ARTICLE IN PRESS

BDR:57

Big Data Research ••• (••••) •••–•••

Contents lists available at ScienceDirect

Big Data Research

www.elsevier.com/locate/bdr

Prototyping a GPGPU Neural Network for Deep-Learning Big Data Analysis ☆

Alcides Fonseca *, Bruno Cabral

University of Coimbra, Portugal

ARTICLE INFO

Article history:
Received 17 April 2016
Received in revised form 14 November 2016
Accepted 20 January 2017
Available online xxxx

Keywords:
Big-data
Deep-learning
Prototyping
GPGPU
Cluster
Parallel programming

ABSTRACT

Big Data concerns with large-volume complex growing data. Given the fast development of data storage and network, organizations are collecting large ever-growing datasets that can have useful information. In order to extract information from these datasets within useful time, it is important to use distributed and parallel algorithms.

One common usage of big data is machine learning, in which collected data is used to predict future behavior. Deep-Learning using Artificial Neural Networks is one of the popular methods for extracting information from complex datasets. Deep-learning is capable of more creating complex models than traditional probabilistic machine learning techniques.

This work presents a step-by-step guide on how to prototype a Deep-Learning application that executes both on GPU and CPU clusters. Python and Redis are the core supporting tools of this guide. This tutorial will allow the reader to understand the basics of building a distributed high performance GPU application in a few hours. Since we do not depend on any deep-learning application or framework—we use low-level building blocks—this tutorial can be adjusted for any other parallel algorithm the reader might want to prototype on Big Data. Finally, we will discuss how to move from a prototype to a fully blown production application.

© 2017 Elsevier Inc. All rights reserved.

1. Introduction

Deep Learning [1] refers to the usage of Artificial Neural Networks (ANN or NN) with several hidden layers used for data with a high dimensionality. A common example and benchmark for Deep Learning is image classification from the ImageNet dataset [2]. ANNs can be used for classification tasks [3], with several applications in industry, business and science [4]. Examples of applications include character recognition in scanned documents [5], predicting bankruptcy [6] or health complications [7]. Autonomous driving also makes heavy use of ANNs [8].

An ANN begins with random weights, practically deciding everything at random. By training the ANN with several existing instances of the problem, one can evaluate the error produced. Weights are then adjusted, taking into account if it overly or underly estimated the final value.

In order to predict values, ANNs are built connecting layers of neurons. ANNs use the first layer of neurons for each input feature, and the final layer for the classification output. Fig. 1 shows an example of an ANN with four input neurons, four neurons in the hidden layer and two output neurons. All neurons in one layer are connected to all the neurons in the following layer.

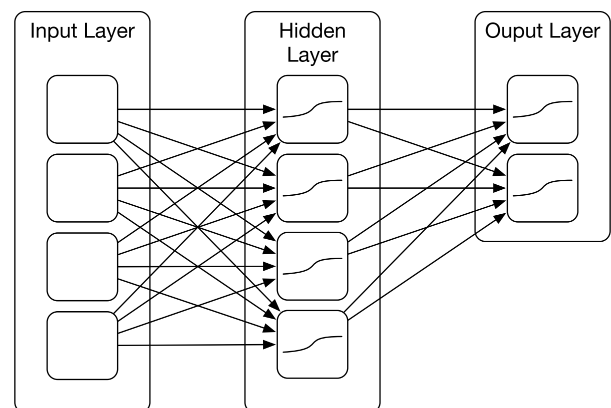

Fig. 1. An example of a neural network with 4 input neurons, 4 neurons in the hidden layer and 2 output layers.

When the number of features increases (high dimensionality), the number of neurons in the hidden layers increases as well, in

☆ This article belongs to HPC Tutorial for Big Data.

* Corresponding author.
E-mail addresses: amaf@dei.uc.pt (A. Fonseca), bcabral@dei.uc.pt (B. Cabral).

http://dx.doi.org/10.1016/j.bdr.2017.01.005
2214-5796/© 2017 Elsevier Inc. All rights reserved.

Please cite this article in press as: A. Fonseca, B. Cabral, Prototyping a GPGPU Neural Network for Deep-Learning Big Data Analysis, Big Data Res. (2017), http://dx.doi.org/10.1016/j.bdr.2017.01.005

ARTICLE IN PRESS

order to compensate for the possible interactions of input neurons [9]. However, a rule of thumb is to use only one hidden layer [10] with the same number of hidden neurons as there are input neurons.

The second scalability issue with ANNs is that for a high accuracy, they have to be trained with a large dataset. Typically, to achieve a good accuracy score, the number of instances should be three orders of magnitude higher than the number of features [10]. Thus, we reach a point in which we need to train an ANN over several iterations, using a high number of features and instances. In these conditions, training an ANN becomes a computationally intensive operation, highly demanding in terms of processing, memory and disk usage. As the amount of data available for training goes above a terabyte, it becomes Big Data problem.

The solution for Big Data processing is to distribute the computation across different machines, splitting data among them and merging results afterwards. In Map-Reduce approaches, it is possible to divide the computation into independent sub-problems that can be combined to produce a final result. Hadoop and Spark [11] are the most used frameworks for Big Data processing.

ANNs are described by the following characteristics: layout (the number of layers and neurons on each layer) and the weights of connections between neurons (the second attribute is dependent on the first). When training an ANN for a specific problem dataset, the weights are being adjusted to minimize the output error.

Because the prediction of ANNs can be described as matrix operations (we are multiplying the same weights to a each row of features of problem instances), graphical processing units (GPUs) are usually a good solution for improving performance and reduce training times. GPUs were designed to perform matrix operations in the context of video processing, but have been widely used for other ends. This technique is commonly referred to as General Purpose GPU programming (GPGPU).

In this tutorial we will demonstrate how to implement a Neural Network for Big Data analysis using GPUs. The sources for this tutorial are available for download at https://github.com/alcides/bigdatagpunn.

In Section 2 we introduce the problem, as well as the ANN being created. In Section 3 it is shown how to implement a basic ANN. In Section 4, we explain how a Master–Worker distributed model can be implemented using a message queue. In Section 5, we describe how to implement a ANN for running on the GPU. Finally, in Section 6, we discuss the merits and shortcomings of our approach and present some thoughts on how to improve it. Section 7 summarizes this tutorial.

2. Problem and dataset

Before starting implementing the ANN, it is necessary to define the problem dataset to use for development and testing purposes. It would be unpractical to use a large dataset in development, since it would impose an unbearable overhead in each test iteration. Instead, a small subset of a larger dataset should be used for prototyping, and the remaining data should be used for evaluating the complete solution.

As such, we are going to use the Wine Data Set [12], which has been utilized for evaluating classifiers when faced with high dimensionality (a high number of features). The dataset has 178 instances, a reasonable number for testing our prototype. Each instance has 13 features (either real or integer values) and three output values. Since each instance can belong to one of three classes, the output values are set to one if the instance belongs to the corresponding class.

In this tutorial, we will show how to build an ANN trained to classify wine when faced with an unknown instance. The ANN will have 13 input neurons and three output neurons, matching the dataset format. There is one hidden layer and 13 input neurons. Bear in mind that this is only an example configuration, many others are acceptable and may even have better results.

In order to adapt more easily to different problems, ANNs use at their core a non-linear function. In our prototype we will use the sine function, the most common function for this use. In order to improve the learning performance, we normalize our dataset to values between 0 and 1.

3. A neural network in Python

```
1 import numpy as np
2 import pandas as pd
3
4 df = pd.read_csv("datasets/wine.txt", sep="\t",
     header=None)
5
6 instances = df.shape[0]
7 train_instances = 20
8 ndims = 13
9 nclasses = 3
```

Listing 1.1. Dataset import.

Importing the dataset, which is in a CSV file, can be accomplished using Pandas and Numpy libraries [13]. These libraries allow to store the data in memory efficiently since they resort to C arrays, which are less demanding in terms of resources than lists in Python. This step is described in Listing 1.1

```
1 def generate_random_config():
2     weights0 = 2 * np.random.random((ndims,
          ndims)) - 1
3     weights1 = 2 * np.random.random((ndims,
          nclasses)) - 1
4     return (weights0, weights1)
```

Listing 1.2. Generation of a random configuration.

Since the layout of the ANN is static and pre-defined, only the weights between nodes can be dynamically adjusted. Listing 1.2 shows the function used to randomly generate weights for the starting ANN. `weights0` refers to the weights applied to each of the 13 values in the input layer (Layer 0) and `weights1` refers to the weights applied to the 13 values in the hidden layer (Layer 1). Random values are scaled to be in the $[-1, 1[$ range, which is suitable to use with sigmoid activation functions.

```
1  def train(X, y, conf, iterations=6000):
2      weights0, weights1 = conf
3      for j in xrange(iterations):
4          # Feed forward
5          l0 = X
6          l1 = sigmoid(np.dot(l0,weights0))
7          l2 = sigmoid(np.dot(l1,weights1))
8          # Back Propagation
9          l2_error = y - l2
10         l2_delta = l2_error*sigmoid_d(l2)
11         l1_error = l2_delta.dot(weights1.T)
12         l1_delta = l1_error * sigmoid_d(l1)
13         weights1 += l1.T.dot(l2_delta)
14         weights0 += l0.T.dot(l1_delta)
15     return (weights0, weights1)
```

Listing 1.3. Training the ANN.

Listing 1.3 depicts the training process, based on the example from "A Neural Network in 11 lines" [14]. The training adjustment is performed a certain number of times, defaulting to 6000. The training process starts by feed-forwarding the input data through

Please cite this article in press as: A. Fonseca, B. Cabral, Prototyping a GPGPU Neural Network for Deep-Learning Big Data Analysis, Big Data Res. (2017), http://dx.doi.org/10.1016/j.bdr.2017.01.005

Free paper freepaper.me

the neural network, calculating the error between the prediction and the recorded value, and then back propagating the error to adjust the weights matrices. In line 6, the weights are applied to the instances data (`X`) using a matrix multiplication. Each neuron has its value modified by the sigmoid function. The sigmoid function returns values between 0 and 1, and has a derivative easy to compute (Listing 1.4). Line 7 repeats the same process of line 6, but from the hidden layer to the output layer. `l2` will be a matrix that will have one of three possible values for each instance row: a prediction of which class the instance belongs to. At this point, the neural network has been used to obtain a prediction. The quality of the current solution (`l2_error`) will be low because we initiated the process using random weights. In order to improve, it is necessary to propagate the error into each weight matrix, adjusting the values so that in future predictions it is closer to the correct value. This is done by multiplying the error by the final value before the sigmoid function (using the derivative), having a small value by which change the weight in the right direction. This process is repeated for the previous layer, thus changing `weights0` and `weights1`. Repetition will approximate the weights, so that the error is minimized.

```
1  def sigmoid(x):
2      return 1/(1+np.exp(-x))
3
4  def sigmoid_d(x):
5      return x*(1-x)
```

Listing 1.4. The Sigmoid function and its derivative.

Listing 1.5 shows an example that applies the training process to the Wine Dataset. Lines 2 and 3 divide the dataset input file into features (input) and classes (output). `output_conf` will have the final configuration of the ANN to classify new instances.

```
1  conf = generate_random_config()
2  X = df.iloc[0:train_instances,0:ndims].as_matrix
       ()
3  y = df.iloc[0:train_instances,ndims:].as_matrix
       ()
4  output_conf = train_fun(X, y, conf_, iterations)
```

Listing 1.5. An example of a call to the training method.

4. A distributed neural network using redis

In the previous section we wrote a function to train an ANN based on a given dataset. For Big Data, things are more complex. To handle large amounts of data with hight dimensionality, it is useful to distribute storage and computation distributed across machines. Map-Reduce is a very common paradigm for parallelizing algorithms across different machines. The problem is divided in several sub-problems, each concerning a subset of the data. Each sub-problem is solved in a different machine, and the results are combined to produce the solution for the overall problem.

Parallelizing an ANN can be done by subdividing the whole into several training sets [15]. Each machine trains the neural network on its dataset, obtaining different weight matrices per machine. Then, matrices are combined by averaging each value across matrices.

The example in Fig. 2 shows a master and two worker processes. The master process manages execution, creates the requests sent to the workers, waits for responses and merges results. Training is asynchronous and occurs in parallel, and consumes the larger part of the computation time because it has to be repeated a high number of times.

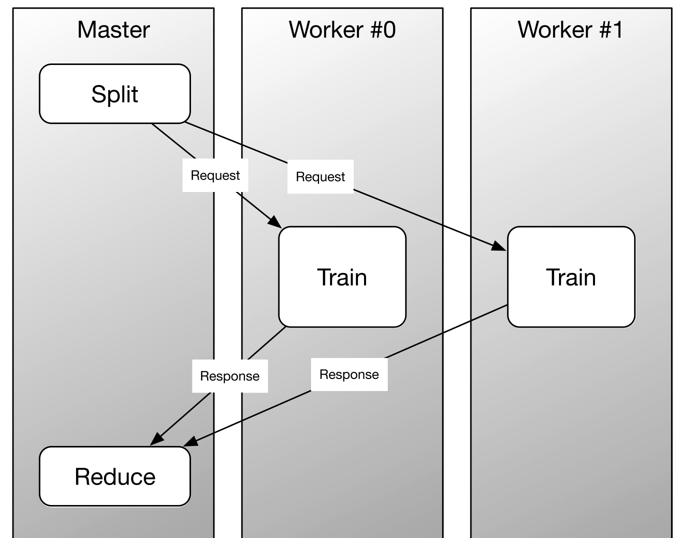

Fig. 2. Example of a Master–Worker model with two workers.

Requests and Responses have to be exchanged over the network, in order to allow distinct machines to collaborate in the computational work. To this end, we used redis [16] database as a message broker. Several other message queue systems could have been used, but we decided to work on redis because of its flexibility, no configuration is necessary and the interface is very straightforward. Other alternatives will be discussed in Section 6. We only need a single redis instance, running on the master node, to enqueue requests and responses. To keep things as simple as possible, requests and responses hold the same data:

- Number of iterations to be performed (can be used to control the task duration)
- The bounds of the data subset to use
- The configuration of the ANN

In requests, the ANN configuration is the initial master configuration. In responses, the configuration is the result of the worker training.

Since redis only accepts strings as values, it is necessary to encode and decode information. Listing 1.6 shows an example of

```
1  import redis
2  r = redis.StrictRedis(host='localhost', port
       =6379, db=0)
3
4  def encode_req(a,b,it,conf):
5      weights0, weights1 = conf
6      metadata = "|".join(map(str,[a,b,it,
           weights0.shape[0], weights0.shape[1],
           weights0.dtype, weights1.shape[0],
           weights1.shape[1], weights1.dtype ]))
7      data = conf[0].ravel().tostring()
8      data2 = conf[1].ravel().tostring()
9      return metadata, data, data2
10
11 def decode_req(metadata, data, data2):
12     a, b, iterations, l, w, array_dtype, l2, w2,
            array_dtype2 = metadata.split('|')
13     weights0 = np.fromstring(data, dtype=
           array_dtype).reshape(int(l), int(w))
14     weights1 = np.fromstring(data2, dtype=
           array_dtype2).reshape(int(l2), int(w2))
15     return int(a), int(b), int(iterations),
           (weights0, weights1)
```

Listing 1.6. Redis encoding functions.

Please cite this article in press as: A. Fonseca, B. Cabral, Prototyping a GPGPU Neural Network for Deep-Learning Big Data Analysis, Big Data Res. (2017), http://dx.doi.org/10.1016/j.bdr.2017.01.005

freepaper.me FREE paper

a possible encoding. Encoding Numpy arrays is not trivial. Arrays have to be reduced to one dimension and converted to string. The dimensions and data type of the array are communicated separately as metadata.

Listing 1.7 shows the source code of the master process. The master divides data by the workers by encoding a message for each one, with different bounds and sends metadata to a queue ("worker_0" for the first worker) and the configuration matrices to two other queues.

```
1  master_conf = generate_random_config()
2  blocks_per_worker = instances/(workers+2)
3  for k in range(10):
4      for i in range(workers):
5          a = blocks_per_worker * i
6          b = blocks_per_worker * (i+1)
7          print "Scheduling to worker", i, " data
             from ", a, " to ", b
8          metadata, data, data2 = encode_req(a, b,
               60000, master_conf)
9          r.rpush("worker_%d" % i, metadata)
10         r.rpush("worker_data_%d" % i, data)
11         r.rpush("worker_data2_%d" % i, data2)
```

Listing 1.7. Master splitting.

The source code for the worker is shown in Listing 1.8. It starts by decoding the request data. Since this is a blocking call, this call can be executed inside a loop to have a worker continuously processing new requests. In lines 6 to 8, the worker prepares data and trains the model. In the end, it encodes a response with the new configuration and pushes the results to a mirror queue, which connects worker to master (e.g., "master_0").

```
1  metadata = r.blpop('worker_%d' % wid)[1]
2  data = r.blpop('worker_data_%d' % wid)[1]
3  data2 = r.blpop('worker_data2_%d' % wid)[1]
4  a, b, iterations, conf = decode_req(metadata,
       data, data2)
5
6  X = df.iloc[a:b,0:ndims].as_matrix()
7  y = df.iloc[a:b,ndims:].as_matrix()
8  output_conf = train(X, y, conf, iterations)
9
10 metadata, data, data2 = encode_req(a, b,
       iterations, output_conf)
11 r.rpush("master_%d" % wid, metadata)
12 r.rpush("master_data_%d" % wid, data)
13 r.rpush("master_data2_%d" % wid, data2)
```

Listing 1.8. Worker code.

Finally, the master node receives results from the workers, and averages the received matrices to achieve a final configuration. The code for this part is omitted, as it is similar to the code used by the worker to receive requests.

This distributed architecture, is also suitable for multi-core platforms. In a 8-core machine, 8 workers can be executed in parallel and, when each get a slice of the data to process, the efficiency of multi-core processors is at its best.

5. A GPU-powered neural network

GPUs were designed to accelerate the processes of graphics generation, which consist on performing thousands of matrix operations, such as matrix multiplications. From Listing 1.3, it is possible to understand how training a neural network can also be achieved by matrix multiplications, and scalar multiplications, additions and subtractions, making it clear that training ANNs in GPUs would be largely beneficial in terms of performance.

Programming for GPUs typically requires using a low level language such as Cuda or OpenCL. Recent efforts have allowed high-level programming languages to be compiled to the GPU, such as Matlab [17], Haskell [18], Java [19] or Python [20]. Also, Nvidia GPU support has been added to Numba [21], a just-in-time compiler for Python functions. In this work we will use Numba and Nvidia hardware. There is also support for HSA AMD GPUs, but it is out of scope for this work.

The first thing we need to know about GPGPU is that GPUs and CPUs do no share the same memory. This means that data and code have to be copied to the GPU and back. As such, in order for functions to be executed on the GPU, they should be annotated as such to be compiled to the GPU assembly. Furthermore, arguments have to be copied to the GPU before execution, and results to the main memory after execution. Listing 1.9 shows the source code if the worker that executes on the GPU. The initial configuration is copied to the GPU, as well as the input and output matrices from the dataset. The kernel, which is the function that executes in the GPU, is invoked in line 11. Note that when invoking the kernel it is necessary to pass the GPU versions of the arguments. Next, the resulting configuration is copied from the GPU to main memory. Lines 9 and 10 are used to compute the kernel work-group and work-items, which are essential steps in GPGPU.

```
1  def train_cuda(X, y, conf, iterations=6000):
2      gpu = cuda.get_current_device()
3      weights0, weights1 = conf
4      weights0g = cuda.to_device(weights0)
5      weights1g = cuda.to_device(weights1)
6      Xg = cuda.to_device(X)
7      yg = cuda.to_device(y)
8      rows = X.shape[0]
9      thread_ct = (gpu.WARP_SIZE, gpu.WARP_SIZE)
10     block_ct = [ int(math.ceil(1.0 * rows / gpu.
           WARP_SIZE)), int(math.ceil(1.0 * ndims /
            gpu.WARP_SIZE))]
11     train_kernel[block_ct, thread_ct](Xg, yg,
           weights0g, weights1g, iterations)
12     weights0g.to_host()
13     weights1g.to_host()
14     return (weights0, weights1)
```

Listing 1.9. Training an ANN on the GPU—Host code.

The kernel is be executed in several threads. The work-group and work-item sizes are GPU layout configurations that the developer uses to decide how many threads will execute the kernel and in how many groups they will be organized. If it is not important for the algorithm, as it is the case here, a good number for the work-group size is the Warp size. Physically, GPU threads are organized in groups, or warps, so matching the virtual layout with the physical layout yields the best performance. Also, the number of threads should be as high as the size of the largest matrix multiplication result, which is the number of instances by dimensions. In this case, the Warp size is 32, resulting in a matrix o 32 by 32 work-items. The resulting 1024 threads will not all perform used work, as the multiplied matrix will be of size 20 (instances) by 13 (features). While in this benchmark there are more threads than work to perform, in a larger program it would be the opposite.

Listing 1.10 shows the auxiliary functions used by the kernel. These are the same Python functions in Listing 1.4, except they have a decorator on top, marking the function as being able to execute on the GPU. Furthermore, by enabling function inlining there is no overhead in function calling on the GPU.

Finally, Listing 1.11 shows the main kernel function. Despite being written in Python, code inside kernel functions is limited to a subset of the language, called NoPython. This subset does not support **try**, `catch`, `with`, `with` and comprehensions. Kernel functions also cannot return anything.

Please cite this article in press as: A. Fonseca, B. Cabral, Prototyping a GPGPU Neural Network for Deep-Learning Big Data Analysis, Big Data Res. (2017), http://dx.doi.org/10.1016/j.bdr.2017.01.005

freepaper.me paper

```
1  @cuda.jit(device=True, inline=True)
2  def sigmoidg(x):
3      return 1/(1+math.exp(−x))
4
5  @cuda.jit(device=True, inline=True)
6  def sigmoidg_d(x):
7      return x∗(1−x)
```

Listing 1.10. Sigmoid function and its derivative on the GPU.

```
1  @cuda.jit()
2  def train_kernel(X, y, weights0, weights1,
       iterations):
3      l1 = cuda.shared.array(shape=(instances,
           ndims), dtype=numba.float32)
4      l2_delta = cuda.shared.array(shape=(
           instances, 3), dtype=numba.float32)
5      l1_delta = cuda.shared.array(shape=(
           instances, ndims), dtype=numba.float32)
6      i, j = cuda.grid(2)
7      if i < instances and j < ndims:
8          for it in range(iterations):
9              acc = 0
10             for k in range(ndims):
11                 acc += X[i, k] ∗ weights0[k, j]
12             l1[i, j] = sigmoidg(acc)
13             cuda.syncthreads()
14             if j < 3:
15                 acc = 0
16                 for k in range(ndims):
17                     acc += l1[i,k] ∗ weights1[k,
                           j]
18                 l2 = sigmoidg(acc)
19                 l2_error = y[i, j] − l2
20                 l2_delta[i, j] = l2_error ∗
                       sigmoidg_d(l2)
21             cuda.syncthreads()
22             acc = 0
23             for k in range(3):
24                 acc += l2_delta[i,k] ∗ weights1
                       [j, k]
25             l1_error = acc
26             l1_delta[i, j] = l1_error ∗
                   sigmoidg_d(l1[i, j])
27             cuda.syncthreads()
28             if j < 3:
29                 acc = 0
30                 for k in range(instances):
31                     acc += l1[k, i] ∗ l2_delta
                           [k, j]
32                 weights1[i, j] += acc
33             acc = 0
34             for k in range(instances):
35                 acc += X[k, i] ∗ l1_delta[k, j]
36             weights0[i, j] += acc
37             cuda.syncthreads()
```

Listing 1.11. Kernel function for ANN training.

Marking a function as a kernel is accomplished by using the `@cuda.jit()` decorator. By doing so, we can use features that are GPU-only, such as getting the position of each thread in the work-group, allocating local and shared memory and using barriers to synchronize threads at a given point.

Shared arrays can be used to store temporary values in the GPU. Accessing these arrays is typically faster than using the global memory, the one the CPU writes to and reads from. Lines 3, 4 and 5 use this feature to allocate the arrays necessary to store the result of intermediate matrices.

Since the kernel function executes in all threads, we need to identify each thread so it acts only on its respective input values. This is done by reading the position of the thread in the work-group. The actual number of threads can be decided later during runtime. Line 6 obtains the 2D position in the work-group grid. In the previous example, the coordinates in the grid are given by (i, j), with i between 0 and 24 and j between 0 and 16.

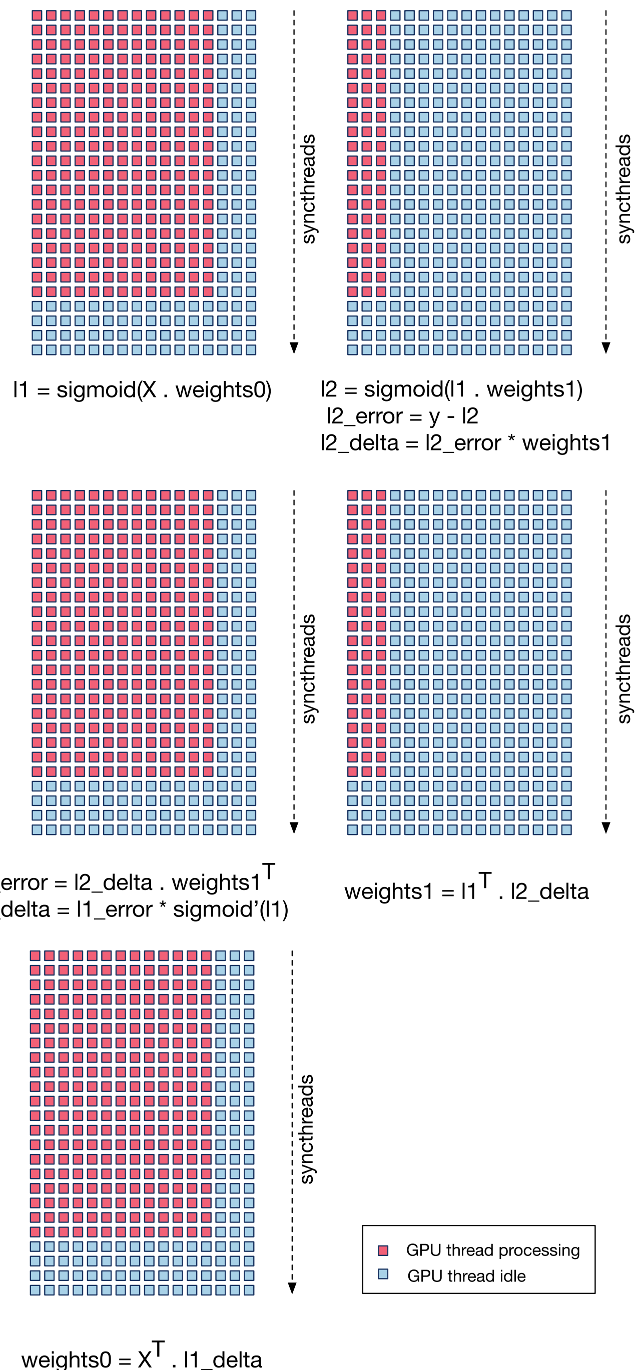

Fig. 3. A representation of threads performing computations between barriers.

Since our kernel performs several matrix multiplications, and not all matrices have the same size, we need to adjust the number of threads for the matrix being multiplied (lines 7, 14 and 28). Fig. 3 shows a representation of what threads are running on which matrix operations, and the barriers used to synchronize among threads.

Each matrix multiplication is expressed as a accumulative sum of the products between the corresponding cells of the input matrix. Examples of this pattern are in Lines 9 to 12, 15 to 18, 22 to 25, 29 to 32 and 33 to 36. These operations could be performed in different kernels, but compiling and scheduling a kernel to the

Please cite this article in press as: A. Fonseca, B. Cabral, Prototyping a GPGPU Neural Network for Deep-Learning Big Data Analysis, Big Data Res. (2017), http://dx.doi.org/10.1016/j.bdr.2017.01.005

ARTICLE IN PRESS

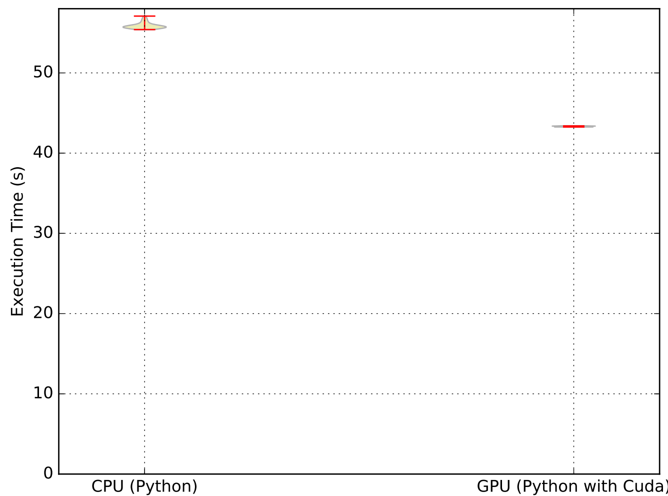

Fig. 4. Comparison of the execution times between CPU and GPU versions.

GPU has its overheads, so executing everything in the same kernel is faster.

Since each Warp can execute at its own pace, threads may be executing different matrix multiplications at a given moment, creating a race condition. In order to prevent this, we introduce a synchronization point in the program, in particular, a barrier among all threads using the `cuda.syncthreads()` function, present in lines 13, 21, 27 and 37. When two matrix operations are not dependent (as it is the case of those starting at lines 29 and 33), they do not required a synchronization barrier between them.

Having the kernel, the auxiliary device functions and the host code to schedule the kernel, we now have a GPGPU Neural Network executing. In order to have an idea of the performance, we compared the GPU version with the regular Python version. We repeated the measurements 30 times on CPython 2.7.6. Our machine has an Intel i7-3520M processor and a NVIDIA GeForce GT 640 LE GPU.

Fig. 4 uses violin plots to represent the distribution of execution times of the GPU and the CPU versions, as well as the quartiles of the distributions. The GPU version executes in one fifth of the time of the CPU version, showing how this type of programs can be easily parallelized on the GPU with speedups.

6. Discussion and homework

Just like any prototype, this is not a finished product. As such, there are several shortcomings and significant room for improvement. This section addresses issues with our work and proposes alternatives that the reader may opt to pursue.

Python is not the best language in terms of performance, given its overhead in interpreting code, expensive dynamic data structures (such as lists) and the Global Interpreter Lock (GIL). While this is true, in our implementation, most of the computation does not execute in Python: the GPU version compiles Python to LLVM, which is compiled to the Nvidia PTX format, similarly to how cuda works; and, the CPU version executes the matrix multiplications in C, with C structures, thanks to the Numpy library. But, it is possible to reduce even further the overhead of Python code in the CPU training function: execute the code through the Pypy interpreter, which features JIT compilation and improves the performance of hot code; Numba also features a JIT compiler for CPU that can be activated using a one-line decorator; the code could also be written in Cython, a typed version of Python that is compiled down to C. Finally, GIL prevents efficient thread-based parallelization inside the same process. That is why our CPU parallelization resorts to separate processes, each with its own individual GIL, taking the greatest advantage of CPU parallelism. But we still have the overhead of Redis communication inside the same machine. This can be reduced using shared memory. The Python multiprocessing module can automate this process, but has some limitations in communicating the results back to the parent process.

Redis is a good choice for a production message queue. But, ActiveMQ [22] and 0MQ [23] are also good alternatives. The traditional advantage of Redis, which is not relevant in this example, is that is also serves as a NoSQL database store, frequently used for caching. Nonetheless, any of these require the programmer to prepare messages as raw bytes, using the metadata technique presented in this paper for serializing Numpy arrays.

Another aspect that is not considered in this work is how to split data through different machines. A Distributed File System such as NFS can be used, but slices of the dataset can also be downloaded from a remote location before the local execution of the training code. Any of these solutions require data copying mechanisms. Fortunately, in our parallelization approach each worker only accesses a subset of the data and does not need to access all of it.

The Neural Network implementation depicted in this paper had only one hidden layer with 13 neurons. But, Deep-Learning assumes the presence of several hidden layers, each with many neurons. Adding more layers and neurons is left as an exercise for the reader. Choosing an ideal configuration implies more work, which can include pruning of useless neurons [24], using genetic algorithms to evolve ANNs [25] or Monte Carlo methods [26]. These approaches require a more extended study of the subjects.

Another aspect is the evaluation of the solution. A simple approach would be to leave a slice of the benchmark data apart for evaluating the trained ANN. Cross-validation is possible, but it requires much more computing power for the same data size.

Finally, the GPU approach presented here is far from being optimized. The first concern is how much memory is sent from the CPU to the GPU. GPUs have limited memories and a worker may be able to store more data in the main memory. A common approach for this scenario is to split the GPU memory in chunks. While the GPU is processing a chunk, the CUDA driver is asynchronously sending the remaining chunks. By synchronizing data copies with kernel executions, it is possible to process more data than what really fits in memory. The overhead of this process can be lower than expected, since copies can be performed in parallel with computations that do not target the same data. The organization of work-groups and work-items can also be improved. Our solution left many GPU cores idle while trying to maximize cache locality.

Maximizing the GPU throughput can be done by selecting the ideal matrix sizes to be multiplied. In our ANN, this is done by changing the number of hidden layers and neurons used. After this configuration layout is decided, then the number of work-group and items can be optimized. These are two optimizations that can easily be applied to our codebase, with a few trial-and-error executions.

Our ANN can be improved by leveraging multi-GPU machines. Although our implementation allows each CPU process to drive a GPU, recent GPUs have direct or cheaper accesses to the memories of other GPUs, which would allow the averaging of weights to occur on the GPU.

7. Conclusion

To conclude, in this work we have presented a tutorial for implementing a Neural Network for classification purposes using back-tracking on both CPU and GPUs. We have also proposed a distributed protocol for training the ANN in parallel across machines,

Please cite this article in press as: A. Fonseca, B. Cabral, Prototyping a GPGPU Neural Network for Deep-Learning Big Data Analysis, Big Data Res. (2017), http://dx.doi.org/10.1016/j.bdr.2017.01.005

freepaper.me paper

freepaper.me paper

CPU cores and GPUs. We addressed the possibility of big data sizes by distributing the dataset across several machines. We have also use the GPUs to improve the processing time. For this tasks, we have used: Python, Numpy, Numba and Redis, open-source tools that can be used for prototyping this and other ANNs, as well as other computation intensive Map-Reduce methods for Big-Data.

We have also address the shortcomings of our prototype, such as the dataset scaling, the choice of language and tools for a final version, the parallelization inside each machine and the ANN features not considered for this prototype.

Acknowledgements

The first author was supported by the Portuguese National Foundation for Science and Technology (FCT) through a Doctoral Grant (SFRH/BD/84448/2012).

Appendix A. Supplementary material

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.bdr.2017.01.005.

References

[1] J. Schmidhuber, Deep learning in neural networks: an overview, Neural Netw. 61 (2015) 85–117.
[2] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: Advances in Neural Information Processing Systems, 2012, pp. 1097–1105.
[3] G.P. Zhang, Neural networks for classification: a survey, IEEE Trans. Syst. Man Cybern., Part C, Appl. Rev. 30 (4) (2000) 451–462.
[4] B. Widrow, D.E. Rumelhart, M.A. Lehr, Neural networks: applications in industry, business and science, Commun. ACM 37 (3) (1994) 93–106.
[5] G.L. Martin, J.A. Pittman, Recognizing hand-printed letters and digits using backpropagation learning, Neural Comput. 3 (2) (1991) 258–267.
[6] G. Zhang, M.Y. Hu, B.E. Patuwo, D.C. Indro, Artificial neural networks in bankruptcy prediction: general framework and cross-validation analysis, Eur. J. Oper. Res. 116 (1) (1999) 16–32.
[7] M.H. Ebell, Artificial neural networks for predicting failure to survive following in-hospital cardiopulmonary resuscitation, J. Fam. Pract. 36 (3) (1993) 297–304.
[8] D.A. Pomerleau, Efficient training of artificial neural networks for autonomous navigation, Neural Comput. 3 (1) (1991) 88–97.
[9] S. Lawrence, C.L. Giles, A.C. Tsoi, What Size Neural Network Gives Optimal Generalization? Convergence Properties of Backpropagation, UMIACS-TR-96-22 and CS-TR-3617, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, 1998.
[10] W.S. Sarle, On Computing Number of Neurons in Hidden Layer, February 1995.
[11] L. Gu, H. Li, Memory or time: performance evaluation for iterative operation on hadoop and spark, in: 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC_EUC, IEEE, 2013, pp. 721–727.
[12] M. Lichman, UCI Machine Learning Repository, http://archive.ics.uci.edu/ml/, 2013.
[13] W. McKinney, et al., Data structures for statistical computing in Python, in: Proceedings of the 9th Python in Science Conference, vol. 445, 2010, pp. 51–56.
[14] A. Trask, A Neural Network in 11 Lines of Python, http://iamtrask.github.io/2015/07/12/basic-python-network/, 2013.
[15] G. Dahl, A. McAvinney, T. Newhall, et al., Parallelizing neural network training for cluster systems, in: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, ACTA Press, 2008, pp. 220–225.
[16] S. Sanfilippo, P. Noordhuis, Redis, 2009.
[17] J. Reese, S. Zaranek, GPU Programming in Matlab. MathWorks News&Notes, The MathWorks Inc, Natick, MA, 2012, pp. 22–25.
[18] M.M. Chakravarty, G. Keller, S. Lee, T.L. McDonell, V. Grover, Accelerating Haskell array codes with multicore GPUs, in: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, ACM, 2011, pp. 3–14.
[19] A. Fonseca, B. Cabral, ÆminiumGPU: an intelligent framework for GPU programming, in: Facing the Multicore-Challenge III, Springer, 2013, pp. 96–107.
[20] B. Catanzaro, M. Garland, K. Keutzer, Copperhead: compiling an embedded data parallel language, ACM SIGPLAN Not. 46 (8) (2011) 47–56.
[21] S.K. Lam, A. Pitrou, S. Seibert, Numba: a LLVM-based Python JIT compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, ACM, 2015, p. 7.
[22] B. Snyder, D. Bosnanac, R. Davies, ActiveMQ in Action, vol. 47, Manning, 2011.
[23] P. Hintjens, ZeroMQ: Messaging for Many Applications, O'Reilly Media, Inc., 2013.
[24] E.D. Karnin, A simple procedure for pruning back-propagation trained neural networks, IEEE Trans. Neural Netw. 1 (2) (1990) 239–242.
[25] D.B. Fogel, L.J. Fogel, V. Porto, Evolving neural networks, Biol. Cybern. 63 (6) (1990) 487–493.
[26] J.F. de Freitas, M. Niranjan, A.H. Gee, A. Doucet, Sequential Monte Carlo methods to train neural network models, Neural Comput. 12 (4) (2000) 955–993.

Please cite this article in press as: A. Fonseca, B. Cabral, Prototyping a GPGPU Neural Network for Deep-Learning Big Data Analysis, Big Data Res. (2017), http://dx.doi.org/10.1016/j.bdr.2017.01.005