CaffePresso: Accelerating Convolutional Networks on Embedded SoCs

Gopalakrishna Hegde, Nanyang Technological University, Singapore *gplhegde@gmail.com*
Siddhartha, Nanyang Technological University, Singapore *sidmontu@gmail.com*
Nachiket Kapre, University of Waterloo, Canada *nachiket@uwaterloo.ca*

CCS Concepts: •**Computer systems organization** → **Embedded hardware;** *Multicore architectures; System on a chip;* •**Computing methodologies** → *Machine learning;*

Additional Key Words and Phrases: FPGA, GPU, DSP, Deep Learning, Embedded, NoC

ACM Reference Format:
Gopalakrishna Hegde, Siddhartha, and Nachiket Kapre, 2017. CaffePresso: Accelerating Convolutional Networks on Embedded SoCs. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 24 pages.
DOI: 0000001.0000001

Auto-tuning and parametric implementation of deep learning kernels allow off-the-shelf accelerator-based embedded platforms to deliver high performance and energy efficient mappings of the inference phase of lightweight neural networks. Low-complexity classifiers are characterized by operations on small image maps with 2–3 deep layers and few class labels. For these use cases, we consider a range of embedded systems with 20 W power budgets such as the Xilinx ZC706 (FPGA), NVIDIA Jetson TX1 (GPU), TI Keystone II (DSP) as well as the Adapteva Parallella (RISC+NoC). In CaffePresso, we combine auto-tuning of the implementation parameters, and platform-specific constraints deliver optimized solution for each input ConvNet specification.

1. INTRODUCTION

Recent advances in deep learning convolutional neural networks [Russakovsky et al. 2015] (ConvNets) have opened the door to a range of interesting computer vision and image processing applications. Modern accelerator-based embedded SoC platforms are able to support novel computer vision applications with demanding requirements such as video analytics in smart cameras, drone-based image processing, medical patient monitoring, automotive navigational intelligence, among many others. Unlike large-scale, high-resolution, deep learning networks, the scope of the embedded classification task is restricted to a few classes (*e.g.* detecting humans, identifying roadblocks, classifying a few faces). They are typically supported by training datasets operating on smaller resolutions and in these circumstances, the primary objective is energy efficiency and low latency of response. For instance, real-time pedestrian detection [Cai et al. 2015] in autonomous vehicles can be performed in a two-step approach where higher-level computer vision routines extract smaller subsets of the image for subsequent processing with deep learning flows.

For embedded scenarios, deep learning computations can be easily offloaded to DSP, GPU, and FPGA accelerators as they support high processing throughputs with very low power consumption. While there is significant interest in design of ASICs customized for deep learning [Dally 2015; Han et al. 2016; Esser et al. 2015; Cavigelli et al. 2015; Chen et al. 2016], along with large FPGA-based [Ovtcharov et al. 2015; Zhang et al. 2015; Gokhale et al. 2014] and GPU-based accel-

Author's addresses: G. Hegde and Siddhartha, School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798; N. Kapre, Department of Electrical and Computer Engineering, University of Waterloo, Canada N2L 3G1. Email: *gplhegde@gmail.com, sidmontu@gmail.com, nachiket@uwaterloo.ca*

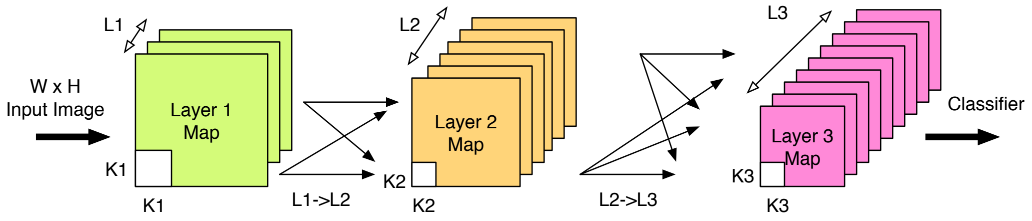


Fig. 1: High-Level Overview of Deep Learning Convolutional Networks (3-layer sample network shown). Showing parameters *i.e.* number of maps L_i, 2D convolutional kernel sizes $K_i \times K_i$, fully-connected layers, and the image resolution $W \times H$.

erators [Wu et al. 2015] for high-performance systems, we investigate the potential of commercial off-the-shelf SoC hardware for efficiently implementing these tasks at lower cost and power requirements in an embedded context. Modern embedded SoCs with accelerators present a unique mapping challenge with differences in the kinds of parallelism best supported by the attached accelerators, the available on-chip buffering capacity and off-chip bandwidths. Furthermore, each platform ships with ready-to-use libraries that are often tricky to optimize when composed to construct large applications. We also consider a direct-convolution approach instead of the popular GEMM-based formulation that requires extensive unrolling of the convolution maps which may be unsuitable for constrained embedded scenarios. In this paper, we develop a Caffe-compatible code generation and optimization framework that allows us to generate deep learning stacks tailored to different embedded SoC platforms. We are able to compare and contrast the performance, power and suitability of embedded multi-core CPUs, DSPs, GPUs and FPGA-based embedded SoC platforms for acceleration of a representative set of deep learning benchmarks. This is an extended version of our earlier work [Hegde et al. 2016]. We highlight the promise and challenge for the platforms we consider below,

- **GPUs**: GPU-based SoC platforms such as the NVIDIA Jetson TK1 and TX1 systems are a popular choice for supporting embedded computer vision problems. They offer high data-parallel processing throughputs and naturally map the convolution-rich nature of deep learning code to floating-point ALUs. Furthermore, we can easily exploit the highly-optimized cuDNN [Chetlur et al. 2014] library for NVIDIA GPUs that reformulate parallelism in the deep learning computations into SIMD matrix operations.
- **DSPs**: DSP-based platforms such as the TI Keystone 2 are a competitive alternative that exploit an energy-efficient multi-core VLIW organization. The TI C66 DSPs considered in this study are organized as VLIW engines that combine multiple instructions (arithmetic, memory, control) into a single-cycle for high performance. While these DSPs have optimized `DSPLib` and `IMGLib` libraries that take full advantage of the DSP cores, we wrap these low-level routines into a patch-based partitioned approach that takes full advantage of the eight DSP cores while keeping intermediate maps resident in the on-chip MSMC RAMs to the fullest extent possible.
- **Multi-Cores**: The Adapteva Epiphany III SoC is an exotic multi-core floating-point architecture supported by a message-passing NoC (network-on-chip). The key benefit of this alternate organization is the low power consumption of the 16-core chip (≈1–2 W for the chip) due to RISC-like CPU organization and energy-efficient on-chip data movement over the NoC. We develop an optimized library for deep learning computations by simultaneously managing compute optimizations on the CPUs along with concurrent data transfers on the NoC.
- **FPGAs**: Usually, FPGAs can deliver higher energy efficiencies through fully-customized dataflow circuit-oriented operation and close coupling with the memory subsystem through a long and complex RTL-based design flow. In this paper, we consider the Vectorblox MXP [Severance and Lemieux 2013] soft vector processor as a way to simplify the FPGA programming burden while retaining the advantages of the FPGA fabric. We develop a customized framework

for writing parameterized vector code, scratchpad-friendly routines, and flexible DMA transactions that are suitably scheduled for high throughput operation.

For efficient algorithm mapping to the different architectures, we must match the structure of the application with the potential of the computing fabric. A key consideration that determines the effectiveness of the mapping is the balance between memory bandwidth and pixel-processing capacity. While we expect NVIDIA GPU-based SoCs to perform particularly well due to the highly-optimized nature of the cuDNN libraries, we observe that other SoC architectures outperform the GPU particularly for smaller datasets. Emerging embedded applications in autonomous vision, drone-based surveillance, and other intelligent vision-based systems will often integrate a heterogeneous mix of SoC-based boards. In these circumstances, we need to be prepared to exploit available parallel processing capacity to deliver deep learning services where required.

The key contributions of this paper include:

1. We develop Caffe-compatible backends for Deep Learning configurations including small networks for datasets such as MNIST, CIFAR10, STL10, and Caltech101 as well as larger networks such as AlexNet for the ImageNet dataset on various accelerator-based SoCs. To our knowledge, there is no existing Caffe support for the Keystone II SoC (DSP portion), the Epiphany-III, or the MXP vector overlay.
2. We also provide in-depth performance analysis for the Keystone-II DSP for explaining the performance wins over the GPU. We characterize the performance of a GEMM-based approach on the Keystone-II DSP and identify conditions under which the direct-convolution approach delivers better results.
3. We write parametric code for the different kernels and implement auto-tuning flows for the TI Keystone II (DSP), Parallella Epiphany (Multi-core), and Xilinx ZC706 Zynq system (FPGA). For the NVIDIA X1 SoC, we use the already-optimized cuDNNv4 library.
4. We quantify and analyze the performance and energy efficiency for different datasets and various optimization strategies across the embedded SoC platforms.
5. Finally, we package our code and tuning scripts as an open-source release at https://github.com/gplhegde/caffepresso.git. We hope the code serves as a starting point for further research, analysis and development in the community.

2. BACKGROUND

2.1. Convolutional Neural Networks

Convolutional neural networks (ConvNets) are a powerful machine learning framework that finds widespread application in computer vision and image processing. For our embedded implementation context, we are primarily interested in energy-efficient acceleration of classifiers with few class labels. For instance, MNIST and CIFAR10 detect ten handwritten digits and ten object classes respectively. In these scenarios, the classification model is trained offline for high accuracy and requires fast evaluation of the forward pass where we perform actual classification with real data. The backward training pass is a one-time task that can be performed per dataset and is neither critical for acceleration for small embedded datasets nor constrained by platform limitations (*i.e.* training can be done on a more capable machine(s) with compatible arithmetic libraries). In this paper, we do not perform an training on the embedded platform. Beyond these simplistic datasets, deep learning computations on small image sizes are still important in embedded scenarios such as real-time pedestrian detection [Cai et al. 2015] in autonomous vehicles. In such cases, higher-level computer vision routines can help extract smaller subsets of the image for in-depth analysis with deep learning flows.

ConvNets are organized into multiple layers consisting of a combination of convolution, pooling, and rectification stages followed by a final classifier stage as shown in Figure 1. Caffe [Jia et al. 2014] supports a wide variety of such layers to achieve various classification tasks, but for our scenarios, we only need a few of these layers. The structure of the algorithm is organized to replicate biological behavior in an algorithmic manner that can be trained and customized for vari-

Table I: Various datasets and associated ConvNet configurations tested in this paper. Some combinations do not require the 3rd layer. 2x2 pool and subsample in layers where not specified. Fully connected layers not shown ($<$10% fraction of total time).

Dataset	Layer 1	Layer 2	Others	FC Layer	Ops.
MNIST	5 28×28 maps	50 8×8 maps	-	fc1: 800×500	8.9 M
1×28×28	5×5 kern.	5×5 kern.	-	fc2: 500×10	
	2×2 pool	2×2 pool			
CIFAR10	32 28×28 maps	32 14×14 maps	64 7×7 maps	fc1: 576×500	14.6 M
3×32×32	5×5 kern.	5×5 kern.	5×5 kern.	fc1: 500×10	
	2×2 pool	2×2 pool	3×3 pool		
STL-10	32 92×92 maps	32 42×42 maps	-	fc1: 14112×500	139 M
1×96×96	5×5 kern.	5×5 kern.	-	fc2: 500×10	
	2×2 pool	2×2 pool			
Caltech101	64 142×142 maps	256 19×19 maps	-	fc1: 4096×5000	2.3 G
3×151×151	9×9 kern.	9×9 kern.	-	fc2: 5000×101	
	10×10 pool	6×6 pool	-		
	5×5 subsm.	4×4 subsam.	-		
ImageNet[1]	96 55×55 maps	256 27×27 maps	384, 384, 256 13×13 maps	fc1: 4096 neurons	1.7 G
3×227×227	11×11 kern.	5×5 kern.	3×3 kern.	fc2: 4096 neurons	
	3×3 pool	3×3 pool.	3×3 pool.	fc3: 1000 neurons	

[1]Based on Caffe's implementation of AlexNet – https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet

ous classification requirements. Thus, for each ConvNet, a specific arrangement of layers achieves the classification outcome as desired for that dataset. As we see in Table I, the resolutions, depth of layers, the sizes of kernels and the associated compute complexity of the different network vary dramatically with the dataset. These parameters are derived from published literature and are chosen by computer vision experts. We simply use these as a starting point for our embedded implementation. Within each layer, we generate various feature maps that recognize higher-level patterns to enable simpler final classification. For object detection and recognition in images, an individual layer typically involves a 2D convolution with trained coefficients to extract feature maps, pooling operations to subsample the image to lower resolutions, followed by a non-linear rectification pass. Across layers, we must communicate various feature maps before repeating this compute step. With suitable training data and appropriate training of convolution weights, we can generate classification vectors that can recognize class labels with high accuracy. Our embedded implementation is parameterized in terms of (1) number of layers in the network N, (2) number of feature maps in each layer L_i, (3) kernel sizes for 2D convolutions in each layer $K_i \times K_i$, (4) pooling and subsampling factors (usually 2×2), as well as (5) stride length for the pooling/subsampling layers as well (2–3). We show the various configurations for the embedded datasets we use in this study in Table I.

2.2. Computational View of ConvNets

We can better understand the compute flow through the pseudocode for the complete network shown in Figure 2. The nested for-loop arrangement provides a simplified view of the computational structure of memory access order and arithmetic complexity. The outermost loop l over `LAYERS` is a sequential evaluation and organizes dataflow through the different layers, extracting features along the way. Within each layer, we must loop over the maps N in that layer l as we did in the preceding layer $l-1$. This nested loop over m and n variables requires a careful decision on parallelization strategy, as well as a mechanism for storage of the convolution maps (`in` and `out`) and kernels (`kern`). The inner loops iterate over the pixels in a map x (map rows) and y (map columns), along with that over kernel coefficients r (kernel rows) and c (kernel columns) perform a straightforward

```
for(int l=0;l<LAYERS;l++) {             // different layers of deep ConvNet
  for(int m=0;m<N[l];m++) {             // number of output maps in current layer
    out[m]=memset(0);
    for(int n=0;n<N[l-1];n++) {         // input maps from the previous layer
      for(int x=0;x<ROWS;x++) {         // number of rows in a map
        for(int y=0;y<COLS;y++) {       // number of columns in a map
          for(int r=0;r<K;r++) {        // kernel row count
            for(int c=0;c<K;c++) {      // kernel column count
              out[m][x][y] += in[n][x+r][y+c]*kern[m][n][r][c];
            }
          }
        }
      }
    }
  }
  in=out;                              // output of layer m = input to layer m+1
}
```

Fig. 2: High-Level Code Sketch of a ConvNet. (Only showing 2D convolution, but pooling and rectification have very similar nested loop structure. Each of the loops nest everything below, curly braces now shown for simplicity.)

2D convolution of the map. The innermost statement performs the multiply-accumulate operation for each pixel to deliver a 2D convolution result. We do not show pooling or normalization steps, but those can follow directly after the multiply-accumulate with some adjustments for varying stride lengths. For the ConvNet implementations we consider, we also do not need to use across-map normalization making it possible to compute the result completely local to each map. Apart from the outermost loop over l, all other for loops can be parallelized. As dictated by the constraints of the specific architecture, we may parallelize a different subset of these loops. We also need to carefully consider the storage costs of the intermediate maps `in` and `out` and may have to insert another for loop over patches of the maps (not shown, but see Section 3.2 later) to ensure fully on-chip operation. Except for the smallest ConvNets, and a few embedded platforms, we cannot store all kernel weights across all layers insider the chip. They need to be fetched from off-chip as required but must be done with care to avoid performance loss.

A matrix multiplication (GEMM) formulation of the code in Figure 2 has been shown to be better than direct convolutions for CPU and GPU-based platforms [Chellapilla et al. 2006; Chetlur et al. 2014]. However, this approach requires explicit unrolling of the input data by a factor of K^2 to extract sufficient parallelism at small map sizes. For embedded memory-constrained platforms, we hypothesize that this is not an appropriate approach, and we choose to implement direct convolutions instead. For the Keystone-II DSP, we quantify the performance gap between these approaches in Section 5.6, and identify the conditions under which the GEMM-based formulation may be better.

Table II: Raw timing numbers for one 32×32 image patch on Jetson TX1 (CPU and GPU) using `Caffe` + `cuDNNv4`.

Function	**Caffe API**	**Jetson TX1 Time (ms)** ARM	GPU	Ratio
2D Conv	`conv`	1.196	0.117	10.2
Pool	`pool`	0.124	0.015	8.2
Norm	`relu`	0.191	0.014	13.6

In its simplest form, the ConvNets typically invoke 2D convolutions, pooling, rectification, and fully-connected tasks as required for the particular dataset. For embedded implementation, we quantify the stand-alone computational complexity and runtime (32×32) on the ARMv8 32b (Jetson TX1's CPUs) and Maxwell 256-core GPU (Jetson TX1's accelerator) in Table II. As we can see, beyond the obvious 8–13× acceleration advantage when using the GPU, 2D convolutions are overwhelmingly the slowest computations in this stack. Particularly, as we scale the kernel size, runtime scales quadratically. For pooling, the stride length has some impact on performance as the cache locality is affected for irregular strides. When se-

Table III: Platform specification of the FPGA, GPU and DSP-based SoC boards.

Platform	**Jetson TX1**	**66AK2H12**	**Parallella**	**ZC706**
Vendor/SoC	NVIDIA Tegra X1	TI Keystone II	Adapteva Epiphany-III	Xilinx Zynq Z7045
Technology	20nm	28 nm	65nm	28nm
Processor	ARMv8+NEON	ARMv7+NEON	ARMv7 (Zynq)	ARMv7+NEON
Accelerator	Maxwell GPU 256-core	C66 DSP 8-core	Epiphany-III 16-core	Kintex FPGA 32 lanes
Host Clock	1.9 GHz	1.4 GHz	667 MHz	667 MHz CPU
Accelerator Clock	1 GHz	1.2–1.4 GHz	667 MHz	180 MHz FPGA[1]
Onchip Host Mem	64 KB L1 + 2048 KB L2	32 KB L1 + 1 MB L2	32 KB L1 + 512 KB L2	32 KB L1 + 512 KB L2
Onchip Accel. Mem	64 KB[3]	6 MB MSMC	32 KB/core	64 KB SRAM[2]
Off-chip Memory	4 GB 32b LPDDR4-800	2 GB 72b DDR3-1600 2×	1 GB 32b DDR3-1066	2 GB 32b DDR3-1066
System Power	6 W (Idle), 10 W	11 W (Idle), 14 W	3 W (Idle), 4 W	17 W (Idle), 19 W
Host OS	Ubuntu 14.04	Bare-Metal (no OS)	Ubuntu	Bare-Metal (no OS)
Compiler +	gcc-4.8.4, nvcc 7.0	TI CCSv6, DSPLibv3.1.0.0	e-gcc	gcc-4.6.3
Library	cuDNN v4	IMGLib v3.1.1.0		Vivado 2014.2
Cost[4] (USD)	$599	$997 ($667/chip)	$126	$2275 ($1596/chip)

[1]FPGA peak 250+ MHz, VBX runs at 110 MHz. [2]FPGA 560 KB RAM, VBX uses 64 KB. [3]`__shared__` RAM is 48KB/thread, but 64KB total. [4]Prices from vendors/component resellers are approximate and for high volume use as observed in May 2016.

quencing a series of ConvNet layers, the storage of intermediate maps can become a challenge for embedded platforms with limited on-chip capacity. Hence, the optimization focus for constrained embedded platforms needs to be on faster 2D convolutions (parallel arithmetic operations) as well as smarter data sharing between multiple layers of the ConvNet stack.

2.3. Architecture Potential

We are interested in identifying the fastest and most energy efficient SoC platform for implementing embedded deep learning computations. To understand the potential of our accelerator enhanced SoCs, we first calculate the raw *datasheet* potential of the GPU, DSP, Multi-core and FPGA SoCs. We visualize the high-level organizations of these SoCs in Figure 3 and list the relevant specifications in Table III. We tabulate the key throughput, power and efficiency trends in Table IV. They are all supported with ARM host CPUs coupled to accelerators either via on-chip AXI busses, NoC links, or shared cache/memory interfaces. The TI DSP and FPGA (MXP vector engine) implement pixel operations in 16b fixed-point (64b or 32b extended precision for accumulations for respective platforms) rather than floating-point. The NVIDIA GPU and Epiphany SoC support single-precision floating-point. We note that pixel arithmetic can be easily implemented with a combination of 8b, 16b and 32b fixed-point operations while achieving similar accuracy as floating-point implementations. We enumerate and explain key specifications and capabilities of the various platforms:

- **GPU**: The Tegra X1 SoC contains a quad-core 1.9 GHz ARMv8 CPU (Cortex A-57) with NEON support and a large 2 MB L2 cache. The GPU accelerator in the Tegra X1 SoC contains 256 single-precision floating-point Maxwell cores that run at 1 GHz supported by a 64 KB L1 cache. This translates into a theoretical throughput; 256× 1 GHz CUDA cores = **256 Gops/s** (float, multiply+add counted as one operation). The Jetson TX1 board consumes 10–12 W power under varying load.
- **DSP**: The TI Keystone 2 board (66AK2H12) ships with dual-core ARM (Cortex-A15) with 2 MB shared L2 cache running at 1.4 GHz clock. The Keystone II has eight-core C66 DSPs that can process 32 16x16 integer multiply-accumulate operations per cycle in VLIW fashion running at 1.4 GHz. This gives the Keystone II slightly higher processing capacity than the GPU; 32 × 1.4 GHz × 8 C66 cores = **358.4 Gops/s**. The 66AK2H12 Keystone II board consumes 11-14 W under varying load.

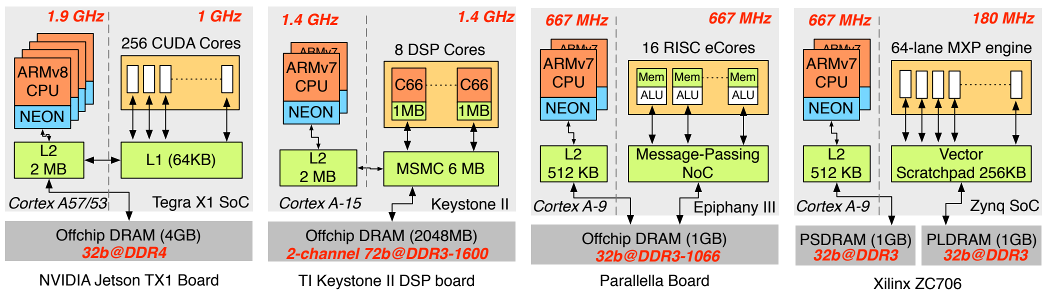


Fig. 3: Comparing various Embedded Accelerator-based SoC Platforms used in this study. Each platform provides a unique mix of accelerator elements and control units and varies in their on-chip memory organizations and communication support.

- **Multi-Core + NoC**: The multi-core Epiphany-III chip relies on a low-end Zynq SoC with an ARMv7 32b CPU as a front-end for programming and data transfers. The Artix-class (low-end) FPGA logic is used to provide a configurable fabric to connect the data and control via eLink connections to the Epiphany chip. The multi-core Epiphany-III SoC supports 16 eCores that implement a simple RISC-inspired instruction set and has a unified 32KB scratchpad memory per eCore to store both program code and data. The eCores run at 667 MHz and are optimized for 32b floating-point multiply-add operations with limited support for 32b integer operations. The Epiphany can deliver roughly $\frac{1}{20}$th the GPU throughput; 16 $\times$ 667 MHz = **10.5 Gops/s** (float, multiply-add counted as one operation). The Parallella board consumes 3–4 W of power (inclusive of the supporting FPGA).
- **FPGA**: The Zynq FPGA SoC on the ZC706 ships with dual-core ARMv7 32b CPU (Cortex A-9) with NEON support and a slow 667 MHz clock frequency. The FPGA on the SoC uses the higher-performance Kintex-class fabric with 218K LUTs, 437K FFs, 900 18$\times$25 bit DSP blocks and 2180 KB of separate non-coherent scratchpad memory distributed across multiple discrete Block RAMs. The CPU and the FPGA communicate over AXI busses that can operate at a throughput of 2 GB/s with shared access to the 1 GB off-chip DDR DRAM. When configured with the 32b fixed-point MXP soft vector processor for pixel processing, we achieve a throughput of 64-lane $\times$ 180 MHz = **11.5 Gops/s**. This is a paltry $\frac{1}{20}$th the throughput of the GPU, further compounded by the lack of fused multiply-add support. The peripheral-rich ZC706 board consumes 19 W under load. In ideal scenario, the peak FPGA throughput is 900 $\times$ 18x25 FPGA DSPs $\times$ 250 MHz = **225 Gops/s** (16b), but when configured with the MXP soft vector processor (see Section 2.4 for details) this drops to 64-lane $\times$ 180 MHz = **11.5 Gops/s** (16b ops), which is 20$\times$ lower. Additionally, the 256 KB scratchpad is only using $\frac{1}{9}$th of the 2.3 MB on-chip Block RAM capacity of the device. These are limitations imposed by the Vectorblox IP itself and beyond the control of the authors. We have provided feedback to the vendor to improve the device utilization of the MXP engines for future releases.

We present the calculations from Table IV as a roofline plot in Figure 4 and see the clear separation between the different platforms. The roofline plot allows us to quickly identify whether a given platform will be memory bandwidth bottlenecked or compute limited based on arithmetic intensity of the user application. Arithmetic intensity is the inverse ratio of bytes fetched from external DRAM to the number of arithmetic operations performed on this fetched byte. For instance, 2D convolution of kernel $k \times k$ will have an arithmetic intensity of k^2 *i.e.* per pixel fetch from DRAM will be followed by k^2 multiply-add operations. The FPGA and Epiphany platforms appear the weakest of the set of systems we consider with the Keystone II DSP dominating the race at high arithmetic intensities. The TX1 GPU has higher DRAM bandwidth (steeper slope in Figure 4) which works well for ConvNet scenarios where the ratio between compute and memory access is

Table IV: Performance-Power specs of various SoC boards. FPGA and DSP operations are fixed-point, while GPU and Multi-core operations are single-precision floating-point.

Platform	**Jetson TX1** GPU	**Keystone II** DSP	**Parallella** Multicore	**ZC706** FPGA
Tput. (Gops/s)	256	358.4	10.5	11.5
B/W (GB/s)	25.6	12.8	1.3	6.4
Power (W)	10	14	4	19
Efficiency (Gops/s/W)	25.6	25.6	2.6	0.3

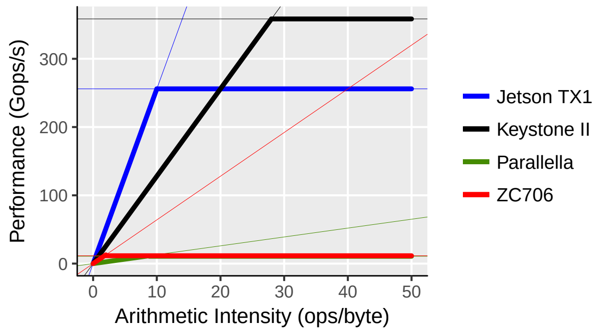


Fig. 4: Roofline analysis based on Peak Gops/s (horizontal lines) and DRAM bandwidths (diagonal lines). The DSP dominated peak throughput while the GPU has greater bandwidth. FPGA and Parallella have much lower peak throughputs. DSPs and FPGAs are better balanced, allowing ALU capacity to be exploited with arithmetic intensity as low as 2–3 operations/byte.

lower (smaller values of kernel size k). The Parallella and FPGA platforms are better balanced in terms of ALU peak throughput and DRAM bandwidth, and allow the application to quickly saturate the ALUs with arithmetic intensity as low as 2–3 operations/byte transferred. While GPUs enjoy widespread popularity among deep learning consumers [Wu et al. 2015], the TI DSP is a formidable competitor. The ability to explicitly manage memory transfers, and account for all operations on a per-cycle basis during mapping and implementation allow DSP-based platforms to outperform the GPU-based SoC as we will see in subsequent sections.

2.4. MXP Soft Vector Processor

While the FPGA is a fully configurable fabric, we choose to program the device with an overlay to simplify programmability. This is similar in spirit to the use of off-the-shelf software APIs for other platforms where available. The MXP soft vector processor is the intermediate overlay architecture programmed on top of the FPGA. It is organized as a flexible vector processor with configurable 180 MHz 64-lane 32b mode, 128-lane 16b mode, or 256-lane 8b mode as appropriate for pixel processing arithmetic. The vector ALUs are connected to banked on-chip FPGA Block RAMs that support a lightweight permutation network to source operands. There is no explicit register file as the FPGA Block RAMs can be easily configured as dual-ported RAMs to fulfill the same role. Furthermore, the FPGA Block RAMs are organized as *scratchpads* instead of caches to provide direct control of off-chip memory bandwidth for high performance transfers. The MXP processor is programmed using a vector API to simplify data movement and vector operation. It is the direct control of memory transfers, and vectorization and handling of instruction issue costs that allows the MXP design to achieve throughputs close to peak potential. For the filtering calculations and associated operations, we observe that we end up using the MXP in mostly 16-lane mode due to 32b writeback of intermediate results for 16b inputs. Furthermore, the limited scratchpad capacity forces us to explicitly manage DMA transfers and schedule operations to minimize DMA costs.

The overlay does not use the full capacity of the ZC706 card, but still helps identify a series of weaknesses that can be rectified by the vendor (Vectorblox). For the ZC706 board, there is a possibility to provide either larger MXP units that board 256-lane SIMD units in a single core. However, after consultation with the vendor, Vectorblox, it became apparent that the design frequency does not scale smoothly for such large SIMD lane widths. A superior alternative is to create multi-core SIMD units with 64 lanes per SIMD unit. This is currently under design and development at Vectorblox based on the results of the CaffePresso library [Hegde et al. 2016].

3. OPTIMIZATION STRATEGIES

Based purely on operation complexity, most mappings of deep learning computations will spend a bulk of their time performing 2D convolutions. Furthermore, the sequencing of various convolutional layers will generate large amounts of memory traffic due to dependencies of inter-layer intermediate maps, while also needing memory management for storing these maps. Thus, when optimizing convolutional networks for selected embedded platforms, it is important to have a strategy for (1) parallelizing the problem, (2) data storage management for intermediate maps, and (3) communication management for moving inter-layer results. Unlike solutions (FPGAs [Ovtcharov et al. 2015; Zhang et al. 2015; Gokhale et al. 2014], GPUs [Wu et al. 2015]) using abundant hardware resources, embedded solutions are severely constrained by the limited capacity of logic, on-chip memory and communication bandwidth. This presents a unique challenge for code generation and optimization. We characterize the systems offline to understand the sustainable bandwidths and ALU throughputs and base our optimizations on top of this initial analysis.

3.1. Parallelism

As discussed earlier in Section 2 for Figure 2, most of the for loops in the simple view of a ConvNet implementation can be parallelized. There is parallelism across maps, pixels, and kernel coefficients. This parallelism at various levels of granularity can be effectively mapped to accelerators to exploit the specific features that may be unique to that platform. We discuss these properties in greater detail below:

1. **Pixel-level**: There are naturally data-parallel operations in convolutional kernels that can easily match the SIMD or VLIW organizations of ALUs in the various accelerators. Each pixel can be processed in parallel provided sufficient storage and bandwidth for neighbouring pixels is available. The platform-optimized libraries directly use VLIW or SIMD intrinsics as appropriate to exploit this obvious parallelism to the fullest extent. For instance, the loops over x (map rows) and y (map columns) can be fully parallelized within a map. A representative timing schedule is shown in Figure 6 along with an associated code sketch in Figure 5.

```
for(int x=0;x<ROWS;x+=SIMD)         // number of rows in a map
  for(int y=0;y<COLS;y+=SIMD)       // number of columns in a map
    SIMD_func(x,y);
```

Fig. 5: Parallel SIMD operation on `SIMD`×`SIMD` pixels.

2. **Row-level**: We have concurrent streaming parallelism in most stages of the ConvNet flow, that allows us to perform parallel work on pixels within a row and also orchestrate optimized data movement in larger DMA chunks for best memory system performance. Pseudocode and a timing schedule of such an overlapped operation is shown in Figure 7 and Figure 8 where DMA reads and writes are overlapped with compute. Pixel operations in 2D convolution and pooling stages require access to neighbouring pixels, and row-level storage greatly helps avoid expensive DRAM accesses. The DSP and GPU memories and the FPGA MXP scratchpads allow multi-ported concurrent operation while the Epiphany provides a NoC with explicit message-passing.

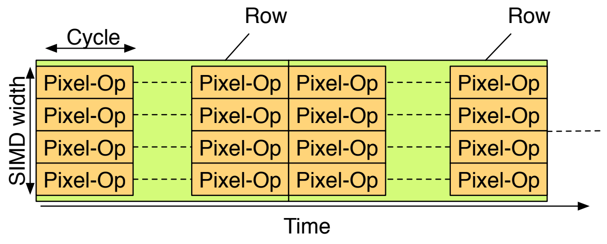


Fig. 6: Pixel-level parallelism through SIMD evaluation.

```
DMA(input[0],COLS);              // first DMA transfer
for(int x=0;x<ROWS;x++)          // number of rows in a map
  DMA(input[1],COLS);            // get next row
  for(int y=0;y<COLS;y++)        // number of columns in a map
    conv2D(x,y);                 // perform 2D convolution at pixel x,y
```

Fig. 7: Overlapped DMA operations on rows.

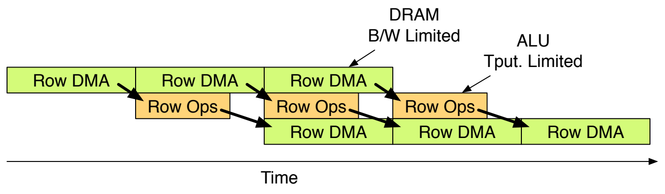


Fig. 8: Row-level *streaming* parallelism through concurrent DMA/compute processing. DRAM B/W $<$ ALU Tput.

3. **Dataflow**: When composing multiple layers of the deep learning computations, it is often possible to serially evaluate dependent functions at the granularity of rows instead of image frames. This optimization allows on-chip caching of data between the dependent functions without spilling over to the off-chip DRAM. This improves performance while also reduces energy use by eliminating DRAM accesses for storage of intermediate maps. In Figure 9, and Figure 10, we illustrate how 2D convolve and pooling optimization can often be fused. The efficacy of this optimization is purely limited by the size of on-chip storage on the various SoC platforms.

```
DMA(input[0],COLS);              // first DMA transfer
for(int x=0;x<ROWS;x++)          // number of rows in a map
  DMA(input[1],COLS);            // get next row
  for(int y=0;y<COLS;y++)        // number of columns in a map
    conv2D(x,y);                 // perform 2D convolution at pixel x,y
    pool(x,y);                   // immediate pool the pixels
```

Fig. 9: Dataflow operation of `conv2D` and `pool` operations.

4. **Function-level**: Finally, since each layer must generate multiple feature maps, those can also be entirely parallelized across maps. This must be managed with care as inter-layer map copies require copious amounts of communication bandwidth. For data that must be spilled out to off-chip DRAMs for inter-function dependency, the DSPs and GPUs use a cache hierarchy while the FPGA and Epiphany require low-level DMA routines. For multi-core, DSPs and RISC machines,

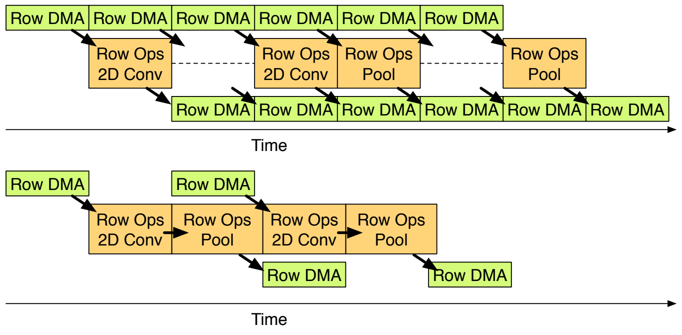


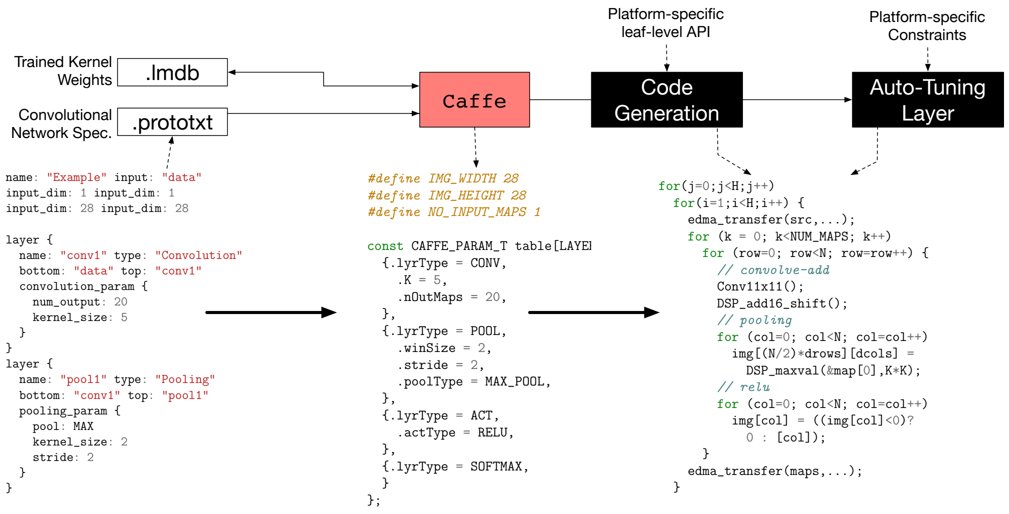
Fig. 10: Dataflow optimization of dependent row-oriented functions. Interleaved operation across consecutive functions possible when dependencies are linear. Top schedule shows original data transfers, while bottom row shows reduction in DMA transfers due to Dataflow optimization.

the first stage of parallelizing the code is to exploit this function-level parallelism across maps to distribute the work across cores.

3.2. Memory Management

Each platform we consider in this study has varying capacities of on-chip storage ranging from 512KB (32KB/core, Epiphany-III), 64KB (TX1 GPU), 256KB (Vectorblox FPGA overlay), to 6MB (MSMC shared RAM, Keystone-II). Additionally, the ConvNet configurations we explore also impose varying requirements for storage of intermediate maps. Hence, we developed a flexible memory management strategy that exploits the available capacity of the different SoCs as appropriate. For small-scale convolutional networks, such as MNIST and CIFAR10 (shown earlier in Table I), we are able to fit the intermediate maps and temporary results in on-chip RAMs of most SoCs. However, networks, such as Caltech101 and ImageNet that are larger than the low-complexity classifiers like MNIST and CIFAR10, have memory requirements that exceed available on-chip capacity.

— The Jetson TX1 manages its on-chip memory through cuDNN and Caffe and we use them directly without modification.
— The Epiphany-III SoC permits fully on-chip operation for the MNIST and CIFAR10 benchmarks, and we use explicit data movement between cores to facilitate the layer-by-layer computation. We use a *patch*-based partitioning strategy to decompose larger image frames into smaller sizes that can operate entirely on-chip with ease. This requires an extra redundant copy of the image border ghost pixel region around each patch to enable fully on-chip operation when the filter kernel window falls outside the patch region. This allows multiple depths of the convolutional network to be calculated for that patch without any inter-patch communication *i.e.* fully on-chip. While the idea of decomposing images into patches is nothing new, our contribution is the design of an **auto-tuning** optimization backend that balances the benefits of fast on-chip processing possible via small patch sizes, vs. the extra DMA copying times for redundant data in choosing a patch-size for each SoC platform. For the Epiphany III SoC, we are required to divide the SRAM space between instruction and data memories manually ourselves. Inter-layer traffic is directly supported by the NoC on the Epiphany. The lower energy implementation possible on the Parallella is directly attributed to the on-chip communication hardware and conscious design of data transfers by the programmer.
— The small 256KB on-chip on the Vectorblox FPGA core only permits simple row buffering optimizations and all intermediate maps are stored externally in the DRAM. To overcome the inefficiencies of vector-scalar operations, we rearrange weights into vectors to exploit the faster performance of DMA transfers from the DRAM to the scratchpad. The FPGA MXP vector processor only provides a configurable 32–256 KB scratchpad, thereby forcing low-level DMA calls


Platform-specific leaf-level API
Platform-specific Constraints
Trained Kernel Weights
.lmdb
Convolutional Network Spec.
.prototxt
Caffe
Code Generation
Auto-Tuning Layer

```
name: "Example" input: "data"
input_dim: 1 input_dim: 1
input_dim: 28 input_dim: 28

layer {
  name: "conv1" type: "Convolution"
  bottom: "data" top: "conv1"
  convolution_param {
    num_output: 20
    kernel_size: 5
  }
}
layer {
  name: "pool1" type: "Pooling"
  bottom: "conv1" top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

```
#define IMG_WIDTH 28
#define IMG_HEIGHT 28
#define NO_INPUT_MAPS 1

const CAFFE_PARAM_T table[LAYEI
  {.lyrType = CONV,
    .K = 5,
    .nOutMaps = 20,
  },
  {.lyrType = POOL,
    .winSize = 2,
    .stride = 2,
    .poolType = MAX_POOL,
  },
  {.lyrType = ACT,
    .actType = RELU,
  },
  {.lyrType = SOFTMAX,
  }
};
```

```
for(j=0;j<H;j++)
  for(i=1;i<H;i++) {
    edma_transfer(src,...);
    for (k = 0; k<NUM_MAPS; k++)
      for (row=0; row<N; row=row++) {
        // convolve-add
        Conv11x11();
        DSP_add16_shift();
        // pooling
        for (col=0; col<N; col=col++)
          img[(N/2)*drows][dcols] =
            DSP_maxval(&map[0],K*K);
        // relu
        for (col=0; col<N; col=col++)
          img[col] = ((img[col]<0)?
            0 : [col]);
      }
    edma_transfer(maps,...);
  }
```

Fig. 11: High-Level View of the CaffePresso flow. (Showing dummy prototxt, intermediate header, and skeleton DSP code.)

even for small network sizes. We are able to overlap most DMA calls with useful compute and minimize the performance impact of this resource constraint. For this platform, the patch sizes are rectangular and tend to favor contiguous DMAs of multiple row vectors.

— The TI DSP has substantially more RAM than the other platforms with the 6 MB MSMC shared RAM, and merits a separate design of the memory management logic. The TI DSPs allow the L1 and L2 memories to be configured as scratchpads instead of caches. In our experiments, we exploited this freedom to configure L1 memories as a cache, L2 memories as SRAM, and MSMC as the shared write-through memory for the L1 data cache. Here, the 6 MB MSMC RAMs are shared across all eight DSP cores while the 32 KB L1 and 1 MB L2 caches are private to each core. For small networks, we split the kernel weight storage between the L2 SRAM and MSMC RAMs, while larger networks often need weight storage in the external DDR memories due to their large sizes. For these larger networks, we pre-fetch the weights into the L2 and MSMC RAMs on a per-layer basis. For all cases, the fully-connected layer (the final classification stage) requires weight storage that exceed the on-chip capacities and are hence stored in the external DRAM. We use EDMA transfers (optimized for large transfers) when fetching weights into on-chip RAMs, but use `memcpy` for data movement within the chip as that proved to be a faster solution.

4. CAFFEPRESSO FLOW

In this section, we describe CaffePresso, our Caffe-based code-generation and auto-tuning framework along with platform-specific optimization notes that are relevant for integrating the complete solution and providing a template for generalizing to other platforms. While an end-user only needs to supply a ConvNet specification and choose the relevant platform, we expect the inner workings of our framework to assist in integration of new embedded platforms.

4.1. Mapping methodology

Our mapping flow, seen in Figure 11, is decomposed as follows:

— **Caffe input**: We use the open-source Caffe [Jia et al. 2014] deep learning framework as a frontend in our experiments. Caffe accepts representations of different convolutional networks in Google ProtoBuf format (`prototxt` files) to facilitate training (backward pass) and execution (forward pass) on CPU and GPU platforms out-of-the-box. This is coupled with the use of LevelDB (`lmdb` files) for storage of trained kernel weights. We achieve best-published accuracies (70–99$\%$) for each dataset and associated ConvNet configurations. Our pre-processor converts the ConvNet specifications into a sequence of calls to the platform-specific low-level APIs.

— **Code-Generation**: Caffe implements various layers required for assembling a ConvNet structure on a CPU. cuDNN [Chetlur et al. 2014] bindings allow Caffe to target the NVIDIA GPUs. We generalize the API to support multiple platforms and permit smooth integrating of different architecture backends beyond CPUs and GPUs. We compile the Caffe `protobuf` specification of the ConvNet structure into low-level platform-specific API calls for individual Caffe layers such as convolutions, subsampling, and pooling. These low-level routines (Caffe layers) are nested for loops that are hand-written in parametric fashion to expose optimization hooks to the auto-tuning flow. This translation of the ConvNet specification into platform-specific code forms the basis of our automated code-generator. If a developer wishes to add support for a new embedded platform, they need to supply implementations for each of the Caffe layers. These low-level APIs must be written in parametric manner to exploit our automated performance tuning flow.

— **Auto-Tuning**: As the low-level APIs are parameterized in terms of various options such as patch sizes, DMA burst lengths, storage choices in the memory hierarchy, and even compiler options, we explore a large space of possible solutions before determining the optimized mapping. This search is handled by our auto-tuner that tailors the final mapping for a given ConvNet specification to the target platform. We also build a predictive performance model based on sweeps of the parameter tuning space for each API call. This helps guide the optimization and also allows us to evaluate new ConvNet configurations quickly prior to execution on the device.

To the best of our knowledge no known Caffe-compatible backends [1] for these SoC platforms exist. We use Caffe to train the convolutional network and extract the trained weights. This is a one-time task and must be performed for each dataset. We then run the forward pass that used these trained weights for actual classification tasks for timing measurement and energy-efficiency comparisons. We build parametric implementations of the underlying vision tasks for each embedded platform used in our study and exploiting existing low-level libraries where possible.

4.2. Optimization Formulation

For network specifications that can fit entirely within the on-chip memories of the embedded SoCs, we select best performing functions for those particular sizes. This characterization step involves sweeping various parameter combinations such as number of maps (L_i), kernel sizes ($K_i \times K_i$) and pooling strides. For larger resolutions and problem sizes, we develop a lightweight optimizer that minimizes total processing time which is the sum of DMA time and actual arithmetic evaluation cost as a function of the *patch* size H subject to on-chip capacity constraints. This is implemented as a simple check of all feasible sizes (numbering in hundreds of combinations). This also allows us to build a predictive model for performance to estimate the runtimes for arbitrary ConvNet configurations on a given platform. A designer may use this tool to help select a platform that best meets requirements prior to system-level integration.

4.3. Platform-Specific Optimization

We use platform-specific optimizations to develop the parametric leaf-level APIs that can support any ConvNet specification. This recipe for optimization is generalizable to similar high-level organizations (VLIW, SIMD, NoC-based Multi-cores) and constraints.

(1) Before code generation, we first identify the performance limits of each platform through microbenchmarking to understand the ALU processing and memory constraints beyond the datasheet specifications of Table IV.

[1]TI Keystone II has an ARM-only Caffe-compatible backend, but that does not use the C66 DSPs.

```
for(int l=0;l<LAYERS;l++) {
  for(int m=0;m<N[l];m++) {
    kern1 = edma_transfer(kern[l][m],...)         // pre-fetch weights per layer and map
    for(int n=0;n<N[l-1];n++) {
      in1 = memcpy(in[n][0],...)                  // copy rows for processing
      for(int x=0;x<ROWS;x++) {
        memcpy(in[n][x+1],...)                    // copy rows for double buffer (next x)
        // perform 3x3 convolution using optimized IMGLib call
        IMG_conv_3x3_i16s_c16s(in1,kern1,temp,..)
        // accumulate across input maps using optimized DSPLib call
        DSPAdd16(temp,out[m][x])
      }
    }
    memcpy(out[m],...)                            // writeback output map
  }
}
```

Fig. 12: DSP Code Sketch for `conv`.

(2) This helps us determine a high-level parallelization and partitioning strategy for the various maps per layer as well as the memory organization for map storage and communication mechanisms for inter-layer movement of maps.
(3) The optimizations identified in Section 3 are then encoded into each implementation as appropriate. In particular, these optimizations target scratchpad-based and NoC-based platforms where movement of data must be explicitly specified.
(4) Finally, our auto-tuning framework chooses specific implementation parameters and degree of optimizations to fully customize the mapping for each platform and ConvNet combination. We expose optimization hooks that enable this tuning and support automated selection of suitable implementation parameters for best performance. For instance, we consider optimizations such as patch sizing, DMA burst length, loop unrolling, partitioning granularity, and scheduling choices to improve performance. These parametric hooks are automatically explored through the auto-tuner during the optimization process.

In the rest of this section, we show code templates for the `conv` layer of Caffe mapped to various platforms. These are used when synthesizing optimized implementations of deep learning stacks for the various Caffe `protobuf` descriptions targeting the different SoC boards.

We now discuss the individual backends and platform-specific optimization notes on implementation issues encountered during mapping. The coded sketches hide some detail in favor of clarity.

(a) TI DSPs (Figure 12): We program the TI Keystone DSP boards using a C-based API supported by optimized image processing libraries for leaf-level routines as shown in Figure 12.

— **Memory Management**: Off-chip memory communication has a significant performance penalty, which encourages us to utilize the on-chip shared memory to the fullest extent. As described earlier in Section 3.2, we write our own memory allocator that allows us to store intermediate maps and weights in the local 6 MB MSMC RAM, and L2 RAM. The multi-ported MSMC RAM also allows us to achieve fast data transfer of intermediate maps between DSP cores (vs. NoCs). With careful memory management and hiding off-chip DMA transfers by overlapping with compute, we are able to improve performance significantly. Clever use of the on-chip MSMC RAM and L1/L2 memories is a key ingredient of delivering high performance on this DSP.
— **Patch-Based Map Partitioning**: For the DSP, we organize the patches as blocks of image rows. Here, the advantage over the other memory-constrained platforms is the significantly greater availability of on-chip memory that allows us to use larger patch sizes, which results in better utilization of the ALU resources.
— **Improving ALU utilization**: For the compute-intensive convolution operation, we directly use DSP intrinsics via the IMGLIB library. These library calls have been optimized by TI to ensure

```
for(int l=0;l<LAYERS;l++) {
  for(int m=0;m<N[l];m++) {
    for(int n=0;n<N[l-1];n++) {
      vbx_dma(in[n][0],..)                          // copy input data row
      for(int x=0;x<ROWS;x++) {
        in1 = vbx_dma(in[n][x+1],,..)               // double buffer
        for(int r=0;r<K;r++) {
          for(int c=0;c<K;c++) {
            kern1 = vbx_dma(kernel[m][n],..)        // copy replicated kernel row
            vbx(VVHW,VMUL,prod,in1,kern1)           // multiply weight with row data
            vbx(VVW,VADD,acc,acc,prod)              // accumulate across kernel coeffs
            // wrap kernel buffers
          }
        }
        vbx_dma(out[m][x],acc,...)                  // writeupback row
      }
    }
  }
}
```

Fig. 13: FPGA/MXP Code Sketch for `conv`.

high ALU utilization for these functions. For deeper layers in AlexNet ConvNet configuration, we fall back to a GEMM-based formulation based on optimized DSPLib calls. We have to hand-write pooling routines as no equivalent APIs exist in the TI libraries. In this case, we unroll the instructions, to reduce loop overheads and achieve better arithmetic intensity.

— **Data Type**: While the C66 DSP supports single-precision floating-point operations, the fixed-point peak is 2× higher. Hence, we use fixed-point 16b `IMGLib` routines for pixel-processing without compromising accuracy of the computation. Ristretto [Gysel et al. 2016] shows that it is possible to even go lower than 16b precision with suitably-modified training routines, but we do not explore this for our work.

(b) FPGA MXP Engine (Figure 13): Our preliminary implementation on the Vectorblox MXP soft vector engine ran very slow due to unforeseen bottlenecks at various system interfaces. For instance, a naïve prototype implementation (28×28 pixels, L1=50, K1=7, L2=128, K2=5, 10 L1→L2 links) took 107 ms/frame.

— **AXI instruction dispatch latencies**: The latency of the ARM-FPGA AXI interface for instruction dispatch is relatively high, particularly when compared to the short vector lengths being processed in embedded deep learning datasets. Avoiding this bottleneck required restructuring the parallel vector operation by fusing together multiple maps and amortizing instruction dispatch cost across multiple maps at once.
— **Kernel Access**: The MXP processor fetches scalar operands in scalar-vector operations (*i.e.* pixel-kernel multiplication and accumulation filters) from the ARM CPU resulting in significant slowdowns. This required pre-assembling kernel coefficients into long vectors with repeated scalar entries on the MXP using vector copies to enable vector-vector processing. At the expense of a few extra bytes, we reduce time to 22 ms/frame when combining this optimization with the previous one.
— **Instruction reordering**: We further manually reorganize the order of `VMUL` and `VADD` operations to avoid sequential dependencies in the vector engines. DMA transfers were carefully scheduled to exploit double buffering optimization. Using these optimizations further shaved runtime down to 13 ms/frame.
— **CPU-FPGA partitioning**: While the 2D convolution and pixel processing tasks perform very well on the FPGA, the final classification stages consisting of multiple fully-connected layers are faster on the ARM than the MXP. In this scenario, we split the deep learning layers between the CPU and the FPGA while ensuring overlapped DMA transfer of the last pixel processing stages.

```
for(int l=0;l<LAYERS;l++) {
  for(int m=0;m<N[l];m++) {
    for(int n=0;n<N[l-1];n++) {
      in1 = e_dma_copy(in[n],..)           // load image map/patch from DRAM
      kern1 = e_dma_copy(kern[m][n],..)     // load weights from DRAM
      for(int x=0;x<ROWS;x++) {
        for(int y=0;y<COLS;y++) {
          float sop=0;
          // manually unrolled loops for high performance
          sop += in1[x+0][y+0]*kern1[0][0];
          sop += in1[x+0][y+1]*kern1[0][1];
          sop += in1[x+0][y+2]*kern1[0][2];
          // unroll for K*K are required..
          out[m][x][y]+=sop;
        }
      }
    }
    e_dma_copy(out[m],..)            // writeback output map
  }
}
```

Fig. 14: Epiphany Code Sketch for `conv`.

The pooling layer with stride length and kernel size other than 2 is less efficient on MXP. We run pooling on the CPU.

Despite these heroic optimizations, we observe poor performance for all ConvNets when using the MXP vector overlay (See Section 5.1). We identify redesign of the scalar-vector instruction processing as crucial to enhance the compatibility of the MXP engine for deep learning tasks. Additionally, the generic 32b ALU datapath design requires an expensive fracturing across multiple FPGA DSP blocks which could be repurposed for wider SIMD operations. We provide this result as motivation for further improvements to the FPGA design to help them achieve their compute potential.

(c) Epiphany (Figure 14): The Epiphany eCores are programmed directly in C with special APIs for handling data movement. The host ARM processor loads the binaries and orchestrates data movement and overall program flow. The overall programming effort for the Epiphany was hampered by the need to explicitly manage program and data memories, and lack of debugging support for diagnosing performance bottlenecks.

— **Map Parallel**: At a high level, we adopt a map-parallel approach, where each of the 16 Epiphany eCores shares the workload of evaluating all the maps in a given layer in parallel (*e.g.* 256 maps in a layer are split across 16 eCores).
— **On-chip NoC**: When the deep learning network is small enough (*e.g.* MNIST/CIFAR10), we can fit the entire stack on the on-chip memory and model the communication between different layers using the on-chip Epiphany NoC. The on-chip NoC has a significant power and performance advantage over DRAM transfer channels, which is one of the main reasons why we observe very good performance with small deep learning datasets.
— **Data Type**: The Epiphany supports single-precision floating point operations, and is, in fact, optimized for floating-point arithmetic. We could potentially reduce memory storage costs by using 16b fixed-point data types, but this comes at a performance cost. This is due to the `e-gcc` compiler automatically inserting expensive type-casting operations to upgrade precision to floating-point. Hence, we use floating-point types for all maps.
— **Patch-based Map Partitioning**: If intermediate map storage exceeds available on-chip memory, we use a patching approach, where only a small patch is transferred to on-chip memory. We statically determine the largest patch size that we can accommodate on-chip for a given network specification based on allocation of instructions and data on the 32 KB scratchpad/eCore.

— **DMA optimizations**: On the Epiphany, DMA read bandwidth is 3× smaller than DMA write bandwidth (wider NoC). To exploit this, we flatten our 2D patches into 1D contiguous structures and issue longer burst DMA reads instead.
— **Instruction unroll**: On the eCores, the main workhorse operations are the 2D convolutions. In order to achieve ideal performance, we parametrically unroll the multiply-add operations in the 2D convolutions to fully utilize the floating-point units. We balance the increased instruction storage costs due to unrolling with reduced space for data (thereby increased DMAs) as both data and instructions compete for space in the same scratchpad.

(d) NVIDIA GPU: For the GPU implementation, we use the optimized cuDNN library that *lowers* [Chetlur et al. 2014] convolutions into highly-parallel SIMD matrix multiplication operations. Matrix arithmetic based on BLAS routines are some of the fastest operations possible on a GPU due to abundant SIMD parallelism and register availability. These routines require matrix unrolling that creates matrices that are K_i^2 larger than the original maps ($K_i \times K_i$=kernel size for layer i). They are not directly applicable to other platforms due to limited memory capacities and DRAM throughputs. As we will discover in Section 5.6, the matrix-multiplication approach still holds value for layers with small image maps. There have been algorithmic modifications using Winograd algorithm (Nervana [Lavin 2015]), FFTs (Facebook [Vasilache et al. 2014]) or Binarization ([Rastegari et al. 2016]). Winograd is better for small filter sizes such as 3×3 while FFTs are better for larger kernels. XNOR-nets is particularly well-suited for FPGA mapping, but the prediction accuracies are still not on par with the standard approaches. While important areas for further research, algorithmic changes are beyond the scope of this paper.

5. RESULTS

In this section, we describe our experimental outcomes of mapping and optimizing ConvNet configurations on various embedded platforms. For the ARM CPU measurements, we compile our C code with the `-O3` switch. This enables NEON optimizations and vectorizes appropriate loops. This is true for vectorized loads/stores for memory transfer to the accelerators, and fast evaluation of the final fully connected layers that is retained on the host CPU for MXP (FPGA) and Epiphany (Multi-core) scenarios. For MXP acceleration, we use the vector API that is compiled directly as bare-metal C code. We use the PAPI v5.4.0 profiling library on the ARM, CUDA timers for the GPU, MXP hardware timers on the FPGA, hardware timers in the Epiphany-III SoC, and platform-specific timing measurement APIs for the DSPs. These measurements are conducted by averaging the measurements across hundreds of Caffe iterations to minimize measurement noise. We use the Energenie power meter for measuring total system power under 60s steady-state load. We also provide a quantification of the benefits of using a direct convolution-based approach over a GEMM formulation on the TI Keystone-II DSP and identify conditions under which this is valid.

5.1. Overall Performance and Energy Trends

In Figure 15, we present combined trends for performance and energy efficiency across platforms and ConvNet architectures. Across all configurations except ImageNet, the Keystone II DSP offers the best performance **and** energy efficiency. The gap over other architectures is as much as 4–5× for smaller ConvNets such as MNIST and CIFAR10. For larger configurations such as Caltech101, the GPU manages to close this gap while outperforming the DSP for ImageNet. Our DSP routines on small image maps are a bottleneck which only occur for deeper networks such as ImageNet due to repeated subsampling on the inputs. The `DSPLib` calls are optimized but unable to keep the ALUs occupied with useful work as the row sizes become too short. Under these circumstances, our library selects the unrolled `GEMM`-based formulation to create more parallel work for the ALUs. The DSP also suffers from poor pooling performance due the irregularity or pooling writebacks. The Parallella implementation is surprisingly competitive for MNIST and CIFAR10 where all intermediate maps can be stored entirely within the 32 KB scratchpads/core. However, for larger and more complex ConvNets, we are forced to offload storage to DRAM losing both performance and en-

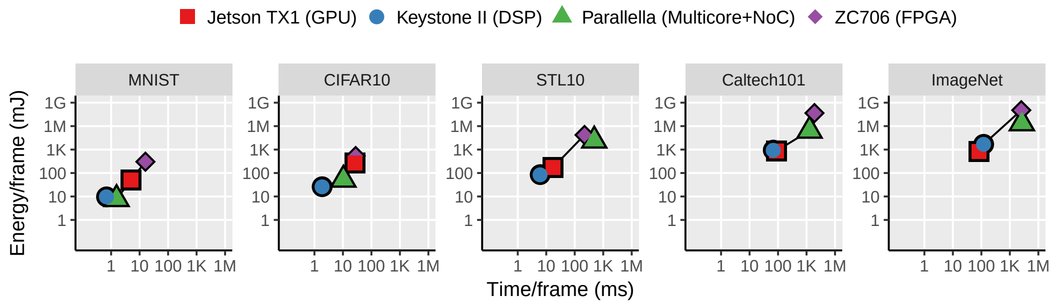
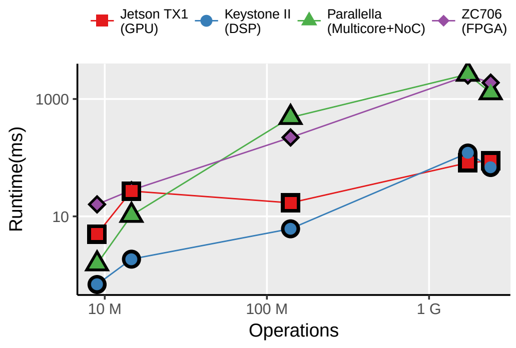
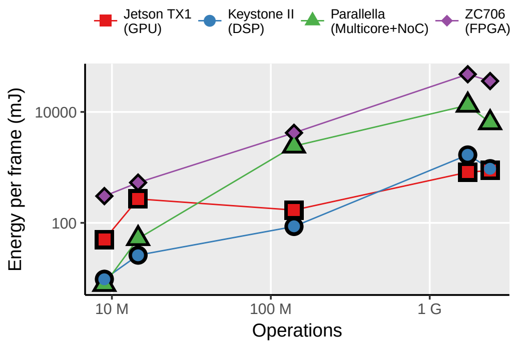


Fig. 15: Comparing Throughput and Energy Efficiency for various platforms and ConvNet configurations. Keystone II DSP dominates performance and energy efficiency in all cases. Jetson TX1 only starts catching up for larger configurations.



(a) Performance Scaling Trends.



(b) Energy Scaling Trends.

Fig. 16: Performance and Energy Scaling trends (left to right: MNIST, CIFAR10, STL10, ImageNet, Caltech101)

ergy efficiency. The FPGA platform is never particularly competitive due to limited ALU peak, and bottlenecks due to scalar-vector instruction issue design. The low efficiency of the GPU for small ConvNets is primarily due to the CUDA launch overheads, and lack of low-level control of memory transfers (CUDA-compatible GPUs do allow cache prefetch in `__shared__` 64 KB memory). In contrast, all other architectures offer complete control over memory DMAs thereby providing another precise and predictable degree of freedom for optimizations.

5.2. Impact of ConvNet Complexity

In Figure 16a, we show the variation of the runtime as a function of the total number of arithmetic operations for different ConvNet architectures. Here, the operation counts on the x-axis are taken from Table I. For a given ConvNet architecture, the total number of operations mainly depends on (1) number of feature maps in each layer, (2) the convolution kernel size and (3) image resolution. All platforms exhibit mostly linear scaling in runtime when increasing operation counts. The Parallella starts off performing better than the FPGA at low operation counts and then generally gets worse for more complex configurations. This is primarily a result of fusing map computations into long high-performance vector operations (amortized vector launch overheads), and explicit scheduling of longer DMA operations (amortized DMA startup cost) on the FPGA. The Jetson TX1 GPU starts off with high runtimes for smaller problem sizes. This indicates poor utilization of hardware resources for smaller datasets with limited control over performance of small memory transfers, and CUDA launch overheads. DSP runtimes are significantly better at low ConvNet complexities as a result of tight VLIW scheduling of ALU operations and precise control of DMA memory traffic. However, the lack of optimized pooling implementations allows the GPU to exceed DSP performance for the deep ImageNet configuration.

When considering energy efficiency in Figure 16b, we see a clearer separation of trends across the various platforms. The ZC706 FPGA platform with its 19 W power draw is clearly the least efficient of the set of systems. In contrast, the lower power 3–4 W Parallella board offers competitive energy efficiency for smaller ConvNets when DRAM transfers for intermediate maps are absent. As before, the DSP beats the GPU in energy efficiency for almost all cases with larger wins for smaller ConvNets. The GPU exceeds the DSP efficiency only for the deepest ConvNet configuration for ImageNet.

5.3. Accelerator Efficiency

In Figure 17a and Figure 17b, we visualize the hardware usage efficiency (fraction of ALU and DRAM B/W peak actually achieved) of all platforms for different configurations. The theoretical peaks are identical to the ones shown earlier in Figure 4 and Table IV. Here, we clearly see that the DSP and GPU implementations are able to utilize their respective ALUs resources ($<$10%). For the smaller ConvNets, the GPU utilization is particular poor and explains the slow runtimes. For larger ConvNets, the GPU starts to beat the DSP resulting in a better runtime than the DSP for ImageNet. For Caltech101, the higher raw ALU peak for the DSP helps it outperform the higher utilization of the GPU. The DRAM interface for the GPU was known to be faster than the DSP from the roofline plot in Figure 4. The GPU generally delivers superior DRAM utilization than the DSP across all cases, but these are still quite low $<$10% of the peak.

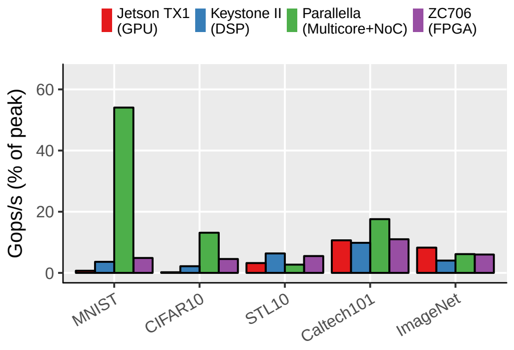
There are two odd behaviors that hold clues for how best to design clean-slate new architectures for deep learning computations. (1) The ALU performance of the Parallella is high, $\approx$55%, for MNIST. This high utilization is better than all other platforms due to the relatively small inter-layer core-to-core communication overheads and the use of unrolling to keep ALUs busy with work. However, the Parallella is generally very slow and suffers from DRAM access penalties for larger ConvNets. (2) The memory bandwidth utilization of the FPGA (ZC706) exhibits steadily increasing efficiency. This is because as all intermediate maps are explicitly transferred to/from scratchpad with DMA calls resulting in high B/W demand on a relatively poor DRAM interface. Despite this, the low ALU utilization, and redundant copying of data for the small scratchpad ultimately limits FPGA performance. Based on (1) and (2), we can observe the benefits of using a NoC to move local data within a chip and a software-exposed memory architecture (without caches). We anticipate novel clean-slate architectures for deep learning to adopt these design principles.

5.4. Runtime Breakdown

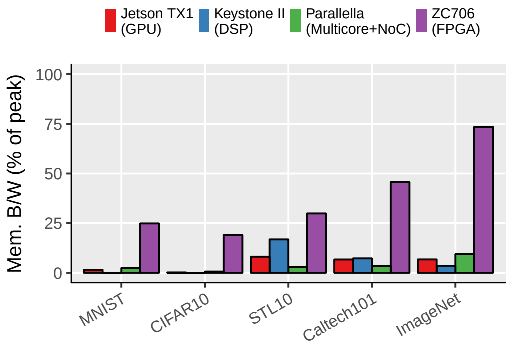
We can further understand the source of performance limits from Figure 18 (CIFAR10). The 2D convolution time dominates overall time in most cases as expected. This is particularly severe for the FPGA MXP mapping due to lack of support for fused multiply-add operations and higher penalty for scalar-vector operations. For the Keystone II DSP, a larger fraction of time is spent in the `pool` stage. This computation is not natively supported through intrinsics, leading to poorer hardware utilization. The GPU and Parallella performance breakdown match expectations with the slight advantage for `pool` on the GPU, which can be attributed to higher bandwidth to the shared RAMs.

5.5. Understanding Performance Tuning

In Figure 19, we represent the impact of patch size on performance (Gops/s) of the 2D convolution phase. In particular, this analysis is used by our optimizer to select the best patch size per platform when decomposing the implementation to exploit high on-chip bandwidths. We conduct experiments with a single layer of the network with 10 maps to illustrate the choice of best configuration.



(a) Achieved Gops/s (%) efficiency.



(b) Achieved Memory B/W (%) efficiency.

Fig. 17: Compute and Memory Bandwidth Efficiency across platforms.

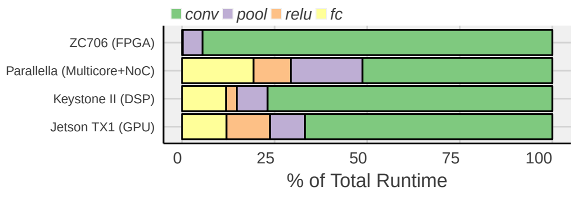
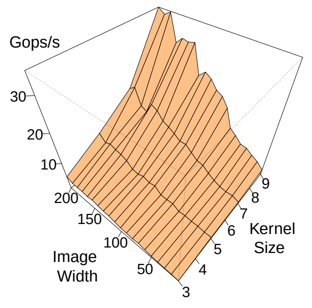
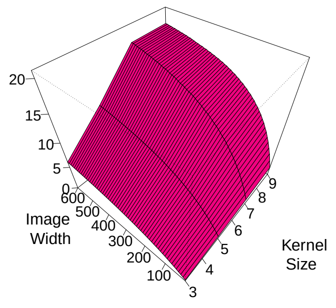


Fig. 18: Breakdown of total runtime for CIFAR10 dataset (labels in inverse order of bar colors, for MXP/FPGA the runtimes of `relu` and `fc` too small to be visible clearly).

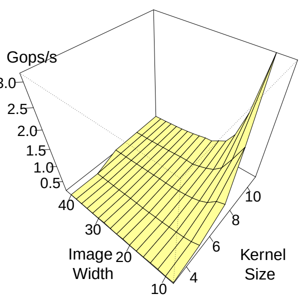
- As expected, the GPU outperforms all platforms and peaks at 35 Gops/s for larger patch sizes and kernel widths. Even when operating over smaller kernels, the throughput is 10 Gops/s.
- The Keystone II DSP mostly matches performance of the GPU with a slightly lower peak throughput of $\approx$20 Gops/s. There is also a saturation in performance for the 9×9 kernel as this is not directly available in IMGLIB and composed from a more complex 11×11 version that is available. For the smaller kernel sizes, performance drops to $\approx$5 Gops/s. While these numbers are lower than equivalent GPU measurements, overall DSP performance is a result of optimized eDMA transfers and other optimizations. The performance of the DSP for smaller image sizes $<$100 is cause for concern when considering deeper ConvNets and repeated use of subsampling to reduce image size. We observed the impact of this on ImageNet when the standard direct convolution approach does not yield the best performing solution despite the use of `DSPLib` APIs. In these cases, we have to resort to a matrix-multiplication formulation.
- Parallella shows characteristics that are starkly different from the other boards. The performance peaks for small patch size and large kernel sizes because (1) the ops/pixel is higher for larger kernels as seen in Caltech101 behavior in Figure 17a, and (2) amount of data transfer is lower for smaller patches. This clearly indicates the bottleneck in data transfer when we have low arithmetic intensity (ops/pixel are less).
- The ZC706 MXP implementation saturates at a paltry 1.5 Gops/s. We see that the MXP throughput increases with the image width before saturating. We observe negligible improvements for larger kernels. The lack of fused multiply-add, poor performance of scalar-vector operations, over-engineered 32b ALUs, small 256 KB scratchpad, and underutilization of the ZC706 resources due to Vectorblox engineering constraints are the main culprits that limit FPGA per-



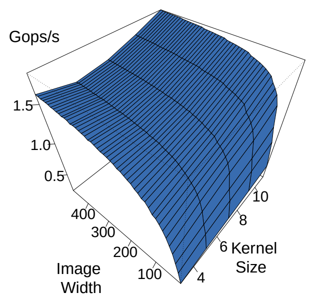
(a) Jetson TX1
(GPU)



(b) Keystone II
(DSP)



(c) Parallella
(Multi-core+NoC)



(d) ZC706
(FPGA)

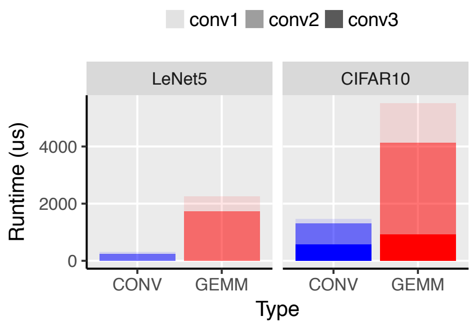
Fig. 19: Comparing Gops/s for various kernel sizes K, and patch size (or image width) W that fit on-chip for the convolution (`conv`) step.

formance. Based on our feedback, Vectorblox is working towards enhancing their compute units for better compatibility with deep learning workloads.

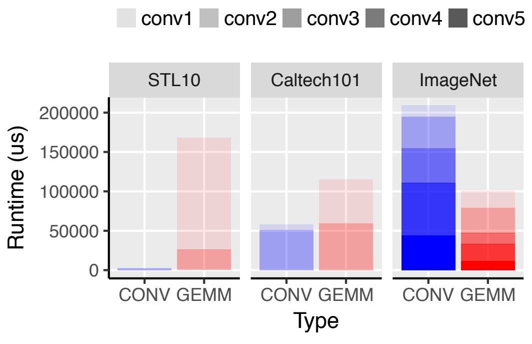
These experiments supplemented by other experiments on number of maps and DMA transfer time help us to decide the patch size for given kernel size and number of maps to consider in each iteration that optimize the overall performance. Beyond patch-size selection, our performance tuning framework also explores DMA transfer scheduling optimizations, compiler switches, and intermediate map storage strategies to fully optimize the mappings.

5.6. Direct Convolutions (CONV) vs. GEMM-based approach on Keystone II DSP

For embedded platforms with limited memory, we need to select approaches that maximize their use. As indicated earlier in Section 2, we hypothesize that a GEMM-based formulation (matrix multiplication) of the 2D convolutions would be inappropriate for these platforms due to the K^2 memory cost of unrolled inputs. We implement 16b fixed-point GEMM on the TI Keystone II DSP to better understand the performance limits of this approach. We parallelize the problem across the eight DSP cores by partitioning the rows of the unrolled matrix. Due to the size of the intermediate maps, they are stored in the off-chip DRAM. We pre-fetch blocks of the matrix B on-chip to ensure fully



(a) MNIST and CIFAR10



(b) STL10, Caltech101, and ImageNet

Fig. 20: Comparing direct convolution, or CONV, (blue) and matrix multiplication, or GEMM, (red) approaches for ConvNet implementations on the TI Keystone II DSP. Different layers of the deep network shows as a stack. The GEMM-based approach deliver 2× faster solution for ImageNet but is slower in all other cases for the smaller networks by 2–5×

on-chip memory access for the block-level matrix multiplication operations. The matrix unrolling (`im2col` function in Caffe) on the block is performed on-chip. The evaluation is double-buffered to ensure the slower memory access operations can be overlapped with the arithmetic computation across the eight cores. We also use optimized `DSPLib` fixed-point APIs to ensure high utilization of the ALUs in the DSP core.

In Figure 20, we show the runtime comparison for the different convolution layer configurations used in our study (number of maps, kernel sizes, map sizes). We observe that the direct convolution (CONV) implementation outperforms all GEMM-based configurations except the larger ImageNet dataset. For the AlexNet configuration (ImageNet dataset), after the second layer, the use of GEMM is preferred over the convolutions as the `DSPLib` APIs offer superior ALU occupancy over the `IMGLib` convolution APIs. The root cause of this disparity is the smaller sizes of the rows involved in the deeper layers of the AlexNet configuration. This exposes a limited number of concurrent ALU operations that are overwhelmed by memory access. Under this scenario, the GEMM-based formulation is able to supply sufficient arithmetic operations into the VLIW DSP core to keep it occupied with useful work. However, this is only beneficial for the ImageNet at only at the deeper layers.

We also sample the design space of possible matrix sizes, and convolution layer configuration in Figure 21 for a 3×3 kernel. The plot axes represent the configuration for a convolutional layer. Each ConvNet layer is a co-ordinate in this space. The color of each explored combination represents the fastest implementation possible for the design. The empty space in the plot without any points are design combinations that need more storage space for the input and output maps than we can fit in the 6 MB MSMC RAM. Our patching approach decomposes the maps into one of the other feasible combinations. From the plot, it is clear that the GEMM-based approach only delivers benefits when the sizes of the maps are below 20–30 pixels. This region (shows as red points in the plot) covers a narrow space to the left of the plot. Most ConvNet configurations operating on large maps will be faster when implemented with the CONV approach. However, when considering deeper layers in the AlexNet configuration for ImageNet, the maps become increasingly smaller and more abundant. This pushes their preferred convolution approach to GEMM instead of CONV. As we increase kernel size, the design space where the GEMM-based approach is preferred also grows in size.

6. CONCLUSIONS AND DISCUSSION

We develop CaffePresso, a Caffe-compatible code generation and auto tuning framework for mapping the inference phase of various embedded-friendly ConvNet configurations to accelerator-based

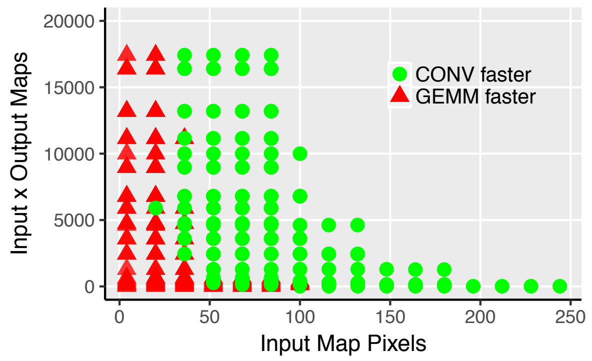


Fig. 21: Design space comparing effectiveness of the GEMM and CONV-based approaches. (3×3 kernel, map size (x axis), and number of maps (y axis) swept over a combination that fits the on-chip 6 MB MSMC RAM.)

SoC platforms. We found the TI Keystone II SoC (28nm) outperforming all other organizations including the NVIDIA TX1 GPU (20nm) for all ConvNet configurations except AlexNet. For lightweight ConvNet configurations such as MNIST and CIFAR10, the DSP and Epiphany SoCs offer competitive performance and energy efficiency. As the complexity of the ConvNet configurations increase, the GPU catches up and outperforms the DSP only for the ImageNet dataset. The DSP performed particularly well as our framework was able to exploit the high-bandwidth on-chip SRAMs effectively through auto-generated low-level DMA schedules, thereby keeping the VLIW DSP ALUs occupied with useful work (10–40% efficiency).

Overall, we found the cuDNN-based GPU flow to be effortless to use, with the TI DSP being a close second. The TI DSP suffered from a high barrier to setup the platform for first use unlike the out-of-the-box experience possible with the NVIDIA GPU. The MXP vector processor and Epiphany SoCs were hard to program primarily from the perspective of functional correctness, but were easy to optimize. We hope our CaffePresso infrastructure becomes a useful platform and model for integrating new architectures and backends into Caffe for convolutional network acceleration. The Epiphany SoC offers competitive energy efficiency in a small package for small ConvNet configurations and is ultimately constrained by limited on-chip capacity.

Looking forward, we expect *software-exposed* architectures with large on-chip scratchpads (rather than caches), VLIW-scheduled processing units, and programmer-visible networks-on-chip (NoCs) to deliver high-performance with low energy cost. Our framework is geared to take advantage of such SoCs where decisions regarding memory management, code generation, and communication scheduling are under software control, and thus available for automated optimization.

The source code for the different backends and tuning scripts are available at https://github.com/gplhegde/caffepresso.git. We envision two users of our platform:

— A user who wants to supply a new ConvNet needs to only provide the correct prototxt file that encodes the layer configuration. A one-time pre-processing step will generate optimized implementations for this ConvNet targeting the chosen hardware platform.
— A developer wishing to integrate a new embedded accelerator should supply their own implementations for each of the low-level Caffe APIs. These functions should be written in a flexible, configurable, parametric manner to allow automated design space exploration. Our tuning scripts would need to be modified to consider the new parameters specific to the new accelerator.

ACKNOWLEDGMENTS

The authors would like to thank Jeremiah Chua, Bibin Chandrashekharan, Vamsi Buddha, and Nachiappam Ramasamy for their assistance during the project in various forms.

REFERENCES

Zhaowei Cai, Mohammad J. Saberian, and Nuno Vasconcelos. 2015. Learning Complexity-Aware Cascades for Deep Pedestrian Detection. *CoRR* abs/1507.05348 (2015). http://arxiv.org/abs/1507.05348

Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. 2015. Origami: A Convolutional Network Accelerator. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI (GLSVLSI '15)*. ACM, New York, NY, USA, 199–204. DOI: http://dx.doi.org/10.1145/2742060.2743766

Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Université de Rennes 1, Suvisoft, La Baule (France). https://hal.inria.fr/inria-00112631 http://www.suvisoft.com.

Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerato for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759* (2014).

William Dally. 2015. High-Performance Hardware for Machine Learning. https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf. (2015).

Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S Modha. 2015. Backpropagation for Energy-Efficient Neuromorphic Computing. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 1117–1125. http://papers.nips.cc/paper/5862-backpropagation-for-energy-efficient-neuromorphic-computing.pdf

V. Gokhale, Jonghoon Jin, A. Dundar, B. Martini, and E. Culurciello. 2014. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*. 696–701. DOI: http://dx.doi.org/10.1109/CVPRW.2014.106

Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented Approximation of Convolutional Neural Networks. *arXiv preprint arXiv:1604.03168* (2016).

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *CoRR* abs/1602.01528 (2016). http://arxiv.org/abs/1602.01528

G. Hegde, Siddhartha, N. Ramasamy, and N. Kapre. 2016. CaffePresso: An optimized library for Deep Learning on embedded accelerator-based platforms. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. 1–10. DOI: http://dx.doi.org/10.1145/2968455.2968511

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). http://arxiv.org/abs/1509.09308

Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. 2015. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. (February 2015). http://research.microsoft.com/apps/pubs/default.aspx?id=240715

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR* abs/1603.05279 (2016). http://arxiv.org/abs/1603.05279

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. DOI: http://dx.doi.org/10.1007/s11263-015-0816-y

Aaron Severance and Guy GF Lemieux. 2013. Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*. IEEE, 1–10.

Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. *CoRR* abs/1412.7580 (2014). http://arxiv.org/abs/1412.7580

Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. 2015. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876* (2015).

Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. ACM, New York, NY, USA, 161–170. DOI: http://dx.doi.org/10.1145/2684746.2689060