Implementation of Alternating Direction Method of Multipliers (ADMM) Algorithm for Given Energy Function

Xusong Wang Tzu-Ching Hung

***Abstract*—Super-resolution (SR), a technique that aims to enhance the resolution of an imaging system, is an active research field for image restoration over the past few decades. Since it can reconstruct a clearer image from the noisy one generated by lower-dosed X-ray, it is often used in Computer Tomography (CT) to reduce the potential risk of DNA damage and cancer. Recently, new frameworks have been proposed to improve existing SR methods, one of them is Alternating Direction Method of Multipliers (ADMM).**

I. Introduction

Given multiple LR images, our main task in this paper is to take them as input and then output one reconstructed HR image based on ADMM framework with two different energy functions using Euclidean $\ell 1$-norm and $\ell 2$-norm respectively. Though in theory ADMM can be implemented on both CPU and GPU, we will mainly focus on GPU implementation and use Python implementation as CPU reference in order to compare their performance and efficiency.

After ADMM is adopted, energy functions are split into several subproblems and can be solved via different algorithms, e.g., conjugate gradient (CG), or proximal operator. These implementation details, including the algorithms used and the decomposition of energy functions will be given in later sections. The results, comparison and a short conclusion, will also be provided at the end of this paper.

II. Background

A. Energy Function

Generally, the following energy function should be constructed to estimate one high-resolution (HR) image from m low-resolution (LR) images:

$$J = \sum_{i}^{m} ||y_i - A_i x||_L + \lambda \sum_{d=(0,0)}^{(w-1,w-1)} \gamma(d)||x - S_d x||_1. \quad (1)$$

In this equation, A_i are constant matrices of size $M \times N$, λ is a constant indicating weight, y_i are the input LR images of size $M \times 1$ and x is the output HR image of size $N \times 1$. m expresses the number of input images, L denotes p-norm (e.g., Euclidean $\ell 1$-norm or $\ell 2$-norm), S_d is the shift operator with $d = (d_x, d_y)$ being the shifted distance along x-axis and y-axis within a window of size w while $\gamma(d)$ equals $\alpha^{|d_x|+|d_y|}$, where α is a constant.

In our tasks, we consider two cases where L is equal to 1 and 2 respectively, so our energy functions can be formulated as following:

$$\begin{aligned} J &= \sum_{i}^{m} ||y_i - A_i x||_1 + \lambda \sum_{d=(0,0)}^{(w-1,w-1)} \gamma(d)||x - S_d x||_1, \\ J &= \sum_{i}^{m} ||y_i - A_i x||_2^2 + \lambda \sum_{d=(0,0)}^{(w-1,w-1)} \gamma(d)||x - S_d x||_1. \end{aligned} \quad (2)$$

Although these two equations look quite similar to each other, implementation differs slightly. Details will be explained in section III.

B. Alternating Direction Method of Multipliers (ADMM)

ADMM [1] is an optimization problem solver for method of multipliers with good robustness. It deals with the following problem:

$$\begin{aligned} &minimize \;\; f(x) + g(y) \\ &subject \;\; to \;\; Ax + By = c. \end{aligned} \quad (3)$$

$f(x)$ and $g(y)$ are assumed to be convex here. The corresponding augmented Lagrangian of such a problem can be formulated by introducing a dual variable (or Lagrange multiplier) p:

$$\begin{aligned} L_\rho(x, y, p) = f(x) + g(y) \\ + p^T(Ax + By - c) \\ + \frac{\rho}{2}||Ax + By - c||_2^2. \end{aligned} \quad (4)$$

In this augmented Lagrangian, ρ denotes the update step size (or penalty parameter) for dual variable p, which can be a constant or updated adaptively. Based on three different variables x, y and p, we can split the original problem into separate subproblems and solve them in the following steps iteratively:

$$\begin{aligned} x^{k+1} &= \underset{x}{\operatorname{argmin}} L_\rho(x, y^k, z^k) \\ y^{k+1} &= \underset{y}{\operatorname{argmin}} L_\rho(x^{k+1}, y, z^k) \\ p^{k+1} &= p^k + \rho(Ax^{k+1} + By^{k+1} - c) \end{aligned} \quad (5)$$

C. Proximal Operator

Proximal operators [2] are often utilized in mathematical optimization. Such an operator of any arbitrary function $f(x)$ is defined as:

$$prox_f(v) = \underset{x}{\operatorname{argmin}} f(x) + \frac{1}{2}||x - v||_2^2. \tag{6}$$

In our tasks, we decomposed the energy functions in a way such that some of the ADMM steps match the form shown above in order to take the use of proximal operators. One main advantage is that they can be solved analytically, which accelerates the computation speed with a relatively low computation complexity.

III. Numerical Solutions

For both energy functions, they share the same constant matrices A_i and m, which are given in advance and are known without computation. λ, α, and w can be tuned and thus lead to different S_d and $\gamma(d)$ accordingly. y_i depend on the chosen input images and are also known for both energy functions, as shown in Eqn. 5, therefore, there is no need to compute some of them.

A. $\ell 1$-Norm Energy Function in ADMM

The $\ell 1$-norm energy function in Eqn. 2 can be reformulated in a more concise way:

$$\begin{aligned} J(x,z) &= \sum_{i=1}^{m+w^2} g_i(z_i), \quad subject \;\; to \\ A_i x - y_i - z_i &= 0 \qquad\qquad i \in [1,m], \\ T_i x - z_i &= 0 \qquad\quad i \in [m+1, m+w^2]. \end{aligned} \tag{7}$$

In this form, new matrices T_i and functions $g_i(z_i)$ are introduced and defined as following:

$$T_i = I_{n\times n} - S_d \quad i \in [m+1, m+w^2], \tag{8}$$

$$g_i(z_i) = \begin{cases} ||z_i||_1 & i \in [1,m], \\ \lambda\gamma(d)||z_i||_1 & i \in [m+1, m+w^2]. \end{cases} \tag{9}$$

Applying dual variable p, the augmented Lagrangian can also be reformulated:

$$\begin{aligned} L_H(x,z,p) &:= \sum_{i=1}^{m+w^2} L_{H_i}(x,z_i,p_i) \\ &:= \sum_{i=1}^{m} (g_i(z_i) + \langle p_i, A_i x - y_i - z_i\rangle \\ &\qquad + \frac{\rho_i}{2}||A_i x - y_i - z_i||^2) \\ &+ \sum_{i=m+1}^{m+w^2} (g_i(z_i) + \langle p_i, T_i x - z_i\rangle \\ &\qquad + \frac{\rho_i}{2}||T_i x - z_i||^2). \end{aligned} \tag{10}$$

Based on the augmented Lagrangian, x can be solved via conjugate gradient and z_i via proximal operator. After x and z_i are updated, we can then update p_i and ρ_i. Algorithm 1 shows the rough workflow of $\ell 1$-norm ADMM, while implementation details can be found in the next section.

Algorithm 1 ADMM Workflow of $\ell 1$-norm

```
Initialize λ, w, μ, σ, ρ, α, admmIter, ε1, ε2
Load LR images y_i, i ∈ [1, m]
procedure SOLVING ADMM
    z_i^0 = 0, i ∈ [1, m + w^2], x^0 = 0, p_i^0 = 0, i ∈ [1, m + w^2]
    while k < admmIter do
        CG(x^k)
        for i = 1 to m + w^2 do
            Prox(z_i^k)
            Update p_i^k
            Update ρ_i^k
        if Σ_i ||r_i^k||_2^2 < ε1 and Σ_i ||s_i^k||_2^2 < ε2 then
            break
        k = k + 1
    return x^k
```

B. $\ell 2$-Norm Energy Function in ADMM

Similarly, the $\ell 2$-norm energy function in Eqn. 2 can be reformulated as

$$\begin{aligned} J(x,z) &= \sum_{i=1}^{m+w^2} g_i(z_i), \quad subject \;\; to \\ T_i x - z_i &= 0 \quad i \in [1, m+w^2], \end{aligned} \tag{11}$$

with matrices T_i, functions $g_i(z_i)$, and the augmented Lagrangian being:

$$T_i = \begin{cases} I_{n\times n} & i \in [1,m], \\ I_{n\times n} - S_d & i \in [m+1, m+w^2], \end{cases} \tag{12}$$

$$g_i(z_i) = \begin{cases} ||y_i - A_i z_i||_2^2 & i \in [1,m], \\ \lambda\gamma(d)||z_i||_1 & i \in [m+1, m+w^2]. \end{cases} \tag{13}$$

$$\begin{aligned} L_H(x,z,p) &:= \sum_{i=1}^{m+w^2} L_{H_i}(x,z_i,p_i) \\ &:= \sum_{i=1}^{m+w^2} (g_i(z_i) + \langle p_i, T_i x - z_i\rangle + \frac{\rho_i}{2}||T_i x - z_i||^2). \end{aligned} \tag{14}$$

Algorithm 2 shows the rough workflow of $\ell 2$-norm ADMM, which is pretty similar to $\ell 1$-norm ADMM. But since $g_i(z_i)$ is non-convex for $i \in [1,m]$, z_i in this range should be updated using scaled conjugate gradient (SCG) instead.

C. Conjugate Gradient (CG)

Conjugate gradient (CG) is a powerful algorithm that can be used to find the solutions of linear systems or unconstrained optimization problems with symmetric and positive-definite matrices. It is based on general optimization strategies, but the search directions and step sizes are determined by second order information instead. In our tasks, it is used to update x^k, and its pseudo code is shown in Algorithm 3.

Algorithm 2 ADMM Workflow of $\ell 2$-norm

Initialize $\lambda, w, \mu, \sigma, \rho, \alpha, admmIter, \epsilon_1, \epsilon_2$
Load LR images $y_i, i \in [1, m]$
procedure SOLVING ADMM
  $z_i^0 =$ Upscaling(y_1), $i \in [1, m]$, $z_i^0 = 0$,
  $i \in [m+1, m+w^2], x^0 = 0, p_i^0 = 0, i \in [1, m+w^2]$
  while $k < admmIter$ **do**
    CG(x^k)
    for $i = 1$ to $m + w^2$ **do**
      if $i \leq m$ **then**
        SCG(z_i^k)
      else
        Prox(z_i^k)
      Update p_i^k
      Update ρ_i^k
    if $\sum_i ||r_i^k||_2^2 < \epsilon_1$ and $\sum_i ||s_i^k||_2^2 < \epsilon_2$ **then**
      break
    $k = k + 1$
  return x^k

Algorithm 3 Conjugate Gradient

Input: $A_q, b, maxIter, \epsilon$
Output: x
$i = 0$
$x = [0, ..., 0]$
$r = b - A_q x$
$d = r$
$\delta_{new} = r^T r$
$\delta_0 = r^T r$
$tolerance = \epsilon^2 \delta_0$
while $i < maxIter$ and $\delta_{new} > tolerance$ **do**
  $q = A_q d$
  $\alpha = \delta_{new} / d^T q$
  $x = x + \alpha d$
  $r = r - \alpha d$
  $\delta_{old} = \delta_{new}$
  $\delta_{new} = r^T r$
  $\beta = \delta_{new} / \delta_{old}$
  $d = r + \beta d$
  $i = i + 1$
return x

D. Scaled Conjugate Gradient (SCG)

One of the largest drawback of CG is that it only works for positive definite cases and fails to converge to a stationary point for other cases. As a result, a modified version of CG, called scaled conjugate gradient (SCG) [3], has been introduced to fix this problem. Algorithm 4 shows how z_i^k with $i \in [1, m]$ in $\ell 2$-norm is updated using SCG. One main difference is that CG uses line-search technique to estimate the step size and direction, while SCG utilizes a non-symmetric approximation of the gradient instead. Also, a new scalar is introduced and is used to regulate the indefiniteness of the second order information.

Algorithm 4 Scaled Conjugate Gradient

Input: $A, x, y, z_1, p, \rho, maxIter, N$
Output: z_k
$\sigma_0 = 10^{-4}$
$\lambda_1 = 10^{-6}$
$\bar{\lambda_1} = 0$
$d_1 = r_1 = -\nabla_z J(z_1)$
$k = 1$
$success = true$
while $k < maxIter$ **do**
  if $success = true$ **then**
    $\sigma_k = \sigma_0 / |d_k|$
    $\delta_k = d_k^T [\nabla_z J(z_k + \sigma_k d_k) - \nabla_z J(z_k)] / \sigma_k$
  $\delta_k = \delta_k + (\lambda_k - \bar{\lambda_k}) |d_k|^2$
  if $\delta_k \leq 0$ **then**
    $\bar{\lambda_k} = 2(\lambda_k - \delta_k / |d_k|^2)$
    $\delta_k = -\delta_k + \lambda_k |d_k|^2$
    $\lambda_k = \bar{\lambda_k}$
  $\mu_k = d_k^T r_k$
  $\alpha_k = \mu_k / \delta_k$
  $\Delta_k = 2\delta_k [J(z_k) - J(z_k + \alpha_k d_k)] / \mu_k^2$
  if $\Delta_k \geq 0$ **then**
    $z_{k+1} = z_k + \alpha_k d_k$
    $r_{k+1} = -\nabla_z J(z_{k+1})$
    $\bar{\lambda_k} = 0$
    $success = true$
    if k mod $N = 0$ **then**
      $d_{k+1} = r_{k+1}$
    else
      $d_{k+1} = r_{k+1} + d_k (|r_{k+1}|^2 - r_{k+1}^T r_k) / \mu_k$
    if $\Delta_k \geq 0.75$ **then**
      $\lambda_k = 0.25 \lambda_k$
  else
    $\bar{\lambda_k} = \lambda_k$
    $success = false$
  if $\Delta_k < 0.25$ **then**
    $\lambda_k = \lambda_k + [\delta_k (1 - \Delta_k)] / |d_k|^2$
  $k = k + 1$
return z_k

IV. IMPLEMENTATION

A. Sparse Matrix-Vector Multiplication

A sparse matrix or sparse array is a matrix in which most of the elements are zero. Sparse matrix vector multiplication (SpMV) is one of the most common operations in scientific and high performance applications, and it is often responsible for the application performance bottleneck. While the sparse matrix representation has a important impact on the performance result of the application , selecting the right representation normally depends on professional knowledge and trial. A SpMV can be formally defined as $y = Ax$, where the input matrix, A $(M \times N)$, is sparse, while the input, x $(N \times 1)$ and the output, y $(M \times 1)$, vectors are dense. In the

simple example below, the shape of sparse matrix A is 4×4, with the input x and the output y being 4×1.

$$\begin{bmatrix} 0 & 6 & 1 & 0 \\ 2 & 0 & 8 & 3 \\ 0 & 0 & 4 & 0 \\ 0 & 7 & 5 & 0 \end{bmatrix} * \begin{bmatrix} 2 \\ 4 \\ 6 \\ 8 \end{bmatrix} = \begin{bmatrix} 34 \\ 76 \\ 24 \\ 58 \end{bmatrix}$$

B. Sparse Matrix Representation

Because most of the elements of a sparse matrix are null, considering the arithmetic operations and storage, it would be a waste of space and time on them without compression. Therefore, researchers have designed a number of compressed storage representations to store only the nonzeros.

1) Compressed Sparse Row (CSR): Compressed sparse row (CSR) format is the most popular, general-purposed sparse matrix representation. It stores column indices and nonzeros in array indices and data explicitly, and then uses a third array ptr to store the starting nonzero index of each row in the sparse matrix (i.e., row pointers). For an $M \times N$ matrix, ptr is of size $M+1$ and stores the offset into the ith row in $ptr[i]$. Thus, the last entry of ptr is the total number of nonzeros. CSR format is a natural extension of COO (coordinate format) format by using a compressed scheme. In this way, CSR can reduce the storage requirement. More importantly, the introduced ptr facilitates a fast query of matrix values and other interesting quantities such as the number of nonzeros in a particular row.

If we represent the 4×4 matrix A situated above using CSR format, it becomes:

$$ptr = [0, 2, 5, 6, 8],$$

$$indices = [1, 2, 0, 2, 3, 2, 1, 2],$$

$$data = [6, 1, 2, 8, 3, 4, 7, 5].$$

C. Sum Reduction

The sum reduction is a step that is required for the calculation of the dot product operation. In comparison, all products can be computed at the same time. Meanwhile, the reduce requires a combination of all indices. In a naive approach, one work-item sums up all values one after another, which results in the complexity of $O(n)$. The most common algorithm [Harris 2007][4] uses one work-item for a pair of inputs, updates one value with their sum, and disregards the other values, thereby halving the number of remaining values. This step is repeated until only one value remains so the total complexity is $O(logn)$.

D. CSR-Stream

CSR-Stream: Although CSR is a general-purposed sparse matrix representation and it has compressed the storage, this format is not designed for arithmetic operations on GPU, such as SpMV. On one hand, CSRs usually has good performance on the CPU, on the other hand, since CSRs have irregular memory access patterns, load imbalance and lack of parallelism, generally, leads to low performance if GPU applies multiplications based on the CSR format. Although there are matrix formats which are specialized for arithmetic computations, they are not widely adopted and conversion from or to CSR causes a lot of overhead. Greathouse et al. [5] introduced a fast SpMV algorithm which uses the CSR format, called CSR-Stream.

CSR-Stream takes benefit of multiple work groups. Each work group processes a certain number of non-zero coefficients of the matrix and then multiplies them with respective coefficients of the input vector. The number of processed rows might differ between different work groups. Therefore, we use $RB[]$ to indicate each work group the number of the rows that should be processed. For example, a work group needs to process $RB[workgroupID + 1] - RB[workgroupID]$ rows. The result of these multiplications is stored in local memory. This streaming into local memory is designed to be coalesced in terms of global memory access. After this stream, each work item of a work group performs the last reduction step, i.e. a summation to calculate a resultant coefficient. Each work item might need to perform sum reduction multiple times on local memory. The advantage of local memory is that access is much more efficient than global memory. Algorithm 5 shows the pseudo-code of CSR-Stream.

CSR-Streaming and sum reduction are two important steps, and they are implemented in the kernels csr_stream and $csr\_sum\_reduction$ of our programs respectively.

Algorithm 5 CSR-Stream

Input: $Val[], OS[], II[], RB[], x[]$
Output: $output[]$
  $startRow = RB[workgroudID]$
  $nextStartRow = RB[workgroudID + 1]$
  $numNonZeros = OS[nextStartRow] - OS[startRow]$
  $Stream(val, OS, II, numNonZeros)$
  $numRows = nextStartRow - startRow$
  $localRow = startRow + localID$
  while $localRow$ ¡ $numRows$ **do**
    $threadStart = OS[localRow] - OS[startRow]$
    $threadEnd = OS[localRow + 1] - OS[startRow]$
    $temp = Sum - Reduction(threadStart, threadEnd)$
    $output[localRow] = temp$
    $localRow = localRow + workGroupSize$
  return x

E. Conjugate Gradient

As algorithm mentioned in the previous section, the implementation of CG needs a sparse matrix A_q and a right-hand side vector b as inputs. Based on given $\ell 1$-norm energy function, the inputs A_q and b can be formulated as:

$$A_q = \begin{cases} \rho_i^k A_i^T A_i & i \in [1, m], \\ \rho_i^k T_i^T T_i & i \in [m+1, m+w^2], \end{cases} \tag{15}$$

$$b = \begin{cases} A_i^T[(y_i + z_i^k)\rho_i^k - p_i^k] & i \in [1, m], \\ T_i^T(\rho_i^k z_i^k - p_i^k) & i \in [m+1, m+w^2]. \end{cases} \tag{16}$$

For given $\ell 2$-norm energy function, the input parameters are formulated in a quite similar way:

$$A_q = \begin{cases} \rho_i^k I_i^T I_i & i \in [1,m], \\ \rho_i^k T_i^T T_i & i \in [m+1, m+w^2], \end{cases} \tag{17}$$

$$b = \begin{cases} \rho_i^k (z_i^k - p_i^k) & i \in [1,m], \\ T_i^T(\rho_i^k z_i^k - p_i^k) & i \in [m+1, m+w^2]. \end{cases} \tag{18}$$

The adaptive coefficient ρ_i^k is updated in each iteration, so both the matrix A_q and right hand side vector b are updated in each iteration as well. The latter one is done with the kernels $cgUpdateVectorB1$ and $cgUpdateVectorB2$ based on the value of i while the former one is done in the main ADMM loop with C++ function $calcQuadMatrix$. The implementation process is described as below:

- There are three main initialization kernels, one called $initZero$ which initializes vector x (line 2) to zeros. Another kernel named $cgInitZero$ initializes the padded vector. And initialization of the vectors r and d (lines 3-4) depends on x and so they are initialized in an additional kernel $cgInit$.
- Dot products for δ and α are done by calculating pointwise products obtained in the previous kernels using $sumKernel$. The latter one also requires additional values such as δ_{new} and thus uses a separate kernel $cgUpdateAlpha$.
- The convergence property is checked on the host, so it requires reading the value of δ_{new} from GPU to the host.
- Vectors q_update, x and d are calculated in the kernels $cgUpdateVec\_Q$, $cgUpdateVecs\_XR$ and $cgUpdateVec\_D$ respectively.
- Computation of β is done in the main CG iteration routine on host and then passed as kernel argument after update.

F. Scaled Conjugate Gradient

SCG is only used when updating z_i^k with $i \in [1,m]$ in $\ell 2$-norm. Similar to CG, we need to initialize some parameters at the beginning. For simple scalars like σ_0, λ_1, $\bar{\lambda}_1$, k, and $success$, they can be set by the host directly. As for the gradient and the energy function in the initialization part and afterwards, the following kernel functions are necessary:

- $GradientKernel1$ calculates vector $G = y - Az$, which is used for further cost function and gradient computation.
- $GradientKernel2$ performs the rest of the gradient computation by first calculating $A^T G$ and then $-2A^T G + \rho(x - z) - p$.
- $CostFunction1$ calculates the dot product of two vectors. For cost function computation the input vectors are G obtained by $GradientKernel1$.
- $CostFunction2$ calculate the rest of the energy function and adds it to the result obtained by $CostFunction1$.

For other operations in the main SCG iteration routine, $CostFunction1$ is modified and reused for those that need to perform dot product. And another simple kernel function $UpdateZ$ is also added for the update of z and d.

G. Proximal Operators

For calculating z_i^{k+1} with $i \in [1,m]$, $g_i(z_i)$ can be calculated according to the proximal operator of the $\ell 1$ norm as following:

$$\begin{aligned} z_i^{k+1} =& \underset{z_i}{\text{argmin}} ||\mathrm{z_i}||_1 + \frac{\rho_i^k}{2} ||z_i - A_i x^k - y_i + \frac{p_i^k}{\rho_i^k}||_2^2 \\ =& prox_{p_i^{-1}||\bullet||_1}(A_i x^k + y_i - \frac{p_i^k}{\rho_i^k}), \end{aligned} \tag{19}$$

$$[z_i^{k+1}]_j = \begin{cases} [A_i x^k + y_i - \frac{p_i^k}{\rho_i^k}]_j - \frac{1}{\rho_i^k} & [A_i x^k + y_i - \frac{p_i^k}{\rho_i^k}]_j \geq \frac{1}{\rho_i^k}, \\ 0 & [A_i x^k + y_i - \frac{p_i^k}{\rho_i^k}]_j \leq \frac{1}{\rho_i^k}, \\ [A_i x^k + y_i - \frac{p_i^k}{\rho_i^k}]_j + \frac{1}{\rho_i^k} & [A_i x^k + y_i - \frac{p_i^k}{\rho_i^k}]_j \leq -\frac{1}{\rho_i^k}. \end{cases} \tag{20}$$

For calculating z_i^{k+1} with $i \in [m+1, m+w^2]$, $g_i(z_i)$ can be calculated according to the proximal operator of both the $\ell 1$ norm and $\ell 2$ norm as following:

$$\begin{aligned} z_i^{k+1} &= \underset{z_i}{\text{argmin}} \lambda\gamma(\mathrm{d}) ||\mathrm{z_i}||_1 + \frac{\rho_i^k}{2} ||z_i - T_i x^k - \frac{p_i^k}{\rho_i^k}||_2^2 \\ &= prox_{\lambda\gamma(d) p_i^{-}1||\bullet||_1}(T_i x^k + \frac{p_i^k}{\rho_i^k}) \end{aligned} \tag{21}$$

$$[z_i^{k+1}]_j = \begin{cases} [T_i x^k + \frac{p_i^k}{\rho_i^k}]_j - \frac{\lambda\gamma(d)}{\rho_i^k} & [T_i x^k + \frac{p_i^k}{\rho_i^k}]_j \geq \frac{\lambda\gamma(d)}{\rho_i^k}, \\ 0 & [T_i x^k + \frac{p_i^k}{\rho_i^k}]_j \leq \frac{\lambda\gamma(d)}{\rho_i^k}, \\ [T_i x^k + \frac{p_i^k}{\rho_i^k}]_j + \frac{\lambda\gamma(d)}{\rho_i^k} & [T_i x^k + \frac{p_i^k}{\rho_i^k}]_j \leq -\frac{\lambda\gamma(d)}{\rho_i^k}. \end{cases} \tag{22}$$

H. Update of p

There is only a little difference between the implementation of $\ell 1$-norm and $\ell 2$-norm for the update of p. Both solutions are still straightforward to implement. The update according to $\ell 1$-norm is:

$$p_i^{k+1} = \begin{cases} p_i^k + \rho_i^k (A_i x^{k+1} - y_i - z_i^{k+1}) & i \in [1,m], \\ p_i^k + \rho_i^k (T_i x^{k+1} - z_i^{k+1}) & i \in [m+1, m+w^2]. \end{cases} \tag{23}$$

In the case where $i \in [1,m]$, a single SpMV operation between the ith constant matrix A_i and vector x is needed. This relevant kernel is called $Solve\_P2$. As for the case where $i \in [m+1, m+w^2]$, the kernel is named $Solve\_P1$, and matrix T_i equals $I - S_d$.

The update of p according to $\ell 2$-norm is as following:

$$p_i^{k+1} = \begin{cases} p_i^k + \rho_i^k (x^{k+1} - z_i^{k+1}) & i \in [1,m], \\ p_i^k + \rho_i^k (T_i x^{k+1} - z_i^{k+1}) & i \in [m+1, m+w^2]. \end{cases} \tag{24}$$

It also includes only one SpMV operation and some coefficient-wise operations. For $i \in [1, m]$, there is even no SpMV operation necessary.

I. Update of ρ

The penalty parameter ρ_i is updated in the end of each iteration:

$$\rho_i^{k+1} = \begin{cases} c^{inc}\rho_i^k & ||r_i^k||_2 > c||s_i^k||_2, \\ \rho_i^k / c^{dec} & ||s_i^k||_2 > c||r_i^k||_2, \\ \rho_i^k & otherwise. \end{cases} \tag{25}$$

where c^{inc}, c^{dec}, and c are constants with $c^{inc} > 1$, $c^{dec} > 1$, and $c > 1$. The primal and dual residuals r_i^k, s_i^k for $i \in [1, m]$ are calculated as:

$$\begin{aligned} r_i^{k+1} &:= A_i x^{k+1} - z_i^{k+1} - y_i, \\ s_i^{k+1} &:= -\rho_i^k A_i^T (z_i^{k+1} - z_i^k). \end{aligned} \tag{26}$$

While the primal and dual residuals r_i^k, s_i^k for $i \in [m+1, m+w^2]$ are calculated as:

$$\begin{aligned} r_i^{k+1} &:= T_i x^{k+1} - z_i^{k+1}, \\ s_i^{k+1} &:= -\rho_i^k T_i^T (z_i^{k+1} - z_i^k). \end{aligned} \tag{27}$$

V. Evaluation

A. Parameter Settings

Both energy functions take $m = 4$ LR images as input. The window size w and α can be tuned but we choose them to be 2 and 0.4 respectively in our cases to make it consistent with Python reference. Initial ρ_i equals to 0.001 for both energy functions with c^{inc}, c^{dec} being 2 and c being 10. λ can also be tuned, and we choose it to be 0.5 for $\ell 1$-norm and 0.1 for $\ell 2$-norm.

Although our program works for various images, most of them show similar behaviour. As a result, we only take one of them as the representative and explain its results in the following parts. The image we choose is shown in Fig. 1.

Fig. 1. Ground Truth of Camera_180+1.8

B. Performance of ADMM

One of the most common ways to compare performance is by using peak signal-to-noise ratio (PSNR). For the resultant HR image $\hat{x}$ with a given ground truth image x^*, its PSNR can be calculated as:

$$PSNR(\hat{x}, x^*) = 10log_{10}(\frac{I_{max}^2}{MSE}), \tag{28}$$

with I_{max} being the maximum possible pixel value of the image and MSE denotes the mean squared error between $\hat{x}$ and x^*.

Fig. 2 shows the PSNR of both energy functions over 400 iterations. Python is our CPU reference, and CPU is the result obtained when implementing our GPU codes on a CPU. Not surprising, GPU and CPU achieve the same performance, which is very close to our Python reference but slightly better than it for both energy functions. From this figure, we can see that PSNR grows rapidly in the first 10 iterations and then eventually converges to 26.5 dB for $\ell 1$-norm. As for $\ell 2$-norm, it is completely different. Its PSNR starts at the highest peak and keeps decreasing during every iteration.

Fig. 3 shows all reconstructed HR images after 400 iterations. Images in the upper row are generated by Python while those in the lower row are generated by our GPU codes. We can see that our codes generate brighter images with sharper boundaries. The left column is based on $\ell 1$-norm energy function while the right column is based on $\ell 2$-norm energy function. Obviously, using $\ell 2$-norm will lead to some artifacts which do not appear in $\ell 1$-norm cases.

C. Runtime of ADMM

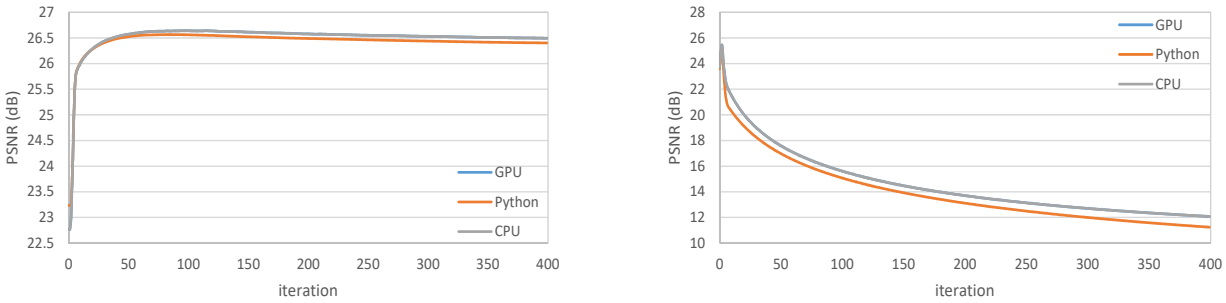
Table 1 shows the time required for each method to complete 400 iterations. Kernel indicates the computation time spent on a device platform, communication expresses how long it takes to pass the buffers between device and host, and the total is the sum of all operations on both device and host. From this table, we can see that although $\ell 2$-norm is worse than $\ell 1$-norm in terms of PSNR, it is much faster than $\ell 1$-norm regardless of platform type.

TABLE I
ADMM Runtime For 400 Iterations

Methods		Runtime (sec)		
Platform	Energy Function	Kernel	Communication	Total
Python	$\ell 1$	-	-	625.92
	$\ell 2$	-	-	180.13
CPU	$\ell 1$	50.49	3.48	376.28
	$\ell 2$	37.76	1.02	131.13

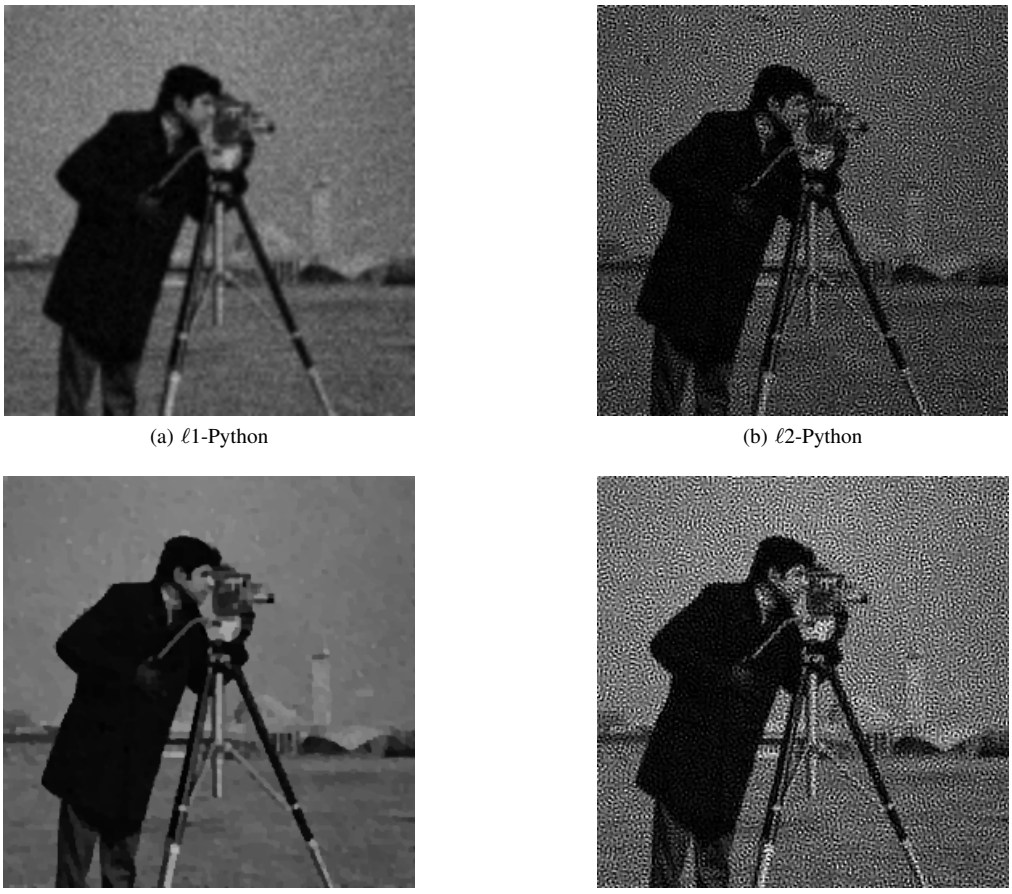
VI. Conclusion

In this paper, we first briefly introduce the purpose of ADMM and demonstrate how it can be used to reconstruct HR images from LR images based on different energy functions. We also elaborate some mathematical techniques used to solve the subproblems, e.g. CG, SCG, and proximal operator. The



(a) $\ell 1$-PSNR (b) $\ell 2$-PSNR

Fig. 2. ADMM PSNR over 400 iterations



(a) $\ell 1$-Python (b) $\ell 2$-Python

(c) $\ell 1$-GPU (d) $\ell 2$-GPU

Fig. 3. Resultant HR images over 400 iterations

performance (PSNR) and runtime for $\ell 1$-norm and $\ell 2$-norm energy functions on different platforms are given, and it is proven that by utilizing GPU and CSR representation we can indeed speed the process since matrix multiplication is the most time-consuming part of all calculation.

The results also show that although $\ell 2$-norm with SCG leads to poor performance and artifacts, it is much faster than $\ell 1$-norm, which achieve better performance. As a result, we cannot say that using $\ell 1$-norm is definitely a better choice. The choice of energy function type and hence the suitable mathematical approach depends on the application.

REFERENCES

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato and J. Eckstein, *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers*. Foundations and Trends® in Machine Learning, vol. 3, no. 1, pp. 1-122, 2010.

[2] N. Parikh and S. Boyd, *Proximal Algorithms*. Foundations and Trends® in Machine Learning, vol. 1, no. 3, pp. 123-231, 2013.

[3] M. F. Møller, *A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning*. Neural Networks, vol. 6, pp. 525-533, 1993.
[4] M. Harris, *Optimizing parallel reduction in cuda*. NVIDIA Developer Blog, 2007.
[5] J. L. Greathouse and M. Daga, *Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format*. International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 769–780, 2014.