High-Performance Deep-Learning Coprocessor Integrated into x86 SoC with Server-Class CPUs

Industrial Product

Glenn Henry* Parviz Palangpour* Michael Thomson* J Scott Gardner† Bryce Arden* Jim Donahue*
Kimble Houck* Jonathan Johnson* Kyle O'Brien* Scott Petersen* Benjamin Seroussi* Tyler Walker*

*Centaur Technology †Advantage Engineering LLC

***Abstract*—Demand for high performance deep learning (DL) inference in software applications is growing rapidly. DL workloads run on myriad platforms, including general purpose processors (CPU), system-on-chip (SoC) with accelerators, graphics processing units (GPU), and neural processing unit (NPU) add-in cards. DL software engineers typically must choose between relatively slow general hardware (e.g., CPUs, SoCs) or relatively expensive, large, power-hungry hardware (e.g., GPUs, NPUs).**

This paper describes Centaur Technology's Ncore, the industry's first high-performance DL coprocessor technology integrated into an x86 SoC with server-class CPUs. Ncore's 4096 byte-wide SIMD architecture supports INT8, UINT8, INT16, and BF16 datatypes, with 20 tera-operations-per-second compute capability. Ncore shares the SoC ring bus for low-latency communication and work sharing with eight 64-bit x86 cores, offering flexible support for new and evolving models. The x86 SoC platform can further scale out performance via multiple sockets, systems, or third-party PCIe accelerators. Ncore's software stack automatically converts quantized models for Ncore consumption and leverages existing DL frameworks.

In MLPerf's Inference v0.5 closed division benchmarks, Ncore achieves 1218 IPS throughput and 1.05ms latency on ResNet-50-v1.5 and achieves lowest latency of all Mobilenet-V1 submissions (329µs). Ncore yields 23x speedup over other x86 vendor per-core throughput, while freeing its own x86 cores for other work. Ncore is the only integrated solution among the memory intensive neural machine translation (NMT) submissions.

I. Introduction

In recent years, deep learning (DL) applications have become widespread. The rush to support DL workloads on a wide range of platforms has accelerated research efforts, advancing DL models and frameworks at breakneck speed. Significant research has also pushed DL hardware architectures forward, with a relatively recent emphasis on inference hardware. This is evidenced by the development and deployment of custom accelerators from hyperscalers such as Google, Amazon, Microsoft, and Alibaba [13] [3] [7] [2]. However, most hardware advances have been targeted at datacenters and are only available as cloud-based services. Other solutions, such as graphics processing units (GPU) and neural processing unit (NPU) expansion cards, tend to be large, expensive, and power-hungry. Other alternatives tend to be relatively low-performance solutions, such as general-purpose processors (CPU) and system-on-chip (SoC) accelerators. Existing technologies leave a large gap in DL hardware solutions with regard to performance, cost and form factor.

This paper is part of the Industry Track of ISCA 2020's program.

Many established semiconductor companies and startups are striving to design new inference hardware to fill this market gap. However, developing a high performance, low-cost, small form factor DL accelerator has a high barrier to entry. This is true especially for startups, which must build up infrastructure and a wide breadth of technical expertise in one of the most rapidly changing landscapes. Initial efforts can become obsolete before hardware is fabricated, and the high-risk startup environment can make it difficult to recruit experts. However, the ever-changing DL frontier and huge demand for wide ranges of hardware will inevitably lead many to pursue various areas of the market.

Centaur Technology's unique position to leverage existing infrastructure, in-house x86 designs, expert engineers, and regular development and fabrication cycles removes many barriers to the DL inference hardware space. This position caters to Centaur's historical design goals of adding high value to its low-cost offerings. This paper describes Centaur's 20 tera-operations-per-second DL coprocessor technology (Ncore), integrated with eight 64-bit x86 CPUs utilizing Centaur's new CNS microarchitecture into a single SoC platform (called CHA). CHA is the industry's first x86 SoC with integrated high-performance DL coprocessor and server-class CPUs.

We show that the inference hardware arena is prime for Centaur's entry by demonstrating a full hardware and software solution delivered with minimal risk, high-performance, and small form factor. This solution has been deployed in third-party video analytics prototypes, as well as in engineering development platforms internally.

Ncore's latencies beat those of established DL giants in the MLPerf Inference v0.5 Closed-division benchmark, while also providing high throughput and multifaceted flexibility. CHA's integration of both x86 and Ncore can flexibly support ever-changing DL models and can further expand performance capabilities through multi-socket, multi-system, and multi-expansion card solutions as desired.

The goals, tradeoffs, and approach used to design Ncore are discussed in section II. Sections III and IV describe

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00013

the resulting hardware implementation of the x86 SoC and Ncore. Section V describes the software stack developed for supporting Ncore in existing DL frameworks. Section VI interprets Ncore's results in the MLPerf Inference v0.5 Closed-division (Preview) benchmark [22]. We discuss related work in section VII and summarize conclusions in section VIII.

II. Background / Design Goals

The Ncore project began with Centaur's CHA SoC and CNS microarchitecture already under development. Having a new SoC design underway contributed to Centaur's relatively low barrier to introducing a new DL accelerator, but some initial challenges were still present:

- Clean slate – no legacy DL hardware.
- No off-the-shelf DL architecture IP matching our goals.
- Unclear how much CHA silicon available for Ncore.
- Must live with constraints inherited from CHA.
- All engineers already busy working on CNS / CHA.
- Future DL model uncertainty – future-proofing.

The most significant challenge would be developing a new in-house DL accelerator from a clean slate, with all the research, analysis, tradeoffs, and work hours associated with such an endeavor. However, most of the challenges were partially offset by our initial inherent strengths:

- Clean slate – can design whatever we want.
- Ncore reuses existing CHA power, I/O, etc.
- Fast access to cache, DRAM, x86 cores.
- A company full of expert logic designers.
- Several recent university hires to borrow.
- x86 cores can help support new, changing models.

Designing from a clean slate also meant that the design could be geared toward our strengths, further reducing risk, and allowing the focus to remain on Ncore-specific design decisions and tradeoffs rather than the surrounding infrastructure (like power, memory, clocking, I/O, circuit libraries, etc.).

A. Design Tradeoffs

The main design options that needed to be resolved are shown in Table I, with Centaur's eventual choices indicated with bold text. CHA's potential for low latency, low risk, and flexibility became the main driving forces behind each of the tradeoff areas described below.

1) Target environment: The bulk of existing DL inference hardware targets cloud environments, and our CNS microarchitecture was being driven toward server-class capabilities. It was clear that CNS would be suited to servers, workstations, and notebooks. With the addition of Ncore, CHA is particularly well-suited to edge servers and commercially in-demand models and applications such as real-time video analytics.

2) Design optimization: Ncore raw performance would inherently be limited by not having exotic memories such as those utilized in GPUs and NPUs deployed in cloud environments. Additionally, CHA peak power will always include not only Ncore, but also all eight x86 cores. A clear win is to target performance per total system cost, with Ncore's cost being relatively "free" at the system level.

TABLE I
Tradeoff options and decisions (selected option marked with bold text).

Tradeoff Area	Options
Target environment	Cloud **Edge server** Notebook Phone
Design optimization	Raw perf **Perf / $** Perf / W
Connectivity w/ CHA	Direct-to-CPU In-cache I/O **Ring, mem-mapped**
Architecture type	Systolic array Distributed CPU **SIMD**
Voltage and frequency	Independent **Match CHA**
Data types	**INT8** **UINT8** **INT16** **BFloat16** Float32

3) Connectivity: The designers explored various connectivity options between Ncore and the rest of CHA. Ultimately, placing Ncore on the existing CHA ring bus provided the lowest risk, lowest cost, and most flexible solution. Designing Ncore to appear as a PCI device on the ring bus also allowed Ncore's software stack to leverage existing PCI libraries provided by the operating system, while still maintaining tight integration with the SoC's memory hierarchy.

4) Architecture: Many DL accelerators implement a systolic array. However, as an x86 company, Centaur is especially adept at designing SIMD logic. Initial evaluation convinced us that we could make the accelerator more flexible and higher performing with a SIMD approach. A SIMD-based architecture also reduced the risk associated with not knowing early on exactly how much CHA silicon would be allocated for Ncore – the SIMD architecture was easy to slice and expand as needed for the area allocated.

5) Voltage and frequency: Centaur's desire to reduce risk and reuse existing CHA infrastructure, combined with the confidence to produce a highly-clocked SIMD inference engine, drove the decision to keep Ncore at the same frequency and voltage as the rest of CHA.

6) Data types: At the time of deciding which data types to support, it was not yet clear that 8-bit inference would be as widespread as it is now. Early work demonstrated that reduced precision inference was viable, but these techniques required quantization-aware training of the DL models, which is a non-trivial effort [12] [8] [9]. However, specific 8-bit quantization schemes have emerged that do not require re-training and achieve small reductions in accuracy [5]. We decided to support 8-bit datatypes, as well as include bfloat16 and int16 as fallbacks for networks that require higher precision to maintain reasonable accuracy. As bfloat16 has emerged as a viable training data type, migrating bfloat16 trained models to inference

Fig. 1. High-level CHA block diagram.

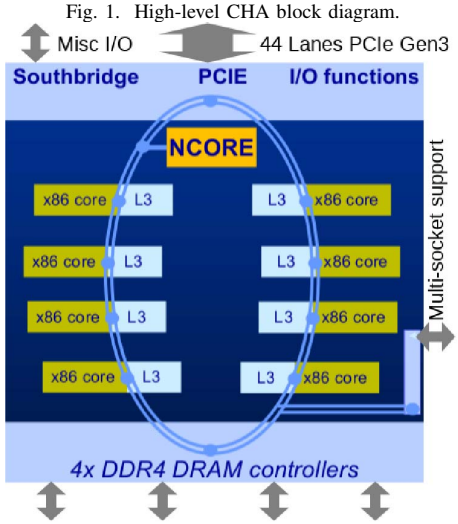


TABLE II
PEAK THROUGHPUT (GOPS/SEC)

Processor	8b	bfloat16	FP32
1x CNS x86 2.5 GHz	106	80	80
Ncore 2.5 GHz	20,480	6,826	N/A

on Ncore has become straightforward and avoids accuracy loss [14]. Int16 is particularly useful to maintain precision when working with int8 quantized values with different ranges. Ncore's expected peak throughput per data width is compared against respective x86-only peak throughput in Table II.

III. CHA ARCHITECTURE

CHA consists of Ncore and eight 64-bit x86 cores utilizing Centaur's CNS microarchitecture, shown in Figure 1. CHA's bidirectional ring bus is 512 bits wide in each direction, with 1-cycle latency between ring stops. At 2.5GHz, each ring direction provides up to 160 GB/s of bandwidth, or 320 GB/s combined. The ring bus includes ring stops for each x86 core, Ncore, I/O, memory controllers, and multi-socket logic. The memory controller supports four channels of DDR4-3200 DRAM, providing 102 GB/s peak theoretical throughput.

CHA was fabricated using TSMC's 16 nm FFC technology. CHA's die layout is shown in Figure 2. CHA is 200 mm^2, with Ncore accounting for 17% of the total area.

Table III compares Centaur's CNS microarchitecture against Intel's Haswell and Skylake Server microarchitectures [10]. Details between the three microarchitectures are similar. Compared against Haswell, CNS has higher L2 cache associativity, larger store buffer, larger scheduler, and smaller per-core L3 cache. Compared against Skylake Server, CNS has a larger per-core L3 cache, but smaller L2 cache, store buffer, reorder buffer, and scheduler.

Fig. 2. CHA SoC die layout, 200 mm^2 in TSMC 16 nm FFC technology.

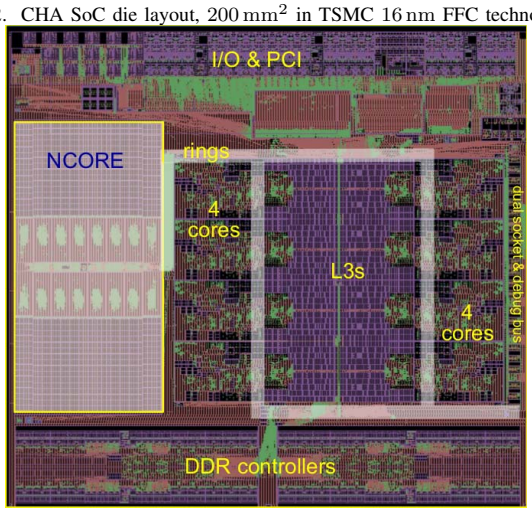


TABLE III
CNS MICROARCHITECTURE VS HASWELL, SKYLAKE SERVER

	CNS	Haswell	Skylake Server
L1I cache	32KB, 8-way	32KB, 8-way	32KB, 8-way
L1D cache	32KB, 8-way	32KB, 8-way	32KB, 8-way
L2 cache	256KB, 16-way	256KB, 8-way	1MB, 16-way
L3 cache/core	2MB shared	2MB shared	1.375MB shared
LD buffer size	72	72	72
ST buffer size	44	42	56
ROB size	192	192	224
Scheduler size	64, unified	60, unified	97, unified

IV. NCORE MICROARCHITECTURE

A. Connectivity

Ncore connects to the rest of the SoC through CHA's bidirectional ring, as shown in Figure 3. It has the benefits of PCI device programmability, while being integrated directly with the SoC's memory hierarchy. The x86 cores configure Ncore through its PCI and memory-mapped slave interfaces. The x86 cores can also access data and weights in Ncore, although this is usually only done at the beginning and end of latency-critical applications. Normally, for optimal throughput, x86 will place data and weights in system memory and allow Ncore to access them directly through its DMA interface. Ncore can handle simultaneous DMA reads, DMA writes, x86 reads, and x86 writes concurrently while Ncore is executing a workload. Ncore also has the ability to use DMA to read CHA's shared L3 caches, which will subsequently retrieve the data from system DRAM if not present in the L3. The extra hop through the L3 minimally increases the latency to DRAM, so the feature isn't needed for purely streaming workloads. Ncore reads from L3 are coherent, while Ncore internal memory is not coherent with the SoC memory system. The L3 cache DMA feature was not used as part of the performance evaluation presented in section VI. All CHA logic runs in a single frequency domain.

Fig. 3. Ncore-CHA ring bus connectivity (Ncore blocks highlighted orange).

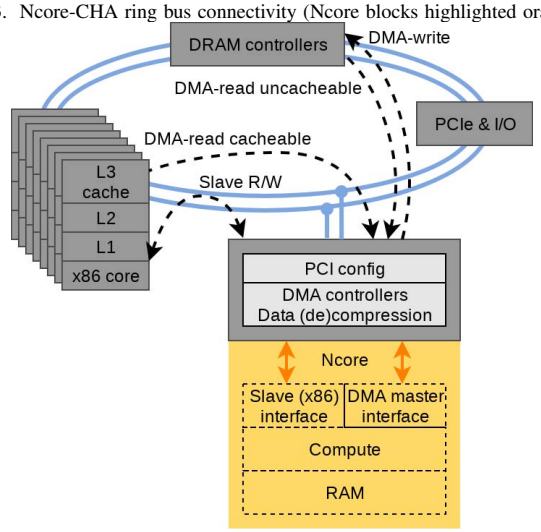
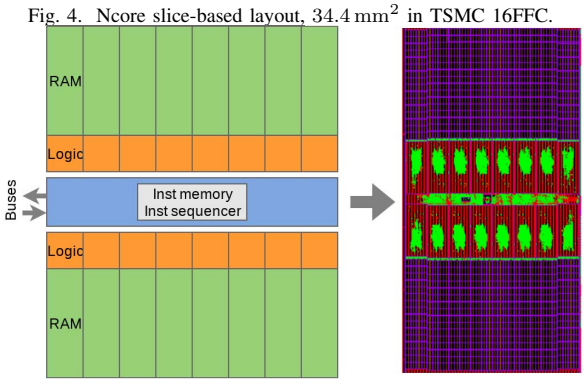


Fig. 4. Ncore slice-based layout, $34.4\,\mathrm{mm}^2$ in TSMC 16FFC.



B. Physical Configuration

Ncore comprises 16 slices, as shown in Figure 4. Each slice is 256 bytes wide, giving a total width of 4,096 bytes. Each slice's RAM consists of two SRAM banks, each with 2,048 rows. Ncore's layout was designed such that its size and aspect ratio could be easily modified to fit whatever area in CHA would eventually be reserved for Ncore. Adding or removing slices alters Ncore's breadth, while increasing or decreasing SRAM capacity alters Ncore's height. The final CHA area limitations allowed for 16 slices with 16MB of RAM. Ncore's area is $34.4\,\mathrm{mm}^2$, which accounts for 17% of CHA's total area.

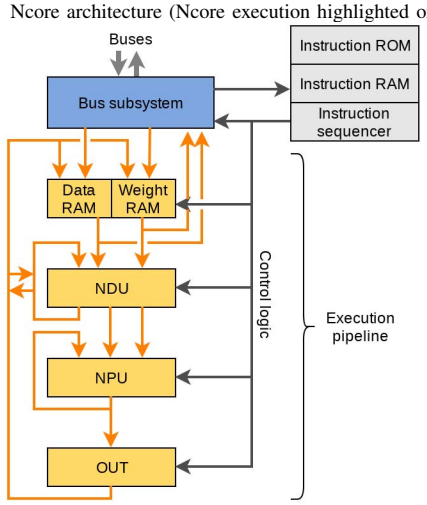
C. Memory

Ncore has 16MB of RAM, split up into separate data and weight RAMs of 8MB each (512KB per slice). 16MB across 4,096 multiply-accumulate (MAC) units equates to a relatively large average of 4KB per MAC. Ncore's RAM provides a total of $20\,\mathrm{TB/s}$ internal throughput. Ncore's die layout image in Figure 4 shows that Ncore's RAMs account for approximately two-thirds of total Ncore area.

Ncore has 8KB of instruction RAM, which is double buffered and can be filled by any x86 core during Ncore execution. For all workloads tested so far, the instruction RAM double-buffering allows instruction RAM loading to not hinder Ncore's latency or throughput. Ncore's instruction RAM is also augmented with a 4KB instruction ROM for storing commonly executed code and self-test routines.

Ncore's DMA engines can access system DRAM, as configured by Ncore's kernel driver. Ncore can access up to 4GB of system DRAM for a fully configured set of DMA base address registers. However, Ncore can potentially access all available system memory (i.e., much more than 4GB) if the driver and runtime track Ncore execution and dynamically modify the DMA base address registers as needed.

Fig. 5. Ncore architecture (Ncore execution highlighted orange).



D. Ncore Execution

Figure 5 shows the Ncore architecture at the slice level. The architecture consists of the instruction RAM and ROM, instruction sequencer, data and weight RAMs, and execution pipeline. The execution pipeline comprises the neural data unit (NDU), neural processing unit (NPU), and output unit (OUT).

1) Instruction Sequencer: The instruction sequencer decodes Ncore instructions, which are 128 bits wide and similar to VLIW. All instructions execute in a single clock cycle. The example code in Figure 6 shows the large amount of work that can be done in a single Ncore instruction. The code example is a convolution's inner loop, and the entire loop can be encoded in a single Ncore instruction that executes each iteration in one clock cycle. This code listing is representative of the actual code syntax our tooling consumes to generate instructions for Ncore. This level of code is abstracted away from the end user via the tooling described in section V, so there is no need for users to learn a new domain-specific language.

2) Data and Weight RAMs: Reads and writes to these RAMs take one clock for an entire 4,096-byte row. Both the data and weight RAMs can be read each clock, although only one can be written per clock. The bus subsystem can read and write Ncore's RAMs, but these accesses are row-buffered to avoid interfering with Ncore execution. The RAMs implement

Fig. 6. Convolution inner loop, internal code representation for Ncore. This entire loop can be encoded in a single Ncore instruction, executing a full iteration in one clock cycle. The rotate operation is not typically within a convolution inner loop, but can be included to optimize specific convolutions.

```
1   for (int loop_cnt[1] = 0; loop_cnt[1] < 3; loop_cnt[1] += 1) {
2     w_mov_reg = broadcast64(wtram[addr[3]], addr[5], increment);
3     acc += (d_last_latched >> 1) * w_mov_reg;
4     d0_mov_reg = d0_mov_reg.rotate_left(64);
5   }
```

64b ECC which can correct 1-bit errors and detect, but not correct, 2-bit errors. When the RAMs contain 16-bit data, the data is split into low bytes held in one RAM row and high bytes in the next row.

3) Neural Data Unit: The NDU performs data bypass, data row rotation, data block compression, byte broadcasting, and masked merge of input with output. The NDU performs up to three (typically two) of these operations in parallel, all within a single clock cycle. The results are stored in up to four NDU output registers. The NDU operations have nine possible input sources: data RAM, weight RAM, instruction immediate data, the NDU's four output registers, and the OUT unit's high / low byte output registers. Additionally, each slice's NDU unit is connected to neighboring slices' NDU units for rotating across slice boundaries. An entire 4KB row can be rotated in either direction, up to 64 bytes per clock cycle.

4) Neural Processing Unit: The NPU performs MACs, additions, subtractions, min/max, logical operations, and other variations. The NPU optionally converts unsigned 8-bit values to signed 9-bit values by subtracting a zero offset, with separate offsets used for data and weights. The NPU has a 32-bit saturating accumulator, which can be conditionally set via predication registers and operations. 8-bit NPU operations execute in one clock cycle, Bfloat16 operations take three clock cycles, and INT16 operations take four. NPU data inputs can be forwarded to the adjacent slice's NPU, with wraparound from the last slice back to the first. Thus, data can "slide" across all 4,096 bytewise execution elements with different weights. This full-width data forwarding is critical to our high-performance convolutional algorithms.

5) Output Unit: The OUT unit has specialized functionality. It performs requantization of the 32-bit accumulator to the desired 8-bit and 16-bit data types. The requantization is performed by multiplying the accumulator with a range value, shifting the result left or right based on a scale value, and adding an offset value. The OUT unit also performs activations such as ReLU, tanh, and sigmoid. It can store different transformations of the 32-bit accumulator.

E. Convolution Execution

As Ncore is programmable, a number of implementation strategies may be used to maximize utilization for a given convolution. Figure 7 illustrates a dataflow representation [15] of one convolution implementation strategy. One spatial dimension (width or height) is selected and rounded up to the nearest power-of-2. For simplicity, this example assumes a convolution where $N = 1$, $W = 64$, and $K = 64$. $W \times K$ is parallelized over Ncore's 4096 SIMD width, meaning that Ncore is accumulating into 64 output channels that are each 64 elements wide, all in parallel. This strategy fits well with common networks, where increasing network depth tends to coincide with decreased spatial dimensions and increased number of channels. Thus, sufficient parallelism is maintained for Ncore to exploit. The loop nest in Figure 7 relates to the inner-loop previously illustrated in Figure 6, in that Figure 6 uses a `filter_width` of 3 and a `broadcast64` operation to broadcast a single element across each group of 64 accumulators. Note that in order to map the dataflow representation in Figure 7 onto Ncore, we must first perform data and code transformations such that the vector loads and stores operate on contiguous rows of 4096 bytes.

F. Debug Features

Ncore implements three configurable debug features: event logging, performance counters, and n-step breakpointing. Events can be logged to a 1,024-entry buffer, then read out by an x86 core. The event logs can be written and read dynamically without interfering with Ncore's execution. Thus, logging poses no performance penalty on Ncore. Performance counters can be configured with an initial offset and with breakpointing at counter wraparound. Performance counters do not interfere with Ncore execution unless configured to breakpoint at counter wraparound. Ncore's n-step breakpointing allows the runtime to pause Ncore execution every n clock cycles, allowing Ncore state inspection. Event logging, performance counters, and n-step breakpointing are all used extensively for performance and workload debugging. Ncore's runtime software, described in section V, controls these debug features.

V. Software Design

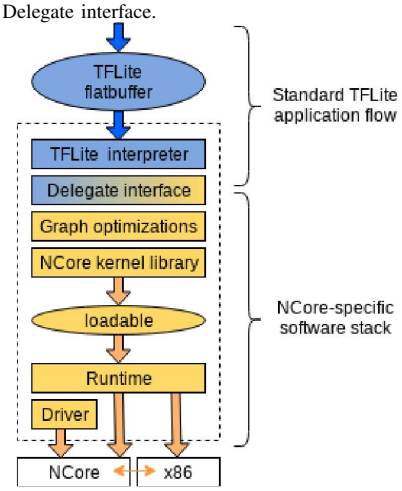
A. Framework

Tensorflow is a large DL framework for distributed training and inference [1]. TensorFlow-Lite is a specialized DL framework for running on-device inference. While TensorFlow-Lite is largely targeted at embedded and mobile platforms, its focus on performance and low-latency inference brings advantages to x86 server platforms as well. TensorFlow-Lite can be easily extended with custom kernel implementations that utilize AVX-512 instructions, which are available on CHA, and provides improved performance over previous SIMD extensions for portions of the workload not offloaded to Ncore. While the software stack presented in this work can support native Tensorflow, TensorFlow-Lite is preferred as it provides the Delegate API, a dedicated mechanism for offloading compatible portions of a DL workload from the CPU to an accelerator. In addition, TensorFlow-Lite has more support for executing quantized models. Figure 8 shows

Fig. 7. Loop nest representation of a convolution dataflow implementation, as mapped to Ncore's 4096-wide SIMD architecture. The $W \times K$ loops (lines 3-4) are parallelized across Ncore.

```
 1   for (int n=0; n < N; n++) {
 2     for (int out_y = 0; out_y < H; out_y++) {
 3       pfor (int k = 0; k < out_channels; k++) {       // W x K loops parallelized on Ncore
 4         pfor (int out_x = 0; out_x < W; out_x++) {
 5           acc = 0;
 6           for (int r = 0; r < filter_height; r++) {
 7             for (int c = 0; c < in_channels; c++) {
 8               for (int s = filter_width-1; s >= 0; s++) {
 9                 if (not_padded)
10                   acc += input(n, y, x, c) * filter(r, s, c, k);
11               }
12             }
13           }
14           output(n, out_y, out_x, out_channel) = acc;
15         }
16       }
17     }
18   }
```

Fig. 8. Ncore software stack when integrated with the TensorFlow-Lite framework's Delegate interface.



how Ncore's software stack integrates with TensorFlow-Lite through the Delegate interface.

B. Graph Compiler and Code Generation Libraries

The first step in the toolflow involves exporting a trained DL model from a DL framework. Popular frameworks such as TensorFlow, TensorFlow-Lite, PyTorch and MXNet utilize their own native dataflow graph formats. Each of these graph formats can be characterized as a graph intermediate representation (GIR) with subtle differences that go beyond just the on-disk serialization format. For example, the definition of padding for some convolutions leads to different results for TensorFlow vs PyTorch.

To support multiple GIRs from different frameworks, the Ncore Graph Compiler Library (GCL) provides frontends that can import framework-specific GIRs into Ncore's own GIR. For example, the TensorFlow-Lite Delegate interface can provide a compatible subgraph that is then converted to the Ncore GIR. Once the graph is imported into the GIR, a number of graph-level passes can be performed that progressively optimize and lower the graph workload, producing an optimized binary that can be executed on Ncore. Initially, the Ncore tools perform a series of generic graph-level optimizations. An example optimization pass is to eliminate batch-normalization operations by folding the batch-normalization constants into adjacent bias-addition operations and convolution filters. A common, subsequent optimization pass fuses the element-wise bias-addition and activation functions into operations such as convolution. Some optimization passes arise from the subtle differences in the framework-specific GIRs. For example, the ResNet-50-V1.5 reference graph provided by MLPerf for TensorFlow has four explicit pad operations. A graph-level optimization pass fuses these explicit pad operations into an adjacent convolution.

After the generic graph-level optimizations have been applied to the GIR, the graph is lowered to Ncore. To maximize the use of hardware features such as hardware loop counters, circular buffer addressing modes, and many special-purpose registers, we utilize hand-tuned inner kernels written in assembly. The code-generation library for Ncore is the Ncore Kernel Library (NKL). The NKL is similar in spirit to popular vendor-optimized deep learning libraries such as NVidia's cuDNN and Intel's MKL-DNN. The NKL is responsible for generating the complete kernel implementation at the assembly level to maximize performance. Rather than support commonly used data layouts such as row-major NHWC, the NKL kernels only provide implementations for a number of internal data layouts that are optimized for Ncore. Since the conversion between an internal Ncore data layout and the external application's data layout only occurs at the edges of an accelerated subgraph, the cost of converting the buffer layout is amortized over the whole subgraph rather than a single operation. The GCL first performs a pass to determine the optimal data layout for data buffers, based on the tensor shapes as well as the nodes that consume and produce the buffers.

Once data layout selection has been determined, the amount of memory required per node in the GIR is known, and

Fig. 9. Graph modification via TensorFlow-Lite's Delegate interface assigns work to Ncore.

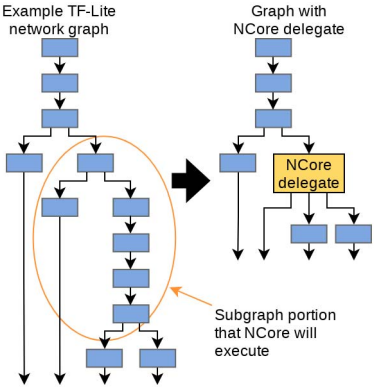
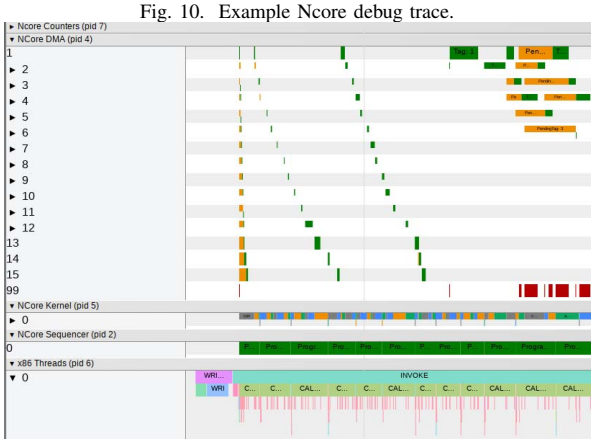


Fig. 10. Example Ncore debug trace.



operations can be scheduled. Since Ncore uses software-managed scratchpad memories rather than a cache, the GCL and NKL perform the appropriate memory management during code generation. As weights must be transferred via DMA into the Ncore scratchpad memories from DDR, the GCL attempts to schedule the weights to be non-speculatively prefetched as early as possible. In the case of MobileNetV1, the GCL determines that all the model's weights fit in on-chip SRAM, and promotes the weight buffers to become *persistent* rather than transferred during execution. The final result is an Ncore Loadable which contains everything needed to execute the DL model on Ncore.

C. Runtime

The Ncore runtime / API provides a high-level abstraction of the low-level, memory-mapped Ncore interface. The runtime exists as a standalone library without third-party dependencies, allowing it to be utilized in implementations ranging from low-level tests to high-level third-party framework plugins. In this paper, we evaluate Ncore benchmark performance with its runtime integrated into TensorFlow-Lite's Delegate interface.

Figure 9 shows how TensorFlow-Lite's Delegate interface splits a network's graph into subgraphs, assigning execution of each subgraph to a specific target. This example shows a portion marked to be executed by Ncore, with the remaining subgraphs executed on x86 cores. At run time, TensorFlow automatically runs the x86 portions and handles the callbacks to pass control to the Ncore runtime where applicable.

The Ncore runtime is responsible for configuring DMA, writing instructions to Ncore's double-buffered instruction RAM, accessing Ncore's debugging features (event logging, performance counters, n-step breakpointing), and more. Figure 10 shows an example runtime trace generated during an Ncore run using Ncore's debugging features.

A flexible runtime that can leverage existing frameworks was vital to Ncore's early development and subsequent success in submitting full-system MLPerf scores in a relatively short timeframe. This approach has yielded quick bring-up of new systems integrating Ncore, from in-house test benches to commercial video analytics systems.

D. Driver

Ncore is connected directly to the SoC's ring bus and reports itself to the system as a standard PCI device. Ncore is detected through the system's typical PCI enumeration as a coprocessor type. The operating system then associates the enumerated device with Ncore's kernel mode driver, which configures Ncore for general use. The kernel driver leverages Linux's robust PCI environment, and augments the standard low-level PCI routines with a number of Ncore-specific tasks:

- Power up Ncore and clear state.
- Reserve / allocate system DRAM for Ncore DMA.
- Configure protected Ncore settings.
- Regulate memory-mapping to Ncore space.
- Provide basic `ioctl` access to user-mode runtime.

The kernel driver accepts requests from the user runtime for memory-mapped access to Ncore's registers, strobes, and internal SRAM. The user runtime uses the memory mapping to configure and run workloads on Ncore, while the driver prevents more than one user from simultaneously gaining ownership of Ncore's address space.

While most Ncore abilities are user configurable via its memory mapped address space, not all features are directly configurable in this way. For example, allowing user code to define Ncore's DMA address ranges would present security concerns. As such, the kernel driver is the sole gatekeeper for configuring system-critical settings like DMA address ranges and powering Ncore up or down. These protected settings live as custom fields in Ncore's PCI configuration space, which by standard is only accessible from system kernel code.

E. Design Methodology

During the design of Ncore, several tools were used to model and verify Ncore's performance. An instruction simulator was developed as the golden model to drive hardware verification efforts and verify correctness of assembly generated by the libraries. Prototyping algorithms with new SIMD instructions, changes to the machine width, and changes to RAM sizes were modeled using a custom C++ vector class library (VCL). The VCL provided a path for quick

TABLE IV
NCORE TEST PLATFORM.

	System configuration
x86 CPU	8-core Centaur SoC
x86 CPU caches	L1: 32KB instruction + 32KB data (per core) L2: 256KB (per core) L3: 16MB shared
x86 CPU frequency	2.5GHz
Ncore	1-core, 4096-byte SIMD
Ncore frequency	2.5GHz 2.3GHz (for GNMT)
Ncore memory	8KB instruction (+ 4KB ROM) 16MB data 4GB system DDR accessible
Storage capacity	128GB SSD drive
Memory capacity	32GB PC3200 DDR4
Operating system	Ubuntu Linux 18.04
ML Framework	TensorFlow-Lite v1.14rc0 TensorFlow v1.14rc0 (for GNMT)
Benchmark	MLPerf Inference v0.5 Closed division

iteration to verify the numerical correctness of algorithms and performance impact before any changes had to be made to the hardware design. The GCL described in Section V was also used to report utilization and DMA stalls based on a high-level performance model that uses VCL instrumentation. As the design matured, hardware emulation was utilized to run DL benchmarks on the full SoC.

VI. EVALUATION

A. Test Platform

Evaluation is performed on the system described in Table IV. The SoC in which Ncore is integrated serves as the host processor. Note that the x86 cores' L3 caches can be shared with Ncore, but this ability was not used in this evaluation.

Until recently, there has been a dearth of generalized DL inference benchmarks. This changed with the introduction of the MLPerf Inference benchmark [22], which has been embraced in both academic and industry circles. Centaur evaluated the Ncore system with the MLPerf benchmark suite, and has submitted official certified results for MLPerf Inference v0.5.

The 0.5 version of MLPerf Inference consists of five benchmarks: MobileNet-V1, ResNet-50-V1.5, SSD-MobileNet-V1, SSD-ResNet-34, and Google's Neural Machine Translation (GNMT). We targeted four of these benchmarks, deciding not to run SSD-ResNet-34 due to schedule constraints. The basic characteristics of the evaluated benchmarks are shown in Table V. The number of MACs executed for the GNMT benchmark will vary as a function of the sentence being translated. We have characterized the number of MACs executed by GNMT for an input and output sentence length of 25 words.

MLPerf provides several modes of testing for each benchmark. We chose to focus on the SingleStream (latency) and Offline (throughput) benchmark modes for all benchmarks except GNMT. Due to the low arithmetic intensity (specifically MACs/weight) and the large number of weights required for GNMT, we would be severely memory-bound executing GNMT in SingleStream mode. As a result, we only ran GNMT in Offline mode with a batch size of 64 to increase the arithmetic intensity.

TABLE V
EVALUATED BENCHMARK CHARACTERISTICS.

Model	Input Type	Total MACs	Total Weights	MACs/ weight
MobileNet-V1	Image	0.57B	4.2M	136
ResNet-50-V1.5	Image	4.1B	26.0M	158
SSD-MobileNet-V1	Image	1.2B	6.8M	176
GNMT	Text	3.9B	131M	30

TABLE VI
TYPES OF MLPERF SUBMITTERS.

Type	Submitter
Chip vendors	Centaur, Intel, NVIDIA, Qualcomm
Cloud services	Alibaba, Google
Systems (Intel-based)	DellEMC, Inspur, Tencent
Chip startups	FuriosaAI, Habana Labs, Hailo

Several companies have submitted results to the MLPerf Inference v0.5 benchmark. The companies can be divided into the categories shown in Table VI: chip vendors, cloud services, system integrators, and chip startups. We compare Ncore's results with the other chip vendor submissions, as these represent targets from the most similar industry segment. Ncore's results fall under MLPerf's "Preview" system category, since Ncore was not available for rent or purchase. The other chip vendor submissions compared here were submitted as either "Preview" or "Available" systems.

B. Performance

Ncore's MLPerf results are compared against those of other chip vendors[1], with latency reported in Table VII and throughput reported in Table VIII.

As seen in Figures 11 and 12, the reported results span multiple orders of magnitude. Ncore achieves the lowest latency in MobileNet-V1 ($0.33\,\mathrm{ms}$) and ResNet-50-V1.5 ($1.05\,\mathrm{ms}$), with near-best latency in SSD-MobileNet-V1 ($1.54\,\mathrm{ms}$).

Ncore yields substantial speedup over the Intel i3 1005G1 and Qualcomm SDM855 QRD, while offering performance on par with the NVIDIA Jetson AGX Xavier SoC. MobileNet-V1 throughput for Xavier ($6521\,\mathrm{IPS}$) and Ncore ($6042\,\mathrm{IPS}$) are within 8% of each other, while Xavier's ResNet-50-V1.5 result shows a 77% speedup over Ncore. Ncore suffers a relatively low SSD-MobileNet-V1 score that is multiple-times slower than anticipated due to our software still being relatively immature at the time of our MLPerf submission. This slowdown was expected due to batching effects that will be discussed in section VI-C.

The Intel CLX 9282 and Intel NNP-I 1000 submissions achieve the best throughput, partly helped by instantiating two

[1]MLPerf v0.5 Inference Closed SingleStream and Offline. Retrieved from www.mlperf.org 27 January 2020, entries 0.5-22, 0.5-23, 0.5-24, 0.5-28, 0.5-29, 0.5-32, 0.5-33. Centaur Ncore and (2x) Intel NNP-I 1000 submissions are from MLPerf "Preview" category, all others are from MLPerf "Available" category. MLPerf name and logo are trademarks.

TABLE VII
LATENCY OF INTEGRATED CHIP VENDOR MLPERF SUBMISSIONS[1] (MILLISECONDS).

Target System	MobileNet V1	ResNet50 V1.5	SSD MobileNetV1	GNMT
Centaur Ncore	0.33	1.05	1.54	-
NVIDIA AGX Xavier	0.58	2.04	1.50	-
Intel i3 1005G1	3.55	13.58	6.67	-
(2x) Intel CLX 9282	0.49	1.37	1.40	-
(2x) Intel NNP-I 1000	-	-	-	-
Qualcomm SDM855 QRD	3.02	8.95	-	-

TABLE VIII
THROUGHPUT OF INTEGRATED CHIP VENDOR MLPERF SUBMISSIONS[1] (INPUTS PER SECOND).

Target System	MobileNet V1	ResNet50 V1.5	SSD MobileNetV1	GNMT
Centaur Ncore	6,042.34	1,218.48	651.89	12.28
NVIDIA AGX Xavier	6,520.75	2,158.93	2,485.77	-
Intel i3 1005G1	507.71	100.93	217.93	-
(2x) Intel CLX 9282	29,203.30	5,965.62	9,468.00	-
(2x) Intel NNP-I 1000	-	10,567.20	-	-
Qualcomm SDM855 QRD	-	-	-	-

targets per system (i.e., two CLX 9282 processors, and two NNP-I 1000 adapters).

Note that each Intel NNP-I 1000 adapter [27] consists of 12 inference compute engines (ICE), each ICE being 4096 bytes wide. A single ICE is the same width as Ncore, making for a natural comparison between the two. The 2x NNP-I 1000 submission uses two NNP-I 1000 adapters, giving a total of 24 ICEs. The 2x NNP-I 1000 achieved 10,567 IPS on ResNet-50-V1.5, which equates to 440 IPS per 4096-byte ICE. This is compared to Ncore's ResNet-50-V1.5 score of 1218 IPS, which is 2.77x higher than a single 4096-byte ICE[2].

Each Intel CLX 9282 processor [11] contains 56 Xeon cores with the latest vector neural network instruction (VNNI) extension. The 2x CLX 9282 submission consists of two of these processors, yielding 112 Xeon cores with VNNI running at a 2.6GHz base frequency. 2x CLX 9282 achieves 5966 IPS on ResNet-50-V1.5, or the equivalent of 53.3 IPS per core. Ncore's throughput is equivalent to approximately 23 of these VNNI-enabled Xeon cores[2].

Centaur was the only chip vendor to submit results for the relatively memory-intensive GNMT. GNMT was run only in Offline mode, and using the more fully-featured TensorFlow framework rather than TensorFlow-Lite due to framework compatibility with the model. Due to time constraints and the use of TensorFlow instead of TensorFlow-Lite, we implemented GNMT using bfloat16 rather than 8-bit integer. We anticipate Ncore's GNMT throughput (12.28 IPS) to increase significantly as Ncore's software stack continues to mature. Centaur did not submit a GNMT SingleStream (latency) result.

[2]Normalized per-core performance is not the primary metric of MLPerf.

Fig. 11. Latency of integrated chip vendor MLPerf submissions[1] (milliseconds, log-scale).

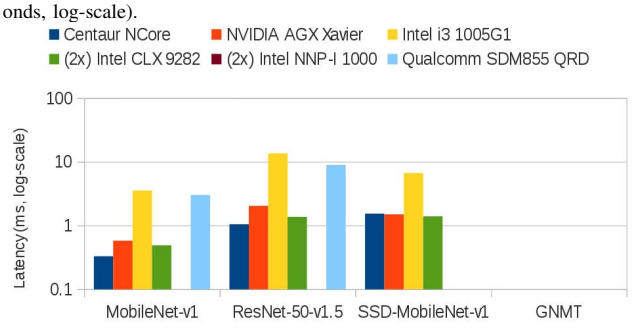
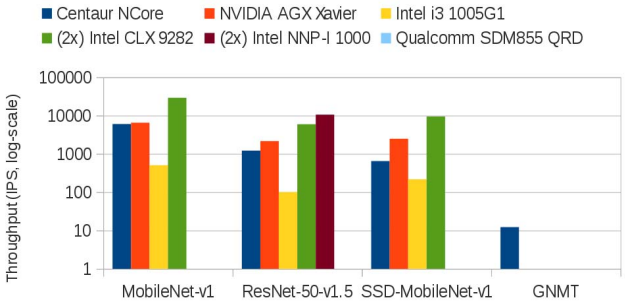


Fig. 12. Throughput of integrated chip vendor MLPerf submissions[1] (inputs per second, log-scale).



C. Batching Effects

Ncore typically executes a single batch at a time for CNN networks. However, Ncore's MLPerf Offline results were run with multi-batched inputs to better hide the latency of the x86 portions of the workload. This x86 code batching accounts for the speedup seen in Ncore's MLPerf Offline throughput results compared to the effective MLPerf SingleStream throughput (i.e., the reciprocal of SingleStream latency).

Batching the x86 portions leads to varying overall speedup, depending on each network's proportion of x86 work to Ncore work. Table IX shows this Ncore versus x86 workload breakdown for CNN models. The Ncore portion of the latency was measured with Ncore's built-in logging capability, and subtracted from the total latency from our MLPerf SingleStream results to determine the x86 portion of the total latency.

The Ncore portion of the workload typically consists of executing the network, while the x86 portion consists of preprocessing, postprocessing, framework (TensorFlow-Lite) overhead, and benchmark (MLPerf) overhead.

MobileNet-V1 is a relatively small network. Despite being a small network, the proportion of x86 work, consisting mostly of pre- and post-processed data movement, still makes up 67%

TABLE IX
PROPORTIONS OF X86 AND NCORE WORK IN TOTAL LATENCY.

Model	Total latency	Ncore portion	x86 portion
MobileNet-V1	0.33ms	0.11ms (33%)	0.22ms (67%)
ResNet-50-V1.5	1.05ms	0.71ms (68%)	0.34ms (32%)
SSD-MobileNet-V1	1.54ms	0.36ms (23%)	1.18ms (77%)

of the total single-batch latency. Thus, the batched throughput (6042 IPS) produces a large speedup – nearly 2x speedup over the single-batch's maximum throughput ($1/0.33\,\text{ms} = 3030\,\text{IPS}$) for MobileNet-V1.

ResNet-50-V1.5, on the other hand, has very different Ncore versus x86 proportions of the workload. ResNet-50-V1.5 has a relatively similar amount of x86 work as MobileNet-V1, but has a much larger Ncore workload. As such, batched ResNet-50 throughput (1218 IPS) yields only 1.3x speedup over the single-batch throughput ($1/1.05\,\text{ms} = 952\,\text{IPS}$).

Our SSD-MobileNet-V1 throughput results (651 IPS) showed negligible speedup relative to the single-batch latency result ($1/1.54\,\text{ms} = 649\,\text{IPS}$), because we ran SSD-MobileNet-V1 throughput tests with only single batches. In Table IX, SSD-MobileNet-V1 has a much larger x86 latency than MobileNet-V1 or ResNet-50-V1.5. This difference is largely attributed to SSD's non-maximum suppression (NMS) operation which is executed on x86. TensorFlow-Lite's implementation of the NMS operation does not support batching. Manually adding batching support to TensorFlow-Lite's NMS operation at the end of the network was infeasible for our schedule as we approached the MLPerf submission deadline. We only report our official MLPerf results here, although we have unofficially achieved 2x throughput speedup[3] for SSD-MobileNet-V1 in the weeks following the MLPerf deadline and expect 3x speedup as we continue to refine our software.

Theoretically, all of the x86 portion of any network could be hidden by Ncore's latency, given enough x86 cores executing concurrently with Ncore. For networks with a small x86 proportion of work, like ResNet-50-V1.5, fewer x86 cores are needed to hide the x86 latency. Networks with larger x86 proportions would need more x86 cores to hide the latency. Figure 13 shows this relationship – the expected maximum throughput as x86 core count increases, based on the data from Table IX. Figure 14 shows the actual throughput achieved as a function of x86 core count (reduced core counts not submitted as official MLPerf results).

To achieve maximum throughput, we would expect to need only two x86 cores to hide the x86 workload behind Ncore's ResNet-50-V1.5 latency and achieve maximum throughput. MobileNet-V1 would need four x86 cores, and SSD-MobileNet-V1 would need five cores to achieve maximum throughput. The observed MobileNet-V1 and ResNet-50-V1.5 results in Figure 14 do exhibit high throughput, although they appear to become limited by other x86 overhead not accounted for in either the TensorFlow-Lite or MLPerf frameworks. As previously discussed, our MLPerf results for SSD-MobileNet-V1 do not track higher due to a lack of time to optimize our software before submitting performance numbers.

[3]Result not verified by MLPerf.

Fig. 13. Expected MLPerf maximum throughput (IPS) with varying x86 core count. Assumes batching so that x86 overhead runs concurrently with Ncore, hiding the x86 latency.

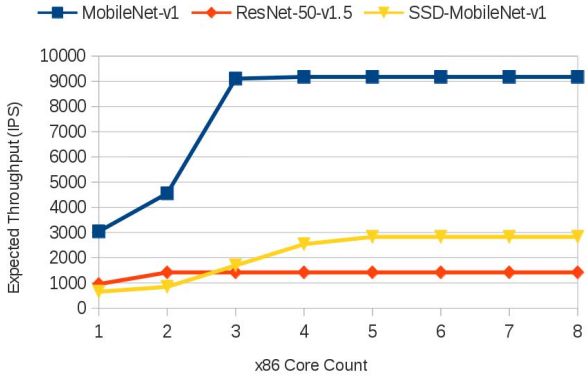
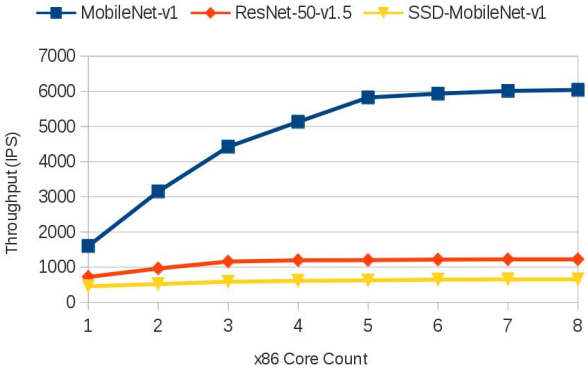


Fig. 14. Observed MLPerf throughput (IPS) with varying x86 core count[3]. Batching was used for MobileNet-V1 and ResNet-50-V1.5 to run x86 overhead concurrently with Ncore, hiding x86 latency.



We further tested these convolutional networks in our own low-level, low-overhead frameworks, which do not use either TensorFlow-Lite or MLPerf. In this setting, we see results in hardware that track much more closely with the theoretical expected throughput in Figure 13. This is true for all three networks: MobileNet-V1, ResNet-50-V1.5, and SSD-MobileNet-V1. We leave our low-level framework analysis brief, as this paper's scope is to evaluate our system with a known benchmark platform, and not to compare frameworks.

In practice, we observed that the MLPerf benchmark suite required two dedicated x86 cores to prevent MLPerf's run manager application from interfering with target performance measurements. While MLPerf's run manager can be configured at runtime to use only one thread, doing so had a negative effect on target performance measurement. MLPerf allows replacing the reference python-based run manager application with a custom implementation, and we intend to explore this opportunity in future submissions.

VII. Related Work

The success of DL in a wide range of domains has fueled an explosion in DL accelerator research activity [23]. Researchers have explored FPGA-based accelerators that exploit specialization at synthesis time, systolic-array and SIMD-based ASIC accelerators, accelerators that focus on exploiting sparsity, and analog and neuromorphic computing approaches. While some accelerators have focused on particular DL application domains, Ncore was designed to be flexible and offer strong performance on a variety of DL workloads.

DL workloads can vary wildly in the dimensions of the tensors and the required precision to achieve acceptable accuracy [20]. Due to the flexible nature of FPGAs, a large body of research has focused on developing FPGA-based DL accelerators [25] [21] [24] [17] [28] [19]. By deploying specialized hardware designs on a per-model basis, parameters such as matrix-engine dimensions and data type can be optimized per-model. Microsoft's Project Brainwave demonstrated low-latency inference for small batches through hardware specialization and *model pinning* [7]. Model pinning is a strategy that distributes a model's weights across as many FPGA nodes as necessary to fit all the weights in on-chip memory and serve models requiring high memory bandwidth in real-time. Implemented on an Arria 10 1150 FPGA, Brainwave achieved 559 images per second throughput and 2.17ms latency for batch-1 ResNet-50.

A number of ASIC-based accelerators have been proposed from academia. Many of these designs use novel dataflow architectures, circuits, or memory hierarchies to achieve low power consumption for vision-based DL models. The DaDianNao accelerator utilized model pinning and spatially distributed eDRAM to achieve orders-of-magnitude improvement in performance and power compared to a GPU [6]. The Eyeriss project demonstrated a novel spatial array architecture that maximized data reuse and could execute AlexNet inference at 35 images/second while consuming 10X less energy than a mobile GPU [4]. The follow-on work, Eyeriss v2, focused on exploiting sparsity in weights and data through a novel on-chip network [5]. Eyeriss v2 demonstrated increased energy efficiency and performance for sparse DL models. The accelerator presented in this work includes a hardware decompression engine for sparse weights, but does not exploit data sparsity.

As commercial interest in DL has exploded, a number of DL accelerators have been announced from startups, silicon vendors, and large cloud service providers. Google's TPUv1 focused on inference and the follow-on TPUv2 and TPUv3 support both inference and training. Unlike Ncore, the TPUs utilize systolic arrays and rely on large batch sizes to reach peak throughput [26]. Several startups have focused on accelerators for DL training, as opposed to Ncore's focus on inference, including GraphCore, Wave Computing, and Cerebras [18]. Nvidia's NVDLA is a specialized, low-power accelerator for DL inference on vision models. The Habana Goya is a general-purpose DL inference accelerator that combines many independent programmable VLIW SIMD processors with a large matrix engine. Intel announced the NNP-I 1000, which combines multiple x86 cores and DL inference accelerators on a single SoC and requires an additional external host processor to operate. Performance comparisons between the Centaur Ncore system, an Intel NNP-I-based system, and an NVidia NVDLA-based system were discussed in Section VI.

VIII. Conclusion

This paper presented the Ncore DL coprocessor with eight x86 cores, which demonstrated the lowest latency of all submissions to MLPerf's Inference v0.5 closed division for one benchmark, as well as competitive latency and throughput for several other benchmarks. The ability to submit competitive scores to the MLPerf benchmarks with a relatively small team and short time-frame is a testament to our design decisions, risk minimization with a more-programmable SIMD architecture, and accelerator integration within a server-class x86 SoC. The achieved performance is the result of tight integration between the Ncore accelerator, the x86 cores and the SoC's memory system. The software libraries were co-designed alongside the development of the SIMD ISA, resulting in efficient code generation compared to a more conventional compiler-based code generation approach. A graph-based compiler was developed to perform optimal scheduling of data movement and data layout at a graph-level, rather than per-operation. By extending existing DL frameworks such as TensorFlow-Lite, we were able to seamlessly integrate our accelerator's software stack and focus on optimizing the performance of critical operations.

Ncore has proved to be a competitive entrant in the DL inference arena, thanks to effective design efforts that focused on low risk, high flexibility, high value, and significant CHA infrastructure reuse. We anticipate Ncore performance will continue to increase through ongoing optimization of Ncore's software stack, such as targeting more DL frameworks, implementing more batching support, leveraging Ncore's ability to use its DMA engines to access CHA's L3 caches, and exploring methods to further hide third-party framework latency overheads. While the software stack presented here scheduled and optimized x86 and Ncore operations independently, MLIR is an exciting new project that explores a unified compiler framework for heterogeneous hardware [16]. We are exploring ways to leverage MLIR to provide a unified compiler stack for co-optimizing the scheduling and code-generation between x86 and Ncore in order to increase concurrency and further leverage the availability of x86 cores.

IX. Acknowledgments

The Ncore coprocessor team ramped up to eight full-time engineers covering RTL, DV, and all software efforts at the time of submission to the MLPerf Inference v0.5 benchmark suite. We acknowledge that this effort would not have been possible without the support and contributions from other individuals within the organization: John Bunda, Stephan Gaskins, CJ Holthaus, Al Loper, Terry Parks, Doug Reed, Dan Weigl, and all other Centaurians.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[2] "Announcing hanguang 800: Alibaba's first ai-inference chip," https://www.alibabacloud.com/blog/announcing-hanguang-800-alibabas-first-ai-inference-chip_595482, Alibaba Group, accessed: 2019-10-28.

[3] "Announcing aws inferentia: Machine learning inference chip," https://aws.amazon.com/about-aws/whats-new/2018/11/announcing-amazon-inferentia-machine-learning-inference-microchip/, Amazon Inc, accessed: 2018-11-28.

[4] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.

[5] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[6] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 609–622.

[7] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caufield, E. S. Chung, and D. Burger, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 1–14.

[8] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[9] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, Conference Track Proceedings*, May 2016. [Online]. Available: http://arxiv.org/abs/1510.00149

[10] "Intel 64 and ia-32 architectures optimization reference manual," Intel Corporation, September 2019. [Online]. Available: https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual

[11] "Intel xeon platinum 9282 processor," Intel Corporation, 2019. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html

[12] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[13] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. K. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. T. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

[14] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of BFLOAT16 for deep learning training," *CoRR*, vol. abs/1905.12322, 2019. [Online]. Available: http://arxiv.org/abs/1905.12322

[15] H. Kwon, L. Lai, T. Krishna, and V. Chandra, "HERALD: optimizing heterogeneous DNN accelerators for edge devices," *CoRR*, vol. abs/1909.07437, 2019. [Online]. Available: http://arxiv.org/abs/1909.07437

[16] C. Lattner and J. Pienaar, "Mlir primer: A compiler infrastructure for the end of moore's law," 2019.

[17] Y. Ma, Y. Cao, S. Vrudhula, and J. sun Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, Inc, February 2017, pp. 45–54.

[18] C. Nicol, "A dataflow processing chip for training deep neural networks," in *2017 IEEE Hot Chips 29 Symposium (HCS)*. IEEE, 2017.

[19] K. Ovtcharov, O. Ruwase, J. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Toward accelerating deep learning at scale using specialized hardware in the datacenter," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015, pp. 1–38.

[20] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. S. Khudia, J. Law, P. Malani, A. Malevich, N. Satish, J. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. M. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, and M. Smelyanskiy, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," *CoRR*, vol. abs/1811.09886, 2018. [Online]. Available: http://arxiv.org/abs/1811.09886

[21] A. Podili, C. Zhang, and V. Prasanna, "Fast and efficient implementation of convolutional neural networks on fpga," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2017, pp. 11–18.

[22] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," *arXiv*, vol. abs/1911.02549, 2019. [Online]. Available: http://arxiv.org/abs/1911.02549

[23] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators," in *IEEE-HPEC conference*, September 2019.

[24] Y. Shen, M. Ferdman, and P. Milder, "Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 93–100.

[25] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.

[26] Y. Wang, G. Wei, and D. Brooks, "Benchmarking tpu, gpu, and CPU platforms for deep learning," *CoRR*, vol. abs/1907.10701, 2019. [Online]. Available: http://arxiv.org/abs/1907.10701

[27] O. Wechsler, M. Behar, and B. Daga, "Spring hill (nnp-i 1000) intel's data center inference chip," in *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE, 2019, pp. 1–12.

[28] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, February 2015. [Online]. Available: https://www.microsoft.com/en-us/research/publication/optimizing-fpga-based-accelerator-design-for-deep-convolutional-neural-networks/