System Languages Comparison

EECE5640 Final Project Report

Micah Weston

April 16, 2022

Contents

1 Introduction

To fully exploit computing resources, the programmer needs to be close to the hardware and fully understand the abstractions between code and the metal. Systems languages are created to allow software developers low level access without losing the capabilities of high level abstractions These languages try to balance ease of use, portability, and expressiveness while still allowing performance and control.

Early systems languages were often created by companies for their specific hardware. *C* was the first to gain wide popularity, eventually becoming the basis of much modern low-level software. About a decade later, *C++* would spawn as an extension to *C* which added new paradigms and higher level abstractions. *C++* would eventually diverge as a language but it has maintained its compatibility with *C* in certain ways to continue using shared infrastructure.

More recently, newer system languages have been created to improve on what people have seen as mistakes in the older languages. *Rust* is a popular new comer which adds safety in its unique memory model which is validated by the compiler. *Go* also is a popular new systems language that tries to keep code system and provide tools for green threading and concurrency. It has diverged partially from the norm by adding a garbage collection system.

This project compared three of the popular systems languages—*C*, *C++*, and *Go*—in not just performance but also easy of development, code readability, language and library features, and parallel programming support. Two benchmarks were chosen for analysis: Matrix Multiplication and Fast Fourier Transform. Matrix Multiplication provided a classic parallel program to test the threading capabilities of each language. FFT used complex arithmetic which allowed for the demonstration of unique language features.

In addition to the algorithms, input data and output results were handled in a simple text base file format to test the IO capabilities of the language. In order to test language support for the different features, an idiomatic approach to the software development was taken. This meant avoiding systems calls and other low-level optimizations which would break portability but allow speed ups. The code in each language was written to be able to run on an POSIX environment.

2 Specification

Three languages were used in the project: *C*, *C++*, and *Go*. C11[1] was used for the *C* standard and Pthreads were chosen as the parallel support since the threading standard has not been implemented by the library maintainers. C++20[1] was used for the *C++* standard with the features implemented by GCC and Clang. Go 1.17.4[2] was used for the *Go* version since it was the most modern available on the Discovery Cluster.

For build infrastructure, *CMake* was used for both *C* and *C++* while *Go* used the Go Tool and Modules.

Two benchmarks—Matrix Multiplication and FFT—were implemented in each language. The algorithms were developed with similar interface behavior between the languages. Matrix Multiplication accepts two input matrices of doubles and returns a new matrix as the result. FFT takes in a single vector of complex numbers and applies the result in place.

Matrix Multiplication provided a classic example of a parallel program with many known optimization techniques. Each language used similar optimizations but utilized the language idioms when implementing these.

FFT provided utilization of language and library features through the usage of complex numbers. Each language has support for complex arithmetic: *Go* has it in the language, *C++* has it in the library, and *C* mixes it into the language and preprocessor. To keep the algorithm manageable, only inputs with length of a power of two were permitted.

The file format for matrix multiplication uses ASCII text to encode a matrix of 64-bit floating point values. The first line of the file includes the number of rows and the number of columns separated by a space. The rest of the file has N lines with M numbers each, where N is the number of rows and M is the number of columns. Each number has a single space following it, including the last number in the row. The last row has a new line at the end.

The file format for FFT uses ASCII text as well to encode a vector of 128-bit complex numbers, two 64-bit floats representing the real and imaginary parts. The first line includes the number of elements. The following lines each contain a single complex number. The complex number is wrapped in parenthesis. There are no spaces in the number and the imaginary portion has a sign at the front and an i following it. Each number is followed by a new line.

(Note: The file format used differs from the project proposal since it was quickly realized that CSV was a poor format to encode data due two no uniform column widths.)

Each language followed the file format and printed numbers with at least 6 precision in the floating point value. During matrix multiplication, the matrices could be read in separate threads to allow concurrent IO.

The executables used in benchmarking were generated by building each program with the release setting. Since *Go* does not differentiate builds, this meant the default compilation. For *C* and *C++*, optimizations were set to the maximum "-O3" and debug information was not generated "-g0".

For the input data, a simple program written in Go was used to generate random matrix and vector files. The values were generated from a seeded random number generator producing Floating point values normalized to the range of $[-50, 50]$.

Four inputs were tested for each program. The matrix dimensions were given by three values so that the inner dimensions matched: $(m, k) \times (k, n)$. The vector length was chose as powers of 2. The dimensions used in the benchmark are given in the following table.

	Matrix Dimensions		
Test	M	K	N
1	512	512	512
2	2,000	2,000	2,000
3	4,096	4,096	4,096
4	800	900	700

	Vector Length	
Test	N	2^N
1	16	65,536
2	20	1,048,576
3	25	33,554,432
4	28	268,435,456

Table 1: Dimensions used for Matrix Multiplication and FFT during benchmark tests.

The tests were in a batch script that automated the building of executables, generation of input data, running of tests, and accumulation of results. This script works with the Discovery Cluster's SLURM system. It first builds all the executables from scratch. Next the input data is generated and placed into separate directories. These first two portions are performed with parallelism for most of the processes. Next, each benchmark is repeated twenty times for each test input with results written to a log file. These benchmarks are ran one at a time so that the full resources of the node are dedicated to the single process. Finally, all the results are averaged out and written to the results file.

3 Implementations

Each language had similarities in the project organization, API behavior, and speed optimization. Differences typically manifested through the idioms of the language.

Projects were organized in a modular manner, grouping code by the behavior. For example, all matrix operations were implemented in their own directory. The software tried to follow the DRY principle with only minor functions in the main files, such as timing helpers, to be repeated.

For the APIs, the file IO followed the patterns of the language. For input files, each implementation returned the resulting data structure or errored in some way. For output files, each implementation took a data structure and wrote to the provided output stream until complete or an error occurs.

For the optimization of matrix multiplication, the same techniques were used in each implementation. First the work was distributed across multiple threads. The work distributions algorithm mimicked the observed behavior of analyzing an OpenMP implementation of matrix multiplication. Next the loops were blocked as to provide better utilization of cache memory. Finally, the inner column based looped was unrolled, having multiply cells computed at the same time.

3.1 Go

As a part of the language's push for simplicity, coding style is handed down by the *Go* standard. The *Go* Tool comes with auto formatters and many expressions in the language have strict rules for spacing and format. This made it easy for picking a coding style since it is provided by the language itself.

The project structure partially mimics the *Go* compiler structure in the way that directories and binaries are laid out. The main files for each program had its own directory was put in the *tools* directory. The unique directories are required by the language. The other code, such as matrix operations, fft operations, and utilities each had their own directory.

Parallel code was done through Goroutines. These green threads follow the example of higher level languages with the use of channels and message passing. In order to synchronize, waitgroups have to be used since there is no way to directly wait on the return of the Goroutine.

For IO, *Go* strongly follows their interface approach to objects. This always for abstractions of shared behavior and easy composition of objects. For importing and exporting data, the functions took in a Reader and Writer interface respectively. The caller has the responsibility of providing the IO object but they are not limited to just files. Additionally, the buffered IO utilities make it easy to ensure that you can efficiently cache data for read and write operations. These techniques are similar to those seen Java and other OOD languages that provide mechanisms for composition.

For the FFT operations, the built-in complex numbers could be used. These values could be manipulated using the usual mathematical operators while more advanced functions, such as sin and cos, were available in the library.

3.2 C++

For the project, the LLVM style was followed in code formatting and idioms. The recent creation of formatter like ClangFormat allows for much easier development of *C* family languages removing the responsibility of these tiny details from the coder. Additionally, larger projects can now automate the enforcement of code styles rather than relying of reviewers being diligent.

The project structure mimics the structure of the LLVM project as well. Executable main files each of a subdirectory in the *tools* directory. The API declarations in the public header files were placed in the *include* directory while the implementation files existed in the *lib* directory.

Build infrastructure was done through the use of *CMake*. Unlike newer languages, *C++* and *C* do not have structured requirements to their build organization due to the usage translation unit model that the standard uses. Meta-build tools like *CMake* have become a common use for large projects since directly managing the build system for tens and hundreds of files can become excessive and prone to errors.

The threading for *C++* became standardized in C++11. On POSIX systems this requires that the project is linked with Pthreads since the standard library implements threads that way under the hood. The standard library features allow for easy create and management of threads. The construction of threads is more flexible through the usage of metaprogramming to allow the invocation of an callable object, not just a single type of function.

The IO of *C++* is done through streams that can provide flexibility through the standard APIs. Similar to *Go*, the import and export of data takes in any stream, allowing the caller to use not just files. The standard library does not include as much utilities for composition so improvements like buffering require a more direct approach.

The complex arithmetic used in FFT is implemented all in the library rather than being built into the language. C++ allows for the creation of such types through templated classes and operator overloading. This allows for expressive code and unique features without modification to the base language.

3.3 C

For the *C* code in the project, the Google style was followed and enforced using ClangFormat. The LLVM style was not used since it is not suited for *C* idioms and naming conventions.

The project structure mimicked the *C++* code for convenience. Since *C++* shares a lot of the build infrastructure that it inherited from *C*, the organization of the two languages could closely follow one another. Again, CMake was used for management of building the software.

Threading was added to the *C* standard in C11 as an optional feature. Unfortunately, it is not widely implemented by library maintainers rendering mostly unusable. As such, Pthreads were chosen as the thread library to use since the experiments would be ran on POSIX systems. These low level APIs require the developer to create thread entry point functions that conform to the function pointer type needed by the library. This is different from *C++* and *Go* where the any callable object is acceptable to the thread API.

The IO of *C* was done through file pointers. This does not provide the same flexibility as the other languages, requiring that all IO take place through just files. It does match the other languages in being portable since system calls are not directly exposed to the developer.

Complex arithmetic support was added in C99. It follows a similar pattern seen with C features where it utilizes the preprocessor heavily for the API. Simple arithmetic could be performed through operators built into the language while more advanced operations were library functions.

Due to the nature of C, two APIs were developed for all matrix and FFT operations. One API took raw inputs where an array and dimensions are specified for each component of the computation. This is similar to BLAS libraries that do not impose custom data types. A second API took the custom object for convenience and just dispatches the computation to the raw API. This convenience would allow a developer an easier time since the data of a matrix or vector is encapsulated to one object.

4 Benchmark Results

Each experiment was tested on a single node on the Discovery Cluster. The CPU was an Intel Xeon E5-2680 v4 running at 2.40GHz. All 28 cores were allocated to the job when running the experiments. Each program was repeated 20 times to get an average of the execution times.

4.1 Matrix Multiplication

Total Runtime: The results for the total runtime of matrix multiplication can be seen in Table 2 and Figure 1. This includes importing the two matrices, calculating the matrix multiplication, then exporting the result.

MatMul (seconds)	Go	C++	C
(512x512)(512x512)	0.348	0.435	0.404
(800x900)(900x700)	0.790	0.897	0.860
(2000x2000)(2000x2000)	5.306	4.440	4.429
(4096x4069)(4096x4096)	42.193	20.923	20.215

Table 2: Table of language performance for the total runtime of matrix multiplication.

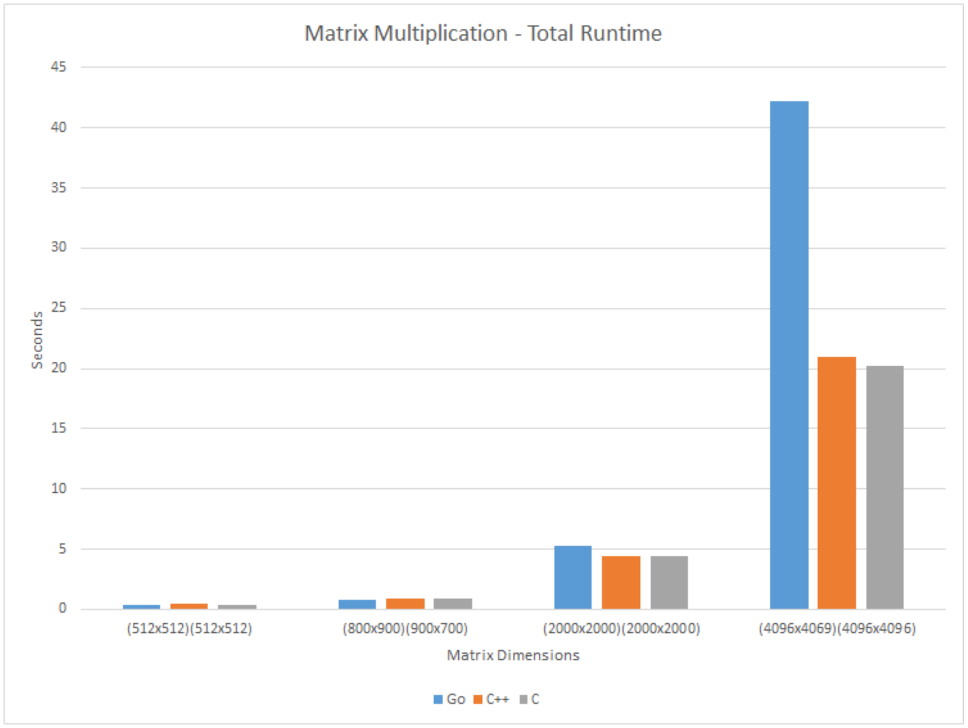

Figure 1: Plot of language performance for the total runtime of matrix multiplication.

Import Runtime: The results for time to import the matrices can be seen in Table 3 and Figure 2.

MatMul (seconds)	Go	C++	C
(512x512)(512x512)	0.108	0.137	0.120
(800x900)(900x700)	0.227	0.324	0.269
(2000x2000)(2000x2000)	0.971	1.434	1.262
(4096x4069)(4096x4096)	3.314	5.167	4.407

Table 3: Table of language performance for the importing the matrices.

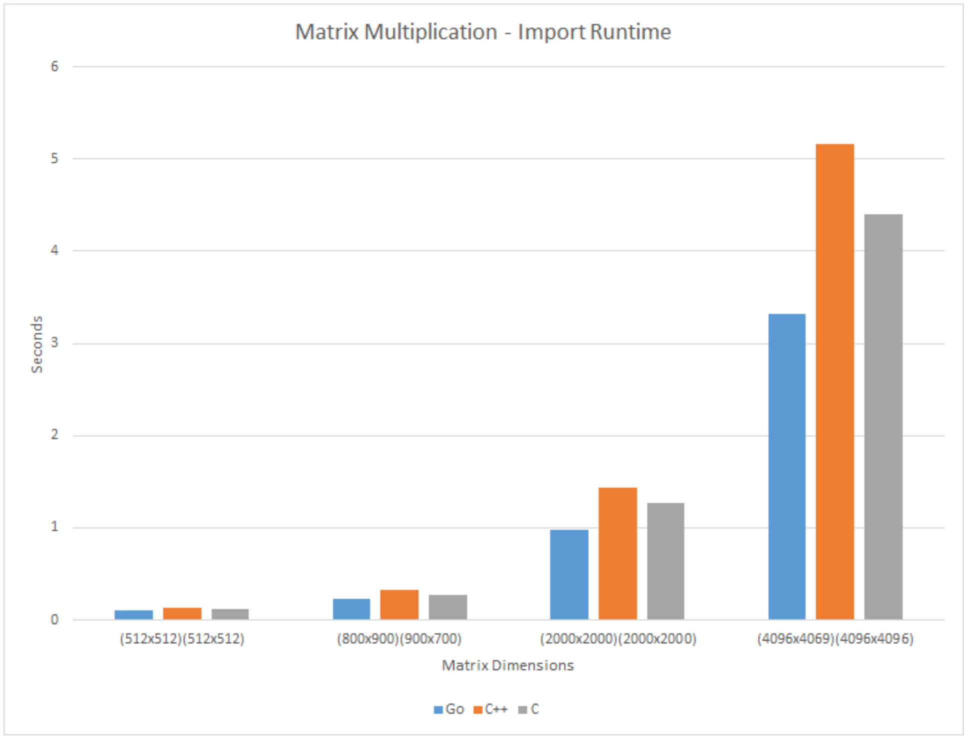

Figure 2: Plot of language performance for the importing the matrices.

Computation Runtime: The results for time to compute the matrix multiplication can be seen in Table 4 and Figure 3.

MatMul (seconds)	Go	C++	C
(512x512)(512x512)	0.102	0.015	0.011
(800x900)(900x700)	0.274	0.023	0.020
(2000x2000)(2000x2000)	2.718	0.224	0.199
(4096x4069)(4096x4096)	32.218	4.164	3.796

Table 4: Table of language performance for the computation of the matrix multiplication.

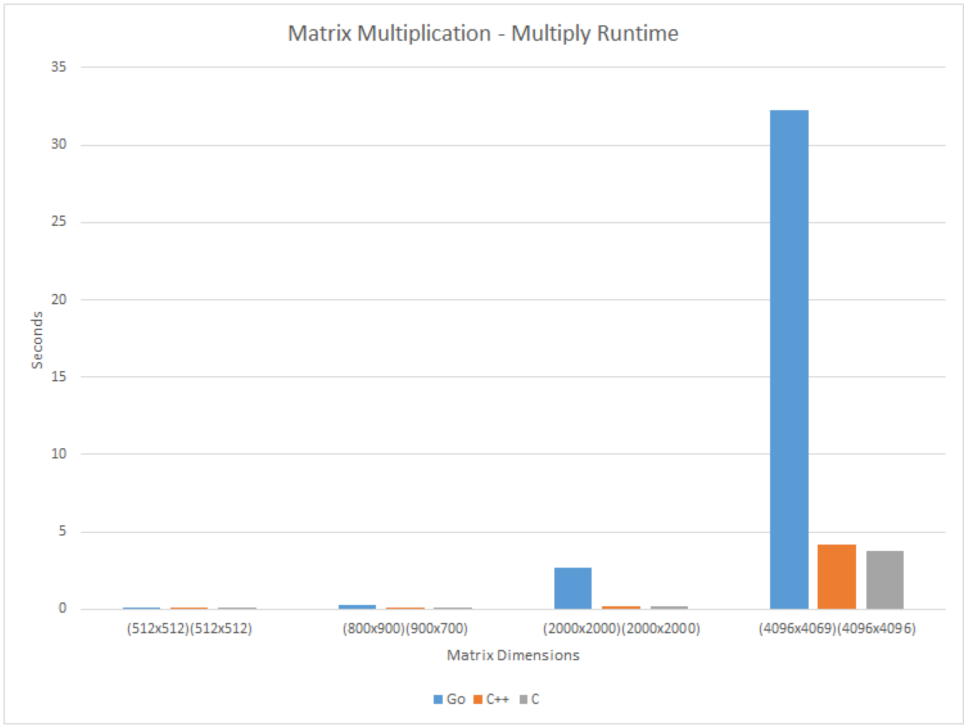

Figure 3: Plot of language performance for the computation of the matrix multiplication.

Export Runtime: The results for time to export the result can be seen in Table 5 and Figure 4.

MatMul (seconds)	Go	C++	C
(512x512)(512x512)	0.137	0.282	0.272
(800x900)(900x700)	0.289	0.550	0.571
(2000x2000)(2000x2000)	1.617	2.782	2.968
(4096x4069)(4096x4096)	6.661	11.591	12.012

Table 5: Table of language performance for the exporting the result.

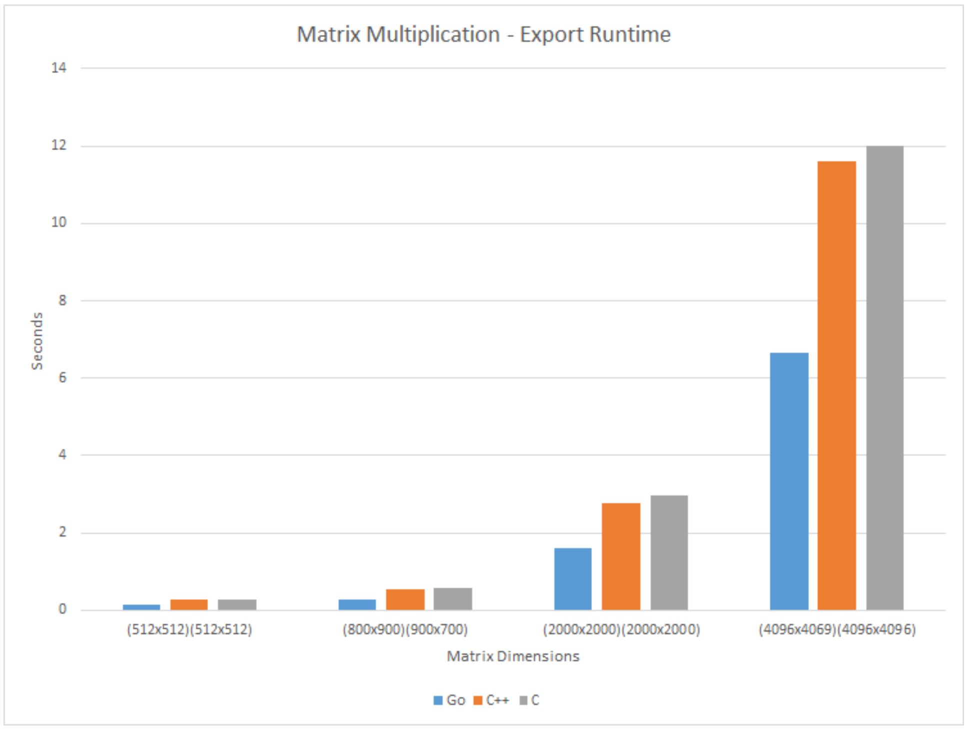

Figure 4: Plot of language performance for the exporting the result.

4.2 Fast Fourier Transform

Total Runtime: The results for the total runtime of fast fourier transform can be seen in Table 6 and Figure 5. This includes importing the vector, calculating the fft, then exporting the result.

FFT (seconds)	Go	C++	C
2^{16}	0.092	0.220	0.197
2^{20}	1.547	2.511	2.240
2^{25}	44.068	88.614	82.918
2^{28}	376.190	746.302	727.333

Table 6: Table of language performance for the total runtime of fft.

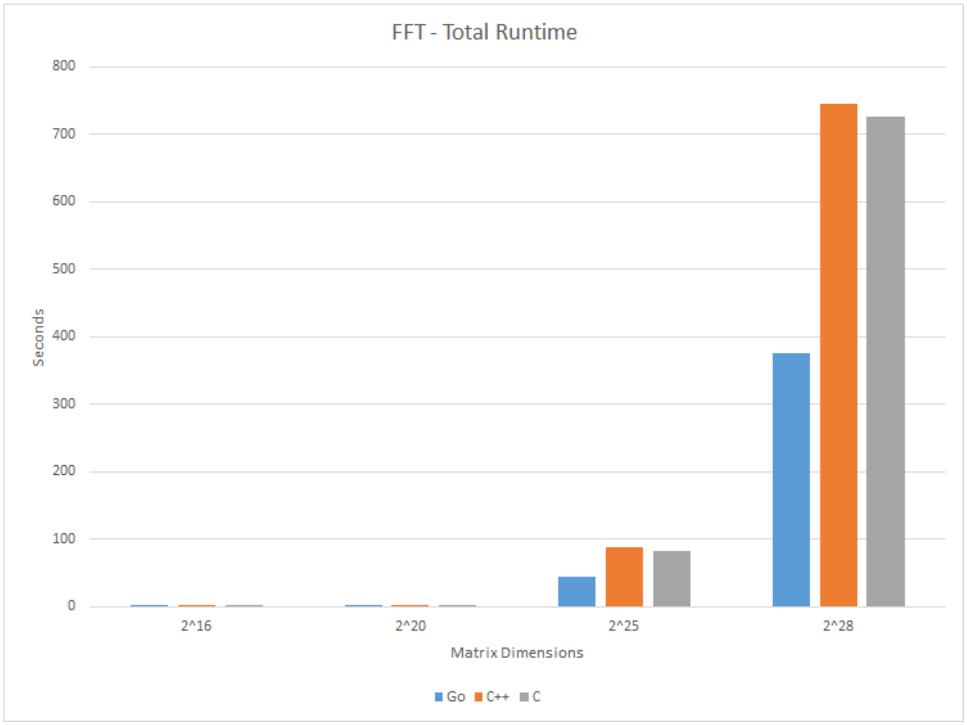

Figure 5: Plot of language performance for the total runtime of fft.

Import Runtime: The results for time to import the vector can be seen in Table 7 and Figure 6.

FFT (seconds)	Go	C++	C
2^{16}	0.018	0.077	0.062
2^{20}	0.335	0.853	0.650
2^{25}	7.243	20.293	15.287
2^{28}	54.652	160.327	122.005

Table 7: Table of language performance for the importing the vector.

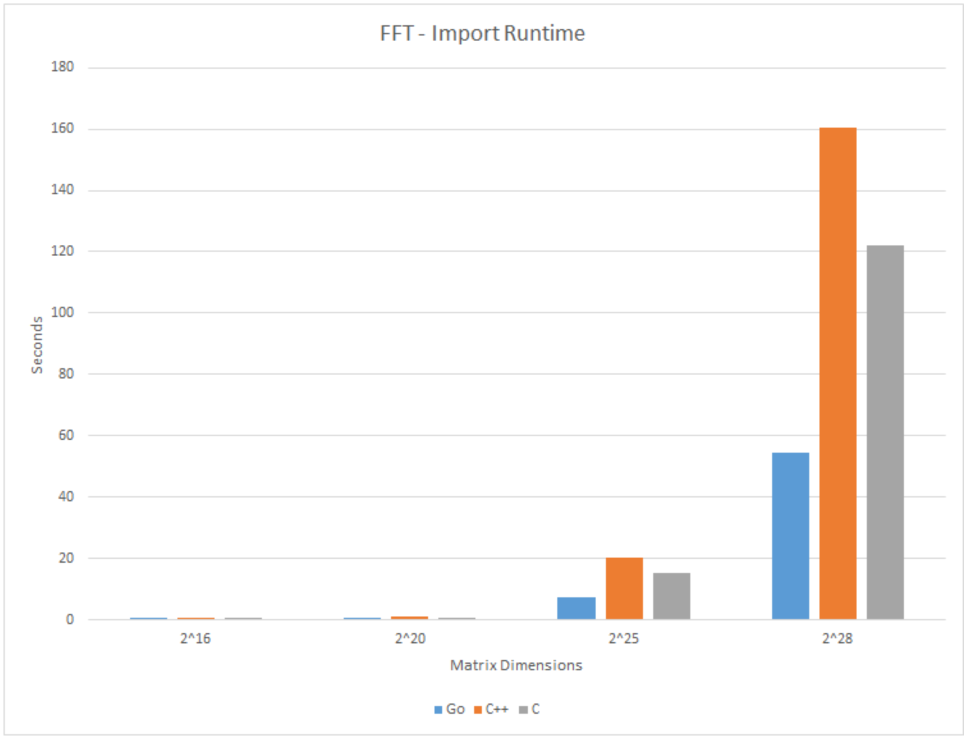

Figure 6: Plot of language performance for the importing the vector.

Computation Runtime: The results for time to compute the fft can be seen in Table 8 and Figure 7.

FFT (seconds)	Go	C++	C
2^{16}	0.005	0.007	0.007
2^{20}	0.117	0.099	0.101
2^{25}	7.386	11.091	11.024
2^{28}	73.994	108.629	107.635

Table 8: Table of language performance for the computation of the fft.

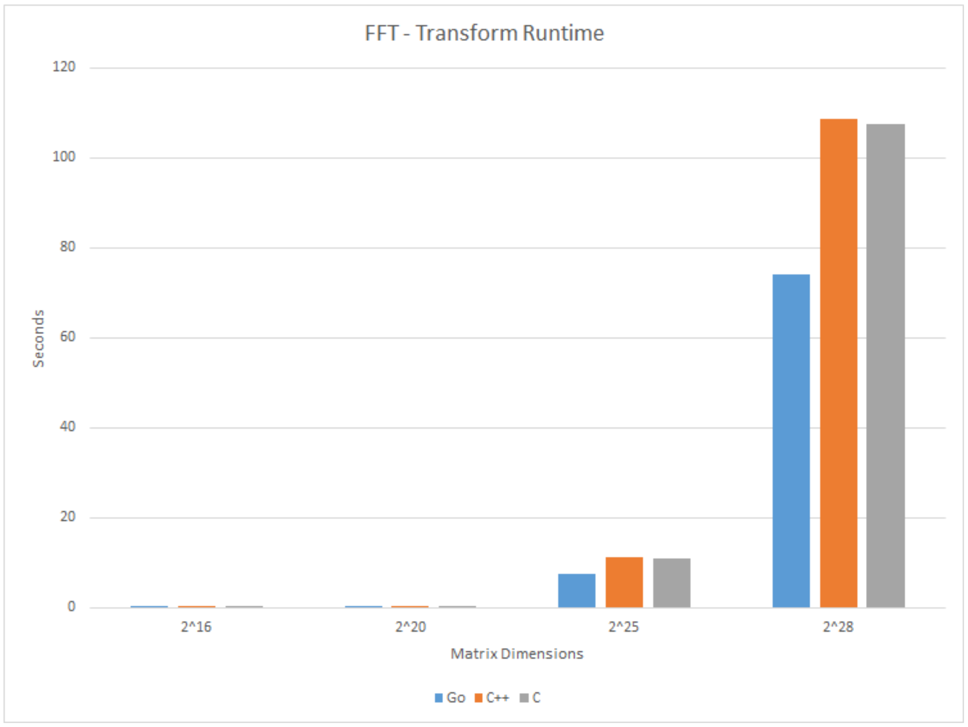

Figure 7: Plot of language performance for the computation of the fft.

Export Runtime: The results for time to export the result can be seen in Table 9 and Figure 8.

FFT (seconds)	Go	C++	C
2^{16}	0.064	0.129	0.121
2^{20}	0.976	1.460	1.388
2^{25}	22.054	46.139	45.583
2^{28}	173.549	368.718	390.058

Table 9: Table of language performance for the exporting the result.

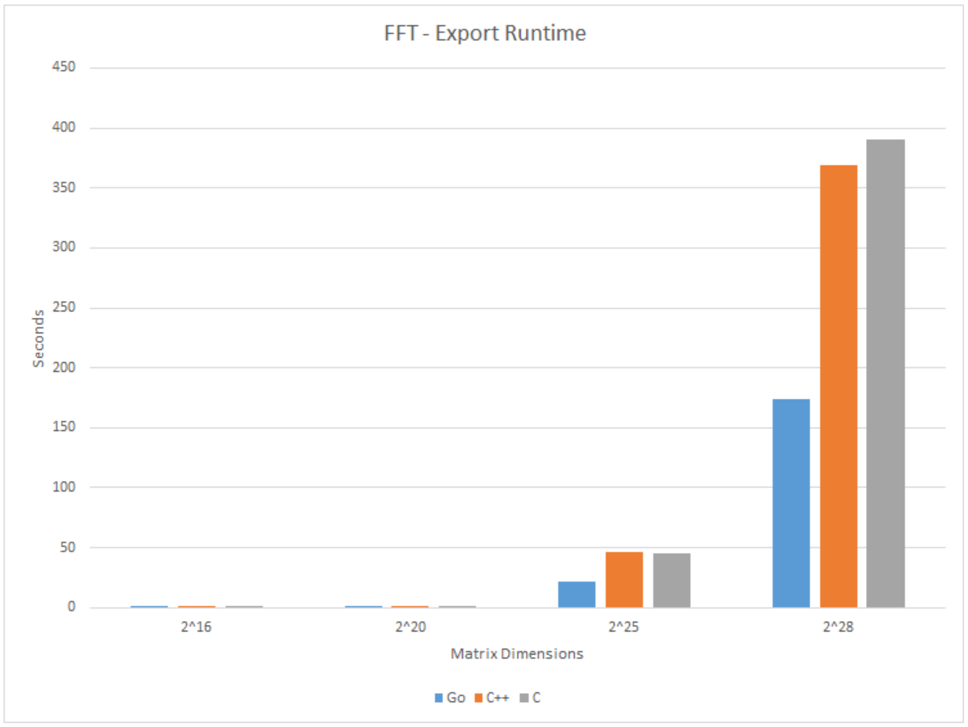

Figure 8: Plot of language performance for the exporting the result.

5 Analysis

This project seeks to analyze not just the overall performance of the code but also the ease of development, language and library features, as well as expressiveness of code.

The performance is an object measure of the benchmark results. This performance might not show the full potential of the each language due to limitations set by following a strictly idiomatic approach and the short coming of the developers skills.

Ease of development, language and library features, and expressiveness of code are more of subjective measures. There is no one write answer and people have different preferences that might suit one language over the others. This project tries to highly pros and cons of each language.

5.1 Performance Results

The measured performance results might show the full potential of each language. For each program, standard library features were preferred when possible which might not allows be the fastest approach. Often times, these libraries aim for the maximum portability which might have limitations.

5.1.1 Input and Output

By far the winner for IO was *Go*. Its library supports IO utilities the let you compose the interface with a buffering system. Just through the usage of a *bufio* Reader/Writer, the IO saw a massive improvement during development. This continued into the final result where Go consistently had faster IO.

The troubles of *C++* and *C* had some similarities and differences. Both languages like easy methods to add buffering in the way that *Go* provides. *C++* has an ability to specify a buffer to IO streams but it is unclear whether this actually had a benefit.

C++ would narrowly beat *C* when it came to output. This is likely due to the nature of the *C++* compiler exposing more information to the compiler allowing for more optimizations. The *C* IO function have to process raw strings for formats when a call is made to *fprintf* and other functions.

Additionally, the IOStreams of *C++* are generally disliked by much of the community. For example, the Google style guide[3] discourages its use and the LLVM Coding Standards[4] outright forbids the usage. Open source libraries exist to make up for its short falls. A common library is fmtlib[5] which is being used by Microsoft, Facebook, and many more.

5.1.2 Computation

The results differed between Matrix Multiplication and FFT. For the Matrix Multiplication, *C* and *C++* far exceeded the performance of *Go*. For FFT, it was the opposite.

Generally *C* was a little bit faster than *C++* in the computations, but they both had very similar performance. This is somewhat expected as much of the abstraction of *C++* can be stripped away by the optimizing compiler and the end result is close to the same operation as *C*. Additionally, complex numbers in *C++* use the underlying implementation of the *C* complex feature.

The matrix multiplication saw the *C* languages beat *Go* by a factor of ten in speed. There are likely a few contributing factors.

One likely cause is the lack of optimization in the *Go* compiler compared to the aggressive *C* compilers. Another area the would hinder performance is safety guarantees of slices in *Go*. All indexing operations must be check to ensure they fall within the bounds. These checks will put stress on the branch prediction units and extend the loop bodies. Lastly the cost of the garbage collector might be hindering the system.

However for the FFT *Go* ended up being the winner. The difference was less pronounced than the multiplication performance with *Go* winning by about 1.5 times faster. The difference also only appears at large input sizes since the lower sizes were much closer with *C++* being faster for one of the cases.

At the large sizes, the slow down for the *C* languages appears to be the result of GCC. Some testing with Clang saw the *C* and *C++* implementations close the gap on *Go* but still a bit slower. More testing showed that the maximum optimizations ”-O3” were slower than ”-O2” for the FFT computation.

From analysis of the assembly output of the computation functions produced by the different compilers, there seems to be two reasons why GCC had massive slowdowns. One reason is the inability to inline complex multiplication. This forces a call to occur during each loop body. Clang on the other hand is able to inline so that the function is only used as a fall back was a simple approach fails, avoiding unnecessary calls. The other reason seems to stem from too much vectorization of the code. When optimizations are turned down with a lower option or specifically using ”-fno-tree-vectorize”, the GCC implementation sees a speed up.

Additional work can likely be done to improve the FFT algorithm in all implementations. A simpler version was used for this project due to the complex nature of the operation. Additional methods could be used to precompute some of the coefficients that were not taken due to limits in development time.

5.1.3 Overall

The total runtime was similar to that of the computation. This results from the fact that the computational algorithms grow in runtime faster asymptotically than the IO operations.

In the matrix multiplication, *Go* was able to compensate for low computational speed by having the fastest IO. Unfortunately as the sizes grew, the IO had less of an impact on the overall speed. If better IO approaches are used for *C* and *C++*, they would likely far outside the pace of *Go*.

In the FFT, *Go* was fastest on all fronts and outperformed the rest. Some pragmas could be used to help GCC optimize the FFT function. Improved IO for the *C* languages would also help them reach the same level as *Go*. Further investigation would also be needed to determine the root causes of the speed differences.

5.1.4 Executable Size

Another factor that can be influential to language usage is the generated executable size. Certain systems like phones and embedded devices have a limited amount of memory. Table 10 shows that the *C* and *C++* programs are a fraction of the size of the *Go* programs.

Size (KB)	Go	C++	C
MatMul	2057	33	18
FFT	2044	19	17

Table 10: Executable size for each program in the different languages.

This is unsurprising since *Go* must include the runtime in each executable to allow the green thread scheduler and garbage collector to exist. *C++* was a little bigger than *C* for the FFT and a bit under double the size of Matrix Multiplication. This is also expected since *C++* template library features can often generate more code.

An interesting investigation for *C++* and *C* would be to see how Link Time Optimizations can reduce the code size.

5.2 Software Development

A major portion of software is the development and maintenance of code overtime. Very few programs are written once and never changed again. Some code can squeeze out the most performance from a specific architecture but might be garbage on any other machine.

Depending on the developers goals, certain costs might not be worth the rewards. *Python* has gain tons of popular due its image of being quick to develop but it often has some of the slowest speeds out of major languages. Older companies might need stability in a language and end up sticking with old technology. Young companies can keep up the newest technologies since they handle the instability better.

All goals for a project must be considered when choosing a language since performance is only one metric for the varied field of software development.

5.2.1 Go

A driving factor of the *Go* language design is simplicity. I lacks many of the fancier features of *C++*, *Rust*, and other languages that try to support many paradigms.

One of the strongest idioms in *Go* is the usage of multiple returns to also specify an error. Functions can have multiple return values so it is common to have the last value be an error specifier. This feature is supported by the fact that the Go API tends to pass parameters via the stack, unlike the *C* languages often using registers.

The usage of error values leads to many function calls being directly followed by an if-statement to check if things went wrong. This pattern helps to encourage consist error handling in the code. Unfortunately it can quick create repetitive blocks of code. Each if statement will use at least three lines of code due to mandatory scope brackets in the language. There are no ways to reduce the redundancy since the language does not have macros or other features that might help. The extra usage of vertical space can bloat function lengths, making the developer keep track of more information.

Go also has strict rules which elevates many issues from warnings in other languages to errors here. One example is the requirement that every variable be used. This can sometimes slow down development since during earlier stages ideas are quickly changing. It does help to catch bugs which should ultimately speed up the development process.

The standard library is also much larger than the *C* languages. Being developed initially by Google and coming to open-source, *Go* does have the same baggage as the ISO standardized languages. The library includes many features that developers might never encounter, such as web server tools, JSON and CSV parsing, and many other features.

A notable library feature is the buffered IO utilities. These are simple wrappers that can drop into any IO code an instantly make the operation buffered. A single line of code to wrap the input and output interfaces in each function gave a massive speed up to the overall performance. These features allow easy composition and flexibility since they maintain all the same APIs.

The usage of a garbage collector helps take the weight of memory management off the developers shoulders. It guards the risk of leaks that are often see in languages like *C* that put all the responsibility on the engineer. This comes at the cost of code size and runtime performance. The code size is unavoidable since it must always be included in the executable. The performance cause should not be too bad when the code is intelligently designed. Reducing allocations and keeping them uniform can help the garbage collector stay efficient.

The simplicity of the language has its downside in lack of features. Many simple utility functions—minimum, maximum, etc—and higher order function—map, fold, search, etc—were impossible due to the lack of generics. This leads to a lot of redundant functions that do the same thing but for different types. Only recently in 2022 were generics added to the language. It will take time for them to become adopted.

5.2.2 C++

C++ is known for having many features as it tends to support many of the newer language paradigms as they are invented. At the same time, it exposes the low level information that you can get with *C*. This adds greater control and options to the developer but at the cost of complexity. Through a strong system of abstractions, these complexities can be greatly reduced however the developer is responsible to create such abstractions. The language is associated with a high learning curve that can pay off in high performance and advanced systems.

A famous feature of the language is the usage of templates. This is similar to generic code but is completely determined at compile time. The current implementations could quickly be refactored to support other arithmetic types, for example integer matrices through the template feature. It can greatly reduce code and it is the basis of the standard template library. A downside of templates is the weight it puts on compilation, increases to executable size, and complex diagnostics that are generated from errors.

Another notable feature that came in C++17 is copy elision. This feature allows the compiler to avoid copies when returning a large object by having the API pass a pointer to the function instead. It is similar to using an out variable but allows the simpler and readable appearance of a return value. Combined with move semantics, these features allowed the matrix multiplication API and code to be simple without running into memory copy costs.

Another nice feature is scoped initializers and ranged based loops. These allow for strict encapsulation and reduce unnecessary information. They can also help encourage better codegen since they enforce a common pattern in the source code.

The lack of safety requirements like bounds checks to element access can lead to bugs. Luckily strong tooling exists in the form of sanitizers and profiling. These can quickly find small bugs and greatly accelerate the development process.

A byproduct of its origins, *C++* lacks some of the build system features of newer languages. The way things evolved out of *C* has helped *C++* become so popular by being able to share libraries but it has also held it to many of the same decisions of *C*.

The way the compilation module works makes it a bit complex and often requiring both build and metabuild systems. Famous and common build systems include GNU Make and Ninja. Some of the popular metabuild systems include CMake and GNU Autotools. Library management has also developed recently with VKPKG and Conan. Many of these features are builtin with newer languages like *Python*, *Go*, *Rust*, and more. Recent additions of modules to the standard are trying to bridge this gap but they still need to be implemented and widely adopted by users.

5.2.3 C

C is the oldest of the languages used and it has made it so far through stability and compatibility. While sharing similar processes, the evolution of the *C* standard is much slower than *C++* with the younger language influencing its predecessor as of resent. Many features, notably the threading library which would have been helpful to the project, are optional features of the standard that compilers tend not to support. This has allowed for its widespread compatibility and frequent usage in embedded systems and other low level software.

An addition force that pushes the language towards stability is its frequent usage as an interface between systems. Most operating systems expose their API through C and inter-language communication tends to use C as a middle person. This requires stability in the API and ABI of the language.

The standard library is also more limited. *C* was the only program in the language that relied on systems not specified by their standard, as needed with some POSIX system calls and Pthreads library support.

Overall the tools in the language are more limited compared to *C++* and *Go*. This means much more code is required to do the same thing that is done with less in other languages. A notable example in this project is the limitation of IO only through files.

This is not necessarily a downside. I can provide much more direct control that is often vital to resource limited systems. Having fewer layers of abstraction between the developer and hardware allows for easier creation of code for operating systems, drivers, and runtimes.

Another advantage of its age and popularity is the wide range of open source libraries. Many codebase exist for compression, mathematics, and other features expected in a modern software project. This can help accelerate development since reuse is more readily available than compared to younger languages.

Similar to *C++*, *C* can reuse tools for sanitizing and analyzing programs. As a result of their similarities, the too languages share many of the same compilers, build systems, and other toolchain programs. This lack of standardization though can make it difficult to approach and increases the learning curve.

Although the language presents more difficulties when approaching the development, the performance results showed great potential. The limitation of features does push a simplicity that can be better optimized in most scenarios.

6 Conclusion

Systems languages provide programs with much more control to allow greater performance and interactions with hardware, often at the cost of simplicity and safety. Much more responsibility is placed on the programmer as they are no longer working in the strong confines of higher level languages.

This project helps to show the way the three languages approach this philosophy. *Go* takes a more modern approach combined with simplicity. This makes code development easier in certain scenarios but can limit the full potential of the computer. *C++* takes a wide breath of features while still exposing low level controls. This can greatly increase the possibilities for a developers approach but it comes at the cost of a difficult learning curve and greater responsibility for correctness. *C* takes an approach of stability to a long lasting language used for low-level controls. This has allowed a wide range of compatibility and software but at the cost of extra work and greater responsibility on the the developer.

Each language showed weaknesses and advantages. *Go* had great tools for quick development but often hit a ceiling faster when compared to the other languages. *C++* provides nice abstractions and more expressive code but often came with difficulty in development. *C* allows for great control close to the hardware but with the cost of extra work and responsibility.

A notable takeaway is the need for a strong understanding of the language and toolchain being used. Much of the improvements to the *Go* performance came from utilizing the composition features in the library. Both *C* and *C++* saw odd performance trends in FFT due to the way GCC worked.

In common practice, *Go*[2] often competes in a different realm than *C* and *C++*. *Go* has become widely popular in cloud software through its use for Docker and promotion by Google.

6.1 Further Work

This project only looked a limited slice of the world of system languages. Many other aspects should be investigated to gain a better understanding of the field.

Rust [6] is probably the most popular competitor to *C* and *C++*. This newer language provides some of the modern safety features directly baked into the language without the baggage of runtimes for memory management.

Smaller languages like Nim, Orin, and Zig are trying to dig out niches in the systems world. Each tries to approach different issues seen with past languages.

Different compilers exist for many of the languages. GCC has been a long standing staple for open source compilers. LLVM has both compilers and other language infrastructure that has become foundation to many modern tools. There are also some closed source compilers for the *C* languages.

Different platforms exist throughout computing. Often the software developed for a large server might start on consume devices as the developer begins with new ideas. Certain architectures might require different approaches to optimization.

References

[1] (2022) Cpp reference. [Online]. Available: https://en.cppreference.com/w/

[2] Google. (2022) Go. [Online]. Available: https://go.dev/

[3] (2022) Google c++ style guide. [Online]. Available: https://google.github.io/styleguide/cppguide.html

[4] (2022) Llvm coding standards. [Online]. Available: https://llvm.org/docs/CodingStandards.html

[5] V. Zverovich. (2022) fmt. [Online]. Available: https://github.com/fmtlib/fmt

[6] (2022) Rust. [Online]. Available: https://www.rust-lang.org/