CUDA Kernel for Invariant Scattering Transform

Cheng Tang
Courant Institute of Mathematical Sciences
(Dated: June 4, 2019)

This is the final project report for High Performance Computing, Spring 2019. I wrote CUDA kernels that perform invariant scattering convolutional network and outputs scattering coefficients.

I. INVARIANT SCATTERING TRANSFORM

The scattering transform network was established as a theoretical tool to understand the effectiveness of convolutional network that builds large scale invariants that could be used for classification. The scattering network cascades wavelet transforms and non-linear modulus to output scattering coefficients. This transformation is stable to deformation, preserves high frequency information and is tranlational invariant. The computed coefficients output by this transformation could thus be used for classification, as described in Bruna et al.

The architecture of scattering transform network is highly parallelizable, with convolutional operations, modulus and down-sampling operations prevailing through out the architecture, and as its depth increases, the number of operations grows exponentially. This section describes essential components in a scattering network.

A. Wavelet Transform

A wavelet transform filters the signal x with a family of wavelets: $\{x \star \phi_\lambda(u)\}_\lambda$. For two dimensional directional wavelets, $\lambda = 2^j r$ denotes a family of wavelet with scale $|\lambda| = 2^j$ and rotated with angle r.

In my implementation I used the complex Morlet wavelet given by:

$$\phi(u) = C_1(e^{iu\cdot\xi} - C_2)e^{-|u|^2/(2\sigma^2)} \tag{1}$$

Rotated to n_r different angles evenly spaced from 0 to π. In the experiments, $\sigma = 0.85$ and $\xi = \frac{3}{4}\pi$.

Computing wavelet transform of $x \in \mathbb{R}^2$ is essentially performing convolutions of x with each of the rotated filters. 2 dimensional convolution of two square images of same sizes N requires $\mathcal{O}(N^2)$ FLOPS, while circular convolution theorem enables us to do fast convolution leveraging Fourier transform and reduce the complexity to $\mathcal{O}(N \log N)$.

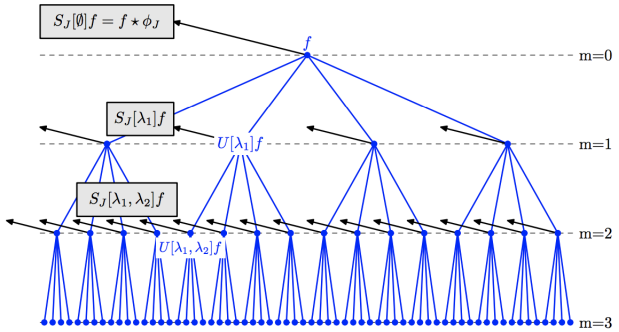
B. Non-linear modulus and averaging operators

The complex wavelet transform commutes with translation and is therefore not translational invariant by itself; to build translational invariant representation of an image x, we apply the non-linear modulus after convolution(compare this with modulus of Fourier coefficients-which are also translational invariant representation of a signal).

Denote $U[\lambda]x = |x \star \phi_\lambda|$, then any sequence of $p = (\lambda_1, \lambda_2, ..., \lambda_m)$ defines a *path* and the series of non-linear operations on the 2-dimensional signal

$$U[p]x = U[\lambda_m], ..., U[\lambda_2]U[\lambda_1]x$$

outputs the *propagatedsignal* at the $m-th$ layer on this path.



(a) Scattering Transform with 3 layers and 4 rotated filters

As recommended by Bruna et al., it is often more useful to use representations that are localized to a scale: thus we like to build local translational invariance and keep translational variability for larger scales. This is done with convolution of the cascaded wavelet transform with a Gaussian filter with a width proportional to the desired scale: this operation takes the average at each position of the propagated signal. It is also reasonable to sub-sample the blurred(convoluted) propagated signal to limit the number of output scattering coefficients while keep enough useful information.

II. PARALLELIZATION

Parallelization are realized using in the CUDA programming language, which enables a high degree of parallelization and is widely popular in the training of deep learning models. Below are discussions of how to implement parallelization and the implementation details.

A. Levels of Parallelization

By the descriptions of the scattering transformation, we can identify the first level of parallelization: the operators apply naturally in parallel at every layer.

Second level of parallelization is on the data set level. Since the (2D) scattering transformations would be applied on large numbers of images, to avoid data transfers overhead between host to GPU, it is desirable to perform the transformation in batches.

Lastly, we can use the cuFFT package to speed up convolution, the most compute intensive and prevalent operation of the scattering transform:

$$f \star g = IFFT(FFT(a) \cdot FFT(b))$$

For floating point operations, a speed up of up to 10 times faster can be gained.

B. Implementation Details

There are many things to consider to implement efficient CUDA kernels: as already mentioned, minimizing number of data transfer between host and device (by batching), memory coalescing (utilizing memory access patterns or shared memory) whenever possible for efficient global memory access, and overlapping memory access and kernel excecution, just to name a few. There are possibilities to optimize the performance of cuFFT as well. In my implementation, I sometimes traded off efficiency with the ease of coding and debugging without degrading performance too much.

I aimed to minimize the number of data transfers to as few as possible: one time to transfer as many raw images as possible to the NVIDIA chip and one time to transfer back all the scattering coefficients after the transformation is complete. To utilize GPU memory efficiently under this condition, it is then important to consider the maximum memory needed throughout the scattering transformation. Since we propagate signals till the deepest layer, the maximum memory scale as $\mathcal{O}((Nx)^m)$ for N images of size x at the m-th layer. In this respect, the optimal thing to do may be to allocate just enough memory for this and overlap transferring the computed scattering coefficients at previous layer with continue transformation at future layers. However, I choose not to do this since this requires to carefully doing in-place operations and may result in deadlocks, and would be hard to debug. Instead, I allocate enough memory to store all the propagated signals in the whole network. This results in an increase of a factor of $\mathcal{O}(1 + \frac{1}{r_n} + \frac{1}{r_n^2} + ...)$ in memory, which is negligible for large r_n(the number of filters), and implementation would be much cleaner.

Applying the same operation on one large batch of images requires writing specific kernel for each operations, and requires some engineering effort. Please see the code for details of the exact implementations. Kernels are written to perform batched 2D-FFT, dot products with complex numbers, etc. to perform the scattering transform.

III. EXPERIMENTS AND PERFORMANCE

All the experiments are done on a Tesla K40c computing processor, with compute capability 3.5 and 15 streaming multi-processors.

The image dataset I am using is the MNIST dataset, which consists of gray scale images of size $28x28$. The memory Images are loaded into the host and copied to GPU in one long 1D array. In the experiment where scattering transform is applied on $50,000$ images, the memory usage was 7.42 GB out of the 10.91 GB on the NVIDIA chip. As the number of images increases, the run-time increases linearly, which is desirable behavior. As the layer of the network increases, the algorithm quickly becomes memory bound. The implementation is scalable with the increase of GPU chips. For practical usage of the scattering transform on 2D signals, each data point(image) should not increase in size so drastically that would degrade the performance. For example, the ImageNet dataset consists of images of size $256x256$. The Complexity of the scattering transform for each image scales as $\mathcal{O}(NlogN)$, dominated by the FFT operation, and we should expect to see an increase of $\mathcal{O}(NlogN)$ with the increase of image sizes. Further experiments are needed for conclusion.

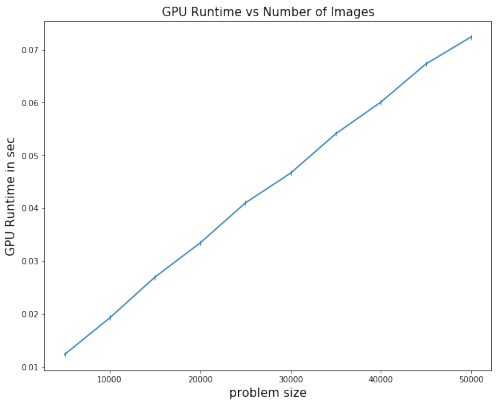


FIG. 2: Runtime of scattering transform with 2 layers and 6 filters for increasing number of images

In general, there are two levels of parallelism to increase the efficiency of the code for GPU programming. One is to increase the instruction level parallelism, the other is to increase occupancy whenever possible.

To compute occupancy of the code, one would check at compile time the registers count and share memory usage using $-Xptxas - v$, and calculate the portion of occupied threads in a streaming multi-processor. This can be done automatically using an online calculator. For

the CUDA kernels I wrote, there's no obvious factors limiting occupancy for each of the kernels.

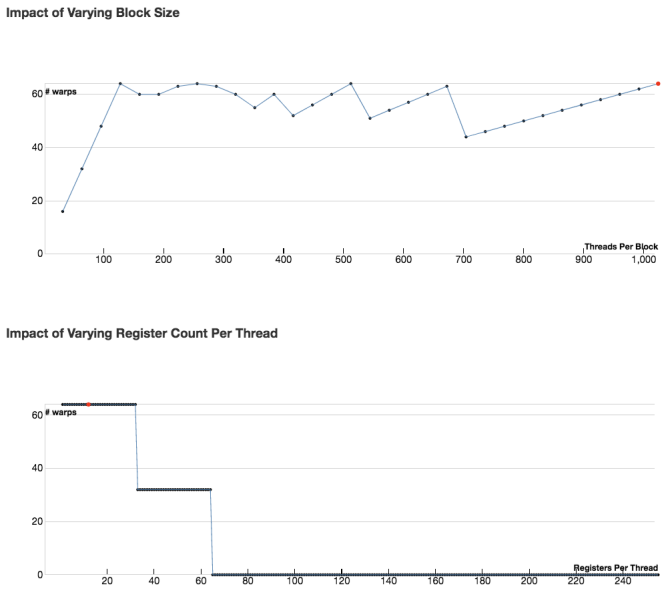


FIG. 3: Result of occupancy for the *rescale* kernel function

IV. CONCLUSION AND FUTURE IMPROVEMENTS

The kernels written to perform each operations of the scattering transform are fairly straight forward, but to achieve high efficiency, it needs to be done with careful planning and require some engineering efforts. For possible future improvements, if the amount of data increases then deploying them to multiple GPU would suffice(higher level SIMD). Different types of filters can also be added for completeness. For application purposes, scattering transform network typically does not need to be very deep or have large number of filters, but if the size of each image increases, then memory storage would always become the problem and the transformation need to be done in smaller batches. For data sets with moderate amount of data points, this implementation could be practical enough.