GPU Performance Enhancement

A Guide to CNNs and GPUs

Over the years of development in CNN technologies, we see a rise of complexity and computational power consumption coming with every generation of networks. The previous sections gave an insight to performance optimisation in terms of software and algorithms. Another useful option to speed up CNNs is exploiting advantages of specific hardware like FPGA or custom designed processor units. This site evaluates advantages, requirements and benchmarks considering GPUs (graphics processing unit) instead of CPUs (central processing unit) for training CNNs.

1 A Guide to CNNs and GPUs
   1.1 Bottlenecks in CNN performance
   1.2 Benefits from GPU architecture
      • 1.2.1 Introduction to GPUs
      • 1.2.2 Parallel computing
   1.3 Designing parallelised Convolutional Neural Networks
      • 1.3.1 CUDA
   1.4 Benchmarks and Examples
      • 1.4.1 Benchmarked example Network
   1.5 Literature
   1.6 Weblinks

Bottlenecks in CNN performance

In each generation AI evolves and gives better error rates. Nets get more flexible and deliver more remarkable results. They are trained with datasets consisting of over 14 Mio images like the image-net library for example. Each image has to pass through several layers of filters, Relu, Pooling etc. while large sets of parameters have to be adapted and saved.

As a result, research on CNNs can be expensive and time consuming. The process of training networks with big data sets often takes weeks on smaller computers. Using supercomputers at research institutes often require long waiting times and can be expensive.

These problems led many researches to use GPUs to train CNNs, as they can be considered as high performance Parallel computers for affordable prices. (weblink 1)

Convolutional layer

Figure 1: Runtime breakdown of typical real-life CNN models: GooleNet, VGG, OverFeat and AlexNet. (source 1)
Researchers at laboratories in China and the USA analyzed 4 popular real-life CNN models (AlexNet, GoogLeNet, OverFeat, VGG) in order to find out their hotspot layers. The networks were broke down to their different layers and tested for their average runtime performing 10 training iterations. Each iteration consisting of one forward- and backward-propagation.

All tested networks involve basically the same layers: Convolutional layer, pooling layer, Relu layer, fully connected layer and Concat layer (in GoogLeNet). The results shown in Figure 1 make clear that the convolutional layer dominates the training time in all of the four real life CNNs (up to 94%).

Comprising large amounts of computation-intensive operations due to the increasing use of filters and layers, smaller strides and their combinations, the convolutional layer is clearly a hotspot considering computation time. Therefore making use of hardware features, which reduce processing costs of the conv-layer seem practical. (1)

Benefits from GPU architecture

Recent developments in GPU hardware revealed new opportunities to scientific research while demanding comparably low budgets. The introduction of the supercomputer Tianhe 1A in November 2010 can be seen as a milestone in the shift from CPU only to CPU-GPU computing. Consisting of thousands of Intel Westmeres and NVIDIA Fermi boards it assumed the No.1 position in the Top-500 list of supercomputers.

In the last years we have seen a shift from increasing clock frequencies of processors to increasing the amount of cores due to power limitations. Any future performance increase might as well come mostly from an increase of core count. Although CPUs are still more flexible than todays GPUs, there is a huge benefit in massive parallel processing for specific tasks like applying filters on images. These advantages can be useful when applied to CNN training. (2)

Introduction to GPUs

History

GPUs evolved from fixed function Graphics Pipelines to massive parallel numeric computing processors over the last 30 years. This progress was heavily influenced by the fast growing game industry, which demands massive numbers of floating-point calculations per video frame.

Starting with processing units specialized on calculating polygons, colors and shades there has been a change to multi purpose machines in the last years. With the introduction of NVIDIA’s CUDA programming environment in 2006 GPU computation power was opened to developers and no longer limited to computer graphics. (3)

Many-core vs. Multicore

Today’s microprocessors are mainly following two different lines: The multicore and the many-core architecture. The multicore architecture is focusing on enhancing sequential tasks with its multiple processor cores. Starting with 2 cores, the number of cores is approximately doubling each design generation. The cores themselves come with a full instruction set.

Many-core architectures are focusing on the execution throughput of parallel applications. They provide large numbers of cores each of which is a heavily multithreaded, in-order, single-instruction issue processor with shared caches.

The many-core architecture has seen massive improvements considering performance over CPUs over the last years. Figure 2 shows how GPUs are beating CPUs in terms of Floating Point operations per second (FLOPS). Regarding this fact, developers are starting to put computationally intensive parts of their software to the GPU. These high performance tasks often also require high parallelism.
Structural differences

GPUs and CPUs are designed for different purposes. They are both able to provide significant advantages when used at the right place. CPUs are optimized for sequential code performance. Their sophisticated control logics allow instructions from a single thread execution to run in parallel. It also has the power to change the sequential order while globally maintaining the appearance of sequential execution. Big cache memories reduce the instruction and data access latencies of large complex applications.

On one hand GPUs' memory interfaces show higher latency, which is readily hidden by massive parallel execution. But on the other hand, they provide a large bandwidth, which is usually many times higher than on CPUs (Up to 150 GB/s). As stated earlier, GPUs are also capable of performing huge numbers of floating-point calculations within short times.

Figure 3 shows both microprocessor design strategies. The GPU’s highly parallelized architecture with its multiple processing- and sub-processing units contrary to the CPU’s big cache memory and comparably small amount of cores. (3)

Parallel computing
Parallel and distributed algorithms are a common method to overcome high training times in CNNs. Training a neural net on a multicore processor with \( n \) cores can be up to \( n \)-times faster than on a single core chip. With the recent developments in GPU processors and new developer frameworks like CUDA, parallelism can be a great enhancement in CNN research.

There are different common methods to implement parallel structures:

- **Local training:**
  
The model and data is stored on a single machine. The net is trained with the CPU, which hands computation intensive tasks over to the GPU.

- **Distributed training:**
  
The data or the model is stored on different devices.
  
  - **Data parallelism:** The data is distributed among multiple machines. This method is useful if the data is too big for one machine or to achieve faster training.
  
  - **Model parallelism:** If a model requires too much memory to be run on a single machine, it can be split to multiples. This method is more likely to be used with memory issues than too speed up training, as it’s often harder to implement.

**Drawbacks**

Before switching computation from CPU to GPU there are some drawbacks which developers should be aware of. Not all GPU models especially older ones are well suited for executing the refactored parallelized code.

Firstly not all GPUs support the full Institute of Electrical and Electronics Engineers (IEEE) floating-point standard. This standard assures predictable results across processors from different vendors. Though almost all of the latest graphic processors support the single precision floating-point arithmetic by now, double precision is still not available on many devices.

Another aspect is that GPUs which not support the CUDA programming environment are harder to work with, as all tasks have to be mapped to either the OpenGL or Direct3D interface in order to be executed by the GPU. This means a lot of more work as those interfaces are designed for computing graphics, not CNNs.

**Designing parallelised Convolutional Neural Networks**

As mentioned before, GPU computing can increase specific computational tasks but the code has to be adapted in order to generate faster results than the usual CPU code would. Major benefits can be achieved on massive parallel tasks, which have to be distributed intelligently. Many researchers already ported their Networks to GPU successfully. This section will give insights on several problems of CNN programming using GPUs.

Choosing the right model of parallelization can be game changing. Using model parallelism or hybrid data-model over data-parallelism showed performance benefits, but can be harder to implement (Examples and evaluation: 5). The individual tasks and threads should be fitted to the hardware settings, in order to make use of the whole power of each individual core. This means that the code has to be split up into small sub-tasks while keeping in mind how many threads can be executed in parallel. For example if a GPU can handle 32 sub-tasks per core, it would be inefficient to optimize the Code for 33 sub-tasks, as the processor is forced to split the whole task for the execution.

Memory accesses are another important factor in GPU optimization of CNNs. It is well recommended to reuse loaded data as often as possible, since memory accesses are quite costly, especially for the small sized shared memory.

In general it is very important to be aware of the hardware specifications, its limits and potentials to get a performing CNN as a result.
Back propagation

The standard implementation of a Neural Net forwards its activation to the next layer during forward propagation and pulls the error back during back propagation. This method is disadvantageous for use in CNNs. Due to border effects not all neurons have equal numbers of outgoing connections in a convolutional layer, which makes it hard to implement. Furthermore every layer has to know the type of its subsequent layer.

Instead it’s easier to push the error back to the previous layer. The advantages are: constant numbers of input connections (even from convolutional layers), no knowledge about neighboring layers is needed and the resulting normed backpropagation process is easier to optimize. (7)

CUDA

CUDA (Compute Unified Device Architecture) is a general parallel computing platform and programming model launched in 2006. It opens the parallel compute engine in NVIDIA GPUs to software developers. It is a C-like software environment with extensions for parallelization and special memory accesses. Other languages and application programming interfaces like FORTRAN, DirectCompute, OpenACC are supported as well. The launch of Multicore CPUs and many-core GPUs came with the demand of parallelism in code. Modern software has to scale with the hardware it’s running on in order to leverage the increasing numbers of cores. It needs to adapt in a way like 3D graphics applications scale their parallelism to different hardware.

CUDA is designed to address this challenge while being easy to learn for developers with programming experience in languages like C. It adds three major abstractions to the usual C language: hierarchical thread groups, shared memories and barrier synchronization. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism.

With this toolbox, the programmer is able to divide problems into independently solvable sub-problems. These can be run in parallel by blocks of threads. Further each sub-problem is broken up into finer pieces, which can be solved cooperatively in parallel by all threads of a block. Each block of threads can be scheduled to any of the available multiprocessors in any order, in parallel or sequentially. So the processor can be used efficiently at any time of the execution. Furthermore the code is automatically scaled up or down depending on the number of multicores available on the chip.

This scalability allows much flexibility concerning the use of hardware. The same software can be run on low-priced and high-end CUDA enabled GPUs. Contrary to this flexibility is the obligatory binding to NVIDIA hardware. (3, 6, web link 2)

GPU supporting frameworks /libraries

There are a lot of frameworks and libraries, which support GPU enhancement. The following list depicts some of the most popular ones:

- Tensor Flow
- Cuda-convnet
- Cuda-convnet2
- Theano
- Torch
- Decaf
- Caffe
- cuDNN
- fbfft

A detailed comparison of these tools can be read in (1)

Convolution techniques
Many deep learning frameworks and libraries already support CNN on GPUs. Especially convolutional layers gain a lot of optimization from these implementations, as they represent a central part of CNNs. Several ways to realize the convolution part have been explored by researchers. In general there are three mainstream methods used: direct convolution, unrolling-based convolution and FFT (Fast Fourier Transformation) based convolution.

**Direct Convolution**

During direct convolution, a small window slides within an input feature map and a dot production between the filter bank and local patch of the input feature map is computed. The result of dot production is then passed into a non-linear activation function, e.g., Sigmoid and Tanh. Outcome results from this activation function are organized into a new feature map as output. Repeating the above process for each filter bank, produce a set of two-dimensional feature maps as the output of the convolutional layer.

Implementations: cuda-convnet2, and Theano-legacy.

**Unrolling Based Convolution**

A more efficient method on GPUs is the unrolling-based convolution. The key idea behind unrolling convolution is to reshape the input and the filter bank to double large matrices. The local regions of input image are unrolled into columns and the filter banks are unrolled into rows using im2col. The final convolution can be converted into a clean and efficient matrix-matrix production by using highly-optimized libraries such as cuBLAS on GPUs. Finally, the results should be remapped back to the proper dimension using col2im.

Implementations: Caffe, Torch-cunn, Theano-CorrMM, and cuDNN.

**FFT Based Convolution**

This strategy is based on the convolution theorem that a discrete convolution in the spatial domain can be converted into the product of the Fourier domain. The performance of FFT-based convolution can be significantly improved thanks to its lower computation complexity. In general, FFT-based convolution can be implemented by three main steps. First, inputs and filter banks are transformed from the spatial domain to the Fourier domain with Fast Fourier Transformation (FFT). Second, those transformed matrices are multiplied in the Fourier domain. Finally, the product results are inversed from the Fourier domain to the spatial domain.

Implementations: fbfft, Theano-fft. \(^{(1)}\)

**Benchmarks and Examples**

**Benchmarked example Network**

Researchers at the Facebook AI group tested a standard image classification network in 2014 considering the use of 1 to 4 GPUs using model-, data- or hybrid-parallelism models. Their experimental results showed how the nets training time could be reduced to a half using a hybrid parallelism model with 4 instead of 1 GPUs. In figure 5 a comparison over 100 complete training epochs for different configurations can be seen.

The graph on figure 4 shows how the error rate of the test set decreases in respect to the time. It depicts how different parallelization methods compete with standard non-parallelized models. For further detailed information on the experiment read the referenced documentation. \(^{(5)}\)
Further benchmarks and experiments can be found in: 4, 6, and 7.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time to complete 100 epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GPU</td>
<td>10.5 days</td>
</tr>
<tr>
<td>2 GPUs Model parallelism</td>
<td>6.6 days</td>
</tr>
<tr>
<td>2 GPUs Data parallelism</td>
<td>7 days</td>
</tr>
<tr>
<td>4 GPUs Data parallelism</td>
<td>7.2 days</td>
</tr>
<tr>
<td>4 GPUs model + data parallelism</td>
<td>4.8 days</td>
</tr>
</tbody>
</table>

Literature


NVIDIA, the NVIDIA logo, CUDA, GeForce, Fermi and Tesla are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries.

Weblinks