EFFICIENT BATCHED CPU/GPU IMPLEMENTATION OF ORTHOGONAL MATCHING PURSUIT FOR PYTHON

Ariel Lubonja

Johns Hopkins University
Dept. of Applied Mathematics and Statistics
Baltimore, MD, USA

Sebastian Præsius

Technical University of Denmark
DTU Compute
Kongens Lyngby, Denmark

ABSTRACT

Finding the most sparse solution to the underdetermined system $\mathbf{y} = \mathbf{A}\mathbf{x}$, given a tolerance, is known to be NP-hard. A popular way to approximate a sparse solution is by using Greedy Pursuit algorithms, and Orthogonal Matching Pursuit (OMP) is one of the most widely used such solutions. For this paper, we implemented an efficient implementation of OMP that leverages Cholesky inverse properties as well as the power of Graphics Processing Units (GPUs) to deliver up to 200x speedup over the OMP implementation found in Scikit-Learn.

Index Terms— Compressed Sensing, Sparse Recovery, Orthogonal Matching Pursuit, Graphics Processing Unit

1. INTRODUCTION

Exploitation of the sparsity of a signal is key to reconstruction from significantly fewer samples than the Nyquist Rate [1]. A signal vector $\mathbf{x}$ is said to be S-sparse if it includes at most S non-zero entries. The signal x is recovered by solving

$$\boldsymbol{y} = \mathbf{A}\boldsymbol{x} + \boldsymbol{\varepsilon}$$

where $\boldsymbol{y}$ is a $M \times 1$ measurement vector, $\mathbf{A}$ is a wide dictionary matrix, i.e. $M \times N$ where $M \ll N$, and $\boldsymbol{x}$ is the sparse recovered signal with S non-zero elements. $\boldsymbol{\varepsilon}$ is the error in our reconstruction. Many variations of OMP have been proposed, but the simplest implementation remains widely used because of its understandability and ease of implementation.

There are three main ways to reduce the complexity of the OMP algorithm: through Cholesky factorization, the Matrix Inversion Lemma, or the QR factorization and our implementation builds on work from the Efficient implementations for Orthogonal Matching Pursuit paper [2].

In an effort to keep our implementation as widely applicable as possible, we made the no assumptions on the properties of the $\mathbf{A}$ dictionary. Considerable research has gone into leveraging properties of the certain specific dictionary matrices [3, 4].

This paper is organized as follows: Section 2 describes the algorithms. Section 3 describes our most important findings while implementing them. Finally, section 4 provides benchmarks and comparisons with other implementations.

```
Algorithm 1: Orthogonal Matching Pursuit
input  : $\mathbf{A} \in \mathbb{R}^{M \times N}$ dictionary
         $\mathbf{y} \in \mathbb{R}^{M}$: measurement vector
         $S$: sparsity level
         $\varepsilon$ target error (optional)
output: $\hat{\mathbf{x}}$ signal reconstruction
auxiliary variables: $\mathbf{r_i} \in \mathbb{R}^{M}$-residual at $k$-th
  iteration, $\mathcal{S}_0$-initial support set
initialization: $\hat{\mathbf{x}}_0 = \mathbf{0}$, $\mathbf{r}_0 = \mathbf{y}$
for $k = 1 : S$ do
    $n^* = \arg\max_{1 \leq n \leq N} \frac{|\langle \mathbf{r}_{k-1}, \mathbf{a}_n \rangle|}{\|\mathbf{a}_n\|}$
    $\mathcal{S}_k = \mathcal{S}_{k-1} \cup n^*$
    $\hat{\mathbf{x}}_k = \arg\min_{\mathbf{x}} \|\mathbf{y} - \mathbf{A}_{\mathcal{S}_k}\mathbf{x}\| = \left(\mathbf{A}_{\mathcal{S}_k}^\top \mathbf{A}_{\mathcal{S}_k}\right)^{-1} \mathbf{A}_{\mathcal{S}_k}^\top \mathbf{y}$
    $\mathbf{r}_k = \mathbf{y} - \mathbf{A}_{\mathcal{S}_k}\hat{\boldsymbol{x}}_k$
end
Or stop when $\|\mathbf{y} - \mathbf{A}_{\mathcal{S}_k}\hat{\boldsymbol{x}}_k\| \leq \varepsilon$
```

2. IMPLEMENTATIONS OF OMP

OMP achieves good recovery performance, and if the dictionary $\mathbf{A}$ satisfies the Restricted Isometry Property (RIP), exact recovery is guaranteed [5].

Orthogonal Matching Pursuit is a greedy algorithm, which tries to solve the following problem:

$$\underset{\boldsymbol{x} \in \mathbb{R}^N}{\operatorname{argmin}} \left\| \mathbf{A}\boldsymbol{x} - \boldsymbol{y} \right\| \quad \text{s.t.} \quad |\operatorname{supp} \boldsymbol{x}| \leq S$$

Given the regression matrix $\mathbf{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n]$ and sparsity level S, it tries to find the S-sparse $\boldsymbol{x}$, which gave rise to the observation $\boldsymbol{y}$ by $\mathbf{A}\boldsymbol{x} = \boldsymbol{y}$. It is a recursive algorithm, which greedily at every step picks the column of $\mathbf{A}$ which correlates mostly with the current residual. (Essentially a depth first search of the power set) We can see how the residual $\boldsymbol{r}_k$ is a function of the current regression subset $\mathcal{S}_k$, and there are $\frac{N!}{S!(N-S)!}$ possibilities for $\mathcal{S}_S$, By using a greedy approach rather than an exhaustive search, this is reduced to just S steps, but it does not necessarily converge to the global optimum.

We will use a *dense representation* of $\mathbf{A}_{\mathcal{S}_k}$ (denoted $\mathbf{A}_k$), and we will use the *exact solution* to $\mathbf{A}_k^\top \mathbf{A}_k \hat{\boldsymbol{x}} = \mathbf{A}_k^\top \boldsymbol{y}$ in our steps. Using a dense representation gives us fast linear equation solving and matrix-multiplication, since the variables are stored in contiguous memory, and these operations are the main bottlenecks for OMP.

2.1. Our “Naïve” Algorithm

We should stress that this ”naïve” algorithm is quite heavily optimized in terms of memory layout and other engineering aspects; more details available in Section 3. It is conceptually identical to the OMP algorithm described in algorithm 1, hence the term ”naïve”. It iterates as follows:

$$\mathbf{A}_k := \begin{bmatrix} \mathbf{A}_{k-1} & \boldsymbol{a}_{n^*} \end{bmatrix} \text{ by appending } \boldsymbol{a}_{n^*} \in \mathbb{R}^M \tag{1}$$

$$\mathbf{A}_k^\top \boldsymbol{y} := \begin{bmatrix} \mathbf{A}_{k-1}^\top \boldsymbol{y} & \boldsymbol{a}_{n^*}^\top \boldsymbol{y} \end{bmatrix} \text{ by appending } \boldsymbol{a}_{n^*}^\top \boldsymbol{y} \in \mathbb{R} \tag{2}$$

$$\begin{gathered} \mathbf{A}_k^\top \mathbf{A}_k := \begin{bmatrix} \mathbf{A}_{k-1}^\top \mathbf{A}_{k-1} & \mathbf{A}_{k-1}^\top \boldsymbol{a}_{n^*} \\ \cdots & \boldsymbol{a}_{n^*}^\top \boldsymbol{a}_{n^*} \end{bmatrix} \\ \text{by ”appending” } \mathbf{A}_k^\top \boldsymbol{a}_{n^*} = \begin{bmatrix} \mathbf{A}_{k-1}^\top \boldsymbol{a}_{n^*} \\ \boldsymbol{a}_{n^*}^\top \boldsymbol{a}_{n^*} \end{bmatrix} \in \mathbb{R}^k \end{gathered} \tag{3}$$

As such, $\mathbf{A}_k$ is a dense representation of $\mathbf{A}_{\mathcal{S}_k}$. We omit the lower triangle in $\mathbf{A}_k^\top \mathbf{A}_k$ since it is symmetric.

One can see that this reformulation of OMP allows us to iterate using fewer computations than if we compute all of $\mathbf{A}_{\mathcal{S}_k}^\top \boldsymbol{y}$ and $\mathbf{A}_{\mathcal{S}_k}^\top \mathbf{A}_{\mathcal{S}_k}$ each iteration; there is likely no way to get around having to use at least $\mathcal{O}(S^2)$ memory for the two inputs to the linear equation solvers, but one can store $\mathbf{A}_k$ in-place inside $\mathbf{A}$ by swapping columns, so the total memory comes out at $MN + \mathcal{O}(S^2) = \mathcal{O}(MN)$, since $S \leq M \leq N$. We use a Cholesky factorization of $\mathbf{A}_k^\top \mathbf{A}_k$ to then solve $\mathbf{A}_k^\top \mathbf{A}_k \hat{\boldsymbol{x}} = \mathbf{A}_k^\top \boldsymbol{y}$, and finally get $\boldsymbol{r}_k = \boldsymbol{y} - \mathbf{A}_k \hat{\boldsymbol{x}}$. [1]

The approach used in **Scikit-Learn is similar**. They pre-compute $\mathbf{A}^\top \boldsymbol{y} \in \mathbb{R}^N$, and optionally also $\mathbf{A}^\top \mathbf{A}$ (which we also allow for in our Naïve alg.), thereby trading memory for computational time. The values of $\mathbf{A}_k^\top \boldsymbol{y}$ are stored in-place inside $\mathbf{A}^\top \boldsymbol{y}$ by swapping (i.e. reordering). The main difference is that the Scikit-Learn approach directly stores and progressively updates the Cholesky factorization $\mathbf{V}_k \in \mathbb{R}^{k \times k}$ of $\mathbf{A}_k^\top \mathbf{A}_k$, instead of storing $\mathbf{A}_k^\top \mathbf{A}_k$, which is potentially faster. Iterates as:

$$\mathbf{V}_1 := \sqrt{\boldsymbol{a}_{n^*}^\top \boldsymbol{a}_{n^*}} = \|\boldsymbol{a}_{n^*}\| \tag{4}$$

$$\mathbf{V}_k := \begin{bmatrix} \mathbf{V}_{k-1} & \mathbf{0} \\ \boldsymbol{z}^\top & \sqrt{\|\boldsymbol{a}_{n^*}\|^2 - \|\boldsymbol{z}\|^2} \end{bmatrix} \tag{5}$$

Where $\boldsymbol{z}$ is the solution to $\mathbf{V}_{k-1}\boldsymbol{z} = \mathbf{A}_{k-1}^\top \boldsymbol{a}_{n^*}$.

It can be seen that see this is the Cholesky-factorization of $\mathbf{A}_k^\top \mathbf{A}_k$ by the fact that $\mathbf{V}_1 \mathbf{V}_1^\top = \|\boldsymbol{a}_{n^*}\|^2 = \mathbf{A}_1^\top \mathbf{A}_1$ and:

$$\begin{aligned} \mathbf{V}_k \mathbf{V}_k^\top &= \begin{bmatrix} \mathbf{V}_{k-1}\mathbf{V}_{k-1}^\top & \mathbf{V}_{k-1}\boldsymbol{z} \\ \cdots & \|\boldsymbol{a}_{n^*}\|^2 - \|\boldsymbol{z}\|^2 + \boldsymbol{z}^\top \boldsymbol{z} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{A}_{k-1}^\top \mathbf{A}_{k-1} & \mathbf{A}_{k-1}^\top \boldsymbol{a}_{n^*} \\ \cdots & \boldsymbol{a}_{n^*}^\top \boldsymbol{a}_{n^*} \end{bmatrix} = \mathbf{A}_k^\top \mathbf{A}_k \end{aligned} \tag{6}$$

Re-factorizing the way we do in our naive algorithm takes $\mathcal{O}(k^3)$ time. Cholesky-updating involves solving triangular systems of equations, taking just $\mathcal{O}(k^2)$ time.

Our implementations are adapted for batch processing. Precomputing $\mathbf{A}^\top \boldsymbol{y}$ is not efficient in terms of space/time, as we have B of the $\boldsymbol{y}$'s, so memory will be increased by $\mathcal{O}(BN)$, while doing so only saving around 1% of the running time in practice ($\boldsymbol{a}_{n^*}^\top \boldsymbol{y}$ in eq. (2) is just a simple dot product.) Similarly, reordering $\mathbf{A}$ is not an applicable memory-optimization as $\mathbf{A}_k$ is different for each batch element. But precomputing and using $\mathbf{A}^\top \mathbf{A}$ is possible and can save up to 15% of the running time (in our experience), while spending $\mathcal{O}(N^2)$ memory. It is used in eq. (3) where $\mathbf{A}^\top \boldsymbol{a}_{n^*} = [\mathbf{A}^\top \mathbf{A}]_{n^*}$.

2.2. Algorithm v0

The **algorithm v0** from [2] uses an inverse Cholesky factorization scheme. It uses precomputed $\mathbf{A}^\top \boldsymbol{y}$ and $\mathbf{A}^\top \mathbf{A}$, and is essentially based on the update formulae:

$$\mathbf{F}_1 := 1/\|\boldsymbol{a}_{n^*}\| \tag{7}$$

$$\mathbf{F}_k := \begin{bmatrix} \mathbf{F}_{k-1} & -\gamma \mathbf{F}_{k-1}\boldsymbol{z} \\ \mathbf{0} & \gamma \end{bmatrix} \tag{8}$$

$$\text{Where } \boldsymbol{z} = \mathbf{F}_{k-1}^\top \mathbf{A}_{k-1}^\top \boldsymbol{a}_{n^*} \text{ and } \gamma = \frac{1}{\sqrt{\|\boldsymbol{a}_{n^*}\|^2 - \|\boldsymbol{z}\|^2}}$$

One can see that this is the first factor in the Cholesky factorization of $(\mathbf{A}_k^\top \mathbf{A}_k)^{-1} = (\mathbf{V}_k \mathbf{V}_k^\top)^{-1} = \mathbf{V}_k^{-\top} \mathbf{V}_k^{-1}$ by

$$\begin{aligned} \mathbf{F}_k \mathbf{V}_k^\top &= \begin{bmatrix} \mathbf{F}_{k-1} & -\gamma \mathbf{F}_{k-1}\boldsymbol{z} \\ \mathbf{0} & \gamma \end{bmatrix} \begin{bmatrix} \mathbf{V}_{k-1}^\top & \boldsymbol{z}^\top \\ \mathbf{0} & 1/\gamma \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{F}_{k-1}\mathbf{V}_{k-1}^\top & 0 \\ \mathbf{0} & 1 \end{bmatrix} = \mathbf{I} \Rightarrow \mathbf{F}_k = \mathbf{V}_k^{-\top} \end{aligned} \tag{9}$$

[1] Using QR-factorization is also possible, but Cholesky is about twice as fast. (Half FLOPS) And it requires a positive definite $\mathbf{A}_k^\top \mathbf{A}_k$.

$$\boldsymbol{z} = \mathbf{F}_{k-1}^{\top}\mathbf{A}_{k-1}^{\top}\boldsymbol{a}_{n^*} \Rightarrow \mathbf{V}_{k-1}\boldsymbol{z} = \mathbf{A}_{k-1}^{\top}\boldsymbol{a}_{n^*} \tag{10}$$

Which follows by induction from the base $\mathbf{F}_1\mathbf{V}_1^{\top} = 1$. (Not a complete proof.)

Using only matrix-vector products one gets:

$$\begin{aligned}\hat{\boldsymbol{y}} &= \mathbf{A}_k\hat{\boldsymbol{x}}\\ &= \mathbf{A}_k\left(\mathbf{A}_k^{\top}\mathbf{A}_k\right)^{-1}\mathbf{A}_k^{\top}\boldsymbol{y}\\ &= \mathbf{A}_k\mathbf{F}_k\mathbf{F}_k^{\top}\mathbf{A}_k^{\top}\boldsymbol{y} = \mathbf{A}_k\mathbf{F}_k(\mathbf{A}_k\mathbf{F}_k)^{\top}\boldsymbol{y}\end{aligned} \tag{11}$$

And this is especially desirable in parallel compute environments compared to solving triangular systems since less synchronization is needed. It turns out one does not need to store the inverse Cholesky $\mathbf{F}_k$ to iterate. This is because all we need to perform an iteration are the new projections, and the algorithm v0 instead directly updates these, while filling out the matrix $\mathbf{D}_k = \mathbf{A}^{\top}\mathbf{A}_k\mathbf{F}_k$ (and $\mathbf{F}_k$ if this is needed to find $\hat{\boldsymbol{x}}$).

This is done by a single matrix-vector multiplication per iteration in $\mathcal{O}(Nk)$ time, meaning the batched implementation running time is $\mathcal{O}(N^2 + BNM + BNS^2)$ for the Gramian, initial projections and per-iteration costs.

The update steps are notationally a little involved, so we refer to the original paper [2]. The paper also contains less memory-consuming (but slower) versions, which can be useful since the asymptotic memory use is $\mathcal{O}(NM + N^2 + BNS)$ (inputs, Gramian and $\mathbf{D}_S \in \mathbb{R}^{S\times N}$), compared to just $\mathcal{O}(NM + B(N + MS))$ for the previously discussed algorithms. Especially on a GPU the difference between storing $N^2 + BNS$ compared to BMS floating point numbers can be significant.

In the interest of time we focused only on implementing and optimizing the two algorithms naive and v0. With most focus on CPU-optimizations for the naive, and GPU-optimizations for v0.

3. IMPLEMENTATION DETAILS AND ENGINEERING TRICKS

Most matrix operations done in Python's Numpy call the underlying linear algebra libraries BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package). The same is true for MATLAB, and the reason is that these libraries are highly optimized. For example Intel's MKL, contains processor-specialized versions of these libraries that are optimized to use the full instruction set of the CPU and exploit it's cache as much as possible, often achieving close to the theoretically possible number of FLOPS (first limited by clock speed, then by memory throughput for problems that do not fit in cache). Which brings us to the first main takeaway.

3.1. Memory layout

Ensure data is contiguous as much as possible. Since BLAS handles the matrix-multiplication in a highly optimized way (and this is the main bottleneck), we can leverage our control of the memory-layout of the inputs to get more speed. A useful programming pattern we found was to separate memory layout from it's use:

```
As = np.zeros([B, M, S])
As = np.zeros([B, S, M]).transpose([0, 2, 1])
```

If we look at the two definitions of As above, they both have the same shape $B \times M \times S$ and can be used interchangeably with another, but it will be faster to write a column into the second one as columns are stored contiguously - One column spans a single line in memory.

By separating layout from use one can quickly find which layout is the best-performing, even in cases where it is not obvious, or trade-offs must be made.

3.2. Matrix batched-matrix products

A less known fact is that you can sometimes get a higher performance in a matrix times batched-matrix product (in this case batched vectors $\boldsymbol{r}$), by exploiting:

$$\begin{bmatrix}\mathbf{A}^{\top}\boldsymbol{r}^1 & \cdots & \mathbf{A}^{\top}\boldsymbol{r}^B\end{bmatrix} = \mathbf{A}^{\top}\begin{bmatrix}\boldsymbol{r}^1 & \cdots & \boldsymbol{r}^B\end{bmatrix} \tag{12}$$

Notice how this matrix batched-vector product is equivalent to a simple matrix-matrix product, which can be performed through *a single* call to gemm, giving BLAS full control of producing the result we need in the most efficient manner. This is done through transposing and reshaping, which should be noted only changes the meta-data for the tensors. In our experience doing this gives a 2-8 times speed-up. See batch_mm in the code.

We assume that $\mathbf{A}$ has normalized columns (if not, give normalize=True to run_omp), such that (appendix A): $\langle \boldsymbol{a}_n, \boldsymbol{r}_{k-1}\rangle / \|\boldsymbol{a}_n\| = \boldsymbol{a}_n^{\top}\boldsymbol{r}_{k-1} = [\mathbf{A}^{\top}\boldsymbol{r}_{k-1}]_n$.

3.3. Packed representation

For our naive algorithm we found it useful to store $\mathbf{A}_k^{\top}\mathbf{A}_k$ in a *packed representation* and then use the BLAS functions ppsv instead of posv (positive-definite solver), as this significantly cut down on the time to write and especially fetch a sub-matrix of $\mathbf{A}_S^{\top}\mathbf{A}_S$, since then any sub-matrix is contiguous in memory.

This means that instead of storing the $k\times k$ submatrix in a strided manner inside a memory block which which is $B \times S \times S$, we only store the $k(k+1)/2 = 1 + 2 + \cdots + k$ triangle of this. Then all operations becomes contiguous!

This optimization, and the one in section 3.2 are mostly relevant for CPU. For GPU there already exists specialized calls for batched matrix-matrix multiplication, batched Cholesky factorization and batched triangular system solving. There is no ppsv equivalent in cuBLAS/cuSOLVER for CUDA (the GPU Parallel Computing platform available on NVIDIA graphics cards). We did not experiment with calling CUDA manually, but used the PyTorch library for this.

3.4. Efficient batched argmax

A core part of the OMP loop is argmax, which can be performed on a batch by:

```
# (B, N) -> (B,)
n_star = abs(projections).argmax(1)
```

One issue is that abs creates an intermediate $B \times N$ array in the first pass, and then a second pass over this is needed to get the argmax. For CPU we ended up using the BLAS function i_amax, which finds the index of the absolute maximum value in an array.

First we tried running a loop over the B batch elements using the scipy.linalg.blas wrapper, but for small problems the Python overhead is so significant that we do not get much benefit. Therefore we switched to implementing this loop in Cython, which is a superset of Python that compiles directly to C++. Through this we call i_amax and ppsv in a loop, which gave a major speed-up: Around 5 times for the argmax compared to 1-2 times if using a loop in Python!

It can be hard to get the arguments for these calls right since they take fortran-style arguments (was originally written in FORTRAN) See argmax_blast in code.

Direct calls to cuBLAS on GPU for e.g. i_amax could potentially give a speed-up. There are only around 3 batched kernels in cuBLAS, and this is not one of them, so we would have to launch the kernels in a loop; but then we get an overhead just from the simple transfer of maybe thousands of kernel calls and arguments to the GPU! It would probably be worthwhile for small B.

The argmax line takes 5-25% of the GPU computation time in our experience, so next step may be to implement a custom reduction kernel for this, which is not trivial if it must be efficient due to the specific hardware architecture. We would recommend CuPy for this.

3.5. Batched stopping criteria

We did not spend too long on this part since it is not essential, so there is great space for improvement here.

For the naive algorithm we implement early stopping by keeping an active set of batch elements, along with their individual data – and then we remove all their data when they are done, such that we are left with a block of $B-1$ elements (and their data).

There is a slight overhead of having to move all this memory, but in our experience it is outweighed by the improvement in speed on subsequent iterations, which then became faster as there are fewer batch elements.

For v0 however, due to the large size of $\mathbf{D}_S \in \mathbb{R}^{S \times N}$ even just allocating it with new_zeros takes significant time, which is why we use new_empty to get *uninitialized* memory for it. This also means that the naive batching strategy of moving all this memory is not efficient (especially on GPU, where parallel threads are used to move every single byte).

For v0 we opted to just save the result when the stopping criteria was met, but still keep all the data inside the batch. The two approaches we chose mean that the run-time is relatively unchanged from before. But it could be probably be done more efficiently.

3.6. Other possible optimizations

One possible optimization is *custom functions / kernels*. This can cut down somewhat on intermediate results, and improve memory locality. Since BLAS operates as an efficient black box, it can be hard to e.g. *fuse* the matrix multiplication with the abs argmax, but in v0 we have to scale a lot different of vectors by γ, which could possibly be faster if it was done with a single kernel. See e.g. Appendix C for a line-by-line profiling.

One can use *streams* in CUDA to launch kernels in parallel. There are only few places where we could potentially get a speed-up from this due to the fact that results on previous lines are frequently needed on the subsequent ones. For example eq. (2) and eq. (3) are independent, but the first takes almost no time compared to the latter. Algorithm v0 could possibly get a speed-up of around 5% with this, since calculating the inverse Cholesky is independent from the iterations. The whole reason for batching is so that we can do multiple of these non-parallelizable problems in parallel.

For the naive algorithm the next step may be to switch to a Cholesky-update scheme. This approach is likely strictly faster than re-factorizing every iteration.

One can give half-precision tensors to our implementation. While this does not seem to give a speed-up, it will reduce the memory requirements. Especially in large problems on GPU, memory can become an issue when using v0 (meaning one has to use a lower batch size, possibly reducing efficiency). One could look into the memory-saving variants. Using double precision reduces speed by half on GPU.

It will be simple to modify the v0 code to have multiple different design matrices along with the corresponding $\boldsymbol{y}$'s, but we thought it was reasonable to assume they are the same. (As Scikit Learn also does)

4. BENCHMARKS

The interface for our code has the same functionality as that from Scikit-Learn, except that the y is batched in the first dimension and not the second.

In fig. 1, the benchmarks are run with $\mathbf{A} \in \mathbb{R}^{8M\times M}$, $y = \mathbb{R}^{100\times M}$ $(B = 100)$, and $S = M/4$, where M is the variable shown in the x-axis, ranging from 16 to 2048. Larger values of M could not be attempted due to memory exhaustion on GPU for $B = 100$. In this case the GPU version is orders of magnitude faster than CPU for large problems; this is also partly because we are using a low batch size, so the GPU is not fully utilized for small M. $(B = 100000$ is more appropriate for $M \le 64)$.

Generally v0 will outperform the naive algorithm when S is large. And the GPU will almost always outperform the CPU provided the batch size is large enough, except maybe in case of tiny problems and a multi-core CPU.

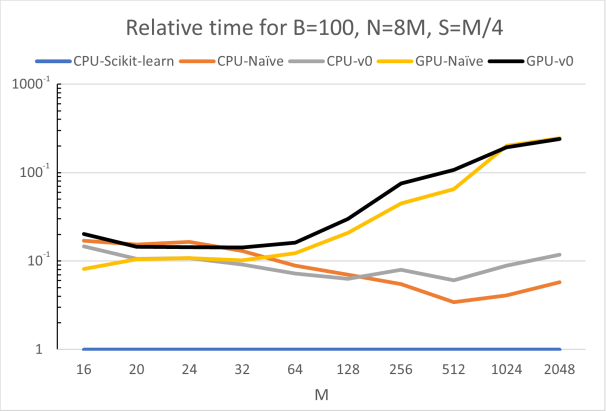

Fig. 1. Relative time for running OMP.

The raw data for this plot is in appendix B.

We also inspected the numerical error of the algorithms, but they are all around the same.

4.1. Yale Face Classification (HW7)

In this perhaps more relatable benchmark, we use our algorithms to classify the 1207 test images from the Yale dataset in Homework 7. To make the problem more challenging, the dictionary $\mathbf{A}$ contains the entire set of training images, not downsampled, therefore, $\mathbf{A} \in \mathbb{R}^{8064\times 1207} \simeq \mathbb{R}^{96x84x1207}$. All images from the test set have been batched into a matrix $\mathbf{y}$, with dimensions $\mathbb{R}^{1207\times 8064}$. We ran the problem in the algorithms described above, as well as our simple sequential implementation of OMP in Matlab from Homework 5. Sparsity level was set to $S = 30$.

HW5 OMP (Matlab)	Sklearn	Naïve CPU	V0 CPU	Naïve GPU	V0 GPU
496.2	1443	13.09	1.214	0.609	0.111

Table 1. Solving time [s] for 1207 96x84px test images.

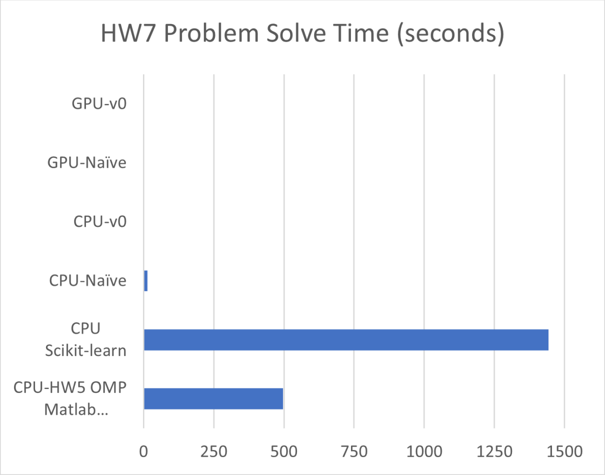

Fig. 2. Bar plot emphasizing the order-of-magniture difference in performance for Homework 7

5. CODE

We've set up a demo notebook with all the required code on Colab, and our code is available at https://github.com/Ariel5/omp-parallel-gpu-python. If you encounter any difficulties, please do not hesitate to email us at alubonj1@jhu.edu and sebastian.devel@gmail.com or s164198@student.dtu.dk!

6. CONCLUSION

We implemented a "naive" version of OMP, and a newer algorithm called v0. Both of these outperform the commonly used Scikit-Learn implementation by a large margin, while having the same functionality.

With a GPU we could get a speed-up of several orders of magnitude: 200x faster than Scikit-Learn. But there is still room for improvement.

7. APPENDIX

A. NORMALIZED COLUMNS OF DICTIONARY

The main approaches to speeding up the algorithm is to minimize the number of operations to perform each iteration. Many algorithms assume normalized columns in $\mathbf{A}$ such that correlation $\langle \boldsymbol{a}_n, \boldsymbol{r}_{k-1} \rangle / \|\boldsymbol{a}_n\|$ turns into a simple projection $\langle \boldsymbol{a}_n, \boldsymbol{r}_{k-1} \rangle = \boldsymbol{a}_n^\top \boldsymbol{r}_{k-1} = [\mathbf{A}^\top \boldsymbol{r}_{k-1}]_n$ – this is valid since the algorithm is invariant to column norm, as it will be divided out in the correlation step, and lastly, the least-squares estimate $\hat{\boldsymbol{y}}_k := \mathbf{A}_{\mathcal{S}_k} \mathbf{A}_{\mathcal{S}_k}^+ \boldsymbol{y} = \mathbf{A}_{\mathcal{S}_k} \operatorname{argmin}_{\boldsymbol{x} \in \mathbb{R}^k} \left\| \boldsymbol{y} - \mathbf{A}_{\mathcal{S}_k} \boldsymbol{x} \right\|$ is unique, thus also invariant to column scaling. For the final estimate $\hat{\boldsymbol{x}}$ from $\mathbf{A}_S^\top \mathbf{A}_S \hat{\boldsymbol{x}} = \mathbf{A}_S^\top \boldsymbol{y}$ one should then not use the pre-normalized $\mathbf{A}$, or at least scale $\hat{\boldsymbol{x}}$ appropriately (by the reciprocal of column the norm) to account for this.

B. BENCHMARK DATA

Example of an image used for benchmarking (HW7):

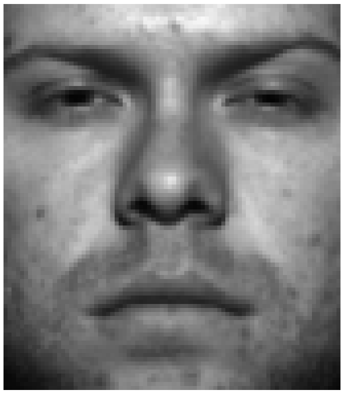

Fig. 3. Example 96*x*84 image used for benchmarking (HW7)

CPU-Scikit-learn	CPU-Naïve	CPU-v0	GPU-Naïve	GPU-v0
0.038	0.002	0.003	0.005	0.002
0.040	0.003	0.004	0.004	0.003
0.047	0.003	0.004	0.004	0.003
0.062	0.005	0.007	0.006	0.004
0.142	0.016	0.020	0.012	0.009
0.489	0.070	0.078	0.023	0.016
2.393	0.436	0.300	0.053	0.032
11.173	3.265	1.838	0.172	0.104
109.597	26.796	12.326	0.546	0.567
1077.419	186.313	91.159	4.392	4.475

Table 2. The timing results [seconds] for varying M.

C. GPU PER-LINE PERFORMANCE

For the implementation of the naive algorithm on GPU, with a large problem instance, the per-line time is:

```
Line #    % Time  Line Contents
==============================
29  def run_omp(precompute=True, alg='naive'):
55  11.0    precompute = X.T @ X
59  89.0    omp_naive(..., XTX = precompute)

115 def omp_naive(...):
174 26.4    projections = XT @ r[:, :, None]
175 1.2     sets[:, k] = projections.abs().argmax(-1)
208 65.5    solutions = cholesky_solve(ATA, ATy)
214 3.9     torch.baddbmm(...) # update r
```

With around 2% of time spent on transferring to/from GPU. And if we take a tiny problem instance (but huge batch size), the transfer will instead be around 14% of the time spent. Other differences: Line 174 takes then 8.5%, but line 175 takes 21.1%. Also line 208 takes only 40.6%, but line 214 then takes 18.6%. The rest is spread out on many different places.

And the break-down for v0 on GPU with tiny problem instances:

```
Line #   % Time  Line Contents
===============================
29  def run_omp(..., alg='v0'):
61  99.6    ... = omp_v0(...)

218  def omp_v0(...):
258 6.2    sets[k] = projections.abs().argmax(1)
266 8.1    D_mybest[:, k] *= temp_F_k_k
267 65.8   D_mybest[:, k, :, None].baddbmm_(...)
272 10.1   projections -= temp_a_F * D_mybest[:, k]
276 1.9    torch.bmm(..., out=F[:, k, None, :])
```

It can be seen how majority of time is spent on the matrix multiplications to update the D-matrix (D_mybest), and around 2% is spent on the F-matrix which is used to get $\hat{x}$. And for v0 on large problem instances (small batch size), the transfer time is 6.8%. A break-down:

```
Line #    % Time    Line Contents
=============================
29  def run_omp(..., alg='v0'):
55  45.2    precompute = X.T @ X
61  54.8    ... = omp_v0(...)

218  def omp_v0(...):
258 4.6     sets[k] = projections.abs().argmax(1)
260 2.5     torch.gather(...)  # Get from XTX
262 4.2     D_mybest_maxindices = ... # New D column
263 4.8     torch.rsqrt(1 - innerp(D_mybest_maxindices))
267 58.5    D_mybest[:, k, :, None].baddbmm_(...)
276 6.2     torch.bmm(..., out=F[:, k, None, :])
```

We see that the precomputation time is actually very significant (especially since v0 is comparably so fast). But this precompute time can be shared between batches, so subsequent batches can potentially execute almost twice as fast! Since line 267 is a cuBLAS bottleneck, we likely cannot get more than a 50-70% further speed-up.

D. REFERENCES

[1] Emmanuel J. Candes and Michael B. Wakin, "An introduction to compressive sampling," vol. 25, no. 2, pp. 21–30.

[2] Hufei Zhu, Wen Chen, and Yanpeng Wu, "Efficient implementations for orthogonal matching pursuit," vol. 9, no. 9.

[3] David Donoho, Yaakov Tsaig, Iddo Drori, and Jean-Luc Starck, "Sparse solution of underdetermined systems of linear equations by stagewise orthogonal matching pursuit," vol. 58, pp. 1094–1121.

[4] Boris Mailhe, Remi Gribonval, Frederic Bimbot, and Pierre Vandergheynst, "A low complexity orthogonal matching pursuit for sparse signal approximation with shift-invariant dictionaries," in *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 3445–3448.

[5] Emmanuel Candès, "The restricted isometry property and its implications for compressed sensing," vol. 346, pp. 589–592.