PARALLELIZATION OF FAST $\ell_1$-MINIMIZATION FOR FACE RECOGNITION

Andrew Wagner, Victor Shia, Allen Yang, Mark Murphy, and Yi Ma

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign
**Title:** Parallelization of Fast $\ell_1$-Minimization for Face Recognition  

**Authors:** Andrew Wagner, Victor Shia, Allen Yang, Mark Murphy, and Yi Ma  

**Abstract:** Recently a family of promising face recognition algorithms based on sparse representation and $\ell_1$-minimization ($\ell_1$-min) have been developed. These algorithms have not yet seen commercial application, largely due to higher computational cost compared to other traditional algorithms. This paper studies techniques for leveraging the massive parallelism available in GPU and CPU hardware to accelerate $\ell_1$-min based on augmented Lagrangian method (ALM) solvers. For very large problems, the GPU is faster due to higher memory bandwidth, while for problems that fit in the larger CPU L3 cache, the CPU is faster. On both architectures, the proposed implementations significantly outperform naive library-based implementations, as well as previous systems. The source code of the algorithms will be made available for peer evaluation.
Parallelization of Fast $\ell_1$-Minimization for Face Recognition

Andrew Wagner  
University of Illinois at Urbana-Champaign  
awagner@illinois.edu

Victor Shia  
University of California Berkeley  
vshia@eecs.berkeley.edu

Allen Yang  
University of Illinois at Urbana-Champaign  
yang@eecs.berkeley.edu

Mark Murphy  
University of California Berkeley  
mmurphy@eecs.berkeley.edu

Yi Ma  
University of Illinois at Urbana-Champaign  
yima@illinois.edu

Abstract

Recently a family of promising face recognition algorithms based on sparse representation and $\ell_1$-minimization ($\ell_1$-min) have been developed. These algorithms have not yet seen commercial application, largely due to higher computational cost compared to other traditional algorithms. This paper studies techniques for leveraging the massive parallelism available in GPU and CPU hardware to accelerate $\ell_1$-min based on augmented Lagrangian method (ALM) solvers. For very large problems, the GPU is faster due to higher memory bandwidth, while for problems that fit in the larger CPU L3 cache, the CPU is faster. On both architectures, the proposed implementations significantly outperform naive library-based implementations, as well as previous systems. The source code of the algorithms will be made available for peer evaluation.

1. Introduction

$\ell_1$-minimization ($\ell_1$-min) has received much attention in recent years due to important applications in compressive sensing [4] and sparse representation [21]. The problem refers to finding the minimum $\ell_1$-norm solution to an underdetermined linear system $b = Ax$:

$$\min \|x\|_1 \text{ subj. to } b = Ax. \tag{1}$$

It is now well established that, under certain conditions [5, 8], the minimum $\ell_1$-norm solution is also the sparsest solution to the system (1).

Among its many applications, $\ell_1$-min has been recently used to reformulate image-based face recognition as a sparse representation problem [22]. If we stack the $m$ pixels of the $n_i$ training images of $K$ subject classes into the columns of matrices $(A_1 \in \mathbb{R}^{m \times n_1}, \ldots, A_K \in \mathbb{R}^{m \times n_K})$, combine the matrices into a larger matrix $A = [A_1, \ldots, A_K] \in \mathbb{R}^{m \times n}$, and arrange the pixels of a new query image into a vector $b \in \mathbb{R}^m$, sparsity-based classification (SBC) solves the following minimization problem:

$$\min_{x,e} \|x\|_1 + \|e\|_1 \text{ subj to } b = Ax + e. \tag{2}$$

If the sparsest solutions for $x$ and $e$ are recovered, $e$ provides a means to compensate for pixels that are corrupted due to occlusion of some part of the query image, and the dominant nonzero coefficients of $x$ reveal the membership of $b$ based on the training image labels associated with $A$.

In this paper, we study parallelization of $\ell_1$-min on many-core CPU and GPU architectures. Although $\ell_1$-min (1) is a convex program, conventional algorithms such as interior-point methods [6, 18] scale poorly for applications such as face recognition. Recently, a number of accelerated algorithms have been proposed that explicitly take advantage of the special structure of $\ell_1$-min problems [13, 24]. We investigate parallelization of a state-of-the-art $\ell_1$-min solution based on augmented Lagrangian methods (ALM) [2, 24].

In addition, the accuracy of the solution given by (2) relies on the assumption that the query image is well aligned with the training images. In [19], SBC was extended to align a query image to each subject class individually. The alignment algorithm solves the following problem:

$$\hat{\tau}_i = \arg \min_{x,e,T} \|e\|_1 \text{ subj. to } b \circ \tau_i = A_i x + e, \tag{3}$$

where $\tau_i \in T$ is in a finite-dimensional group $T$ of transformations (e.g., affine or homography) acting on the image domain. The sequence $\tau_i$ converges to a transformation that aligns the test image $b$ with the training images from the $i$-th class $A_i$. In other words, the algorithm extends Lucas-Kanade iterative alignment [14] to use the $\ell_1$-norm as a robust error function, while simultaneously estimating the coefficients of the linear illumination model $A_i x$.

Although ALM has been applied to improve the speed of a prototype face recognition system [20], due to the high per-class cost of the alignment step, the system still fails to achieve real-time performance against datasets of hundreds or thousands of subjects. In this paper, in addition to accelerating the generic $\ell_1$-min objectives (1) and (2), we also discuss how to efficiently accelerate the face alignment step (3) on multi-core CPUs and GPUs.

To clarify discussion, we clearly distinguish between parallelism, which is a property of an algorithm, and concurrency, which is a property of a hardware architecture. Parallelism in an algorithm provides the opportunity to perform computations concurrently on hardware. Algorithms
often exhibit multiple levels of parallelism, and hardware often provides multiple levels of concurrency. In these terms, the primary contribution of this paper will be to determine the optimal mappings between the parallelism available in \( \ell_1 \)-min and the concurrency available in multicore CPU and GPU architectures.

Finally, we present extensive benchmarks to compare the performance of the generic ALM \( \ell_1 \)-min algorithm and the corresponding face recognition routines on massively parallel CPUs and GPUs. To this end, a face recognition system for security and access control applications has been implemented, and is diagrammed in Figure 1. This pipeline follows the work of [20], but the fast implementations presented in this paper are novel. Since the parallelization of any algorithm is highly dependent on the target hardware, we will give a brief overview of the architecture of our test system in detail in Section 3. Finally, note that while this paper focuses on the parallelism available in a single (shared memory) machine, applications targeting thousands of subjects or more will likely require computation to be distributed over a cluster or network of computers. These systems are not addressed here. The source code of our implementations, which target NVIDIA Fermi GPUs as well as Intel x86-64 CPUs, will be made available for peer evaluation.

1.1. Literature Review

Traditionally, \( \ell_1 \)-min (a.k.a. basis pursuit (BP)) has been formulated as a linear program [6]. Several variations of the solution are also well known in optimization, including a noisy approximation via quadratic programming called the LASSO [18] and truncated Newton interior-point method (TNIPM) [12].

One of the drawbacks of most interior-point methods for \( \ell_1 \)-min is that they require the solution sequence to follow an interior path via gradient descent or conjugate gradient methods, which are computationally expensive. To mitigate these issues, an approach called Homotopy has been recently studied to accelerate the speed of \( \ell_1 \)-min [17, 11, 15, 9]. Although Homotopy can be shown to exactly estimate BP when the solution is sufficiently sparse [9], the algorithm still involves computationally expensive operations such as matrix-matrix multiplication and linear least-squares problems with varying \( A \) matrices.

In Section 2, we contend that ALM is a better choice for implementation on many-core CPUs and GPUs. The ALM algorithm belongs to a category of approximate \( \ell_1 \)-min solutions called iterative shrinkage-thresholding (IST) methods [23, 1]. IST algorithms mainly involve elementary operations such as vector algebra and matrix-vector multiplication. Therefore, when the dimension of the problem becomes high, IST-type algorithms are particularly suitable for hardware systems with a high degree of concurrency. In [24], the authors showed that ALM is able to significantly improve the solver speed, while achieving estimation accuracy competitive with other \( \ell_1 \)-min solutions. Therefore, we choose ALM as the core algorithm for implementation of \( \ell_1 \)-min in the parallel face recognition pipeline.

In terms of the past works in parallel \( \ell_1 \)-min, the literature has been limited, to the best of our knowledge. In [3], Borghi et al. developed a special proximal gradient \( \ell_1 \)-min algorithm based on Moreau-Yosida regularization. In [16], Murphy et al. presented parallel implementation of the \( \ell_1 \)-SPIRIT MRI reconstruction algorithm on the same GPU architecture addressed in this paper.

2. Augmented Lagrangian Method

In this section, we briefly describe the ALM algorithm for \( \ell_1 \)-min (1) [24] and analyze its complexity. Lagrange multipliers have been frequently used in convex programming to eliminate equality constraints via adding a penalty term to the cost function for infeasible points. ALM methods differ from other penalty-based approaches by simultaneously estimating the optimal solution and Lagrange multipliers in an iterative fashion. For \( \ell_1 \)-min (1), the aug-
mented Lagrange function is defined as:

\[ L_\mu(x, y) = \|x\|_1 + y^T(b - Ax) + \frac{\mu}{2}\|b - Ax\|_2^2 \]

(4)

where \( \mu > 0 \) is a constant that penalizes infeasibility and \( y \) is a vector of Lagrange multipliers.

In Lagrange Multiplier Theory [2], if there exists a multiplier \( y^* \) that satisfies the second-order sufficiency conditions for optimality, then for a sufficiently large \( \mu \), the optimal \( \ell_1 \)-min solution also minimizes

\[ x^* = \arg \min L_\mu(x, y^*) \]

(5)

In practice, the optimal values for the triplet \((x^*, y^*, \mu)\) are all unknown. Furthermore, it has been observed that solving (5) with a large initial value of \( \mu \) tends to lead to slower convergence speed [23, 24]. In [2, 25], an alternating procedure has been shown to iteratively update \( x \) and \( y \):

\[
\begin{align*}
x_{k+1} &= \arg \min_x L_{\mu_k}(x, y_k) \\
y_{k+1} &= y_k + \mu_k(b - Ax_{k+1})
\end{align*}
\]

(6)

where \( \mu_k \to \infty \) increases monotonically. The iteration terminates when the estimates \((x_k, y_k)\) converge.

Note that in the iterative procedure (6), the second step only involves vector algebra and matrix-vector multiplication. Therefore, the procedure is computationally efficient only if it is easy to minimize the augmented Lagrangian \( L_{\mu_k}(x, y_k) \) compared to solving the original problem (1) directly. In fact, this problem can be solved element-wise iteratively by a soft-thresholding algorithm [23, 1], whose time complexity is bounded by \( O(n^2) \) and can be easily parallelized. Algorithm 1 summarizes the generic ALM \( \ell_1 \)-min algorithm. 1 The soft-thresholding algorithm involves the element-wise operator \( \text{shrink}(x) = \text{sign}(x) \max(|x|, 0) \).

Algorithm 1 Augmented Lagrangian Method (ALM)

**INPUT:** \( b \in \mathbb{R}^m \), \( A = [A_1, \cdots, A_K] \in \mathbb{R}^{m \times n} \), \( \tau \leftarrow \max \text{eig}(A^TA) \), and constant \( \rho > 1 \).

1. while not converged (\( k = 1, 2, \ldots \)) do
2. \( t_1 \leftarrow 1 \), \( z_1 \leftarrow x_k \), \( u_1 \leftarrow x_k \)
3. while not converged (\( l = 1, 2, \ldots \)) do
4. \( u_{l+1} \leftarrow \text{shrink}(z_l - \frac{1}{\mu_l}A^T(Az_l - b - \frac{1}{\mu_k}y_k) - \frac{1}{\mu_k}b) \)
5. \( t_{l+1} \leftarrow \frac{1}{\rho}(1 + \sqrt{1 + 4t_l^2}) \)
6. \( z_{l+1} \leftarrow u_{l+1} + \frac{t_{l+1}-1}{t_{l+1}}(u_{l+1} - u_l) \)
7. end while
8. \( x_{k+1} \leftarrow u_{l+1} \)
9. \( y_{k+1} \leftarrow y_k + \mu_k(b - Ax_{k+1}) \)
10. \( \mu_k \leftarrow \rho \cdot \mu_k \)
11. end while

**OUTPUT:** \( x^* \leftarrow x_k \).

1For conciseness, we only present the ALM algorithm in the primal domain. There also exist implementations in the dual domain [25, 24].

### 3. Hardware Concurrency

In this section we discuss the levels of concurrency available in the hardware architectures considered in this paper, as well as other aspects of the hardware that are important for performance. In particular, since caches (regions of on-chip memory) are often orders of magnitude faster than off-chip memory, their sizes and speeds have a dramatic effect on performance. We give a brief overview of the caches that are available in our target architectures, and defer discussion of their performance effects to Sections 4.1 and 4.2.

Our discussion and experiments will address the most common hardware configuration for engineering workstations: a motherboard with two quad-core processors and a PCIe card with a single high-end GPU. Recognition involving more than a few hundred subjects with contemporary hardware would require a server (or cluster) configuration with an expandable number of processors. We will not address efficient parallelization for these systems, which may have additional challenges associated with their non-shared memory model. Unless otherwise specified, all implementations utilize single precision floating point datatypes.

#### 3.1. CPU Hardware Concurrency

The main defining characteristics of contemporary multi-core CPU architectures are that they have two levels of concurrency, relatively large amounts of cache, and relatively high clock speeds. The baseline architecture for our experiments is a Linux workstation with two quad-core Intel Nehalem E5530 processors clocked at 2.4 GHz. Each processor has its own memory interface, and is directly interfaced to half of the RAM installed in the machine. The amount of RAM exceeds the amount used by the algorithms, and is not an important performance consideration.

Each core has a private 32 KiB L1 data cache and a private 256 KiB L2 cache. Each processor further has 8 MiB of L3 cache that is shared by the four cores. Overall, the algorithm has approximately 16 MiB of L3 cache available for a dual-processor configuration.

For floating-point instructions, each core also has a vector processing unit (VSE) capable of performing the same arithmetic operation on four single-precision floating point values simultaneously. There are thus two important levels of concurrency that need to be exploited to efficiently use a modern CPU: core-level concurrency and SSE-level concurrency.

#### 3.2. GPU Hardware Concurrency

The main defining characteristics of contemporary multi-core GPU architectures are that they have two (much wider) levels of concurrency, relatively small amounts of cache, and relatively low clock speeds. Whereas most of the transistors on a typical GPU are dedicated either to cache or to hardware that enables higher clock speeds
roughly analogous to a CPU core. For Fermi architecture capabilities. CUDA programmable GPUs are comprised of will instead frame our discussion in terms of hardware ca-
ness would take more space than is available here, so we programming model (CUDA) at a useful level of complete­
480 GPU used in this paper). An explanation of the GPU
(such as branch prediction, out-of-order instruction execu-
tion, etc...), most of the transistors on a GPU are dedicated
to arithmetic logic units. For our GPU implementations,
we target NVIDIA Fermi GPU architecture (e.g., the GTX
480 GPU used in this paper). An explanation of the GPU
programming model (CUDA) at a useful level of complete-
ness would take more space than is available here, so we
will instead frame our discussion in terms of hardware ca-
pabilities. CUDA programmable GPUs are comprised of several streaming multiprocessors (SM), each of which is
roughly analogous to a CPU core. For Fermi architecture
GPUs there are up to 16 SMs, and each SM is capable of
executing up to 64 single precision floating point operations concurrently. 2

Each SM contains its own L1-cache, which is divided be-
tween hardware managed and software managed (“shared”) memory. Additionally, all SMs share a common L2-cache.
For our system, each SM has 64 KiB of L1 cache, and all
SMs share 768KB of L2 cache. The relatively small amount
of cache (1/23 as much as CPU L3) on the GPU is balanced by a significantly higher bandwidth between the processor chip and off-chip memory (DRAM) compared to the CPU. A scale drawing of the caches available on the GPU can be

2In CUDA terms, the warp width is 32, and the floating pipeline can issue up to two warps simultaneously

seen in Figure 2(c). The GPU has its own memory system, and any data the GPU uses must first be transferred from
CPU DRAM to GPU DRAM over PCIe. For our applica-
tion, this transfer overhead can be amortized over a large
amount of computation and is not a major concern.

While the programming model for the SIMD units of
each SM is somewhat more flexible than on the CPU, lever-
aging the flexibility typically comes at the cost of reduced concurrency. 3 Therefore, for the purposes of comparing
hardware architectures, the SIMD units on the GPU are
roughly analogous to the CPU SIMD units. Thus, in sum-
mary, the GPU hardware provides two levels of concur-
cency: SM-level concurrency and thread-level concurrency. Note that the GPU provides more concurrency than a CPU
at both levels (14 SMs vs. 8 cores) and (64 wide vs. 4 wide
SIMD units).

4. Parallelism in the Face Alignment Stage

We will next discuss the parallelism available in the face
recognition pipeline, and propose techniques for exploiting this parallelism on the GPU and CPU hardware described
in the previous section. In this section we first focus on the
face alignment stage (see Figure 1).

Face alignment (3) estimates an image transformation \( \tau \) that rectifies the query image \( \vec{b} \) with possible pose variation w.r.t. each training class \( A_i \), which leads to the minimal sparsity in error \( e \) after the alignment. Note that directly solving (3) is inefficient since it is a non-convex problem and may exhibit local minima. However, given a good initial estimate of the transformation \( \tau \) (e.g., provided by an accurate face detector), the optimal solution for \( \tau \) can be sought iteratively by linearization at each \( j \)th step:

\[
\min_{x, e, \Delta \tau} \| e \|_1 \quad \text{sub to} \quad \vec{b} \circ \tau_j + J_j \Delta \tau_j = A_i x + e. \tag{7}
\]

where \( J_j = \nabla_{\tau_j} (\vec{b} \circ \tau_j) \) is the Jacobian, and \( \Delta \tau \) is a step update to \( \tau \). Denote \( \vec{b}_j = \vec{b} \circ \tau_j, B_j = [A_i, -J_j] \) and \( \vec{w}^T = [x^T, \Delta \tau^T] \), then the update \( \Delta \tau \) can be computed by solving the following linear program:

\[
\min_{\vec{w}, e} \| e \|_1 \quad \text{sub to} \quad \vec{b}_j = B_j \vec{w} + e. \tag{8}
\]

The per-class alignment algorithm via ALM is summarized in Algorithm 2.

In summary, the alignment stage essentially contains two
levels of available parallelism. At the higher level, there are per-class alignment problems that are solved independently, or problem-level parallelism. At a lower level, the first-order linear algebraic operations exhibit parallelism within
Algorithm 2 (Face Alignment via ALM)

Input: \( b, A_i, x_0 = 0, y_0, \) and \( J_0 \).

1: while not converged \((j = 1, 2, \ldots)\) do
2: Update \( b_j \leftarrow b_{j-1} - \frac{1}{x_{j-1}} \), 
3: Initialize \( w_0 = 0, y_0 = 0 \).
4: while not converged \((k = 1, 2, \ldots)\) do
5: \( u_k = u_{k-1} \), \( z_0 = e_{k-1} \).
6: while not converged \((l = 1, 2, \ldots)\) do
7: \( z_l \leftarrow \text{shrink} \left( b_j - B_j u_{l-1} + \frac{y_{k-1}}{\mu_{k-1}} \right) \).
8: \( u_l \leftarrow B_j^T \left( b_j - z_l + \frac{y_{k-1}}{\mu_{k-1}} \right) \).
9: end while
10: \( w_k \leftarrow u_l, e_k \leftarrow z_l \).
11: \( y_k \leftarrow y_{k-1} + \mu_{k-1} (b_j - B_j u_l - e_k) \).
12: \( \mu_k \leftarrow \rho \mu_{k-1} \).
13: end while
14: Update \( e_j, \tau_j, \) and \( J_j \).
15: end while

Output: \( T_j^* \leftarrow T_j, e_j^* \leftarrow e_j \).

image operations, i.e., at the pixel level. We call this pixel-level parallelism. The following two sections discuss methods for exploiting these two levels of parallelism on CPU and GPU architectures.

4.1. CPU Implementation

Optimal implementation of Algorithm 2 on a multi-core CPU must take into account the properties of the cache hierarchy. In general, cache that is closer to the core (i.e., L1 cache) has higher bandwidth, but smaller size compared to cache that is farther from the core (L3 cache). For reference, Figure 2(b) shows the sizes of L2 and L3 caches of the Intel E5530 CPU, which is a representative example of a modern multi-core CPU.

For the E5530, the L2 cache in each core is only able to store about 16 images. For image alignment problem, the number of training images per class is typically larger than 16. In this paper, we compare two mappings of the parallelism available in the alignment stage to the concurrency available on the CPU: a naive implementation that is purely based on stock libraries and compilers, and a manually optimized version we advocate.

The naive implementation disregards problem-level parallelism, and instead maps pixel-level parallelism onto both the core-level and the SSE-level concurrency provided by the CPU. In other words, alignment of the query image is performed against a single subject at a time using all available cores on the processors. The potential advantage of this implementation is that all of the variables for the inner loop, most notably \( B_j \) and \( B_j^T \), fit in L2 cache.

In contrast, our manually optimized implementation maps problem-level parallelism onto core-level concurrency, and maps pixel-level parallelism onto SSE-level concurrency. In other words, eight instances of (8) are executed in parallel with each problem running on one of the eight cores. The advantage of this implementation is that since the cores are operating on different alignment problem instances, no data is shared between cores, and therefore no synchronization is necessary between the cores. Furthermore, even with eight problem instances running concurrently, the local variables for the inner loop still fit in L3 cache. Because of the sequential data access patterns of the solver, the CPU hardware is able to use the full L3 cache bandwidth, which is high enough to make the program CPU limited, rather than memory limited. For these reasons, this implementation outperforms the previous naive solution.

In both implementations, most of the operations take advantage of Intel Math Kernel Library (MKL), a commercial implementation of the standard Basic Linear Algebra Subprograms (BLAS). For the image resampling step, we make use of the Intel Integrated Performance Primitives (IPP) library. Both MKL and IPP are optimized for Intel multi-core CPUs, and are able to automatically utilize both core-level and vector-level concurrency (for the first implementation), but are also available in single-threaded versions (for the second implementation). For operations that are not optimized by Intel in-house libraries, such as the soft thresholding operator in step 7 of Algorithm 2, we achieve vector-level concurrency via the automatic vectorization facilities of the Intel C++ compiler (ICC) [10]. To achieve core-level concurrency we make use of the Open MP API. [7]

4.2. GPU Implementation

While on the CPU, algorithm performance is highly dependent on effective cache usage, cache is relatively unimportant on the GPU for ALM based \( \ell_1 \)-min. As can be seen in Figure 2(c), the GPU has a very small amount of cache compared to the CPU. While it might be possible to fit a single instance of the alignment problem (with a slightly reduced problem size) into L2 cache, this would not be an efficient use of the GPU’s resources. The strength of the GPU’s memory architecture for our purposes is its ability to sustain a very high bandwidth to DRAM. This bandwidth is achieved by having a very large number of threads issuing interleaved memory requests. Solving many alignment problem instances concurrently increases the number of threads that can be used.

Therefore, our recommended GPU implementation is strikingly similar in spirit to our recommended CPU implementation: on the CPU we use the cores to solve multiple instances concurrently, and on the GPU we use multiple SMs for the same purpose.4 The number of subject classes that are actually scheduled to run concurrently is chosen by

4In CUDA terms, our proposed alignment stage implementation consists of a single kernel that performs alignment for all subject classes. Each instance of the alignment problem is assigned its own thread block, and the
the GPU hardware, but can be indirectly influenced by manual tuning of the code (i.e., the kernel launch configuration in CUDA terms). We have empirically determined that performance is highest with 5-7 subject classes aligned simultaneously on each SM.

Several other aspects of the implementation merit discussion. First of all, we take advantage of the GPU’s special hardware dedicated to bilinear interpolation for the computation of \( b(\tau) \) and \( J(\tau) \), which essentially consist of resampled versions of the test image and its derivatives. Second, since there are no standard CUDA libraries that work at the SM level, we use a custom routine for computing \( B_j^T = (B_j^T B_j)^{-1} B_j^T \), with \( G \) inverted using Gaussian elimination with partial pivoting. To achieve precision comparable to the single precision LAPACK routines in Intel’s MKL with this simplistic algorithm, we use double precision. Since \( G \) is only \( n \times n \) and \( B_j^T \) is only computed once per optimization problem, the cost of the inversion is dwarfed by the cost of other steps. Similarly, sums and dot-products of large vectors are also performed in double precision.

5. Parallelization of the Face Recognition Stage

After the face alignment stage is complete, the 20 subject classes with lowest alignment error are selected for recognition, \( b \) and \( A_i \), are re-sampled to a common alignment using \( \tau_i \), and \( A_i \) are concatenated into a new matrix \( A \). A sparse representation of \( b \) w.r.t \( A \) is then recovered by solving a single \( \ell_1 \)-min problem, as shown in (2), using Algorithm 3. The coefficients in \( x \) are then used to compute error residuals that are used for classification. In this stage of the algorithm, there is no problem-level parallelism to exploit, so this section will discuss how to exploit pixel-level parallelism on both CPU and GPU hardware. Then in Section 6, we will benchmark the performance of the two architectures and further demonstrate the speed gains achieved by our proposed implementations over previously published implementations.

5.1. Recognition Stage Implementation

Compared to problem-level parallelism, the exclusively pixel-level parallelism in Algorithm 3 is relatively straightforward to exploit: On the CPU, most of the operations map well onto MKL BLAS calls, and operations that do not can be easily parallelized using OpenMP and auto-vectorization.

On the GPU, most of the operations map well onto similar calls in NVIDIA’s GPU BLAS library (CUBLAS) which, like MKL, can take advantage of both levels of concurrency available in the hardware architecture. Operations that do not map well onto the CUBLAS API were implemented directly in CUDA code. Additionally, some operations that could have been implemented via multiple BLAS calls, performance improvements were achieved by combining multiple vector-vector operations into a single kernel, due to reduced bandwidth and kernel call overhead.

In order to avoid expensive data transfer across the bus connecting the GPU card and the CPU motherboard (the PCI express bus), all of the data is transferred to GPU DRAM once, and all non-trivial tasks are performed on the GPU on data stored in GPU DRAM.

6. Experiments

In this section, we benchmark the performance of our parallel implementations of \( \ell_1 \)-min on CPU and GPU platforms. In order to show how our \( \ell_1 \)-min algorithms scale with problem size, in Section 6.1 we will begin with benchmarks for the general \( \ell_1 \)-min problem on synthetic data. We will then progress in Section 6.2 to demonstrating the speed and accuracy of our implementations as applied to the alignment and recognition stages of the face recognition problem.

6.1. Simulations on Random Data

The first experiment compares the performance of our proposed CPU and GPU implementations of the general \( \ell_1 \)-min solver (Algorithm (1)). The \( m \times n \) measurement matrix \( A \) is a random Gaussian matrix, with each entry generated from the standard normal distribution and normalized to unit column norm. The ratio of \( m/n \) is fixed at 1/2 with \( n \) varying from 1000 to 8000. The ground truth signal, \( x_0 \) has a sparsity rate of 10% of \( m \) with elements sampled from a normal distribution and normalized to unit column norm. Because the ground truth signal \( x_0 \) is known, the algorithm terminates when \( ||x - x_0|| < \tau \) with \( \tau = 10^{-3} \). The measurement vector is generated by \( b = Ax_0 \).

The results of this benchmark can be found in Figure 3. The \( x \)-axis represents the size of the \( A \) matrix and the \( y \)-

---

Algorithm 3 (Face Recognition via ALM)

1: Input: \( b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}, x_1 = 0, e_1 = b, y_1 = 0 \).
2: while not converged \((k = 1, 2, \ldots)\) do
3: \( e_{k+1} = \text{shrink}(b - Ax_k + \frac{1}{\mu} y_k, \frac{1}{\mu}) \);
4: \( t_k \leftarrow 1, x_k \leftarrow x_k, w_k \leftarrow w_k \);
5: while not converged \((l = 1, 2, \ldots)\) do
6: \( w_{l+1} \leftarrow \text{shrink}(x_l + \frac{1}{\mu} A^T(b - Ax_l - e_{k+1} + \frac{1}{\mu} y_k), \frac{1}{\mu}) \);
7: \( t_{l+1} \leftarrow \frac{1}{2}(1 + \sqrt{1 + 4t_l^2}) \);
8: \( z_{l+1} \leftarrow w_{l+1} + \frac{t_l}{t_{l+1}}(w_{l+1} - w_l) \);
9: end while
10: \( x_{k+1} \leftarrow w_l, y_{k+1} \leftarrow y_l + \mu(b - Ax_{k+1} - e_{k+1}) \);
11: end while
12: Output: \( x^* \leftarrow x_k, e^* \leftarrow e_k \).
axis represents the average amount of time to complete a single problem instance. The GPU implementation tends to be faster than the CPU implementation at solving a single large problem, whereas the CPU implementation is faster at solving a single small problem. The transition between the two regimes occurs where the size of $A$ reaches $2000 \times 1000 \times 4 = 8$ MB, i.e. the size of the CPU L3 cache.

6.2. Face Recognition Pipeline Benchmark

This section presents benchmarks of the CPU and GPU implementations of the alignment stage (i.e., Algorithm 2) as well as the recognition stage (i.e., Algorithm 3) of the face recognition pipeline.

First, in order to measure the impact of solving many $\ell_1$ problems-per concurrently on the GPU, we benchmark three implementations of the alignment $\ell_1$-min solver with $A$ of size $5120 \times 32$, and a fixed 50 inner loop iterations for each of 50 outer loop iterations. The runtime on a GTX480 GPU is averaged over a large number of trials, which are run sequentially or concurrently depending on the implementation. The results are shown in Table 1. Our proposed parallelization of the $\ell_1$-min used in the alignment stage is eight times faster than an implementation solving a single problem at a time using the stock BLAS libraries.\[5\]

Using an implementation motivated by the previous result, we now benchmark our iterative alignment implementation on real face data. For experiments on face data we use subsets of the CMU Multi-PIE Face Database. For gallery images we use frontal images from session 1, which contains 20 images per subject taken under different illuminations. For test images we use images from session 2. The training images are prepared as follows: The iterative alignment stage seeks a similarity transformation between the coordinate frame of the full-resolution test image and a “canonical” coordinate frame in which images are compared. The training images are aligned by applying a similarity transformation that maps two manually clicked outer eye corners to fixed locations in the canonical frame. A $64 \times 64$ pixel window in the canonical frame is used for resampling.

Figure 4 shows the average run time of the CPU and GPU implementations to align a query image against all the subject classes separately. We vary the total number of subject classes to show how the algorithms scale. The plateaus seen in the GPU curve result from the GPU hardware scheduling the computation of alignment problems in concurrent batches, but the overall trend is linear as expected. The manually threaded CPU implementation slightly outperforms the GPU implementation, and it surpasses the naive library threaded implementation by a wide margin. The new implementation can align the query image in $\approx 40$ ms per subject, while the fastest previously published result [20] required $\approx 600$ ms seconds per subject in a similar setting.

The next experiment compares the speed of the GPU and CPU implementations of the recognition stage. It was determined that keeping 20 gallery subjects is sufficient to ensure that the correct subject is kept for the recognition stage with 95% probability. Since recognition failures are typically caused by a poor alignment, we find that keeping more subjects for the recognition stage does not necessarily improve recognition rate.

Figure 5 shows the recognition stage runtime for canonical images of size: $32 \times 32, 48 \times 48, 64 \times 64, 96 \times 96,$ and $128 \times 128.$ For all image resolutions, the problem size is sufficiently small that the CPU is significantly faster than the GPU. Note also that for both implementations, the recognition stage takes much less time than the alignment stage.

Finally, we perform an experiment verifying the recognition accuracy of the overall pipeline. As shown in Figure 6, at the optimal resolution, which happens to match the alignment stage resolution, the GPU implementation reaches 95% recognition rate, the max achievable given the...
alignment selection process. For significantly lower resolutions, the accuracy drops off significantly. The slight difference in CPU vs. GPU accuracy may be a result of numerical precision differences in our matrix inversion and vector reduction routines.

7. Conclusion

We have demonstrated that on both CPU and GPU algorithms, parallelizations of ALM that solve multiple face alignment problems concurrently are significantly faster than implementations that rely purely on vendor-supplied BLAS libraries. Furthermore, thanks to a combination of faster hardware and more efficient use of the hardware, we have demonstrated dramatic improvements in recognition speed over previously reported implementations. As CPU manufacturers increase the number of cores and vector widths, and as GPU manufacturers increase the amount of cache, both architectures are rapidly converging towards an architectural balance that increasingly favors CPU vs. GPU accuracy may be a result of numerical precision differences in our matrix inversion and vector reduction routines.

References