Accelerating real-time embedded scene labeling with convolutional networks

Accelerating Real-Time Embedded Scene Labeling with Convolutional Networks

Lukas Cavigelli[*]
cavigelli@iis.ee.ethz.ch

Michele Magno[*†]
magno@iis.ee.ethz.ch

Luca Benini[*†]
benini@iis.ee.ethz.ch

[*]Integrated Systems Laboratory, ETH Zurich
Gloriastr. 35, 8092 Zurich, Switzerland

[†]DEI, University of Bologna
Viale Risorgimento 2, 40136 Bologna, Italy

ABSTRACT

Today there is a clear trend towards deploying advanced computer vision (CV) systems in a growing number of application scenarios with strong real-time and power constraints. Brain-inspired algorithms capable of achieving record-breaking results combined with embedded vision systems are the best candidate for the future of CV and video systems due to their flexibility and high accuracy in the area of image understanding. In this paper, we present an optimized convolutional network implementation suitable for real-time scene labeling on embedded platforms. We show that our algorithm can achieve up to 96 GOp/s, running on the Nvidia Tegra K1 embedded SoC. We present experimental results, compare them to the state-of-the-art, and demonstrate that for scene labeling our approach achieves a 1.5x improvement in throughput when compared to a modern desktop CPU at a power budget of only 11 W.

Categories and Subject Descriptors

I.4.8 [**Image Processing and Computer Vision**]: Scene Analysis—*object recognition*; I.2.10 [**Image Processing and Computer Vision**]: Vision and Scene Understanding

General Terms

Accelerator, Scene Labeling, Convolutional Networks

1. INTRODUCTION

Today vision technology is successfully used in a wide variety of real-world applications, such as surveillance, robotics, industrial, medical, and entertainment systems [16]. A growing number of researchers are proposing to address the recognition of actions and objects with brain-inspired algorithms featuring multi-stage feature detectors and classifiers which can be customized using machine learning [6, 7, 11]. These techniques, collectively known as *deep learning*, have recently achieved record-breaking results on highly challenging datasets using automatic (supervised or partially unsupervised) learning. Convolutional Networks (ConvNets) are a well-known example of this remarkably versatile, yet conceptually simple paradigm, which can be applied to a wide spectrum of perceptual tasks and CV applications [20]. Lately, Deep learning approaches have been proposed in many variations and several research programs have been launched by major industrial players in the U.S. (e.g., Facebook, Google, IBM), aiming at deploying brain-inspired visual processing within a production environment [19, 20]. All these companies are mainly interested in running these algorithms in large data centers on clusters of very powerful computers.

The importance of digital signal processing (DSP) in imaging continues to grow. The level of sensor computation is increasing to thousands of operations-per-pixel, requiring high-performance and low-power DSP solutions, possibly co-integrated with the imaging circuitry to reduce system cost. The emergence of very powerful, low-cost, and energy-efficient integrated parallel processing engines (multi-core CPUs, GPUs, platform FPGAs) is enabling this new generation of distributed CV systems. The term *embedded vision* is used to refer to such embedded systems that extract meaning from visual inputs at the sensor. It is clear that embedded vision technology can bring huge value to a vast range of applications, reducing data transmission by forwarding only the desired information [13, 16].

There are many research and innovation opportunities lying at the intersection of future situational awareness systems and the evolution of advanced embedded video processing, tightly coupled to the sensors themselves. In contrast to conventional cameras (IP cameras, CCTVs, etc.) that send data to a remote host to be stored or processed, embedded video systems process the image data flow directly on board. Recent studies show that on board processing and local intelligence can significantly reduce the amount of raw data to be transmitted and the required human intervention [1].

Deep neural networks and scene labeling algorithms in general require a lot of computations, making it challenging to bring this computational load within an embedded vision power envelope – in fact, most state-of-the-art algorithms require workstations with powerful GPUs to achieve reasonable performance.

In this paper we present a state-of-the-art scene labeling ConvNet together with a highly optimized implementation towards the ultimate goal of getting most performance out of available accelerators on an embedded platform consuming few Watts. The main contribution of this work focuses on the algorithm, its optimized implementation and the performance evaluation with a state-of-the-art comparison. To evaluate the performance on a real platform, we chose the Nvidia Tegra K1, a low-power embedded SoC targeted at mobile computers and consuming only 11 W. It comprises 4 ARM Cortex A15 cores and a GPU consisting of one streaming multiprocessor (SMX), providing a throughput of up to 320 GFLOPS. In comparison, a modern desktop GPU such as the GTX780 has 12 SMX running at a similar frequency. The implementation presented here is the best-performing demonstration of a real-time, state-of-the-art accurate ConvNet running scene labeling on an embedded platform.

Organization of the Paper: Section 2 presents the related work. Details on the ConvNet and its optimized implementation are given in sections 3 and 4, respectively. Section 5 presents the experimental results for evaluating the proposed solution. Finally, Section 6 concludes the paper.

2. RELATED WORK

Many of the networks which have recently been winning visual object detection and recognition challenges require GPU acceleration in order to be trained within reasonable time [11]. When training a ConvNet the forward pass, i.e., the application of the ConvNet to the input data, takes up a significant part of computation time and is thus already well-optimized. We evaluate the performance of some of them for comparison to our work. However, they all require a large batch size, i.e., process many images in parallel, to achieve their best performance [2, 9, 11] – an option we do not have in our real-time setup. The most actively developed libraries are:

cuda-convnet [11]: It has long been the fastest implementation around and is focused on using multiple GPUs in parallel. It has a very low memory footprint and is thus mostly used to train very large networks with huge amounts of data. However, it requires a batch size of at least 16 and is being outperformed by more recent implementations. This makes it unsuitable for our application. It also has restrictions on the number of input and output channels.

Caffe/SpatialConvolutionMM: The Caffe framework [9] was the first to implement the convolution as a matrix multiplication. It has almost no restrictions and is quite fast at the expense of requiring much more memory. This can be critical when running batches, but it is not an issue for single images. Their approach has been ported to more frameworks including Torch7 [3], where it is available as `SpatialConvolutionMM`. Unfortunately, it does not run on the Tegra K1, but we still evaluate its performance on the GTX780.

Nvidia cuDNN: This library has been released in 2014 [2]. Their concept is the same as in Caffe, also leveraging the availability of extremely optimized GEMM (general matrix multiplication) routine. To put away with the memory problem, the large matrix $\mathbf{X}$ is never really constructed, but some more complicated indexing is being used. Bindings to this library have been developed for many frameworks, including Torch. We also evaluate this library's performance on the GTX780, but it is not available for the Tegra K1 as well.

FFT-based Approach: When requiring a large number of convolutions, there is always the option to use a FFT-based approach. Using the FFT for ConvNets has recently been investigated in [14]. It can strongly outperform spatial-domain convolution, but has several drawbacks – especially for real-time applications. We discuss this in Section 4.

After all, we can say that no viable GPU-accelerated implementation for embedded platforms exist and existing implementations' performance in a real-time setup is not even demonstrated on desktop GPUs.

3. CONVOLUTIONAL NETWORK CLASSIFICATION

In this section, we first present the basic building blocks used in state-of-the-art ConvNets and then introduce our own ConvNet used for the evaluation of the throughput of our implementations. We then also show that this network can be extended using a simple modification to push the accuracy of ConvNet-based scene labeling beyond state-of-the-art, thus proving that it is a representative choice for performance evaluation.

ConvNets are built from several stages, where each stage is a sequence of a convolution layer, a neural activation layer and a pooling layer (cf. Fig. 1). Sometimes the pooling stage is omitted. ConvNets are generally used as feature extractors, transforming hard-to-understand data into a more meaningful higher-dimensional representation, while preserving the locality of information (i.e., the object class of some pixel in the top-left corner of the image is not influenced by pixels in the bottom-right corner) [11, 19]. We call the various images in such a representation *channels*. We start with three channels (e.g., RGB) and quickly grow to 16, 64 and 256 channels. This more expressive representation is then used for classification, which is accomplished with a two-layer fully-connected neural network.

A stage of a ConvNet can be captured mathematically as

$$\mathbf{y}^{(\ell)} = \mathrm{conv}(\mathbf{x}^{(\ell)}, \mathbf{w}^{(\ell)}) + \mathbf{b}^{(\ell)}, \quad \mathbf{x}^{(\ell+1)} = \mathrm{pool}(\mathrm{act}(\mathbf{y}^{(\ell)})),$$

$$y_o(j,i) = b_o + \sum_{c\in\mathcal{C}_{in}} \sum_{(\Delta j,\Delta i)\in\mathcal{S}_k} w_{o,c}(\Delta j,\Delta i)\, x_c(j-\Delta j, i-\Delta i),$$

where ℓ indexes the stages, o indexes the output channels $\mathcal{C}_{out}$ and c indexes the input channels $\mathcal{C}_{in}$. The pixel is identified by the tuple (j,i) and $\mathcal{S}_k$ denotes the support of the filters, typically square with size between 3×3 and 7×7. We have $w : \mathcal{C}_{out}\times\mathcal{C}_{in}\times\mathcal{S}_k\to\mathbb{R}$.

The most promising activation function in use today is the rectified linear unit (ReLU) [11, 19], $\max(0,\cdot)$, which has succeeded other activation functions which are closer mimicking the behavior of actual neurons. This function is applied point-wise to every pixel of every channel. These activation functions are the essential part of neural networks, since their non-linearity is exactly what makes neural networks more powerful than faster and better understood linear concepts.

The third layer type, pooling layers, takes image patches of $n_p \times n_p$ pixels and computes, e.g., the maximum across them. Most commonly we have $n_p = 2$ and a stride of 2×2, thus reducing the number of pixels by a factor of 4. Pooling is performed individually for all channels of an image and is essential in order to keep the overall dimensionality from increasing even more. It can also be interpreted as a means to introduce stability and some translation invariance into ConvNets [20].

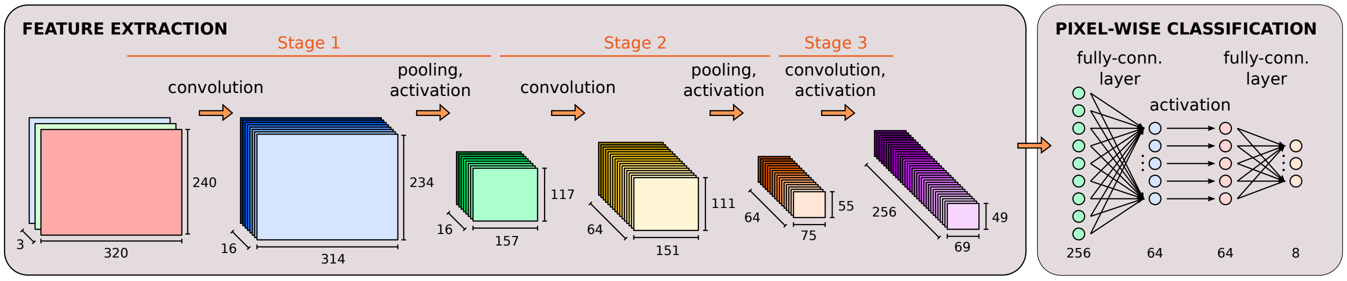

Figure 1: The ConvNet for scene labeling, as used for our performance evaluation. The convolution steps are performed with 7×7 filters, max-pooling is done over 2×2 areas, for the activation we used ReLU.

The filters and biases are learned from labeled data (input images and expected outputs) by numerical optimization.

3.1 State-of-the-Art Scene Labeling

Using the aforementioned building blocks, we have developed a ConvNet for scene labeling and trained it on the Stanford backgrounds dataset [8] (715 images, 320×240 pixels, various outdoor scenes). Each image comes with a corresponding label for every pixel into 8 classes: sky, tree, road, grass, water, building, mountain and foreground object. We split the dataset into 565 training images and 150 test images.

The structure of our feature extractor is shown in Figure 1. After some basic preprocessing (13×13 contrastive normalization in YUV space, mean subtraction, variance standardization; not shown in figure), we start the feature extraction with a color image (3 channels) and approximately quadruple the number of channels in every stage while decreasing the number of pixels in spatial domain by a factor of 4 through 2×2 max-pooling. The convolutions were performed with filter kernels of size 7×7 and we used ReLU for activation.

We have used a multi-scale (MS) approach as in [5], applying this feature extraction to the image at original scale and to scaled versions, downsampled by a factor of 2 and 4, and sharing the learned parameters across the scales. We concatenate the output channels of the various scales' feature extraction results and perform the pixel-wise classification with $3 \cdot 256$ input nodes. For this we use one fully-connected layer with 64 output nodes, followed by one ReLU activation layer and another last fully-connected layer down to 8 output nodes – one for each class. We select the strongest response as our result.

With this setup we achieve a per-pixel accuracy of 80.6% on the test data and compare this to previous results in Table 1. ConvNets achieve close to state-of-the-art accuracy and require orders of magnitude less computation time than other approaches. Our network shows higher precision than all other known ConvNets on the Stanford backgrounds dataset without post-processing.

3.2 Throughput Measurement

Modifications to the Network: Many frameworks are not capable of handling multi-scale ConvNets, making it very hard to compare throughput measurement results. To address this issue, we use a simplified network without the multi-scale concept. Since the structure of the feature extraction is preserved doing so, our network remains representative in terms of computations performed in state-of-the-art scene labeling ConvNets. The pixel-wise classification is then performed with 256 inputs, but otherwise remains the same. The entire reference model is shown in Figure 1.

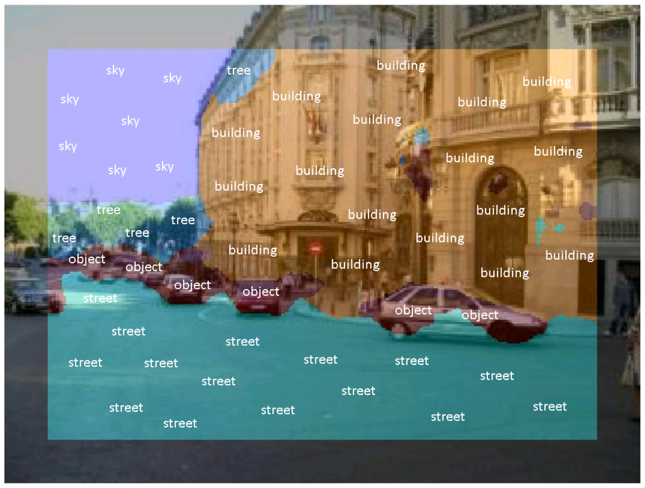

Figure 2: Sample output with reference ConvNet.

The network's parameters (filters, biases) were trained using stochastic gradient descent[1] (SGD). Training took about 12 hours on the GTX780 for the reference ConvNet using the Torch framework for training.

Complexity Metric: To compare the performance of various implementations, there is the need for a common measure. To perform a convolution, there is a minimum number of multiply-add operations that need to be performed. For a convolutional layer with n_{in} input layers, n_{out} output layers, an image size of $h_i \times w_i$ and a kernel size of $h_k \times w_k$, this number is $n_{out} n_{in} h_k w_k (h_i - h_k + 1)(w_i - w_k + 1)$. We count the multiply-add as two operations (Op).

This definition is still ambiguous. Often there is some constant overhead for initialization, and the performance is better for very large images. We thus distinguish between the performance obtained with a real network and measurements obtained with a synthetic benchmark, optimized to squeeze out the largest possible number. The later we name *Peak Throughput*, the former we call *Actual Throughput* or

[1]learning rate $2 \cdot 10^{-5}$ with decay of 10^{-4}, momentum $4 \cdot 10^{-6}$, weight decay 10^{-1}, mini-batch size is one image. We used a pixel-wise one-vs-one margin maximization loss $\ell(\mathbf{x}, t) = \sum_{d \neq t} \max(0, 1 - x_t + x_d)$, where t indexes the target class and $\mathbf{x}$ is a vector with the outputs for each class.

Table 1: Accuracy of Different Scene Labeling Methods on the Stanford Backgrounds Dataset.

Method	Pixel Acc.	Class Acc.	CT [s]
Stacked Hierar. Labeling [15]	76.9%	66.2%	12
Superparsing [22]	77.5%	?	10
Selecting Regions [12]	79.4%	?	600
CHM w/ intra-cls. conn. [18]	82.85%	74.32%	65
Ren, et al. [17]	82.9%	74.5%	?
Farabet's SS ConvNet [5]	66.0%	56.5%	0.35
Farabet's MS ConvNet [5]	78.8%	72.4%	0.60
Farabet's MS ConvNet with superpixels [5]	80.4%	74.6%	0.70
Farabet's MS ConvNet with CRF on gPb [5]	81.4%	76.0%	60.5
Our SS (reference) ConvNet	74.8%	65.0%	0.22
Our multi-scale ConvNet	80.6%	71.1%	0.46

just *Throughput*. A further first impression of a device is given by the number of multiplications and additions it can perform per second as claimed by the vendor of the device. We call this the *Theoretical Throughput*.

For our reference ConvNet with an input of 320×240, the number of operations is 7.57 GOp for the feature extraction and 0.1 GOp for the pixel-wise classification. For a full-HD (1920×1080) frame, 259.5 GOp and 4.1 GOp are required. For large images, the required number of Op/s can be estimated at 127 kOp/pixel, e.g., for 4k UHD about 1050 GOp.

The network for state-of-the-art accuracy requires some more operations. For the feature extraction for a 320×240 image we need 10.30 GOp for scale 1, 4.24 GOp for scale 2 and 1.56 GOp for scale 4. This amounts to 16.10 GOp.

4. IMPLEMENTATION

In this section, we start by defining a measure for performance metering. We continue by describing some previous implementations and introduce a reference implementation obtained using the Torch framework. We then present three implementations we have developed, highlighting our optimizations and disclosing the resulting performance.

We have discussed some related work in Section 2. Since there are not many available performance figures, and those that are available use some artificial network size, we have evaluated Nvidia's cuDNN and the `SpatialConvolutionMM` on our own network (cf. Table 2). They both do not run on the K1 – cuDNN is not available for ARM hosts and `SpatialConvolutionMM` crashes due to a lack of memory even for very small networks. The computation time split with the latter is shown in Figure 3, clearly identifying the convolution layers as the most critical part.

We have also mentioned the FFT-based approach, for which we give an upper bound on the throughput for our setup. Using the FFT for ConvNets has recently been investigated in [14]. For 7×7 filters, 32×32 images, 96 input layers and 256 output layers, they report a peak throughput of up to 6128 GOp/s with a GeForce GTX Titan GPU. While these results are outstanding, they are based on a large batch size of 128, denying real-time application due to the large delay.

To rule out that this approach can easily be adapted to

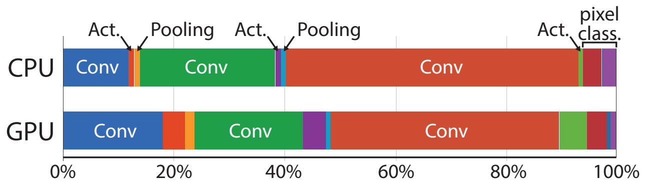

Figure 3: Computation time breakdown for torch-based CPU and GPU forward pass through the single-scale network.

smaller batch sizes, we bound its throughput when considering processing individual images. First we note, that this approach requires significant amounts of memory. For the third stage of our ConvNet, $256 \cdot 64 \cdot 75 \cdot 55 \cdot 4 = 270.3$ MB is needed to store the filters instead of 3.2 MB. While this is acceptable for networks of limited size, the bigger issue is that these filters have to be loaded for every frame for the pixel-wise multiplication (when not creating batches). The GTX780 comes with a bandwidth of 288 GB/s. The loading the filter kernels at the theoretical limit of the device thus takes 0.94 ms. We have measured the time to perform the necessary transforms using cuFFT for this stage of our network, obtaining a minimum of 1.235 ms. This means that the 5.43 GOp/frame could at best be processed within 2.175 ms, amounting to 2497 GOp/s. For the Tegra K1 the situation is disproportionately worse, as it has only 12 GB/s memory bandwidth.

4.1 Proposed Optimizations

In this section, we present our own, fully-optimized implementations. We show a straight forward, direct approach which implicitly minimizes the memory throughput, but has to process the data less regularly. In a second implementation we map the convolutions to a matrix-matrix multiplication problem, making use of the highly optimized cuBLAS GEMM routine. We then further optimize this by using CUDA streams to get as much out of the device as possible. The resulting performance of the various approaches can be found in Table 2.

4.1.1 Direct Approach

There are many options to parallelize the operations of a convolution. We choose that each output channel has its own block, and each block consists of a 2D array of threads, covering a patch of the output image. For this patch of the output image, the required pixels from the input image are loaded into shared memory. Each thread accumulates the results for its output pixel, then the patch is moved to the next position until the whole image has been covered. Thereafter, the filter for the next input channel is loaded into shared memory and the whole procedure repeats until the entire filter response has been calculated.

In its inner-most loop, this implementation needs to fetch one image pixel and one filter pixel from shared memory, and then performs a multiply-add operation. This approach is thus limited by the shared memory throughput and similar to Krizhevsky's cuda-convnet implementation [11], but without the restriction to square images. However, this bottleneck can be overcome by calculating multiple output pixels per thread. The filter coefficient can then be reused for all output pixels and thus there remains only one fetch from shared memory and a multiply-add per pixel in the inner-most loop. With this optimization, we were able to achieve a peak throughput of 893 GOp/s on a GTX780 and 60.4 GOp/s on a Tegra K1 (cf. Table 2).

4.1.2 GEMM-based Approach

We do not see any option to incrementally improve further on the direct implementation. Based on the time split in Fig. 3 we can see that the convolution is the most critical part. To further improve, we investigate the use of the matrix-multiplication in the highly optimized cuBLAS library. We reformulated the convolution in the form of a matrix-matrix multiplication by constructing something similar to the Toeplitz matrix of the input image:

$$\mathbf{Y} = \mathbf{K}\mathbf{X}, \quad \mathbf{Y} \in \mathbb{R}^{|\mathcal{C}_O| \times h_o \cdot w_o},$$
$$\mathbf{K} \in \mathbb{R}^{|\mathcal{C}_O| \times |\mathcal{C}_I| \cdot h_k \cdot w_k}, \quad \mathbf{X} \in \mathbb{R}^{|\mathcal{C}_I| \cdot h_k \cdot w_k \times h_o \cdot w_o}.$$

The image matrix $\mathbf{X}$ is given by $X((kh_k + j)w_k + i, y_o w_o + x_o) = x(k, j + y_o, i + x_o)$ with $k = 1, \ldots, |\mathcal{C}_I|$, $j = 1, \ldots, h_k$, $i = 1, \ldots, w_k$ and $y_o = 1, \ldots, h_o$, $x_o = 1, \ldots, w_o$. The filter matrix $\mathbf{K}$ is given by $K(o, (ch_k + j)w_k + i) = w(o, c, j, i)$ for $o = 1, \ldots, |\mathcal{C}_O|$, $c = 1, \ldots, |\mathcal{C}_I|$, $j = 1, \ldots, h_k$ and $i = 1, \ldots, w_k$. The matrix containing the convolution results is stored as $Y(o, y_o w_o + x_o) = y(o, y_o, x_o)$.

With this approach we achieve a peak throughput of 2684 GOp/s on the GTX780 and 96 GOp/s on the K1 (cf. Table 2). The obvious drawback of this solution is the increase in the required amount of memory due to the need to create the matrix $\mathbf{X}$. For the last convolution layer of our feature extractor, this amounts to a total of $64 \cdot 7 \cdot 7 \cdot (75 - 7 + 1) \cdot (55 - 7 + 1) \cdot 4 = 42.4$ MB instead of less than 1 MB.

4.1.3 GEMM-based Approach with Streams

Using two alternating CUDA streams to exploit this, we were able to further boost the performance to 3059 GOp/s on the GTX780 or 76% of the vendor-claimed theoretical throughput. The GOp measure neglects the remaining operations necessary in a ConvNet such as activation and pooling, such that the actual load is even higher. On the K1 the use of multiple CUDA streams does not have any effect, since the memory bandwidth is limiting in all processing steps. With this approach, applying our reference ConvNet to 320×240 pixel frames on the GTX780 takes 4.25 ms (1781 GOp/s).

5. RESULTS & DISCUSSION

We have compiled the throughput measurement results of our own implementations in Table 2 along with some figures previously published for various platforms. The most recent publication [4], which includes a GTX780, shows results close to those we have obtained with the direct approach, but far from our best performing implementations. However, many new libraries have recently been published and also development without publications has been very active. This prompted us to benchmark these implementations through their bindings for the Torch framework. We show the results in the center part of the table. They significantly outperform previously published implementations. Still, our highly optimized implementation based on the use of GEMM and CUDA streams remains unrivaled on desktop GPUs, achieving more than 70% higher peak throughput and a performance improvement of 55% when processing data for our state-of-the-art scene labeling ConvNet. With a peak throughput of 3059 GOp/s, which only takes into account the operations required for the convolutions and neglects the operations actually performed during activation and pooling, we come very close to the device's theoretical capabilities.

Table 2: Performance Comparison

Implementation	Device	Throughput [GOp/s] act.	peak[a]	theor.
Farabet, et al. [6]	GTX480		294[b]	1350
Dundar, et al. [4]	GTX650m		54[b]	182
Jin, et al. [10]	GTX690		570[b]	5622
Dundar, et al. [4]	GTX780		620[b]	3977
Teradeep, Inc.[c]	Tegra K1		37[b]	326
torch	CPU[d]	50		118
torch-MM	GTX780	1145	1252	3977
torch-cuDNN	GTX780	946	1770	3977
FFT-based, bound[e]	GTX780		2497	
ours direct	GTX780	387	893	3977
ours GEMM	GTX780	1545	2684	3977
ours GEMM, strm.	GTX780	1781	3059	3977
ours direct	Tegra K1	37	60	326
ours GEMM	Tegra K1	76	96	326

[a] Peak performance with batch size 1.
[b] Measured for a favorable problem size (also batch size).
[c] Teradeep, Inc. claims a throughput per power of 3.4 GOp/s/W for the Tegra K1. Throughput based on a platform power of 11 W. [21]
[d] Intel Xeon E5-1620v2
[e] Semi-theoretical upper bound for Stage 3 without activation, as described in Section 4.

For a large scale deployment in smart cameras, an embedded solution is required in terms of size as well as power efficiency, yet providing decent performance. To the same end, Teradeep has evaluated the performance of the Tegra K1 for ConvNets, providing the only comparison point for the K1 [21]. Their performance is similar to our direct implementation's (cf. Tbl. 2). However, with our GEMM-based approach we have been able to provide a performance boost of about 100%, assuming they were testing with a similar ConvNet, or even close to 160% if they were measuring the peak performance.

Implementations achieving a higher throughput have proven to be also more power efficient, since there is no measurable impact on power consumption for the desktop GPU. For the Tegra K1 our direct implementation caused the platform to draw significantly less power at 7 W instead of the 11 W required for the GEMM-based approach. In terms of power efficiency, the latter still performs slightly better. Our results are shown and compared to other implementations in Fig. 4.

We have measured the system power and the differential power. The efficiency of the Tegra K1 and the GTX780 is very similar, which was to be expected given that the architecture of the Tegra K1's GPU accelerator does not differ from the GTX780 in its core, except that the K1 contains only a single streaming multiprocessor instead of 12.

Our implementation running on the Tegra K1 or on the

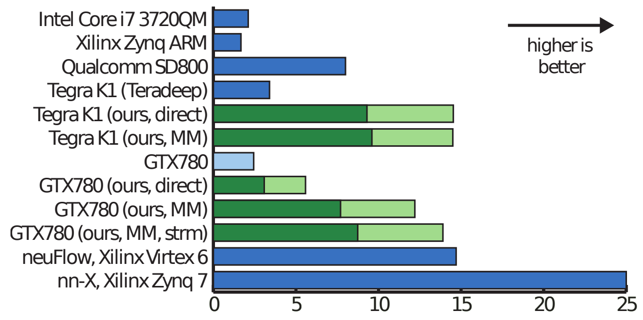

Figure 4: Comparison of performance per power [GOp/s/W] for several implementations and platforms. Reference values (blue) from [6, 7, 21]. Results for system (dark) and differential power (light).

GTX780 is efficient enough to compete with programmable logic implementations, coming close to the efficiency of the neuFlow accelerator for ConvNets on a Virtex 6 working with 16bit fixed-point numbers, but with much higher flexibility and lower implementation effort, as desirable for such a rapidly evolving subject area.

6. CONCLUSION

We have presented a highly optimized implementation of the main building blocks of ConvNets using GPU acceleration and demonstrated its previously unmatched throughput on a state-of-the-art ConvNet for scene labeling. It fulfills all the requirements for usage in real-time embedded CV systems and does not require batches of multiple images to achieve good performance. The resulting performance comes very close to the theoretical capabilities of current desktop GPUs, such that only insignificant further improvement can be expected without a change in the algorithmic approach.

Our implementation provides unmatched performance on Tegra K1-based platforms, making it suitable for embedded systems. We have been able to achieve 1.5 times the throughput of a modern desktop CPU on a single Tegra K1 chip with only 11 W system power, running a state-of-the-art scene labeling ConvNet on 320×240 at 10 frames per second and with a latency of less than two frames.

To sum up, this work provides the fastest and most power efficient implementation of a ConvNet for both, the Tegra K1 as well as current desktop GPUs such as the GTX780.

7. ACKNOWLEDGMENTS

This work was funded by armasuisse Science & Technology.

8. REFERENCES

[1] C. Bobda and S. Velipasalar, editors. *Distributed Embedded Smart Cameras*. Springer, 2014.

[2] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. In *arXiv:1410.0759*, Oct. 2014.

[3] R. Collobert. Torch7: A matlab-like environment for machine learning. *Proc. NIPSW'11*, 2011.

[4] A. Dundar, J. Jin, V. Gokhale, B. Krishnamurthy, A. Canziani, B. Martini, and E. Culurciello. Accelerating Deep Neural Networks on Mobile Processor with Embedded Programmable Logic. In *Proc. NIPS'13*, 2013.

[5] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning hierarchical features for scene labeling. *IEEE Trans. on PAMI*, 2013.

[6] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Proc. IEEE CVPRW'11*, pages 109–116, June 2011.

[7] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In *Proc. IEEE CVPR'14*, pages 682–687, 2014.

[8] S. Gould, R. Fulton, and D. Koller. Decomposing a scene into geometric and semantically consistent regions. In *Proc. IEEE ICCV'09*, 2009.

[9] Y. Jia. Caffe: An Open Source Convolutional Architecture for Fast Feature Embedding, 2013.

[10] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini, and E. Culurciello. An efficient implementation of deep convolutional neural networks on a mobile coprocessor. In *Proc. IEEE MWSCAS'14*, pages 133–136, Aug. 2014.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. NIPS'12*, 2012.

[12] M. Kumar and D. Koller. Efficiently selecting regions for scene understanding. In *Proc. IEEE CVPR'10*, pages 3217–3224, June 2010.

[13] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet inter-domain traffic, 2010.

[14] M. Mathieu, M. Henaff, and Y. LeCun. Fast Training of Convolutional Networks through FFTs. In *arXiv:1312.5851*, Dec. 2013.

[15] D. Munoz, J. Bagnell, and M. Hebert. Stacked hierarchical labeling. In *Proc. ECCV'10*, 2010.

[16] F. Porikli, F. Bremond, S. L. Dockstader, J. Ferryman, A. Hoogs, B. C. Lovell, S. Pankanti, B. Rinner, P. Tu, and P. L. Venetianer. Video surveillance: past, present, and now the future [DSP Forum]. *IEEE Signal Processing Magazine*, 30:190–198, 2013.

[17] X. Ren, L. Bo, and D. Fox. Rgb-(d) scene labeling: Features and algorithms. In *Proc. IEEE CVPR'12*, pages 2759–2766, June 2012.

[18] M. Seyedhosseini and T. Tasdizen. Scene Labeling with Contextual Hierarchical Models. In *arXiv:1402.0595*, Feb. 2014.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. In *arXiv:1409.4842*, Sept. 2014.

[20] Y. Taigman and M. Yang. Deepface: Closing the gap to human-level performance in face verification. In *Proc. IEEE CVPR'13*, 2013.

[21] Teradeep Inc. Teradeep Technology Website, 2014.

[22] J. Tighe and S. Lazebnik. Superparsing: scalable nonparametric image parsing with superpixels. In *Proc. ECCV'10*, 2010.