IR allows for heavily optimized code that allows for optimisations that may by obscured in higher-level languages.

8 Ethics

Artificial intelligence is a rapidly expanding field that is raising growing concern over the environmental footprint of large-scale computing, particularly in AI data centers.

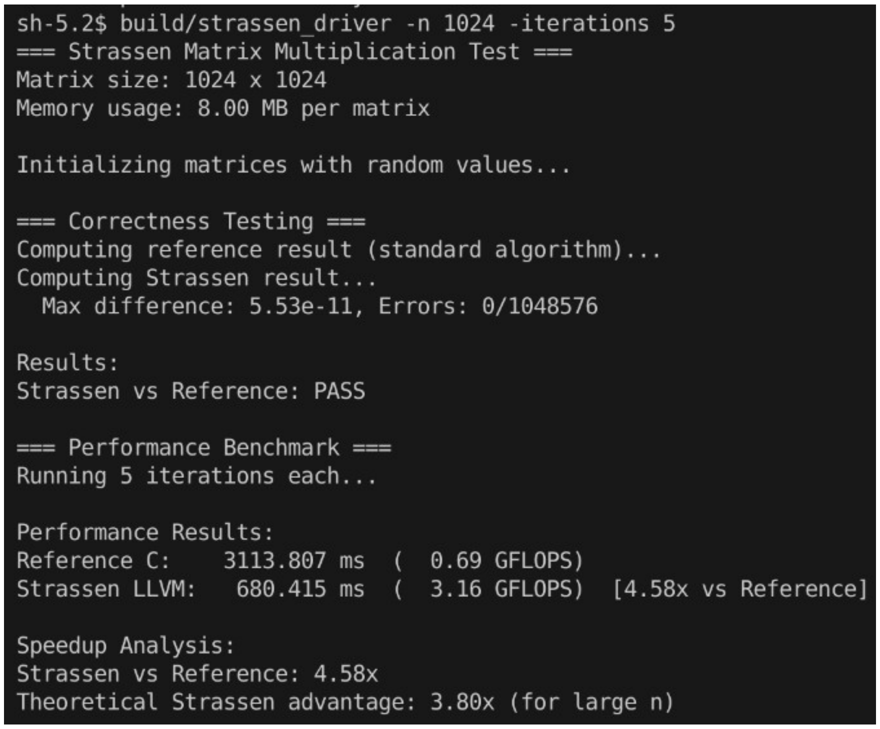
The training and operation of large machine learning modules consumes vast amounts of electrical energy which contributes to a significant carbon footprint. As the demand for AI capabilities and adoption increases, the strain on global and domestic energy resources grows alongside it. This raises serious ethical questions about the sustainability and responsibility this has towards our environmental impact.

This project mainly addresses this concern at the algorithmic level. By implementing methods to drastically improve the runtime efficiency of foundational operations like matrix multiplication, this (and other technologies or improved efficiency) can contribute to a potential reduction in computational resource requirements. The more efficient algorithm as demonstrated in this report, achieves the same result in less time. This directly translates into lower energy consumption per computation which could reduce the operational costs and environmental impact of the large data servers running the code.

9 Conclusion

This project has successfully demonstrated that manual low-level optimisation using LLVM IR can yield significant performance gains over compiler-optimised native C code for complex computational algorithms. The implementation of Strassen's algorithm in my hand-written LLVM IR achieved up to a 5.23x reduction in runtime compared to the standard O(n^3) operations for 1024x1024 matrices, while maintaining its numerical correctness across all test cases as seen in Figure 9.1 below.

These results confirm that hardware-optimised software strategies applied to the fundamental processing operations (such as matrix multiplication), can significantly improve computation and runtime efficiency. This approach provides both immediate performance benefits as well as contributing towards a more sustainable computing practice by reducing overall energy consumption due to decreased processing requirements. A theoretical 5% improvement in runtime efficiency for a large datacenter can lead to a lower hardware requirement and footprint, as more computational power can be utilized by the same volume of resources.



```
sh-5.2$ build/strassen_driver -n 1024 -iterations 5
=== Strassen Matrix Multiplication Test ===
Matrix size: 1024 x 1024
Memory usage: 8.00 MB per matrix

Initializing matrices with random values...

=== Correctness Testing ===
Computing reference result (standard algorithm)...
Computing Strassen result...
  Max difference: 5.53e-11, Errors: 0/1048576

Results:
Strassen vs Reference: PASS

=== Performance Benchmark ===
Running 5 iterations each...

Performance Results:
Reference C:    3113.807 ms  (  0.69 GFLOPS)
Strassen LLVM:   680.415 ms  (  3.16 GFLOPS)  [4.58x vs Reference]

Speedup Analysis:
Strassen vs Reference: 4.58x
Theoretical Strassen advantage: 3.80x (for large n)
```

Figure 9.1– 10x24x1024 matrix result

10 References

[1] H. Kinsley, "Reinforcement Learning," PythonProgramming, [Online]. Available: https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/. [Accessed 02 02 2021].

[2] [Online].

[3] MakeSigns, "Scientic Posters Tutorial," [Online]. Available: https://www.makesigns.com/tutorials/scientific-poster-parts.aspx. [Accessed 09 02 2021].

[4] Arduino. [Online]. Available: https://www.arduino.cc/. [Accessed 09 02 2021].

[1] GeeksforGeeks, "Introduction to Compilers," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/compiler-design/introduction-to-compilers/. [Accessed: Aug. 11, 2025].

[2] F. Boisvert, "LLVM in 100 Seconds," Fireship, YouTube, Nov. 23, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=BT2Cv-Tjq7Q. [Accessed: Aug. 09, 2025].

[3] LLVM Project, "LLVM Language Reference Manual," LLVM.org. [Online]. Available: https://llvm.org/docs/LangRef.html. [Accessed: Apr. 12, 2025].

[4] C. Lattner and V. Adve, "An Introduction to LLVM," in *Gelato ICE Meeting*, 2006. [Online]. Available: https://llvm.org/pubs/2006-04-25-GelatoLLVMIntro.pdf. [Accessed: Aug. 11, 2025].

[5] Baeldung, "Matrix Multiplication Algorithms," Baeldung on Computer Science, May 11, 2023. [Online]. Available: https://www.baeldung.com/cs/matrix-multiplication-algorithms. [Accessed: Aug. 13, 2025].

[6] A. Klappenecker, "Divide and Conquer: Strassen's Algorithm," Texas A&M University, 2019. [Online]. Available: https://people.engr.tamu.edu/andreas-klappenecker/csce411-s19/csce411-divideconquer2.pdf. [Accessed: Aug. 13, 2025].

[7] D. Chisnall, "The Definition of an Insanity: Designing an IR for a Compiler," in *2017 European LLVM Developers' Meeting*, 2017. [Online]. Available: https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf. [Accessed: Aug. 15, 2025].

[8] Free Software Foundation, "Aligned Memory Blocks," in *The GNU C Library Reference Manual*. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html. [Accessed: Aug. 13, 2025].

[9] LLVM Project, "DataLayout Class Reference," LLVM Doxygen Documentation. [Online]. Available: https://www.few.vu.nl/~lsc300/LLVM/doxygen/classllvm_1_1DataLayout.html. [Accessed: Aug. 18, 2025].

[10] Intel Corporation, "Introduction to Intel Advanced Vector Extensions," Lawrence Livermore National Laboratory. [Online]. Available: https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf. [Accessed: Aug. 20, 2025].