CUDA implementation of a biologically inspired object recognition system

Sharat Chikkerur

Abstract

The HMAX model is a biologically motivated architecture for computer vision whose components are in close agreement with existing physiological evidence. The model is capable of achieving close to human level performance on several rapid object recognition tasks. However, the model is computationally bound and has limited engineering applications in its current form. In this work, we wish to increase the computational efficiency using a two-fold approach: (i) We present several filtering approximations that improve the algorithm itself and (ii) We make use of the inherent parallelism in the computation using SIMD capabilities of the CUDA platform. We compare the performance both in terms of accuracy and timing with the baseline matlab implementation and also a multi-threaded C implementation. This work will outline the benefits of generic multi-threaded programming vs. multi-core SIMD computation.

1 Introduction

The HMAX model is a biologically motivated model that mimics the organization of the first few stages for the human visual pathway. More specifically, it tries to explain the purely feed forward mechanism for rapid object recognition that occurs in the first 100-150ms of visual processing. The model is organized with alternating layers consisting of simple(S) units and complex(C) units (See Figure 1). S units have specific preferred input patterns (such as oriented lines) and perform simple pattern matching. Complex units spatially pool the outputs from the S units and select the maximum, thus providing spatial invariance. Complexity of the preferred input to the S units and the degree of location and scale invariance achieved at the C units increases as we move up the hierarchy. The model requires very few training examples and competes with the state-of-the-art system in object recognition task. A detailed description of the HMAX model can be found in [2]. While the HMAX model is surprisingly accurate and a biologically faithful, it is currently bounded by computation and is not very useful for real-life application. In this report, we study a specific instantiation of the model, with two S,C layer pairs [3] that simulate visual processing in area V1/V2-V4/PIT.

1.1 Computational Complexity

- **S1 layer**: The S1 layer computation consists of performing normalized cross correlation between a bank of 64 gabor filters (16 scales and 4 directions and the image. This operation is non-linear and has a computational complexity of $O(N^2M^2)$, where $M \times M$ is the size of the filter and $N \times N$ is the size of the image.
- **C1 layer**: The C1 layer is computed by performing a spatial and scale-level max pooling operation on the S1 outputs.The computational cost of this operation is $O(N^2M)$, since max operation can be performed in a separable fashion (x direction first and then y direction). In the current instantiation, the C1 layer consists of 8 bands of pooling.
- **S2 layer**: During training, several hundred C1 patches of different sizes are randomly selected from the training images and serve as prototype features that represent a single object (class specific dictionary)

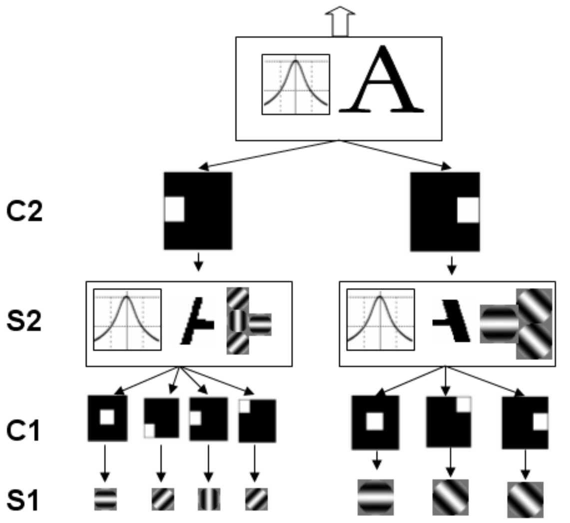

Figure 1: HMAX Model Architecture. S1 layers consists of simple gabor filter at several scales and orientation. C1 pools the filter outputs spatially and across adjacent scales. S2 is tuned to conjunctions of orientations. C2 provides further spatial and scale invariance. The C2 outputs are directly fed to a classifier

or all objects (universal dictionary) [3]. Each S2 unit is designed to detect the presence of the prototype within the C1 unit responses. Each patch serves as a filter that is correlated with the 8 C1 bands to generate a tuning response map. Deriving the S2 unit outputs is the most computationally intensive part of the model, involving $O(PN^2M^2)$ computation. Here P represents the number of patches.

- **C2 layer**: The C2 layer provides the final invariance by detecting the maximum response of the corresponding S2 units over all scales and positions. There are as many C2 units as the number of patches in the representation dictionary. The computational complexity of this step is $O(N^2MP)$.

1.2 Parallelism

It can be seen from the previous section that while the algorithm is serial(feed-forward) in nature, each 'S' stage requires computing responses of several filters on the same input. Each of the responses can be computed independently, making it *embarrassingly* parallel in nature. We attempt to improve the performance of the algorithm on two fronts. Firstly, we attempt at improving the theoretical efficiency of the algorithm itself. Secondly, we attempt to improve the computational performance by using parallel execution. In embarrassingly parallel applications these provide independent speed-ups.

1.2.1 Algorithm speedup of the S1 layer

The S1 layer is obtained by performing pattern matching with short lines at several different scales and orientations. The oriented lines are modeled as anisotropic Gabor filters given by,

$$G(x,y) = \exp\left(-\frac{u^2+\gamma v^2}{2\sigma^2}\right) \times \cos\left(\frac{2\pi}{\lambda}\right) \quad (1)$$

$$\text{where,} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (2)$$

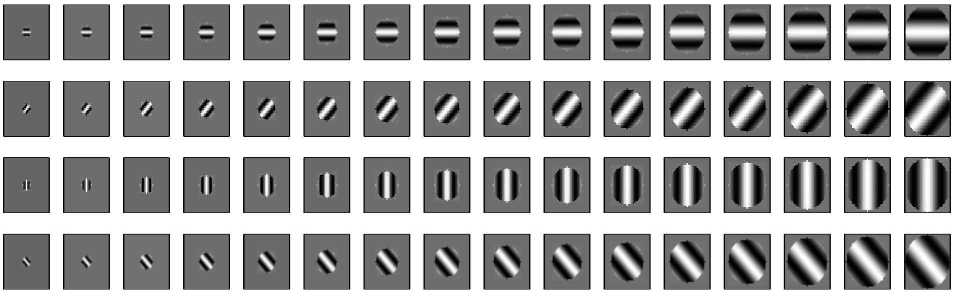

Figure 2: Bank of gabor filters used to compute S1 features. The receptive fields sizes and other parameters are outlined in [3].

Here γ represents the eccentricity/anisotropy of the filter, θ is the orientation and λ is the wavelength of the filter. S1 unit response $S1(s,d)$ at scale s and direction d is computed using a normalized cross correlation operation

$$S1(x,y) \quad = \quad \frac{\hat{G}(x,y,\theta) * I(x,y)}{I^2(x,y) * H(x,y)} \tag{3}$$

Here $\hat{G}$ is the zero mean,unit normal version of the filter and $H(x,y)$ a filter with all ones and the same size as the Gabor filter. The responses at different scales are obtained by performing normalized convolution with filters of different sizes as shown in Figure 2.

However, it is to be noted that in the next stage (C1), the filtered output are sub-sampled after the max operation, with the range of max-pooling being proportional to the filter size. Mathematically, the input to the next stage from a filter of scale 's' is

$$S1_s(x,y) = \Big(\hat{G}(x/s, y/s) * I(x,y)\Big) \downarrow s \tag{4}$$

This can be simplified to

$$S1_s(x,y) \propto \hat{G}(x,y) * (I(x,y) \downarrow s) \tag{5}$$

Thus, instead of convolving the image with increasingly larger filter (computational complexity of $O(N^2M^2)$, where M is the filter size), we reduce it to $O(N^2/s^2M^2)$ at scale S, saving considerable computational time. The timing comparisons are shown in 3.

2 C++ Parallel Implementation

We implemented the algorithm in C++ using open source OPENCV library to implement image processing primitives. As a common strategy to parallelize the algorithm, the output of each filter in the S1 and S2 stages are computed in parallel. Similarly, the max-pooling at each C unit is also computed in parallel. The parallelism is implemented by the use of threads. Multi-thread programming is convenient, since all the threads share the same address space and communication among threads is transparent, but still requires explicit synchronization. The performance of the multi-threaded program automatically scales with the increasing number of processors in a computer. Approaches such as MPI use multiple-processes that exist in isolated address spaces and requires explicit communication and synchronization. The speed gain obtained from the muluti-threaded implementation is shown in Figure **??**.

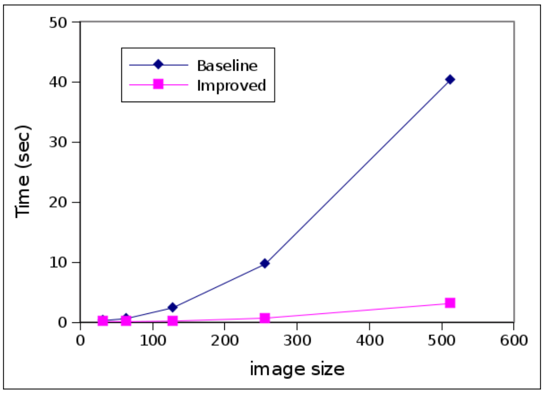

Figure 3: The graph shows the time required to compute S1 on images of different sizes. It shows the matlab baseline algorithm that uses filters of different sizes vs. the one that uses image pyramids and a fixed filter size. It is to be noted that both use pure matlab code.

3 NVIDIA CUDA Implementation

We also implemented the algorithm to run on a NVIDIA GTX8800 device. The NVIDIA CUDA framework allows us to use the GPU (graphics processor unit) as a general purpose computing device. In the following we describe the device architecture and our implementation in detail.

3.1 Device Architecture

The NVIDIA CUDA [1] capable family of graphics processor units allow us to access the highly parallel processors for general purpose (SIMD-type) computation. Here we contrast the GPU architecture with that of the CPU.

- Existing CPUs that have in the order of 10s of computing cores (PS3 has 8+1 core, Tirela chips have 64 compute cores). In contrast, CUDA enabled GPUs have in the order of 100s of compute cores (GTX 8800 has 96 compute cores).
- CPUs can spawn several 10s of threads each of which can share the same address space. In contrast, each processor in the GPU can spawn several 100s of threads. Each of the threads share a slower(but larger) global memory space and a fasther (but smaller) shared memory.

3.2 Execution Model

In a traditional multi-threaded programming model, multiple threads operate independently of each other. However, the NVIDIA GPU follows a slightly different programming model. The code to be executed in the device is called a **kernel**. The GPU can execute only one kernel at a time. However, the code can be executed simultaneously on all the cores in paralle. Further, each kernel is divided into a **grid** of **blocks**. Each block in the grid has access to a fast (but small) amount of shared memory. Each block is further composed into an array of **threads**. Typically, each block invokes 100s of threads. The kernel has access to information that indicates which block and which thread is currently under execution. A common strategy for programming is to split the problem into blocks and execute each kernel block in parallel.

3.3 Memory Hierarchy

When programming for the CPU, most programming languages allow transparent access to memory abstracting the memory hierarchy. However, when programming for the GPU, the programmer has the ability to directly address any part of the memory hierarchy and in fact is encouraged to consider the hierarchy to optimize the running time. The GPU memory is divided into

- **Register:** Each core in the GPU has access to 128 registers that are local. This serves as the fastest read write location (one cycle) available to each thread.
- **Shared:** Threads in each block can access a small about (16 Kb) of shared memory that is several times faster than the global (1Gb) memory. A common strategy in GPU programs is to stage the data in shared memory, use each thread to operate on a separate part of the shared memory and then copy the results back into main memory.
- **Global:** The GPU has a large amount (1 Gb) of main memory that can be accessed by all grids, blocks and threads. However, reading and writing to this area is slow requiring several hundred cycles. This cost can be amortized if the computational cost is much greater than the communication cost.
- **Constant:** This memory is read only, but can still be access across all grids.
- **Texture:** Texture memory is optimized for spatially local memory access (common in graphics and image processing). It provides a much faster read-only access when compared to global memory. Further, memory regions bound to a texture can be accessed in normalized co-ordinates.

3.4 Implementation Strategy

In this section, we outline our strategy for implementing different layers within the model.

3.4.1 S1 Layer

Computing the S1 layer involves convolving a pyramid of images (scaled to a constant factor) with a filter-bank (of size four in our case). We parallelize this computation by iterating over different sizes of images and assigning each filter to a different block. Within each block, the rows of the image are equally divided among the different threads.

3.5 C1 layer

Computing the C1 layer involves computing maximum in a local neighborhood across all the layers of the S1 (four in our case, corresponding to the four filters). We parallel this computation by iterating over the different image sizes and assigning each layer/dimension of the S1 unit to a different block. Within each block, the rows of the image are equally divided among the different threads.

3.6 S2 layer

The computation is similar to the S1 layer except for the fact that both the input and the filters are now three dimensional. We parallelize this computation by iterating over C1 inputs of different sizes and assigning each filter to a different block. It is to be noted that unlike the S1 layer, the S2 layer is computed by measure similarity of the C1 layer with several hundred patches/filters instead of just four as in the S1 layer. Thus, not all the blocks are executed in parallel by the GPU. However, the scheduling is handled by the CUDA framework and is transparent to the programmer.

	C1				C2		
size	MATLAB	CPU	GPU		MATLAB	CPU	GPU
64	0.41	0.05	0.03		2.41	0.53	0.17
128	0.79	0.46	0.07		4.57	1.31	0.33
256	2.53	1.13	0.16		12.18	7.91	0.72
512	10.39	5.14	0.46		45.56	17.91	1.81

Table 1: Left: The table shows the time required to compute C1 across the various platforms. All the numbers are in seconds. Right:The table shows the time required to compute C2 on images of different sizes across the various platforms.i Both the matlab and multi-threaded versions were run on a 3.0 GHz dual core machine with 2GB main memory. It can be seen that the speedup is less dramatic than at C1 since most of the time is spent in computing S2. However, the CUDA version of the algorithm still gives us a large improvement in speed.

3.7 C2 layer

Computing the C2 layer involves taking the global maximum across all sizes(bands) in each dimension of the S2 units. For simplicity, we compute the C2 layer entirely on the CPU, since the communication and initialization cost would outweigh the computational benefit obtained by transferring it to the GPU.

4 Conclusion

The HMAX/CBCL hierarchical object recognition algorithm is limited in its application due to its computational complexity. In this report, we showed speed-ups due to changes in algorithm and implementation. We show that both the multi-threaded and GPU versions give considerable speedup. We discussed the GPU architecture, execution model and memory hierarchy and showed that while the programming model offers lesser abstraction, it gives us far more computational benefit that the multi-threaded CPU implementation.

A Code

The code can be obtained from the following links:

- Matlab version: http://code.google.com/p/cbcl-model-matlab/
- Multi-threaded version: http://code.google.com/p/cbcl-label-server/
- CUDA version: http://code.google.com/p/cbcl-model-cuda/

References

[1] Nvidia cuda(tm). http://www.nvidia.com.

[2] T. Serre, M. Kouh, C. Cadieu, U. Knoblich, G. Kreiman, and T. Poggio. A theory of object recognition: Computations and circuits in the feedforward path of the ventral stream in primate visual cortex. *MIT AI Memo*, 036, 2005.

[3] Thomas Serre, Lior Wolf, and Stanley Bileschi. Robust object recognition with cortex-like mechanisms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2006.