Undergraduate Research Project: Report II

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.br

Abstract

Computer software has become an invaluable tool in our times, advancing the frontiers of what is deemed possible on every field, and that includes research. As scientific applications fringe on the capabilities of the conventional processor, researchers need alternatives to compute massive amounts of data reasonably quickly, opting for routines that have been optimized for parallelism. These routines exploit the capabilities of *graphics processing units* (GPUs) in what is called *general-purpose computing on GPUs* (GPGPU). In this research project, we aim to explore the usage of such solutions, mainly through the lens of reproducibility – a pillar of the modern scientific method – by comparing the usage of open-source and proprietary solutions for computing on GPUs. We begin by analyzing project statistics and categorizing them according to their descriptions using unsupervised machine learning. Then, we compare performance and usability of the solutions, proposing enhancements to better support the development of open-source GPGPU tooling.

Index Terms

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

I. Introduction

A. Motivation

The role computer software has acquired in our times is undeniable: from taking us to the moon, to providing optimized routes for delivery; it has become a staple to modern human endeavors. Naturally, research has also taken advantage of this tool, mainly with respect to numerical problems that are largely unsolvable through analytical methods. As datasets grow larger, researchers have adopted increasingly complex approaches to better utilize hardware, looking for possibilities to take full advantage of their capabilities. This has not come without its price, as there is an accompanying need for 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 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. FLOPS were once very straightforward to measure, but as [**?**] notes, modern hardware complexity now requires a variety of benchmarks to arrive at a more accurate measure of actual performance. As we can see in the figure fig. 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.

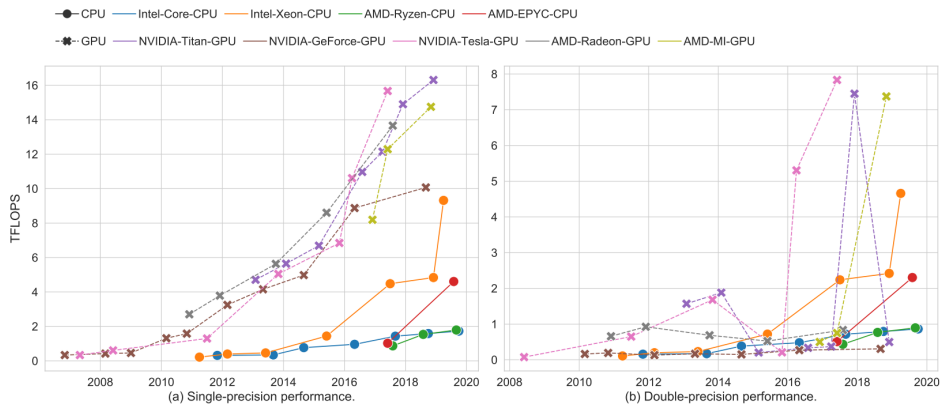

Figure 1: Comparison of single-precision and double-precision performance between CPUs and GPUs. (Source: [**?**])

With this in mind, we can see that GPUs have been consistently faster than CPUs for a long time (see [**?**]), 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 [**?**] for a common abstraction technique). We call the former, abstract cores, *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. But we also ought to be careful when comparing raw component numbers between different GPUs, as hardware design differences across manufacturers can be more significant than in CPUs.

Although 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 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 [**?**].

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

One particular development strategy that appeals to modern scientific standards is that of open source, in which the code is fully available to the public. Most obviously, as open-source software is auditable, it becomes easier to verify reproducibility of scientific results created with it (see [**?**]), but this also allows for early collaboration between researchers and developers, which can lead to better software design and performance (see [**?**]).

Building on the practice of open source we also have *free software* (commonly denoted by FLOSS or FOSS): a development ideology that focuses around volunteer work and donations, criticizing corporate interests, and is usually accompanied by permissive or *copyleft* licenses. There is emerging work on the role of FLOSS in science, such as [**?**], and some initiatives which praise a similar approach (see [**?**], [**?**]). 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 among 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 providing different functionality. Such APIs exist precisely because of them being independent pieces of hardware, needing to be commanded as a separate entity. Given the complexities that arise from hardware specifics, abstractions are a must, so 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: We refer to *code* as a meaningful, possibly executable, piece of text. All code is, to a varying degree, abstracted away for the programmer, and needs to be understood and translated to some format more useful to silicon. These steps are performed by an interpreter or a compiler.

In the context of GPU programming we write *shaders* or *kernels*. They are written in, unimaginatively, *shading languages* or *kernel languages*, respectively. Those are usually a subset of C-like languages, with some extensions to support the specific needs of the API of interest. Common examples include GLSL and HLSL, but those 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 among 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 is, then, translated into the target machine code, and can also be optimized, again, 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. On Linux this is a small part of the execution of a shader, as we can see in Figure fig. 2, represented by Mesa – Linux's standard userspace graphics driver provider.

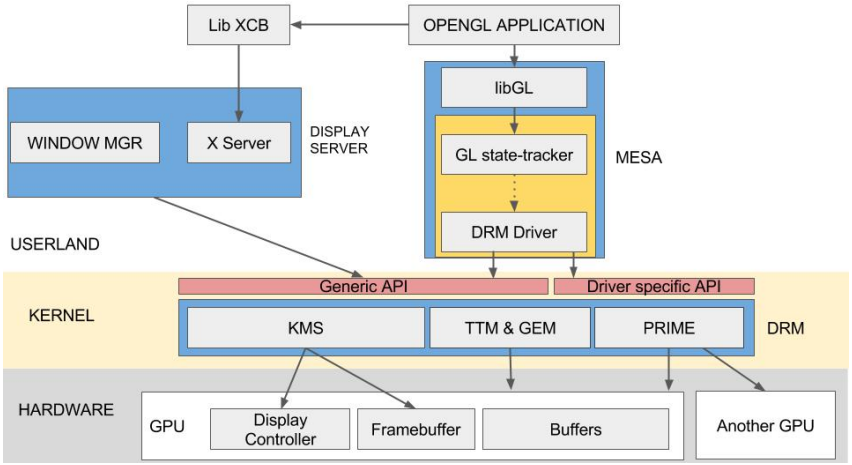

Figure 2: The graphics stack used in Linux kernel-based operating systems.

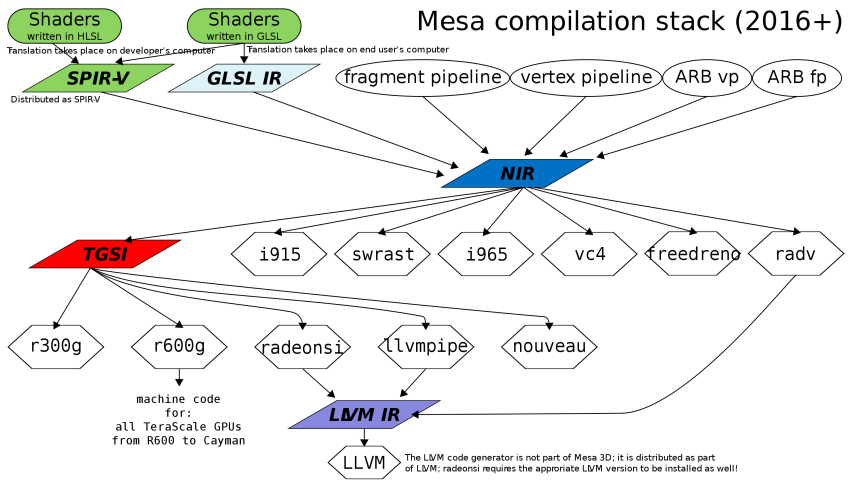

Figure 3: Most recent compilation pipeline used by Mesa, the software responsible for providing open source implementations for the userspace part of the Linux graphics stack.

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

In the context of GPGPU, **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 the 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.

We can estimate CUDA's "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 among API adoption rates, especially taking into account the popularity of the applications in each group.

Given that Vulkan tries to cover more ground than any other API, it also has a significantly steeper learning curve as compared to CUDA, making its adoption more challenging when compared to other alternatives. OpenCL-related tooling has recently gained more focus among open-source software consultancies, as we can see in [**?**].

C. Objectives

Given approximate usage statistics of CUDA, we lay down some of our objectives:

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) Find ways to overcome such differences and make it easier for maintainers' to add support for open source APIs.

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 Vulkan and OpenCL are Khronos Group standards, of which more than 170 companies are members, including AMD, ARM, Intel, NVIDIA, and Qualcomm. Their API specifications are open 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

A. First term

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 [**?**], which should help me understand the compilation process of GPU 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 both Vulkan and OpenCL. I have also started to look up literature on benchmarks and performance analysis.

B. Second term

I have learned about the main aspects of operating systems in the MAC5753 course, including various scheduling algorithms, details about memory management, as well as the main aspects of file systems' design. This knowledge has already helped me break down multiple applications and systems I have encountered, and should help me understand the internals of the driver implementations for each API. This course was also very code-intensive, which has helped me improve my C programming skills. I will also be making a presentation as part of the course, which will be about Contiki-NG, an operating system for the *Internet of Things* (IoT).

In the MAC5742 course, I have learned about the main aspects of parallel programming, and how to use the CUDA API. I have also studied code profiling and benchmarking, and how to use the `nvprof` tool to profile CUDA code, which is useful knowledge for evaluating the performance of the applications I have chosen to analyze. I have also made a group presentation about federated learning, which is a distributed machine learning technique.

In July 2023 I will present an extended abstract of this paper at the 2023 ERAD-SP conference.

C. 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 [**?**]'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, that is, to break the text into smaller pieces, called tokens. Then we lemmatize it, a process by which we reduce 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. That way we can abstract 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 *latent Dirichlet allocation* 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. LDA assigns every word in every document to a topic randomly at first, then updates its assignments according to the probability of a word belonging to a topic and the probability of a topic being assigned to a word. It will then repeat this process until it converges, so that the topics do not change anymore. We train the model assuming a number N of topics.

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 for how many topics should be in each document (α).

3) Hyperparameter tuning: We analyze hyperparameters by performing a 'GridSearchCV', using *coherence score* as the metric to optimize, as it measures how well the model is able to group words that are semantically linked.

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 indexed words existent in the documents, in which repetitions have been removed. 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`.

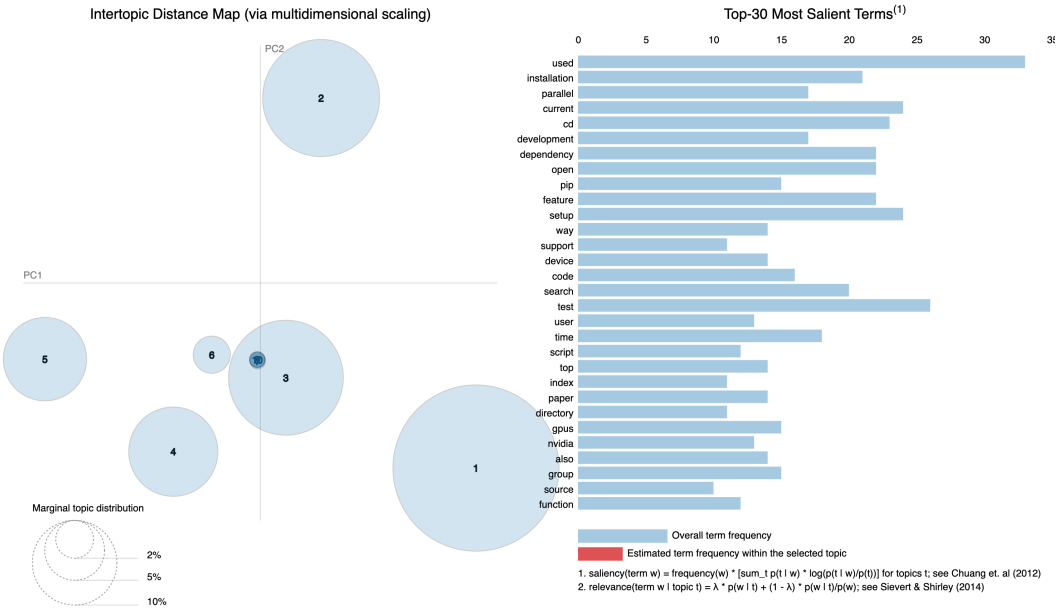

Figure 4: Visualization of gensim's LDA results (untrained model with default dimensionality reduction method).

`scikit-learn`'s LDA implementation 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.

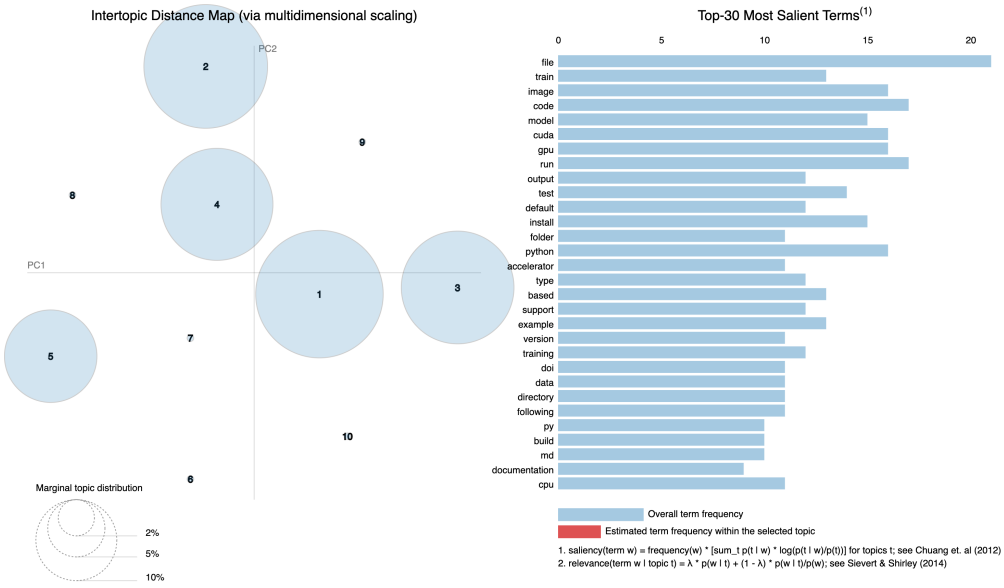

Figure 5: Visualization of SciKit-learn's LDA results (untrained model with t-SNE dimensionality reduction method).

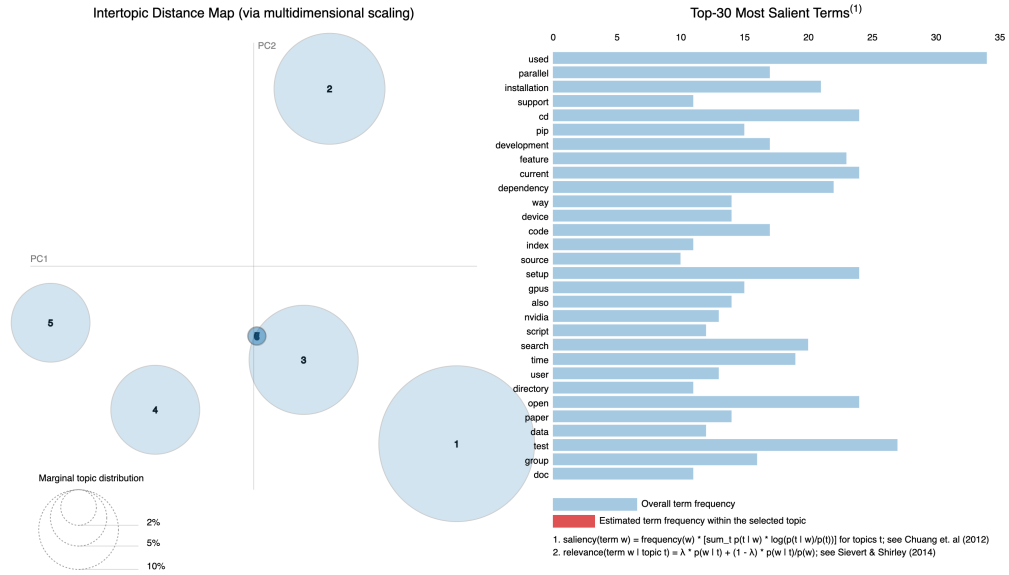

Figure 6: Visualization of gensim's LDA results (trained model with default dimensionality reduction method).

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 fig. 5 and fig. 6. Upon further inspection it became clear that we were using a rather simplistic

visualization technique for `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. 7. T-SNE works well with sparse data, as it is better able to preserve local structure, but it needs manual tuning.

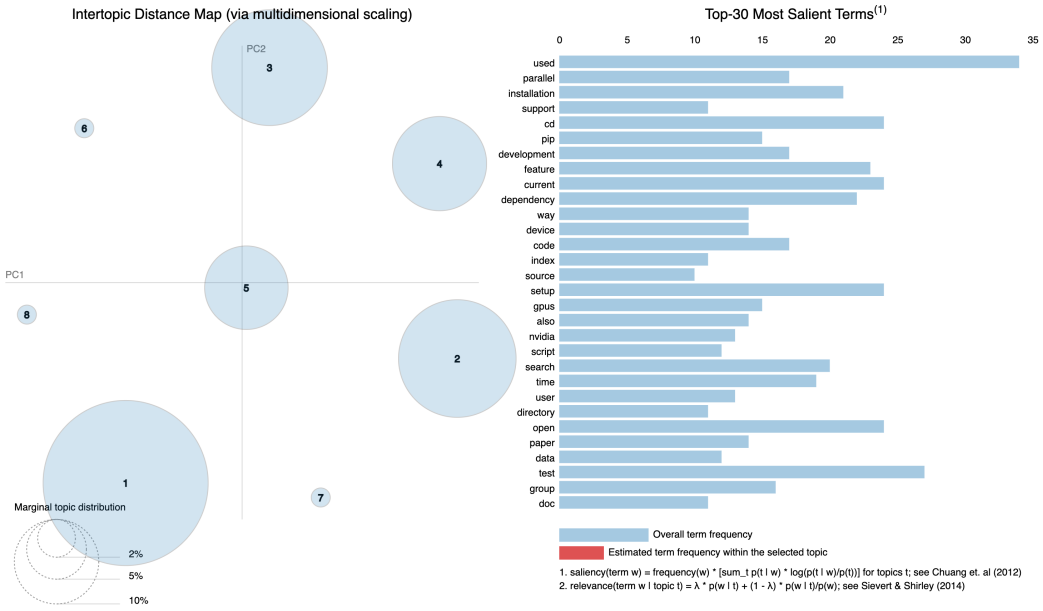

Figure 7: Visualization of gensim's LDA results (trained model with t-SNE dimensionality reduction method).

Unfortunately our results with LDA alone were not satisfying. See **??** for more details.

III. Discussion

By taking the most likely topic for each document, we were able to plot fig. 8. 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 [**?**]'s approach wanted to generate a recommendation model as its main goal, we believe our analysis needs to take on a different approach, as discussed in section IV-A.

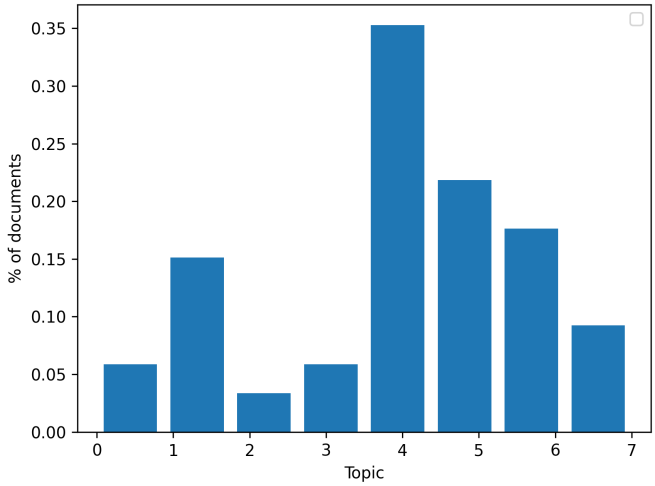

Figure 8: Topic probability distribution the documents in the corpus.

IV. Next steps

A. Statistical analysis

Reviewers from ERAD-SP suggested using *named entity recognition* (NER) to extract the repositories' subjects, as NER allows for collecting labelled data, which can aid in the training of a classifier.

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 will require more experience with the different APIs, as well as detailed analyses of the projects' source code. It will also require finding useful benchmarks for each application, and tools that are compatible to each.