AN EFFECTIVE OPTIMIZATION ON LEAST ANGLE REGRESSION

Li Chang

Department of Computer Science
ETH Zürich
Zürich, Switzerland

Yu-Chen Tsai

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Dictionary learning is a technique that learns a basis set and applies it to a specific data. Such approach has been proven to be very effective for signal processing and classification problems. Recent paper [**?**] has proposed an algorithm that can processed large data set using the idea of *online learning*. The backbone of this dictionary learning algorithm is Least Angle Regression (LARS) that reconstructs the specific data, often called after signals, using the basis in dictionary.

While training a model, the shorter time each iteration needs, the more data can be consumed and the more general the trained model can be. Therefore, in this project, we aim to optimize the LARS algorithm which is not only the main component of online dictionary learning but is also widely used for applications related to feature selection.

We started with implementing a baseline version from scratch, referencing a more general implementation in C++ and CBLAS [**?**]. We optimize the baseline implementation by reusing computations, improving locality, and applying instruction-level parallelism and vectorization. We achieved a 7.2 times speed-up in total and increase the performance from 0.57 flops/cycle to 4.10 flops/cycle.

1. INTRODUCTION

Least Angle Regression (LARS) is a well-known sparse coding algorithm proposed by Bardley Efron et al[**?**]. The algorithm is known for being less greedy and more computationally efficient to its ancestors, Forward Selection and Forward Stagewise. One of the most common application of LARS is features selection. Often, while training on really large dataset, the number of features within a data significantly increase the time to train the model. To decrease the number of features that represent a data while preserving the characteristics, the algorithm that is used to select the remaining features is important.

Dictionary learning, widely used in machine learning, signal processing, and image processing, is a typical case that uses feature selection algorithms. The goal of dictionary learning is to approximate a given signal using only a few basis elements from the learned dictionary. More specifically, given a bunch of signals Y in $\mathbb{R}^d$ and a dictionary X in $\mathbb{R}^{d\times k}$, with k columns referred to as atoms, the algorithm is meant for finding a linear combination of most representative atoms from the dictionary.

To handle very large training sets and dynamic training data changing over time, Mairal et al. proposes an online learning algorithm for dictionary learning[**?**]. In each iteration i of the algorithm, sparse coding is used to compute the sparse representation β_i of a random sample y in $\mathbb{R}^d$ from a distribution $p(y)$. β_i is then used to update the dictionary X.

While feature selection algorithm like LARS are implemented to decrease the amount of time needed for training, it is obvious that the overhead of applying it should be as little as possible. And therefore, the goal of this paper is to proposed optimization methods that are applicable to implementations of LARS and its variants, such as LASSO, and justify these optimization methods do speed up the entire procedure and enhance the performance.

In the following sections, we formally define LARS, introduce incremental Cholesky decomposition and shows a detail cost analysis on the entire procedure in Section 2. We present the optimization we applied in Section 3 and show the result in Section 4. At last, we summarize our work in Section 5.

2. BACKGROUND

In this section, we provide necessary information to understand the implementation of the LARS algorithm. We first formally define LARS and explain the algorithm that solve the given formulas in Section 2.1 We then introduce Incremental Cholesky decomposition in Section 2.2 and formulate how we calculate the cost of the program in Section 2.3. At last, in Tabel 2 we present the exact number of operations of the entire program.

2.1. A Formal Definition of LARS

A L1-regularized linear regression problem and its dual form, solved by LARS, is described respectively as follows:

$$min_\beta ||y - X\beta||_2^2 \; s.t \; ||\beta||_1 \leq l$$

$$min_\beta ||y - X\beta||_2^2 + \lambda ||\beta||_1$$

A more detailed explanations of how LARS produces a piece-wise linear solution path is described in Algorithm 1. Initially, the current residual is equal to the input y. The correlation between the current approximation and the current residual is initialized (line 2). In each iteration, the correlation is updated and the most correlated basis to current residual is added into the active set A (line 4 - 6). The unit vector of the equiangular direction, u_A, is computed in line 7 - 11, and the distance to walk is computed at line 12. The current approximation of y is then updated at line 13. The `while` loop continues until the $\hat{\lambda}$, computed at line 17 is greater than the given parameter λ.

Algorithm 1 Compute sum of integers in array

1: **procedure** LARS(X, y, λ)
2: $\hat{c} = X^T y$, $\hat{\beta} \leftarrow 0$
3: **while** COMPUTE_$\lambda() < \lambda$ **do**
4: $\hat{c} \leftarrow \hat{c} - X^T \hat{\beta}$
5: $\hat{C} \leftarrow \max_j |\hat{c}_j|$, $A \leftarrow \{j, |\hat{c}_j| = \hat{C}\}$
6: $s_A \leftarrow \{sign(\hat{c}_j), j \in A\}$
7: $G_A \leftarrow X_A^T X_A$
8: $w_A \leftarrow G_A^{-1} s_A$
9: $A_A \leftarrow \sqrt{1_A^T w_A}$
10: $u_A \leftarrow A_A X_A w_A$
11: $a \leftarrow X^T u_A$
12: $\hat{\gamma} = \min_{j \in A^c}^+ \left\{ \frac{\hat{C} - \hat{c}_j}{A_A - a_j}, \frac{\hat{C} + \hat{c}_j}{A_A + a_j} \right\}$
13: $\hat{\beta} \leftarrow \hat{\beta} + \hat{\gamma} u_A$
14: **end while**
15: Return $\hat{\beta}$
16: **end procedure**
17: **procedure** COMPUTE_λ
18: $\Lambda = X_A^T (X_A \hat{\beta} - y)$
19: Return $\max_{\lambda_i \in \Lambda} \{|\lambda_i|\}$
20: **end procedure**

2.2. Incremental Cholesky

w_A (line 8, Algorithm 1 is the weighting function that composes the new direction to march along. w_A is decided by the inversion of the correlation matrix of all the current active basis, G_A. Since the active set grows by one in every iteration, G_A has to be recalculated according to the current active set. Instead of solving the entire inverse matrix, which is $O(n^3)$ in complexity, in every iteration, we only update the inverse matrix according to the newly added basis. Such technique is called Incremental Cholesky, and is only of $O(N^2)$ complexity for each update. We separate Incremental Cholesky into two parts:

Update Cholesky solver. Suppose we have a correlation matrix $X_{A_{t-1}}^T X_{A_{t-1}}$, in iteration t, to update the correlation matrix with the new basis v, we only add a new column and a new row to the previous correlation matrix, getting the new correlation matrix:

$$G_{A_t} = \begin{bmatrix} X_{A_{t-1}}^T X_{A_{t-1}} & X_{A_{t-1}}^T v \\ v^T X_{A_{t-1}} & \sqrt{v^T v} \end{bmatrix}$$

. We then update the lower triangle of G_A with:

$$L_t = \begin{bmatrix} L_{t-1} & 0 \\ v^T X_{A_{t-1}} (L_{t-1}^{-1})^T & \sqrt{v^T v - |v^T X_A (L_{t-1}^{-} 1)^T|^2} \end{bmatrix}$$

To compute $v^T X_{A_{t-1}} (L_{t-1}^{-1})^T$, we solve w for

$$L_{t-1} w = X_{A_{t-1}} v$$

with Gaussian elimination.

backsolve the target. Inverting correlation matrix G_{A_t} is inverting the corresponding Cholesky decomposition LL^T, which can be done by solving a lower triangular system and a upper triangular system sequentially.

operation	cmp	add	mul	fma	div	sqrt	abs
flop count	1	1	1	1	16	24	1.5

Table 1: Relative flop count of operations.

2.3. Cost Analysis

Each math function (add, mult, div, sqrt, etc.) is associated with a specific number of floating point operations. We derived the corresponding floating point operations according to the ratio of its throughput to the throughput of floating point addition. For example, a division of two doubles is counted as 16 floating point operations, because the throughput of `_mm256_div_pd` is $1/8$ while `_mm256_add_pd` is 2[**?**]. The relative floating point operation counts of all code segments in the algorithm is listed in Table 1.

Since we are solving sparse and over-complete representation of the input, dictionary size k is usually larger than signal dimension d. The number of iterations in LARS depends on regularization parameters λ and the value of target signals. For simplicity of analysis, we set $k = 2d$ and $\lambda = 0$, so that the algorithm always terminates on the d^{th} iteration.

For optimization purposes, we counted the exact floating point operations within different parts of the algorithm

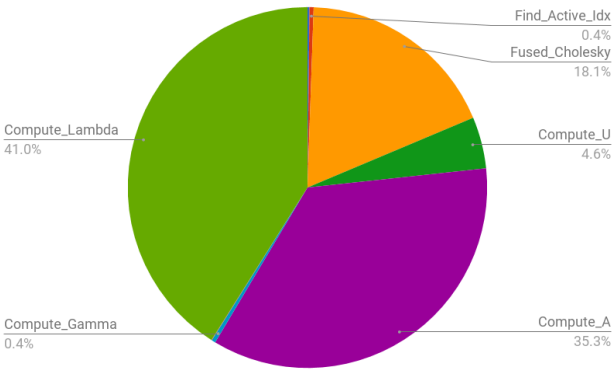


Fig. 1: Run-time before optimization

LARS as shown in Table 2. With Table 1 and Table 2, we can calculate the floating point operation counts of each code segment.

3. OPTIMIZATION ON LARS

In this section, we fist briefly explain the baseline implementation of the LARS algorithm and state assumptions made to simplify the code. We then elaborate on optimization on most time-consuming parts of the algorithm and go through some general optimization.

Since each step of the algorithm requires results from the previous step, we segmented our code into several segments accordingly, as mentioned in Table 2.

The baseline is a straight-forward implementation of Algorithm 1 in C++ from scratch. The dictionary X, the largest array of the algorithm, is store in a column-majored format before any optimization. For implementation simplicity, we assume both dimensions of the dictionary, d and k, are power of two and divisible by 16. Figure 1 shows that COMPUTE_LAMBDA, COMPUTE_A, and CHOLESKY are the most expensive components. Optimizing these parts therefore has the highest priority.

3.1. Reuse previous computation

Most parts of our algorithm are computational bounded. Therefore, we try to find redundant computations and reuse the result of previous computation. Possible methods proposed are storing previous computed value and reuse previous computation according to mathematical implications.

Trade computation with memory. In the beginning of each iteration, LARS check if the current approximation hit the regularize parameter λ. The equation used to compute λ is shown in line 17 in Algorithm 1. It is clear that

$$\begin{aligned}\Lambda &= X_A^T(X_A\hat{\beta} - y) \\ &= X_A^T X_A \hat{\beta} - X_A^T y \\ &= G_A\hat{\beta} - X_A^T y\end{aligned} \tag{1}$$

Since G_A is calculated when updating the Cholesky solver (line 7, Algorithm 1), we store this value in a new array. This requires more memory for variables since, previously, solving an incremental Cholesky system only need the last row of G_A in each iteration. This method not only save computation time but also reduce accessed memory. It decreases the number of memory access from $\mathbb{R}^{|A|\times k}$ to $\mathbb{R}^{|A|\times|A|}$, where $|A|$ is the size of the active set. We achieve significant speed-up in this segment with this modification.

Remove computation using mathematical implication. In the baseline version of the implementation, the target value s_A is re-computed at every iteration before fed into the Cholkesy solver(line 6, Algorithm 1). The reason to do so is that the correlation between the current residual and the target is updated according to the current approximation in each iteration (line 4, Algorithm 1). Since the algorithm assures convergence, it implies that the sign of the correlation does not change once it is added into the active set. Therefore, instead of recomputing the sign of each entry in every iteration, only the sign of the added entry is calculated.

3.2. Improve locality

Since there are a lot of memory access throughout the algorithm. We proposed several effective methods to take advantage of CPU caching.

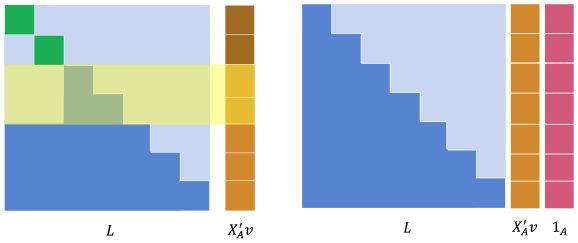
Compress stored data. To keep track of the Cholesky decomposition of the correlation matrix G, we record the lower triangular matrix L, where $G = LL^T$. Since all the entries of L above the main diagonal are zero and never used, instead of storing L as a square matrix, we store only the lower triangle as a flattened one dimensional array. Even though in this case we lose much data alignment, it turns out to bring more benefits by fitting more of L into cache.

The gram matrix G_A also appears in the optimized version for computing λ in the beginning of each iteration. Instead of storing the entire gram matrix, we only stored the lower triangular matrix using the compressed form.

Switch to column orientation for L^T. The Cholesky updating step solves a system of L while the Cholesky inverting step solves a system of L and then a system of L^T. To solve two systems of L and one of L^T, the program needs to go through every entry in L three times. We don't store L^T explicitly. Instead, while solving a system of L^T, the program accesses L in the required order. As a Gaussian elimination can be conducted in either row orientation or column orientation, we pick the proper orientation for the system of L and the system of L^T respectively.

Code Segment	Line	cmp	add	mul	fma	div	sqrt	abs
INIT_CORRELATION	2	0	0	0	$2D^2$	0	0	0
FIND_ACTIVE_IDX	4-6	$7D^2 + D$	0	0	0	0	0	$\frac{D(D+1)}{2}$
FUSED_CHOLESKY	7-9	D	$\frac{D(D+1)}{2}$	0	$\frac{D(D+1)(8D-2)}{6}$	$\frac{D(D+1)}{2}$	$2D$	$\frac{D(D+1)}{2}$
COMPUTE_U	10	0	0	$\frac{D(D+1)}{2}$	$\frac{D^2(D+1)}{2}$	0	0	0
COMPUTE_A	11	0	0	0	$2D^3$	0	0	0
COMPUTE_γ	12	$4D^2 + 3D$	$2D^2 + 2D$	0	0	$D^2 + 2D$	0	0
UPDATE_β	13	0	0	0	$\frac{D(D+1)}{2}$	0	0	0
COMPUTE_λ	18	$2D^2$	0	0	$\frac{D^2(D+3)}{2}$	0	0	$2D^2$
TOTAL	*	$13D^2 + 4D$	$\frac{5D(D+1)}{2}$	$\frac{D(D+1)}{2}$	$\frac{D(29D^2+42D+1)}{6}$	$\frac{D(3D+5)}{2}$	$2D$	$D(3D+1)$

Table 2: Cost analysis on each code segments of Algorithm 1. D is the dimension of the target signal.



(a) Block triangular solve (b) Fused triangular solve

Fig. 2: Optimization of Cholesky decomposition

Fuse L solving. Furthermore, we fuse the Cholesky updating step with the Cholesky inverting step for both of them are solving a system of L. We can solve the system with different right-hand sides simultaneously. This way we only need to go through L twice, one for the systems of L and the other for the system of L^T.

While reusing the value of the gram matrix G_A to compute λ, both the lower triangular matrix of G_A and its transpose are needed for computation: $G_A\hat{\beta} = L_{G_A}L_{G_A}^T\hat{\beta}$. We fuse the two multiplication by accessing each entry of L_{G_A}, the lower triangular matrix of G_A, only once.

Blocking. All calculations done on the triangular solver in Cholesky is blocked with block size 128, so that each block fits in the L2 cache. Similarly, when computing u_A and a (line 10 and line 11 in Algorithm 1), the matrices and vectors are also blocked to size 512 to improve locality. Computing λ (line 17 in Algorithm 1) is also blocked and is set to a block size of 16 after several testings on the optimal block size.

Access large data structure in order. Since X_A is only a permuted subset of columns in X, we does not really record every entry in X_A. Instead, we only record the order of the number of column in X that is added into X_A in our baseline version.

To get the direction to move on, it is necessary to compute $u_A = A_A X_A w_A$ and $a = X^T u_A$, as shown in line 10 in Algorithm 1. The problem here is that to access X_A in order, we will be jumping from rows to rows in X. Since X is large, it might cause multiple larger level cache misses.

To prevent this from happening, the best way is to sort the active set A so that you access everything throughout the algorithm in order. However, since we use incremental cholesky in our algorithm, we could not directly sort the active set in every iteration. Therefore, we only attempt to access large data chunk, X, in order, while accessing smaller component u_A incontinuously.

3.3. Instruction-level parallelism and vectorization

After improving the locality we vectorize most of our code with Intel X86 AVX2 intrinsics. With the specific operations provided by the library, we are able to indicate FMA operations, unroll for loops and remove if branches.

unroll and scalar replacement. To avoid blocking and increase instruction-level parallelism, we also implemented unrolling plus scalar replacement in for loops. We also tested unrolling in each segment of the code until we get the best unrolling factor.

remove if branches. To calculate the absolute value, the sign value, checking whether a column is already activated and getting only values that satisfy some constraints create expensive if branches throughout the algorithm. We therefore replace these if branches by combining `blendv` and `cmp` to get rid of the overhead caused by if branches.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the optimization strategies proposed in Section 3. We detail the settings of the machine that we performed our test on, and then present the results of our optimization.

	line size	# ways	# sets	size
L1 data cache	64	8	64	32K
L1 instruction cache	64	8	64	32K
L2 unified cache	64	4	1024	256K
L3 unified cache	64	16	8192	8M

Table 3: Specification of Intel i7-6700 CPU @ 3.40GHz (skylake).[**?**]

4.1. Experimental setup

The program is implemented in double-precision floating points due to precision problems. All performance experiments are conducted on Red Hat Enterprise Linux 7 running on a machine with the specifications as shown in Table 3.

The maximal memory bandwidth advertised is 34.1 GB/s, but in practice it's guaranteed to be less than that. We hence use *bandwidth*[**?**] to measure sustained memory bandwidth for different access patterns. From the benchmark result in Figure 5, the memory bandwidth of random read and write are 21.2 GB/s and 3.10 GB/s respectively for data of size 512 MB. We thus take these two bandwidth as the memory bounds as the upper-bound of the memory in our roofline model.

The intersection of the peak performance bound and the memory bandwidth bound is at $I = \pi/\beta$. For scalar code, the intersection is between 0.64 and 4.38 flop/byte. As for vectorized code, the intersection is between 2.56 and 17.55 flop/byte.

We use Linux `perf` command [**?**] to monitor hardware event such as cache miss and number of load/store invocation. The number of last-level cache (LLC) miss can give us an idea of the data traffic between CPU and main memory.

The compilers we use for all the statistics in this paper are Intel C/C++ Compiler (*icc 14.0.0.20131008*) For the baseline, the compiler flag is *-O3 -ansi-alias -no-vec -unroll=0*. As for the our best optimized version, we set the flag to *-O3 -std=c++11 -xHost -ansi-alias -unroll=4 -march=core-avx2 -auto-ilp32*.

4.2. Results

Figure 4 shows the performance gain from all the optimization techniques we adopt. With proper unrolling and vectorize compiler flags, the performance is improved by 3.7 times, from 0.57 flops/cycle to 2.12 flops/cycle. With all the other approaches mentioned in Section 3 that improves data locality and reduce computation, we further enhanced the performance to 4.10 flops/cycle, which gives us 7.2 times speed-up in total comparing to the baseline version. The run time proportion after optimization is shown in Figure 3

As the input size increases, the overhead of optimization, such as extra branches for corner case of blocking and unrolling, become insignificant as the benefit grows faster. On the other hand, the performance and the speed-up gain drop dramatically on input sizes larger than 1024. One reason might be that for input size of 1024, the allocated memory space exceed the 8MB physical memory space. Then the run time is dominated by the access time of hard disk.

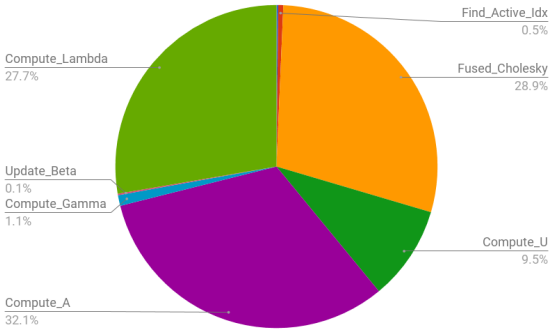


Fig. 3: Run time after optimization

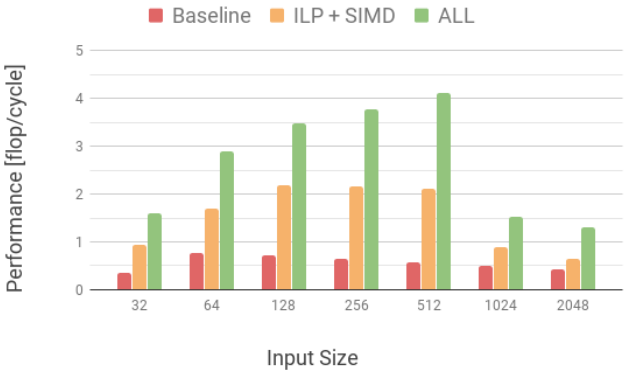


Fig. 4: Performance Comparison. BASELINE is compiled w/o optimization flags, ILP+SIMD is baselin compiled with optimization falgs and ALL is our final optimized version.

The memory traffic is estimated as:

$$\#LLC\ misses \cdot LLC\ line\ size$$

As argued by Georg Ofenbeck et al.[**?**] the actual traffic might be more than, even twice, the amount of observed LLC cache misses, depending on caching mechanism. Even though, it's still good upper bound of of memory traffic. Combining the measured memory traffic, measured run time cycles, and calculated floating point operation counts, we draw the roofline plot and examine the limitation of optimizing this algorithm.

From the roofline plot in Figure 6, we know that the larger the input size is (larger than 256), the more likely the

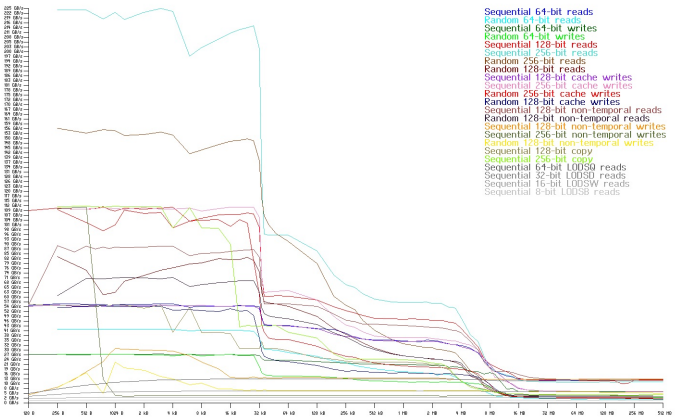


Fig. 5: Measured memory bandwidth

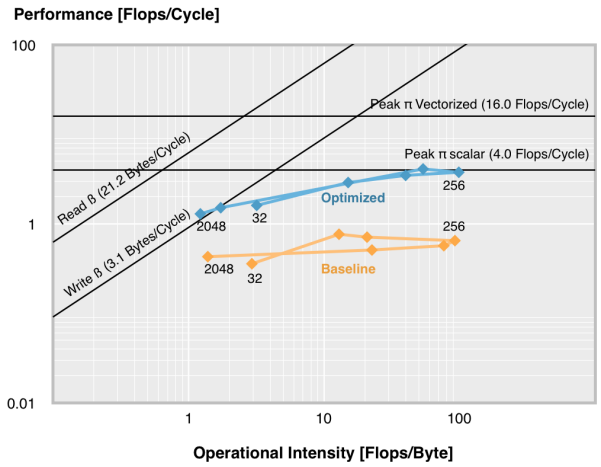


Fig. 6: Roofline plot of baseline and optimized program.

algorithm is memory bounded.

5. CONCLUSIONS

In this paper, we proposed several optimization strategies that can be applied to the LARS algorithm and give a summary on their performance. Besides optimization that can also be done by the compilers, we also proposed specific optimization according to its mathematical implication and data access pattern. We achieved an overall $7.2\times$ speed-up, reaching 4.10 flops/cycle, in performance against the baseline implementation without optimization flags (0.57 flops/cycle) and a $2.0\times$ speed-up from a compiler optimized version (2.12 flops/cycle).