Project Leader - Gabriel Maliakal (CMSE)

Parallel Tricubic Interpolation to resize 3D Images

Interpolation of data is very important in all fields of research. It is the process of fitting a curve on ALL points in the data in order to interpolate values in between any two data points that are not actually present in the data. Image interpolation is a prerequisite step in a wide variety of Computer Vision related projects.

Examples of use -

A specific example of this is that I have observed is that during the training of a 3D segmentation network, we have to resize volumes so that they have the same voxel spacing (also known as isometric images) in order to make the neural network not have to figure out zoom invariance. I have observed that this is a very important step in 3D medical image segmentation, otherwise the neural network will have poor segmentation as it would be trained on a dataset having varying slice thicknesses.

Packages like scipy.ndimage can be used but this is very slow for 3D images. Especially if you want to resize 1000s 3D images to some isometric spacing of [1,1,1] this takes a very long time. I want to see if I can come up with a faster implementation and call this function in my python scripts.

Serial implementation -

Before I move onto parallelization, let us look at how I think this works sequentially -

1) First step is to compute the size of the new matrix using the zoom factor. If we have a 3D image of size Z x R x C, and if the zoom factor for each dimension is say (z,r,c) where z,r,c > 0 , then the new size of the resized matrix will be zZ x rR x cC.

2) Mapping existing pixels to new locations - The coordinates of the existing pixels will also be multiplied by the zoom factor. For example if there is a pixel at some coordinate (d,e,f) in the original image, then it will now be in coordinate (int(zd),int(re),int(cf)) . We have to make sure the new coordinates are integers, but this can be done by typecasting.
3) Perform 1D cubic interpolation along each axes-
    a) 1D cubic interpolation for every row voxel, i.e. keep the z index constant and row index constant, but vary column index from 0 to C, we get an array of voxels. Do 1D cubic on this for every value of z index and row index.
    b) Using the above sub-interpolated matrix, Repeat while keeping the C and Z index constant, vary row from 0 to R, 1D cubic interpolation this array of voxels.
    c) Using a sub-interpolated matrix obtained from steps a,b now do 1D cubic interpolation while keeping the r and c fixed and vary the z from 0 to Z for all values of r and c.

The process of getting the coefficients of each spline between every consecutive pairs of points is obtained by solving the below equation -

$$\mathbf{Ax = b}$$

where A is the system matrix containing information about the part of the coefficients of each spline and some information using the pairs of points. The vector x contains the unknown coefficients to be determined while b contains the linear slope between each consecutive point. The formula for obtaining each component is outlined in [1]. The system matrix A is a symmetric toeplitz tridiagonal matrix, the equation can be solved using a modified version of Gauss elimination in O(n) time complexity [2] using Thomas Algorithm.

Parallelization strategy -

1) Parallelize the mapping of existing pixels to new locations. This is an embarrassingly parallel operation.
2) The next step could be thought of as having two tiers of parallelism (nested) -
    a) Tier 1 - The process in each sub-step in Step 3) is an embarrassingly parallel process. However, steps a,b,c have to be done sequentially but it could be done in any order, but the steps within step a or b or c are embarrassingly parallel.

    b) Tier 2 - We can go a step further and try to parallelize the 1D cubic interpolation itself. Instead of using Thomas algorithm which is done in O(n) complexity we can parallelize the solving of tridiagonal matrix equations using many methods. One particular method is in [3] that involves splitting the tri diagonal matrix into subparts as well as the x vector that is to be solved and the b vector mentioned in the previous section.
    Note - If I am unable to perform tier 2 then I might just have to use a library like Eigen [4] that performs tridiagonal solving, since this itself can be a major homework/project.

The parallelization can be achieved by native C++ threading and openMP.
Since most operations are embarrassingly parallel a very naive native threading implementation is possible without worrying about mutexes given that I do not perform tier 2 parallelization.

To enable tier 2 parallelism I will have to figure out using multiple mutexes or I can just do it using openMP and let the package figure that out.
I suspect there will be memory issues for very large resizings, so the operations may have to be done using CUDA instead of openMP. I will have to explore how to do it in CUDA.

Benchmarking

To compare my implementation I will be comparing with the scipy and pytorch-

1) **Native threading implementation(without tier 2) vs scipy.ndimage.zoom vs torch.nn.functional.interpolate**
2) **OpenMP implementation vs scipy.ndimage.zoom vs torch.nn.functional.interpolate**
3) **OpenMP implementation** Cuda implementation vs scipy.ndimage.zoom vs **torch.nn.functional.interpolate**

In the above, I will attempt the first 4 parts with high priority. If time allows I will attempt to improve the algorithm using CUDA, based on the expectation that GPUs handle big data better parallelly than threaded CPUs.
The c++ function will be called in python. This can be done using Boost[5] The numpy array containing the image will be saved as a .npy file and passed to the boost c++ function. The internal c++ function will read this numpy array using another library called cnpy[6].
Strong scaling and weak scaling will be observed for my implementation vs existing implementation.

References-

[1] https://www.giassa.net/?page_id=274
[2] https://arxiv.org/pdf/1402.5094.pdf
[3] https://web.alcf.anl.gov/~zippy/publications/partrid/partrid.html
[4] http://eigen.tuxfamily.org/index.php?title=Main_Page
[5] https://www.boost.org/doc/libs/1_59_0/libs/python/doc/tutorial/doc/html/index.html
[6] https://github.com/rogersce/cnpy