**Description**

First give one or two sentences to point out what you want to provide to the reader in this section and how these material is organized. Then very often you want to cover the following subjects in this section.

In this section, we elaborate the details about the implementation of the dictionary learning program. We will first describe the algorithm, and then point out the most time-consuming part of the program – the K-SVD by analyzing the computational complexity. Finally, we will discuss in detail about how to parallelize the K-SVD.

**2.1. Objectives and Technical Challenges**

Give the detailed, very often are enumerated objectives which can be derived from the goal of the project, then describe briefly the corresponding technical challenges.

Implement the parallelized version of dictionary learning algorithm for SR, including

Two objectives are derived:

<enumerate >

design the dictionary learning program whose routines include extracting patches signals, training the low -resolution dictionary and computing the high-resolution dictionary.

parallel the K-SVD part of the dictionary learning program. It involves paralleling orthogonal matching pursuit algorithm and handling random memory accessing issues.

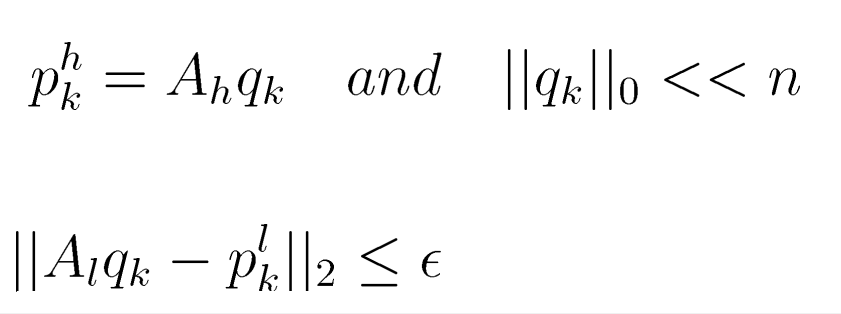
**2.2. Problem Formulation and Design**

Give the detailed, one-to-one correspondence description of your design to attach/solve the problem and to achieve the objectives and the goal of the project:

1. Use engineering language and mathematical formulation;
2. Provide system block diagrams and circuit schematics for your hardware design;
3. Give flow chart and pseudo code description for the step-by-step discussion of your software design.

3 step: extract patch signals, learning the low-resolution dictionary with K-SVD, computing the high-resolution dictionary

The dictionary learning algorithm tries to find the optimal dictionary pair Al and Ar and sparse representation for training signals that minimize the representation error. It can be formulated as the optimization problem:



Dictionary learning routine can be generally divided into 3 steps, which is depicted by flow chart figure

<enumerate>

step 1 extract the patch signals from the training images

step 2 train the low-resolution dictionary using K-SVD

step 3 compute the high-resolution dictionary from the low-resolution dictionary.

The training process takes around 20 images and extracts over 20,000 patch signals for training. This process needs a great amount of computation, most of which is caused by K-SVD. The K-SVD is used to get the low-resolution dictionary, in other word, to solve the sub-problem of the whole problem equation

<k-svd objective>

In each iteration of K-SVD, the algorithm computes? the sparse representation of all training signals under current dictionary with orthogonal matching pursuit(OMP) algorithm and then uses the spare representation to refine the dictionary. The pseudo code for K-SVD is shown as belowalgorithm

<K-SVD pseudo code>

The main cause of the great amount of computation of K-SVD is that it will perform over 10,000 OMP in each iteration. Hence, the acceleration the OMP procedure is the key to accelerate the entire K-SVD algorithm. In the next paragraph, the computation complexity of OMP will be analyzed in detail.

Algorithm is the pseudo code of OMP algorithm. In each iteration, the computation performed during OMP procedure includes computing M times of inner products, solving a over complete linear system and performing a matrix multiplication and subtraction. For a typical super resolution problem, low-resolution is consisted of over 1000 atoms and the size of each atom vector is around 50. Besides, each sparse representation will contain less than 5 non-zeros elements. Hence, for each iteration of OMP, there are over 1000 inner products, a linear system with size of 30 X 5 and matrix multiplication and subtraction with a matrix size of 30 x 5 in total.

The scale of computation for OMP brings a challenge: it is too large to fit into a single CUDA thread and it is also not large enough to use a CUDA grid to compute, which will cause a great waste of GPU resources. Therefore, the batch technology is used. With batch, GPU resource can be shared by more than one streaming, each of which computing a linear system solver or matrix multiplication for one signal OMP operation.

Besides, the resource allocation and the memory access pattern of OMP operation also raise a challenge. For each iteration of single OMP, the program loads the dictionary atom corresponding to the largest inner product, which means which atom is loaded is not determined until the run time, depicted by figure

<random access image>

The atom selecting and loading tasks are executed by CPU with cudaMemcpy function in traditional way. It will cause a great number of memory access between host and device, which is an intolerably time-consuming. To solve the problem of random memory access, we write series of dedicated GPU kernel program to do the atom loading.

K-SVD cost the most of computation🡪 focus

<ksvd persudocode>

🡪omp costs the most of computation of K-SVD 🡪 performs over 10,000 times for each iteration because of the great number of patch signals needed.

<omp persudocode>

analyze omp: 🡪 thousands of vector inner product, 1 solve overcomplete linear system(persudo inverse) 🡪 1 matrix computation (1 multiplication and 1 substraction)

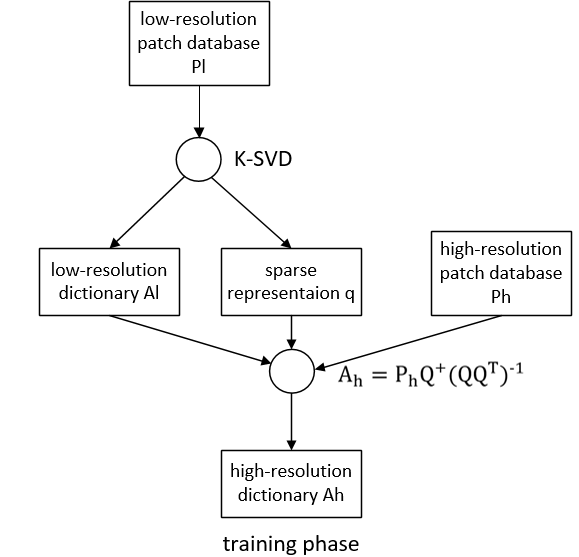
🡪not a simple computation routine🡪brings massive computation when omp is performed hundreds of thousands of times in only one single iteration.

using GPU to accelerate omp computation. Parallelly perform omp for batches of signals

challenges:

computation scale for each omp 🡪 too large to fit into a single thread of CUDA. too small to parallelize the computation with grid of threads, cause a great waste of GPU resources. Using batch technology and compute matrix multiplication and persudo inversion for several signal simultaneously.

randomly memory access. load the dictionary atom corresponding to the largest inner product. 🡪 demined in the running time. 🡪 control atom loading with CPU and cudaMemcpy function 🡪 frequent memory accessing from GPU and CPU ruins the performance of program 🡪 write the dedicated kernel



<access image>

<persudo code of parallelized OMP>

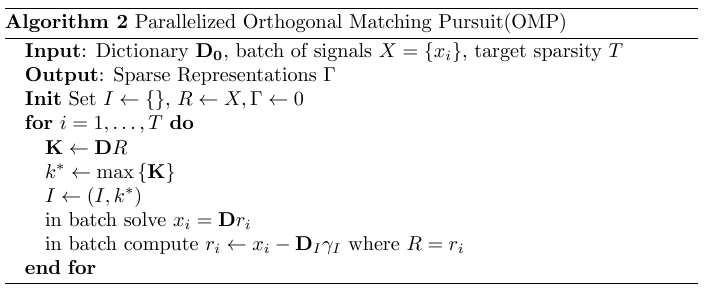
**2.3 Software Design**

Provide the following description and discussion:

1. Flow chart or flow charts, very often one should provide one top level flow chart, then additional flow charts for detailed lower level implementations.
2. Algorithm, e.g., description of the step by step implementation.
3. Pseudo code for each section of the implementation.
4. Bibucket link

To make the report concise, the pseudo code for the whole dictionary learning program is placed in the appendix, and we focus the parallel part of our code in this section. The algorithm describes K-SVD and the algorithm shows the parallelized version OMP.

<pseudo code of parallelized OMP>



The parallelized OMP process one batch of signals for each run. It has 5 steps for each iteration. We will discuss the implementation details of these 5 steps respectively.

The first step is to compute the inner products between the dictionary atoms and the signal vectors. This operation can be realized with a single matrix multiplication as shown in the pseudo code, where X is the combination of batch signals whose each column corresponding to a single signal.

The second and third steps are to find the position of the atom with the max inner production and assign the found indices to the atom loading pointer I. These two steps are implemented within one kernel program, which finds the max element of each column of input matrix and sets the corresponding element to I? in the output matrix.

The fourth step is to solve linear system for each signal in batching fashion. Before this, the corresponding matrix DI and xi should be loaded into device according to the atom loading pointer I.

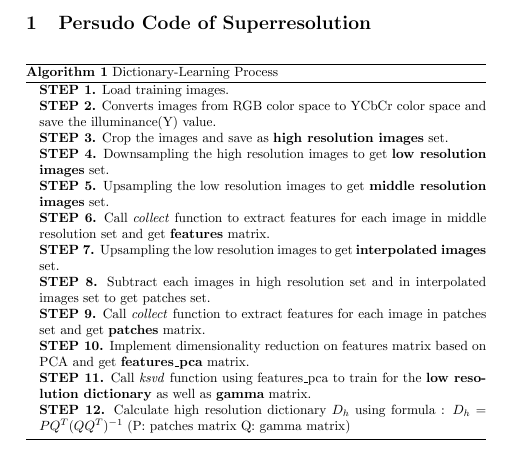
The final step is to calculate the current residues for each signal between its representation and itself, and, the same as the fourth? step, it also should load the related data into device.

We write the dedicated kernel to do the data loading job for step 4 and step 5, and use the cuBlas API to solve the linear system.

At the end of the OMP iteration, the output should be converted from dense format into sparse format. Another kernel has been written to perform this conversion.

kernel list for each step OMP

1. Cublas 🡪 batch function 🡪 persudo inverse and matrix multiplication
2. dedicated kernel find max and renew gammatab
3. dedicated kernel to load b, atom
4. dedicate kernel to build the sparse vector from c to get the final output



<collection persudo code>

K-SVD persudo code

parallel OMP persudo code

**3. Results**

Provide detailed

1. Description of the platform, tools, conditions
2. Description of results
3. Figures, plots
4. Testing, verification

1.divide the program into two part-> one in local, on in server

parallel version : C library 🡪 python ctypes call

output dictionary succeeds in restoring images

2, performace comparison of K-SVD(to be d)

3impact of batch computing

<batching findmax nvvp>

5.impact of batch size

4.impact of random access memory

**5. Discussion and Further Work**

Provide the assessment and critique of the obtained results. Do results seem reasonable. Are the results anticipated by previous work, are they worse or better, and why. How complete are the results. In retrospect, what could have been done differently.

What else can be done, better, different, or more?

We succeed in parallelizing the K-SVD and achieve an around 100 times improvement in speed. However, this program can be further optimized with respect of speed. On one hand, besides the OMP part of K-SVD, the atom refinement phase can also be parallelized. It has the similar operation like OMP and can also be speeded up with batching technology and dedicated kernel for loading. On the other hand, there exists parallelized code outside the K-SVD in the dictionary learning program including principle component analysis(PCA) process in the patch-signal-extracting routine.

**5. Conclusion**

Provide summary of this project, briefly review the statements made in the abstract, in particular, if the enumerated objectives and goal are achieved. Emphasize and highlight the lessons learned, point out the direction for further improvement if needed.

In our project, we succeed in implementing the dictionary learning algorithm and parallelizing the core of this algorithm – K-SVD. With the parallelized version of K-SVD, the speed of the dictionary training process was improved to a large extent. To parallelize the K-SVD, we learned the batching technology that can make GPU run several kernels simultaneously. After implementing and comparing the performance of parallel version programs implemented with cudaMemcpy function and dedicated kernel respectively, we noticed the memory access between host and device is usually the most time-consuming part of the whole program. To achieve better performance, the CUDA program should avoid the data interaction between host and device.

Although our work has improved the speed of dictionary training algorithm, there exists other processes in the whole program which can be parallelized to achieve further improvement such as the principle component analysis in the patch-signal-extraction phase.