Coarse Grain Parallelization of Deep Neural Networks

Marc Gonzalez Tallada
Universitat Politecnica de Catalunya-BarcelonaTech
marc@ac.upc.edu

Abstract

Deep neural networks (DNN) have recently achieved extraordinary results in domains like computer vision and speech recognition. An essential element for this success has been the introduction of high performance computing (HPC) techniques in the critical step of training the neural network. This paper describes the implementation and analysis of a network-agnostic and convergence-invariant coarse-grain parallelization of the DNN training algorithm. The coarse-grain parallelization is achieved through the exploitation of the batch-level parallelism. This strategy is independent from the support of specialized and optimized libraries. Therefore, the optimization is immediately available for accelerating the DNN training. The proposal is compatible with multi-GPU execution without altering the algorithm convergence rate. The parallelization has been implemented in Caffe, a state-of-the-art DNN framework. The paper describes the code transformations for the parallelization and we also identify the limiting performance factors of the approach. We show competitive performance results for two state-of-the-art computer vision datasets, MNIST and CIFAR-10. In particular, on a 16-core Xeon E5-2667v2 at 3.30GHz we observe speedups of 8× over the sequential execution, at similar performance levels of those obtained by the GPU optimized Caffe version in a NVIDIA K40 GPU.

Categories and Subject Descriptors [*Computing Methodologies*]: Parallel Algorithms, Machine Learning, Neural Networks

General Terms Performance, Coarse-grain Parallelism, Shared Memory Algorithms

Keywords Deep Learning, Neural Networks, OpenMP, Stochastic Gradient Descent

1. Introduction

Recently, deep neural networks (DNN) have achieved extraordinary results in domains like computer vision and speech recognition [15, 25]. One key and essential element for this success has been the introduction of high performance computing (HPC) techniques within one critical step in the network deployment: the network training. Neural networks require a training stage which typically has very high computational demands. Currently, the optimization of this stage has become the main bottleneck for a reasonable trade-off between the final network accuracy and the actual length of the training process [8, 10]

During the training process, a gradient descent algorithm minimizes a pre-defined cost function. Most computations along this process correspond to basic linear algebra operations. In general, an optimal training algorithm is built upon a very specific data layout mainly composed of matrices and vectors, and an iterative procedure that invokes specialized and highly optimized basic linear algebra subroutines (BLAS). All the widely used DNN frameworks (e,g.: Caffe [17], Theano [1] or Torch [9]) follow this approach. When it comes to optimization, GPU acceleration has been the most widespread adopted strategy and it has shown extraordinary performance levels. GPU's are massively parallel architectures for very fine grain parallelism and BLAS computations make a perfect fit for them. But in terms of programmability, porting the original code to GPU kernels requires significant programming efforts. The general solution has been to use specialized libraries like cuBLAS and cuDNN [5]. Unfortunately, this approach lacks from generality. During the research cycle to create a DNN, the optimal network topology, parameters and data layout are unknown. So, the specialized GPU libraries are not yet useful. Even for the case of cuDNN, only DNN transformations that are very well understood (e.g.: convolution and pooling operations), are supported, and has a strong bias towards computer vision applications.

An unexplored strategy is a CPU parallelization at a coarser level. Neural networks are trained with batches of data where each batch contains a fixed number of data samples. Each sample is used to advance the minimization of the cost function and all samples in the batch can be processed in parallel. This defines a much coarser thread level parallelism but adds complexity to the parallelization process: thread level parallelism has to be explicitly introduced and race conditions appear for the network coefficient update. The batch-level parallelism is inherent to the gradient descent algorithm and produces immediate acceleration irrespective of the nature of the network computations. It does not rely on a particular optimal data layout nor the availability of any specialized and highly optimized library for the network layers. So, its activation does not require any code porting efforts. We name this property as **network-agnostic**. Moreover, a batch-level parallelization does not change any training parameter, thus it does not affect the convergence of the gradient descent. This is important, because parallelization strategies that do change the training parameters do not ensure convergence invariance. For instance, multi-GPU systems have been explored when the batch and model parameters do not fit in the memory of a single GPU. The solution has been to reduce the batch size and make it fit in memory. But this changes the batch size and alters the convergence. In general, doubling the number of GPUs having halved the batch size is never a guarantee of observing a 2x speedup for the training process [15]. We name this property as **convergence-invariance**.

PPoPP '16, March 12-16, 2016, Barcelona, Spain.

http://dx.doi.org/10.1145/2851141.2851158

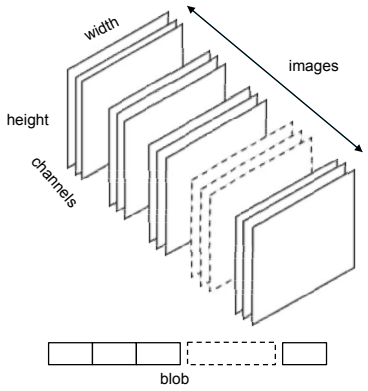

Figure 1. Example of blob structure and data segments within the blob. Each image is composed of data of three channels. Each channel is stored in one blob segment, so that every image occupies three blob segments. Images are stored sequentially one after the other within the blob.

The main contribution of this paper is the implementation and analysis of a **network-agnostic** and **convergence-invariance** coarse-grain parallelization of the DNN training algorithm. It exploits a parallelism level that is independent from the support of specialized and optimized libraries. Therefore, the optimization is immediately available for accelerating the DNN training. And it is compatible with multi-GPU execution without altering the algorithm convergence rate. The parallelization has been implemented in Caffe [17], a state-of-the-art DNN framework. The paper describes the code transformations for the parallelization and we also identify the limiting performance factors of the approach and expose appropriate optimizations. We show competitive performance results for two state-of-the-art computer vision datasets, MNIST [11, 22] and CIFAR-10 [18]. In particular, in a 16-core Xeon E5-2667 at 3.30GHz we observe speedups of 8×, at similar performance levels of those obtained by the GPU optimized Caffe version in a NVIDIA K40 GPU.

The rest of this paper is organized as follows: Section 2 describes the main design aspects of Caffe regarding its parallelization. Section 3 describes the code transformations and optimizations for a batch-level parallelization of the training core of Caffe. Section 4 analyses the performance limiting factors of the approach and its overall performance. Section 5 presents the related work and finally Section 6 concludes the paper with its main conclusions.

2. The Caffe DNN Framework

Caffe is a deep learning framework for research in deep neural networks. It is widely used across the DNN community and has become one main platform for neural network algorithm development. Two main reasons justify this. Firstly, Caffe supports many network architectures and different training algorithms for neural networks. Secondly, Caffe is optimized with GPU acceleration. All supported layers in Caffe have two implementations, one for CPU one for GPU. This section describes the main operational aspects of Caffe regarding its parallelization. The objective is to describe the neural network training algorithm in terms of how the computations happen across the network, how the data flow across these layers and the different types and levels of parallelism that exist within each layer.

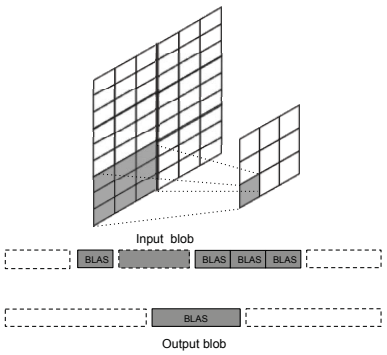

Figure 2. Example of layer transformation and its corresponding blob organization. Every 9 input segments generate the content of one segment in the output blob. This scheme is usual in deep neural networks for dimensionality reduction purposes. Pooling layers (e.g.: Caffe AVERAGE or MAX pooling layers) perform such type of computations.

2.1 Neural Networks in Caffe

Caffe allows a user to specify the network structure in a prototext format [14] that can capture any kind of arbitrary DAG (directed acyclic graph). Caffe allows users to define their own *networks* and each network is composed of a set of *layers*. Each layer has a pre-defined generic interface. As users compose a network of layers it becomes very easy to define extremely complicated graphs in a very compact notation. Caffe also allows users to specify a solver and its parameters to implement the neural network training. The training process uses back propagation and implements several solver algorithms such as SGD[4], ADAGRAD[13] and NESTEROV[23].

In a feed-forward network each layer is stacked such that the output of one layer becomes the input to the next immediate layer in the network graph. The first layer processes the input (Data) that is fed to the neural network. Each of the subsequent layers apply a data transformation according to their specific computation. The output of Caffe is the set of the coefficients in each layer after the network training process is completed. Regarding the parallelization of the training process, the main aspects to consider are how the input/output data flow across the network, the structure of the computation in each layer and the training algorithm itself.

2.1.1 Input-Output Data

Caffe stores and communicates data using *blobs*. Blobs provide a unified memory interface holding data; e.g., batches of images, model parameters, and derivatives for optimization. Mathematically a Blob is an N-dimensional array stored in a C-contiguous fashion. Blobs also conceal the computational overhead of mixed CPU/GPU operation by synchronizing from the CPU host to the GPU device as needed. The conventional blob dimensions for batches of image data are number N x channel K x height H x width W. For example, if a network is trained on image data, the input data would be organized as a 4-dimensional blob and the value at index (n, k, h, w) is physically located at index ((n * K + k) * H + h) * W + w within the sequential Blob data structure. The first dimension corresponds to the image index in a batch of images and the other three indicates the number of channels (e.g. 3 for an RGB encoded image) and the image dimensions (height and width). Figure 1 shows an example of an input blob. For this case, the blob stores one image with 3 data segments, one per each image channel. Images are sequentially stored in the blob following this pattern.

2.1.2 Layer Computation

Each layer performs a data transformation by operating on the input blobs and generates output blobs. These operations are computed in a piecewise manner throughout the input blob. The input blobs are organized in data segments where the linear algebra operations are applied. The size of the segments is constant and layer dependent. The output of the computation are blobs, again organized in fixed size segments. The computation of a layer is built upon an iterative structure that traverses the segments in the input blobs and applies a specific transformation based on one or more BLAS computations on each segment. The output of the processing of one input data segment is stored in one segment in the output blob. Figure. 2 shows an example for this computational scheme as well as the relation between the input and output blobs for a layer computation. Shadowed patches correspond data segments in the blobs. In this case, 9 data segments in the input blob are used to compute one segment in the output blob.

In general, such transformations are implemented using basic algebra operations like matrix-×-vector or matrix-×-matrix products and are often referred to as Level 2/3 BLAS operations [3, 12]. A layer can be understood as a procedure based on several functions in the form of $f_i(x, W_i, b_i) = W_i x + b_i$. The layer coefficients would correspond to the matrices W_i and vectors b_i. Input x would correspond to a data segment in the input blob. The training process of a neural network optimizes the W_i and b_i coefficients so that the network accuracy is maximized for a particular data model.

2.1.3 Neural Network Training

The training process of a neural network is based on the gradient descent algorithm. The algorithm continuously seeks the minimization of a cost function along training epochs, where one epoch constitutes of processing all the training data samples. These are grouped in batches of fixed size and during every epoch each batch is processed in two steps. First, the batch samples are used to compute the average error of the network. The samples traverse the network where each layer applies its transformation and temporally stores its output in a blob. This phase is identified as the *forward pass* of the network. Second, the network computes the gradient of its transformation as a whole. In this phase, every layer applies the chain rule for propagating the derivatives across the layers in the backward direction. From the last layer up to the first one in the network, each layer propagates its output multiplied by its derivative value. This phase is identified as the *backward pass*. Both the forward and backward pass are inherently sequential as both the output and the gradients have to be propagated through the layers in an upward and downward manner across the network. Algorithms 1, 2 and 3 show the high level structure of the training algorithm as well as how the data flow across the layers in the forward and backward pass.

Algorithm 1 corresponds to one training iteration. Each data batch *b* is propagated through the network. First layer processes the batch and outputs the first transformation for each sample in it. This is then forwarded to the second layer to apply the next transformation. This process continues until all layers have applied their transformation. The last layer of the network is the one responsible to check the output of the network and evaluate the network accuracy. The loop in line 3 implements the batch network computation, the network forward pass. This loop is inherently sequential and each layer transformation happens within the *forward* call. The output of this call is stored in vector *top* for all data samples in the batch. Each *forward* invocation uses as input the *top* produced by the previous layer. Loop in line 6 corresponds to the computation of network gradients. Once the network has been evaluated with one batch, the algorithm computes every layers gradient with respect its input. This process seeks the minimization

Algorithm 1: Iteration of the DNN training algorithm.

input : Set of B batches of S data samples each
output: Network layer coefficients

```
1  begin
2      while loss not acceptable do
3          for b ← 1 to B do
4              for s ← 1 to S do
5                  top[1]=layers(1)→ forward(batches[b][s]);
6                  for l ← 2 to L do
7                      top[l]=layers(l)→ forward(top[l-1]);
8                  diffs[L]=layers(L)→ backward(top[L]);
9                  for l ← L − 1 to 1 do
10                     diffs[l]=layers(l)→
                       backward(top[l],diffs[l+1]);
11                 updateCoefficients(layers, top, diffs);
12         loss = evaluateNetwork(layers);
13 end
```

of the network error. This loop is also inherently sequential and the *backward* call computes the layer gradient, *diffs*. This process requires both the evaluation of the layer transformation (*top*) and the previous gradient computed in the immediate previous layer (*diffs*) in backward manner. It corresponds to the network backward pass. Once both the forward and backward passes are completed, the training algorithm updates the network coefficients in all layers. This corresponds to line 8 and procedure call to *updateCoefficients*.

Algorithm 2: Layer forward pass

input : Bottom blob (S, N+1, $D_1, D_2, \ldots, D_N$)
output: Top blob for the layer transformation

```
1  for s ← 1 to S do
2      for d1 ← 1 to D1 do
3          for d2 ← 1 to D2 do
4              ...
5              for dN ← 1 to DN do
6                  top[f(s, d1, d2, ..., dN)]=BLAS(W(d1, d2,
                   ..., dN), b(d1, d2, ..., dN), bottom[g(s,
                   d1, d2, ..., dN)]);
```

Algorithm 2 corresponds to the description of the structure of one layer transformation. In general, the layer computation is organized in a nest of loops that traverse the *N+1* dimensions (*S,* D_1*,* D_2*, …,* D_N) of the layer input blob *bottom* and produces a transformation stored in the output blob *top*. The first dimension corresponds to the data samples in the batch. These are indexed with the variable *s*. According to layer specific functions (*f* and *g* in line 5 of the algorithm), data segments in the input/ouput blobs are processed through basic linear algebra transformations (*BLAS* call in line 5) that are also dependent to the data segment being processed (*W* and *b* depend on the loop induction variables). Algorithm 3 follows a very similar structure, but operating with blobs that store the gradient computation (*diffs_bottom* and *diffs_top*).

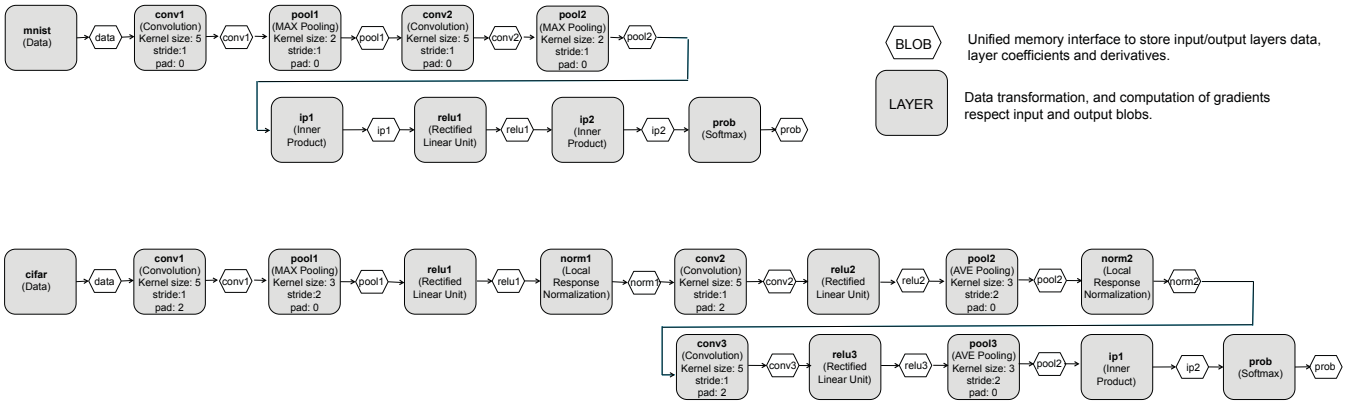

Figure 3. MNIST network: composed of 9 layers, organized in two sections. First data plus convolutional and pooling layers. Second, inner product and rectified linear units and loss. CIFAR network: composed 14 layers, organized in two sections. First, data plus convolutional, pooling, rectified linear units and local response normalization layers. Second, pooling and inner product and loss.

Algorithm 3: Layer backward pass

input : Gradient w.r.t. top blob (S, N+1, $D_1, D_2, \ldots, D_N$)
output: Gradient w.r.t. to bottom blob

1 **for** $s \leftarrow 1$ **to** S **do**
2 **for** $d_1 \leftarrow 1$ **to** D_1 **do**
3 **for** $d_2 \leftarrow 1$ **to** D_2 **do**
4 ...
5 **for** $d_N \leftarrow 1$ **to** D_N **do**
6 diffs_bottom[f(s, $d_1, d_2, \ldots,$ d_N)]=**BLAS**(W($d_1, d_2, \ldots, d_N$), b($d_1, d_2,$ $\ldots, d_N$), top[h(s, $d_1, d_2, \ldots, d_N$)], diffs_top[g(s, $d_1, d_2, \ldots, d_N$)]);

2.2 Network Examples: MNIST and CIFAR-10

The MNIST [11, 22] and CIFAR-10 [18] datasets are standard datasets for computer vision. The MNIST dataset is composed of 60,000 images, each of dimension 28×28 pixels of handwritten digits for training and 10,000 test examples. The Caffe distribution includes the LeNet [21] network for generating an image classifier for this dataset. Given a handwritten digit, the network outputs which digit is represented (e.g.: 0 . . . 9).

The CIFAR-10 is a CNN (Convolution Neural network) [20] included in the Caffe distribution that performs as an image classifier for the CIFAR dataset. CIFAR is composed of 60,000 32×32 color images with 10 classes: *airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck* with 6000 images per class. Both the networks are built with a subset of the available layers within Caffe.

2.2.1 Networks

The aim of this section is to briefly introduce the two networks used in this paper. The section describes the network layers and characterizes them in terms of feature learning layers and dimensionality reduction layers. Regarding the parallelization, it is important to notice that deeper the layers, smaller the size of the input/output blobs. Thus, the level in the network will affect the work granularity of the existing parallelism within the layer.

Figure 3 shows the layer composition for the MNIST network. This network is composed of an initial data layer followed by the combination of two instances of convolutional and pooling layers. The convolutional layers are responsible for feature learning while the pooling layers correspond to dimensionality reduction. This is followed by a inner product layer (ip) and ReLU layer which furthermore reduce the dimensionality of the data traversing the network, until the last inner product layer which outputs a vector of 10 elements, one per digit class. The last layer corresponds to a loss layer.

Figure 3 shows the layer composition for the CIFAR network. As in the previous network, the first layer is a data layer, followed by the combination of convolutional, ReLU and pooling layers. Again, convolutional layers perform the feature learning and pooling layers perform the dimensionality reduction. The ReLU layer controls the saturation of the output of the convolutional coefficients. CIFAR also includes normalization layers between the first two instances of convolution and pooling layers. The inner product layer (ip) and loss layer (SoftMax) are the last layers of the network.

3. Caffe Coarse Grain Parallelization

This section describes the parallelization process for a coarse-grain parallelization of the DNN training process in the Caffe DNN framework. First, we identify the different sources of parallelism and their granularity. Second, we describe the methodology to generate a coarse-grain parallelism version of the algorithms discussed previously in section 2. The coarse-grain approach targets a CPU execution. Finally, we compare our approach to that of a fine-grain parallelization targeting a GPU execution.

3.1 Sources of Parallelism

Given the structure of the computation in algorithms 1, 2 and 3, the training procedure of a neural network exposes several levels and types of parallelism.

3.1.1 BLAS Level Parallelism

This level of parallelism corresponds to the computations that are based on basic linear algebra operations. It appears in the BLAS computations executed for each data segment in the input/output blobs. In algorithms 2 and 3 the BLAS calls in line 6 in both cases hide this level of parallelism. In general, these computations correspond to matrix and vector operations like matrix-×-matrix products. Caffe includes a native and limited BLAS implementation that can be substituted with specialized libraries like Atlas, OpenBLAS or Intel MKL [3, 12]. These implementations are highly optimized and exploit both thread-level parallelism and SIMD level parallelism with vectorized code.

3.1.2 Blob Level Parallelism

This level of parallelism corresponds to the nested loops in the forward and backward passes in lines 2 and 5 in algorithms 2 and 3. These loops traverse the data segments in the input and output blobs and perform BLAS calls for each segment. This kind of parallelism can be achieved with thread-level parallelism as each BLAS call can be executed in parallel. In general, the dimensions of the input blob are layer dependent. Therefore the number of iterations in each loop level is also layer dependent. All loops are interchangeable and parallel, given the appropriate data privatization. So, in order to have control over the most appropriate loop scheduling, loops can be rearranged in different manners.

3.1.3 Batch Level Parallelism

This level corresponds to the loop in line 1 of algorithms 2 and 3. This loop traverses the data samples in the batch and its parallelism can be exploited but with limitations. Specifically for the backward pass, it requires reduction operations for the network coefficient update. The training algorithm averages all gradients computed with each sample in the batch. This averaging has to be protected with mutual exclusion operations or ordered loops plus data privatization for the blobs that temporally store the gradients.

Algorithm 4: Coarse-grain parallel layer forward pass

input : Bottom blob (S, N+1, $D_1, D_2, \ldots, D_N$)
output: Top blob for the layer transformation

```
1  #pragma omp parallel {
2  /* Object Privatization */ ...
3  #pragma omp for private(s, d_1, d_2, ...)
4  for civ ← 1 to S * D_1*D_2*...*D_k do
5      s = f_s(civ);
6      d_1 = f_1(civ);
7      d_2 = f_2(civ);
8      ...
9      d_k = f_k(civ);
10     for d_{k+1} ← 1 to D_{k+1} do
11         ...
12         for d_N ← 1 to D_N do
13             top[f(s, d_1, d_2, ..., d_N)]=BLAS(W(d_1, d_2, ...,
               d_N), bias(d_1, d_2, ..., d_N), bottom[g(s, d_1, d_2,
               ..., d_N)]);
14 }
```

3.2 Code Transformation

The Caffe framework is designed using object-oriented methodologies and uses a C++ implementation. Its main data structures and algorithms have been designed with functional principles with the aim of giving support for neural network development. In this paper, we followed a methodology to parallelize Caffe so that these principles are kept intact. We wanted to minimize code reorganization for the optimization process. To indicate the parallelism to be exploited, only directive-based transformations have been applied. We used OpenMP [2] directives to indicate which loops have to be parallelized. Whenever possible, we used the OpenMP primitives to indicate data privatizations and reduction operations, as well as the necessary synchronization points along the parallel code. To increase performance, we applied very simple manual code transformations like loop coalescing or loop interchange. Therefore, in order to enable a coarse-grain parallelization, no object data structure has been modified and the implementation of every Caffe layer has not been recoded with any platform-specific optimization.

3.2.1 Coarse grain parallelization and optimizations

Algorithm 4 shows a high-level description of a coarse grain parallelization of the forward layer transformation described in section 2. The parallelization is specified with directive-based annotations following the OpenMP syntax and semantics. Coarse parallelism is defined by a parallel region that englobes all the layer code (lines 1-16). The original loop nest appears with a coalescing transformation where some of the outermost loops have been collapsed into a single loop statement (line 6). A parallelizing directive for this loop is inserted (line 5) specifying the parallel execution of this loop under a static loop scheduling (default scheduling for OpenMP [2]). The loop coalescing transformation is related to the parallelization process and is done to have effective control over the work distribution under a static loop scheduling. To understand this issue, recall the code structure in Algorithm 2. In that version, the outermost loop corresponds to the loop that processes every data sample in the batch (loop with variable *s*). In general, after the parallelization under a static scheduling, one iteration is the minimal work unit for work distribution. If that loop were parallelized, the amount of work per iteration is coarse enough so that any difference in the number of assigned iterations per thread can cause a significant work unbalance. To maintain the coarse level parallelization, but minimize the size of the work unit, the coalescing transformation increments the total number of iterations, but every iteration having a smaller amount of work. In general, the number of coalesced loops is layer dependent. Some of the layers are parallelized with no coalescing, other coalesce the whole loop nest. In general, this optimization was made manually. Although that, some of the loops define a perfect nest so that they could be automatically transformed by the usage of the *collapse* construct in OpenMP.

Algorithm 5 shows the coarse-grain parallelization of the backward layer pass. For parallelism specification and work distribution, the transformations are the same as in the previous case for the layer forward pass. The same coalescing transformation has been applied and parallelism has been specified at the same levels. The number of coalesced loops is layer dependent, and this optimization has been manually performed. The main difference resides on the special treatment for the gradient update (*diffs_top* and *diffs_bottom* variables). Recall that the DNN training algorithm averages all gradients computed within a batch. This happens during the layer backward pass. Now, this computation has been parallelized, so the gradient update has to be implemented through a reduction operation and thread mutual exclusion mechanisms. In the algorithm, this corresponds to lines 18-20. An ordered loop is added so that every thread incorporates its gradient computation to the global variable that stores the gradients. This mechanism requires the per-thread privatization of the blob storing the gradients. The private storage for the gradients has to be properly initialized to the neuter value of the reduction operation, in this case the zero value (lines 4-5 in the algorithm). Notice the usage of the ordered construct. A reduction-based solution would also be valid, but would not ensure the same update value with any number of threads. In particular, only the ordered execution will produce the value obtained through the sequential execution. Along the research cycle for neural networks, this issue is critical for tuning and debugging purposes. In general, neural network developers use the loss vaule to monitor the correct evolution of the training process. Many parameters affect the convergence of the SGD algorithm. The parallelization of the SGD algortihm

adds another one: it happens that the gradient computation affects the overall loss value for the neural network training. Therefore, during tuning and debugging stages developers prefer to keep the sequential update for the gradient computation. Once appropriate convergence is ensured, the reduction-based solution would also be applicable.

Algorithm 5: Coarse-grain parallel layer backward pass

input : Gradient w.r.t. top blob (S, N+1, $D_1, D_2, \ldots, D_N$)
output: Gradient w.r.t. to bottom blob (M, $O_1, O_2, \ldots, O_M$)

```
#pragma omp parallel
{
/* Object Privatization */
Blob private-diffs_bottom(M, O_1, O_2, ..., O_M);
caffe_zero(S * O_1*O_2*...*O_M, private-diffs_bottom);
...
#pragma omp for private(s, d_1, d_2, ...)
for civ ← 1 to S * D_1*D_2*...*D_k do
    s = f_s(civ);
    d_1 = f_1(civ);
    d_2 = f_2(civ);
    ...
    d_k = f_k(civ);
    for d_{k+1} ← 1 to D_{k+1} do
        ...
        for d_N ← 1 to D_N do
            private-diffs_bottom[f(s, d_1, d_2, ..., d_N)]=
            BLAS(W(d_1, d_2, ..., d_N),
            bias(d_1, d_2, ..., d_N),
            top[h(s, d_1, d_2, ..., d_N)],
            diffs_top[g(s, d_1, d_2, ..., d_N)]);
#pragma omp for ordered
for th ← 1 to omp_get_num_threads() do
    caffe_average(diffs_bottom[th], private-diffs_bottom, ...);
}
```

In terms of memory utilization, the batch-level parallelism requires extra memory related to data privatization. In particular, the gradient computation needs temporal storage per each thread. The amount of additional memory is determined by the number of coefficients computed by every layer. This memory allocation happens within the layer implementation, and the memory usage never crosses the layer boundaries in the network. This means that the temporal storage can be reused accross layers, so that the total extra memory is determined by the layer with more coefficients. For the studied networks (CIFAR-10 and MNIST), these correspond to the convolutional layers which depend on the size of the convolutional filters. In general, this type of layers never required more than 1250KB (CIFAR-10) and 640KB (MNIST) of additional memory (16 threads configuration). This corresponds to an increment in the order of 5% given that the sequential executions allocate 36MB (CIFAR-10) and 8MB (MNIST) of total memory.

3.3 Coarse-grain vs Fine-grain parallelization

The aim of this section is to describe the main differences between the coarse-grain and fine-grain approaches when it comes to the parallelization of a layer transformation. In the coarse-grain case the outermost loop is parallelized after appropriate loop coalescing. This loop corresponds to the batch-level parallelism, so coarse-grain and batch-level parallelism are equivalent. In contrast, the fine-grain approach focusses on the innermost loops. This approach coalesces as much as possible inner loops so that enough work is generated for the fine-grain threads It is this process which requires considerable programming efforts. The inner loop coalescing has to be done keeping in mind the work distribution mechanisms available for fine-grain parallelism. Specifically for the CUDA case, here we refer to the thread *blocks* and *grids* concepts. In general, the more loops are coalesced, the more effective the parallelization. Besides, optimal data layouts have to be designed so that this work-thread association translates into an efficient data-thread association. After this loop transformation, the resulting loop has to be extracted to generate a GPU kernel and introduce all necessary data transfers between host and guest devices before and after the kernel execution. Clearly, when compared to the coarse-grain parallelization, the programming efforts are much more significant in the case of the fine-grain parallelization. At this point, what we identified as network-agnostic feature becomes evident and important. Because the coarse-grain transformations are done from the outermost loop (batch-level parallelism) to the inner loops, the recoding efforts are independent of the layer transformation. The programmer is required to parallelize the batch-level loop without any other consideration than applying the necessary data privatization. In contrast, the fine-grain parallelization forces the programmer to know about the exact nature of the layer computation to design the most optimal data layout and work distribution schemes.

4. Performance Analysis

We evaluated the coarse-grain parallelization in the Caffe DNN framework. All experiments have been performed in a 16-core Xeon E5-2667v2 at 3.30GHz with a NVIDIA K40 GPU. The machine runs a Red Hat Enterprise Linux Server release 6.6 (Santiago), and we used the GCC compiler suite version g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-11). We configured the Caffe framework to use OpenBLAS for the implementation of basic linear algebra subroutines. For the GPU programming, we used the CUDA toolkit 7.0 and cuDNN v2. The datasets used for the evaluation are the MNIST and CIFAR-10 image classifiers, which are available within the Caffe framework. We did not introduce any architectural dependent optimization. Therefore, the portability of both the coarse-grain and fine-grain approaches is ensured by the OpenMP, CUDA and cuDNN portability.

4.1 MNIST dataset

For the performance analysis of the MNIST dataset we first developed a per-layer study of both the coarse-grain and the fine-grain parallelizations. For the coarse-grain case we identify what are the main limiting performance factors. Then we describe the overall performance of both versions.

4.1.1 CPU Layer Performance

Figure 4 shows the absolute execution time per-layer and the relative weight in the overall execution time. Horizontal bars correspond to executions with 1, 2, 4, 8, 12 and 16 threads. In general, two layers dominate the whole execution: the convolutional and pooling layers. No matter the number of threads, these two type of layers always account for almost 80% of total execution time, adding their forward and backward passes. Notice that there are different instances of the

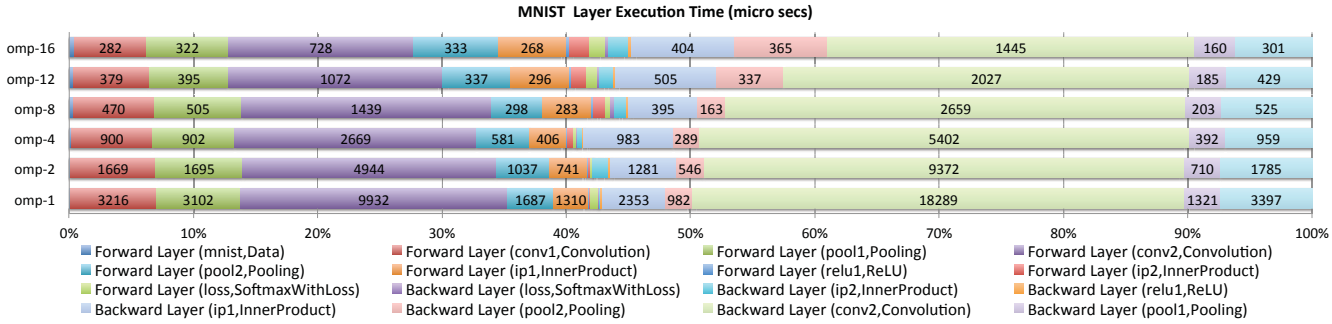

Figure 4. MNIST - Relative and absolute execution layer time for CPU executions. Layers in the legend are ordered from left-to-right in each horizontal bar. Horizontal bars correspond to the cases of 1, 2, 4, 8, 12 and 16 threads. All execution times are in microseconds.

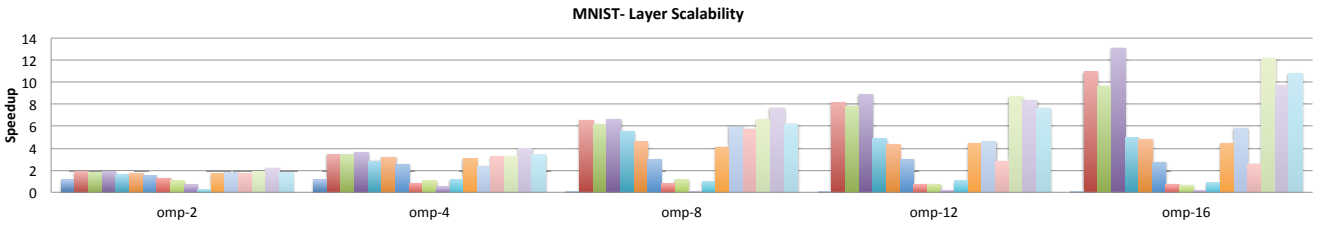

Figure 5. MNIST - Layer scalability for the CPU executions. Layers are identified as in the legend for Figure 4. Layers in the legend are ordered from left-to-right in each cluster. Clusters correspond to the cases of 2, 4, 8, 12 and 16 threads. Y-axis measures speedup factors from the serial CPU execution.

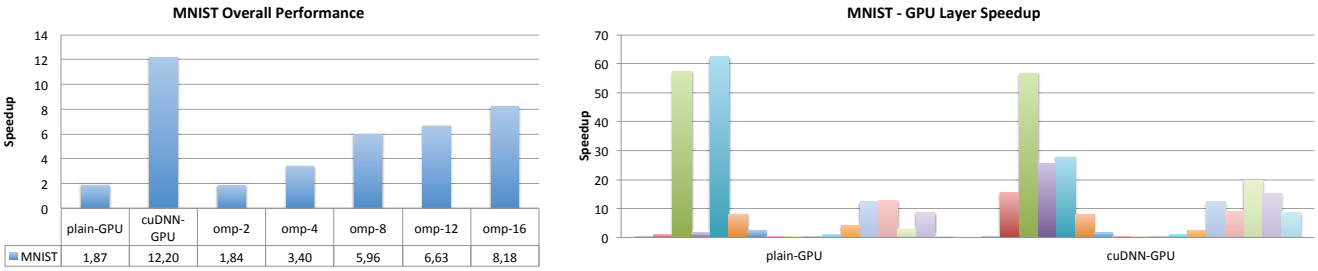

Figure 6. MNIST - **Left side**: Absolute speedup factors for OpenMP (2, 4, 8, 12 and 16 threads), plain-GPU and cuDNN-GPU versions. performance. **Right side**: GPU layer scalability for plain-GPU and cuDNN-GPU versions. Layers are identified as in the legend for Figure 4.

same type of layer but with a very different absolute execution time. For instance, the conv1 and conv2 layers, and in smaller magnitude, the pool1 and pool2 layers. After the convolutional and pooling layers, the next significant layer is the inner product ip1. The rest of layers, expose a very small contribution to the overall execution time (e.g.: loss, ReLU and ip2 layers in their forward and backward passes). In general, notice that in each horizontal bar there is a zone where the work in each layer phase decreases. This corresponds to the center part, composed of the forward and backward passes of pool2, ip1, ReLU, ip2 and loss. This behavior is associated to the dimensionality reduction that neural networks do, and affects the work granularity for the parallelization process. Moreover, this behavior limits the overall scalability of the training process.

Figure 5 shows the scalability curve of each layer. We identify three layer behaviors. First, notice the u-shape of the scalability trends for any number of threads. The center points correspond to the layers that we previously identified as not significant in the overall execution time (e.g.: loss, ReLU and ip2 layers). These layers do not scale at all, but they do not represent a limitation for the overall performance. The two sides on the center values correspond to the forward and backward passes of the rest of layers. For these, we detect two types of layer behavior.

Layers ip1 and pool2 present very poor scalability curves. For ip1, in both the forward and backward passes, the layer presents speedups of 4.58 and 5.93 for the forward and backward pass respectively and with 8 threads. The layer does not improve the speedup with more threads. The pool2 layer exposes the same behavior with maximum speedup of 5.52 and 5.73 with 8 threads in the forward and backward pass respectively. The reason for this behavior is two-fold. First, notice the two layers are immediately stacked one on top of the other. This means that the output of the pool2 layer is the input for the ip1 layer. It happens that the blob shapes between the two layers do not match. The pool2 layer is parallelized according to its input blob dimensions, and produces its output blob (input blob for ip1) following the resulting work and data distribution coming out from the parallelization. When it comes the execution of the ip1 layer, its parallelization is done according to its input blob dimensions (output blob of pool2), which do not

match those of the input blob for the pool2 layer. Thus, there is an unavoidable lost of locality for the execution of the ip1 layer. Second, both layers suffer from a poor granularity when executing with more than 8 threads. In Figure 4 we observe that with more than 8 threads the forward and backward passes of the two layers are in the range of 350 microseconds.

Layers conv1, pool1 and conv2 present good scalability curves. They correspond to the layers in both sides of the center part of the scalability layer curves. In general, these layers respond well to the increments on the number of threads. This is explained by two factors. First, all these layers expose a considerable amount of work, as it has been indicated with Figure 4. Second, all of them are stacked one next to each other within the network, and all of them match their input/output blob dimensions. Thus, data locality is preserved along the their execution in both the forward and backward passes. We detected that although being exactly the same layer, the conv1 and conv2 layers have close to a 10% difference in speedup. In particular, the conv2 layer exposes greater speedups than conv1. This happens more noticeably with more than 8 threads and for the forward pass. The difference between the two layers is the position they occupy in the layer stack. Conv1 is the immediate layer after the data layer in the MNIST network. The data layer is responsible for feeding the network with data samples. The layer executes sequentially, so the data associated to all images generates a memory footprint that is not matching the one generated along the parallel execution of the conv1 layer. Therefore, the conv1 layer suffers from a poor data locality with its immediate previous layer.

4.1.2 GPU Layer Performance

The fine-grain layer parallelization is available in Caffe in two versions. All available layers come with a native GPU implementation of their forward and backward pass. We identify this version as the *plain-GPU* version. Specifically for the convolutional and pooling layers, Caffe includes a cuDNN-based version. We identify this version as the *cuDNN-GPU* version. Rightmost side of Figure 6 shows the per-layer speedup numbers for the plain-GPU and cuDNN-GPU versions. For the plain-GPU version, all layers present speedups below the 10× bar except for specific exceptions. The pool1 and pool2 layers expose extraordinary speedups of 57× and 62× for their forward passes respectively. The pool2 layer presents a speedup of 12.81× in its backward pass and the ip1 layer is also above the 10× bar with a speedup of 12.25× in its backward pass. In contrast, the convolutional layers present a very poor speedup, with 1.11, 1.63 for the forward passes of conv1 and conv2, and 0.43× and 2.86× in their respective backward passes.

For the cuDNN-GPU, the results are similar, except for the convolutional and pooling layers. The conv1 and conv2 layers experiment an extraordinary improvement reaching speedups of 15×, 25×, 19× and 8× in their forward and backward passes. In contrast, the pool2 layer experiments a dramatic loss of performance: it drops from 62× to 27× in its forward pass and from 12.81× to 8.81× in its backward pass. More moderately, the ReLU layer also suffers a performance drop from 2.47× to 1.74× and from 4× to 2,41× in its forward and backward passes respectively. In general, cuDNN corresponds to a case where the industry has deployed a highly optimized implementation of layer transformations that are well understood and no longer in a research stage. In this situation, the fine-grain parallelism makes a difference, after the corresponding recoding efforts.

4.1.3 Overall Performance

Figure 6 shows the overall performance of the coarse-grain parallelization and the fine-grain parallelization in its two versions GPU and GPU-cuDNN. The coarse-grain reaches a speedup close to a 6× with 8 threads, and 8× with 16 threads. The lack of the scalability for the CPU version is related to the poor scalability of fine-grained layers that when executing with 16 threads drag down the performance. In addition, we suspect the serial initialization of the network structures is giving a suboptimal memory allocation in the NUMA nodes. All of this affects the final scalability of the coarse-grain version. The fine-grain GPU version shows a modest speed up close to 2×. The reason for this difference is related to the performance of the convolutional layers. In general, this version corresponds to a base line defined by the Caffe native implementation of the GPU acceleration. It represents a case for the performance the DNN community can obtain with a fine-grain parallelization and very significant coding efforts. Remember that within Caffe, all layers must have both a CPU and GPU implementation to guarantee GPU acceleration. In conclusion, the coarse-grain approach minimizes the coding efforts and delivers better performance levels. Of course, when compared to the cuDNN case, the fine-grain approach makes a difference. It delivers a 12× speedup. But solutions like the cuDNN framework are only available when the layer types and their implementation have become a product and are no longer in a research stage. Thus, they can have a highly optimized implementation. For a DNN framework like Caffe, which is aimed to give support for research in new network architectures with novel layer types, the fine-grain parallelism imposes hard recoding efforts. In contrast, the coarse-grain option is much more immediate and effective.

4.2 The CIFAR-10 case

For the performance analysis of the CIFAR-10 dataset we followed the same methodology as for the MNIST case. First, we developed a per-layer study for the coarse-grain and fine-grain parallelizations. For the coarse-grain case we identify what are the main limiting performance factors. Then we describe the overall performance of the coarse-grain parallelization.

4.2.1 CPU Layer Performance

The CIFAR-10 dataset generates a work granularity greater than the MNIST case. This can be observed in Figure 7. The figure shows per-layer absolute execution time and the relative weight in the overall execution time. It is clear that just a few layers dominate the execution, no matter the number of threads. These layers are the convolutional and pooling layers, and with lesser weight, the local response normalization layers. In general, these layers account for almost 85% of total execution time in all thread configurations. Therefore, the layers with very small granularity and exposing very poor scalability curves will not determine the overall performance. Only the scalability of these dominating layers will determine the effectiveness of the parallelization.

Figure 8 shows the scalability curves of all layers. Notice the appearance of the u-shape form as long as the number of threads increases. The center part of each cluster corresponds to layers of very small granularity which do not affect the overall performance. These layers are the pool3, ip1 and loss. The center part includes both the forward and backward pass of these layers. Leftmost part of each cluster corresponds to the layer forward pass, the rightmost part corresponds to the layer backward pass.

The CIFAR-10 network is organized in three levels all of them with a similar organization. First level corresponds to a sequence of a data layer plus conv1+pool1+ReLU1+norm1. During the forward pass, the data layer fetches the input data (e.g: the batch images) sequentially so the conv1 layer suffers from poor locality respect its input data (the same situation observed with the MNIST case). The work distribution is constant across the conv1, pool1 and ReLU1 layers, and then changes for the norm1. According to this, the conv1 has a reasonable speedup up to 8 threads (5.87×) but for 16 threads the scalability drops with a 9× speedup. This is explained by the sequential execution of its immediate previous

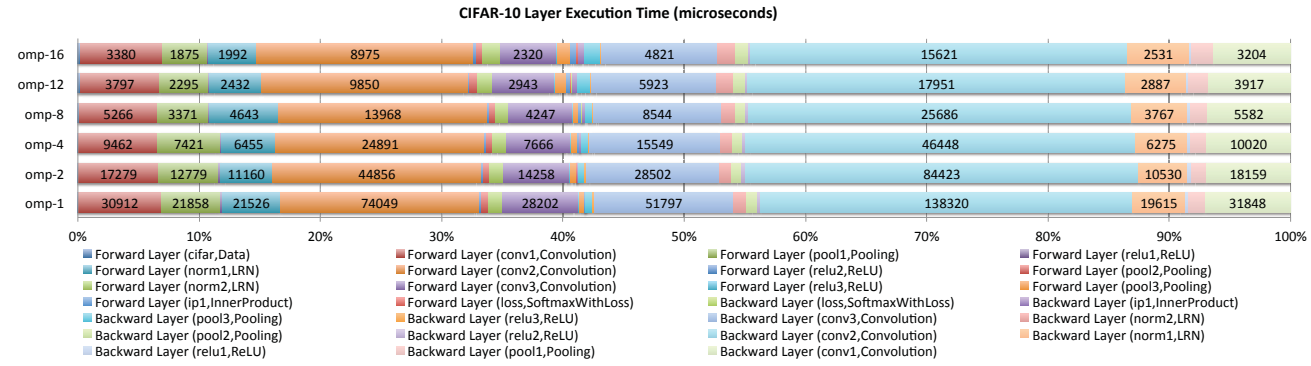

Figure 7. CIFAR-10 - Relative and absolute execution layer time for CPU executions. Layers in the legend are ordered from left-to-right in each horizontal bar. Horizontal bars correspond to the cases of 1, 2, 4, 8, 12 and 16 threads. All execution times are in microseconds.

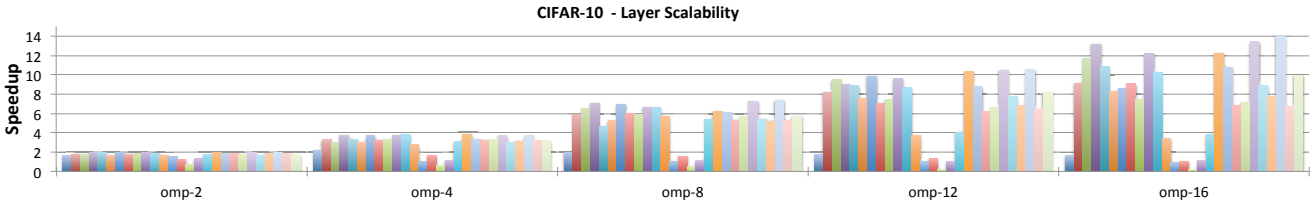

Figure 8. CIFAR-10 - Layer scalability for CPU executions. Layers are identified as in the legend for Figure 7. Layers in the legend are ordered from left-to-right in each cluster. Clusters correspond to the cases of 2, 4, 8, 12 and 16 threads. Y-axis measures speedup factors from the serial CPU execution.

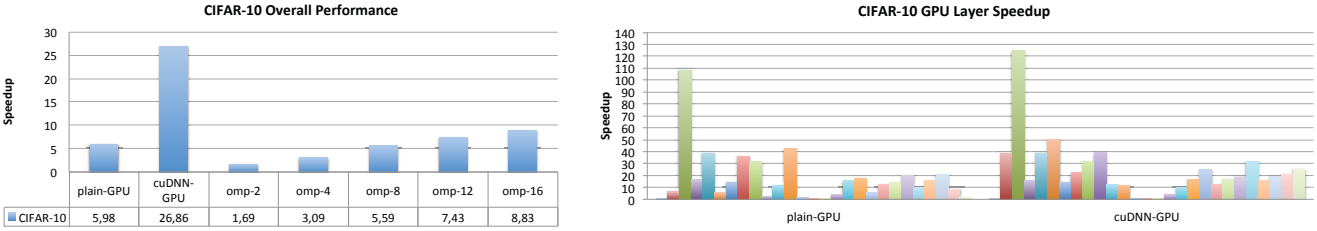

Figure 9. CIFAR-10 - **Left side**: Absolute speedup factors for OpenMP (2, 4, 8, 12 and 16 threads), plain-GPU and cuDNN-GPU versions. performance. **Right side**: GPU layer scalability for plain-GPU and cuDNN-GPU versions. Layers are identified as in the legend for Figure 7.

layer and by the fact that when crossing the 8 thread border, NUMA considerations come into play. The effects of data movement are much more visible than when just executing in one NUMA node. After the conv1 forward execution, the pool1 and ReLU1 layers keep the same work distribution and expose reasonable speedups: 6.5× and 7× respectively with 8 threads. These layers scale up to 11× and 13× with 16 threads. The norm1 layer exposes a different trend. This layer executes changing the data-thread distribution. With 8 threads reaches a 4.6× speedup and with 16 threads 10.8×. Regarding the backward pass of all the layers in this first level, the relation between the layers are similar, but with less scalability. The maximum speedup for 16 threads are 10×, 6.6×, 7.75× for the conv1, pool1, ReLU1 and norm1 layers respectively. The reduction operations within the backward pass are negligible, given the work size in each layer for the CIFAR-10 case. This happens in all the network levels.

The second level in the network is composed of layers conv2, ReLU2 and norm2. In this level, the sizes of the input/output blobs decrease and the work granularity too, but with the exception of layer conv2. This working set size reduction affects the overall scalability, specially with 16 threads. In this level, the maximum speedups are 8.25×, 8.5×, 9× and 7×. In this case, the data-thread relation between the layers is constant, except for the norm2 layer. Specifically for the first layer in this level, the conv2 layer, its poor scalability is related to its immediate previous layer, the norm1 layer. This layer changes the data-thread distribution and the conv2 layer is affected by this fact. For the backward pass, the performance trends are similar, and again the reduction operations do not limit the overall layer scalability.

Finally, the third level is composed by the conv3, ReLU3 and pool3 layers. This level follows a similar performance trends as the second level, with the same relations between its layers and between the immediate layer in the second level.

4.2.2 GPU Layer Performance

Rightmost side of Figure 9 shows the per-layer speedup numbers for the plain-GPU and cuDNN-GPU versions. For the plain-GPU version, the layer speedups are impressive. In the forward pass, all layers present speedups above 10×, and for the pooling and LRN layers the speedups are close to 110× and 40× respectively

(depending on the layer instance and pass in the network). The convolutional layers are the bottleneck. They present speedups between 1.8× and 6× depending on the layer position in the network and in the pass (forward or backward).

For the cuDNN-GPU, the trends are similar as in the MNIST case regarding the convolutional and pooling layers. The convolutional layers expose impressive improvements. They switch to performance levels close to 50× of speedup in some cases (conv2 layer). Some of the pooling layers expose drastic performance drops (pool3 forward pass switches from 42× to 11.75×). Others improve the performance (pool1 from 8.6× to 20.9×). As it was stated with the MNIST dataset, cuDNN corresponds to a case where the industry has deployed a highly optimized implementation of specific layer transformations that are well understood and no longer in a research stage. In this situation, the fine-grain parallelism makes a difference, though after the corresponding recoding efforts.

4.2.3 Overall Performance

Figure 9 shows the overall performance of the coarse-grain parallelization and the fine-grain parallelization in its two versions plain-GPU and GPU-cuDNN. The coarse-grain reaches a speedup close to a 6× with 8 threads, and 8.83× with 16 threads. The lack of scalability for the CPU version is related to the poor scalability of some of the convolutional layers. This is caused by poor data locality caused by consecutive layers with different data-thread patterns. In addition, again the serial initialization of the network structures is giving a suboptimal memory allocation in the NUMA nodes. All of this is affecting the final scalability of the coarse-grain version. The fine-grain GPU version shows a modest speed up close to 6×. Both the fine-grain GPU version and coarse-grain CPU version expose problems with the convolutional layers. In general, the fine-grain plain-GPU version corresponds to the Caffe native implementation. This implementation is representative of the performance the DNN community can obtain after the associated recoding efforts. The coarse-grain parallelization gives similar performance levels, but not requiring the reimplementation of all layers to target the GPU device. As with the MNIST case, when compared to the cuDNN case, the fine-grain approach makes a difference and this time even greater. cuDNN gives extraordinary speedups and it corresponds to a highly tuned implementation of the convolutional and pooling layers made by NVIDIA. It delivers a 27× speedup. Unfortunately, this performance levels are only available for these two layer types. In contrast, the coarse-grain approach is immediately available no matter the nature of the deep neural network.

4.3 Coarse-grain vs Fine-grain: Pros and Cons

In this section we present the lessons learnt regarding the coarse-grain and fine-grain parallelizations of the DNN training process. We identify several limiting factors that affect the performance of the coarse-grain parallelization. These factors are layer dependent and are mainly related to specificities of the layers that determine their work distribution, memory footprint, data privatization and ordered operations. Also, we identify the advantages and disadvantages of the coarse-grain approach in front a fine-grain parallelization. The following paragraph describes these limiting factors, advantages and disadvantages.

Network agnostic: one important advantage of the coarse-grain approach is that it exploits a parallelism level that is independent of the nature of the neural network. The batch-level parallelism is intrinsic to the gradient descent algorithm. Thus, this approach is immediately available and does not require programming efforts to generate GPU layer implementations. In addition, the coarse-grain approach delivers similar performance levels as the fine-grain approach. **Convergence invariance**: the coarse-grain parallelization does not change any training parameters. Thus, the convergence rate is kept invariant between the serial and the parallel executions. For the DNN community this is an important advantage. Before the training, neural networks need a parameter tuning process to ensure appropriate convergence. The parallelization has to ensure that the effects of this tuning are kept for the parallel execution. **Sequential memory allocation**: the network memory allocation happens during the network initialization. This process is sequential, which causes the layer memory be allocated following a pattern generated by the initialization code. In terms of performance, this pattern is not compatible with those that arise during the training process. **Locality between layers:** the input/output relation across layers defines a lost of data locality for specific layers. During the forward and backward passes, each layer distributes the work according to the dimensions of the input blobs. Regarding the data locality, it is possible that input and output blobs do not match their dimensions. Consequently, the work distribution and data-thread association defined in one layer will not match that one of the next immediate layer in the stack. One particular case of this situation corresponds to the data layers in Caffe. These layers feed the network with input data organized in batches. Data layers execute sequentially. Therefore, one thread first accesses all data and then the data is distributed across the cores and the memory hierarchy when the first parallel processing layer in the network executes in parallel. **Work unbalance:** coarse grain parallelism is open to work unbalance. For the Caffe case, we detected that batch-level parallelism defines very heavy iterations for the parallelized loops. Therefore, one single loop iteration can cause a high unbalance between the executing threads. Loop coalescing has been applied to reduce the effect of this problem. **Work granularity:** neural networks do a dimensionality reduction over the processed data. This affects the size of the working sets in the network layers. At some level in the network, the layer input/output data start decreasing their sizes. When this happens, a thread level parallelization starts suffering from small work granularity with modest performance levels. **Data privatization and reduction operations:** specifically in the backward pass, the network coefficients are updated per each sample in the batch. At batch-level, this requires mutual exclusion mechanisms to guarantee a correct update. Data privatization has been needed for this purpose. In terms of performance, given the layer granularity, we did not detect these updates to be a performance limiting factor.

5. Related Work

This section describes some state-of-the-art of deep learning frameworks. In general, due to the hard computational requirements for the training of deep neural networks, there has been generalized strategy based on GPU systems[7, 16, 19, 24, 26]. This has produced good results when data models fit in a small number of GPUs in a single server. But this approach is limited in terms of scalability. A coarse-grain approach has the potential of scaling up to a greater number of cores due to the fact that the limitations regarding the fitting of the data model are less strict.

For large scale training, there have been other approaches. In [10] the authors present a software framework called *DistBelief* that enables model parallelism within a machine (via multi-threading) and across machines (via message passing), with the details of parallelism, synchronization and communication managed by the framework. In addition to supporting model parallelism, the *DistBelief* framework also supports data parallelism, where multiple replicas of a model are used to optimize a single objective function. To facilitate training of very large deep networks *DistBelief* supports distributed computation in neural networks and layered graphical models. The user defines the computation that takes place at each node in each layer of the model, and the messages that should be passed during the upward and downward phases of computation.The authors mention in the paper that in certain cases partitioning the model on

more than certain number of machines actually slows down training, as communication overhead across hosts starts to dominate in fully-connected layers. One of the main disadvantages of the this framework is that one has to efficiently map the compute blocks onto the host processors such that communication across hosts are minimized to maximally leverage the model parallelism. The coarse level parallelism proposed in this paper is **network-agnostic** and hence more universally applicable.

In [8] presents a deep learning framework with COTS HPC systems: a cluster of GPU servers with Infiniband interconnects and MPI. Their system was able to train 1 billion parameter networks on just 3 machines in a couple of days, and they showed that it can scale to networks with over 11 billion parameters using just 16 machines. While the *DistBelief* framework manages to train a neural network using 16000 CPU cores (in 1000 machines) in just a few days, yet this level of resource is likely well beyond those available to most deep learning researchers. In this paper the authors present an alternative approach for training large scale networks that leverages inexpensive computing power in the form of GPUs and introduces the use of high-speed communications infrastructure to tightly coordinate distributed gradient computations. The authors in this paper makes significant efforts to make the most efficient use of the HPC resources, which may be sometimes well beyond the capabilities of machine learning researchers.

In [15] presents a state-of-the-art speech recognition system developed using deep learning. Their approach uses a well-optimized RNN (recurrent neural network) training system that uses multiple GPUs and use of a novel model partition scheme to improve parallelization. The authors in this paper develop specific data layout techniques and optimization schemes to scale their training for recurrent networks. The cuDNN [5] libraries are not general enough and doesn't provide an API for efficiently mapping recurrent computations on the GPU. The coarse grained parallelism proposed in this paper is a simplistic approach that can inherently applied for any feed-forward or recurrent neural network. Hence the proposed parallelism is more universally applicable across diverse application domains.

The Project Adam [6] implements a distributed system for large scale training of deep learning architectures. The system follows a similar approach as the one described in this paper: coarse-grain methodologies are applied to assign work to the executing threads. This approach is embedded in a larger system with asynchrony is a key aspect to achieve scalability. An asynchronous version of the stochastic gradient descent algorithm is implemented. This work succeeds in both achieving good performance results for the training process through a coarse-grain parallelization approach, and it avoids the recoding efforts related to acceleration with GPU programming.

6. Conclusions

In this paper we analyzed the implementation of a coarse-grain parallelization of the DNN training process. The approach has two significant properties. On one side, the coarse-grain parallelization is network-agnostic. It does not rely on any specialized and highly tuned library for the specificities of the network layers. Therefore, it does not require recoding the implementation to adapt to such libraries. Second, it is convergence-invariant as it does not change any of the training parameters, thus ensuring consistency with the sequential convergence rate. The implementation and analysis has been done within Caffe, a general and research oriented DNN framework. The coarse-grain parallelization follows a directive-based style, using OpenMP directives. We have identified its main limiting performance factors as well as the main advantages and disadvantages in front of a fine-grain parallelization. In particular, the implementation of fine-grain strategies requires the recoding of all the network layers in order to port them to the GPU device. For two state-of-the-art datasets (MNIST and CIFAR-10), the observed performance levels are similar for both strategies. So, the coarse-grain approach is not dependent on any recoding efforts given its network-agnostic key feature. This property is critical when the network implementation is in research stages, with a constant tuning of its parameters. Once the network is understood in terms of its convergence and accuracy, specialized and highly optimized libraries can be built. This is the case for cuDNN and other. These libraries include specific layer transformations that clearly make the difference in terms of performance that justify all the programming efforts associated to its usage.

Acknowledgments

This work was supported by the Ministry of Science and Innovation of Spain (CICYT) [TIN-2012-34557]. Special thanks to Myron Flickner and Pallab Datta who supported this work while Marc Gonzalez Tallada worked as Visiting Scientist in the Synapse Project within the Brain-Inspired Software department at the IBM Almaden Research Center.

References

[1] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[2] A. Basumallik, S.-J. Min, and R. Eigenmann. Programming distributed memory sytems using openmp. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1--8. IEEE, 2007.

[3] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135--151, 2002.

[4] L. Bottou. The tradeoffs of large scale learning. *Advances in Neural Information Processing Systems*, 2008.

[5] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[6] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[7] D. C. Ciresan, U. Meier, and J. Schmidhuber. Multicolumn deep neural networks for image classification. *Computer Vision and Pattern Recognition. CVPR12.*, 2012.

[8] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1337--1345, 2013.

[9] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011. URL `https://publidiap.idiap.ch/downloads/papers/2011/Collobert_NIPSWORKSHOP_2011.pdf`.

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, et al. Large scale distributed deep networks. In *NIPS*, pages 1232--1240, 2012.

[11] L. Deng. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *Signal Processing Magazine, IEEE*, 29(6):141--142, Nov 2012. ISSN 1053-5888. .

[12] J. Dongarra. Preface: basic linear algebra subprograms technical (blast) forum standard. *International Journal of High Performance Computing Applications*, 16(2):115--115, 2002.

[13] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 2011.

[14] Google. Protocol buffers. `https://developers.google.com/protocol-buffers/`, 2015.

[15] A. Y. Hannun, C. Case, J. Casper, B. C. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014. URL `http://arxiv.org/abs/1412.5567`.

[16] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 2012.

[17] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[18] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL `http://www.cs.toronto.edu/~kriz/cifar.html`.

[19] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 2012.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems 25*, pages 1097--1105, 2012.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient Based Learning Applied to Document Recognition. *IEEE Press*, pages 306--351, 2001.

[22] Y. LeCun, C. Cortes, and C. Burges. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`, 2015.

[23] Y. Nesterov. A method of solving a convex programming problem with convergence rate o(1/k). *Soviet Mathematics Doklady*, 1983.

[24] R. Raina, A. Madhavan, and A. Ng. Large-scale deep unsupervised learning using graphics processors. *International Conference on Machine Learning*, 2009.

[25] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, pages 1--42, April 2015. .

[26] M. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *In Arxiv 1311.2901. http://arxiv.org/abs/1311.2901*, 2013.