Ira A. Fulton Schools of Engineering
School of Computing, Informatics and Decision Systems Engineering

SER 450 **COMPUTER ARCHITECTURE**

Your Name: **Jacob Hreshchyshyn** Jacob Hreshchyshyn

PROJECT 2 WORKSHEET

Step I: Preparation

Fill out the following table for the computer on which your program runs were made:

Parameter	**Value**
Processor Type (e.g., Intel Core i5)	AMD Ryzen 7 5800X 8-Core Processor
Processor Frequency (e.g., 2.5 GHz)	3.80 GHz
Operating System (e.g., Ubuntu Linux)	Windows 10
C++ Compiler (e.g., Visual Studio)	G++ Compiler with GCC Version 6.3.0
Java Runtime Environment (e.g. Java 15 SE)	Java 17 SE

Step II: Assembly File Output Analysis

For this portion of the assignment, please view assembly_optimized.txt and assembly_unoptimized.txt. These files are portions of the assembly-language outputs from the C++ compiler for the optimized (release) and non-optimized (debug) versions of our code.

A.

Interpret any line that begins with a semicolon character (';') as a comment line. For instance:

```
; 46   :  double zx_next = zx * zx - zy * zy + xpos;
```

This comment lets us know that the following assembly code corresponds to the C code instruction that appears on line 46 of the input file.

Interpret any line that begins with a dollar sign ('$') as a label, indicating where instructions from other places in the program may branch or jump to. For instance:

```
$LN2@calculateP:
```

Is a label that indicates a specific branch destination in the calculatePoint function.

All other lines should be assumed to be instructions. Count the total number of assembly instructions in the calculatePoint loop (assembly file lines 16-55 for optimized, lines 30-86 for unoptimized). Also compute the number of lines that access memory. These can be identified by push or pop instructions or lines that include the keyword ‘PTR’ in the operand. Enter your results in the following table:

	Description	**Optimized**	**Unoptimized**
1	Total instructions per loop	21	30
2	Memory operations per loop	1	24
3	Non-memory operations per loop (row 1 – row 2)	20	6

B.

If memory operations have a typical CPI of 4 clocks and non-memory operations have a CPI of 1 clock, find the Average CPI for each optimization, and the expected speedup for the Optimized code as compared to the Unoptimized code? Please show your work.

Average CPI for Optimized:
We already determined that the total number of instructions per loop is 21.

We can find the number of clock cycles by multiplying the instruction count of each instruction class (memory and non-memory) with their corresponding CPIs and sum each product.

Clock Cycles Optimized = (20 * 1) + (1 * 4) = 24

Finally, we divide the number of clock cycles by the total number of instructions to find the average CPI.

Avg. CPI = 24/21 = 1.14

Average CPI for Unoptimized:
We already determined that the total number of instructions per loop is 30.

We can find the number of clock cycles by multiplying the instruction count of each instruction class (memory and non-memory) with their corresponding CPIs and sum each product.

Clock Cycles Unoptimized = (6 * 1) + (24 * 4) = 102

Finally, we divide the number of clock cycles by the total number of instructions to find the average CPI.

Avg. CPI = 102/30 = 3.4

Speedup of Optimized versus Unoptimized:

To find the speedup of the optimized code versus the unoptimized code, we can use the performance model CPU Time = Instruction Count * CPI * Clock Cycle Time. Since we are using the same processor, the Clock Cycle Time should be the same when we compare the CPU Times of each program, so they will cancel out.

CPU Time Unoptimized/CPU Time Optimized = (30 * 3.4 * Clock Cycle Time)/(21 * 1.14 * Clock Cycle Time)
= 102/24
= 4.25

In other words, we see a 4.25 time speedup of the optimized code versus the unoptimized code.

Step III: Benchmarks

A.

Complete the following table of execution times from your four program runs. To disable JIT in IntelliJ, go to Gear (IDE and Project Settings) -> Settings -> Type JIT in search field, and Disable JIT in Debugger.

To remove optimizations for the G++ compiler, I used -O0, e.g. g++ -O0 mandelbrot.cpp. To include the highest optimization, I used the flag -O3.

	Java, no JIT	**Java, with JIT**	**C++ Unoptimized**	**C++ Optimized**
Execution Time (Seconds)	153.2828797	7.307398501	36.8273	17.307

B.

From the information in step III-A, calculate the following speedups with compared to C++ unoptimized.

	Speedup, Java, no JIT	**Speedup, Java, with JIT**	**Speedup, C++ Optimized**
Speedup	36.8273/153.2828797= 0.240257099	36.8273/7.307398501 = 5.039727886	36.8273/17.307 = 2.127884671

C.

Compare your measured speedup of the C++ optimized code in Step III-B to the predicted speedup from Step II-B. If the two are different, what might account for the differences?

To recap, the predicted speedup of using optimized C++ code was 4.25. Instead, according to our benchmarking, we got a speedup of around 2.12. There are a few potential explanations for this discrepancy. One of these explanations is how I set up the G++ compiler to give me unoptimized C++ code. When compiling, I used the -O0 flag, which is essentially the same as compiling the code without any flags and provides fewer optimizations than the -O3 flag. However, according to this site, there still exist optimizations with the -O0 flag. More importantly, however, one can optimize the code with different goals in mind, whether it be for code size, execution time, including accurate math calculations, etc. In short, I don't know in what way the provided assembly code was optimized. If I did, I can perform experiments with those exact optimization parameters so that more similar results might be obtained.

What I find especially strange, however, is the performance of Java with JIT enabled versus the optimized C++ code. It outperforms our optimized C++ code. This could have something to do with the fact that this version of Java, Java 17 SE, is the latest version, which could mean boosts in execution time. Additionally, the time keeping libraries themselves could have affected the execution time of the program, though this is unlikely since they do not directly interfere with the calculations of the image itself. It could be that my AMD processor can better execute my Java program than my C++ optimized program. It could be that the Mendelbrot set algorithm itself lends itself performance-wise to Java rather than C++, which is possible as indicated in Professor Sandy's chart on Bubble Sort performance metrics. In short, there are so many variables physically at play that it appears that our experiments successfully demonstrated that our performance model can only take us so far in predicting performance improvements.

Step IV: Business Impact

A.

Suppose FractalFutures' current product is written in Java and runs primarily on platforms where the JIT compiler is not available or enabled. They would like to update their product to get higher performance, but portability that comes from a Java source-base is still desirable. Use the following numbers to evaluate the programming language that yields the best returns for FractalFutures' new product. Assume that a new Java platform (if used) would enable JIT compiler features:

$1000 revenue for each percent of performance improvement above the base product

Revenue loss from loss of portability if Java is not used – 10% of current product's revenue (current revenue is $100,000)

Revenue of New Product Based on Java with JIT:
Before providing the revenue, it's important to make the assumption that we can estimate the performance of the current product using the performance metrics we obtained in Step III.

With this in mind, we can set the execution time of our Mendelbrot set program as FractalFutures's current product performance. This is

153.2828797 seconds. The performance of the same program with JIT enabled is 7.307398501. We can find the percentage of improvement by doing the following:
((153.2828797 - 7.307398501)/| 153.2828797 |) * 100 = 95.23273668%, which we can round to 95%. Thus, we see that we get a 95% performance improvement by enabling JIT. Since we get an additional $1,000 for each percent of improvement above the base product, we have the following new revenue:
$100,000 + $1,000 * 95 = $195,000, nearly doubling the existing revenue.

Revenue of New Product Based on optimized C++:

Using the same assumptions, the current execution time of our Mendelbrot program is 153.2828797 seconds. The performance of the same program run on optimized C++ is 17.307 seconds. We can find the percentage of improvement by doing the following:
((153.2828797 - 17.307)/| 153.2828797 |) * 100 = 88.70911087%, which we can round to 89%. Thus, we see that we get an 89% performance improvement by using the optimized C++. However, we also must consider the revenue loss from loss of portability, which is 10% of the current product's revenue. With this in mind, we have the following new revenue:
$100,000 + $1,000 * 89 – ($100,000 * 0.1) = $179,000.

Your Recommendation (with justification):

Based on the numbers, the clear recommendation is for FractalFutures to shift their product to Java with JIT enabled. This would allow them to maintain the portability of the current product while also drastically improving the performance of their product, even beyond what the optimized C++ can do. Also, $195,000 revenue is more than $179,000. No contest.

B.

A common mistake of C++ programmers is to forget to optimize their code for release. Suppose the developers used unoptimized C++ code for the FractalFutures new product.

Revenue of New Product Based on unoptimized C++:

Using the same assumptions, the current execution time of our Mendelbrot program is 153.2828797 seconds. The performance of the same program run on unoptimized C++ is 36.8273 seconds. We can find the percentage of improvement by doing the following:
((153.2828797 - 36.8273)/| 153.2828797 |) * 100 = 75.9743%, which we can round to 76%. Thus, we see that we get a 76% performance improvement by using the unoptimized C++. However, we also must consider the revenue loss from loss of portability, which is 10% of the current product's revenue. With this in mind, we have the following new revenue:
$100,000 + $1,000 * 76 – ($100,000 * 0.1) = $166,000.

What is the cost of this mistake in terms of lost revenue?

Considering how high the revenue could be, which is $195,000, this mistake would result in a $195,000 - $166,000 = $29,000 loss in revenue.