Domain-Specific Accelerator Design & Profiling for Deep Learning Applications

From Circuits to Architecture

Andrew Bartolo & William Hwang
{bartolo, hwangw}@stanford.edu

Introduction

Where We've Been

The field of computer architecture is exiting an era of predictable gains, and entering an era of rapid change. For decades, the trends of Dennard scaling and Moore's law improved energy-delay product [Gonzalez96] while allowing for ever-higher numbers of transistors to be integrated on a single chip. These trends set the pace for the entire hardware industry, and ultimately drove the economics of computation. Year over year, computer users could count on hardware that was faster, and oftentimes cheaper. If an application didn't work well on existing hardware, there was a decent chance that next year's refresh would bring about a processor that was up to the task. From the 2 MHz Intel 8008 in 1974, to the 3.8 GHz Pentium 4 Prescott in 2005, CPU clock speed increased nearly 2000X over three decades. (Dennard scaling enables smaller, faster logic to fit within existing power and thermal envelopes. [Dennard74])

From an architect's point of view, device scaling provided an increasing number of transistors to play with. The architect's challenge thus became one of how to organize these transistors cleverly, so as to increase performance. In the 1980s, schemes such as superscalar and out-of-order execution gained popularity, and have remained in general-purpose architectures ever since. Superscalar issue – i.e., issuing multiple instructions at once – promotes increased utilization of a processor's functional sub-units, and out-of-order execution allows processors to hide a good deal of memory latency. The imbalance between compute and memory remains a serious problem – perhaps the most fundamentally important problem facing computer architects today.

Following the development of superscalar and OoO, techniques such as branch prediction and speculative execution became popular. As clock speeds ratcheted up, execution pipelines needed to be decomposed into more stages, so that each stage's critical path would not exceed the clock period. Architects correctly surmised that keeping the pipeline full – even with instructions that weren't guaranteed to be the "right" ones – would lead to more instructions processed per cycle, and with less energy wasted idling. Thus, branch prediction and speculative execution aimed to keep the pipeline as full as possible.

By the mid-2000s, Dennard scaling had come to a halt. However, Moore's law had granted architects such an abundance of transistors that it became possible to build two high-performance, superscalar and out-of-order cores together on a commodity chip. Intel's Pentium D shipped two such cores on a multi-chip module [MCM], and its successor, Core Duo, integrated these two cores onto a single die. By simply clone-stamping multiple cores onto one die, these CPUs did something unprecedented – they shifted the burden of extracting increased performance to the *software layer*. No longer could the average computer user buy this year's Pentium and hope for better performance – without a software rewrite for the multi-core paradigm, there was no performance increase to be had!

Like the first multicore machines, new domain-specific designs will require enhanced software and compiler support for efficient use. Designs such as GPU, FPGA, CGRA, TPU, tiled manycore, and others demand a fundamental rethinking of the software-hardware interface. Frequently, an intermediate representation such as

TensorFlow XLA is used to encode dataflow dependencies before data can actually be processed by hardware [Abadi16]. Therefore, it seems likely that tomorrow's computer architect will need to be as well-versed in software as she is in hardware.

Where We're Going

The mid-late 2010s will be remembered for its "Cambrian explosion" of new computer architectures. However, it turns out that many "new" architectures really aren't so new after all. Designs first introduced in the '70s and '80s, and that have languished since, are now poised to make a comeback. For instance, Google's Tensor Processing Unit (TPU) is, at its core, a large systolic matrix multiplier – a scheme that dates back to work done by H.T. Kung in 1982. [Kung82] NEC's new Aurora vector processor heavily resembles the vector units of the Cray-1 from 1975 [Bell78, NECAurora]. The fundamental reason for these architectures' resurgence is that general-purpose CPUS are ill-equipped to process data in parallel at scale. And, with the dawn of machine learning and massive datasets collected from cheap and abundant sensors, the demand for parallel compute resources has never been higher.

One other reason for these architectures' newfound success is their simplicity – at least, compared to modern superscalar CPUs. Flaws such as Meltdown and Spectre [Mangard18, Kocher18] prove that CPU design carries an unsustainable amount of technical debt. By moving to *simpler*, yet highly parallel, hardware execution units, the field of computer architecture accomplishes two things: 1.) it shifts a good deal of design complexity from hardware to software, which enjoys much more rapid development, and 2.) opens the playing field to a host of smaller, innovative participants.

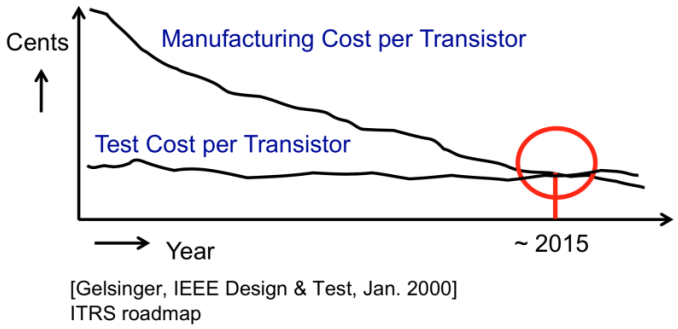

Figure 0: Trends in chip manufacturing and test costs

On one hand, it seems likely that cheap general-purpose cores will displace simpler microcontrollers in all but the lowest-cost and lowest-power devices. (Why buy an Arduino when you can have a Raspberry Pi for the same price?) *However*, in areas where performance, or performance-per-watt, is crucial, domain-specific accelerators are poised to become the architecture of choice.

For these reasons, our project focuses on domain-specific accelerators for deep learning applications.

Part I: MAC and maxpool circuit-level analyses

Our project first considers domain-specific accelerators at the circuits level. To do this, we asked the following question: if we were to unroll the dataflow graph of a contemporary neural network (say, VGG-19), how much parallelism could we extract from this graph in the absence of energy and area constraints?

Experimentally-calibrated studies of conventional manycore processor architectures (e.g., Xeon Phi) have shown that a majority of energy and execution time (greater than ~90%) is spent accessing memory across a range of abundant data applications (e.g., PageRank) [Aly15] due to limited connectivity between compute logic and off-

chip memories (generally DRAM). For particular applications, fixed function accelerators (e.g., Eyeriss [Chen16], EIE [Han16], etc.) can improve overall system energy efficiency through optimized dataflow implementations that maximize memory reuse while limiting off-chip memory accesses. Such implementations provide an isolated snapshot of the full architecture design space, and are not necessarily optimized to fully utilize all available compute and memory resources. As such, a key question remains: How does one design energy-efficient accelerators in the abundant-data era, which fully utilize all available compute and memory resources while maximizing computational throughput? Figure 1 graphically illustrates the crux of the design problem using the roofline model, where the y-axis refers to the computational throughput (in operations per second) of a given accelerator architecture, and the x-axis indicates the operational intensity of an application (in operations per byte of data accessed).

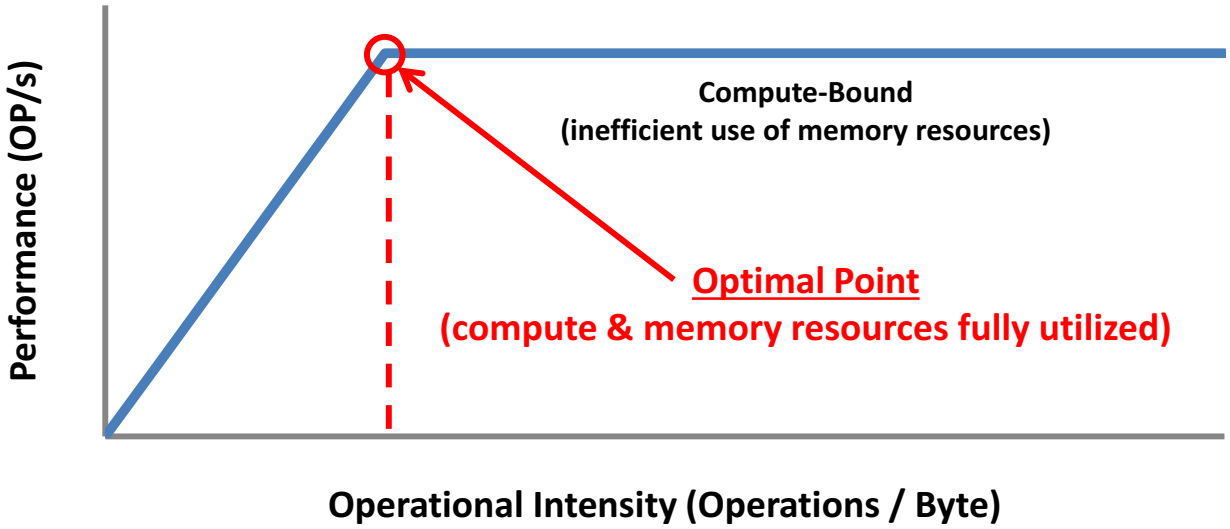

Figure 1: A graphical illustration of the design problem using the roofline model

Design space exploration

In order to understand the design space with greater depth, we constructed a simple analytical model to explore the design space. We first noted that many popular convolutional neural networks (e.g., VGG-19 [Simoyan14]) are comprised of a series of convolutional, fully connected, pooling, and ReLU (rectified nonlinearity [Krizhevsky12]) layers. In this design space exploration, we focused on the convolutional and fully connected layers for the following reasons:

1. Multiply-accumulate (MAC) operations comprise the bulk of the arithmetic operations. The underlying arithmetic kernel for convolutional and fully-connected (e.g., matrix multiply) layers is the MAC operation. The size of the MAC kernel can be parameterized in terms of the size of the convolutional or fully-connected layer as summarized in Figure 2.
2. ReLU operation implement the following function: $ReLU(x) = \max(0, x)$. In hardware, this can be implemented as a bitwise AND operation, where one input is x, and the negation of the sign bit of x is broadcasted to the other input. In this way, the output of the bitwise AND operation is x if $x \geq 0$ and 0 otherwise. Typically, every MAC operation is followed by a ReLU operation, and the energy and execution time of the MAC dominates. This assumption is later substantiated with detailed physical design studies.
3. Max pooling layers implement the following function: $maxpool(x_1, \ldots, x_n) = argmax(x_1, \ldots, x_n)$. For popular networks (e.g., AlexNet, VGG-19, ResNet-152), n is typically 4 or 9, resulting in a reduction tree of depth 2 or 4, respectively. The energy and execution time of such operation is small relative to that of the MAC operations. This assumption is later substantiated with detailed physical design studies.

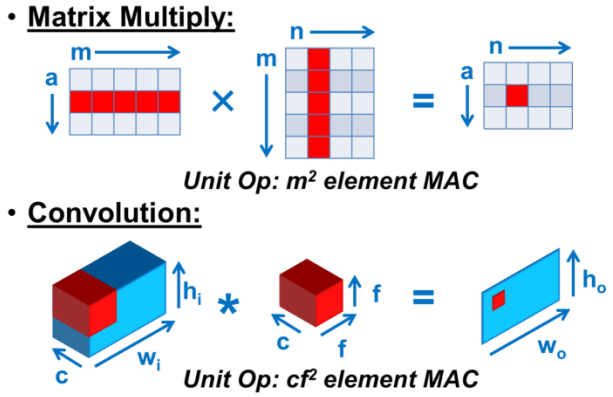

Figure 2: The size of the MAC kernels for both convolutional and fully connected (e.g., matrix multiply) layers.

We make several simplifying assumptions to model the lower bound of the energy and execution time of a convolutional neural network inference accelerator, to understand the inherent parallelism available in the inference phase of convolutional neural networks:

1. Memory access is "free" (e.g., zero energy and delay), as these parameters are not inherent to the compute logic, but rather the integration technique used to connect logic and memory components (e.g., off-chip memories, interposer-based 2.5D integration [Volta], die-stacked 3D-TSV integration [HMC], monolithic 3D [Aly15], etc.).
2. Compute logic consists of multipliers and adders only, due to the reasons stated above.
3. One hardware MAC unit is instantiated per MAC operation in the neural network.
4. All operations are 8-bit fixed point operations, consistent with [Jouppi17].

To estimate the minimum execution time and energy of the accelerator, we first performed detailed physical design studies of the multiplier and adder circuits. Following this, we analytically express the minimum system energy and delay using the MAC energy and delay derived from the circuit-level simulations, ignoring routing overheads.

Reduction-Tree MAC:

We explore the reduction-tree based MAC as our first MAC topology. Such an implementation represents the maximum parallelization that can be achieved at the circuit level, at the cost of increased chip area. A reduction-tree based MAC of size n is comprised of n multipliers and a reduction adder of depth $\log_2 n$ with $(n-1)$ adders, as shown in Figure 3.

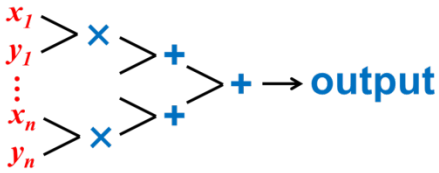

Figure 3: Dataflow of a reduction-tree based MAC.

The energy, execution time, and area of such a MAC topology can be modeled with the following equations, where t, E^{dyn}, and P^{leak}, refer to the delay, dynamic energy, and leakage power, respectively:

1. $delay = t_{mult} + (\log_2 n) \times t_{add}$
2. $add\ energy = (n-1) \times (E_{add}^{dyn} + delay \times P_{add}^{leak})$, where the *add* subscript denotes the delay, dynamic energy, and leakage power of a single adder. $add\ energy$ refers to the total energy of the reduction adder.

3. $multiplier\ energy = n\times(E_{mult}^{dyn} + delay\times P_{mult}^{leak})$
4. $area = area_{mult}\times\#_{mult} + area_{add}\times\#_{add}$, where the add subscript denotes the area and number of a single adder.

We performed detailed physical design studies using an industry 28nm process development kit (PDK) to obtain post-layout values for t, E^{dyn}, and P^{leak}. We used Synopsys tools to perform synthesis (Design Compiler), place & route (IC Compiler), and power simulation (PrimeTime). More specifically, we performed RC extraction on the circuits post place & route prior to power simulation, such that the delay, energy, and power numbers we obtained account for effects present in realistic VLSI circuits (e.g., wire parasitics, etc.). The results are summarized in Figures 4 and 5 below. The Pareto frontier is obtained by specifying timing constraints during synthesis, place and route, and power simulation, allowing Design Compiler to optimize the circuit topology (e.g., ripple-carry vs. carry-look-ahead adders) to meet the specified timing constraint. We select the energy-delay-product (EDP) minimum point on the Pareto frontier as the optimal circuit implementation, trading off energy and delay with equal weight.

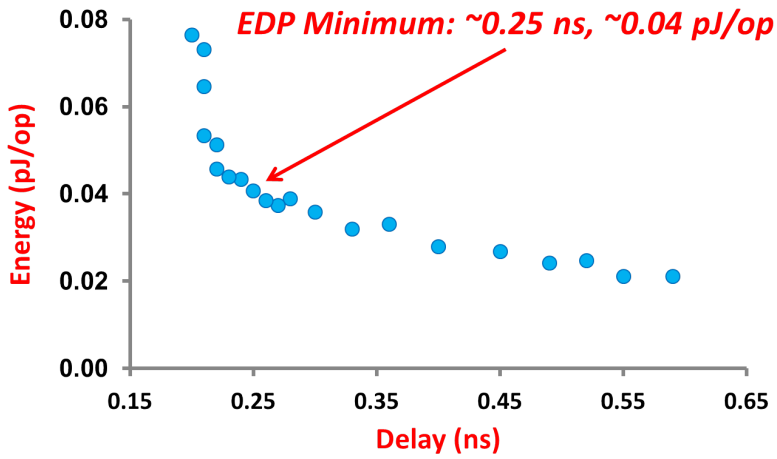

Figure 4: The Pareto frontier of the energy-delay landscape for a 28nm 8-bit adder, derived from post-layout power simulations after RC extraction at the circuit level with an industry PDK.

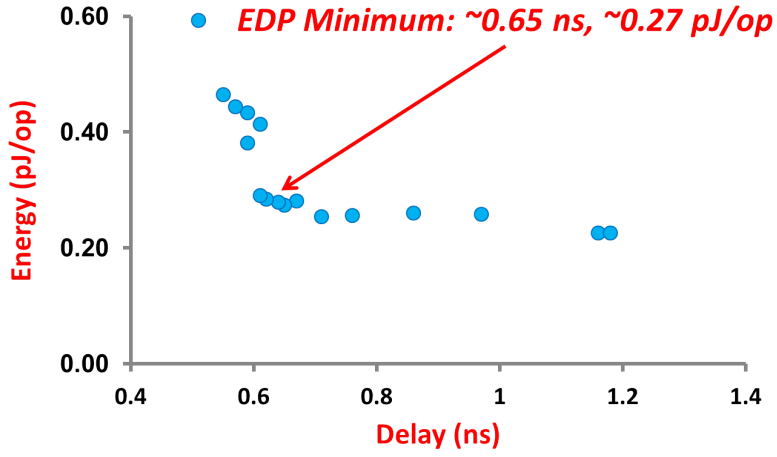

Figure 5: The Pareto frontier of the energy-delay landscape for a 28nm 8-bit multiplier, derived from post-layout power simulations after RC extraction at the circuit level with an industry PDK.

Using the results from our circuit-level simulations, we can model the convolutional neural network inference accelerator as a fully combinational circuit and estimate the energy and execution time of the

accelerator. We note that this approach does not take routing overheads into account, though these overheads can be ignored for now, as we are trying to understand the bounds of the problem. The total system-level energy and execution time can be estimated based on the number of MAC units per layer, and the number of multipliers per MAC. We show VGG-19 [Simoyan14] as an example application in Table 1 below.

Table 1: Estimated minimum energy and execution time of VGG-19 using the model discussed above.

# MAC Elements	Delay (ns)	Multiplier Energy (pJ)	Adder Energy (pJ)	Total Energy (pJ)
27	1.84	3.53E+07	5.70E+06	4.10E+07
576	2.94	7.54E+08	1.26E+08	8.80E+08
576	2.94	3.77E+08	6.30E+07	4.40E+08
1152	3.19	7.54E+08	1.26E+08	8.80E+08
1152	3.19	3.77E+08	6.31E+07	4.40E+08
2304	3.44	7.54E+08	1.26E+08	8.80E+08
2304	3.44	7.54E+08	1.26E+08	8.80E+08
2304	3.44	7.54E+08	1.26E+08	8.80E+08
2304	3.44	3.77E+08	6.31E+07	4.40E+08
4608	3.69	7.54E+08	1.26E+08	8.80E+08
4608	3.69	7.54E+08	1.26E+08	8.80E+08
4608	3.69	7.54E+08	1.26E+08	8.80E+08
4608	3.69	1.88E+08	3.16E+07	2.20E+08
4608	3.69	1.88E+08	3.16E+07	2.20E+08
4608	3.69	1.88E+08	3.16E+07	2.20E+08
4608	3.69	1.88E+08	3.16E+07	2.20E+08
25088	4.30	4.19E+07	7.02E+06	4.89E+07
4096	3.65	6.84E+06	1.15E+06	7.98E+06
4096	3.65	1.67E+06	2.80E+05	1.95E+06
	65.3 ns			**9.34 mJ**

It is important to note that these large EDP benefits come at a large area cost. We estimate the area of a single multiplier and adder to be approximately 270 μm^2 and 50 μm^2, respectively, based on post-place-and-route area estimates. To achieve the level of parallelism described here using the reduction-tree based MAC topology, the area cost is on the order of 6.28 $meters^2$, or approximately 90 industry-standard 300mm silicon wafers! Such an area cost is astronomical and untenable in real-life systems. Thus, we explore the sequential MAC topology, which is more area efficient compared to the reduction-tree based MAC.

Sequential MAC:

The sequential MAC topology instantiates a single adder and multiplier per MAC, and time multiplexes the input to multiply and accumulate the input argument over multiple clock cycles. The sequential MAC is typically more area efficient than the reduction-tree based MAC, as each sequential MAC is comprised of a single adder and multiplier (vs. $n-1$ and n in a reduction-tree based adder, respectively) and flip-flops on the inputs and outputs (Figure 6).

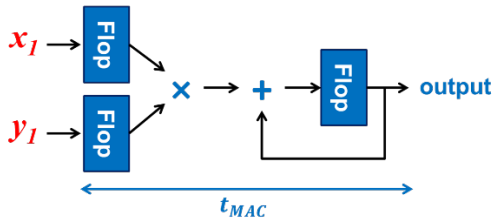

Figure 6: Dataflow of a sequential MAC unit.

The energy and execution time of a sequential MAC can be modeled with the following equations, where t_{MAC}, E_{MAC}^{dyn}, and P_{MAC}^{leak}, refer to the delay, dynamic energy, and leakage power of the sequential MAC, respectively:

1. $delay = n \times t_{MAC}$, where t_{MAC} is defined as the time it takes to propagate a signal from the input of the multiplier to the output of the output flip-flop (e.g., one clock cycle).
2. $energy = n \times (E_{MAC}^{dyn} + delay \times P_{MAC}^{leak})$

We performed detailed physical design studies of the sequential MAC unit using an industry 28nm PDK, similar to that described previously. For brevity, we refer the reader to the description in the reduction-tree MAC section. We select the EDP minimum point on the Pareto frontier as the optimal circuit implementation (Figure 7).

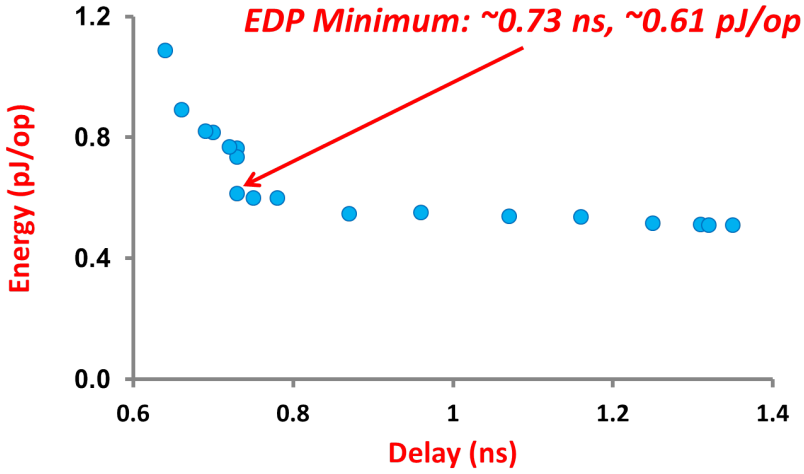

Figure 7: The Pareto frontier of the energy-delay landscape for a 28nm 8-bit sequential MAC unit, derived from post-layout power simulations after RC extraction at the circuit level with an industry PDK.

We then modeled the energy and execution time of an accelerator based on the sequential MAC unit similar to the analysis in the previous section. The total system-level energy and execution time can be estimated based on the number of MAC units per layer, and the number of multipliers per MAC. We show the results for VGG-19 [Simoyan14] as an example in Table 2 below. We estimate the area of the sequential MAC to be approximately 444 μm^2 based on post-place-and-route results, which would correspond to a system-level area cost of approximately of 6,600 mm^2, or approximately 0.1 industry-standard 300mm silicon wafers.

Table 2: Estimated minimum energy and execution time of VGG-19 using the sequential MAC.

# MAC Elements	Delay (ns)	Total Energy (pJ)
27	19.71	2.78E+10
576	420.48	5.94E+11
576	420.48	2.97E+11
1152	840.96	5.94E+11
1152	840.96	2.97E+11
2304	1681.92	5.94E+11
2304	1681.92	5.94E+11
2304	1681.92	5.94E+11
2304	1681.92	2.97E+11
4608	3363.84	5.94E+11
4608	3363.84	5.94E+11
4608	3363.84	5.94E+11
4608	3363.84	1.48E+11
4608	3363.84	1.48E+11
4608	3363.84	1.48E+11
4608	3363.84	1.48E+11
25088	18314.24	3.30E+10
4096	2990.08	5.39E+09
4096	2990.08	1.32E+09
	57.1 µs	**6.3 J**

Takeaways and Revisiting Assumptions:

Our design space exploration has led us to the following conclusions. First, there is a lot of parallelization left on the table. That is, neural network inference applications (e.g., VGG-19) have a significant amount of inherent parallelism. Without area constraints, the minimum bound for energy and execution time is very low when an accelerator is constructed with a fully-combinational network of reduction-tree based MAC units. However, it is important to note that area is an important constraint. A fully-parallelized network with the reduction-tree based MAC unit produces an equivalent area of ~90 industry-standard 300 mm silicon wafers, which is simply unobtainable using today's technology. A sequential-MAC, which time multiplexes the inputs and reuses the same multiplier and adder over multiple clock cycles, achieves a significantly lower area compared to the reduction-tree based MAC (~890× lower). However, this area reduction comes at the cost of ~874× higher execution time, and ~675× higher energy. The higher execution time originates from the time multiplexing, and the increased energy consumption is a result of the leakage power integrated over an ~874× longer execution time. Thus, we can see that it is important to understand the cost of scheduling neural network applications onto a limited number of MAC units, as the naïve, fully-combinational implementation (which is easier to schedule) is untenable in terms of area cost.

We revisit an earlier assumption where we assumed that the maxpool layers were an insignificant contributor to the energy and execution time of a deep learning inference accelerator. We perform a physical design study of the maxpool kernel from VGG-19, where the maxpool kernel has a 2x2 window size with stride 2. More specifically, this is a 4-to-1 maxpool function that computes the argmax over 4 arguments. We perform the physical design study using the same tool flow as described in previous sections. The results are shown in Figure 8.

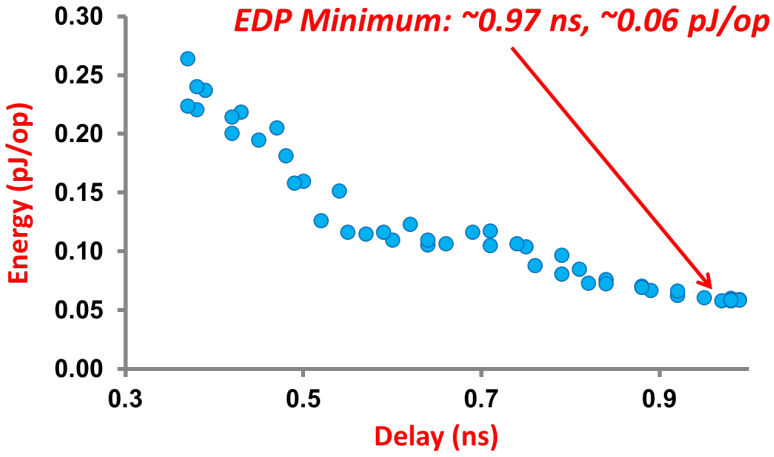

Figure 8: The Pareto frontier of the energy-delay landscape for a 28nm 8-bit maxpool unit with a window size of 2x2, derived from post-layout power simulations after RC extraction at the circuit level with an industry PDK.

At the network-level, we can estimate the effect of the maxpool layers with the following equations:

1. $delay = \#_{maxpool\ layers} \times t_{maxpool}$, where $\#_{maxpool\ layers}$ refers to the number of maxpool layers, and $t_{maxpool}$ refers to the EDP optimal delay through a single maxpool kernel.
2. $energy = \#_{maxpool\ operations} \times (E^{dyn}_{maxpool} + delay \times P^{leak}_{maxpool})$put

For VGG-19, the maxpool delay, energy, and area correspond to ~5 ns, ~25 µJ, and 80 mm^2, respectively, for the sequential MAC topology, which amount to less than ~0.1% of the sequential MAC delay and energy, and less than ~1% of the sequential MAC area. For this reason, our initial assumption that the maxpool layers were insignificant contributors was correct. We also assumed that the ReLU activation function was insignificant. Earlier, we asserted that the ReLU activation immediately follows a MAC operation and can be constructed in hardware with an 8-bit bitwise AND gate an inverter. This translates to 9 logic gates. We inspect the post-place-and-route netlist

of the sequential MAC unit (which has less gates than the reduction-tree based MAC), and note that there are ~400 gates. As such, the activation function accounts for less than 0.25% of the logic gates, and would not contribute significantly energy, as energy roughly correlates to the number of gates in a combinational circuit. The delay would not increase significantly as the ReLU activation would add ~1 gate delay to the critical path, which is not very significant considering the serial nature of the sequential MAC topology.

Part II: Top-down Deep Learning simulation

In search of a Deep Learning Accelerator simulator

Modern computer architecture is undeniably complex. Furthermore, the tremendous time and financial costs associated with taping out a chip dictate that most design space exploration must be done within simulations. Within the architecture research community, it is a commonly-held belief that, due to the often-unpredictable interactions between system subparts, analytical models by themselves are insufficient [Miller10, Sanchez13]. While some opinions differ [Nowatzki14], what is generally required are simulators that directly model the underlying components of the system, and the interactions that can occur between them.

Cycle-level simulators such as GEM5 [Binkert14] suffer from two fundamental problems. The first is that they are prohibitively slow. These simulators do have a place – they can model individual functional units – but are so sluggish as to be almost useless in modeling chip-level behavior of entire applications (rather than just traces). The second is that they support only CPU-like architectures, and extending the base architectures involves writing and validating substantial amounts of SystemC or C++.

We need a simulator that, like zsim, takes in two things: 1.) a *high-level* accelerator configuration, and 2.) an off-the-shelf prepackaged application. For zsim, 1.) takes the form of a simple .config file, and 2.) takes the form of a standard Linux ELF binary. For our deep learning inference simulator, 1.) should also be a .config file, and 2.) should be a high-level input representation such as a pre-trained TensorFlow or CAFFE model.

In light of these requirements, we successfully brought up Nvidia's new NVDLA deep learning inference accelerator and built a lightweight simulator, DLISim, top of it. Our simulator runs entirely in software (though opportunities for hardware acceleration exist) and spits out rich architecture-level statistics, such as: "How much data was used across all convolution (matmul) stages?", "How many operations of each type (matrix multiply, activation function, pooling, etc.) occurred?", and "How were large operations subdivided and scheduled on the hardware?" NVDLA + DLISim is, to the best of our knowledge, the first such open-source simulator to accomplish the goals outlined in the previous paragraph.

NVDLA – what it gives us

The NVDLA project [NVDLA] is comprised of three distinct, complementary code repositories. The first of these is **hw**, the hardware repository. It contains Verilog RTL and a SystemC golden model, as well as testbenches. (The provided Verilog passed its testbenches when we ran them in VCS.) The second is **sw**. It contains the two drivers necessary for NVDLA to interact with its host system: a user-mode driver, *UMD*, for tasks such as .jpeg image decompression, and a kernel-mode driver, *KMD*, responsible for setting control registers and transferring data. In addition, **sw** contains the (as of now) black-box binary `nvdla_compiler`. Finally, **vp** provides an ARM QEmu execution environment for the **sw** drivers to run in. The overall system layout is below.

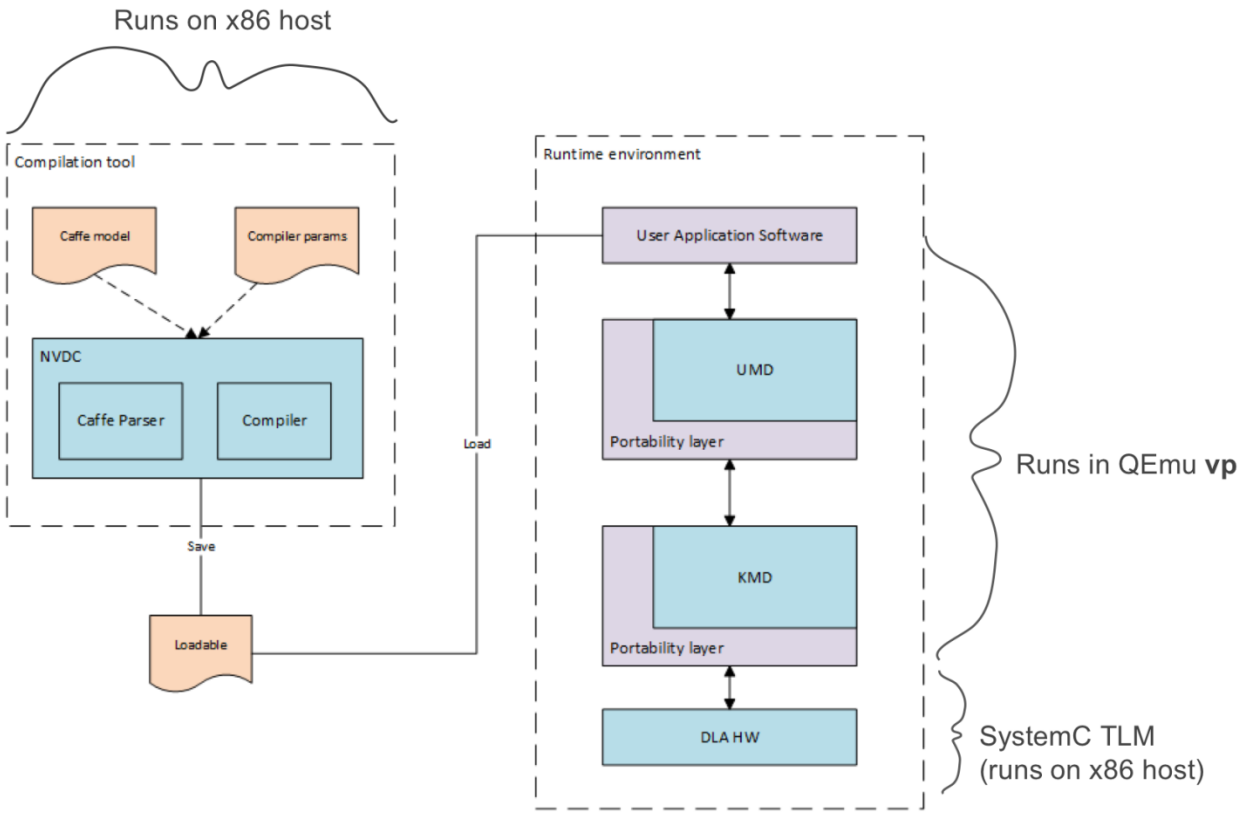

Figure 9: NVDLA platform overview

This gives us quite a bit to work with. It should be noted, however, that the entire platform is very much in an alpha-release stage, and that source code for some components is simply not available yet. For instance, Nvidia's open-source roadmap indicates that source code for the model compiler (NVDC, i.e., `nvdla_compiler`) will be available eventually, though no exact date is given – "The compiler will initially be only released in binary form, with a source release coming later" [Roadmap]. This presented a problem when several popular DL models failed to compile – we had no means of fixing the compiler ourselves. Nonetheless, Nvidia has, as of now (mid-March 2018), stuck to the release schedule at http://nvdla.org/roadmap.html, and has completed 2 of the 3 proposed stages (2017-Q3 and 2017-Q4, with 2018-H1 forthcoming). The **hw** and **sw** components were first made available to the public in October 2017, and the **vp** component in December 2017.

Like any processor, NVDLA contains different types of functional units. The following table details the six types of functional units present in NVDLA, their acronyms/abbreviations, and the specific operation within a neural net that each supports.

Table 3: NVDLA functional units

Acronym/Abbr.	Functional Unit	Deep learning functionality
CONV	Convolutional MAC	Matrix-matrix multiplication
SDP	Scalar Data Processor	Applies activation functions (sigmoid, tanh)
PDP	Planar Data Processor	Applies pooling functions (max, min, average)
CDP	Channel Data Processor	Applies normalization (centers mean, variance)
Rubik	Tensor reshape	Splits/slices/merges data to fit into compute units
BDMA	Bridge Direct Memory Access	Transfers data from DRAM to SRAM cache (not typically used)

The diagram below shows the layout of the individual functional units (lefthand side), as well as how the NVDLA core connects to the SRAM buffer ("Second DBB interface,") DRAM ("DBBIF,") and CPU ("IRQ" for a 1-bit synchronous interrupt signal, and CSB for 32-bit synchronous control signaling). Image and weight data is DMA'ed over the DBBIF interface. [Primer]

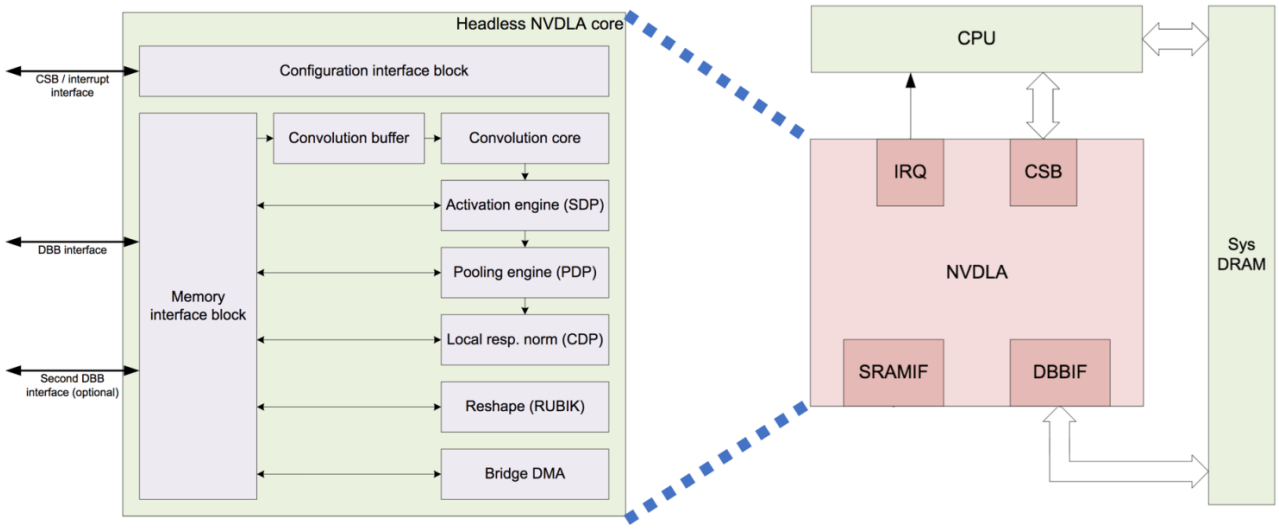

Figure 10: NVDLA core overview

The Convolution Buffer can be thought of as akin to an L1(-D) cache in a traditional CPU, and the SRAM buffer ("Second DBB interface") as an L2 cache. Finally, the DBB interface provides DMA access to DRAM.

Note that the first four functional units are arranged so that, dataflow scheduling permitting, data can move directly from one stage to another, and not have to make a round-trip to SRAM/DRAM and back. In the NVDLA literature, this is referred to as "fused" mode, in contrast to "independent" mode. Physically, fused-mode pipelining is made possible with a set of small FIFOs between each of the functional units [Primer]. However, in our experiments, the NVDLA KMD scheduler never pipelined any operations, even when it could have (a "`fused_parent = [ dla_consumer =>`" annotation should have appeared in the logs, but never did). In the neural networks we tested, activation functions nearly always followed convolutions, etc., so it seems that this early implementation of the KMD scheduler leaves a lot of performance on the table.

Regarding the functional units, **CONV** is really a general-purpose matrix multiply unit. In practice, it's often used to perform convolutions, but also supports more general matrix-matrix multiplication for fully-connected layers. **SDP** currently supports sigmoid and tanh activations, as well as P/ReLU, but, since it's implemented as a simple LUT, could be modified to handle other types of activations by simply changing the LUT ROM. **BDMA** is capable of transferring memory directly from the "Primary" DRAM interface to the "secondary" SRAM cache. None of the four networks we looked at made use of the **BDMA** FU, and it's possible that `nvdla_compiler` doesn't support it yet anyway. Likewise, no network we tested made use of the **Rubik** reshape engine.

DLISim – what NVDLA lacks

Despite the amount of code available in the three repos, NVDLA's pre-compiled drivers grant very little visibility into the runtime workings of the system. After feeding in an NVDLA loadable and input .jpeg image, the provided `nvdla_runtime` binary simply indicates whether the network ran to completion or encountered an error. However, since the driver is open-source, we were able to instrument it to provide a lot of additional information.

To get useful profiling data, we modified the KMD driver to enable a series of op-triggered printouts. Each time one of the six operations runs on the SystemC model, a message is passed to the kernel via *dmesg*. Each of these messages contains information such as the number of input bytes processed, data precision, and stride length through the input array(s). Over the course of execution, these messages collect in the kernel's ringbuffer, and can be dumped as a log file with timestamps upon completion. (For a driver, *dmesg* logging is considered cleaner than printing to stdout/stderr.)

After modifying the source code to enable the printouts, we needed to recompile the driver. The KMD code compiles into a kernel module, which must be loaded in the QEmu emulator's BusyBox Linux environment via `insmod` before simulation can begin. Unfortunately, since the QEmu simulator runs AArch64, the modified KMD module must be cross-compiled. Furthermore, to cross-compile a kernel module, the kernel itself must be cross-compiled first. Fortunately, the Linaro toolchain [Linaro] was a big help in getting both the kernel compiled and modified KMD running on AArch64.

In addition to the KMD source code modifications, we also created some Python scripts to parse the logs output by KMD, and boil the statistics down into a more digestible format. The scripts can be found at https://github.com/andrewbartolo/dlisim.

See below for a schematic of our modified NVDLA setup, with DLISIM attached.

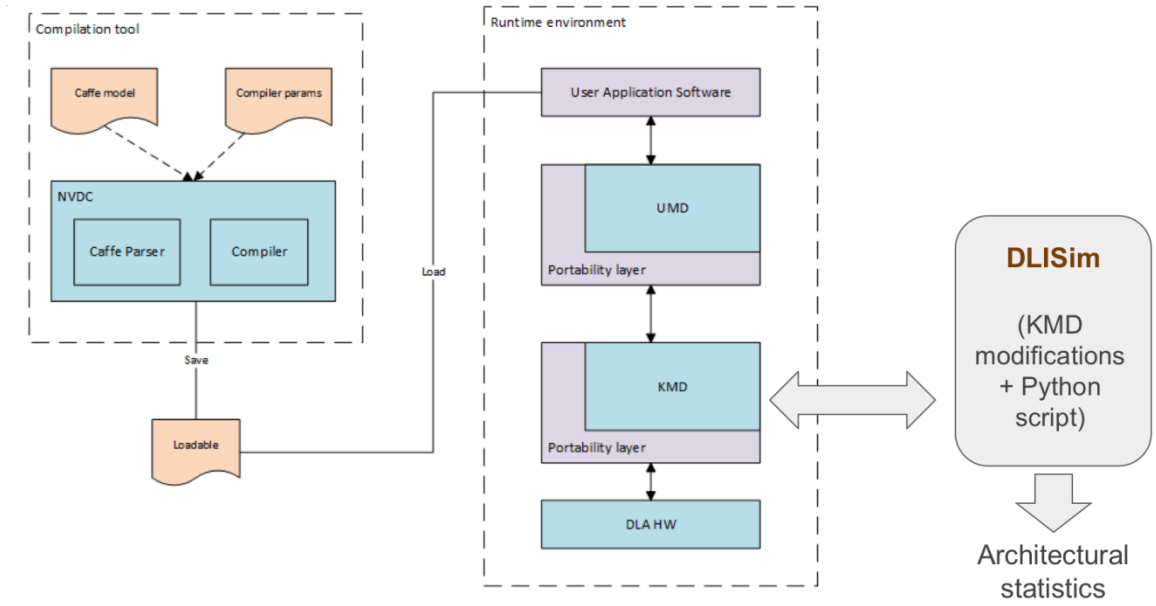

Figure 11: NVLDA overview with DLISim attached

To get a clear sense of the simulator and architecture's capabilities, we tested 13 networks from CAFFE's ModelZoo [ModelZoo], which are shown in the table below. Unfortunately, many of them didn't work with the given `nvdla_compiler`. Since Nvidia hasn't open-sourced the compiler yet, we were unable to do much to remedy this issue. What little insight we have into `nvdla_compiler` is granted by the fact that the provided binary is not stripped. After running GDB, we at least have a stack trace of what caused the error upon the segmentation fault. Ongoing discussion on the nvdla/sw GitHub issues tracker (https://github.com/nvdla/sw/issues) indicates that the NVDLA development team is aware of the issues, and is working to fix them.

Table 4: Neural nets and NVDLA compiler status

Network	Input Dataset	Status	Notes
AlexNet	ILSVRC2012	**WORKS**	
CONV + SDP	n/a	**WORKS**	
R-CNN	ILSVRC2012	**WORKS**	
GoogLeNet	ILSVRC2014	**FAIL**	Compiler could not resolve dependency in dataflow graph (CONV layer)
Hybrid CNN	ILSVRC2012	**FAIL**	Compiler segmentation fault (in parseCaffeNetwork())
LeNet	MNIST	**WORKS**	
MobileNet v1	ILSVRC2012	**FAIL**	Compiler .prototxt parser error (Pooling layer)
MobileNet v2	ILSVRC2012	**FAIL**	Compiler .prototxt parser error (Pooling layer)
Network-in-Network	ILSVRC2012	**FAIL**	Compiler segmentation fault (in parseCaffeNetwork())
SqueezeNet v1.0	ILSVRC2012	**FAIL**	Compiler .prototxt parser error (Pooling layer)
SqueezeNet v1.1	ILSVRC2012	**FAIL**	Compiler .prototxt parser error (Pooling layer)
VGG-16	ILSVRC2014	**FAIL**	Compiler segmentation fault (in parseCaffeNetwork())
VGG-19	ILSVRC2014	**FAIL**	Compiler segmentation fault (in parseCaffeNetwork())

Note that all of these are convolutional neural networks (CNNs). Unfortunately, there aren't as many CAFFE models available for RNNs, such as LSTMs and GRUs. In the future, a TensorFlow frontend may facilitate running these networks on NVDLA. Google's TPU supports recurrent neural networks (in fact, Google claims that 90% of its deployed TPUs are running MLPs or LSTM RNNs, rather than CNNs [Jouppi17]). Since NVDLA is very similar to the TPU, it should be possible to add support for these networks. Also note that all of these are pre-trained networks, with trained weights provided in .caffemodel format. (In this project, we didn't consider the problem of NN training, though HW support for training remains an area of active research.)

One final NVDLA limitation is that it currently only supports one accelerator configuration, "nv_full" – see https://github.com/nvdla/hw/issues/94. Per **hw**/spec/defs/nv_full.spec, this configuration features 2048 multiply-accumulators in the CONV FU, and operates in 8-bit fixed-point (integer) mode for all functional units. The primary and secondary memory buses are each 512 bits wide. Discussion on the **hw** GitHub issue tracker indicates that different configurations, such as nv_small, will be supported in future NVDLA releases.

Results

For all evaluations, LeNet was run on a 28x28 MNIST .pgm input image. AlexNet and R-CNN were run on a 227x227 .jpeg input image, which was decompressed by the host before being sent to the accelerator. The CONV+SDP loadable was run with its integrated input data matrix. All inference operations were performed on 8-bit fixed-point (integer) representations.

Op counts

To get a high-level sense of the networks' compositions, we first consider the number of NVDLA operations necessary to run each of them to completion. The table and chart below show the relative makeup of each network, with the total number of operations listed atop each bar. Recall that none of the networks we tested use the BDMA or Rubik functional units.

Table 5: Neural net op counts

Network	CONV	SDP	PDP	CDP	Total
AlexNet	15	22	3	2	42
R-CNN	15	22	3	2	42
LeNet	4	5	2	0	11
CONV+SDP	1	1	0	0	2

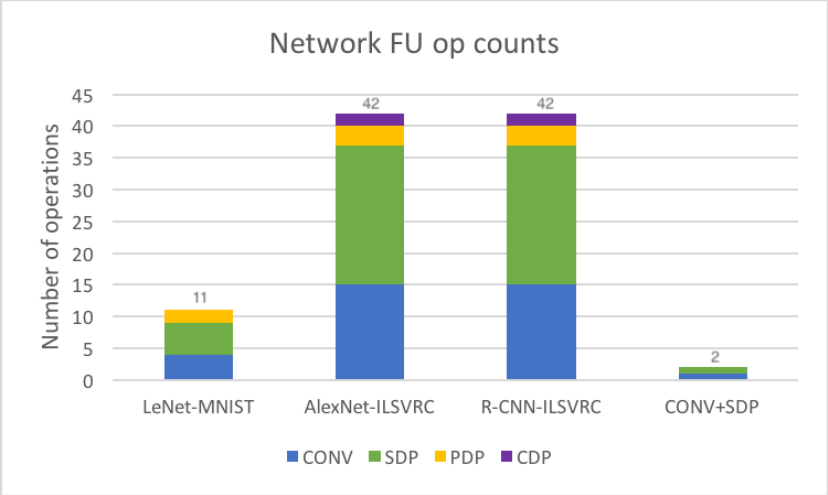

Figure 12: Neural net op counts

At least in terms of number of operations, CONV and SDP ops dominate all other kinds. This makes sense, per the networks' namesake (CNN), and the fact that activations almost always follow convolutions. Recall that the CONV unit is also used for fully-connected layers, but that SDP activations usually follow these, too. What is more interesting is that the number of SDP ops is commonly *greater than* the number of CONV ops. Upon investigating the logs, we found that, for all three of the "real" networks, the scheduler was subdividing some – but not all – SDP stages into two separate ops, one immediately following the other. This behavior occurred most often in the early stages of the network, where the amount of data output by the preceding CONV layer was greatest.

This indicates that a wider SDP unit could *potentially* process all CONV outputs in one operation. Physically, this would be manifested in NVDLA as adding ports to the SDP element-wise LUT. The `SDP_EW_THROUGHPUT` parameter in the **hw**/spec/defs/nv_full.spec file indicates that SDP width will indeed eventually be adjustable. For now, however, its value is set at 4 ports, due to the fixed nv_full config. So, either the SDP unit itself is too narrow, or some other bottleneck exists in the pipeline between CONV and SDP. We surmise that, if such an external bottleneck exists, it is likely related to the pipeline components operating in independent, rather than fused, mode, with data needing to go back and forth on the DRAM interface instead of through the higher-throughput inter-stage FIFOs.

We saw that the scheduler sometimes subdivides ops to make them fit in the FUs. However, this means that some operations, even subdivided ops of the same "macro-op," may have different input data sizes. To resolve this potential imbalance, we now focus on the total amount of data processed by each FU throughout the course of execution of the *entire network*.

FU dataflow analysis

The number of data bytes consumed by each functional unit, across the execution of an entire network, is provided in tabular and graphical formats below. Because the networks have vastly-varying amounts of data (R-

CNN and AlexNet much more than LeNet and CONV+SDN), to maintain reasonable y-axis ranges, they are plotted separately.

Table 6: Bytes processed by each NVDLA FU

Network	CONV input	CONV weight	SDP	PDP	CDP	Total
AlexNet	5,478,592	122,188,416	2,635,104	1,040,576	954,048	990,752
R-CNN	5,478,592	115,634,816	2,635,104	1,040,576	226,496	132,296,736
LeNet	37,376	861,184	47,136	45,056	0	125,015,584
CONV+SDP	16,384	73,728	0	0	0	90,112

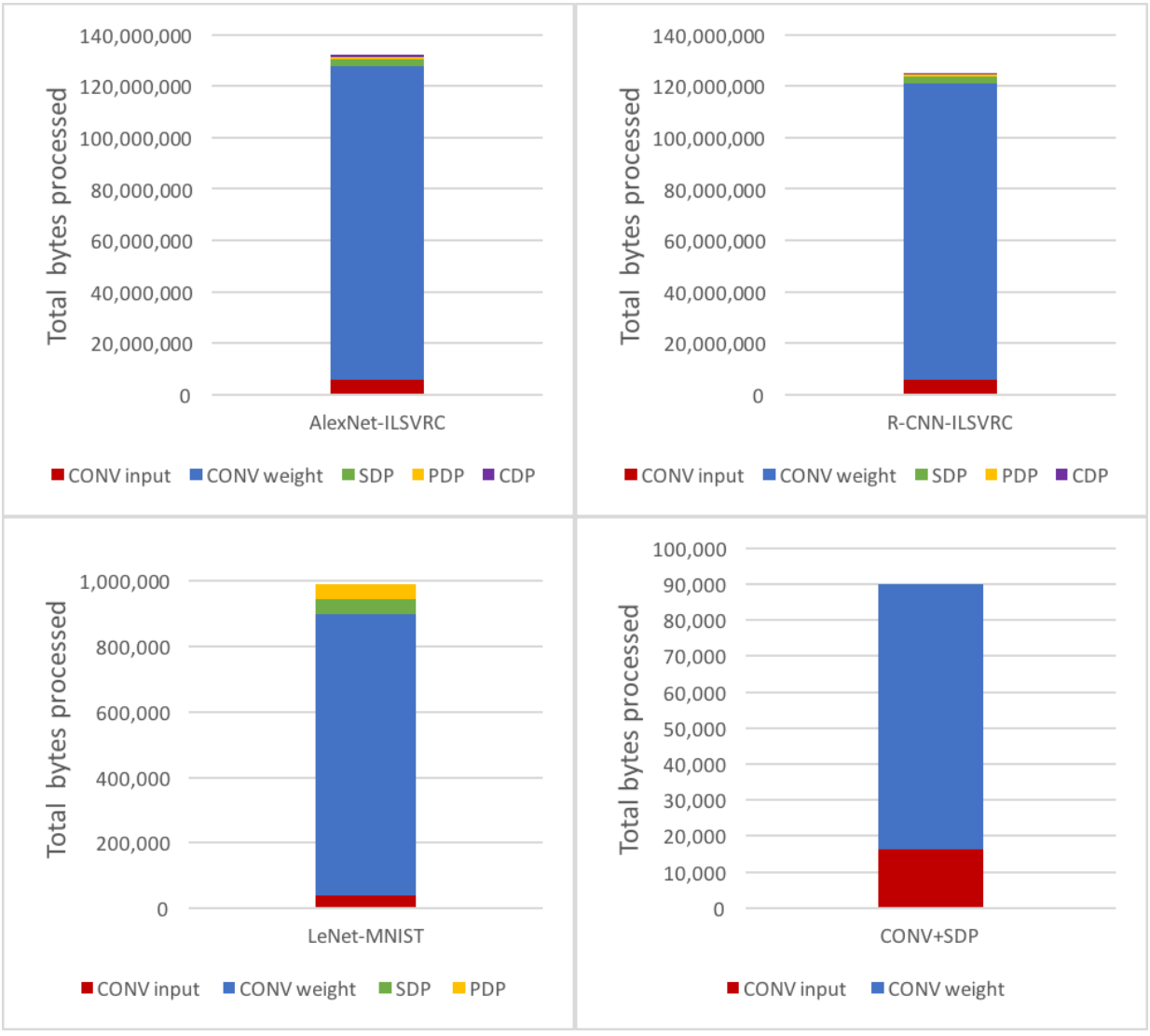

Figure 13: Bytes processed by each NVDLA FU

One interesting thing to note about these graphs is that, compared to the op counts chart, CONV (due to its weights) dominates an even larger portion of the overall percentages. This adds credence to the hypothesis that nv_full's SDP unit is undersized compared to the CONV unit that precedes it. Specifically, we notice that for CONV, there is a lot more weight data than there is input (i.e., activation) data. This makes sense, as input dimensions for R-CNN and AlexNet are 227x227, while each of their .caffemodel (weights) files were around 200 megabytes in

size. Note also that the final layers of all three "real" networks are fully-connected. In contrast to a true convolution, this matrix-matrix multiplication in the FC layers can blow up the output matrix dimensions by multiplying a smaller input matrix by a much larger weights matrix, further increasing the amount of weights data consumed.

Finally, we observe that in CONV+SDP, the SDP operation is really a no-op. This might be related to a scheduler issue that requires an SDP to follow a CONV, even if the SDP doesn't do anything (identity function). Here, however, it doesn't even act as an identity function – it simply takes in no data at all. CONV + SDP is a pre-packaged loadable intended to test only the CONV unit, so this isn't an issue.

Network simulation runtime

Unfortunately, the NVDLA SystemC virtual platform doesn't come with any tools for estimating cycles consumed. In the NVDLA literature, the authors suggest that an FPGA may be used instead for "limited cycle-counting perfomance evaluation" [Primer]. However, FPGA support for nv_full is still forthcoming (see https://github.com/nvdla/hw/issues/90). It may also be possible to instrument the SystemC in the virtual platform to count "virtual" cycles; we did not explore this option.

However, in order to get *some* time-based data, we simply looked at the simulation times required for the four networks. Obviously, simulation time corresponds *very* crudely to real execution time, but does give some sense of the amount of computation necessary to run all the stages to completion.

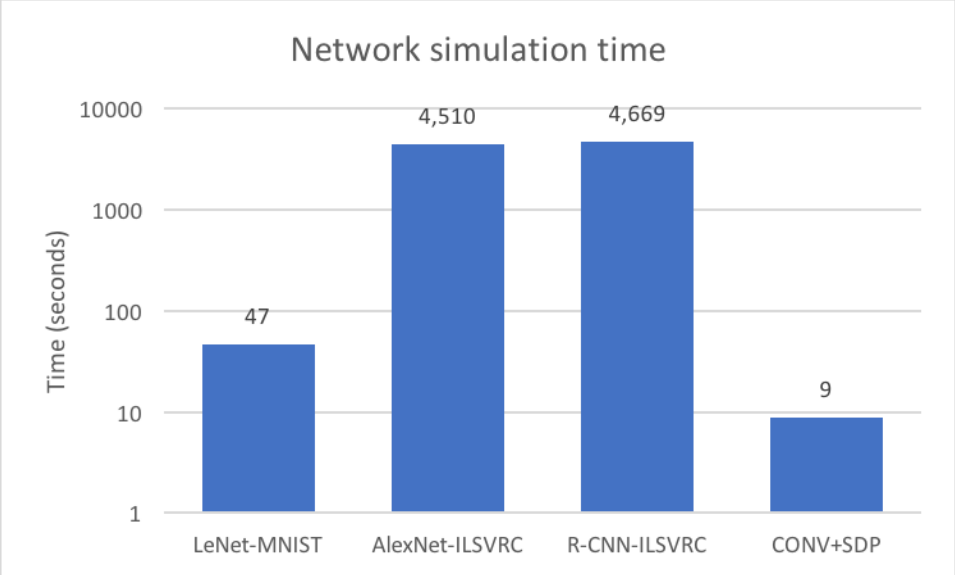

Figure 14: Network simulation times

CONV+SDP and LeNet are significantly shallower networks than AlexNet and R-CNN, and exhibit correspondingly lower runtimes. Also recall that AlexNet and R-CNN operate on 227x227-pixel images, while LeNet's input is 28x28, and CONV+SDP uses an integrated 8x8 input image. As AlexNet and R-CNN both require 42 ops, in approximately the same order, it is reasonable that their runtimes are so similar.

The most interesting takeaway from the data is that simulation time seems to scale fairly well with network size. I.e., there wasn't a great deal of unavoidable overhead present for the smaller networks. We expect that moving the simulation to an FPGA can drastically reduce simulation times, and thus grant us the opportunity to sweep an even broader range of design parameters once the nv_full restriction is lifted.

Future directions

Our primary limitation with the NVDLA platform is that, for the time being, it doesn't give us cycle or timing information. Unfortunately, this prevented us from unifying the circuit-level work with the simulator work to the degree that we would have liked. Nonetheless, we were able to obtain valuable insights from each phase of the project, such as MAC parallelism vs. energy/area, and how neural net execution on a real hardware accelerator might be scheduled. Once the NVDC compiler is open-sourced and made more robust, we can look at a wider variety of networks; specifically, we could simulate VGG-19 on NVDLA to compare to our sequential MAC analysis. Finally, we eventually hope to synthesize and place-and-route the NVDLA design; this would give accurate clock cycle time estimates, as well as permitting us to experimentally change out the default 28nm Si CMOS logic + DRAM for CNFET + RRAM, which may provide significant EDP benefits. [Aly15].

Conclusion

The rise of abundant data computing, where a massive amount of structured and unstructured data is analyzed, has placed extreme demands on the energy efficiency of today's computing systems. With state-of-the-art CMOS circuits pushing the limits of Dennard Scaling [Dennard74], improving the energy efficiency of general purpose processor systems has become increasingly challenging. Several alternate hardware models have been explored as potential successors to general purpose processor cores, including field-programmable gate arrays with bit level configurability [Putnam14], domain-specific accelerators with limited programmability [Volta], or fixed-function accelerators [Jouppi17], each with its own tradeoffs. For example, field-programmable gate arrays are highly programmable, at the cost of higher energy/execution time per operation due to routing overheads and mapping logic primitives (e.g., ANDs, ORs, etc.) to multi-purpose lookup tables. On the other hand, fixed-function accelerators provide lower energy/execution time per operation due to optimized circuit implementations, at the cost of limited programmability. For this project, we focused on fixed-function, deep learning inference accelerators for convolutional neural networks, as they represent a bound on the minimum energy/execution time that can be achieved in hardware for a popular class (e.g., deep learning inference) of abundant-data applications.

Our circuit-level design space gave us a deep understanding of the crux of the design problem. We found that deep learning inference applications such as convolutional neural networks can be parallelized to a high degree. In particular, the area of the resultant accelerator becomes a difficult problem to manage as the application is aggressively parallelized. We found that a sequential MAC-based topology produces a design with a more reasonable area, at the cost of increased execution time and leakage energy. We show that naïve scheduling via aggressive parallelization (e.g., designing a large combinational circuit for the whole neural network inference application) is impractical, and scheduling is an important and necessary overhead that must be characterized in order develop accelerators that run applications at the optimal point on the roofline curve shown in Figure 1. We also show that pooling operations (e.g., maxpool) and activation function (e.g., ReLU) do not comprise a significant fraction of the energy and execution time of a neural network accelerator. Given better NVDLA compiler support, we would have been able to analyze a wider range of neural network inference applications, and characterize the roofline model for the accelerator provided by the NVDLA framework. Using this information, we would have bene able to characterize the scheduling overhead of the NVDLA dataflow. In the end, we were able to get some useful dataflow information out of NVDLA, even though we were not able to measure accurate count cycles with it. In the future, we hope to add hooks into NVDLA to improve the simulation accuracy, validate the simulator against physical design layouts of accelerator, and unify the bottom-up and top-down approaches in Parts I and II, respectively.

Attributions

William performed in-depth circuit-level analysis and obtained adder, multiplier, and MAC energy-delay results in Part I. Andy brought up NVDLA and implemented DLSim on top of it in Part II. Both team members

contributed equally to the Introduction and Conclusion, and to the project in general. All work done and data collected for the project was performed over the 10-week Winter 2017-2018 academic quarter.

DLISim source code is available at https://github.com/andrewbartolo/dlisim. In this repository, you'll find both a collection of logs from the four networks we were able to run, and Python scripts to process and annotate the logs.

Citations

[Abadi16] "TensorFlow: A system for large-scale machine learning." M. Abadi et. al. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.
[Aly15] "Energy-efficient Abundant-data Computing: The N3XT 1,000x." M. M. S. Aly et al. *Computer Magazin*e. 2015.
[Bell78] "The CRAY-1 Computer System." G. Bell et. al. *Communications of the ACM*. 1978.
[Binkert14] "The gem5 simulator." N. Binkert et. al. *ACM SIGARCH Computer Architecture News*.
[Chen16] "Eyeriss: A spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks." Y.-H. Chen et. al. *IEEE International Solid-State Circuits Conference*. 2016.
[Dennard74] "Design of ion-implanted MOSFETs with very small physical dimensions." R. Dennard et. al. *IEEE Journal of Solid-State Circuits*. 1974.
[Gelsinger00] 'The International Technology Roadmap for Semiconductors (ITRS): "Past, Present, and Future."' http://ieeexplore.ieee.org/document/906261/. Retrieved March 2018.
[Gonzalez96] "Energy Dissipation in General Purpose Microprocessors." R. Gonzalez, M. Horowitz. *IEEE Journal of Solid-State* Circuits. 1996.
[Han16] "EIE: Efficient Inference Engine on Compressed Deep Neural Network". S. Han et. al. *International Symposium on Computer Architecture*. 2016.
[HMC] "Hybrid Memory Cube (HMC)." J.T. Pawlowski. *Hot Chips 23*. 2011.
[Jouppi17] "In-Datacenter Performance Analysis of a Tensor Processing Unit." N. Jouppi et. al. *International Symposium on Computer Architecture*. 2017.
[Kocher18] "Spectre Attacks: Exploiting Speculative Execution." P. Kocher et. al. https://spectreattack.com/spectre.pdf. 2018.
[Krizhevsky12] "ImageNet Classification with Deep Convolutional Neural Networks." A. Krizhevsky, I. Sutskever, and G. Hinton. *Neural Information Processing Systems*. 2012.
[Kung82] "Why Systolic Architectures?" H.T. Kung. *IEEE Computer*. 1982.
[Linaro] "Linaro – Leading software collaboration in the ARM ecosystem." https://linaro.org/downloads. Retrieved March 2018.
[Mangard18] "Meltdown and Spectre." S. Mangard et. al. https://meltdownattack.com/meltdown.pdf. 2018.
[MCM] "Multi-Chip Module". *Techopedia*. https://www.techopedia.com/definition/11836/multi-chip-module-mcm. Retrieved Mar. 2018.
[Miller10] "Graphite: A Distributed Parallel Simulator for Multicores." J. Miller et. al. *International Symposium on High-Performance Computer Architecture*. 2010.
[ModelZoo] "BVLC CAFFE ModelZoo." https://github.com/BVLC/caffe/wiki/Model-Zoo. Retrieved March 2018.
[NECAurora] "A deep dive into NEC's Aurora vector engine." T. Morgan. *TheNextPlatform*. https://www.nextplatform.com/2017/11/22/deep-dive-necs-aurora-vector-engine/
[Nowatzki14] 'gem5, GPGPUSim, McPAT, GPUWattch, "Your favorite simulator here" Considered Harmful.' T. Nowatzki et. al. *2014.*
[NVDLA] "The NVIDIA Deep Learning Accelerator." http://nvdla.org. Retrieved March 2018.
[Primer] "NVDLA Primer." http://nvdla.org/primer.html. Retrieved March 2018.
[Putnam14] "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services (Catapult)." *International Symposium on Computer Architecture*.
[Roadmap] "NVDLA Open Source Roadmap." http://nvdla.org/roadmap.html. Retrieved March 2018.
[Sanchez13] "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems." D. Sanchez and C. Kozyrakis. *International Symposium on Computer Architecture.* 2013.

[Simoyan14] “Very Deep Convolutional Networks for Large-Scale Image Recognition.” K. Simoyan and A. Zisserman. *International Conference on Learning Representations.”* 2014.
[Volta] “NVIDIA Tesla V100 GPU Architecture.” http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Retrieved March 2018.