CaffePresso: An Optimized Library for Deep Learning on Embedded Accelerator-based platforms

Gopalakrishna Hegde
hgashok@ntu.edu.sg

Siddhartha
siddhart005@e.ntu.edu.sg

Nachiappan Ramasamy
nachiapp001@e.ntu.edu.sg

Nachiket Kapre
nachiket@ieee.org

School of Computer Science and Engineering
Nanyang Technological University
Singapore 639798

ABSTRACT

Off-the-shelf accelerator-based embedded platforms offer a competitive energy-efficient solution for lightweight deep learning computations over CPU-based systems. Low-complexity classifiers used in power-constrained and performance-limited scenarios 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 5–20 W power budgets such as the Xilinx ZC706 board (with MXP soft vector processor), NVIDIA Jetson TX1 (GPU), TI Keystone II (DSP) as well as the Adapteva Parallella board (custom multi-core with NoC). Deep Learning computations push the capabilities of these platforms to the limit through compute-intensive evaluations of multiple 2D convolution filters per layer, and high communication requirements arising from the movement of intermediate maps across layers. We present CaffePresso, a Caffe-compatible framework for generating optimized mappings of user-supplied ConvNet specifications to target various accelerators such as FPGAs, DSPs, GPUs, RISC-multicores. We use an automated code generation and autotuning approach based on knowledge of the ConvNet requirements, as well as platform-specific constraints such as on-chip memory capacity, bandwidth and ALU potential. While one may expect the Jetson TX1 + cuDNN to deliver high performance for ConvNet configurations, (1) we observe a flipped result with slower GPU processing compared to most other systems for smaller embedded-friendly datasets such as MNIST and CIFAR10, and (2) faster and more energy efficient implementation on the older 28nm TI Keystone II DSP over the newer 20nm NVIDIA TX1 SoC in all cases.

1. INTRODUCTION

Recent advances in deep learning convolutional neural networks [11] (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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CASES '16, October 01-07 2016, Pittsburgh, PA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4482-1/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2968455.2968511

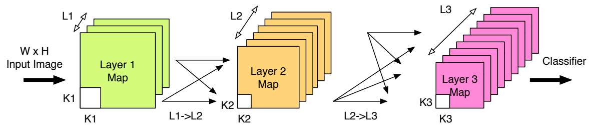


Figure 1: High-Level Overview of Deep Learning Convolutional Networks (3-layer sample network shown). Showing parameters *i.e.* number of maps L_i, 2D convolutional kernel sizes $K_i \times K_i$, fully-connected layers, and the image resolution $W \times H$.

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 [1] 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 [5, 8, 6, 2, 3], along with large FPGA-based [10, 14, 7] and GPU-based accelerators [13] 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. 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 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 [4] 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 ($\approx$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 [12] 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.

The key contributions of this paper include:

1. Development of 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. Automated code generation and performance tuning for the TI Keystone II (DSP), Parallella Epiphany (Multi-core), and Xilinx ZC706 Zynq system (FPGA). NVIDIA X1 SoC uses the already-optimized cuDNN library.
3. Quantification and analysis of performance and energy efficiency for different datasets and various optimization strategies across the embedded SoC platforms listed above.

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).

Table 1: 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	Other Layers	Ops.
MNIST	5 maps	50 maps	-	8.9 M
28×28	5×5 kern.	5×5 kern.	-	
CIFAR10	32 maps	32 maps	64 maps	14.6 M
32×32	5×5 kern.	5×5 kern.	5×5 kern.	
STL-10	32 maps	32 maps	-	139 M
96×96	5×5 kern.	5×5 kern.	-	
Caltech101	64 maps	256 maps	-	2.3 G
151×151	9×9 kern.	9×9 kern.	-	
	10×10 pool	6×6 pool	-	
	5×5 subsm.	4×4 subsam.	-	
ImageNet	96 maps	256 maps	384, 384, 256 maps	1.7 G
227×227	11×11 kern.	5×5 kern.	3×3 kern.	
(AlexNet [1])	3×3 pool	3×3 pool.	3×4 pool.	

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

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. The structure of the algorithm is organized to replicate biological behavior in an algorithmic manner that can be trained and customized for various classification requirements. 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 and average 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 $K_i \times K_i$, along with (4) pooling and subsampling factors (usually 2×2). We show the various configurations for the embedded datasets we use in this study in Table 1.

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 2. 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

Table 2: 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+NEON	GPU	Ratio
2D Convolution	`conv`	1.196	0.117	10.2
Pooling	`pool`	0.124	0.015	8.2
Rectification	`relu`	0.191	0.014	13.6

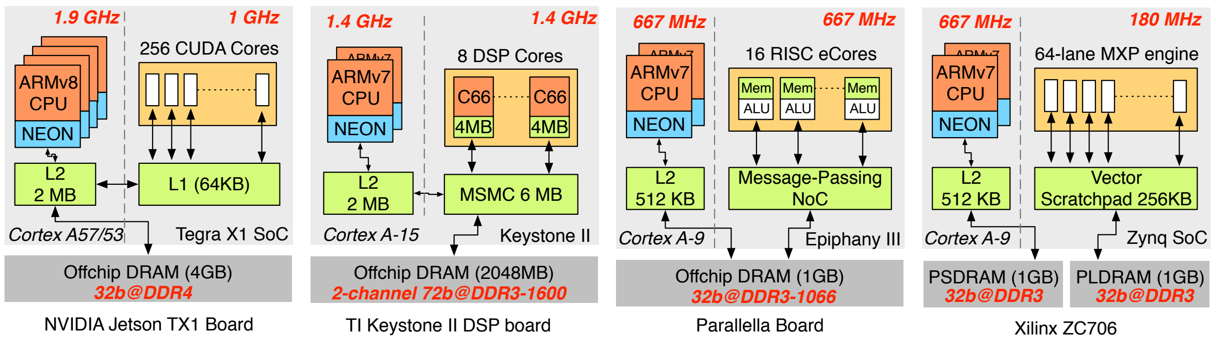


Figure 2: Comparing various Embedded Accelerator-based SoC Platforms used in this study.

Table 3: 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 16–32 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]
On-chip Host Memory	64 KB L1 + 2048 KB L2	32 KB L1 + 1 MB L2	32 KB L1 + 512 KB L2	32 KB L1 + 512 KB L2
On-chip Accelerator Memory	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-Level Power	6 W (Idle), 10 W	11 W (Idle), 14 W	3 W (Idle), 4 W	17 W (Idle), 19 W
Host Operating System	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.

quadratically. For pooling, the stride length has some impact on performance as the cache locality is affected for irregular strides. When sequencing 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 on 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.

2.2 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 2 and list the relevant specifications in Table 3. We tabulate the key throughput, power and efficiency trends in Table 4. 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 MHz × 8 C66 cores = **358.4 Gops/s**. The 66AK2H12 Keystone II board consumes 11-14 W under varying load.
- **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

Table 4: 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

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 $\frac{1}{10}$th the GPU throughput; 16×667 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 18×25 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.

We present the calculations from Table 4 as a roofline plot in Figure 3 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 3) which works well for ConvNet scenarios where the ratio between compute and memory access is 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 [13], 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.

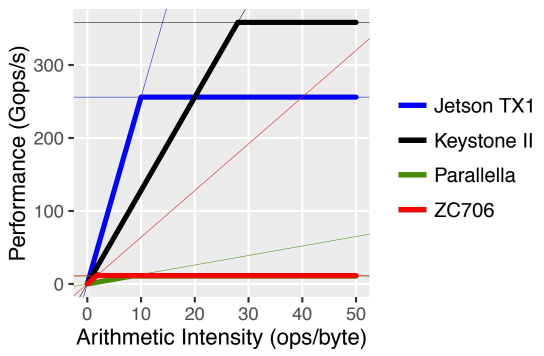


Figure 3: Roofline analysis based on Peak Gops/s (horizontal lines) and DRAM bandwidths (diagonal lines). The DSP dominated peak throughput while the GPU has greater bandwidth. FPGA and Parallella have much lower peak throughputs.

3. EXPLOITING PARALLELISM

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 [10, 14, 7], GPUs [13]) 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.

3.1 Parallelism

Fortunately, ConvNets offer parallelism at various levels of granularity that can be effectively mapped to accelerators:

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. A representative timing schedule is shown in Figure 4.

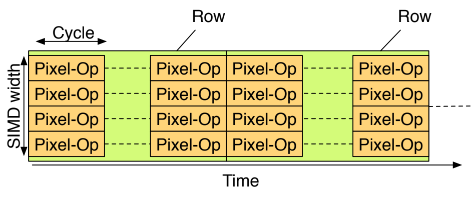


Figure 4: Pixel-level parallelism through SIMD evaluation.

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 per-

formance. A timing schedule of such an overlapped operation is shown in Figure 5 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.

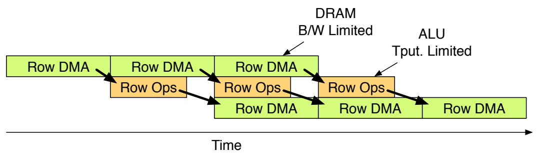


Figure 5: Row-level *streaming* parallelism through concurrent DMA/compute processing. DRAM B/W $<$ ALU Tput.

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 6, 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.

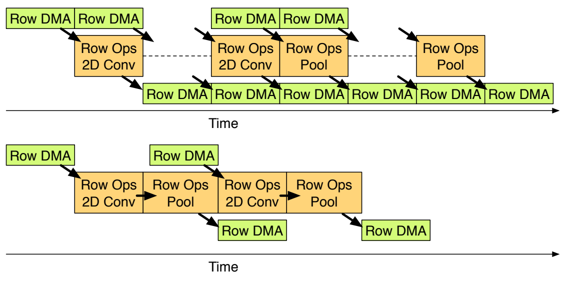


Figure 6: Dataflow optimization of dependent row-oriented functions. Interleaved operation across consecutive functions possible when dependencies are linear.

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, 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 Storage

For small-scale convolutional networks, such as MNIST and CIFAR10 (shown earlier in Table 1), we are able to fit the intermediate maps and temporary results in on-chip RAMs of most SoCs. However, larger networks, such as Caltech101 and ImageNet, have memory requirements that exceed available on-chip capacity. The Jetson TX1 manages its memory through cuDNN and Caffe. While the TI DSPs can be configured to use a cache hierarchy, the performance was so poor, we set them up as SRAMs and explicitly handled memory transfers ourselves. Here, the 6 MB MSMC RAMs are shared across all eight DSP cores. Thus, apart from the GPU, all other platforms require explicit memory management (scratchpads) with no caching or virtual memory support.

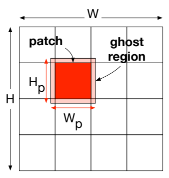


Figure 7: Patching.

Where required, we use a *patch*-based partitioning strategy, as shown in Figure 7, 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. The FPGA MXP vector processor only provides a configurable 32–256 KB scratchpad, thereby forcing low-level DMA calls 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 the Epiphany III SoC, we have a fixed allocation of 32 KB $\times$ 16 cores and can easily accommodate the small convolutional networks onchip. We are, however, 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.

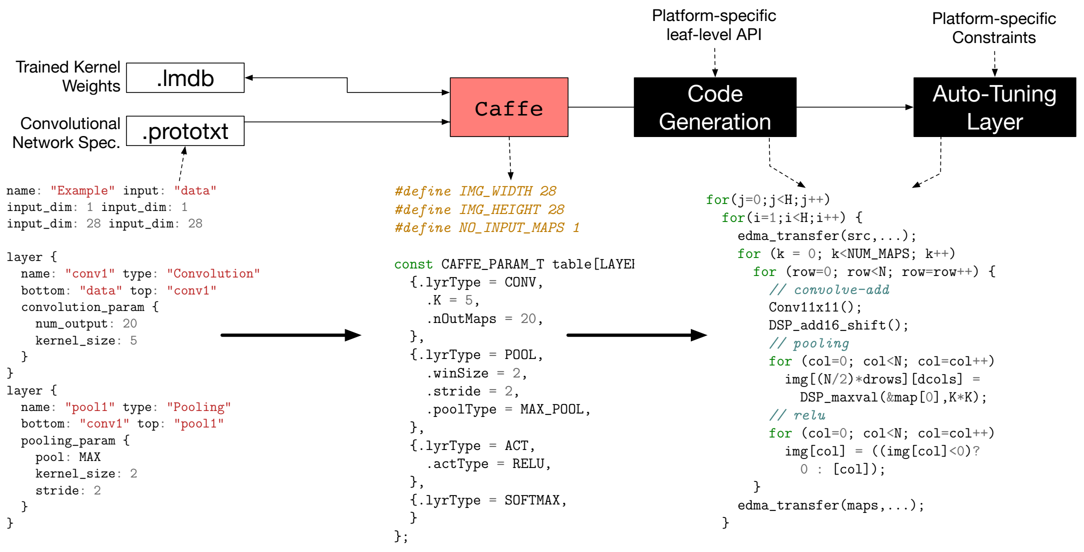
4. CAFFEPRESSO FLOW

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.

4.1 Mapping methodology

Our mapping flow, seen in Figure 8, is decomposed as follows:

- **Caffe input**: We use the open-source Caffe [9] 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. We achieve best-published accuracies (70–99%) for each dataset and associated ConvNet configurations.
- **Code-Generation**: Caffe implements various layers required for assembling a ConvNet structure on a CPU. cuDNN [4] 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


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[LAYE
  {.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,...);
  }
```

Figure 8: High-Level View of the CaffePresso flow. (Showing dummy prototxt, intermediate header, and skeleton DSP code.)

hooks to the auto-tuning flow. This translation of the ConvNet specification into platform-specific code forms the basis of our automated code-generator.

- **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.

4.2 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 micro-benchmarking to understand the ALU processing and memory constraints beyond the datasheet specifications of Figure 4.
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.

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.

```
void dsp_conv_layer(int K, int N, int scale,
  const short* kernel, short* src, short* dest) {
  // load input patch
  edma_transfer(src,...);
  // TI's imglib operates row-by-row
  for (row=0; row<H-K+1; row++) {
    IMG_conv_3x3_i16s_c16s ((src+row*N), (dest+row*N),
      N-K+1, N, kernel, scale);
  }
  // write input patch
  edma_transfer(dest,...);
}
```

Figure 9: DSP Code Sketch for `conv`.

(a) TI DSPs (Figure 9): We program the TI Keystone DSP boards using a C-based API supported by optimized image processing libraries for leaf-level routines.

- **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. We write our own memory allocator that allows us to store intermediate maps in the local 6 MB MSMC 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 onchip MSMC RAM is a key ingredient of delivering high performance on this DSP.
- **Patch-Based Map Partitioning**: This optimization is the same patching optimization on the Parallella described earlier. Here, the advantage over the Parallella is a 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 hand-written pooling routines on the DSP, we unroll the instructions, to reduce loop overheads and achieve better arithmetic intensity. Convolution operation on the patches directly use DSP intrinsics via the IMGLIB library.
- **Data Type**: While the C66 DSP supports single-precision floating-point operations, the fixed-point peak is 2× higher. Hence, we use fixed-point `IMGLib` routines for pixel-processing

```
void fpga_conv_layer(int K, int N, int NUM_MAPS,
  int16_t *src, int16_t *dest, int16_t *kernel) {
  // set vector length for all rows of NUM_MAPS
  vbx_set_vl(N * NUM_MAPS);
  // loop over rows
  for(row = 0; row < H-K+1; row++) {
    // double buffer
    vbx_dma_to_vector(src[row],src+(N-K+1)*row,.);
    // loop over kernel
    for(i = 0; i < K; i++) {
      for(j = 0; j < K; j++) {
        // Load kernel[i][j] into scratchpad
        vbx_dma_to_vector(kernel[1],kernel[i*k+j+1],.);
        // Multiply and add with pointer offset
        vbx(VVHW,VMUL,prod, kernel[0], src[i]);
        vbx(VVW, VADD, acc, acc, prod+j);
        // Wrap kernel buffers
      }
    }
    // writeback results
    vbx_dma_to_host(dest+(N-K+1)*row,acc,...);
  }
}
```

Figure 10: FPGA/MXP Code Sketch for `conv`.

without compromising accuracy of the computation.

(b) FPGA MXP Engine (Figure 10): 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 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. The pooling layer with stride length and kernel size other than 2 is less efficient on MXP. We run pooling on the CPU.

(c) Parallella (Figure 11): 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.

- **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.
- **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.

```
void epiphany_conv_layer(int K, int H, float scale,
  float *kernel, float *src, float *dest) {
  //load patch from DRAM
  e_dma_copy(src,...);
  // loop over rows
  for (row=0; row<H-K+1; row++) {
    // loop over columns
    for (col=0;col<H-K+1;col++) {
      float sop = 0.0f;
      //convolution - KxK unrolled
      sop += kernel[0] * src[ptr[0]];
      sop += kernel[1] * src[ptr[0]+1];
      sop += kernel[2] * src[ptr[0]+2];

      sop = sop*scale;
      *(dest + out_ptr) = sop;
      // update row pointers
    }
  }
  //store output map patch to DRAM
  e_dma_copy(dest,...);
}
```

Figure 11: Epiphany Code Sketch for `conv`.

- **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* [4] 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. They are not directly applicable to other platforms due to different granularities of the ALU fabric. There have been algorithmic modifications in libraries from Nervana systems that have shown to outperform cuDNN for small filter sizes such as 3×3. Algorithmic changes are beyond the scope of this paper and would affect other architectures as well.

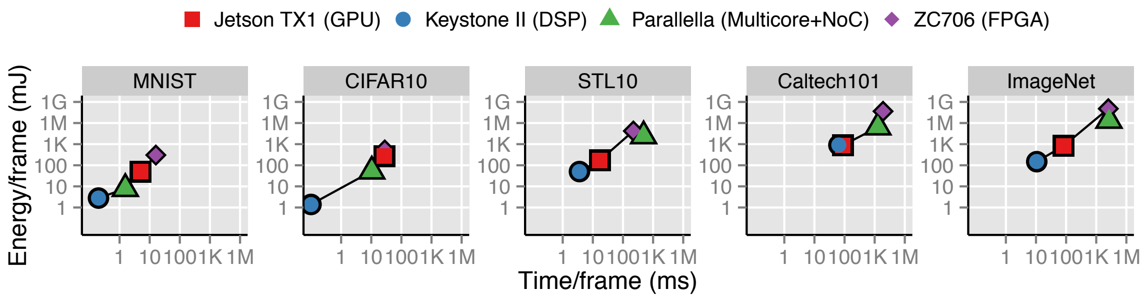


Figure 12: Comparing Throughput and Energy Efficiency for various platforms and ConvNet configurations. Keystone II DSP dominates performance and energy efficiency in all cases. Jetson TX1 only starts catching up for larger configurations.

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.

5.1 Overall Performance and Energy Trends

In Figure 12, we present combined trends for performance and energy efficiency across platforms and ConvNet architectures. Across all configurations, 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. Our DSP routines for pooling are a bottleneck and the DSP loses its lead for pooling-heavy ConvNets such as Caltech101. 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 energy efficiency. The FPGA platform is never particularly competitive due to constrained DRAM bandwidths and limited ALU peak. 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 optimization.

5.2 Impact of ConvNet Complexity

In Figure 13, 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 1. 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 match DSP performance at the largest problem size (Caltech101).

When considering energy efficiency in Figure 14, we see a

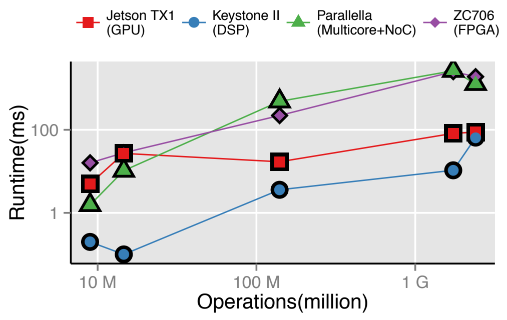


Figure 13: Performance Scaling Trends (left to right: MNIST, CIFAR10, STL10, ImageNet, Caltech101).

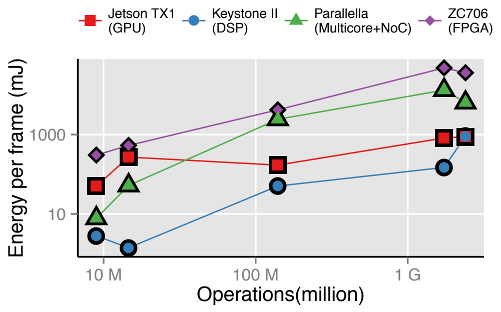


Figure 14: Energy Scaling Trends. (left to right: MNIST, CIFAR10, STL10, ImageNet, Caltech101).

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 matches the DSP efficiency only for the largest ConvNet configuration.

5.3 Accelerator Efficiency

In Figure 15 and Figure 16, 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 3 and Table 4. Here, we clearly see that the DSP implementation makes the most efficient usage of the ALU resources ($\approx$40%) for the smaller ConvNets. The use of VLIW intrinsics and careful DMA operations are key to unlocking this efficiency. The DSP DRAM efficiency is 25–35% for larger datasets where trained weights must be loaded from the DRAM. For smaller datasets, the complexity of memory access is on par with the computation intensity, which results in low utilization of Jetson TX1's GPU resources ($<$1%). For larger problems the GPU achieves 10% ALU throughput but never taxes the DRAM B/W due to optimized GEMM (BLAS matrix-multiplication) formulation in cuDNN and high peak bandwidths. There is an odd spike ($\approx$55%) in Parallella efficiency for MNIST, which can be attributed to relatively small inter-layer core-to-core communication overheads, as MNIST layer 1 only consists of 5 maps. However, the Parallella is generally very slow and does not stress the DRAM either due to compute bottlenecked operation. For the more complex ConvNet configurations such as Caltech101 and ImageNet, most platforms are saturated at 5–10% ALU efficiency. The ZC706 exhibits steadily increasing DRAM B/W efficiency as all intermediate maps are explicity transferred to/from scratchpad resulting in high B/W demand on a relatively poor DRAM interface. Overall, the large number of maps, irregular DMA traffic, and depth of the layers results in poorer ALU utilization for complex ConvNets.

5.4 Runtime Breakdown

We can further understand the source of performance limits from Figure 17 (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.

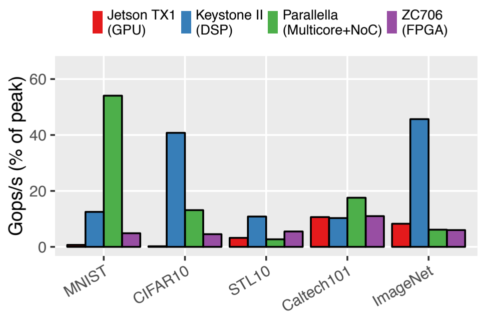


Figure 15: Achieved Gops/s (%) efficiency.

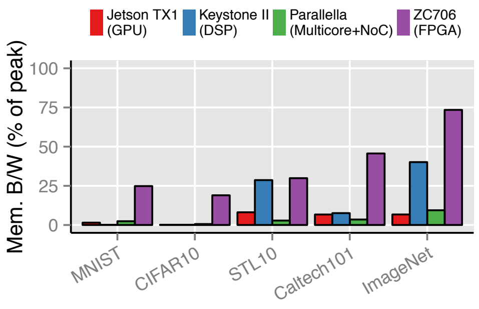


Figure 16: Achieved Memory B/W (%) efficiency.

5.5 Understanding Performance Tuning

In Figure 18, 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.

- 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.
- 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 15, 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 is the key culprit alongside kernel unrolling and loading overheads.

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.

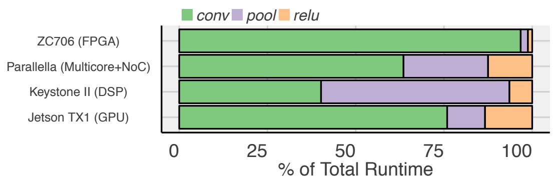
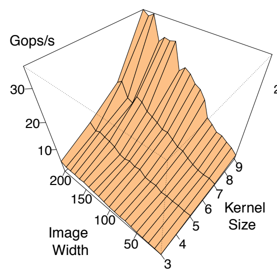
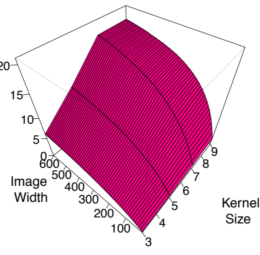


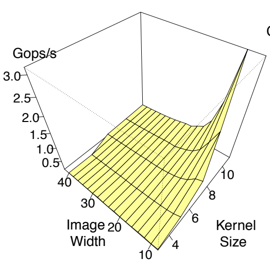
Figure 17: Breakdown of total runtime (CIFAR10 dataset).



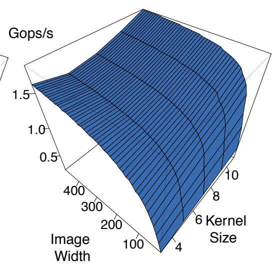
(a) Jetson TX1
(GPU)



(b) Keystone II
(DSP)



(c) Parallella
(Multi-core+NoC)



(d) ZC706
(FPGA)

Figure 18: Comparing Gops/s for various kernel size K, image width W that fit on-chip for the convolution (`conv`) step.

6. CONCLUSIONS AND DISCUSSION

We develop CaffePresso, a Caffe-compatible code generation and auto tuning framework for mapping various embedded-friendly ConvNet configurations to accelerator-based SoC platforms. We found the TI Keystone II SoC (28nm) outperforming all other organizations including the NVIDIA TX1 GPU (20nm) for all the ConvNet configurations tested in this paper. 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 but still is unable to beat the DSP. 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.

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.

7. REFERENCES

[1] Z. Cai, M. J. Saberian, and N. Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. *CoRR*, abs/1507.05348, 2015.

[2] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini. Origami: A convolutional network accelerator. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, pages 199–204, New York, NY, USA, 2015. ACM.

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

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

[5] W. Dally. High-performance hardware for machine learning. https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf.

[6] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha. Backpropagation for energy-efficient neuromorphic computing. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1117–1125. Curran Associates, Inc., 2015.

[7] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*, pages 696–701, June 2014.

[8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528, 2016.

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

[10] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[11] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

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

[13] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876*, 2015.

[14] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. 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, pages 161–170, New York, NY, USA, 2015. ACM.