Speed up a Machine-Learning-based Image Super-Resolution Algorithm on GPGPU

Ke Ma[1], and Yao Song[2]
[1] Department of Computer Sciences
[2] Department of Electrical and Computer Engineering
University of Wisconsin-Madison
{kma32, song82}@wisc.edu

Abstract

We develop a CUDA program to recover the 3x super-resolution version of a given low-resolution grayscale image using a machine-learning-based method. We discuss the implementation of the program, and show that it can produce high-quality super-resolution images and achieve high performance on GPU. We also point out the limitations and potential improvements of the program.

1. Introduction

In many imaging applications, it is desired or necessary to have high-resolution images. For example, in medical imaging applications, it is very difficult for a doctor to perform a correct diagnosis with low-resolution images. Besides, high-resolution displays cannot be well utilized without high-resolution images. However, low-cost image sensors are still used in most situations, and they cannot achieve the desired resolution for many applications. One direct solution to fill this resolution gap is to increase the number of pixels per unit area in the sensor. But this method is usually not practical. If the pixel size decreases to a certain value, the amount of light for each pixel is not enough for the sensor to maintain an acceptable signal-to-noise ratio, thus degrading the image quality.

Image super-resolution (SR), which can generate a high-resolution image from one or more low-resolution images, is a promising approach to this problem. The conventional way to recover a SR image is to combine a set of low-resolution images of the same scene aligned with subpixel accuracy. But this can be extremely difficult when the number of available input images is small. Although single image SR is generally an ill-posed problem due to insufficient information, various methods can be applied to regularize this problem and produce meaningful results. For example, some people use mathematical models such as curves and patches to model the images as surfaces, and sample the missing locations on the surfaces. Other people use machine learning techniques to learn the relationship between low- and high-resolution images. Our project focuses on developing a CUDA program to recover the 3x SR version of a given low-resolution grayscale image based on machine learning techniques.

2. Implementation

In this section, we describe the implementation of our program. We start from the overview of our SR method, and then describe each stage in the processing pipeline in detail. In the end, we briefly talk about how we handle input and output of the image files. All stages in the processing pipeline are implemented both on CPU and on GPU. After we describe our method, the implementation on CPU should be straightforward. Therefore, in the following sections we only talk about how to implement it on GPU and achieve better performance.

2.1. Overview of Our Method

In order to generate the SR version of an input image, we first decompose the image into square patches of the same size. We call them low-resolution (LR) patches to distinguish them with the high-resolution (HR) patches generated later. Each LR patch is represented as a vector of values between 0 and 1, which is then normalized by subtracting its mean value. The normalized LR patches are fed into the coupled dictionary to generate normalized HR patches. The coupled dictionary is comprised of a low-resolution dictionary and a high-resolution dictionary, and their entries have a one-to-one correspondence. The coupled dictionary models the relationship between normalized LR patches and normalized HR patches, because we are more interested in textures in the patches rather than their absolute intensities. As the coupled dictionary has limited size, the normalized HR patches generated by it can be regarded as “templates”, which don’t contain intensities and any patch-specific information. To further polish the HR “template” patches, they are concatenated with the original LR patches, and fed into the neural network to produce the final HR patches. As you can expect, the neural network is at lease responsible for two things: to fill in intensities, and to fine tune the HR patches. The HR “template” patches give the

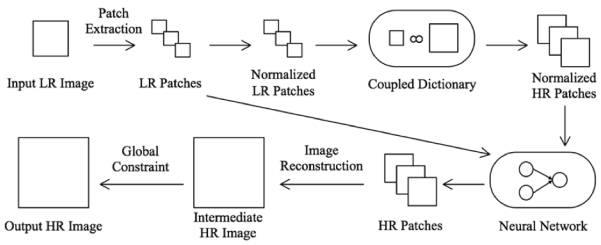


Figure 1. Diagram of the image SR pipeline.

neural network a good starting point to magnify the LR patches, without which the task is too underdetermined to have a reasonable solution. The final HR patches are then used to reconstruct the SR version of the image. There is a final step to further improve the quality of the output image, that is, to enforce the global reconstruction constraint. The entire pipeline is shown in Figure 1.

2.2. Patch Extraction

To ensure that the patches cover all the information in the input image, we should decompose the image into patches systematically. Here we extract 3x3 patches column by column, then row by row. To avoid obvious artifacts near the boundaries of the patches, neighboring patches should have some overlap, and the overlap width is decided to be 1 pixel. There are cases when the last patch in a row reaches the right boundary of the image, or the last patch in a column reaches the bottom boundary, so that it is not complete. We handle these cases by mirroring, that is, pretend that there are mirror images about the image boundaries. This technique gives us meaningful boundary patches.

The implementation of this stage is similar to a commonly used scattering operation. The number of patches in a row and in a column should be calculated in advance, so that each patch has a unique index. Each thread is then responsible for one patch in the input image. It accesses the 9 pixels in the patch column by column, row by row, and puts them in the same column of the output matrix. The column index is decided by the patch index. Handling boundary cases is as simple as mirroring the indices. After this, we change the representation of the input image from a HxW matrix to a 9xP matrix, where P is the number of patches.

2.3. Coupled Dictionary Prediction

As stated before, the LR patches are first normalized and then fed into the coupled dictionary. The size of the coupled dictionary used is 1024. We use 1-NN algorithm to find the most similar entry for each LR patch in the LR dictionary. Their indices are recorded and used to look up the corresponding entries in the HR dictionary. These HR entries then become the output HR "template" patches.

To normalize the LR patches, we first perform a sum reduction on each column of the 9xP LR patch matrix. The reduction result of each column is divided by 9 to calculate the average value of each patch. After this, each element in the matrix is subtracted by its corresponding column average. All these operations can be fused in a single kernel, and give us a 9xP normalized LR patch matrix.

We assume we have already trained the coupled dictionary in the training phase we don't describe here, and the data can be loaded directly into the memory from external files named "dict_low.txt" and "dict_high.txt". The LR dictionary is a 9x1024 matrix, while the HR dictionary is an 81x1024 matrix. To find the most similar LR entry for each LR patch, we first create a 1024xP distance matrix, where the (i, j) element indicates the squared Euclidean distance between the i^{th} LR entry and the j^{th} LR patch. This is achieved by first transpose the LR dictionary matrix, and then perform a matrix-multiplication-like operation between it and the normalized LR patch matrix. As we know, matrix multiplication calculates $(a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1} + \ldots + a_{1,k} \times b_{k,1})$ for each element in the resulting matrix, if we change the involved operators, we can instead calculate $[(a_{1,1} - b_{1,1})^2 + (a_{1,2} - b_{2,1})^2 + \ldots + (a_{1,k} - b_{k,1})^2]$, which is exactly the squared Euclidean distance. Therefore, we adopt the tiling programming pattern and implement a kernel very similar to matrix multiplication to calculate the distance matrix.

The next step is to find the index of the minimum element in each column of the distance matrix. This is similar to a min reduction on each column, but instead of manipulating values, we manipulate their indices. Now we have a 1xP index vector, which is then used to index into the 81x1024 HR dictionary and create an 81xP normalized HR patch matrix. This is like a gathering operation, and each thread is responsible for each element in the output matrix.

2.4. Neural Network Prediction

The HR "template" patches, as well as the LR patches, become the input to the neural network. The architecture of the neural network consists of one input layer, one hidden layer, and one output layer. There are 90 input units, 81 hidden units, and 81 output units. Don't forget we have bias units in the input layer and the hidden layer. Because the task considered here can be regarded as regression, we use sigmoid units in the hidden layer and linear units in the output layer. The neural network propagates the input values to the output layer via the hidden layer, and generates the output HR patches.

First, we need to concatenate the 81xP normalized HR patch matrix with the 9xP LR patch matrix vertically,

which then becomes the input to the neural network. We also need to pad one row of 1s at the top of the matrix, which serves as the value of the bias unit. The input matrix now is a 91xP matrix.

Again, we assume the neural network is trained, and we can load the weights from "weights_in.txt" and "weights_out.txt". The weights between the input layer and the hidden layer are represented in a 81x91 matrix, where the (i, j) element indicates the weight between the i^{th} hidden unit and the j^{th} input unit. Be aware that the 1^{st} input unit is the bias unit. As we know, the net input of each hidden unit is the weighted sum of the values of the input units. This can be translated into a matrix multiplication between the weight matrix and the input matrix, which results in an 81xP net input matrix. This net input matrix is then transformed by a sigmoid function to produce the output matrix of the hidden layer. These operations can also be fused. After these steps, the values are propagated from the input layer to the hidden layer.

Propagating the values from the hidden layer to the output layer is almost the same procedure. In this time, the input matrix is an 82xP matrix that is the output matrix from the last step padded by one row of 1s at the top. The weight matrix is an 81x82 matrix. Because we use linear units in the output layer, the output matrix will be the same as the net input matrix, an 81xP matrix that is the product of the weight matrix and the input matrix.

2.5. Image Reconstruction

The HR patches are then stitched together to reconstruct the HR image, which is exactly the inverse process of the first stage. As in patch extraction, there are patches that fall on the image boundaries. Here we just discard those pixels that are out of bound. We also need to appropriately handle the overlap regions. After the above processing stages, the overlap region from one patch may not be consistent with that from another neighboring patch. We handle this inconsistency by averaging the overlapped pixels.

The implementation is just another scattering operation. Each thread is responsible for one element in the 81xP HR patch matrix. They should be able to put the pixel to the right place in the (3H)x(3W) output image. However, things are not that easy, because we have to handle the overlap regions in the meantime, and it's pretty cumbersome to figure out the overlap regions in a mathematical way. Instead of creating the output image directly, we first create a "sum" matrix and a "count" matrix. In the "sum" matrix, we just accumulate values in each location; while in the "count" matrix, we count how many values are accumulated in each location. As two threads may write to the same location at the same time, we should use atomic operations here. In this way, the output image can be created by elementwisely dividing the "sum" matrix by the "count" matrix.

2.6. Enforcement of Global Reconstruction Constraint

According to our assumptions, the input LR image should be the blurred and downsampled version of the output HR image. We assume that the blurring filter is a 3x3 box filter. Therefore, one pixel in the LR image should correspond to a 3x3 patch in the HR image, and their intensities should satisfy the following equation:

$$I^{(l)} = \frac{\sum_{i=1}^{3}\sum_{j=1}^{3} I_{i,j}^{(h)}}{3\times 3}$$

where $I^{(l)}$ is a pixel in the LR image, and $I^{(h)}_{i,j}$ is a pixel in the corresponding patch in the HR image. Please note that the patch here has totally different meaning from the patches we describe in the above stages. To enforce this global reconstruction constraint, we calculate the difference between the intensity of the pixel and the averaged intensity of the patch, and add this difference to every pixel in the patch. That is:

$$I_{i,j}^{(h)} \leftarrow I_{i,j}^{(h)} + \left(I^{(l)} - \frac{\sum_{i=1}^{3}\sum_{j=1}^{3} I_{i,j}^{(h)}}{3\times 3} \right)$$

Although this looks naïve, it improves accuracy significantly in practice.

We implement this in a straightforward way. Each thread is responsible for one pixel in the LR image, that is, a 3x3 patch in the HR image. It averages the pixels in the patch, calculates the difference, and add it back to all pixels in the patch.

2.7. Image File I/O

We want our program to be able to load the input images from a commonly used image file format, and to save the output images to it as well. The file format of choice is BMP due to its simplicity. Although it is a simple file format, it still has lots of variants. Our program can only support a limited set of variants. To be specific, it can only read 8-bit grayscale or 24-bit RGB BMP files with Windows-type headers. It can only write 8-bit grayscale BMP files with Windows-type headers. It doesn't support OS/2-type headers, compressions, etc.

Our program focuses on grayscale images. Even though it is able to read RGB images, it converts them to grayscale images upon reading using the following formula:

$$I = 0.30R + 0.59G + 0.11B$$

Internally, pixels are represented as single-precision floating-point numbers between 0 and 1. But they are represented as integers between 0 and 255 in BMP files. The BMP file I/O code handles this conversion as well.

3. Evaluation

In this section, we evaluate both the accuracy and the efficiency of our program. First, we compare the result of our method with that of the baseline method in terms of the SR recovery accuracy. Then we focus more on the performance of our program, and do scaling analyses and profiling analyses for it.

3.1. Accuracy Analyses

The baseline method of choice is Bicubic Interpolation (BI), which is widely used in many image processing software suites such as Photoshop. To compare the accuracy fairly, we grab an image on the Internet, downscale it to one-third of its original size. In this way, the original image is the ground truth 3x SR image, and the downscaled image is the input to our program.

We perform the SR process on the input image using both BI and our method, and judge the accuracy of the recovered SR images qualitatively. As shown in Figure 2, our result is generally considered better than the baseline result. In our result, edges are sharper and intensities are better preserved, although there are more aliasing and blocky artifacts as well. While in the baseline result, everything is just smoother, which is sometimes not a good effect.

We also compare the results quantitatively. We calculate the Root Mean Squared Error (RMSE) per pixel between the ground truth and the recovered SR images. For the images in Figure 2, the baseline result has a RMSE of 13.52, while the RMSE of our result is only 12.09. Of course, we also test on many other images, and show that the RMSE of our method is significantly better than that of BI ($p < 0.001$).

3.2. Scaling Analyses

We perform scaling analyses for both the CPU implementation and the GPU implementation of our program. We vary the size of the input image from 4x4 to 1024x1024, and record the overall execution time of the program.

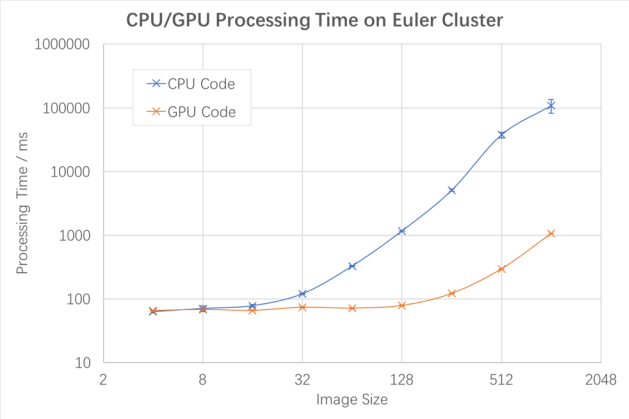
We first do the analysis on a CPU/GPU node of Euler. The CPU used is Intel Xeon E5520, which is of Nehalem microarchitecture, and has 4 hyper-threaded cores whose clock frequency is 2.27 GHz. The GPU used is NVidia GeForce GTX 480, which is of Fermi microarchitecture, and has 15 streaming processors whose core frequency is 700 MHz. As we can see in Figure 3, the GPU implementation doesn't gain any speed-up when the image size is small, because most of the time is spent on housekeeping. But when the image size gets larger and larger, the GPU implementation is significantly more efficient than the CPU implementation, and has more than 100 times speed-up.

We also do the same analysis on our MacBook Pro (retina, 15-inch, mid 2014). The CPU used is Intel Core i7-4870HQ, which is of Haswell microarchitecture, and has 4 hyper-threaded cores whose clock frequency is 2.50 GHz. The GPU used is NVidia GeForce GT 750M, which is of Kepler microarchitecture, and has 2 streaming processors whose core frequency is 967 MHz. The results are shown in Figure 4. We notice that the CPU performance here is even better, but we can still get around 100 times speed-up with a laptop GPU (2 SMs in GT 750M vs. 15 SMs in GTX 480).

3.3. Profiling Analyses

We want to investigate how much of the execution time is spent on each type of work. We profile the GPU implementation of our program on the MacBook Pro.

Figure 2. Comparison between the result of Bicubic Interpolation (middle) and that of our method (right). The input and the ground truth is shown in the left.



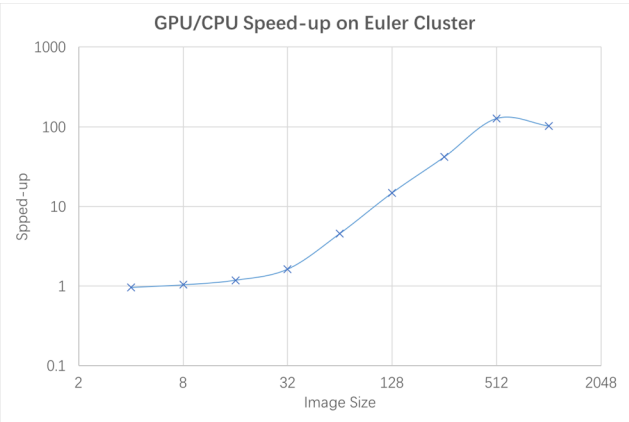
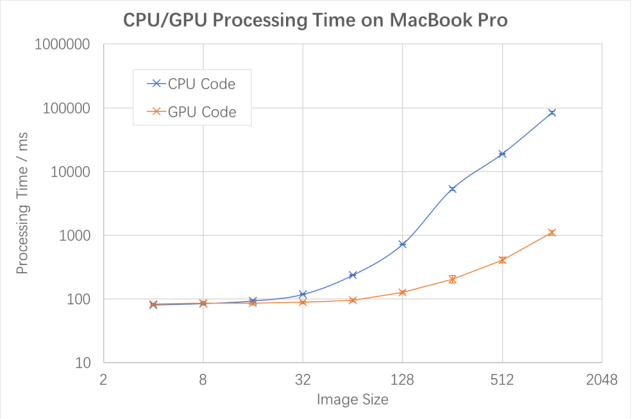


Figure 3. Scaling analyses of the CPU implementation and the GPU implementation on Euler.



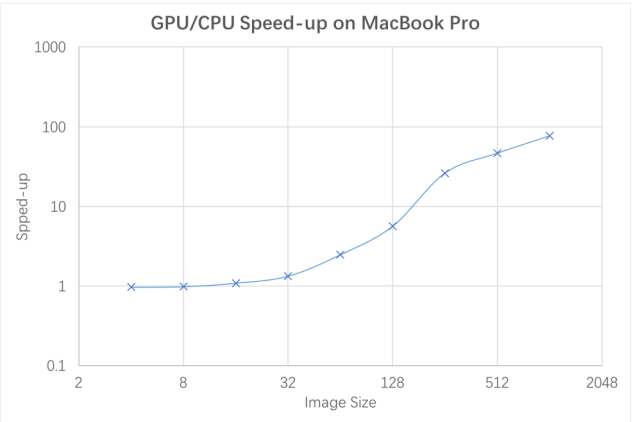


Figure 4. Scaling analyses of the CPU implementation and the GPU implementation on MacBook Pro.

When the image size is relatively small, say, 64x64 shown in Figure 5, we find that most of the time is used for housekeeping, which includes image file read and write, data files read, host and device memory allocation and release, timing, and so on. This kind of cost is really hard to reduce, because it's environment dependent. Even when the image size is small, time used for memory copies is still negligible compared to computation. This is a good news for us, because we only need to optimize the kernels.

When the image size is large, things are different. As shown in Figure 6, processing a 1024x1024 image on GPU is really computation-bounded. This enables very high speed-up over the CPU implementation. Cost of housekeeping takes up a smaller portion of the overall execution time, so we can expect even higher speed-up when images get larger. This is because the computation time scales linearly, but the housekeeping time scales sub-linearly. In terms of computation, we can see that the most expensive kernels are those matrix-multiplication-like kernels.

4. Conclusion

The resolution gap between low-cost imaging devices and high-end displays calls for more advanced image super-resolution techniques. In this paper, we present the implementation of a single image super-resolution method based on coupled dictionaries and neural networks. We show that it outperforms the widely used Bicubic Interpolation method, and that this method is highly parallelized and has impressive speed-up on GPU.

Although our project is carried out as expected, there is still room for improvement. First, there are opportunities for further optimize the host-side and the device-side code. The current implementation is mainly designed for simplicity, clarity, and code reuse. For example, all the matrix-multiplication-like operations share the same template kernel, and so does the column-reduction-like operations. Sometimes this is good for performance, but we can go further. For example, in neural network prediction, concatenating matrices and padding rows can be done implicitly in the kernel for matrix multiplication, but this prevents us for reusing the

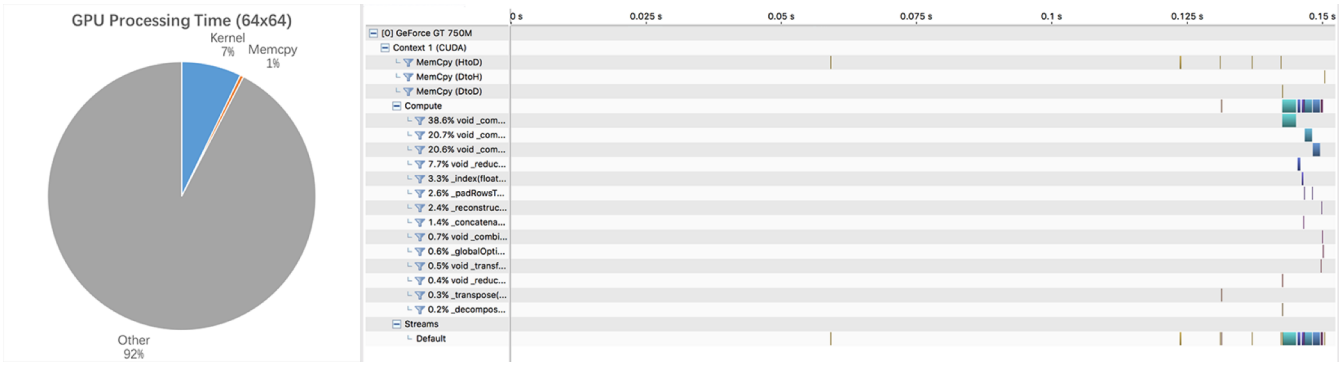


Figure 5. Profiling analyses of the GPU implementation on MacBook Pro when the input image size is 64x64.

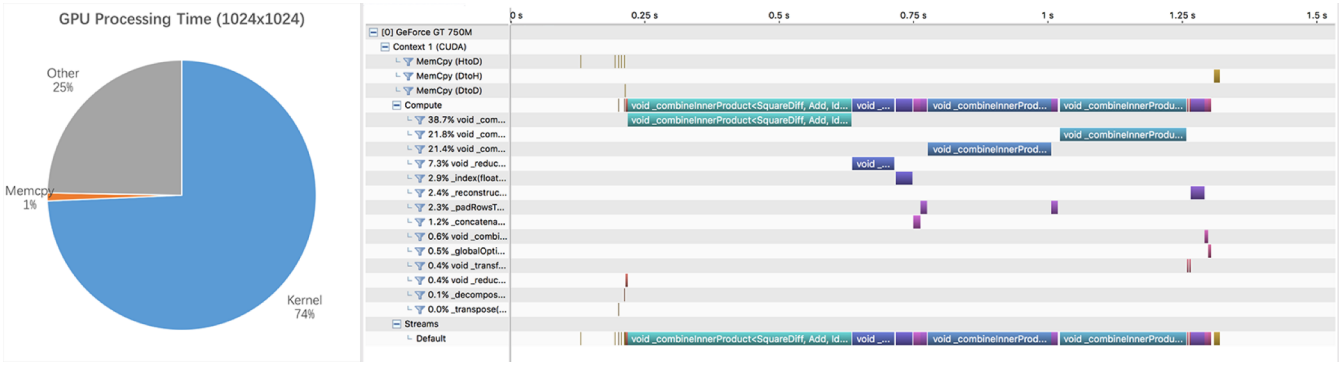


Figure 6. Profiling analyses of the GPU implementation on MacBook Pro when the input image size is 1024x1024.

existing kernels. Second, the current implementation cannot handle very large input images. We show that the GPU implementation is suitable to handle large input images for higher speed-up, but there is a limit. If the input image is too large, the GPU will soon run out of memory. A possible solution to this problem is to work on one small block of the image at a time, and take advantage of streams to accelerate the process. Third, we also want to parallelize the training phase of our SR method. In fact, the training phase takes much more time, and is extremely beneficial to be accelerated. This is also more difficult, because many of the procedures are inherently sequential.

Acknowledgement

We would like to thank Prof. Dan Negrut and his TA Ang Li. Their instructions ultimately led to the creation of this work. We would like to thank Prof. Mark Craven as well. His Machine Learning course helped us develop our method.