OPTIMIZING A NEURAL NETWORK FOR GRAPHICS PROCESSORS WITH THE NVIDIA CUDA FRAMEWORK

TREVOR BARRON
ADVISER: MATTHEW WHITEHEAD

Abstract. Improvements in the field of machine learning will come from the ability to process more data not from the development of more complicated algorithms. Since GPUs are optimized for extreme data-parallel processing they can provide significant speed-ups in processing time when compared to an analogous CPU version. As additional data readily accumulates, 10 weeks of training time is not unheard of but achieving the speed-up that some of researchers have reported would reduce a CPU running time of 10 weeks to less than 1 day on a GPU! This paper describbes an implementation of a neural network for NVidia GPUs using the CUDA framework. It currently achieves a speed-up of greater than 65x in the back propagation algorithm and greater than 90x in the feed forward algorithm when run on large inputs. It handles datasets that do not fit GPU memory by monitoring its memory usage. By using multiple compute streams, it overlaps various computations to use GPU resources as effectively as possible. The paper discusses several methods of optimization and presents a test case using the data intensive learning task face recognition.

1. Introduction

There has been a recent shift toward using graphics processing units (GPUs) for extremely data-intensive applications, as they can be 100s of times faster than central processing units (CPUs). As an interesting acknowledgment of the speed of GPUs, Intel recently released a statement that the NVidia GPU is only 14x faster than the Intel CPU. NVidia commented on the rarity that a competitor would make the claim that its product was only an order of magnitude slower.

NVidia is known for its graphics cards but has put significant effort into popularizing the general purpose GPU (GPGPU) programming model that encourages the use of GPUs for additional applications. GPUs are seen strictly as a powerful computing resource. In the last several years NVidia has been promoting its Compute Unified Device Architecture (CUDA) framework, which simplifies the use of GPUs for applications other than graphics and abstracts graphics-specific hardware or software details. In this paper I will give a quick overview of NVidia GPU architecture, briefly describe the CUDA programming model, and describe my implementation of a neural network in CUDA along with a test case.

2. Why GPUs?

Graphics processors are capable of highly parallel computation. They can handle 1000s of threads simultaneously. Through splitting up a data-parallel problem, one in which each thread operates on its own small piece of a large dataset, speedups of nearly 100x can be achieved. CPUs are optimized for low latency but tend to have low throughput. On the other hand, GPUs are optimized for high throughput but generally have higher latency as well; the emphasis is on running many threads more slowly, as compared to a single CPU thread that must run very fast (the clock speed on the a NVidia GTX 660 GPU is 980 MHz compared to 3 GHz or higher on some of today CPUs). By running 1000s of threads simultaneously, however, the GPU can hide this latency by changing which threads are active when a memory access is needed.

3. The CUDA Framework

NVidia has provided the CUDA framework to facilitate programming on the GPU. CUDA makes handling 1000s of threads and executing functions on the GPU relatively easy. The framework defines a memory model and a programming model (Figures 1, 2).

3.1. **Memory Model.** CUDA makes it possible to access several different levels of memory (Figure 1). I used three of the different memory types in my application. Registers are thread-specific memory. Shared memory is specific to a thread block; all threads within the same block can access the same shared memory. Global memory is accessible by any thread in any block, but is on the order of 100x slower than shared memory and registers. As a result, it is common to have each thread load a small piece of data into shared memory where multiple threads can quickly read several elements as needed.

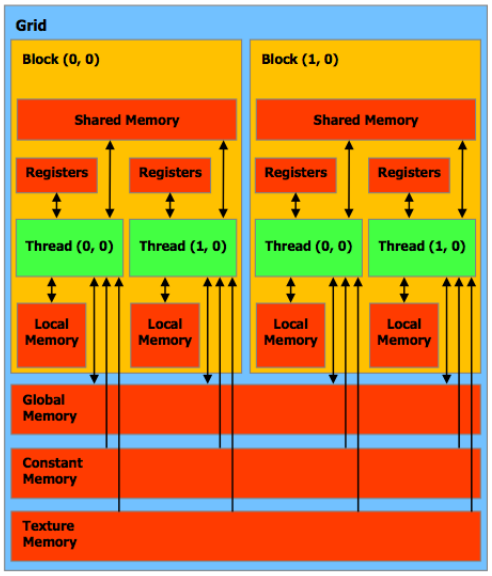

FIGURE 1. The CUDA memory model. Global memory is about 100x slower than shared memory and registers but is accessible from any thread.

3.2. **Programming Model.** The CUDA programming model (Figure 2) lets users define kernels. A kernel is essentially a function that runs on the GPU with many threads simultaneously. Each kernel execution is organized into a grid of blocks. Each block contains a certain number of threads. Both the grid size and the block size are user-defined and are generally calculated based on the size of the input.

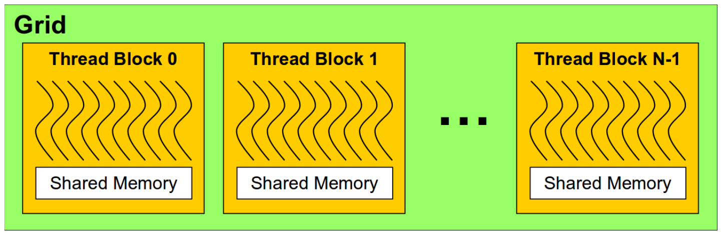

FIGURE 2. The CUDA programming model. A kernel is initialized to run using a certain number of thread with are distributed across several blocks. The blocks are organized into a grid.

4. Neural Network Overview

A neural network is a learning algorithm that uses large datasets of input and output pairs to approximate a function relating the input and output. The neural network structure can be thought of as a graph with an input layer of nodes and a hidden layer of nodes connected to the input layer by a series of weights. The same relationship then exists between the hidden layer and the output layer. It has been proven that such a structure is capable of representing any real-valued function. This suggests that a neural network is capable of representing any arbitrarily complex function, such as a function linking an image of a face to the name of the person depicted.

There are two main operations associated with neural networks: feedforward and backpropagation. The feedforward algorithm takes an input and computes the corresponding output by running the input through the network. This entails computing the values of every node in the hidden layer given the inputs and the weights between the input layer and hidden layer, and then repeating the process to compute output values using the newly computed hidden layer values and the weights between the hidden and output layers.

The sigmoid function is used here as an activation function. Given the linear combination of values and weights for a given node, the sigmoid function produces the output for that node, its activation value. The backpropagation algorithm takes the output value produced from feedforward and compares it to the expected output for the given input. The algorithm then uses the error and the partial derivative of the error with respect to the weights to determine how to update the weights in order to improve classification. The process can be thought of intuitively as searching a multidimensional surface via gradient descent to find a minimum and optimize the function.

5. GPU Neural Network

In order to achieve any speedup from a GPU implementation we need to parallelize the computation. In the feedforward algorithm all the node calculations for nodes in the same layer can be parallelized. Similarly, in backpropagation the gradients and weight update values can be computed simultaneously. The layers themselves, though, are dependent on one another. During feedforward, for example, layer 1 must be computed before layer 2 since layer 2 relies on the computed values from layer 1. In backpropagation, too, the gradients for the output layer must be computed before the gradients for the hidden layer. The weight update and gradient calculation are not dependent from layer to layer. As soon as the gradients for the output layer have been calculated, the gradients for the hidden layer can be computed and the weight update can be executed in a different stream in parallel. These restrictions require that some additional synchronization be added.

6. CUDA Optimization Strategy

I made several different GPU-specific optimizations to my neural network algorithm. I list them in order from the most significant to least significant for my project.

6.1. **Global memory coalescing.** The CUDA platform intelligently batches memory accesses if they are in the same 64-byte block of memory. If the threads in a block access memory within the same block of memory, those accesses will be coalesced and only one memory retrieval will be necessary. If the threads access a different block in memory, the accesses cannot be coalesced and a different access is needed for each retrieval. Since each access to global memory is relatively time-consuming and threads could make several accesses to global memory within a single kernel, memory coalescing results in significant speedups. In my implementation the weights and deltas are stored in a one-dimensional array. This lends itself to two different orderings as shown in Table 1. Depending on which format is chosen, either feedforward or backpropagation will have some strided access. Experimentally I determined that it is more efficient to give feedforward the strided access and give backpropagation the simple access with adjacent indexes in memory. This change provided approximately 5x speedup via Option 2 in the Table 1.

Option 1	$i_0j_0, i_1j_0, i_2j_0, \cdots, i_nj_0, i_0j_1, i_1j_1, i_2j_1, \cdots, i_nj_1, \cdots, i_nj_m$
Option 2	$i_0j_0, i_0j_1, i_0j_2, \cdots, i_0j_m, i_1j_0, i_1j_1, i_1j_2, \cdots, i_1j_m, \cdots, i_nj_m$

Table 1. Two possible ways to organize weights in memory as a single dimensional array. i is the node in the first layer. j is the node in the next layer. There are n nodes in the first layer and m nodes in the second layer. i_0j_0, for example, represents the weight from node i to node j.

6.2. **Proper GPU utilization.** Utilizing all of the GPUs resources is vital for efficient code. The NVidia GTX 660 GPUs that I have in my desktop have 5 streaming multiprocessors (SM), each with 192 CUDA cores. Each SM can operate on only 8 blocks at a single time. If more than 8 blocks are created for a function, the blocks will iterate. The number of threads in a block should always be a multiple of the warp size, or the number of threads that can be run at a single time, so that the hardware optimized for powers of 2 is fully used. It is difficult to determine exactly how much of a difference this makes. One of the feedforward kernel versions that I implemented relies on a block size equal to the number of nodes in the next layer, with the number of threads per block equaling the number of nodes in the previous layer. This version is especially fast since it sums the linear combination of values contributing to the next node in parallel as well. But as the network grows large this version actually becomes slower than a version that does the sum sequentially but uses as many full thread blocks as possible. As the network becomes large, there is large overhead involved with allocating a block for every node in the next layer.

6.3. **Batched memory transfers.** When I first implemented a CPU neural network version, I loaded the dataset into a very nice object-oriented format. Every pattern was contained in an object with an array for both the input and output. This made it very easy to access certain patterns and their corresponding inputs and outputs, but when the time came to copy the dataset to the GPU this structure added complexity. First the objects would have to be copied to the GPU, then each input and output, and then the pointers connecting the two. Have 5 memcpy calls added significant overhead. Flattening the data structure to a single-dimensional array provided a 2.17x increase in copy throughput to the GPU. Since the entire dataset fit in GPU memory in all my tests, this copy is needed only once at the very start of the program, and flattening the array does not provide further speedup.

6.4. **Compute-copy overlap.** My GPUs have only 2 GB of memory. In order to process a dataset larger than 2 GB I would have to transfer memory in and out of the GPU. But memcpy calls have high overhead and copying about 1 GB of data to the GPU could take up to half a second. Consider a dataset that needs to be split into two sections before fitting in memory. Then each epoch requires two memory transfers, which would entail approximately one second of memcpy time per epoch. If we have a complex dataset that takes 10,000 epochs to learn well we will have spent nearly 10,000 seconds copying data or about 2.75 hours! Clearly, this is an important optimization with a large dataset. By dedicating a stream to copying data to the GPU we can overlap the copy and and computation. If we allocate two buffers on the GPU to hold data we can copy to one while doing computations on the other. When the computation has finished, more data are already on the GPU ready to be processed. The GPU can begin processing on the other buffer and new data can be copied to the previous buffer. The GPUs have a designated copy engine that handles copies to and from the device so that important processor time is not spent copying data. This optimization was not very significant for my project because I was not working with datasets larger than 2 GB.

6.5. **Multiple compute streams.** By default every function executed on the GPU runs in a single stream. Computation is parallelized within this stream. But what if we have multiple independent functions and want to run them simultaneously? Executing functions in different streams will enable them to be run at the same time. In the feedforward algorithm, no function calls can be overlapped because the layers are dependent on each other. But in the backpropagation function the current error is calculated in a different stream while the gradients are calculated. As soon as the gradients and deltas have been calculated for the output layer, the weights between the hidden and output layer are updated in a separate stream while the gradients for the hidden nodes and the deltas from the input layer to the hidden layer are calculated. However, this optimization provided a very small speed increase in my tests. Since all my test cases required only a few output nodes, the error calculation and the weight updates between the hidden and output layers were very fast. As a result, using additional streams in this case saves only milliseconds per epoch even on a very large network.

6.6. **Multiple GPUs.** Finally, we can speed up computation by splitting the work across more processors. If we use a batch learning algorithm, identical copies of a neural network can be initialized on multiple GPUs. Then a section of the dataset can be processed by each GPU. At the end of each epoch the deltas on each GPU can be summed and the new weights for the network can be synchronized across all GPUs. The CUDA API provides an interface to transfer data directly between two GPUs on the same system through the PCI bus without passing through the CPU first. This dramatically increases bandwidth from 6 GB/sec up to a peak of 12 GB/sec. There is still overhead resulting from copying data back and forth, but as the dataset becomes very large, the time to copy becomes relatively insignificant compared to the time for a single epoch.

7. Results

The following discussing presents the results of my optimization.

7.1. **Feedforward.** I experimented with two different methods to optimize the feedforward algorithm. One implementation is very efficient on relatively small inputs of fewer than 512 nodes. This version computes every term of the linear combination in a separate thread and then computes the sum of the linear combination in parallel using an optimized reduction algorithm. As the input grows this method becomes inefficient because the number of thread blocks being initialized is dependent on the number of nodes in the following layer, which becomes very large. My second implementation uses the GPU hardware very efficiently. It chooses how many blocks to allocate based on the number of nodes to calculate. It is very difficult to do a parallel sum, however, when the computations are split across different thread blocks. As a result this version calculates each node in the next layer in a different thread. Figure 3 compares the time for 100 iterations of feedforward for the CPU version and the optimal GPU version.

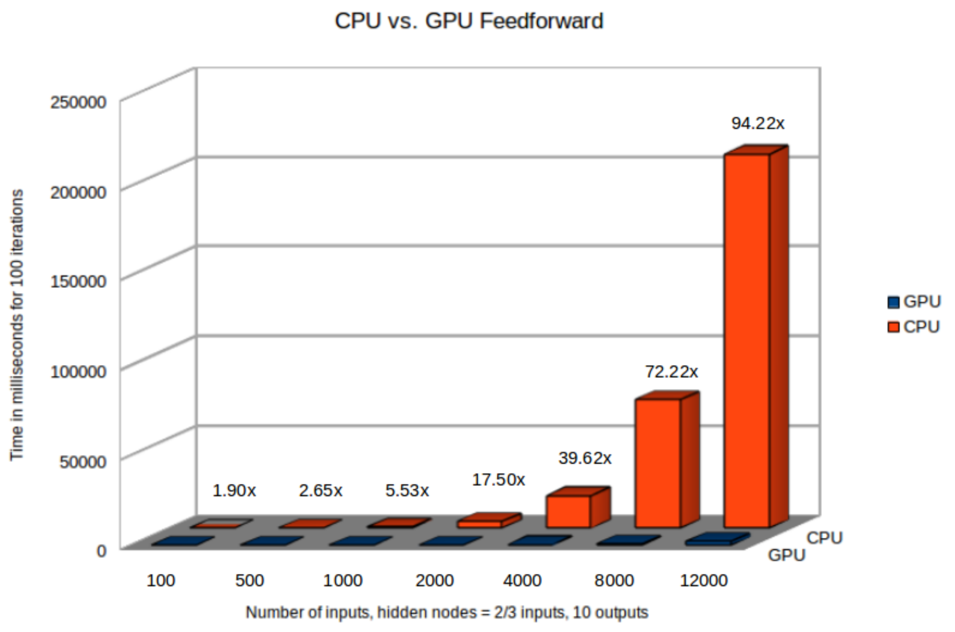

Figure 3. Comparison of feedforward running time for various size networks on the CPU and GPU.

7.2. **Backpropagation.** As with feedforward, I implemented two different backpropagation methods. The first computes the hidden node error gradients by calculating each term in the linear combination in a separation thread and then computes the sum in parallel as well. Likewise it becomes inefficient if the number of outputs is greater than 512. My second implementation computes each hidden node error gradient in a single thread. In my tests these two methods performed almost identically because the number of outputs was small (always less than 10). If the number of outputs were increased to 128 or 256 the first method should be significantly faster. Figure 4 represents the time for 100 iterations of backpropagation comparing the CPU version to the optimal GPU version for the given number of inputs.

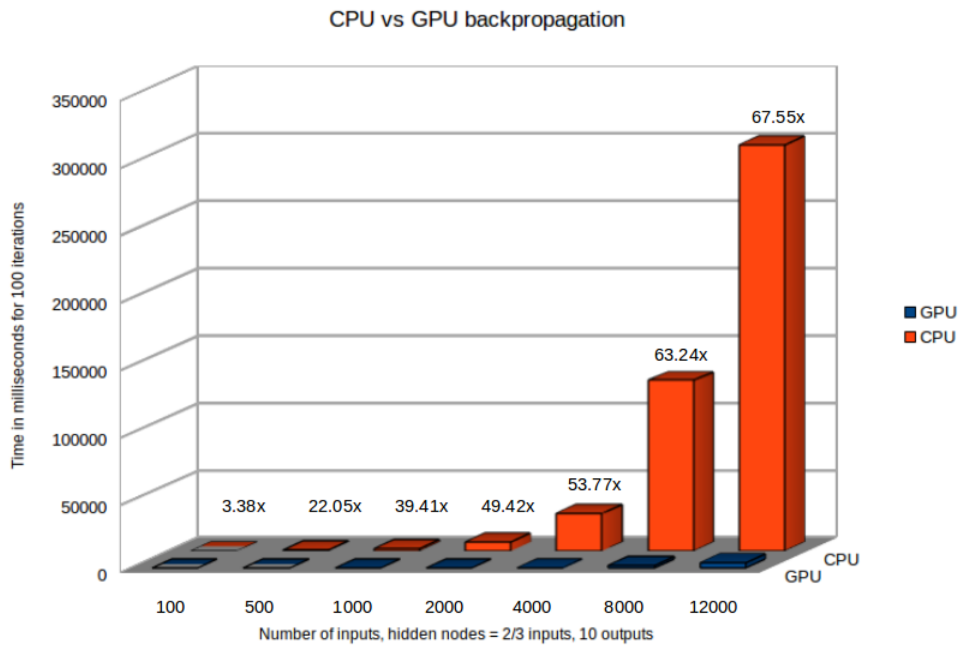

Figure 4. Comparison of backpropagation running time for various size networks on the CPU and GPU.

7.3. **Cumulative.** To find the cumulative speedup from CPU to GPU I compared the time required to run a complete epoch (See Figure 5). Once again, the optimal feedforward and backpropagation methods are chosen based on the input size.

It appears that the time required to complete the calculations using the CPU version increases exponentially as the size of the network increases, whereas the time for the GPU version increases linearly. This is intuitive since the number of weights increases exponentially with respect to the number of inputs as seen in Table 2.

# inputs	100	500	1,000	2,000	4,000	8,000	12,000
# weights	7,370	170,340	673,670	2,681,340	10,694,670	42,725,340	96,080,000

Table 2. Number of weights in a fully connected neural network where the number of hidden nodes equals two-thirds the number of inputs and there are 10 outputs.

I speculate that the GPU version does not scale in the same exponential fashion because the parallel hardware is able to handle a doubling in the number of weights by adding an additional set of kernel executions, causing at worst a 2x slowdown if the hardware is well utilized.

Disclaimer: I spent four months optimizing the GPU version of the neural network and about two weeks working on the CPU version. There are certainly optimizations that could be made to the CPU version that would provide significant improvements. I did my best to ensure that I was not comparing my GPU implementation to a CPU implementation that was doing a lot more work.

8. Test case: face recognition

To test my system with a practical application, I used a face recognition task. I conducted the test before a live audience as part of my presentation on April 10. I used OpenCV to process a live video stream and detect a face. I generated 200, 32x32 images for two different people and then processed these images

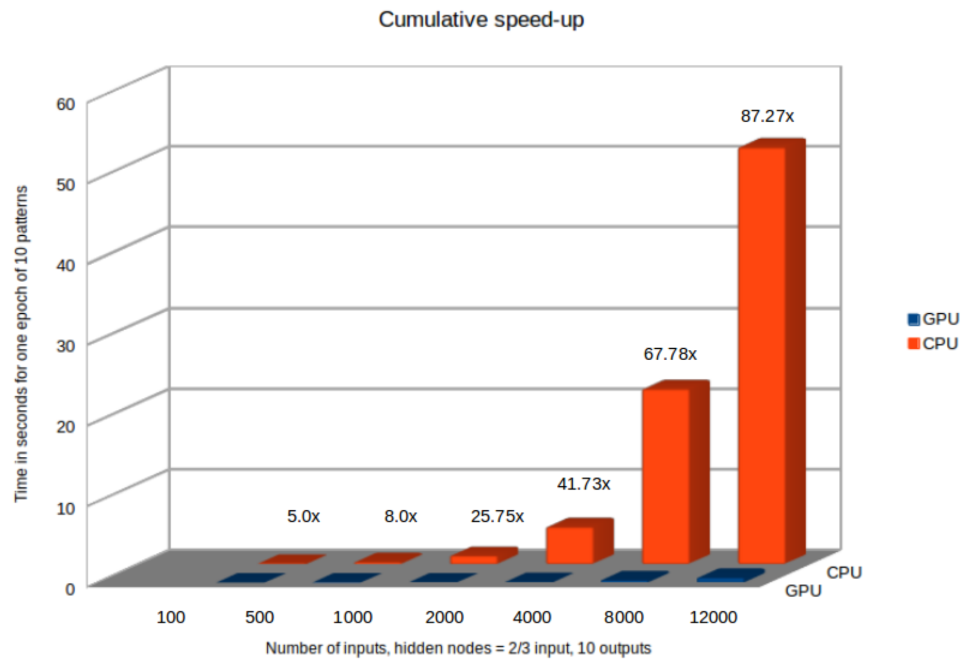

Figure 5. Comparison of time per epoch for various size networks on the CPU and GPU.

to produce a dataset of 10,868 inputs, 1087 hidden nodes ($\frac{1}{10} \cdot$ # inputs), and 2 output nodes (1 for each person). I was able to run 50 epochs and reach 96% accuracy on the training set and 90% accuracy on the test set. On the GPU this test took approximately 3.8 seconds per epoch, or about 3 minutes 10 seconds in all for 50 epochs. On the CPU a single epoch with the same dataset took 120 seconds or 31.4x longer. Completing 50 epochs on the CPU would require 1 hour and 40 minutes!

9. Future work

Every one of my networks was fully connected. This is a standard structure and simple to implement, but the algorithm likely does not need so many connections in order to learn. Additionally, the GPU can run the feedforward and backprop functions much faster if the network conforms to a certain format. Consider the datasets of images that my program produces (as in the face recognition demonstration described above). Each image produced 10,868 inputs. A fully connected network with the number of hidden nodes equaling two-thirds the number of input nodes will have 7,246 hidden nodes and usually just a few outputs, perhaps 2. A fully connected network of this form will have 78,764,020 connections.

If this net is formatted so that every set of 128 input nodes contributes to a single hidden node (and likewise from the hidden nodes to the output nodes) the network will have 85 sections of 128 nodes (fewer in the last section). We could define that approximately half of the 85 hidden nodes will contribute to each output, perhaps 42 and 43, respectively. This results in a total of 10,953 weights. I would estimate conservatively that this version would be at least 3 times faster. My profiling suggests that it would be possible to run all 85 iterations of and 2 iterations of 64 (64 is the first power of two above 42, 43) sequentially in about one-third of the time required by the current method. Assuming decent parallelism with 8 blocks per SM running simultaneously, a speed increase of 128x would not be unreasonable.

While there is an intuitive method for splitting the neural network computation across multiple GPUs while using a batch update algorithm, the division is not as simple for a stochastic update method. I would like to experiment with different network configurations, splitting the data processing across GPUs and periodically merging the learned results from each GPU intelligently in some way.

Finally, deep learning has been a hot topic recently in the machine learning community. As we acquire the ability to process more and more data, deep learning becomes more feasible and also more productive. Accordingly, I would like to add functionality to train networks with more than one hidden layer.

References

[1] Cybenko, G. (1989). Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems, 2*, pp. 303-314.

[2] Keane, A. (2012, July 23). “GPUs are only up to 14 Times Faster than CPUs” says Intel. *NVidia Blog.* Retrieved April 11, 2014, from http://blogs.nvidia.com/blog/2010/06/23/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel/.

[3] NVidia CUDA. (n.d.). Computer software. NVidia CUDA Zone. Vers. 5.5. NVidia, 2014. Retrieved October 1, 2013 from https://developer.nvidia.com/cuda-zone.

[4] OpenCV. (n.d.). Computer software. Open CV. Vers. 2.4.8. Willow Garage, (n.d.). Retrieved April 8, 2014. from http://opencv.org/.