Theme Article: Hot Chips

Compute Solution for Tesla's Full Self-Driving Computer

Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, and Gagandeep S. Sachdev
Autopilot Hardware, Tesla Motors Inc.

***Abstract*—Tesla's full self-driving (FSD) computer is the world's first purpose-built computer for the highly demanding workloads of autonomous driving. It is based on a new System on a Chip (SoC) that integrates industry standard components such as CPUs, ISP, and GPU, together with our custom neural network accelerators. The FSD computer is capable of processing up to 2300 frames per second, a 21× improvement over Tesla's previous hardware and at a lower cost, and when fully utilized, enables a new level of safety and autonomy on the road.**

PLATFORM AND CHIP GOALS

The primary goal of Tesla's full self-driving (FSD) computer is to provide a hardware platform for the current and future data processing demands associated with full self-driving. In addition, Tesla's FSD computer was designed to be retrofitted into any Tesla vehicle made since October 2016. This introduced major constraints on form factor and thermal envelope, in order to fit into older vehicles with limited cooling capabilities.

Digital Object Identifier 10.1109/MM.2020.2975764
Date of publication 24 February 2020; date of current version 18 March 2020.

The heart of the FSD computer is the world's first purpose-built chip for autonomy. We provide hardware accelerators with 72 TOPs for neural network inference, with utilization exceeding 80% for the inception workloads with a batch size of 1. We also include a set of CPUs for control needs, ISP, GPU, and video encoders for various preprocessing and postprocessing needs. All of these are integrated tightly to meet very aggressive TDP of sub-40-W per chip.

The system includes two instances of the FSD chip that boot independently and run independent operating systems. These two instances also allow independent power supply and sensors that ensure an exceptional level of safety

Published by the IEEE Computer Society

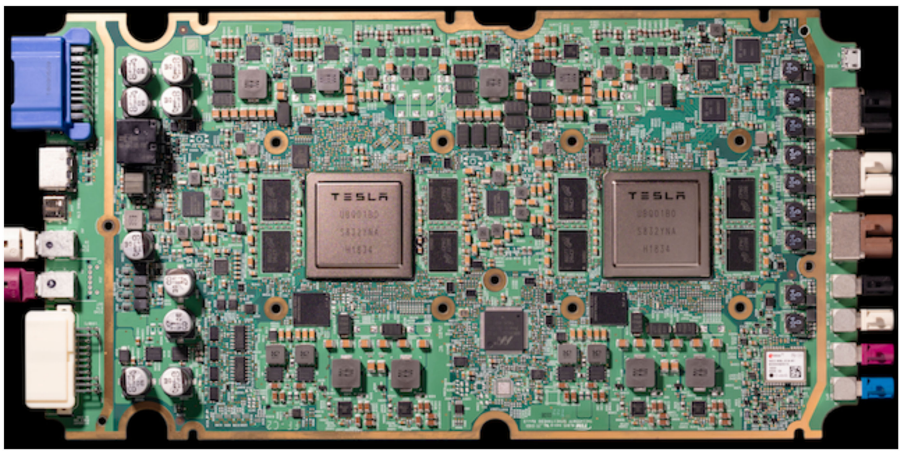


Figure 1. FSD Computer with two Tesla FSD chips in dual configurations including sensors like Cameras.

for the system. The computer as shown in Figure 1 meets the form, fit, and interface level compatibility with the older hardware.

FSD CHIP

The FSD chip is a 260-μm^2 die that has about 250 million gates or 6 billion transistors, manufactured in 14-nm FinFet technology by Samsung. As shown in Figure 2, the chip is packaged in a 37.5 mm $\times$ 37.5 mm Flip Chip BGA Package. The chip is qualified to AEC-Q100 Grade2 reliability standards.

Figure 2(a) shows the major blocks in the chip. We designed the two instances of neural-network accelerator (NNA) from scratch and we chose industry standard IPs such as A72 CPUs, G71 GPU, and ISPs for the rest of the system. Rest of the unmarked area of the chip consists of peripherals, NOC fabrics, and memory interfaces. Each NNA has 32-MB SRAM and 96 $\times$ 96 MAC array. At 2 GHz, each NNA provides 36 TOPs, adding up to 72 TOPs total for the FSD chip.

The FSD SoC, as shown in Figure 2(b), provides general-purpose CPU cores that run most of the autopilot algorithms. Every few milliseconds, new input frames are received through a dedicated image signal processor where they get preprocessed before being stored in the DRAM. Once new frames are available in the main memory, the CPUs instruct the NNA accelerators to start processing them. The accelerators control the data and parameters streaming into their local SRAM, as well as the results streaming back to the DRAM. Once the corresponding result frames have been sent out to the DRAM, the accelerators trigger an interrupt back to the CPU complex. The GPU is available for any postprocessing tasks that might require algorithms not supported by the NNA accelerators.

Chip Design Methodology

Our design approach was tailored to meet aggressive development timelines. To that end, we

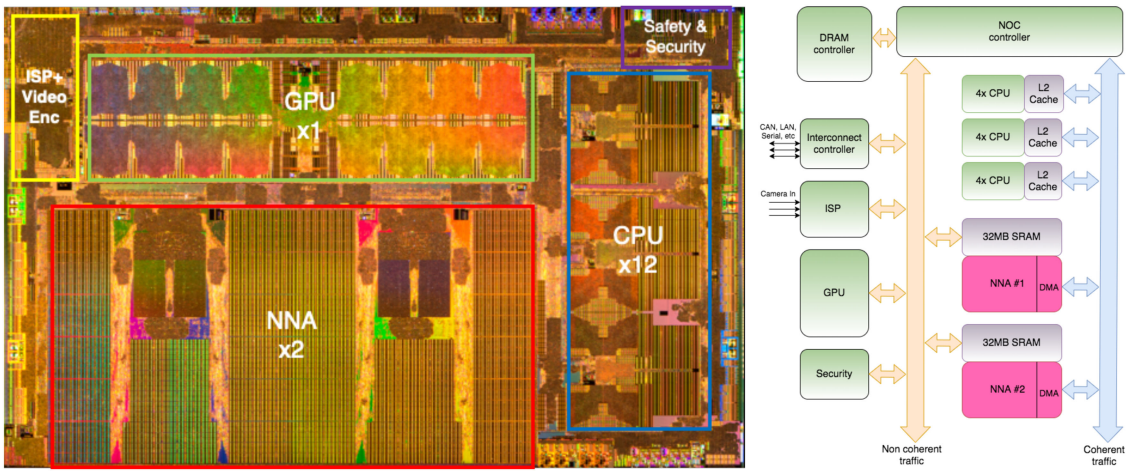
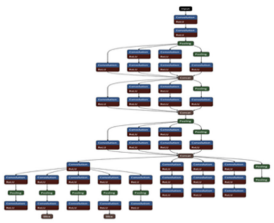


Figure 2. (a) FSD chip die photo with major blocks. (b) SoC block diagram.



Example convolutional neural network

```
A single convolution is a 7 deep nested for loop:
1.  For each Image
2.      For each Output Channel
3.          For each Output X position
4.              For each Output Y position
5.                  For each Input Channel
6.                      For each Input Y within kernelY
7.                          For each Input X within kernelX
```

Operation	MOPS	%
Convolution	34275	98.1
Deconvolution	576	1.6
ReLU	123	0.1
Pooling	13	0.2

Inception like CNNs

Figure 3. Inception network, convolution loop, and execution profile.

decided to build a custom accelerator since this provides the highest leverage to improve performance and power consumption over the previous generation. We used hard or soft IPs available in the technology node for the rest of the SoC blocks to reduce the development of schedule risk.

We used a mix of industry-standard tools and open source tools such as verilator for extensive simulation of our design. Verilator simulations were particularly well suited for very long tests (such as running entire neural networks), where they yielded up to 50× speedup over commercial simulators. On the other hand, design compilation under verilator is very slow, so we relied on commercial simulators for quick turnaround and debug during the RTL development phase. In addition to simulations, we extensively used hardware emulators to ensure a high degree of functional verification of the SoC.

For the accelerator's timing closure, we set a very aggressive target, about 25% higher than the final shipping frequency of 2 GHz. This allows the design to run well below Vmax, delivering the highest performance within our power budget, as measured after silicon characterization.

NEURAL NETWORK ACCELERATOR

Design Motivation

The custom NNA is used is to detect a predefined set of objects, including, but not limited to lane lines, pedestrians, different kinds of vehicles, at a very high frame rate and with modest power budget, as outlined in the platform goals.

Figure 3 shows a typical inception convolutional neural network.[1,2] The network has many layers and the connections indicating flow of compute data or activations. Each pass through this network involves an image coming in, and various features or activations, being constructed after every layer sequentially. An object is detected after the final layer.

As shown in Figure 3, more than 98% of all operations belong to convolutions. The algorithm for convolution consists of a seven deep nested loop, also shown in Figure 3. The computation within the innermost loop is a multiply-accumulate (MAC) operation. Thus, the primary goal of our design is to perform a very large number of MAC operations as fast as possible, without blowing up the power budget.

Speeding up convolutions by orders of magnitude will result in less frequent operations, such as quantization or pooling, to be the bottleneck for the overall performance if their performance is substantially lower. These operations are also optimized with dedicated hardware to improve the overall performance.

Convolution Refactorization and Dataflow

The convolution loop, with some refactoring, is shown in Figure 4(a). A closer examination reveals that this is an embarrassingly parallel problem with lots of opportunities to process the MAC operations in parallel. In the convolution loop, the execution of the MAC operations within the three innermost loops, which determine the length of each dot product, is largely sequential. However, the computation within the three outer loops, namely for each image, for each output channel, for all the pixels within each output channel, is parallelizable. But it is still a hard problem due to the large memory bandwidth requirement and a significant increase in power consumption to support such a large parallel computation. So, for the rest of the paper, we will focus mostly on these two aspects.

First thing to note is that working on multiple images in parallel is not feasible for us. We cannot wait for all the images to arrive to start the

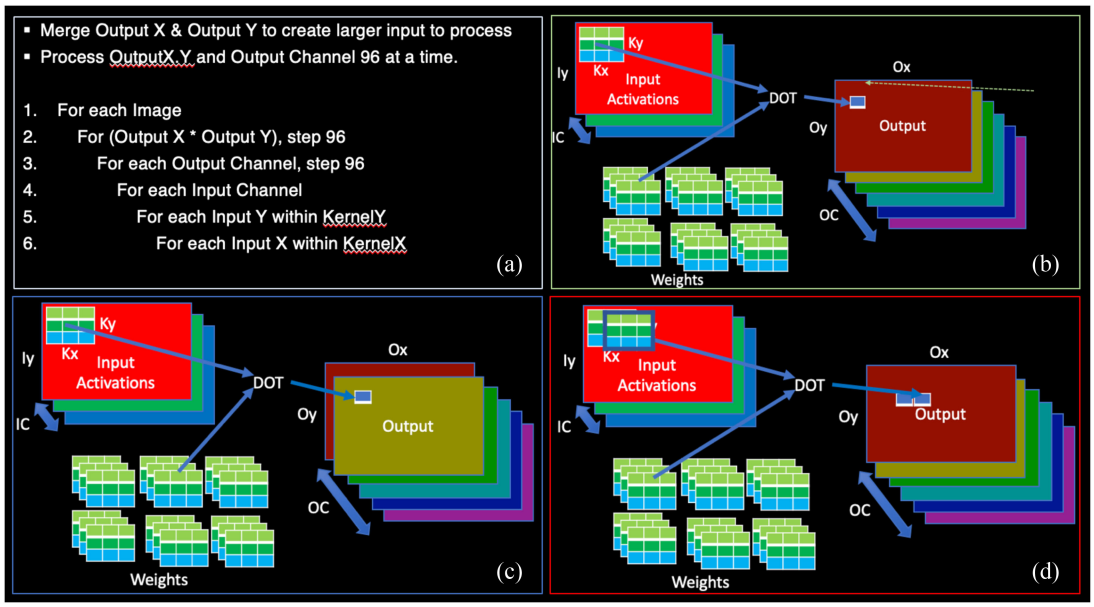


Figure 4. Convolution refactoring and dataflow.

compute for safety reasons since it increases the latency of the object detection. We need to start processing images as soon as they arrive. Instead, we will parallelize the computation across multiple-output channels and multiple-output pixels within each output channel.

Figure 4(a) shows the refactored convolution loop, optimizing the data reuse to reduce power and improve the realized computational bandwidth. We merge the two dimensions of each output channel and flatten them into one dimension in the row-major form, as shown in step (2) of Figure 4(a). This provides many output pixels to work on, in parallel, without losing local contiguity of the input data required.

We also swap the loop for iterating over the output channels with the loop for iterating over the pixels within each output channel, as shown in steps (2) and (3) of Figure 4(a). For a fixed group of output pixels, we first iterate on a subset of output channels before we move onto the next group of output pixels for the next pass. One such pass, combining a group of output pixels within a subset of output channels, can be performed as a parallel computation. We continue this process until we exhaust all pixels within the first subset of output channels. Once all pixels are exhausted, we move to the next subset of output channels and repeat the process. This enables us to maximize data sharing, as the computation for the same set of pixels within all output channels use the same input data.

Figure 4(b)–(d) also illustrates the dataflow with the above refactoring for a convolution layer. The same output pixels of the successive output channels are computed by sharing the input activation, and successive output pixels within the same output channel are computed by sharing the input weights. This sharing of data and weights for the dot product computation is instrumental in utilizing a large compute bandwidth while reducing the power by minimizing the number of loads to move data around.

Compute Scheme

The algorithm described in the last section with the refactored convolution lends itself to a compute scheme with the dataflow as shown in Figure 5. A scaled-down version of the physical 96×96 MAC array is shown in the middle for the brevity of space, where each cell consists of a unit implementing a MAC operation with a single cycle feedback loop. The rectangular grids on the top and left are virtual and indicate data flow. The top grid, called the data grid here, shows a scaled-down version of 96 data elements in each row, while the left grid, called the weight grid here, shows a scaled-down version of 96 weights in each column. The height and width of

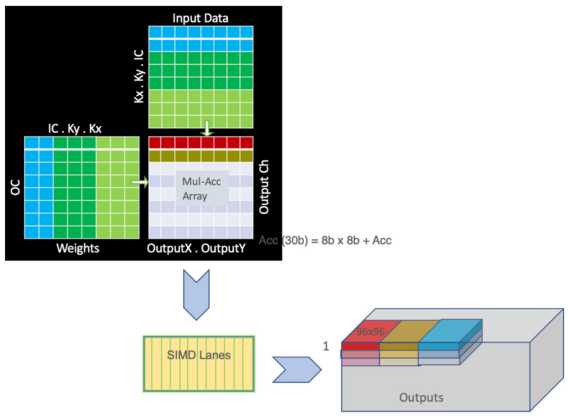


Figure 5. Compute scheme.

the data and weight grids equal the length of the dot-product.

The computation proceeds as follows: the first row of the data grid and the first column of the weight grid are broadcast across all the 96 rows and 96 columns of the MAC array, respectively, over a few cycles in a pipelined manner. Each cell computes an MAC operation with the broadcast data and weight locally. In the next cycle, the second row of the data grid and the second column of the weight grid are broadcast in a pipelined manner, and the MAC computation in each cell is executed similarly. This computation process continues until all the rows and columns of the data and weight grids have been broadcast and all the MAC operations have finished. Thus, each MAC unit computes the dot-product locally with no data movement within the MAC array, unlike the systolic array computations implemented in many other processors.[3,4] This results in lower power and less cell area than systolic array implementations.

When all the MAC operations are completed, the accumulator values are ready to be pushed down to the SIMD unit for post-processing. This creates the first 96 × 96 output slice as shown in Figure 5. The postprocessing, which most commonly involves a quantization operation, is performed in the 96-wide SIMD unit. The 96-wide SIMD unit is bandwidth matched with the 96 element accumulator output associated with each output channel. Accumulator rows in the MAC array are shifted down to the SIMD unit at the rate of one row per cycle. Physically, the accumulator rows shift only once every eight cycles, in groups of eight. This reduces the power consumed to shift the accumulator data significantly.

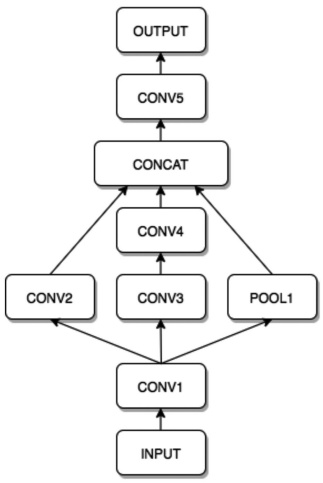
Another important feature in the MAC engine is the overlap of the MAC and SIMD operations. While accumulator values are being pushed down to the SIMD unit for postprocessing, the next pass of the convolution gets started immediately in the MAC array. This overlapped computation increases the overall utilization of the computational bandwidth, obviating dead cycles.

Design Principles and Instruction Set Architecture

The previous section describes the dataflow of our computation. For the control flow, we focused on simplicity and power efficiency. An average application executed on modern out-of-order CPUs and GPGPUs [5–7] burns most of the energy outside of the computational unit to move the instructions and data and in the expensive structures such as caches, register files, and branch predictors.[8] Furthermore, such control structures also introduce significant design complexity. Our goal was to design a computer where almost all the profligate control structures are eliminated, and the execution of the workloads spend all the energy on what matters most for performance, the MAC engine. To that end, we implemented a very flexible yet proficient state machine where all the expensive control flows are built into the state machine, such as loop constructs and fusion.

Another very important performance and power optimization feature is the elimination of DRAM reads and writes during the convolution flow. For inference, the output data of each layer is consumed by dependent layers and can be overwritten. After loading the initial set of activation data, this machine operates entirely from an SRAM embedded in the compute engine itself.

This design philosophy is outlined in the final section. We trade off fine-grain programmability that requires expensive control structures for a flexible state machine with coarse grain programmability. The state machine driven control mechanism lends itself to a very compact yet powerful and flexible ISA. There are only seven main instructions, with a variety of additional control fields that set up the state machine to perform different tasks: data




```
1:  dma_in(src=DRAM_addr0, dest=SRAM_addr0, dest_flag=F0, clear_flag=NONE)

2:  dma_in(src=DRAM_addr1, dest=SRAM_addr1, dest_flag=F1, clear_flag=NONE)

3:  dma_in(src=DRAM_addr2, dest=SRAM_addr2, dest_flag=F2, clear_flag=NONE)

4:  dma_in(src=DRAM_addr3, dest=SRAM_addr3, dest_flag=F3, clear_flag=NONE)

5:  conv(src_a=SRAM_addr0, src_w=SRAM_addr1, dest=SRAM_addr4, src_flag_a=F0, src_flag_w=F1, dest_flag=F4,
clear_flag_a=F0, clear_flag_w=F1, shape_params=...)

6:  dma_in(src=DRAM_addr4, dest=SRAM_addr0, dest_flag=F0, clear_flag=NONE)

7:  conv(src_a=SRAM_addr4, src_w=SRAM_addr2, dest=SRAM_addr5, src_flag_a=F4, src_flag_w=F2, dest_flag=NONE,
clear_flag_a=NONE, clear_flag_w=F2, shape_params=...)

8:  dma_in(src=DRAM_addr5, dest=SRAM_addr1, dest_flag=F2, clear_flag=NONE)

9:  conv(src_a=SRAM_addr4, src_w=SRAM_addr3, dest=SRAM_addr2, src_flag_a=F4, src_flag_w=F3, dest_flag=F5,
clear_flag_a=NONE, clear_flag_w=F3, shape_params=...)

10:  pooling(src_a=SRAM_addr4, dest=SRAM_addr7, src_flag_a=F4, dest_flag=F6, clear_flag_a=F4, shape_params=...)

11: conv(src_a=SRAM_addr2, src_w=SRAM_addr0, dest=SRAM_addr6, src_flag_a=F5, src_flag_w=F0, dest_flag=F3,
clear_flag_a=F5, clear_flag_w=F0, shape_params=...)

12:  conv(src_a=SRAM_addr5, src_w=SRAM_addr1, dest=SRAM_addr0, src_flag_a=F3, src_flag_w=F2, dest_flag=F1,
clear_flag_a=F3, clear_flag_w=F2, shape_params=...)

13:  dma_out(src=SRAM_addr0, dest=DRAM_addr6, src_flag=F1, clear_flag=F6)
```

Figure 6. Typical network program.

movement in and out of the SRAM (DMA-read and DMA-write), dot product (CONVOLUTION, DECONVOLUTION, INNER-PRODUCT), and pure SIMD (SCALE, ELTWISE).

Data movement instructions are 32-byte longs and encode the source and destination address, length, and dependency flags. Compute instructions are 256-byte long and encode the input addresses for up to three tensors (input activations and weights or two activations tensors, output results), tensor shapes and dependency flags. They also encode various parameters describing the nature of computation (padding, strides, dilation, data type, etc.), processing order (row-first or column-first), optimization hints (input and output tensor padding, precomputed state machine fields), fused operations (scale, bias, pooling). All compute instructions can be followed by a variable number of SIMD instructions that describe a SIMD program to be run on all dot-product outputs. As a result, the dot product layers (CONVOLUTION, DECONVOLUTION) can be fused with simple operations (quantization, scale, ReLU) or more complex math functions such as Sigmoid, Tanh, etc.

NETWORK PROGRAMS

The accelerator can execute DMA and Compute instructions concurrently. Within each kind, the instructions are executed in order but can be reordered between them for concurrency. The producer/consumer ordering is maintained using explicit dependency flags.

A typical program is shown in Figure 6. The program starts with several DMA-read operations, bringing data and weights into the accelerator's SRAM. The parser inserts them in a queue and stops at the first compute instruction. Once the data and weights for the pending compute instruction become available in the SRAM, their corresponding dependency flags get set and the compute instruction can start executing in parallel with other queued DMA operations.

Dependency flags are used to track both data availability and buffer use. The DMA-in operation at step 6 overwrites one of the buffers sourced by the preceding convolution (step 5) as shown in Figure 6. Thus, it must not start executing before its destination flag (F0) gets cleared at the end of the convolution. However, using a different destination buffer and flag would allow the DMA-in operation to execute in parallel with the preceding convolution.

Our compiler takes high-level network representations in Caffe format and converts them to a sequence of instructions similar to the one in Figure 6. It analyzes the compute graph and orders it according to the dataflow, fusing or partitioning layers to match the hardware capabilities. It allocates SRAM space for intermediate results and weights tensors and manages execution order through dependency flags.

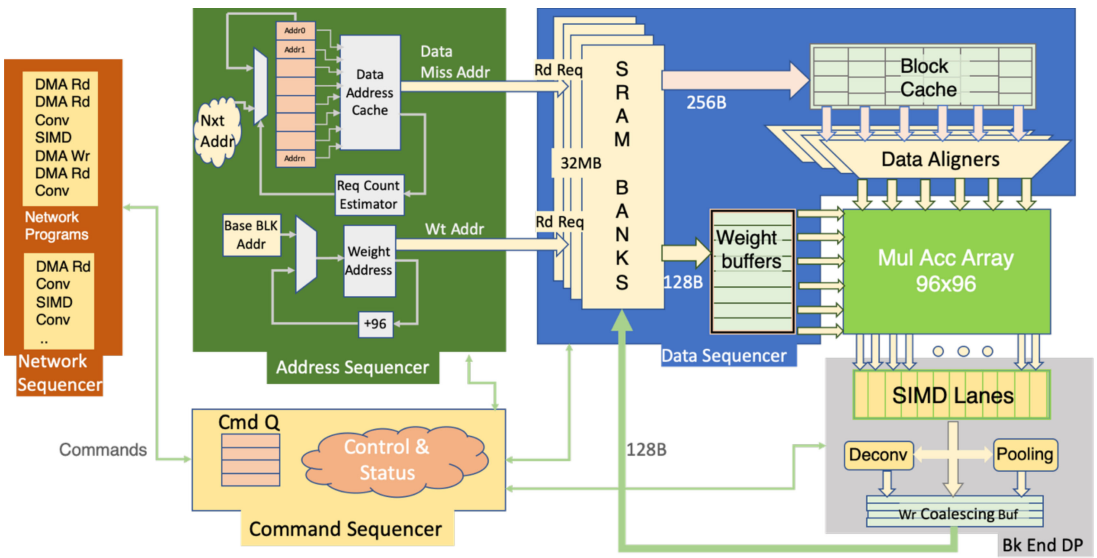


Figure 7. NNA Microarchitecture.

NNA MICROARCHITECTURE

The NNA, as shown in Figure 7, is organized around two main datapaths (dot-product engine and SIMD unit) and the state machines that interpret the program, generate streams of memory requests, and control the data movement into and out of the datapaths.

Dot Product Engine

As described in the "Compute Scheme" section, the dot-product engine is a 96 × 96 array of MAC cells. Each cell takes two 8-bit integer inputs (signed or unsigned) and multiplies them together, adding the result to a 30-bit wide local accumulator register. There are many processors that deploy floating-point operations with single precision or half-precision floating-point (FP) data and weight for inference. Our integer MAC compute has enough range and precision to execute all Tesla workloads with the desired accuracy and consumes an order of magnitude lower power than the ones with FP arithmetic.[9]

During every cycle, the array receives two vectors with 96 elements each and it multiplies every element of the first vector with every element of the second vector. The results are accumulated in place until the end of the dot product sequence when they get unloaded to the SIMD engine for further processing.

Each accumulator cell is built around two 30-bit registers: an accumulator and a shift register. Once a compute sequence is completed, the dot product result is copied into the shift register and the accumulator is cleared. This allows the results to shift out through the SIMD engine while the next compute phase starts in the dot product engine.

SIMD Unit

The SIMD unit is a 96-wide datapath that can execute a full set of arithmetic instructions. It reads 96 values at a time from the dot product engine (one accumulator row) and executes a postprocessing operation as a sequence of instructions (SIMD program). A SIMD program cannot access the SRAM directly and it does not support flow control instructions (branches). The same program is executed for every group of 96 values unloaded from the MAC array.

The SIMD unit is programmable with a rich instruction set with various data types, 8-bit, 16-bit, and 32-bit integers and single-precision floating point (FP32). The instruction set also provides for conditional execution for control flow. The input data is always 30-bit wide (cast as int32) and the final output is always 8-bit wide (signed or unsigned int8), but the intermediate data formats can be different than the input or output.

Since most common SIMD programs can be represented by a single instruction, called FusedReLu (fused quantization, scale, ReLU), the instruction format allows fusing any arithmetic operation with shift and output operations. The FusedReLu instruction is fully pipelined, allowing the full 96 × 96 dot-product engine to be unloaded in 96 cycles. More complex postprocessing sequences require additional instructions, increasing the unloading time of the Dot Product Engine. Some complex sequences are built out of FP32 instructions and conditional execution. The 30-bit accumulator value is converted to an FP32 operand in the beginning of such SIMD programs, and the FP32 result is converted back to the 8-bit integer output at the end of the SIMD program.

Pooling Support

After postprocessing in the SIMD unit, the output data can also be conditionally routed through a pooling unit. This allows the most frequent small-kernel pooling operations (2 × 2 and 3 × 3) to execute in the shadow of the SIMD execution, in parallel with the earlier layer producing the data. The pooling hardware implements aligners to align the output pixels that were rearranged to optimize convolution, back to the original format. The pooling unit has three 96-byte × 96-byte pooling arrays with byte-level control. The less frequent larger kernel pooling operations execute as convolution layers in the dot-product engine.

Memory Organization

The NNA uses a 32-MB local SRAM to store weights and activations. To achieve high bandwidth and high density at the same time, the SRAM is implemented using numerous relatively slow, single ported banks. Multiple such banks can be accessed every cycle, but to maintain the high cell density, a bank cannot be accessed in consecutive cycles.

Every cycle the SRAM can provide up to 384 bytes of data through two independent read ports, 256-byte and 128-byte wide. An arbiter prioritizes requests from multiple sources (weights, activations, program instructions, DMA-out, etc.) and orders them through the two ports. Requests coming from the same source cannot be reordered, but requests coming from different sources can be prioritized to minimize the bank conflicts.

During inference, weights tensors are always static and can be laid out in the SRAM to ensure an efficient read pattern. For activations, this is not always possible, so the accelerator stores recently read data in a 1-kB cache. This helps to minimize SRAM bank conflicts by eliminating back-to-back reads of the same data. To reduce bank conflicts further, the accelerator can pad input and/or output data using different patterns hinted by the network program.

Control Logic

As shown in Figure 7, the control logic is split between several distinct state machines: Command Sequencer, Network Sequencer, Address and Data sequencers, and SIMD Unit.

Each NNA can queue up multiple network programs and execute them in-order. The Command Sequencer maintains a queue of such programs and their corresponding status registers. Once a network runs to completion, the accelerator triggers an interrupt in the host system. Software running on one of the CPUs can examine the completion status and re-enable the network to process a new input frame.

The Network Sequencer interprets the program instructions. As described earlier, instructions are long data packets which encode enough information to initialize an execution state machine. The Network Sequencer decodes this information and steers it to the appropriate consumer, enforces dependencies and synchronizes the machine to avoid potential race-conditions between producer and consumer layers.

Once a compute instruction has been decoded and steered to its execution state machine, the Address Sequencer then generates a stream of SRAM addresses and commands for the computation downstream. It partitions the output space in sections of up to 96 × 96 elements and, for each such section, it sequences through all the terms of the corresponding dot-product.

Weights packets are preordered in the SRAM to match the execution, so the state machine simply streams them in groups of 96 consecutive bytes. Activations, however, do not always come from consecutive addresses and they often must

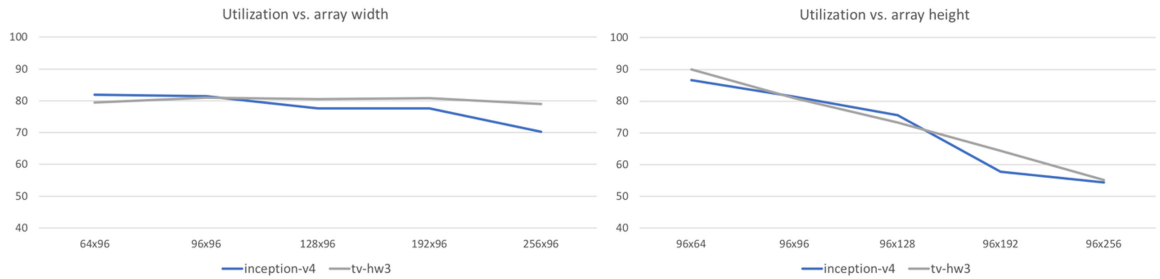


Figure 8. Achieved utilization versus MAC array dimension.

be gathered from up to 96 distinct SRAM locations. In such cases, the Address Sequencer must generate multiple load addresses for each packet. To simplify the implementation and allow a high clock frequency, the 96-elements packet is partitioned into 12 slices of 8 elements each. Each slice is serviced by a single load operation, so the maximum distance between its first and last element must be smaller than 256 bytes. Consequently, a packet of 96 activations can be formed by issuing between 1 and 12 independent load operations.

Together with control information, load data is forwarded to the Data Sequencer. Weights are captured in a prefetch buffer and issues to execution as needed. Activations are stored in the Data Cache, from where 96 elements are gathered and sent to the MAC array. Commands to the datapath are also funneled from the Data Sequencer, controlling execution enable, accumulator shift, SIMD program start, store addresses, etc.

The SIMD processor executes the same program for each group of 96 accumulator results unloaded from the MAC array. It is synchronized by control information generated within the Address Sequencer, and it can decode, issue, and execute a stream of SIMD arithmetic instructions. While the SIMD unit has its own register file and it controls the data movement in the datapath, it does not control the destination address where the result is stored. Store addresses and any pooling controls are generated by the Address Sequencer when it selects the 96×96 output slice to be worked on.

ARCHITECTURAL DECISIONS AND RESULTS

When implementing very wide machines like our MAC array and SIMD processor, the primary concerns are always tied to its operating clock frequency. A high clock frequency makes it easier to achieve the target performance, but it typically requires some logic simplifications which in turn hurt the utilization of specific algorithms.

We decided to optimize this design for deep convolutional neural networks with a large number of input and output channels. The 192 bytes of data and weights that the SRAM provides to the MAC array every cycle can be fully utilized only for layers with a stride of 1 or 2 and layers with higher strides tend to have poorer utilization.

The accelerator's utilization can vary significantly depending on the size and shape of the MAC array, as shown in Figure 8. Both the inception-v4 and the Tesla Vision network show significant sensitivity to the height of the MAC array. While processing more output channels at the same time can hurt overall utilization, adding that capability is relatively cheap since they all share the same input data. Increasing the width of the array does not hurt utilization as much, but it requires significantly more hardware resources. At our chosen design point (96×96 MAC array), the average utilization for these networks is just above 80%.

Another tradeoff we had to evaluate is the SRAM size. Neural networks are growing in size, so adding as much SRAM as possible could be a way to future-proof the design. However, a significantly larger SRAM would grow the pipeline depth and the overall area of the chip, increasing both power consumption and the total cost of the system. On the other hand, a convolutional layer too large to fit in SRAM can always be broken into multiple smaller components, potentially paying some penalty for spilling and filling data to the

DRAM. We chose 32 MB of SRAM per accelerator based on the needs of our current networks and on our medium-term scaling projections.

CONCLUSION

Tesla's FSD Computer provides an exceptional 21× performance uplift over commercially available solutions used in our previous hardware while reducing cost, all at a modest 25% extra power. This level of performance was achieved by the uncompromising adherence to the design principle we started with. At every step, we maximized the utilization of the available compute bandwidth with a high degree of data reuse and a minimalistic design for the control flow. This FSD Computer will be the foundation for advancing the FSD feature set.

Tesla's FSD Computer provides an exceptional 21x performance uplift over commercially available solutions used in our previous hardware while reducing cost, all at a modest 25% extra power. This level of performance was achieved by the uncompromising adherence to the design principle we started with.

The key learning from this work has been the tradeoff between efficiency and flexibility. A custom solution with fixed-function hardware offers the highest efficiency, while a fully programmable solution is more flexible but significantly less efficient. We finally settled on a solution with a configurable fixed-function hardware that executes the most common functions very efficiently but added a programmable SIMD unit, which executes less common functions at a lower efficiency. Our knowledge of the Tesla workloads deployed for inference allowed us to make such a tradeoff with a high level of confidence.

REFERENCES

1. Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Proceeding: Shape, Contour and Grouping in Computer Vision*. New York, NY, USA: Springer-Verlag, 1999.
2. W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural Comput.*, vol. 29, no. 9, , pp. 2352–2449, Sep. 2017.
3. K. Sato, C. Young, and D. Patterson, "An in-depth look at Google's first tensor processing unit," Google Cloud Platform Blog, May 12, 2017.
4. N. P. Jouppi *et al.*, "In-datacenter of a performance analysis tensor processing unit,'' in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, vol. 1, pp. 1–12.
5. I. Cutress, "AMD zen 2 microarchitecture analysis: Ryzen 3000,'' AnandTech, Jun. 10, 2019.
6. "NVIDIA volta AI architecture," NVIDIA, 2018. [Online]. Available: https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/
7. J. Choquette, "Volta: Programmability and performance," Nvidia, Hot Chips, 2017. [Online]. Available: https://www.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.21-Monday-Pub/HC29.21.10-GPU-Gaming-Pub/HC29.21.132-Volta-Choquette-NVIDIA-Final3.pdf
8. M. Horowitz, "Computing's energy problem," in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, 1999, pp. 10–14.
9. M. Komorkiewicz, M. Kluczewski, and M. Gorgon, "Floating point HOG implementation of for real-time multiple object detection," in *Proc. 22nd Int. Conf. Field Programm. Logic Appl.*, 2012, pp. 711–714.

Emil Talpes is a Principal Engineer with Tesla, Palo Alto, CA, USA, where he is responsible for the architecture and micro-architecture of inference and training hardware. Previously, he was a principal member of the technical staff at AMD, working on the micro-architecture of ×86 and ARM CPUs. He received the Ph.D. degree in computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA. Contact him at etalpes@tesla.com.

Debjit Das Sarma is a Principal Autopilot Hardware Architect with Tesla, Palo Alto, CA, USA, where he is responsible for the architecture and micro-architecture of inference and training hardware. Prior to Tesla, he was a Fellow and Chief Architect of several generations of ×86 and ARM processors at AMD. His research interests include computer architecture and arithmetic with focus on deep learning solutions. He received the Ph.D. degree in computer science and engineering from Southern Methodist University, Dallas, TX, USA. Contact him at ddassarma@tesla.com.

Ganesh Venkataramanan is a Senior Director Hardware with Tesla, Palo Alto, CA, USA, and responsible for Silicon and Systems. Before forming the Silicon team with Tesla, he led AMD's CPU group that was responsible for many generations of ×86 and ARM cores. His contributions include industry's first ×86-64 chip, first Dual-Core ×86 and all the way to Zen core. He received the Master's degree from IIT Delhi, Delhi, India, in the field of integrated electronics and Bachelor's degree from Bombay University, Mumbai, Maharashtra. Contact him at gvenkataramanan@tesla.com.

Peter Bannon is a VP of hardware engineering with Tesla, Palo Alto, CA, USA. He leads the team that created the Full Self Driving computer that is used in all Tesla vehicles. Prior to Tesla, he was the Lead Architect on the first 32b ARM CPU used in the iPhone 5 and built the team that created the 64b ARM processor in the iPhone 5s. He has been designing computing systems for over 30 years at Apple, Intel, PA Semi, and Digital Equipment Corp. Contact him at pbannon@tesla.com.

Bill McGee is a Principal Engineer leading a machine learning compiler team, mainly focused on distributed model training on custom hardware. He received the BSSEE degree in microelectronic engineering from Rochester Institute of Technology, Rochester, NY, USA. Contact him at bill@mcgeeclan.org.

Benjamin Floering is a Senior Staff Hardware Engineer with Tesla, Palo Alto, CA, USA, whose research interests include low power design as well as high-availability and fault tolerant computing. He is also a member of IEEE. He received the BSEE degree from Case Western Reserve University, Cleveland, OH, USA, and the MSEE degree from University of Illinois at Urbana-Champaign, Champaign, IL, USA. Contact him at floering@ieee.org.

Ankit Jalote is a Senior Staff Autopilot Hardware Engineer. He is interested in the field of computer architecture and the hardware/software relationship in machine learning applications. He received the Master's degree in electrical and computer engineering from Purdue University, West Lafayette, IN, USA. Contact him at ajalote@tesla.com.

Christopher Hsiong is a Staff Autopilot Hardware Engineer with Tesla, Palo Alto, CA, USA. His research interests include computer architecture, machine learning, and deep learning architecture. He received the Graduate degree from the University of Michigan Ann Arbor, Ann Arbor, MI, USA. Contact him at chsiong@tesa.com.

Sahil Arora is a member of Technical Staff with Tesla, Palo Alto, CA, USA. His research interests are machine learning, microprocessor architecture, microarchitecture design, and FPGA design. He received the Master's degree in electrical engineering from Cornell University, Ithaca, NY, USA, in 2008. Contact him at saarora@tesla.com.

Atchyuth Gorti is a Senior Staff Autopilot Hardware Engineer with Tesla, Palo Alto, CA, USA. His research interests include testability, reliability, and safety. He received the Master's degree from the Indian Institute of Technology, Bombay, in reliability engineering. Contact him at agorti@tesla.com.

Gagandeep S Sachdev is a Staff Hardware Engineer with Tesla, Palo Alto, CA, USA. He has worked as a Design Engineer with AMD and ARM. His research interests include computer architecture, neural networks, heterogeneous computing, performance analysis and optimization, and simulation methodology. He received the Master's degree from University of Utah, Salt Lake City, UT, USA, in computer engineering, with research topic of compiler-based cache management in many core systems. Contact him at gsachdev@tesla.com.