15-418: Parallel Computing Architecture and Programming
Final Project Writeup

Druhin Sagar Goel(dsgoel)
Siddhant Wadhwa(swadhwa)

Summary:

We implemented a CUDA accelerated image classification pipeline. The image classification was based on the popular bag of words model. We identified two key areas which served as bottelnecks to the runtime of the pipeline: convolution and vector quantization. Using inputs from the two papers (cited in the resources section) we were able to achieve roughly 20x speedup on a GPU CUDA version of the pipeline when compared to a single-threaded CPU version with separable implementation of convolution and naive vector quantization. The pipeline is designed to work on a Mac with a dedicated NVIDIA graphics card.

Background:

Bag of Words Model The bag of words model is a popular model used for image classification in computer vision. It is a pipeline consisting of 3 major parts:

1. Building a dictionary of visual words

   In this part, we extract features from training images using filters/kernels. This is done by convolving these images with a set of filters. This convolution is a step that would benefit from parallelization since naively, it can be parallelized over pixels in an image. We then select some number of points, say α to sample each training image. Thus, from each training image, we get α points. Using these pixels, we cluster similar pixels together using the K-means algorithm. With this, we obtain K pixel centroids which act as our visual words. Each centroid is a an nD-vector where n is the number of filters we used. All these centroids together form our dictionary of visual words.

2. Creating the bag of words

   Then for each training image, we 'apply' the dictionary (created in the previous step) to it, creating a word map for the image. This involves assigning each pixel in each image an integer representing the centroid/word in the dictionary that its closest to. Again, this is a step that would benefit greatly from parallelization. We then create a histogram of word/centroid counts for each image and this is our bag of words. We then use these bags of words to build our classifier (in our case, we use nearest neighbor to classify).

3. Evaluating the recognition system/classifier

   In this part, we use the test images to build word maps and histograms as before. (Again this would benefit from parallelization). We then use the nearest neighbor classifer to classify our test images. This is done by comparing the histogram of test image to all our training images and assigning the test image the class of its nearest neighbor. The nearest neighbor is calculated using the Euclidean distance between the two histograms being compared.

Key Data Structures

Most values are manipulated using OpenCV CV_32F Mat structures, that are basically wrappers around 32-bit floating point 2D arrays.

Key Operations on Data Structures

An image is stored as an OpenCV Mat, and in order to extract descriptors from each pixel, the image is convolved with a large number of filter kernels, forming a std::vector of OpenCV Mat structures that can be pictured as a 3D array, where the depth z, any point (x, y) in the OpenCV Mat is indicative of the response of pixel (x, y) to filter kernel z. (Assuming that the image is single-channel, but our implementation is capable of operating with 3-channel images). The formation of this '3D matrix of feature descriptors is very computationally intensive, and is one of our main target bottlenecks that we streamline and optimize using GPU hardware, that greatly speeds up the training phase of our classifier.

In addition, another very frequently used operation is the calculation of a distance metric (in our case Euclidean) between 2 descriptors (1d arrays with nD dimensionality, where n is the number of filters that the pixel associated to the descriptor was convolved with). We found a paper that outlined a much more efficient and faster vector quantization process that takes advantage of GPU hardware to implement a faster vectorized algorithm to calculate descriptor distance. Given the frequency of the operation, optimizing vector quantization considerably sped up the testing phase of our classification pipeline.

Algorithm's Inputs and Outputs

The algorithm requires a large set of labelled training samples and a large set of separable 2D filter kernels that are used to form feature descriptors. Once trained, given a test image, the pipeline outputs the class label of the nearest quantized (histogram of visual words) neighbor to the test image that it can find in terms of image distance.

Parallelism

Convolution is a data-parallel operation, with high row-wise locality owing to the row-major storage policy of OpenCV Mat data arrays. It is better suited to GPU hardware than SIMD execution given the much larger number of vector lanes that can be used in this context.

Approach:

Technologies, languages/APIs, machines targeted

We used CUDA and C++ in this project. The machine targeted was the Macbook Pro Retina Display with dedicated NVIDIA GPU.

Mapping problem to target parallel machine

We will explain this on a per iteration (of optimization) basis.

1. First iteration of parallelization on the GPU : One CUDA thread per pixel in the input image. This cuda thread is responsible for :

   a. Element-wise multiplying the neighborhood of the pixel with the elements of the kernel.

   b. Summing up these products.

   c. Writing this sum value to the output image pixel at the same coordinates.

2. Iteration 2: Same as the first iteration except that we copy over the block of pixels being convolved over to cuda block shared memory to reduce delays and idle time due to memory latency.

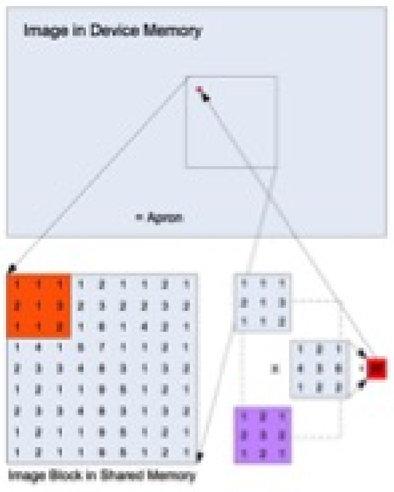
   

   Figure 3: A naive convolution algorithm. A block of pixels from the image is loaded into an array in shared memory. To process and compute an output pixel (red), a region of the input image (orange) is multiplied element-wise with the filter kernel (purple) and then the results are summed. The resulting output pixel is then written back into the image.

   Problems with this approach: Large number of idle threads that are associated with the pixels that end up in the apron. An apron is a perimeter of pixels whose values need to be used to compute the output convolved values of pixels at close to edges of the block. Visually:

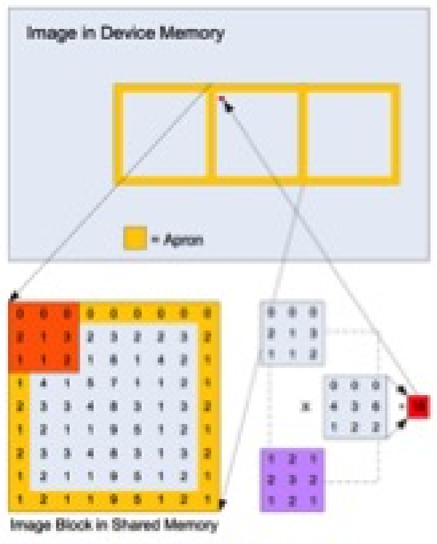
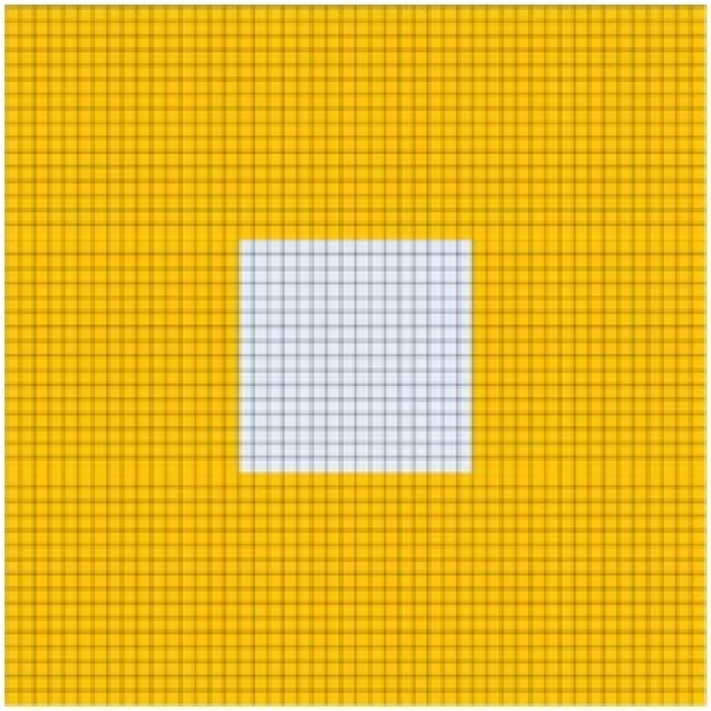
   

   Figure 4: Modification of the naive algorithm of Figure 3 to include the image block apron region.

   Especially when the block size is small and the kernel size is large. For example, when block size is 16x16 and kernel radius is 16 (length of kernel side is 33):



All the yellow pixels here are associated to threads (since we decided on a one-to-one mapping between input image pixels and cuda threads) that sit idle while the white pixels in the middle (16x16 block) undergo convolution. Hence, vector utilization is $\frac{1}{9}$.

3. Iteration 3: Now we break the one-to-one mapping between pixels of the input image and cuda threads.

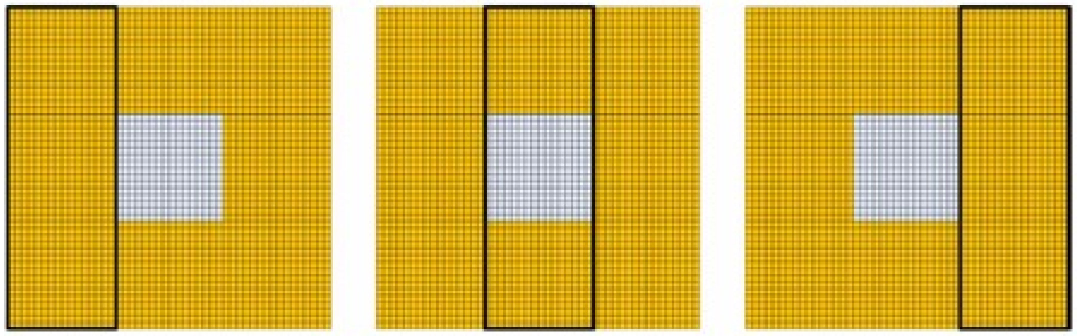


Figure 6: Reduce idle threads by loading multiple pixels per thread.

In fact, in the example provided earlier, when block size is 16x16 and kernel radius is 16, if we loaded the apron pixels in windows of 16x16 using only the 16x16 CUDA threads that will also be responsible for the computation of the 16x16 pixel block colored in white, there will be no idle threads when the actual convolution arithmetic takes place.

4. Iteration 4: Used the algorithmic advantage of separable filters in parallel over cuda threads. (Separability of filters is explained below).

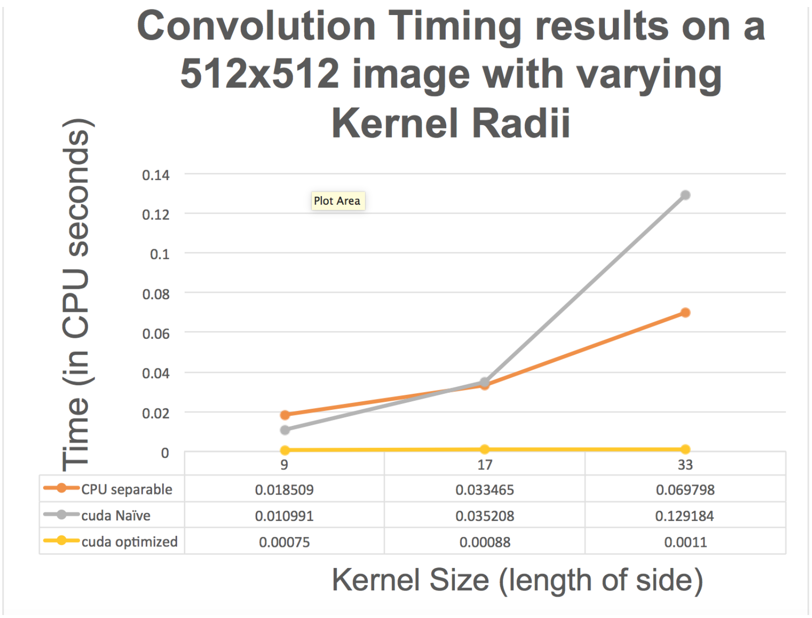
Changing the original serial algorithm to enable better mapping to a parallel machine

We exploited the property of separability of filter kernels to reduce the complexity of convolution from mxn to $m+n$ where (m, n) are the dimensions of the filter kernel.

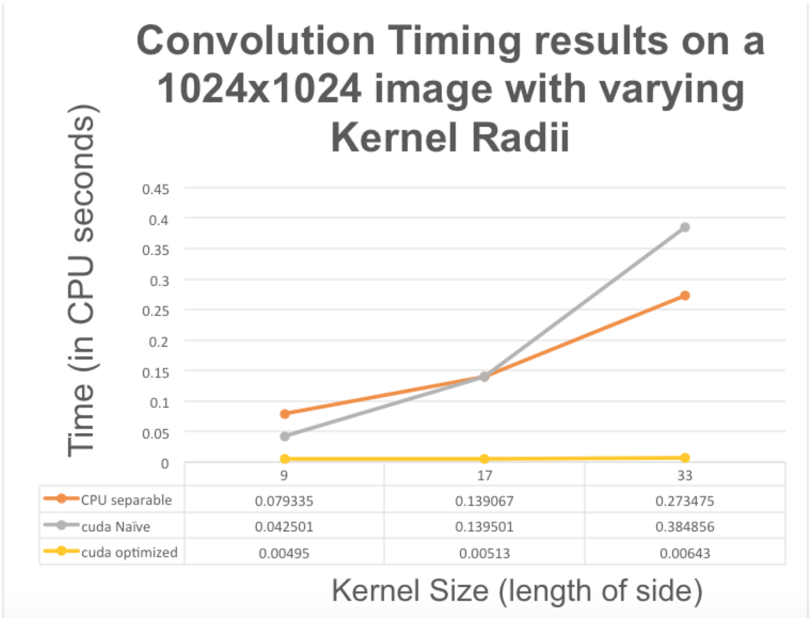
$$\text{Applying } \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ to the data is the same as applying } \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \text{ followed by} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}.$$

Results:

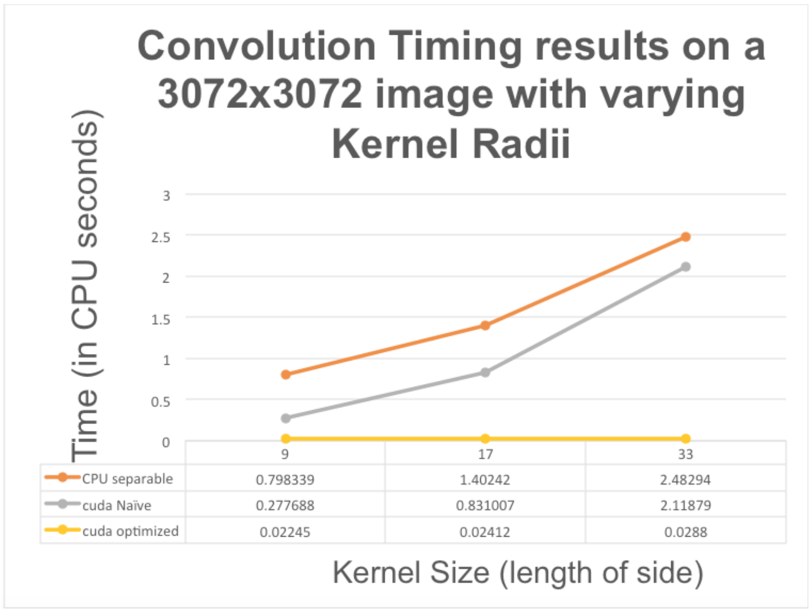
The following are runtime results we obtained on convolution optimizations:



	9	17	33
CPU separable	0.018509	0.033465	0.069798
cuda Naïve	0.010991	0.035208	0.129184
cuda optimized	0.00075	0.00088	0.0011

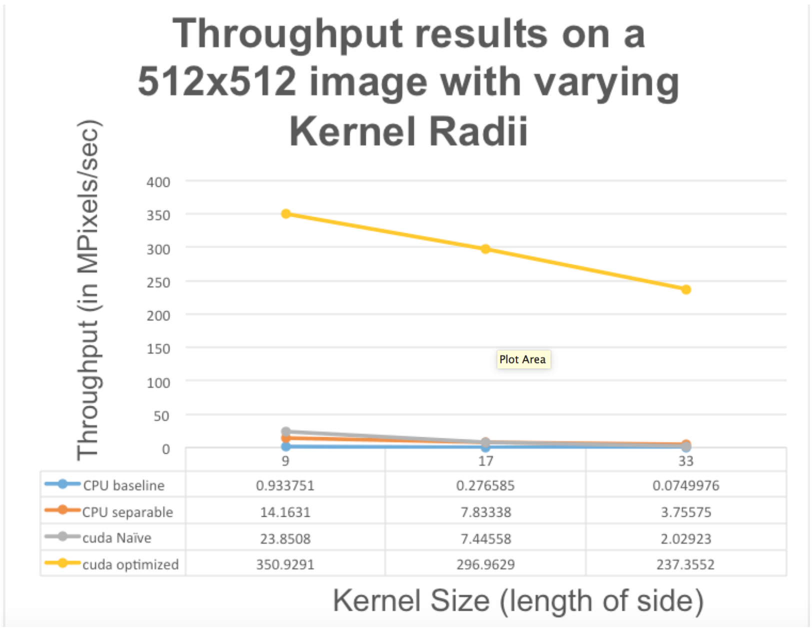


	9	17	33
CPU separable	0.079335	0.139067	0.273475
cuda Naïve	0.042501	0.139501	0.384856
cuda optimized	0.00495	0.00513	0.00643

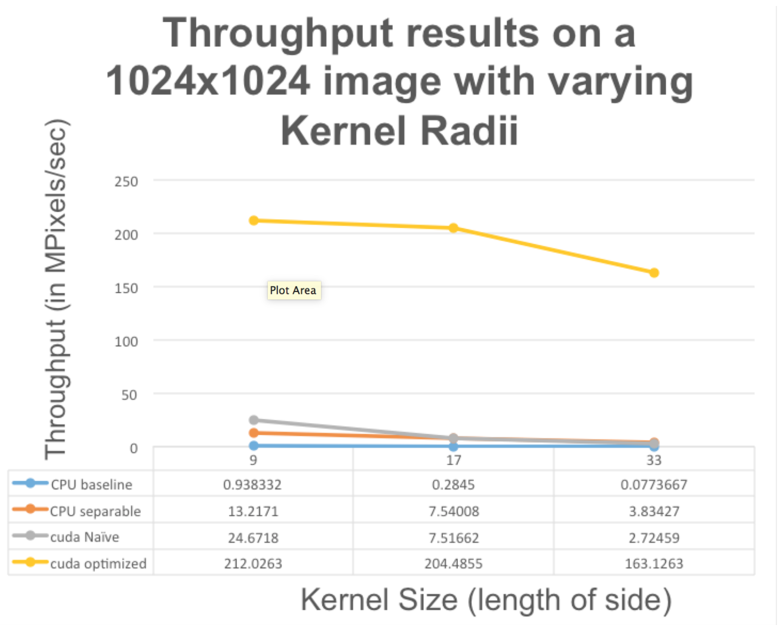


	9	17	33
CPU separable	0.798339	1.40242	2.48294
cuda Naïve	0.277688	0.831007	2.11879
cuda optimized	0.02245	0.02412	0.0288

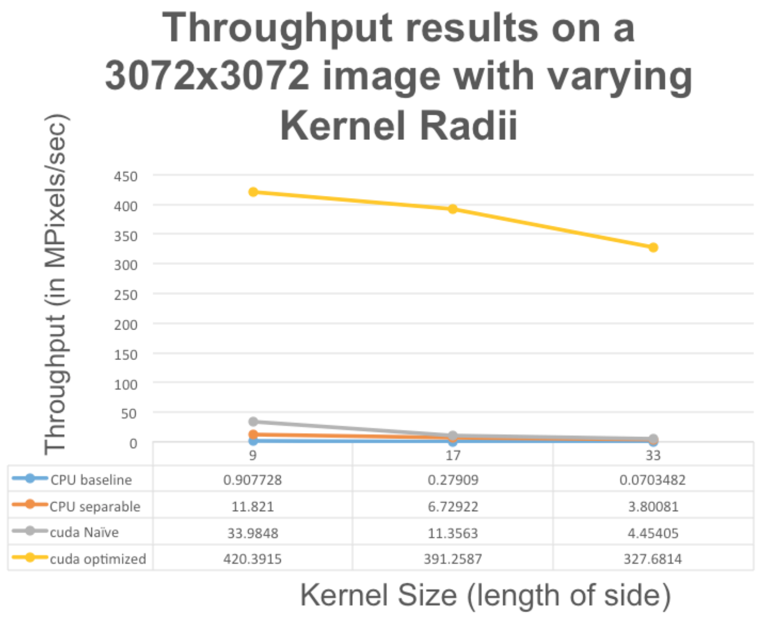
The following are throughput results we obtained on convolution optimizations:



	9	17	33
CPU baseline	0.933751	0.276585	0.0749976
CPU separable	14.1631	7.83338	3.75575
cuda Naïve	23.8508	7.44558	2.02923
cuda optimized	350.9291	296.9629	237.3552

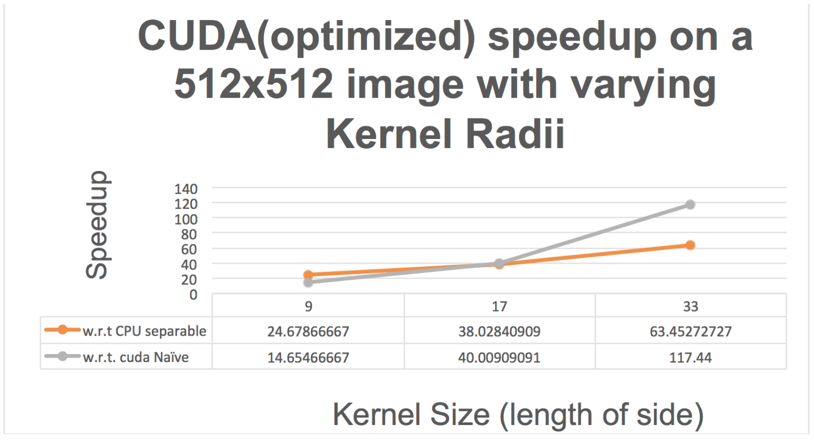


	9	17	33
CPU baseline	0.938332	0.2845	0.0773667
CPU separable	13.2171	7.54008	3.83427
cuda Naïve	24.6718	7.51662	2.72459
cuda optimized	212.0263	204.4855	163.1263

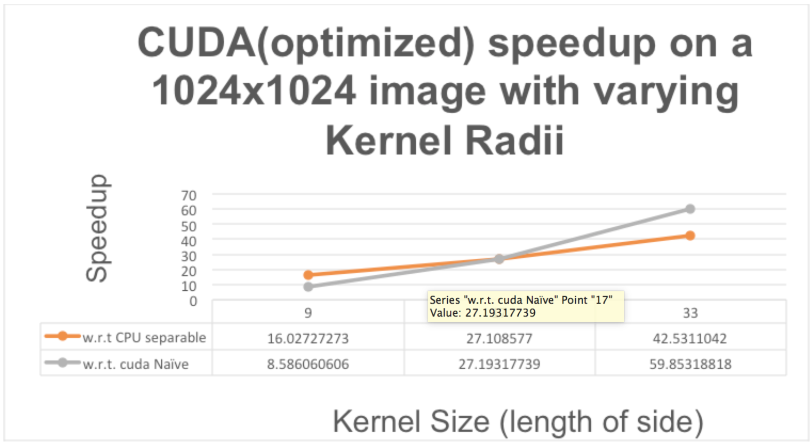


	9	17	33
CPU baseline	0.907728	0.27909	0.0703482
CPU separable	11.821	6.72922	3.80081
cuda Naïve	33.9848	11.3563	4.45405
cuda optimized	420.3915	391.2587	327.6814

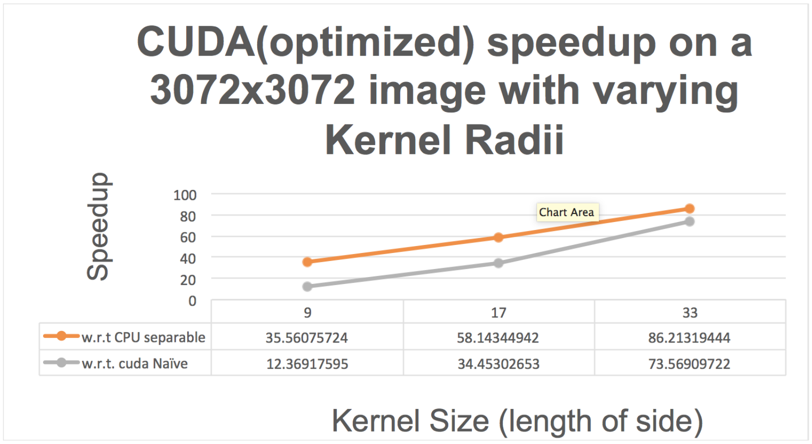
The following are speedup results we obtained on convolution optimizations:



	9	17	33
w.r.t CPU separable	24.67866667	38.02840909	63.45272727
w.r.t. cuda Naïve	14.65466667	40.00909091	117.44

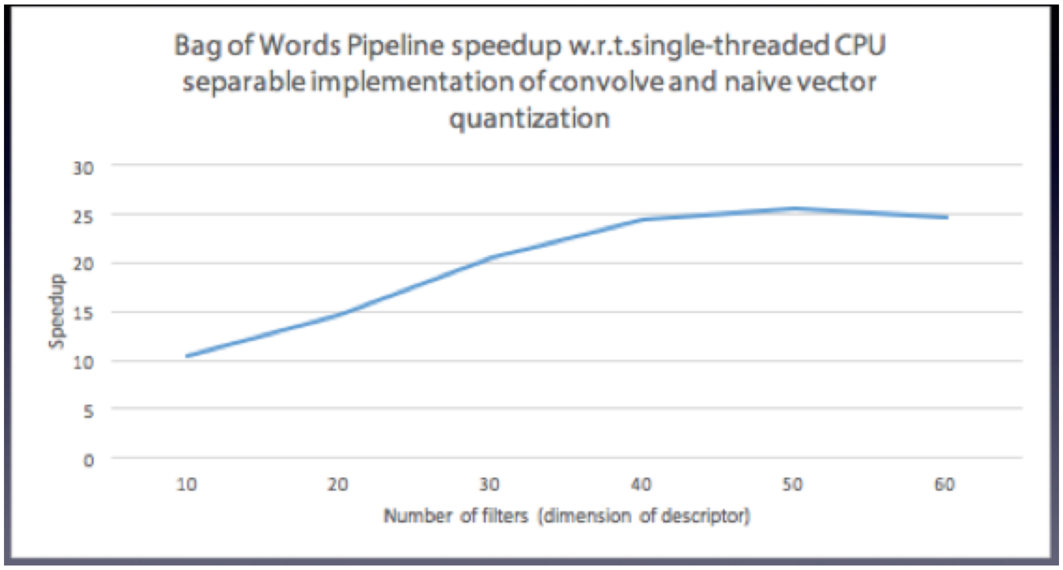


	9		33
w.r.t CPU separable	16.02727273	27.108577	42.5311042
w.r.t. cuda Naïve	8.586060606	27.19317739	59.85318818



	9	17	33
w.r.t CPU separable	35.56075724	58.14344942	86.21319444
w.r.t. cuda Naïve	12.36917595	34.45302653	73.56909722

Finally, the speedup we obatined on the entire pipeline was as follows:



References:

1. http://igm.univ-mlv.fr/ biri/Enseignement/MII2/Donnees/convolutionSeparable.pdf
2. http://koen.me/research/pub/vandesande-itm2011-VisualCategorizationGPU.pdf