# FAST LATENT DIRICHLET ALLOCATION

*Bianca-Cristina Cristescu, Benjamin Gallusser, Frédéric Lafrance, Saurav Shekhar*

Department of Computer Science
ETH Zürich
Zürich, Switzerland

## ABSTRACT

**The problem of automatically discovering and concisely representing what a document is "about", known as topic modeling, has applications in domains such as text mining, object recognition and recommender systems. Latent Dirichlet Allocation is one of the most popular algorithms to tackle this problem. However, most optimized implementations focus on parallelization or mathematical approaches. In this work, we focus on enhancing the performance of the algorithm on a single core. By employing standard approaches to optimization such as memory usage improvements and the use of SIMD instructions, we achieve large single-core gains. On our target architecture, we achieve a performance increase of more than 11x.**

## 1. INTRODUCTION

The amount of data available for mining has been growing at an increasingly rapid pace which shows no sign of slowing down. In order to take advantage of this data, it must be accessible, organized and usable. Just as libraries used to index books by category and topic to allow for more efficient searching, Latent Dirichlet Allocation (LDA) provides a method to determine a "short description" [1] of the documents of a collection through a generative probabilistic model. The original implementation [2] was designed to discover topics in text corpora such as news articles, but LDA has been applied to other tasks: object recognition, natural language processing, video analysis, collaborative filtering, spam filtering, web-mining, authorship disambiguation, and dialogue segmentation [1]. LDA is an unsupervised algorithm and requires minimal preprocessing of data, thus making it very useful in handling large datasets.

The sheer amount of data to manipulate and the demand of users to perform these kinds of tasks on mobile devices require a high-performance implementation of LDA. Thus, in this work we identify which performance optimizations can be applied, implement them and measure their effects.

### 1.1. Related work

Most of the efforts in the scientific community have been focused on improving the algorithmic complexity of LDA [3] [4] [5]. Topic modeling requires inference algorithms that must run over the entire collection of data. Therefore, to parallelize LDA one must undertake a distributed sampling [6] or variational inference [7] strategy. The distributed variational EM implementation shows little speed-up and does not discuss possible precision or accuracy losses. The distributed Gibbs sampling implementation preserves the predictive performance obtained by a one-core LDA implementation, and also shows potential speed-up. The approach presented in [8] used well-known distributed programming models, MapReduce and MPI, to parallelize LDA; results indicate that communication costs dominate the parallel LDA algorithm.

In contrast to the existing LDA performance improvements, our contribution is based on one-core optimizations. Our expectation is that the results obtained on one core can be transferred to a future parallel implementation.

## 2. DESCRIPTION OF THE ALGORITHM

In this section, we outline the main operations performed by LDA, the data structures on which these operations are performed on and the cost measure we use to quantify these operations.

### 2.1. Operations

As previously explained, LDA automatically discovers a user-given number of topics (modeled as probability distributions over words) in a corpus of unlabeled documents. Additionally, it infers the topic mixture of each document and the topic affinities of individual words in the documents. These two tasks are called respectively *estimation* and *inference*. They are not independent: inference requires the corpus-wide information produced by estimation, and estimation requires the document-specific information produced by inference, as part of an expectation-maximization procedure. LDA is therefore an iterative algorithm, where corpus and document data are updated in turn to give progressively better topic and word assignment estimates. Furthermore, estimation and inference are themselves iterative: they loop multiple times over the data they operate on.

## 2.2. Data

The data is represented by various matrices of floating point values. Their dimensions depend on the number of topics $K$, the number of unique words in the vocabulary $V$, the number of documents $N$ and the maximum length of a document in the corpus $D$. The main matrices are:

- Two corpus-wide matrices to record topic probability distributions (dimension $K \times V$).
- One corpus-wide matrix to record topic mixtures for all documents (dimension $N \times K$).
- For each document, one matrix to indicate the topic affinities for each word of the document (dimension $D \times K$).

Our working set typically consists of the information related to one document, namely the topic affinity matrix, and the columns of the topic-word probabilities corresponding to the words in the document. The algorithm iterates over this data multiple times in the inference procedure. In contrast, the estimation procedure runs over the topic-word matrix only once after having called the inference on each document (it then iteratively updates a scalar parameter of the corpus-wide topic distribution). Therefore, in order to fill the cache during the inference procedure, it is necessary to have long documents with many unique words. To this end, we use data from the European Parliament proceedings [9] in Finnish to create a small corpus of synthetic documents with these criteria. Unless mentioned otherwise, all results reported in this paper use this corpus, which has the following parameters: $N = 10$, $V \approx 200,000$, with a maximum of $D \approx 50,000$ unique words per document. Note that such an amount of unique words is possible thanks to the nature of Finnish.

## 2.3. Cost analysis

The computation is dominated by basic floating-point operations as well as logarithms and exponentials. Thus, the cost measure accounts for these operations separately. They are given equal weighting since we have no way of determining the exact cost of a $\log$ or $\exp$ in terms of the other operations. This leads to the following cost measure, determined by analyzing the code:

$$C(N,K) = \{\ \text{adds}(N,K),\ \text{muls}(N,K),$$
$$\text{divs}(N,K),\ \text{exps}(N,K),\ \text{logs}(N,K)\ \}$$

The individual cost of most functions depends on the cost of other functions. In addition, the overall algorithm cost depends on the number of convergence iterations over the corpus $\Gamma$, and over a single document $\Delta$. Due to the length and complexity of the precise cost formula, we only present the asymptotic one:

$$O\big(\Gamma \cdot K \cdot (\Delta \cdot N \cdot D + V + N^2)\big)$$

## 3. METHODOLOGY

In this section we describe the optimizations we performed, how we validate them, and a motivation for each optimization together with the expected results.

## 3.1. Initial analysis

We start from the existing C implementation provided by the authors of LDA [10], which we assume to be reasonably optimized in terms of mathematical and complexity optimizations. We instrument the code using Intel's `rdtsc` counter to perform runtime profiling. We profile each major function separately, excluding input-output operations and adding counters to determine the number of convergence iterations. Using the code analysis supplemented by profiling information shown in figure 1 , we make the following observations:

- The inference function, which iterates multiple times over a large $(K \times V)$ matrix and uses many expensive mathematical operations ($\log$, $\exp$), is very costly in terms of runtime.
- For some mathematical functions, such as `digamma` and `log_sum` we measure billions of calls. Thus, they represent a large share of the total runtime.
- Most of the matrix accesses are not stride-one and therefore not amenable to vectorization.
- There are very few opportunities for ILP and scalar replacement in the algorithm. This is because most computation does not reduce multiple values into one, but rather updates each value in turn, using operations which cannot be broken down further. Therefore, there is nothing that we can do that the compiler cannot do itself.

We decide to focus on architectural/memory optimizations first since they imply a restructuring of the code, which might affect the way we apply the subsequent computational optimizations. This is supported by a preliminary roofline analysis, in which we observe that LDA is memory-bound in respect with the computational vectorized roof.

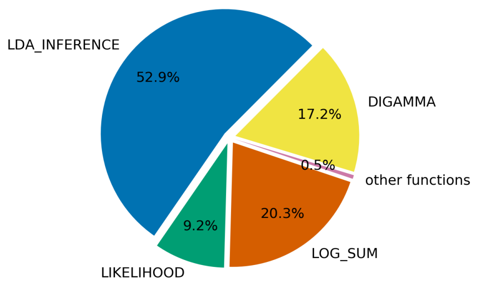

**Fig. 1**: Initial Runtime Analysis

## 3.2. Validation

In order to ensure our optimizations are sound, we perform a semantic validation on the output by comparing the top words in each topic with the output from the reference implementation. We require an overlap of at least $80\%$ among the top 20 words for each topic.

## 3.3. Memory optimizations

### *3.3.1. Spatial locality*

We first note that one of the corpus-wide matrices (dimension $K \times V$), which is accessed many times for each document during the inference procedure, is accessed column-wise. This is a reasonable choice by the authors since it captures intuitively the operation to be done (each column corresponds to a word, and we update topic affinities based on each word of the document). However, since the words vary from document to document, different subsets of columns are accessed in a non-sequential fashion for each document. In addition, rows are not contiguously allocated. These conditions create a recipe for a cache miss at every access: there is no spatial locality within a column, and the next column we access will probably not be in the same block of 8 columns that was loaded into cache. For the same reasons, the page fault rate will be higher than it should be. Specifically, one row spans over approximately 100 pages of size 4KB (using doubles). Because of non-sequential accesses, pages that need to be reused are evicted from the TLB with a high probability.

**Gather/Scatter.** The first approach to solve this issue is to *gather* the columns of the matrix used by one document in a smaller matrix, on which we can iterate sequentially and then *scatter* them back into the larger matrix. These operations must be done respectively before and after every call to the inference procedure. Since inference iterates multiple times over the matrix in a convergence loop, the additional data movement is justified. We expect the cache misses for the large matrices to decrease. However, the improvements might be mitigated by possible conflicts among a column's elements. Finally, by sequentially accessing contiguous columns, we reuse a set of $K$ TLB entries for around 500 columns. This implies that in the best case, we should reduce the page fault rate by a factor of 500.

**Transposition.** One problem with the Gather/Scatter approach is that the matrices are still accessed column-wise, which requires the use of shuffles to use vectorized instructions. We restructure the entire code so that the matrix representation is transposed. In this way, we now access whole rows and not whole columns. We also perform loop interchanges where necessary in order to obtain stride-one access.

In both of the optimizations we allocate the matrices contiguously to reduce the number of memory accesses by a factor of two (with non-contiguous matrices, one access is required for the row, and an additional one for the column).

### *3.3.2. Temporal locality*

We also investigate the feasibility of temporal locality optimizations such as blocking. Unfortunately, we find that we cannot apply them without making a substantial change to the algorithm. This is because chunks of data are reused only across convergence iterations. Inside a convergence iteration matrices are accessed sequentially. It is not mathematically proper to split a convergence into two just to force temporal locality: convergence of the parts may be ill-defined or not imply convergence of the whole. Therefore, we decide not to explore this avenue.

## 3.4. Computational Optimizations

### *3.4.1. Vectorization*

After implementing the memory optimizations, we perform another roofline analysis and find that the improved LDA is now in the compute-bound region. Taking into account the aforementioned limited possibilities for ILP, the consequent step is to vectorize the algorithm using AVX-2. Most of it is readily vectorizable thanks to the fact that we access the matrices with stride one. Given our prior bottleneck analysis (see section 3.1), we decide to exclude from vectorization a few functions (`trigamma`, `opt_alpha`) for which vectorization impact would be minimal.

### *3.4.2. Approximations*

In parallel with the vectorization efforts, we look into using various approximations. Since LDA is a probabilistic algorithm, we may be able to find an appropriate speed/precision trade-off, without influencing the final results. We attempt the following approximation strategies:

**Single precision floats.** The original algorithm uses double precision floating-point values. We modify the code to include a compile-time switch between single and dou-

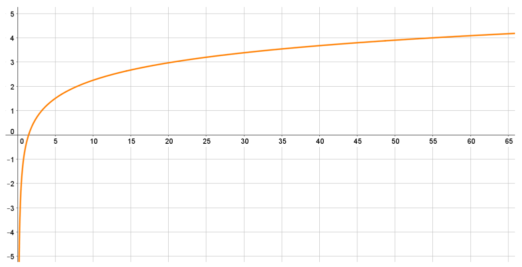

**Fig. 2**: Digamma function

ble precision, since using single precision will increase our performance by a factor of two when using vectorization[1].

**Lookup tables.** We note that after memory optimizations, our operational intensity is fairly high. We can therefore afford additional memory accesses in the form of lookup tables to reduce the amount of computations required. This is made possible by the fact that some mathematical functions we use have a restricted domain. Specifically, a substantial part of the high-impact function `log_sum` consists of computing $\log(1 + x)$, where $x \in [0, 1]$. This call has a latency of around 55 cycles. We implement a lookup table for this function as shown in algorithm 1.

**Algorithm 1** Log Lookup

```
1: Multiply the input $x$ by the number of bins
2: Cast the result to integer to obtain the bin index
3: Load the slope and intercept of the appropriate bin
4: Perform one FMA to linearly interpolate within the bin
```

This procedure requires equidistant bins between 0 and 1. Overall the cost is reduced to approximately 20 cycles assuming the table resides in L1 cache. This means that in practice, we select a relatively small number of bins (e.g. 100) since we must share cache space with the rest of the data.

**Asymptotic series.** For some mathematical functions, it is not feasible to use lookup tables, because they have a large domain which would require uneven bins to capture accurately. Figure 2 illustrates this with the digamma function $\psi(x)$, where many bins between 0 and 5 would be required and fewer after 5 (the domain of this function in the algorithm goes up to 2000). Thus, we try to approximate this kind of function (i.e. digamma and $\log(1 + x)$) using asymptotic series.

To compute digamma, the baseline code uses the following asymptotic series expansion from [11].

$$\begin{aligned}\psi(z) &\sim \ln z - \frac{1}{2z} - \sum_{n=1}^{\infty} \frac{B_{2n}}{nz^{2n}} \\ &= \ln z - \frac{1}{2z} - \left(\frac{1}{12z^2} + \frac{1}{120z^4} - \frac{1}{252z^6} + \cdots\right)\end{aligned} \tag{1}$$

[1]With the exception of a few intrinsics which only exist for single or double precision, and minor custom operations that require different shuffles.

where $B_i$ is $i^{th}$ Bernoulli number. For faster convergence, the authors of the baseline set $z = x + 6$ and use

$$\begin{aligned}\psi(x+1) &= \psi(x) + \frac{1}{x} \\ \Rightarrow \psi(x) &= \psi(z) - \frac{1}{z-1} - \frac{1}{z-2} - \frac{1}{z-3} \\ &\quad - \frac{1}{z-4} + \frac{1}{z-5} + \frac{1}{z-6}\end{aligned} \tag{2}$$

Equation 2 translates into a lot of divisions in the code, while equation 1 can be reduced to multiplications with $p = \frac{1}{z^2}$. We try substituting $\hat{z} = x + 4$ to reduce the number of divisions from 6 to 4. We can increase the number of terms in equation 1 to keep the same precision ($4^8 \approx 6^6$), or use the same number of terms to achieve even more speed.

Furthermore, we remove some data dependencies in the base implementation of the polynomial in equation 1 in order to achieve better ILP.

### 3.4.3. *Memory alignment*

It has been reported [12] that there is a small penalty on unaligned loads from cache on our architecture. Since most of our memory accesses hit the cache after applying other optimizations, we change the code to use aligned allocations, loads and stores. Note that this means we need to restrict the number of topics to multiples of 4 (which is not a problem since the number of topics is a relatively arbitrary input).

### 3.4.4. *Inlining*

As mentioned before in section 3.1, the small mathematical functions are not expensive by themselves, but they are called billions of times. For a small function, the procedure call overhead is considerable. To reduce the overhead, we declare them as inline which should have a positive runtime impact on the callers of these functions.

## 4. EXPERIMENTAL RESULTS

### 4.1. Setup

The experiments are run on a Intel Core i7-6700 processor (Skylake architecture, 3.4GHz, 8MB L3 cache, 34.1 GB/s theoretical memory bandwidth). Hyperthreading and all cores but one are disabled in order to ensure comparable and reproducible results. The compiler settings are: `gcc (4.8.7) -O3 -std=c99 -march=core-avx2 -fno-tree-vectorize` and `icc (14.0.1) -O3 -std=c99 -march=core-avx2 -mkl -no-vec`. When using flags for more aggressive optimizations (for example unsafe math optimizations), we find no measurable differences in runtime.

The peak performance of our system is 2 flops/cycle on sequential code and 16 flops/cycle on vectorized code and

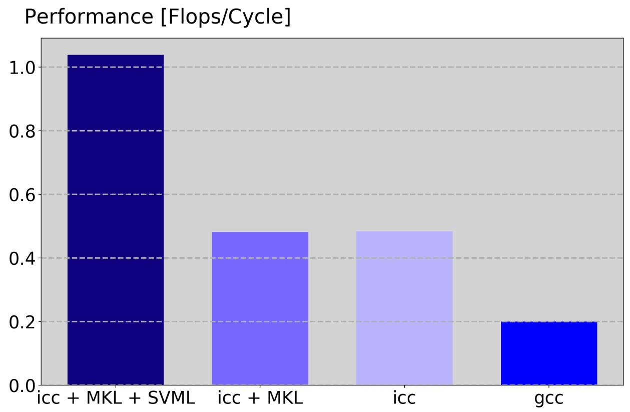

**Fig. 3**: Performance of the optimized LDA with different compiler setups

FMA with doubles. For our program, the actual peak performance is lower because of our operation mix. We have a relatively large amount of exp and log calls, but as explained in the cost analysis (section 2.3), they decompose into more basic instructions whose effect is hard to quantify. Therefore, we use the above numbers as peak.

Unless mentioned otherwise, we run our experiments with the following parameters. As input data, we use the synthetic corpus described in section 2.2. Since the amount of documents does not change the size of the working set, we instead vary the number of topics $K$ between 100 and 400 in increments of 100. Larger amounts of topics are impractical since running the reference implementation with $K = 400$ already takes 2.5h.

## 4.2. Compiler comparison

We decide to use the Intel C Compiler (`icc`), in order to use specific vectorized intrinsics for log, exp (available in SVML[2]) and `lgamma` (available in MKL[3]). We also implement non-vectorized versions of these functions to enable compilation with other compilers. We compare the overall performance of our final code on `gcc`, on `icc`, on `icc` + MKL and also on `icc` + MKL + SVML . The results in figure 3 show that the mere switch to `icc` results in an improvement of 2x in performance. This can be explained by the fact that `icc` is more aggressive in the optimizations performed for `-O3` (e.g. inlining). Using the MKL vectorized `lgamma` function does not impact the overall performance, mainly because it is a low impact function. Adding SVML for the log and exp intrinsics offers an additional 2.5x improvement on performance since these basic operations are heavily used by `LOG_SUM`, `LIKELIHOOD`, and `MLE`.

[2]Intel Short Vector Math Library, available with `icc`

[3]Intel Math Kernel Library

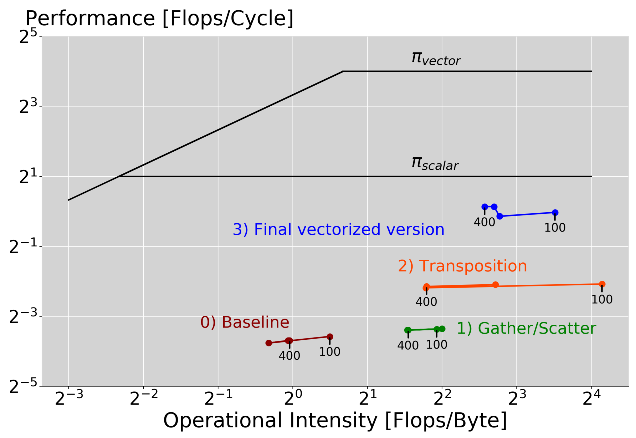

**Fig. 4**: Roofline plot with optimization stages 1 to 5.

## 4.3. Results for memory optimizations

The memory optimizations are applied to increase the operational intensity and move the algorithm into the compute bound area. The results in figure 4 show that the baseline implementation is memory-bound with respect to the vectorized roof. The Gather/Scatter optimization increases the operational intensity as expected, but we do not see a large improvement in the cache miss rate as shown in table 1. This is because there are still column-wise accesses and cache lines might be evicted because of conflicts generated by other operations performed before the next column is accessed. This intuition is confirmed by the results of the Transposition optimization in table 1 which, in addition to increasing the operational intensity, improves performance by a factor of 4x as a result of the row-wise access. This allows for a more accurate cache miss rate estimation. Looking at the TLB results in table 2, the Gather/Scatter optimization reduces the misses drastically. This proves that the accessed columns are indeed scattered over the matrix. The overhead of copying the columns needed for one document is absorbed by the temporal locality in the `INFERENCE` convergence iterations. On the other hand, Transposition does not improve the TLB miss rate further, since we are already accessing the matrix sequentially.

| | **References** | **Cache misses** | **Miss Rate** |
|---|---|---|---|
| Baseline | 264e9 | 39e9 | 14.7 % |
| Gather/Scatter | 144e9 | 11e9 | 13 % |
| Transposition | 3.6e9 | 133e6 | 3.6 % |

**Table 1**: Cache miss rate improvements

| | **References** | **TLB misses** | **Miss Rate** |
|---|---|---|---|
| Baseline | 4.5e12 | 3.5e9 | 0.08 % |
| Gather/Scatter | 4.5e11 | 44.7e4 | 9.8e−8 % |
| Transposition | 3.3e11 | 10e4 | 3.1e−7 % |

**Table 2**: TLB miss rate improvements

## 4.4. Results for computational optimizations

### 4.4.1. Vectorization

Looking at the roofline plot in figure 4, we observe a 4x speedup when we vectorize the algorithm using double precision. However, performance is still below both the vectorized and scalar roofs. It is mostly because, as mentioned in section 3.1, we have few opportunities for ILP and we count the expensive operations log and exp as a single flop even though they are more costly. This effect is notable since these operations have a large runtime impact.

### 4.4.2. Approximations

Since LDA is a highly iterative algorithm, any errors will tend to accumulate and potentially lead to invalid results. We observe this with our attempts at using approximations. Switching from doubles to floats removes enough precision that the inferred topics often fail validation and convergence takes longer. The same effect occurs when we attempt to reduce the precision of the asymptotic series.

Using lookup tables for `digamma` and `log_sum` has a more dramatic effect. Since we linearly interpolate between two points of the function, we systematically approximate a value lower than the actual value of the function (this follows from Jensen's inequality). Therefore, the sign of all errors is the same, they do not cancel out, and this may lead to runtime errors. We decide not to use more bins in the lookup tables since we want them to stay in cache.

### 4.4.3. Memory alignment & Inlining

We find that neither memory alignment nor inlining bring significant improvements. The penalty saved by using aligned memory is small and does not affect performance much on our set-up architecture. In addition, `icc` already has good heuristics for inlining. As a consequence, forcing further inlining does not improve runtime.

## 5. CONCLUSIONS AND FUTURE WORK

Based on our results, we consider that we are reasonably close to the potential peak performance of this algorithm. Indeed, our roofline analysis does not take into account our particular instruction mix and heavy use of exp and log, which means that the actual roof is lower.

There are limited avenues for further optimizations. As previously explained, additional ILP and scalar replacements will be difficult to achieve due to the nature of our computations. Further, temporal locality only occurs between iterations of convergence loops, which cannot be blocked safely. We could use better approximation schemes such as splines. Although it is unlikely that we could beat SVML for exp and log, we could try approximating digamma. The main difficulty is to ensure that the errors remain small while preserving performance. Unfortunately, we did not have the time to explore this further.

One avenue we have not had the time to explore is parallelization. Since the inference step is independent for each document, we could process them independently and then collect the results. This would have an especially noticeable runtime impact on corpora containing many medium-to-small documents (i.e. the situation opposite to the one we tested in this paper). We believe that we would then have one of the fastest non-distributed implementations available.

## 6. REFERENCES

[1] Diane J. Hu, "Latent dirichlet allocation for text, images, and music," .

[2] Andrew Y. Ng David M. Blei and Michael I. Jordan, "Latent dirichlet allocation.," pp. 993–1022.

[3] David Newman Teh, Yee Whye and Max Welling, "A collapsed variational bayesian inference algorithm for latent dirichlet allocation.," vol. 6.

[4] Ian et al Porteous, "Fast collapsed gibbs sampling for latent dirichlet allocation.," .

[5] R. Gomes, M. Welling, and P. Perona, "Memory bounded inference in topic models.," .

[6] Asuncion A. Smyth P. Welling M. Newman, D., "Distributedinferenceforlatentdirichlet allocation.," .

[7] Cohen W. Lafferty J. Nallapati, R., "Parallelized variational em for latent dirichlet allocation: An experimental evaluation of speed and scalability.," .

[8] Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen, and Edward Y. Chang, "Plda: Parallel latent dirichlet allocation for large-scale applications.," .

[9] Philipp Koehn, "Europarl: A parallel corpus for statistical machine translation," in *MT summit*, 2005, vol. 5, pp. 79–86.

[10] David Blei, "C implementation of variational em for latent dirichlet allocation," https://github.com/blei-lab/lda-c, 2004.

[11] Milton Abramowitz and Irene A Stegun, *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*, vol. 55, Courier Corporation, 1964.

[12] Agner Fog and Nathan Kurz, "Agner's cpu blog," http://www.agner.org/optimize/blog/read.php?i=415, 2015, accessed: 2017-06-11.