Undergraduate Research Project: Report I

A comparison between the usage of free and proprietary solutions for General-purpose computing on GPUs (GPGPU)

Isabella Basso do Amaral

Advisor: Alfredo Goldman Vel Lejbman
Mathematics and Statistics Institute (IME)
University of São Paulo
São Paulo, Brazil
gold@ime.usp

Abstract

The prevalence of code in modern research calls for a better understanding of the tools used to develop it. Certain mathematics routines have been exploiting the capabilities of Graphics Processing Units (GPUs) in what is called General-purpose computing on GPUs (GPGPU). This has become a common practice, and the development of software for this purpose has been facilitated by the availability of libraries and frameworks. This work aims to compare the usage of open source and proprietary solutions for GPGPU. We begin by analyzing project statistics and categorizing them according to their descriptions using unsupervised machine learning. We then compare performance and usability of the solutions, proposing enhancements to better support the development of GPGPU code with open source tooling. Finally, we discuss the implications of our findings for the development of GPGPU code.

Index Terms

Graphics APIs, GPGPU, open source, proprietary, machine learning, text mining, Vulkan, Vulkan Compute, OpenCL, CUDA

I. Introduction

A. Motivation

As computers became increasingly important in academic research, mainly with respect to numerical problems that are largely unsolvable through analytical methods, researchers have been looking for possibilities to take full advantage of their capabilities. In that attempt, software has become increasingly complex, accompanying evermore specific hardware to optimize common routines at the most basic level. One such piece of hardware is the Graphics Processing Unit (GPU), which emerged as a viable alternative to the Central Processing Unit (CPU) for a specific set of computational tasks, namely those that can be expressed as many independent operations that can be executed in parallel.

1) Parallelization: We usually measure compute performance by means of *floating point operations per second FLOPS*, as it conveys the notion of the average computation one would usually perform. It is important to notice that this was once a very straightforward measure, but as [1] notes, as hardware becomes more complex, we now need a variety of benchmarks to arrive at a more accurate measure of actual performance. As one can see in the figure 1, GPUs have, for a long time, been able to outperform CPUs in terms of FLOPS, even though the gap in performance has been closing in recent years.

With this in mind, we can see that GPUs have been consistently faster than CPUs for a long time (see [2]), which points us to the fact that parallelization is a key factor for performance.

In short, by taking advantage of uncoupled data, parallelization can cut execution time proportionally to the amount of computational units available, which are often presented as "cores". When talking about CPU cores, we commonly have abstractions as an attempt at enhancing performance even further than their, more raw, physical counterparts (see [3] for a common abstraction technique). We call the former, abstract cores,

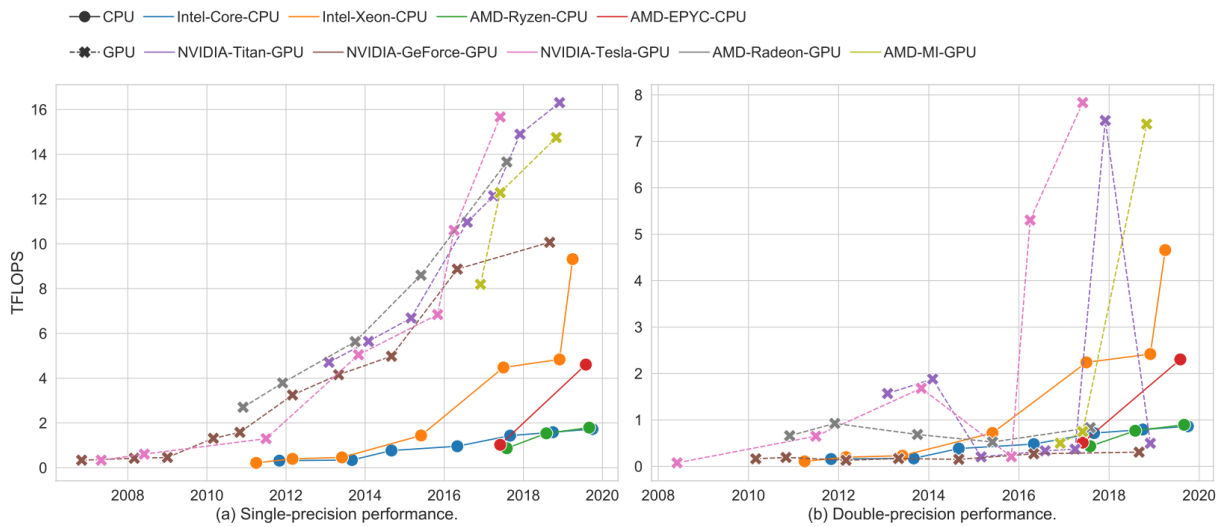

Figure 1: Comparison of single-precision and double-precision performance between CPUs and GPUs. (Source: [2])

logical cores, or, more commonly, *threads* and the latter *physical cores*, or simply *cores*. The same concept applies to GPUs, but often with a different nomenclature depending on the manufacturer of the chips and, more importantly, as hardware design differences can be more significant than in CPUs, we have to be careful when comparing raw component numbers between different GPUs.

Even though it is not always possible to convert a serial task into a parallelized one, some routines, such as those from linear algebra, can be thoroughly optimized from their most basic operations, which are often performed in uncoupled data a great many number of times – it is, of course, the perfect candidate for parallelization. The Graphics Processing Unit (GPU) is one such hardware that is optimized for these tasks, and has been gaining wide adoption in various markets – more importantly to us, in science, especially with the growing popularity of machine learning. We can clearly see that it outperforms CPUs in such tasks, as shown in [4].

2) Open source usage in science: Contemporary science's reliance on software is a relatively new phenomenon, and it does not seem to be held at standards as high as more traditional content (see [5]). It is also important to notice that such software is often developed by researchers inexperienced with real world development practices and, as such often lacks the quality and robustness that one would expect for long-term sustainability (see [6]).

One particular development strategy that appeals to scientific standards is that of open source, in which the code is fully available to people possibly unrelated to the project. In the peer review model it is expected that methods are auditable, and thus it would be beneficial to follow such a strategy (see [7]).

Building on the practice of open source we also have *free software*, commonly denoted by FLOSS or FOSS, a development ideology in which software also should not be dependent on corporations or even capital, praising volunteer work and donations, and criticizing corporate interests, usually accompanied by permissive or *copyleft* licenses. There is emerging work on the role of FLOSS in science, such as [8], and some initiatives which praise a similar approach (see [9], [10]). It is, therefore, in our best interest to explore just how much of the current scientific tooling is composed of free or open source software, and to also look for bottlenecks, where the community is not providing appropriate support for those applications.

B. Proposal

Distinct pieces of software usually communicate with each other through an Application Programming Interface (API), which allows for the exchange of information between them in an organized and standardized manner. GPUs are usually coupled to the rest of the operating system (OS) with a *Graphics API*, which should

provide a smooth bridge from high-level code to the actual hardware, with some APIs abstracting more than others or giving different functionality. Such APIs exist precisely because, as GPUs are independent pieces of hardware, for each vendor or product family there is likely a different set of instructions that can be executed on them, thus requiring a different set of drivers to interface with them. Given the difficulties generated by hardware specifics we generally prefer to abstract such implementation details, in a manner that we can have precise control of what matters to us without compromising performance, leaving it up to vendors to implement such APIs on their drivers in an optimal manner. Thus, we can see that the choice of API is a crucial decision for the performance of a given application, and it is important to have a good understanding of the trade-offs that come with each one.

1) Compilers: All "code" (generally meaning a meaningful, possibly executable, piece of text) is, in some sense, abstracted away for the programmer, and needs to be understood and translated to some format more useful to silicon. Those are the steps performed by an interpreter or a compiler.

In the context of GPU programming we write *shaders*. They are written in, unimaginatively, *shading languages*, which are usually a subset of C-like languages, with some extensions to support the specific needs of the API. Common examples include GLSL and HLSL, but shaders are actually a small part of GPU utilization as, for the most part, control of its routines is still performed on the CPU. Graphics APIs provide such control to the programmer, as they are made to orchestrate the entire pipeline: from managing and compiling the shaders to synchronizing CPU and GPU and even GPU cores amongst themselves.

Of course, calls to the graphics card are not free, and shader code also has to be compiled anyway, so during this process it undergoes further optimizations in, roughly, two steps:

1) The code is parsed and transformed into an *intermediate representation* (IR) which can then be optimized by the compiler using generic directives (i.e. independently of the target hardware);
2) The IR can then be translated into the target machine code, and also be optimized again, now for the target architecture.

The two translation steps are generally known as *front-end* and *back-end* of a compiler. After those steps the shader can finally be sent to the GPU. In Linux this is a small part of the execution of a shader, as we can see in Figure 2, represented by Mesa.

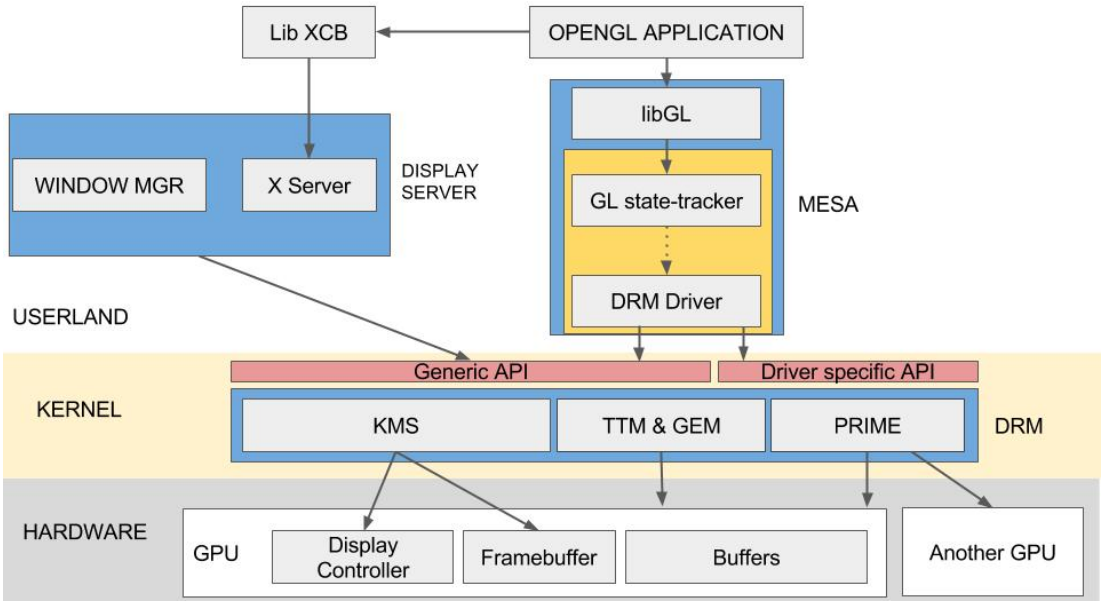

Figure 2: The graphics stack used in Linux kernel-based operating systems.

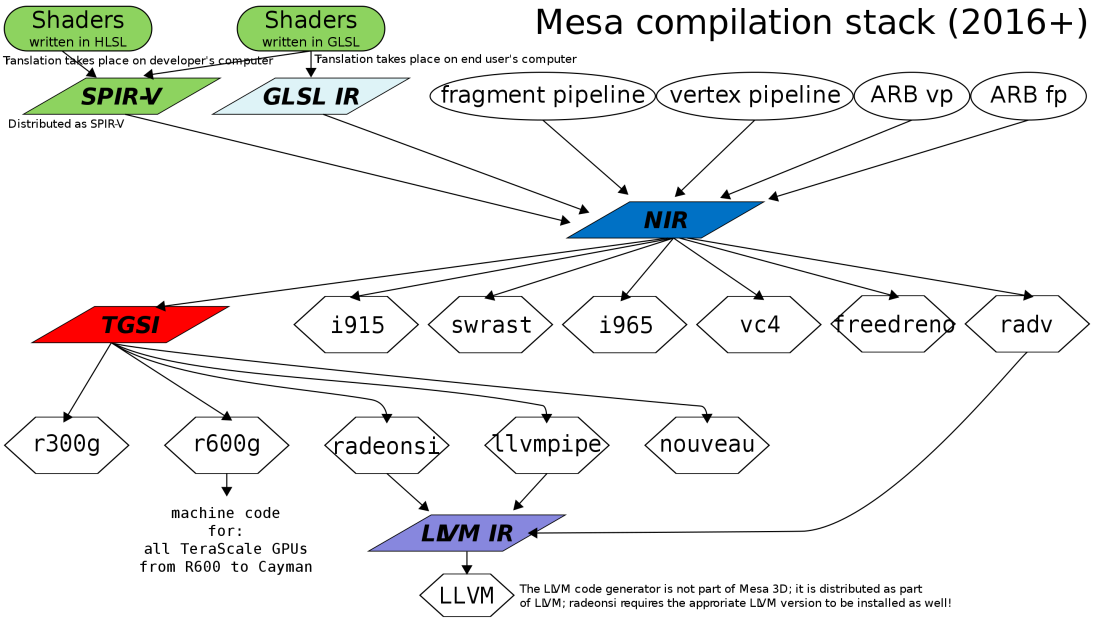

Figure 3: Most recent compilation pipeline used by Mesa, the software responsible for providing open source implementations for the userspace part of the Linux graphics stack.

2) A comparison of open source and proprietary graphics APIs:

In the context of General-purpose computing on GPUs (GPGPU), we notice that **CUDA**, a proprietary API projected and maintained by NVIDIA, has a significantly more widespread adoption than any other API, proprietary or not. On the one hand, most open source alternatives that provide the same functionality are either in their infancy or have not provided what was needed to replace CUDA, examples are, respectively, HIP by AMD, and OpenCL by Khronos Group. On the other hand, we have Vulkan as an alternative. It has a great upside of being more general, as it is not proposed solely for computation but also as a standard graphics API, providing display functionality. Thus, manufacturers (including NVIDIA) have implemented this standard and optimize it proactively, which is exactly what OpenCL has lacked since its release.

Given the perception of NVIDIA's technologies being the go-to choice for scientific applications.

We can estimate its "literature-share" with back-of-the-envelope calculations, as this fact lacks research to back it:

1) We query GitHub for `cuda language:cuda language:c language:c++ stars:>=5 followers:>=5` so that we can see how many projects have been specifically designed to leverage the CUDA API, and get 1835 results.
2) Now we make a similar query for the other, open source, APIs, and get 680 results for OpenCL and 895 for Vulkan. Results for HIP are not as reliable as "HIP" is often part of another word, and it only gives is about 100 results, so there is not enough room to isolate the noise.
3) As Vulkan is, for the most part, not used for GPGPU, we can rule out at least 2/3 of results, which gives us $1835/(680 + 895 * 1/3 + 1835) \gtrsim 65\,\%$ of potentially relevant applications supporting CUDA.

It is clear, then, that there is a great disparity amongst API adoption rates, especially taking into account the popularity of the applications in each group.

Given that Vulkan tries to cover much ground than any other API, it also has a significantly steeper learning curve as compared to CUDA, so that is an obvious challenge we are already aware of.

C. Objectives

Given approximate usage statistics of CUDA, we have some objectives to achieve:

1) Get a more precise figure of the usage of graphics accelerated open source software used in scientific applications.

2) Find out how CUDA stands out from its open source alternatives.
3) How can we overcome such differences and make it easier for maintainers' to add support for open source APIs.
4) Is there a way to make open source APIs more appealing to the scientific community?

We choose to analyze **Vulkan** and **OpenCL** as they are the most prominent open source alternatives to CUDA, and we will also take a look at **HIP** as it is the most recent addition to the list. Both APIs are Khronos Group standards, of which more than 170 companies are members, including AMD, ARM, Intel, NVIDIA, and Qualcomm. Their API specifications are open source and have ample documentation and adoption, which makes them a good choice for our analysis. HIP (Heterogeneous-Compute Interface for Portability) provides a solution to ease the transition from CUDA to more portable code, thus making it interesting to us as well.

II. Methodology

In October 2022 I have attended X.Org Developers Conference 2022 in order to present another project I have developed in Google Summer of Code 2022. While it was unrelated to the topic of this thesis, I got to meet many people from the open source graphics community, and learn much about the state of the art when it comes to open source graphics APIs, namely with respect to their usage and development.

In the MAC0414 course, I have learned many things about the theory behind compilers, mainly through [11], which should help me understand the compilation process of shader code, and how the pipeline is structured for each API. Then, in the MAC0344 course I have learned about the main aspects guiding performance in modern hardware, and also about some of their most common optimizations. I have also started to learn about graphics programming in general, starting by OpenGL, which is the most common API, and should provide a good basis for understanding Vulkan and OpenCL. I have also started to look up literature on benchmarks and performance analysis.

A. Statistics

So far, we have made some progress on item 1 by analyzing popular graphics-accelerated projects using topic modeling.

1) Data collection: First we assumed that most scientific applications use CUDA, and then by looking up all projects with the `Academia/Higher Education`, `HPC/Supercomputing`, and `Healthcare/Life Sciences` labels on NVIDIA's list of accelerated apps we got a list of 494 projects that should be using CUDA and, as recognized by NVIDIA, are at least somewhat relevant. From the 494 projects available with the tags above, we have narrowed the list down to 141 projects that are open source and use `git` as their version control system through manual inspection. After that, following [12]'s methodology, we gathered the repositories' `README` files and removed sections that were not relevant to the project, such as `Installation`, and `License`, then we used Latent Dirichlet Allocation (LDA) to group projects into topics, which should help us find out what each project is about.

2) Data analysis: As a first step, it was necessary to clean the data further, as there are many unnecessary words that would skew the results (such as `the`, `and`, etc.) which are generally known as **stop words**. We also remove symbols, such as punctuation, numbers and very short words, as they are not relevant to the analysis. Then it is necessary to tokenize the data, which means splitting the text into words, then lemmatize it, which means reducing each word to its root form, so that, for example, `computing` and `computations` are both reduced to `compute`. This makes analysis easier, as it reduces the number of words to a more manageable amount, and also makes it easier to compare words that are similar in meaning.

We then store the results of preprocessing in a count vector, which is a matrix where each row represents a document, and each column represents a word, and the value in each cell is the number of times that word appears in that document. So that we are abstracting away from the order of the words, thus making it easier to compare documents. In this step we also remove words that appear in less than 10 % of documents, or in more than 90 % of them, as they are either too common or too rare to be relevant to the analysis.

After that, we use LDA to group the documents into topics, which is a probabilistic model that assumes that each document is a mixture of topics, and each topic is a mixture of words. We train the model assuming

a number N of topics, which is a hyperparameter that we can tune to get the best results. LDA will then assign every word in every document to a topic, then look at the proportion of words in each document that belong to each topic, and try to find the best topic that fits that word. It will then repeat this process until it converges, which means that the topics do not change anymore.

Of course that are other hyperparameters that can be tuned, such as the densities for how many words should be in each topic (η), and how many topics should be in each document (α).

3) Hyperparameter tuning: We analyze `gensim`'s hyperparameters by looking taking a range of them and training the model as usual, then we look at the *coherence score*, which is a metric that measures how well the model is able to group words that are semantically linked.

First, let's look at the coherence score for different numbers of topics, see fig. 4. We can see that the coherence score is highest for $N = 8$, so we will use that value for the rest of the analysis.

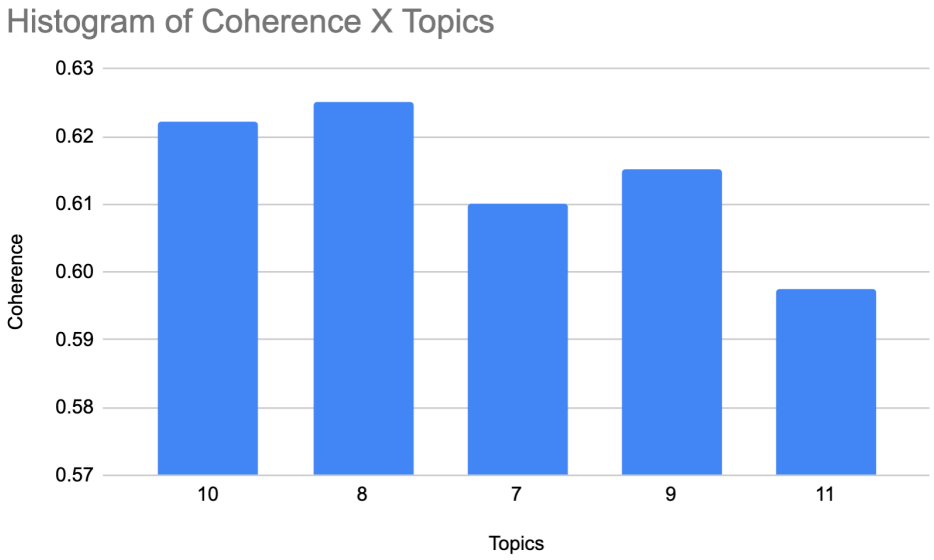

Figure 4: Average coherence score for a range of topics.

Now, for the other parameters, it's interesting to look at different samples of the data so we don't overfit the model to the data. Overfitting is a problem that happens when the model is too complex for the data, and it memorizes the data instead of learning the underlying patterns. So by optimizing results for the whole corpus but also for a smaller sample, we can avoid overfitting. In figures 5 and 6 we can see that the coherence score is highest for $0.4 \leqslant \alpha \leqslant 0.7$ and $0.1 \leqslant \eta \leqslant 0.4$. As the step between these values is 0.3, we can't really tell which value is optimal, so we reduce the range of both parameters to the intervals we found to be most promising, and look at the coherence score using a smaller step. Notice it's often a good idea to make ranges a little bigger, so that we can take into account optimal values might be near the edges of the range.

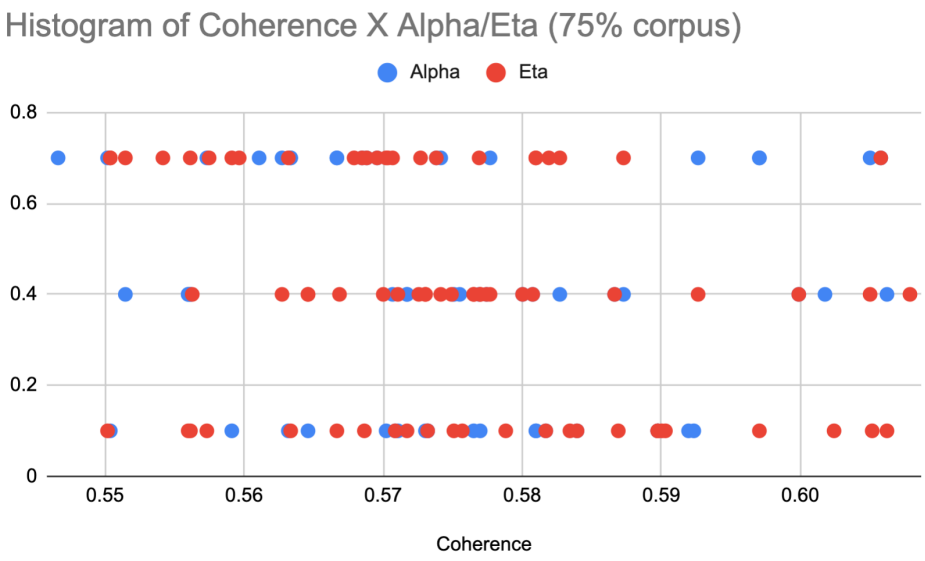

Figure 5: Average coherence score for different α and η using 75 % of the data (corpus).

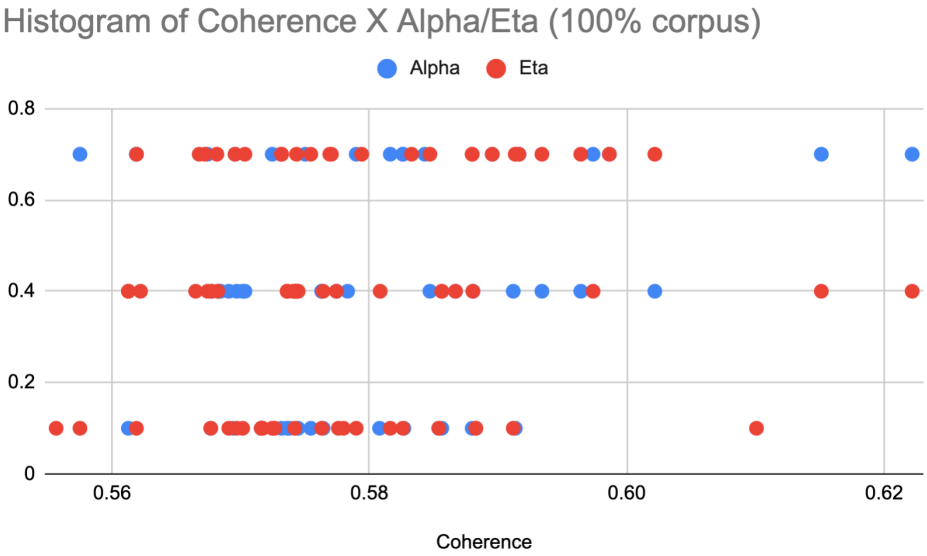

Figure 6: Average coherence score for different α and η using all of the data (corpus).

We continue narrowing down the intervals for α and η up to a precision of 0.01, for which the coherence score seems stable for neighboring values.

4) Implementation issues: We started by using the `gensim` library's implementation of LDA, namely the `LdaMulticore` class to train the model, and use the `CoherenceModel` class to calculate the coherence score and find the best hyperparameters. `LdaMulticore` is a parallel implementation of LDA, and in `gensim` it takes in a Bag-of-Words (BoW) and a corpus. The BoW is a list of lists, where each list is a document, and each document is a list of tuples, where each tuple is a word and its frequency. The corpus is a list of the words existent in the documents and repetitions have been removed, in it each word is associated with an index. We then used the `pyLDAvis` library, which is a visualization tool for LDA models, but it did not work too well, as the topics were either too broad or too overlapping, so we decided to try using the `scikit-learn` library instead, which uses a different algorithm to train the model. Then, using the same visualization library we

were able to get better results, which was indicative that there was something wrong with the way we were processing the data in some step on the attempt to use `gensim`.

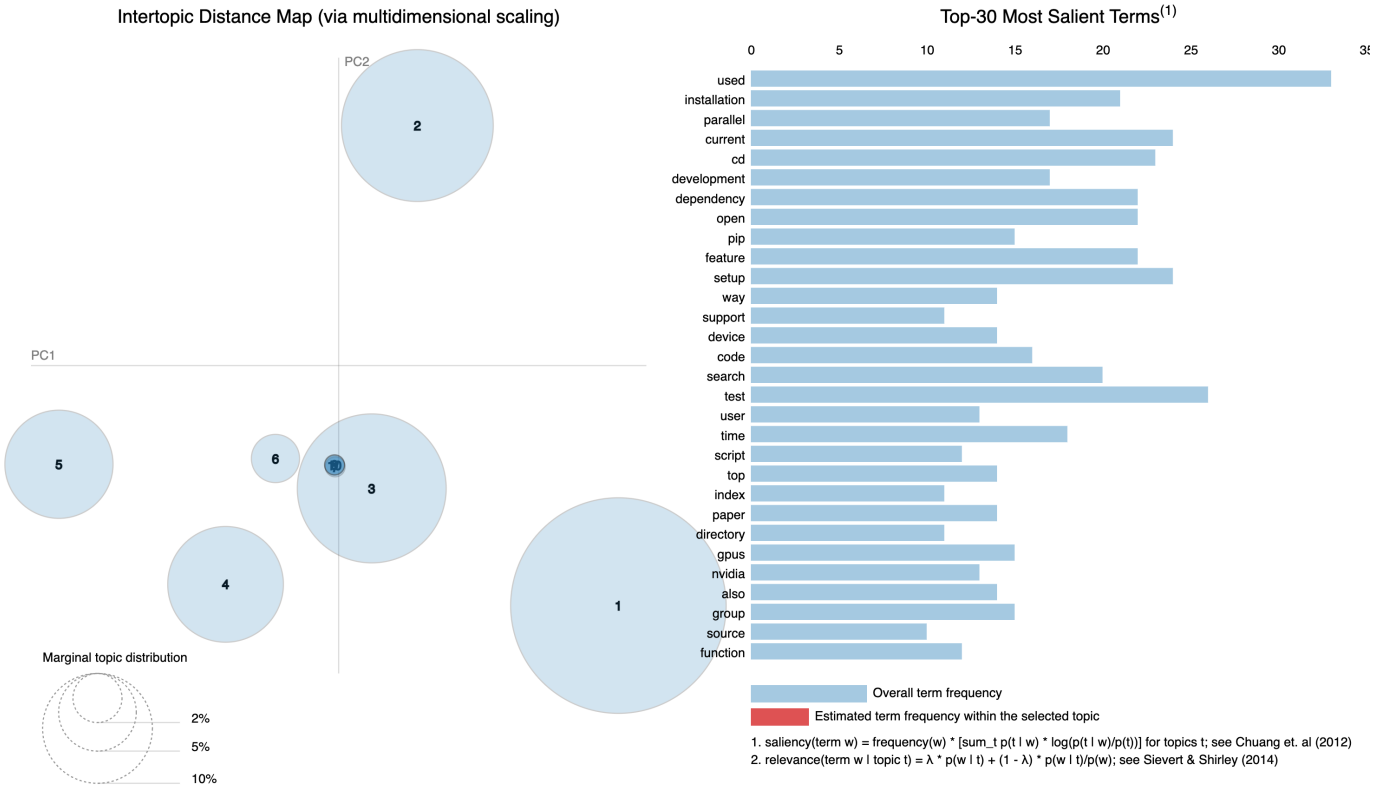

Figure 7: Visualization of gensim's LDA results (untrained model with default dimensionality reduction method).

`scikit-learn`'s LDA implentation takes in a different format of the data than `gensim`, as it uses a count vector. This required significant changes to the preprocessing steps, as we had to incorporate the count vector into the pipeline, and also change the way we tokenized the data (the preprocessing steps described above represent the latest implementation). As both models were trained with the same data, which was just converted from the count vector to a BoW and a corpus, we were unable to find the cause of the such a great difference in results at the LDA step.

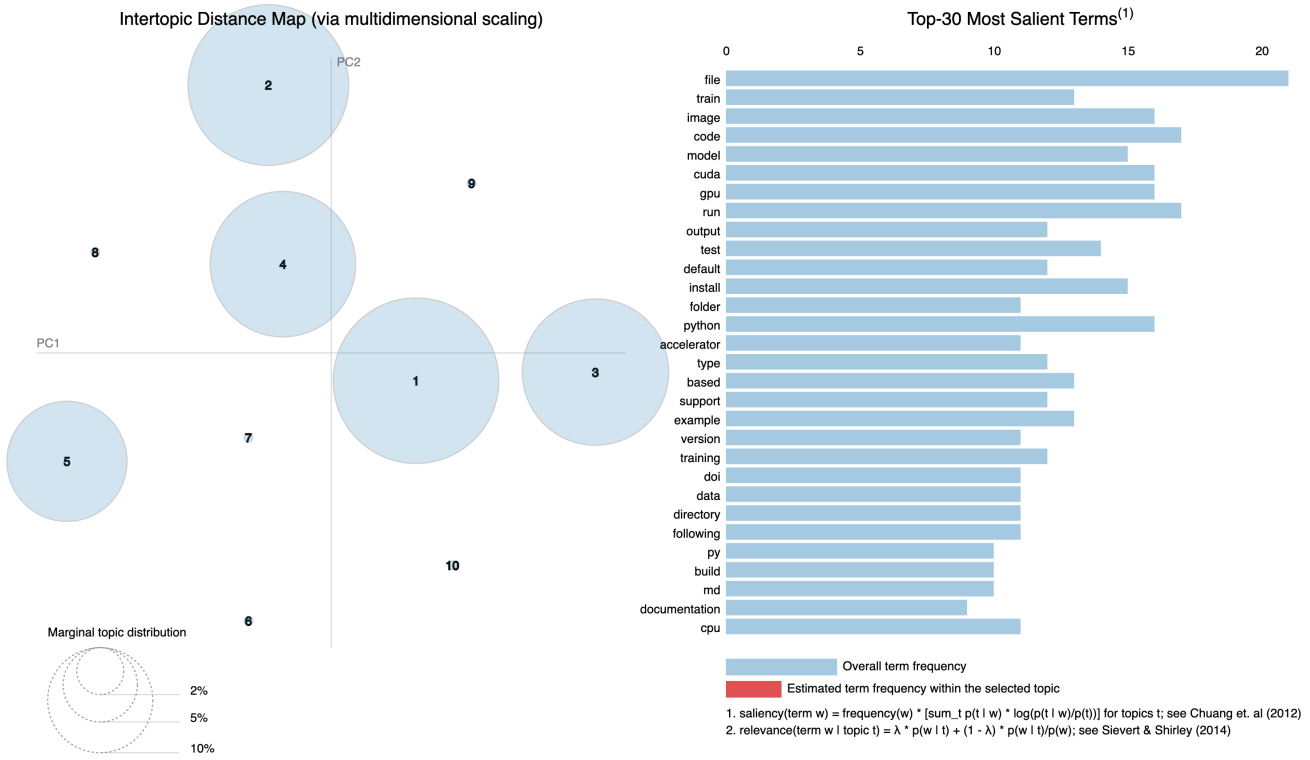

Figure 8: Visualization of SciKit-learn's LDA results (untrained model with t-SNE dimensionality reduction method).

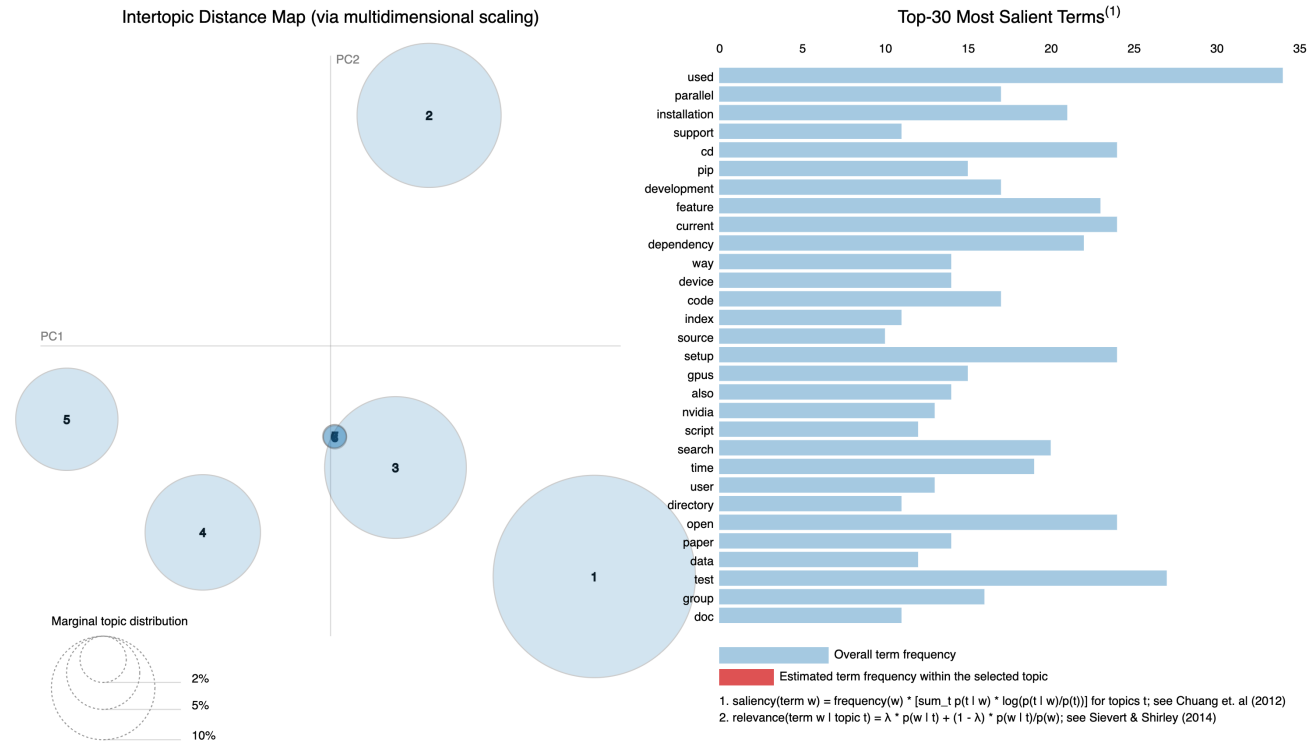

Figure 9: Visualization of gensim's LDA results (trained model with default dimensionality reduction method).

We can notice that `pyLDAvis` was able to generate better results with `scikit-learn`, even after tuning

gensim's hyperparameters, as the topics are more distinct (i.e. non-overlapping) and the words are more relevant to the topic, as we can see in figures 8 and 9. Upon further inspection it became clear that we were using a rather simplistic visualization technique for visualizing gensim's results. pyLDAvis, under the hood, uses dimensionality reduction techniques, which are needed to make the visualization work, and by default, it uses principal component analysis (PCoA), which does not need much manual tweaking, but it also does not work well with sparse data, which is our case. We were able to generate better results than either previous iteration by using the t-distributed stochastic neighbor embedding (t-SNE) method for pyLDAvis with a tuned gensim model, as we can see in fig. 10. T-SNE works well with sparse data, as it is better able to preserve local structure, but it needs manual tuning.

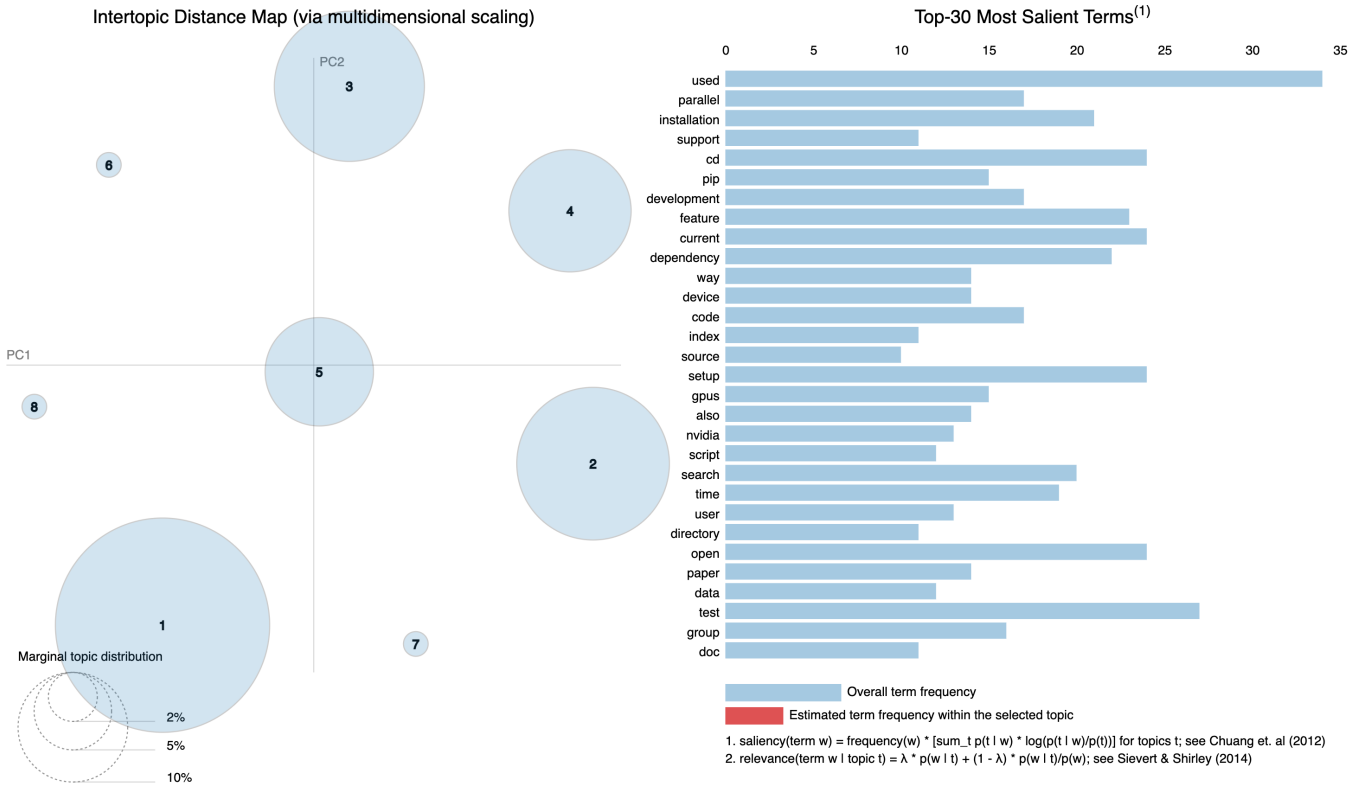

Figure 10: Visualization of gensim's LDA results (trained model with t-SNE dimensionality reduction method).

Unfortunately our results with LDA alone were not satisfying. See section IV-A for more details.

III. Discussion

By taking the most likely topic for each document, we were able to plot fig. 11. Looking at the words composing each topic, there was not a clear distinction between them, thus we were not able to make any conclusions about the repositories' actual subjects. Given that [12] were able to get better results with with a more complex model, we believe our analysis was too simplistic, as discussed in section IV-A.

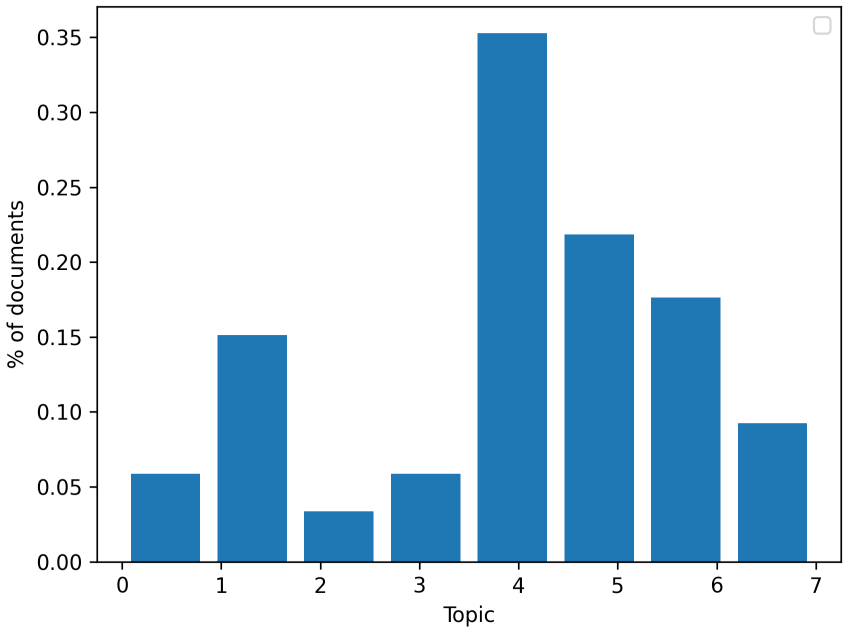

Figure 11: Topic probability distribution the documents in the corpus.

IV. Next steps

A. Statistical analysis

Not having much experience with data analysis, we have started looking into LDA as a single-step topic modeling technique, which is one of its most common usages, but this proved to be too simplistic for our needs, as [12] point out in their paper. We have then noticed the need to apply more complex techniques, namely Collaborative Topic Regression (CTR) in which one can use the latent space outputs of LDA as inputs to a regression using Probabilistic Matrix Factorization (PMF) to predict relevance ratings between projects. We also look at other relevance ratings to rank projects in a specific topic, such as the number of stars, forks and watchers on GitHub, as well as its commit frequency, number of contributors, and the number of issues and pull requests. CTR together with relevance ratings taken into account would allow for a more refined project categorization, making it simpler to select a sample of projects to use in the next steps, as we will be able to select the most relevant projects for each topic.

B. Graphics APIs

As we finish the statistical analysis phase, we then proceed to benchmark the projects we have selected, giving special attention to those that implement another graphics API besides CUDA. We will then compare the performance between top projects in each category, and analyze different implementations of similar routines. This requires understanding the different APIs, and how they work. It will also require finding useful benchmarks and tools, as well as possibly implementing our own.

Aside from personal studies, I'll also be taking courses in parallel programming and operating systems. Parallel programming will be useful for learning an overview of the CUDA API as well as for benchmarking the projects we have selected. Operating systems will be useful for learning about the internals of the operating system, compute pipelines, and how to optimize them.

References

[1] R. Dolbeau, "Theoretical peak flops per instruction set: a tutorial," *The Journal of Supercomputing*, vol. 74, no. 3, pp. 1341–1377, 2018.

[2] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, "Summarizing cpu and gpu design trends with product data," *arXiv preprint arXiv:1911.11313*, 2019.
[3] W. Magro, P. Petersen, and S. Shah, "Hyper-threading technology: Impact on compute-intensive workloads." *Intel Technology Journal*, vol. 6, no. 1, 2002.
[4] E. Buber and D. Banu, "Performance analysis and cpu vs gpu comparison for deep learning," in *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*. IEEE, 2018, pp. 1–6.
[5] S. Sufi, N. C. Hong, S. Hettrick, M. Antonioletti, S. Crouch, A. Hay, D. Inupakutika, M. Jackson, A. Pawlik, G. Peru *et al.*, "Software in reproducible research: advice and best practice collected from experiences at the collaborations workshop," in *Proceedings of the 1st ACM sigplan workshop on reproducible research methodologies and new publication models in computer engineering*, 2014, pp. 1–4.
[6] J. C. Carver, N. Weber, K. Ram, S. Gesing, and D. S. Katz, "A survey of the state of the practice for research software in the united states," *PeerJ Computer Science*, vol. 8, p. e963, 2022.
[7] L. A. Barba, "Defining the role of open source software in research reproducibility," *arXiv preprint arXiv:2204.12564*, 2022.
[8] L. Fortunato and M. Galassi, "The case for free and open source software in research and scholarship," *Philosophical Transactions of the Royal Society A*, vol. 379, no. 2197, p. 20200079, 2021.
[9] D. S. Katz, L. C. McInnes, D. E. Bernholdt, A. C. Mayes, N. P. C. Hong, J. Duckles, S. Gesing, M. A. Heroux, S. Hettrick, R. C. Jimenez *et al.*, "Community organizations: Changing the culture in which research software is developed and sustained," *Computing in Science & Engineering*, vol. 21, no. 2, pp. 8–24, 2018.
[10] M. Barker, N. P. Chue Hong, D. S. Katz, A.-L. Lamprecht, C. Martinez-Ortiz, F. Psomopoulos, J. Harrow, L. J. Castro, M. Gruenpeter, P. A. Martinez *et al.*, "Introducing the fair principles for research software," *Scientific Data*, vol. 9, no. 1, pp. 1–6, 2022.
[11] M. Sipser, "Introduction to the theory of computation," *ACM Sigact News*, vol. 27, no. 1, pp. 27–29, 1996.
[12] Z. Zheng, L. Wang, J. Xu, T. Wu, S. Wu, and X. Tao, "Measuring and predicting the relevance ratings between floss projects using topic features," in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, 2018, pp. 1–10.