Analyzing Compute Performance Across Languages and Hardware

Darren Baker, Stanford University

1 SUMMARY

There are thousands of programming languages in existence, with at least dozens or hundreds of these in common use in a broad range of industries and applications. Different languages offer different strengths and weaknesses along many dimensions that programmers may care about, including safety, performance, portability, developer productivity, and so on. The objective of this project is to explore specific trade-offs along two dimensions – performance and code complexity – for a few simple but numerically intensive algorithms implemented in several commonly-used programming languages. My experimental results show that while "dynamic" or "scripting" languages like Python appear to offer a moderate advantage in terms of code complexity and readability, "static" or compiled languages like C++ have a clear performance advantage – on the order of 10x for the simple algorithms under consideration here. Converting plain C++ code to Nvidia's CUDA – a C++ variant specifically designed to interface with Nvidia's GPU hardware – adds some additional complexity, but the performance benefit varies from minor to massive (~100x), depending on the specific algorithm being run and its adaptability to a massive parallel computing environment.

2 PROBLEM BACKGROUND

Though there are many programming languages available for both domain-specific and general-purpose computing tasks, not every language is created equal. Most programming languages in use today were created with certain purposes or design goals in mind; those goals are often clearly reflected in the structure and syntax of the language, as well as the many decisions that the designers made for the language's environment and runtime. For example, the Rust programming language that we studied in CS 242 was created by Mozilla to serve as a replacement for C/C++ in the organization's Firefox web browser, and so its design prioritizes features like speed and (safe) concurrency that are highly visible in its use of pointers and its data ownership/borrowing rules.

One common axis along which programmers evaluate the suitability of a language for a particular task is the spectrum from "lower-level" to "higher-level" languages. Though there is no precise boundary defined between "low-level" and "high-level" languages, or canonical agreement on which high-level languages may be higher than others, it is generally understood by computer scientists that the higher a language's level, the more abstraction it provides from the details of the underlying computer hardware. By modern definitions, languages like C are often considered relatively low-level, as they provide direct access to key elements of the computer hardware (primarily the memory), and they typically offer very little runtime support (in the form of a garbage collector or other execution environment above the level of the operating system). On the opposite end of the spectrum, very high-level languages like Python have extensive runtime environments that act as an intermediary between the program code and the operating system, and they rely heavily on abstractions and data structures that often simplify common tasks for the programmer, but also mask the structure and behavior of the underlying hardware. The usual trade-off is that higher-level languages allow for simpler, more compact, and more readable code, but code written in these languages usually runs more slowly than code in lower-level languages, because the

language runtime has to translate the language's programmer-friendly abstractions into a form that matches the computer's physical hardware and operating system interfaces. Thus, most programmers would bias toward C/C++ for performance-critical applications, while perhaps preferring Python for programs where performance is less critical but the benefits of Python's flexible data structures and rich standard library are pronounced.

The trade-off between various languages and programming models becomes even more stark when parallelization enters the picture. Parallel computation has long been a top strategy for accelerating performance, and it has become particularly important over the past 5-10 years as clock speeds and circuit density have begun to hit physical limits. The challenge is that parallel execution does not happen automatically: though compilers can introduce some forms of optimization, it is typically up to the programmer to write explicitly parallel code in order to take full advantage of distributed or parallel computing environments. This type of software is almost always substantially more complex than single-threaded code, requiring careful design on the part of the programmer and usually decreasing the readability and debuggability of the code. In order to allow the programmer to exert significant control over memory management and thread execution, parallel programming often needs a somewhat lower-level language that exposes specific details of the underlying hardware and operating system.

From my own experience working in a variety of programming languages, I was already somewhat familiar with the qualitative benefits and drawbacks of working in relatively lower-level languages (e.g. systems languages) and higher-level languages (e.g. scripting languages). In terms of the themes of CS 242, I believe these qualitative elements correspond well to the idea of "expressiveness." However, it was less clear to me how these "expressiveness" factors might stack up against the more directly quantitative "performance" factors resulting from the choice of one language or another. For example, if C were only 10% faster than Python for a particular computational task, but the Python code for that task were much more readable or significantly easier for the programmer to write and debug, then perhaps the speed gain from switching to C would not justify the corresponding loss of expressiveness and clarity. However, if C were 10 times faster than Python for that same task, the case for Python might be much less compelling, regardless of its user-friendly abstractions.

To explore these trade-offs more concretely, I decided to build my project around an experimental analysis of both the qualitative and quantitative factors of choosing different languages for the same task. The idea was to quantify the performance gaps between the languages while also considering the complexity of their code, so as to have a more reliable basis for judging which languages came closest to the "sweet spot" of great performance and great expressiveness.

3 EXPERIMENTAL APPROACH

The foundational element of my quantitative benchmarking approach was to identify one or more algorithms that were simple enough to be implemented similarly in multiple programming languages, but complex (and computationally intensive) enough to expose any meaningful performance differences between these languages. Since I intended to benchmark each algorithm in both sequential and parallel execution environments, it was also important that I choose algorithms that were relatively adaptable to a parallel computation approach. For example, a sequential search or sorting algorithm might have the

right level of complexity for my experiment, but it would not necessarily translate well to a parallel language. After some consideration, I settled on the following two algorithms:

- **Matrix multiplication:** Linear algebra computations play an important role in many classes of software applications, including graphics, robotics, finance, CAD, artificial intelligence, and more. With high-dimensional datasets or large numbers of variables, running even conceptually straightforward routines like plain matrix multiplication can quickly become computationally demanding. I originally considered implementing a semi-optimized form of matrix multiplication, such as Strassen's algorithm [2], but ultimately settled on the "naïve" algorithm since my purpose was not to maximize speed, but to compare speed across languages.
- **Numeric data compression:** Compression is a ubiquitous technique for reducing the size of data files, and is frequently applied to a broad range of data formats, including images, video, sound, text, and more. The computational requirements of compression algorithms vary by technique, but in most cases they are non-trivial, and both hardware and software systems are often optimized to maximize compression and decompression performance for common applications. For benchmarking purposes, I decided to implement an index compression routine like those often used in databases and web search systems. This routine takes a pre-built index – i.e., a mapping from terms to documents where the term appears – and uses bitwise arithmetic to encode each document ID using the minimum number of bytes required to represent the ID in binary form. This technique is known as variable-byte encoding [3].

The following table gives an overview of the steps involved in each algorithm:

Processing step	Matrix multiplication	Index compression
Data generation / preparation	A Python script is invoked with parameters for the number and dimension of matrix files to produce; for each file, a matrix is produced by selecting a random integer for each element, and the matrix is stored to disk in text format.	A corpus of Stanford web pages is processed to produce an index that maps string tokens to a list of numeric document IDs containing that token; the index is stored to disk as a single file in binary format.
Data loading	Matrix files are read into memory one at a time; each multiplication is carried out before the next file is loaded.	Entire index file is read into memory, with appropriate accounting for the binary format and endianness of the bytes being read; each term ID corresponds to a list of integer document IDs.
Core algorithm execution	The matrix is multiplied by itself, and the time required for the multiplication operation is measured. (All matrices in the test dataset are square, so dimensions are fully compatible.)	Each list of document IDs is encoded using gap encoding, and the resulting gap value is compressed into an array of bytes using variable-byte encoding. The time required for both gap encoding and VB encoding is measured.

Table 1: Description of algorithms used for performance benchmarking

The other key choice for my experimental design was the set of languages/environments in which I would benchmark the performance of these two algorithms. Based on the factors discussed in the "problem background" section above, I chose the following three languages:

- For a higher-level (a.k.a. scripting) language, I selected Python, since I have substantial personal experience with it, and it has become a widely used language for diverse applications in both industry and academia. Python is an interpreted, dynamically-typed language that offers a rich standard library and many convenient built-in data structures and syntactic shortcuts that make it very programmer-friendly in most situations.
- For a lower-level language, I chose C++, which is probably the closest thing to a global standard for writing performance-sensitive code. C++ is a compiled language that allows developers to access powerful capabilities of the underlying computing hardware, but it also has extensive libraries and object-oriented capabilities for building large software systems.
- To learn about the possible performance and complexity trade-offs of moving to a parallel computing environment, I also chose to implement each algorithm using Nvidia's CUDA language. CUDA is based on C++ and shares most of its core concepts with that language, but it has been extended in specific ways to give programmers access to the unique capabilities of Nvidia's GPU hardware. (For example, programmers can write functions that are specifically intended to be run in parallel on the hundreds or thousands of small processing cores built into every Nvidia GPU.)

In each language, I attempted to follow best practices for high-resolution tracking of the processing time required to run my core algorithms. I ran my experiments on a cloud-based compute instance provided by Google Cloud Platform. The machine I selected had 2 virtual CPU cores, 7.5 GB of main memory, and a single Nvidia Tesla K80 server-class GPU.

4 RESULTS & DISCUSSION

4.1 Evaluating Performance: Quantitative Results

After completing the implementations of each algorithm in each of the three languages described in the previous section, I carried out multiple benchmarking runs to evaluate the performance of each language on each task. My approach differed slightly across the two problems: in the matrix multiplication case, I was able to experiment with different numbers of matrices and different dimensions, since generating additional test data was fast and simple. In the compression case, since I was using a fixed corpus of real data, I didn't have as much latitude to expand the scope of the problem. Nevertheless, both algorithms revealed some interesting observations.

- First, the general trend of performance was clear and consistent with my original expectations. Python was the slowest language on each task, often falling behind C++ by roughly a factor of 10x-20x. CUDA on the GPU was the fastest on each task.
- Performance differences between the tasks became more pointed as the tasks became more computationally demanding. For example, on the multiplication task with small matrices (only 10x10 elements), the gap between Python and C++ was still relatively small in an absolute sense

– perhaps because both languages required some overhead to read the matrix files from disk. However, when dealing with matrices with 1000 elements in each dimension, the gap widened substantially. I expect this is because the overhead of reading files was still only an $O(n^2)$ operation, but the process of computing the actual product matrix is $O(n^3)$ – so the actual mathematical operations take up a much larger proportion of the CPU time required for the end-to-end run.

- One notable result was that the benefit of the parallelized GPU code in comparison with the "plain" C++ code was much larger for the matrix multiplication task than the index compression task. For example, on a 1000x1000 matrix, the CUDA version of the computation required less than a tenth of a second to multiply the entire matrix, vs. nearly 10 seconds required by the original C++ code (and more than 3 minutes required by Python). This represents a speedup of more than 2700x between Python and CUDA, and still more than a 100x speedup between the serial and parallel C++ implementations. In contrast, the index compression task showed only a 15 percent gain in performance between the baseline C++ implementation and the parallel version. This is likely because the index compression task still has an element of sequential processing that is difficult to avoid – and it also requires some extra data manipulation in order to work well on a GPU, as discussed in the next section.

The table and chart below show an example of average computation times for both tasks in each language. (Note the logarithmic scale on the chart's vertical axis, which allows a more clear reading of the relative times for each language on each algorithm.) Speedup factors for each language vs. the others are also highlighted in the lower part of the table.

	Task 1: Matrix multiplication			**Task 2: Index compression**
Data parameters	*Dimension 10x10*	*Dimension 100x100*	*Dimension 1000x1000*	*~99K documents, ~347K terms, ~13.8M index entries*
Python	2.04E-04	1.65E-01	196.560	21.350
C++	1.70E-05	8.76E-03	9.897	2.183
CUDA	1.25E-05	2.23E-04	0.072	1.904
	Speedup factors			
C++ vs. Python	12.02	18.85	19.86	9.78
CUDA vs. Python	16.35	740.04	2719.76	11.21
CUDA vs. C++	1.36	39.26	136.94	1.15

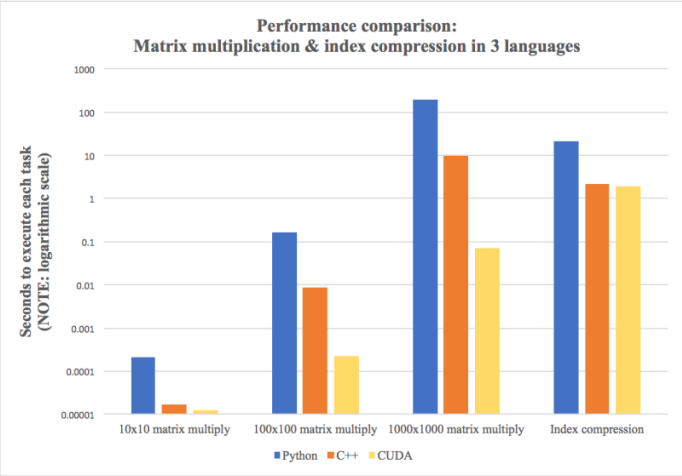

Table 2 (left): Time in seconds required to execute each algorithm, and speedup factors for each language relative to others. Times for matrix multiplication are for each single matrix product, averaged over 100 different matrices of the given dimension.
Figure 1 (right): Graphical representation of execution times for each language on each task, shown on logarithmic scale.

4.2 Evaluating Expressiveness & Code Complexity: Qualitative Factors

Developing code for the same task in 3 different languages and programming environments provided a useful foundation on which to compare the expressiveness of each language and its place on the spectrum from relative simplicity to relative complexity. Key observations that I made during this process included the following:

- Python was unquestionably the most compact of the 3 languages I considered. I was able to write code for each task in about 75 lines of Python code, while my C++ code averaged about 140 lines, and CUDA required closer to 200.
- I also found the Python code to be substantially more readable and easier to understand on later inspection than the C++ and CUDA code, which are relatively comparable to each other. This is a subjective judgment, but I believe it's a reasonable one – and I don't believe it's a result of a specific bias toward Python in my own programming experience, as I've almost certainly written more code in compiled languages than in scripting languages like Python over my software development career.
- A large part of Python's advantage in compactness and simplicity appears to arise from its easy built-in mechanisms for working with data and data structures of diverse types. Some of this functionality is core to the language itself, while some other parts are included in the standard library. For example, my implementation of matrix multiplication included some code to open the directory where the test data was stored, enumerate its contents, filter out files not matching the expected format, and open the remaining files to read and tokenize each line. This whole process took less than 10 lines of simple code in Python, while the same thing in C++ required roughly 25 lines plus two auxiliary functions that were not supplied by the standard library. Similarly, populating and iterating over lists of data in Python typically took only a single simple expression in Python, while C++ required explicit memory allocation and indexing for most structures of this type.
- One area where Python's advantage is less clear is in working with binary data. The index compression tasks relies heavily on binary manipulation: the original (uncompressed) index is read in binary format, and then the individual document IDs are compressed using bitwise manipulations of the underlying binary data. On one level, C++ has a meaningful advantage here, as its strong type system (and explicit types for things like "unsigned 8-bit integer") allows the programmer to be confident about how binary manipulations will operate on variables. Then again, Python still has some valuable library code for working with binary data – for example, to read multiple bytes directly from a file and convert them automatically into a specific data type – that still appears to be lacking in C++.
- Finally, it's worth saying a word about CUDA C++ in comparison with ordinary C++. The languages themselves are very similar, and I feel that Nvidia has done a good job making its CUDA-specific modifications easy to understand for programmers who are already familiar with C++. (Even with just some basic tutorials [4], I was able to get my parallel code up and running on the GPU without much difficulty, despite never having programmed in CUDA before.) The biggest challenge that arises with CUDA is the need to redesign specific computational tasks to work well in a parallel environment. For example, CUDA "kernel functions" (the bits of code

that run on GPU cores) must generally adhere to a specific format: they cannot have a return value, but must instead operate on shared memory, and each sub-task must be able to execute independently without dependencies on the results of any other sub-task. In the case of my index compression task, this actually required the index data to be converted from its "natural" format (used in the Python and C++ code) to a contiguous, fixed-size memory block, and then converted back again after the parallel computation was done. The overhead of these conversions almost certainly negated much of the performance boost that I achieved by performing the actual compression step in parallel, and ultimately meant that my CUDA implementation of the overall compression task was only slightly faster than the original C++ implementation.

5 CONCLUSIONS & FUTURE WORK

On one level, my observations of both quantitative and qualitative benefits for each of these three languages were unsurprising: performance goes up but expressiveness and simplicity go down when moving from a higher-level language (Python) to a lower-level language (C++) to a hardware-dependent language (CUDA). However, one thing that was somewhat surprising to me was the magnitude of the performance differences I observed, especially on the matrix multiplication task. Although Python may offer some benefits in terms of readability and developer productivity, it seems unlikely that its convenience makes things 10x or 100x easier on any qualitative dimension – while my results show that performance gains of that magnitude or greater may be available when making the switch to a natively-compiled language and/or a GPU. In this scenario, I find it difficult to claim that any performance-sensitive code should be written in a language like Python, regardless of its "expressiveness" advantages.

Moving forward, I would be interested in expanding my simple benchmarks to a wider range of languages. In particular, I am curious how the commonly-used class of bytecode-compiled languages – including industry stalwarts like Java and C# (or the .NET family more broadly) – would fare in this type of comparison. Based on personal experience, I believe that languages like C# actually strike the best balance between safety, performance, expressiveness, and availability of rich code libraries. It would be enlightening to test this hypothesis more rigorously by extending the experiments described here.

REFERENCES

[1] Miłosz Ciżnicki, Michał Kierzynka, Piotr Kopta, Krzysztof Kurowski, Paweł Gepner. *Benchmarking Data and Compute Intensive Applications on Modern CPU and GPU Architectures.* In Procedia Computer Science, Volume 9, 2012, Pages 1900-1909.

[2] "Strassen Formulas." Wolfram Mathworld, accessed 11/3/2017. http://mathworld.wolfram.com/StrassenFormulas.html

[3] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schutze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[4] Mark Harris. "An Even Easier Introduction to CUDA." Nvidia blog post, accessed 11/27/2017. https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

[5] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming.* Addison Wesley, 2013.