A Chipyard Comparison of NVDLA and Gemmini

Abraham Gonzalez
abe.gonzalez@berkeley.edu
University of California, Berkeley

Charles Hong
charleshong@berkeley.edu
University of California, Berkeley

ABSTRACT

With the explosion of machine learning being used in everyday life, machine learning accelerators have undergone significant development, becoming more performant and energy efficient. With the abundance of new accelerators comes the need to measure and compare them against the state-of-the-art on common benchmarks. In this project, we first integrate the open-source NVIDIA Deep Learning Accelerator (NVDLA) into Chipyard then compare it against the open-source Gemmini Systolic Array Generator. We show that the NVDLA is up to 2.93x faster than a default Gemmini configuration and up to 3.77x faster on an equivalently sized Gemmini configuration running ResNet-50. By having NVDLA implemented within Chipyard, we allow further work and analysis comparing Gemmini and NVDLA while also providing an industry-level accelerator for others to attach to their own SoC designs.

1 INTRODUCTION

With the rising importance of machine learning in systems like self-driving cars, drones, and mobile devices, companies and research teams around the world have introduced accelerators for machine learning workloads, in particular deep neural networks (DNNs). These accelerators are incorporated into system-on-chips (SoCs) and are generally applied to DNN inference operations, which are computationally intensive with large memory requirements [11]. Two such accelerators are the NVIDIA Deep Learning Accelerator (NVDLA), open-sourced in 2017, and Berkeley's Gemmini, which came later in 2019. They both serve the purpose of accelerating deep learning workloads, but approach the problem in different ways. A few of the differences are:

Convolution: Convolutional neural networks are a type of DNN of particular interest to hardware designers because of their computational intensity and the ability to reduce convolution to matrix multiplication [15]. NVDLA computes convolutions directly and uses an adder tree structure for spatial computation [20]. Gemmini reduces convolutions to matrix multiplication and implements a systolic array similar to Google's TPU [14, 17].

SoC integration: Gemmini is integrated into the Rocket Chip environment using Rocket Chip Coprocessor (RoCC) interface custom instructions. NVDLA is programmed using MMIO register accesses. This is discussed further in Section 4.

Architecture: NVDLA is an industry product that consists of several different engines and implements a number of performance optimizations, discussed further in Sections 3 and 5. Gemmini is a research tool and as such, does not implement as many optimizations as NVDLA.

To measure and quantify these differences, we wanted to integrate NVDLA and Gemmini into an environment where they can be compared, namely Chipyard. This will also facilitate further study of NVDLA and deep learning accelerators in future work. The contributions are as follows:

- Integration of the NVDLA into Chipyard while supporting its default configurability
- Wrapped FireMarshal workloads to easily build, add, and run inference tasks
- Preliminary evaluation of NVDLA runtimes on ResNet-50, AlexNet, and YOLO3
- Comparison of various Gemmini configurations against the NVDLA accelerator

The remainder of this paper is organized as follows: In Section 2 we discuss the open-source hardware ecosystem and prior integration work. In Section 3 we give a brief overview of the NVDLA. In Section 4 we discuss the changes/additions to Chipyard in order to run the accelerator. In Section 5 we do a preliminary evaluation of NVDLA and compare it against the Gemmini generator. In Section 6 we discuss future work and conclude in Section 7.

2 BACKGROUND AND RELATED WORK

The NVDLA was first open-sourced in late 2017 with the goal being to release a usable industry level accelerator that was scalable and came with a complete package of HW, SW, and verification material [6]. Since that point, multiple projects have either written software projects based around it, written networks to target it, and measured it for autonomous vehicles, and much more [13, 19–21]. Additionally, with the abundance of open-source frameworks using RISC-V to build SoCs, NVDLA has been a go-to accelerator for open-source machine learning SoCs. NVDLA has been integrated into projects like Princeton's Bring-Your-Own-Core (BYOC) and Columbia's ESP Platform [9, 10]. Each of these allows you to add an NVDLA to the system with a variety of other accelerators or RISC-V cores.

Before integration into the BYOC and ESP Platforms, NVDLA was integrated into the FireSim project, a FPGA-accelerated cycle-accurate hardware simulation platform [12, 18]. This involved writing a NVDLA wrapper in Chisel3 to blackbox the RTL, but also required extensive modifications to the FireSim backend so that external Verilog could be clock-gated. While this clock-gating work was eventually used in further FireSim projects and was mainstreamed [16], the main integration work for NVDLA was left alone. Recently the Chipyard framework was introduced, supporting a wide variety of open-source cores, accelerators, and tooling IP (including FireSim) making integrating NVDLA into it a logical next step [8]. Additionally, Chipyard has its own machine learning accelerator, Gemmini, targetting IoT workloads making it an ideal comparison to the NVDLA [14].

3 NVDLA OVERVIEW

NVDLA consists of several engines as shown in Figure 1. They target the four main operations that make up deep learning inference [3]:

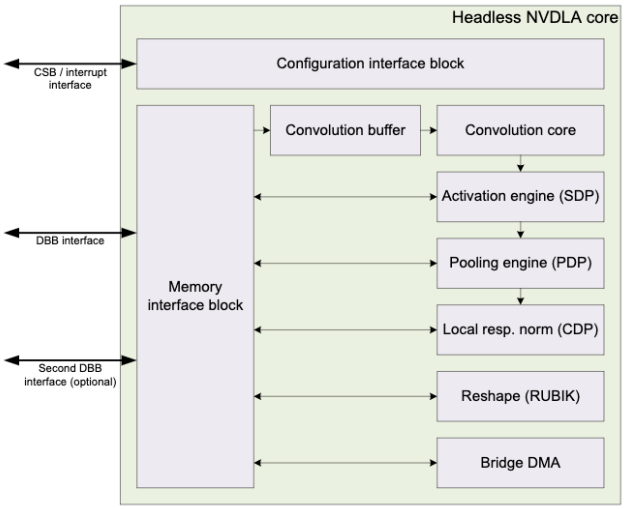

Figure 1: NVDLA Block Diagram [2]

- Convolution: The convolution engine has its own on-chip SRAM (the convolution buffer) and can be parameterized for four different methods of doing convolution - direct, image-input, Winograd, and batching convolution.
- Activation: The single data point processor (SDP) applies linear or non-linear activation functions to individual data points
- Pooling: The planar data processor (PDP) handles pooling.
- Normalization: The multi-planar data processor (CDP) is built specifically to apply the local response normalization function [21].

In addition to these four engines, NVDLA provides a data reshape engine and a bridge DMA engine for accelerating data movement between system DRAM and on-chip buffers. Most of these six units have "ping-pong" register groups, meaning the next operation can be programmed on one group of registers in an engine while it is operating on a second group of registers that has already been programmed [20]. This hides the CPU's reprogramming latency. Additionally, large number of NVDLA parameters can be configured, including but not limited to number of MAC units, convolution buffer size, the presence of a second memory bus for enhanced bandwidth bandwidth to DRAM, and precision. A full list of configurable parameters is available in the NVDLA Primer [3]. There are two example configurations included in the open-source hardware repository, called *nv_small* and *nv_large*. Table 1 shows the main differences between the two configurations.

NVIDIA also provides several software tools in its open-sourcing of NVDLA. First, the software stack contains a compiler for Caffe models. The compiler generates a platform-agnostic *loadable* from a Caffe model. Next is a runtime environment, consisting of the User Mode Driver (UMD) and (KMD). The UMD takes in an inference job in *loadable* format and binds input/output tensors to memory locations. It then submits a *task* to the KMD, which executes the *task* by scheduling engine operations and receiving *completion* interrupts from the engines. The KMD's execution can be summarized by the following events [4]:

Program: program one register group for an operation.

Enable: set engine register group as "ready-to-run". doesn't mean operation will run immediately.

Complete: operation on an engine completed and register group is freed.

Finally, NVIDIA provides a virtual platform for simulating NVDLA within an SoC. This tool is meant to accelerate user software development for NVDLA.

4 CHIPYARD INTEGRATION

Large additions into Chipyard consist of correctly connecting the peripheral to the multi-bus hierarchy, properly blackboxing external Verilog, adding software tests, and building FPGA images for testing. Connecting to the multi-bus hierarchy was done in a similar fashion to prior work done in [12]. However, additional work was done to blackbox the external Verilog properly for FireSim emulation and wrap the software stack into the FireMarshal workload management tool for ease of use [7].

4.1 Hardware Modifications

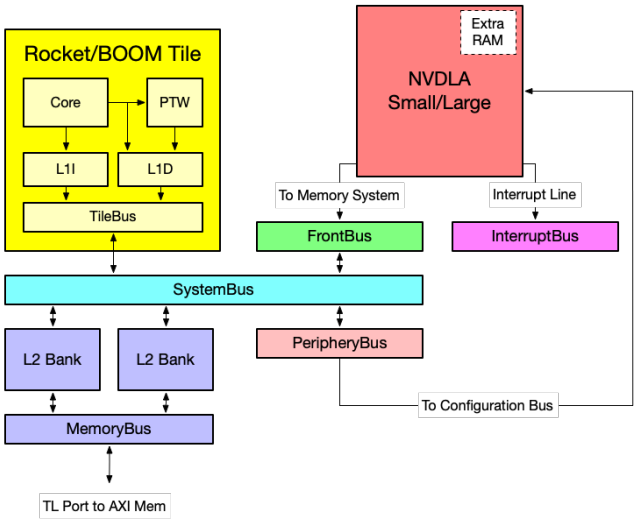

Figure 2: NVDLA Integration into Chipyard

Building off of prior work done in [12], the NVDLA was wrapped and connected to the FrontBus, InterruptBus, and PeripheryBus as shown in Figure 2. To program the NVDLA, a TileLink transaction is sent from the control core (i.e. Rocket or BOOM) over the SystemBus and PeripheryBus into the NVDLA wrapper module. The main NVDLA DBB interface is connected to the FrontBus where memory transactions can eventually go through the memory hierarchy (in the default case, through the shared-L2 into DRAM). The NVDLA interrupt port (used to signal operation completion) connects to the InterruptBus. While these are the default bus connections, these can be modified to obtain higher performance at the cost of larger crossbars. For example, the DBB interface can instead connect to

Configuration	nv_small	nv_large
# of MACs	64	1024
Precision	INT8	FP16/INT16/INT8
Conv. Buffer Size	128KB	512KB
Second Mem. Bus	No	Yes
Winograd Conv. Support	No	Yes
Engine Throughput (# of Pipelines)	1	4
Activation Function Estimation	Scaling	Scaling+LUT
Reshape Support	No	Yes
Bridge DMA Support	No	Yes

Table 1: NVDLA Small and Large Configurations [2]

the SystemBus, so that NVDLA can have a faster connection to the memory hierarchy at the cost of a larger SystemBus crossbar.

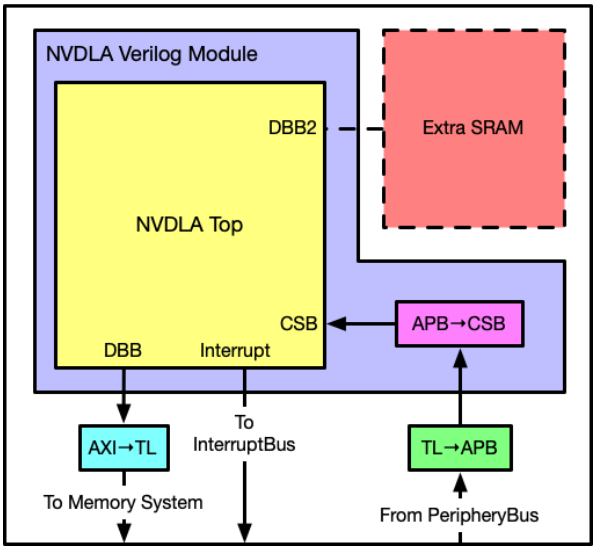

Figure 3: NVDLA Wrapper

All these bus connections are connected to the NVDLA wrapper module written in Chisel3. This module properly wraps the external Verilog into a single clock blackboxed module that can be passed through the FireSim flow. This wrapper module also converts the TileLink transactions into APB transactions then into NVDLA CSB transactions used to program the accelerator and converts the AXI memory transactions into TileLink memory transactions. Finally, if using the Large NVDLA RTL, an extra SRAM can be connected to the second DBB interface. Instead of connecting to the bus hierarchy, this SRAM is local to the NVDLA wrapper module and is only accessible by the NVDLA. Figure 3 shows the NVDLA wrapper with the different module splits, converters, and optional SRAM.

After determining where the NVDLA was located in the Chipyard system and what the module hierarchy was, significant work was done to update the NVDLA to the mainstream NVDLA repository. Since the original NVDLA wrapper repository that [12] used was based on an older NVDLA and Rocket Chip, both the external RTL and the Chisel wrapping it needed to be updated. This included regenerating the new RTL for Small and Large NVDLA, cleaning up unneeded files, solving lint issues, bumping to Chisel3, and updating the DTS. Of these updates, regenerating the RTL for NVDLA proved to be the most troublesome. While the Verilog for NVDLA was lint clean, unneeded dirty Verilog files were also emitted by the NVDLA generator. This in combination with not knowing which files were needed to build the design led to extensive time used trying to solve lint problems that were unnecessary since the files were not needed in the end. Another smaller integration problem that occurred was trying to deal with include directives in the external Verilog. Since the blackbox integration flow for Chipyard doesn't support include directives, a new pre-processing script was created to replace include directives with its respective Verilog file. This was more amenable than running the Verilator pre-processor to replace the files since the Verilator pre-processor ends up removing extra comments which might contain useful pragmas or other constructs [1].

4.2 Software Modifications

The NVDLA software stack consists of a user mode driver (UMD) and a kernel mode driver (KMD). Each had to be updated to build with the RISC-V toolchain and the newer FireSim Linux kernel.

The KMD was based off an older version of the Linux kernel which required updates for the 5.3 kernel. Unlike prior work, the KMD was separated out of the kernel into a out-of-tree module that would be loaded on boot. This was done by using FireMarshal, Chipyard's software management platform, to build the KMD after the kernel binary as built. This required adding a new feature to FireMarshal so that you could execute a script after the binary was created. Additionally, to get the KMD to build, extra Linux kconfiguration flags were added to add DMA shared buffers as well as GEM CMA helpers to the Linux build. This required bugfixes to FireMarshal to properly inherit kconfiguration flags properly. The UMD was in a similar situation to the KMD. First, the prebuilt binaries were out-of-date with the FireSim kernel. In order to rebuild the binaries, external dependencies (specifically *libjpeg.a*) also needed to be rebuilt. This proved to be tricky because there is no documentation on what version of the external dependencies are used and the most up-to-date version wouldn't work. After finding the correct version of the library, the UMD was built properly.

By using the FireMarshal utility, both the KMD and UMD are added automatically to the workload/test. This is demonstrated

Machine Setup	Workload	Inference Cycle Time
4x Rocket + Small NVDLA + L2 + LLC	ResNet-50	1,371,152,204
	AlexNet	2,003,149
4x Rocket + Large NVDLA + L2 + LLC	ResNet-50	1,433,350,701
	AlexNet	1,769,364
	AlexNet (w. random weights)	1,781,644
	YOLO3	20,844,294

Table 2: Overall NVDLA Inference Runtime Results

by adding all the prebuilt NVDLA regressions into the software repository as well as a ResNet-50 test. By using the FireMarshal workloads, users can easily get a working Linux configuration and run something on the NVDLA with minimal to no effort other than to compile the loadable needed.

5 EVALUATION

Evaluation for the project is split into two parts. We first evaluate both the Small and Large configurations of NVDLA running AlexNet, YOLO3, and ResNet-50. We analyze the overall runtime as well as breakdown the individual engine runtimes and stalls. Then, we compare the NVDLA to two main Gemmini configurations, one matching the NVDLA Small configuration and the default Gemmini configuration.

5.1 NVDLA Evaluation

NVDLA evaluation occurred with both the default Small and Large configurations that came from the NVDLA hardware repository using FireSim. Both simulations included Quad-core Rockets, 512KB of L2, and a 4MB simulated LLC. In addition to running the default regression loadables, we compiled ResNet-50 and used YOLO3 from prior work in [12]. Table 2 shows the overall inference runtime results for the different workloads. You can see that ResNet-50 takes a significant amount of time on either Large or Small NVDLA compared to any other workload. This is because ResNet-50 runs a softmax on the network output that is emulated on the host CPU instead of in a NVDLA engine. Additionally, you can see variation between the two AlexNet versions that are run on the Large configuration. This seems to be noise in the measurements (maybe due to using invasive measurements with *printf* statements). Here YOLO3 was pre-compiled into three sub-graphs then run simultaneously on the NVDLA. It runtime was measured by totaling the amount of cycles for each sub-graph.

For the rest of the NVDLA evaluation, we use ResNet-50 on the Small NVDLA configuration to look deeper into the results. Figure 4, shows the enable to completion time of the first few operations of running the inference. You can see that the SDP operation is enabled before its corresponding CONV operation (i.e. CONV0 and SDP1) because the CONV data is streamed to the SDP before written back to memory. Additionally, you can see the overlap between pairs of operations on the same engine (i.e. CONV0 and CONV2). This highlights the register group ping-pong strategy where you can program and enable an operation while another previous operation is resident on the engine. While programming latency is small (on the order of 10K to 30K cycles), this ping-pong strategy hides programming latency when multiple operations are occurring in a row on the same engine. Unfortunately, NVDLA only provides a mechanism to measure from enable to completion for each operation. Thus determining a single operation running on a engine isn't possible, you are only able to get the aggregate with significant post-processing to ignore overlapping cycles.

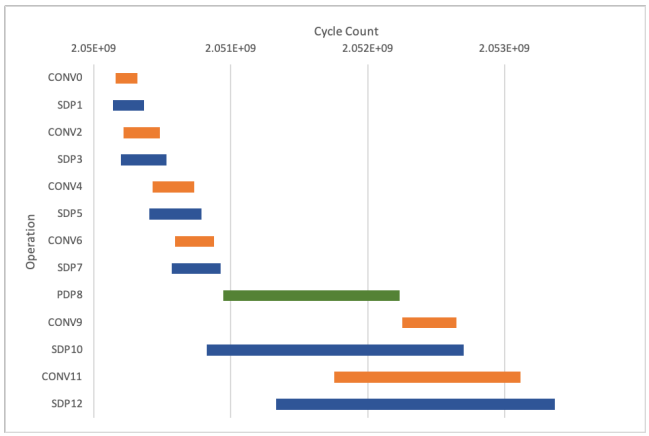

Figure 4: Starting Timeline from ResNet-50 on Small NVDLA

Table 3 shows the individual cycle runtimes for each of the execution engines running ResNet-50. As expected, SDP and CONV are the two main engines that take up the majority of the runtime. Here the SDP runtime encompasses CONV because SDP is enabled before and CONV streams its results into it. When compared to the overall runtime including calculating the softmax, both CONV and SDP only take up 9.2% and 11.7% respectively. Pooling takes a minimal amount of the total combined execution time at around 1% and total inference time at less than 1%. This data shows that the main bottleneck of running ResNet-50 on NVDLA is running the softmax operation on the CPU which could get a sizeable speedup if there was a dedicated engine for it.

In addition to overall engine runtimes, NVDLA also provides performance counters for measuring stalls and read latencies for various operations. As shown in Table 4, a significant amount of time is spent reading out data and weight data from the convolutional buffer. Specifically, the weight read operation stalls for up to 30.57% of the CONV engine runtime. This makes sense since NVDLA is a WS dataflow accelerator and requires stalls on reading weight data. When writing back data, the SDP engine exhibits a

Engine	Total Engine Cycles	% of Combined Engine Time	% of Overall Inference Time
CONV (Convolution Engine)	126,193,914	77.93%	9.20%
SDP (Activation Engine)	160,446,024	99.08%	11.70%
PDP (Pooling Engine)	1,440,188	0.89%	0.11%

Table 3: Small NVDLA Engine Runtime Results for ResNet-50

Engine	Performance Measurement	Cycle Count	% of Engine Runtime
CONV (Convolution Engine)	Data Read Stall	6,312,655	5.00%
	Weight Read Stall	38,571,509	30.57%
	Data Read Latency	6,438,142	5.10%
	Weight Read Latency	0	0%
SDP (Activation Engine)	Write Stall	1,564,106	0.97%
PDP (Pooling Engine)	Write Stall	0	0%

Table 4: Small NVDLA Engine Statistics

minimal amount of stalling up to 1%. The main outlier in the performance counter data is the Weight Read Latency for the CONV engine. While there is no documentation on what the counters exactly mean, it is implied that the read latency should encompass the stall cycles. However, this data shows that there was 0 cycles of latency for reading weight data.

5.2 NVDLA vs Gemmini

To compare NVDLA and Gemmini, we continued targeting the Small NVDLA configuration. In order to isolate only the key differences between the two accelerators, we gathered data using a configuration of Gemmini that was as close as possible to NVDLA's Small configuration in computational and memory capacity. This "Small" configuration of Gemmini includes an equal number of MACs, the same precision, and as close as possible scratchpad and accumulator sizes. As a baseline, we also measured the performance of the default configuration of Gemmini. Table 5 shows the exact parameters that were used for the comparison. We ran a ResNet-50 inference on each of these configurations in FireSim. A summary of the results can be seen in Table 6 and in Figure 5. In total, for ResNet-50 we see that Gemmini Small takes 3.77x longer than NVDLA Small to run the same inference job. Gemmini's default configuration takes 2.93x as long. Much of this slowdown can be attributed to Gemmini requiring *im2col* to run on the CPU before matrix multiplication can occur. *Im2col* made up 46% of Gemmini Small's runtime. NVDLA can directly apply convolution to the image input, and therefore does not need to complete an *im2col* operation [2]. Even without *im2col*, however, Gemmini Small takes 2.02x as many cycles as NVDLA. This includes the NVDLA's additional softmax CPU computation, which takes up a large portion of NVDLA's ResNet-50 runtime. When counting just the matrix multiplication and activation function computation cycles, NVDLA Small is 8.74x and 1.48x faster than the small and and default configurations of Gemmini respectively. We are uncertain why Gemmini took a significantly different amount of time to compute *im2col* for the small and default configurations, but didn't investigate this further as our focus in the time we had was NVDLA.

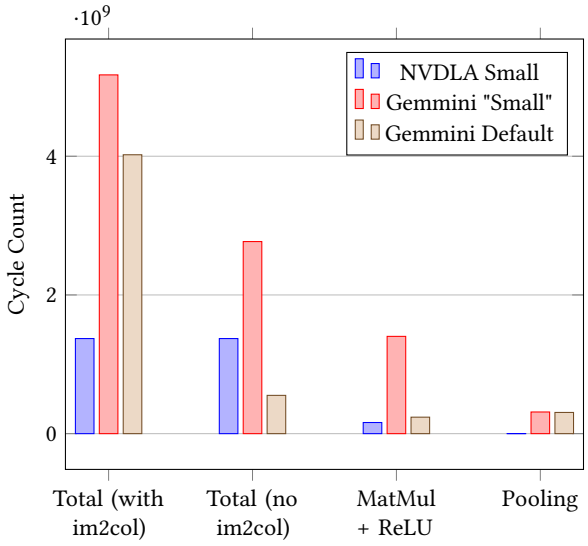

Figure 5: NVDLA vs Gemmini Cycle Count Breakdown

One important difference between the two accelerators is that NVDLA does has a dedicated pooling engine, whereas Gemmini does pooling on the CPU. Because of this, we wanted to isolate the convolution performance of each accelerator as much as possible, as this is the core component of DNN computation. To do this, we looked at the number of cycles until the first pooling layer. This is essentially the runtime of the first convolutional layer of the DNN. Here, even ignoring *im2col*, Gemmini takes 65.43x as many cycles to complete the convolutional layer as shown in Table 7. This shows that the disparity between accelerators is not only from having to compute operations like pooling on the CPU. Gemmini also has room for improvement in convolution, the core computation that it accelerates. The default configuration of Gemmini, which has 4 times as many MACs and twice the scratchpad and accumulator size of NVDLA Small, is still up to 13.29x slower on the first convolutional layer.

	NVDLA Small	"Small" Gemmini	Default Gemmini
MAC Architecture	Adder Tree	Systolic Array	Systolic Array
# of MACs	64	64	256
Scratchpad Banks	32	32	4
Scratchpad Size	128KB	128KB	256KB
Accumulator Size	36KB	32KB	64KB
Precision	INT8	INT8	INT8
Dataflow	WS	WS+OS	WS+OS

Table 5: NVDLA vs Gemmini Parameters

	NVDLA Small	"Small" Gemmini	Default Gemmini
Total Cycles (including *im2col*)	1,371,152,204	5,173,149,486	4,021,416,734
Total Cycles (no *im2col*)	1,371,152,204	2,769,679,988	551,932,110
MatMul + ReLU Cycles	160,446,024	1,402,658,021	237,572,748
Pooling Cycles	1,440,188	312,270,532	305,640,427

Table 6: NVDLA vs Gemmini Cycle Count Breakdown

	NVDLA Small	"Small" Gemmini	Default Gemmini
Cycles until 1st Pooling Layer (including *im2col*)	786,916	239,312,881	198,360,887
Cycles until 1st Pooling Layer (no *im2col*)	786,916	51,489,687	10,457,630

Table 7: NVDLA vs Gemmini Single Layer Cycle Count

We believe a combination of factors contribute to this large performance gap. First, Gemmini doesn't take advantage of having a large number of scratchpad banks [14]. Although the documentation on it is limited, we hypothesize that NVDLA is able to have greater memory bandwidth through a large number of scratchpad banks, since by default is has 32 and 64 banks for the Small and Large configurations respectively. NVDLA also has a double-buffered accumulator, which further reduces memory bottlenecks by allowing accumulated sums to be both stored to by the convolution MACs and read from by the SDP [5]. NVDLA also implements optimizations such as ping-pong register groups, which reduce the latency between consecutive operations, and weight compression to further reduce memory usage.

6 FUTURE WORK

In the future, testing with more configurations of both NVDLA and Gemmini could help expose the sources of differentiation between the two accelerators. Currently, we compare NVDLA Small to only two different Gemmini configurations, making it difficult to identify how much each optimization in NVDLA contributes to its performance advantage over Gemmini. We could toggle each performance optimization on and off to identify the most advantageous optimizations for Gemmini to implement in future work.

There are also improvements that can be made to the current measurement setup. For example, we could take more accurate measurements of *task* completion time. Currently, we measure the time a *task* takes from *enable* to *completion*, which can block on the *completion* of another, prior task. So, it is difficult to accurately measure how much each individual task contributes to the overall runtime of an inference job. In the future, measuring cycle counts with less invasive techniques, for example out-of-band profiling, may provide slightly more accurate data than the current results, which were measured using *printf* statements in the NVDLA firmware. If more precise performance counters were added to NVDLA, it could help solve this challenge.

This is among a number of potential changes to NVDLA that would enhance our evaluation. In our exploration of NVDLA software, we found that pre-built workloads provided in the software stack are not synced to the latest release of NVDLA hardware. This makes it impossible to run these simple workloads and compare them to Gemmini on the current NVDLA hardware. For example, one measurement that would help quantify the differences between the two accelerators in more detail would be a simple GEMM. Such benchmarks already exist for Gemmini. However, we were unable to measure the performance of a single GEMM on NVDLA because we were only able to build functioning NVDLA loadable inputs from Caffe models. Users would also benefit from NVIDIA providing compiler support for more recent machine learning frameworks such as PyTorch and TensorFlow, which would allow NVDLA to run on a larger number of models available to the public.

7 CONCLUSION

The NVIDIA Deep Learning Accelerator (NVDLA) is an open-source highly performance deep learning accelerator that can now be used with the Chipyard framework [1] to create customized machine learning SoCs. When compared to the Gemmini systolic array generator,

[1]The work can be viewed at https://github.com/ucb-bar/chipyard on the developer branch.

it is up to 3.77x faster running ResNet-50 on an equivalent configuration using the same system setup. Combined with documentation on how to use the NVDLA software stack, this integration enables future Chipyard users and machine learning accelerator developers to add NVDLA to their projects to compare their accelerator implementation with an industry proven commercial accelerator.

REFERENCES

[1] 2006. Verilator. https://www.veripool.org/wiki/verilator.

[2] 2017. NVDLA Hardware Architectural Specification. http://nvdla.org/hw/v1/hwarch.html.

[3] 2017. NVDLA Primer. http://nvdla.org/primer.html.

[4] 2017. NVDLA Software Manual. http://nvdla.org/sw/contents.html.

[5] 2017. NVDLA Unit Description. http://nvdla.org/hw/v1/ias/unit_description.html.

[6] 2017. NVIDIA Deep Learning Accelerator (NVDLA). https://nvdla.org/.

[7] 2019. FireMarshal: Workload Generation Tool for RISC-V Systems. https://firemarshal.readthedocs.io/en/latest/.

[8] 2020. Chipyard. https://chipyard.readthedocs.io/.

[9] 2020. ESP: An Open-Source Platform for Interdisciplinary Research on SoC Design and Programming. In *ASPLOS: Architecture Support for Programming Languages and Operating Systems (ASPLOS 2020)*.

[10] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M Nguyen, Yaosheng Fu, Florian Zaruba, et al. 2020. BYOC: A" Bring Your Own Core" Framework for Heterogeneous-ISA Research. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 699–714.

[11] Mingyu Gao et al. 2017. Tetris: Scalable and efficient neural network acceleration with 3D memory. *OSR* 51, 2 (2017), 751–764.

[12] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2019. Integrating nvidia deep learning accelerator (nvdla) with risc-v soc on firesim. *arXiv preprint arXiv:1903.06495* (2019).

[13] S. Feng, J. Wu, S. Zhou, and R. Li. 2019. The Implementation of LeNet-5 with NVDLA on RISC-V SoC. In *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*. 39–42.

[14] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. 2019. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. *arXiv preprint arXiv:1911.09925* (2019).

[15] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy Of High-Performance Deep Learning Convolutions On SIMD Architectures. (2018).

[16] Qijing Huang, Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Krste Asanovic, and John Wawrzynek. 2019. Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-Acceleration. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, 1–12.

[18] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 29–42.

[19] Wei-Fen Lin, Cheng-Tao Hsieh, and Cheng-Yi Chou. 2019. ONNC-based Software Development Platform for Configurable NVDLA Designs. In *2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, 1–2.

[20] Shenbagaraman Ramakrishnan. 2020. Implementation of a Deep Learning Inference Accelerator on the FPGA. (2020).

[21] G. Zhou, J. Zhou, and H. Lin. 2018. Research on NVIDIA Deep Learning Accelerator. In *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. 192–195.