Research on the Functionality of Apple's Neural Processing Unit

Kseniia Kretsula
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
kretsula.pn@ucu.edu.ua

Khrystyna Mysak
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
mysak.pn@ucu.edu.ua

Mariia Ivanchenko
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
ivanchenko.pn@ucu.edu.ua

Dzvenyslava Butynets
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
butynets.pn@ucu.edu.ua

Ihor Babin
ADVA Soft
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
babin@ucu.edu.ua

Yaroslav Romanus
ADVA Soft
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
yaroslav.romanus@ucu.edu.ua

***Abstract*—This report presents an in-depth exploration of the Apple Neural Engine (ANE) as a key player in the on-device AI inference trend. The project is structured into three parts, each addressing specific aspects of ANE functionality, optimization, and potential contributions to the broader AI community.**

***Index Terms*—Apple Neural Engine, Core ML, Core ML Tools, Stable Cascade**

I. Introduction

With the advancement of computer technology, 2023 emerged as a year marked by a significant surge in the progress of Artificial Intelligence (AI), characterized by frequent releases from major corporations and an increase in startups integrating AI into their products. Despite this development, the demand for server computational resources, particularly Graphics Processing Units (GPUs), remains a significant challenge, affecting both network training and inference on a massive scale. OpenAI has highlighted the cost disparity, asserting that inference is notably more resource-intensive than training [1].

In response to this challenge, there is a growing trend towards on-device inference, where AI models execute directly on user devices. This approach offers advantages such as reduced server infrastructure costs, enhanced user data privacy, and a decreased ecological footprint through the utilization of energy-efficient on-device resources.

Apple, a prominent contributor to this paradigm shift, introduced the Apple Neural Engine (ANE) in 2018. ANE is a specialized neural network unit designed for optimal on-device inference, showcasing superior efficiency for specific operations. Unlike GPUs, which excel in parallel processing, ANE's specialization allows for highly efficient execution of a limited set of operations. Noteworthy examples include MobileOne for 1 ms inference and Stable Diffusion for 10 seconds.

Despite its potential, working with ANE poses challenges due to its closed nature. Apple provides access but restricts developers from understanding its internal workings, offering only a few examples of effective usage.

II. Aim

This project aims to study the operation of the Apple Neural Engine (ANE). The project can be roughly divided into two parts — the research of the Neural Engine itself and the application and optimization of the Stable Cascade model to grasp a deeper understanding of the working process of ANE.

The stages of the research include:

1) Explore the process of the neural network conversion and automatic optimization.
2) Develop optimizations based on the insights gained from the previous step and a deep understanding of how ANE can work more efficiently.

III. Setup

A. Apple Neural Engine

Apple Neural Engine (ANE) represents a type of Neural Processing Unit (NPU) comparable to a GPU. However, its focus is on accelerating neural network operations, such as convolutions and matrix multiplications, rather than graphics.

Since 2017, every chip in Apple's series A, designed for devices like the iPhone, certain iPad models, and Apple TV, has been equipped with the Neural Engine. Beginning in 2020, all Apple silicon chips, including M1, M2, and M3, have also integrated ANE.

The initial version featured two neural cores, capable of processing up to 600 billion operations per second, while the chips M2 Pro/Max are capable of processing up to 15.8 trillion operations per second.

ANE facilitates the offloading of specific tasks that were traditionally handled by the central processing unit (CPU) or graphics processing unit (GPU). A machine learning model optimized for ANE demonstrates enhanced productivity compared to these processors.

Nevertheless, ANE has its limitations. Unfortunately, Apple does not provide developers with explicit guidelines on optimizing their models to fully exploit the benefits of ANE. This process primarily relies on experimentation to determine the most effective approaches and identify any potential drawbacks.

B. Core ML

Core ML is an Apple framework to integrate machine learning models into the app. It provides a unified representation for all models. Your app uses Core ML APIs and user data to make predictions and fine-tune models, all on the user's device. Core ML optimizes on-device performance by leveraging the CPU, GPU, and Apple Neural Engine (ANE) while minimizing its memory footprint and power consumption [3].

Running a model strictly on the user's device removes any need for a network connection, which helps keep the user's data private and your app responsive.

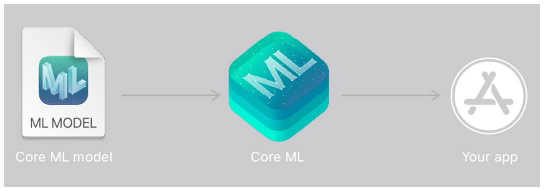


Fig. 1. Core ML Scheme [3].

C. Core ML Tools & Conversion

Core ML Tools enables us to convert machine learning models from third-party libraries to the Core ML format, as well as read, write, and optimize Core ML models [7].

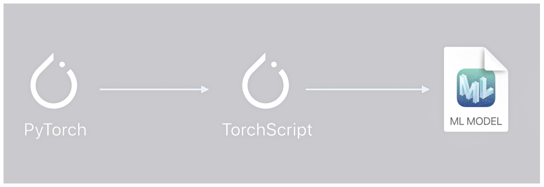


Fig. 2. Convertion from PyTorch to Core ML [7].

The conversion happens using mainly the Core ML Tools Python package. Here are the detailed steps of a PyTorch ML model conversion:

- Start with the PyTorch model we want to convert.
- Use PyTorch's JIT module to convert to a representation called TorchScript.
- With a TorchScript model in hand, we will invoke the new Core ML converter to generate an ML model. The converter works by iterating over the operations in the TorchScript graph and converting them one by one to their Core ML equivalent. Sometimes, one TorchScript operation might be converted into multiple Core ML operations. Other times, the graph optimization pass might detect a known pattern and fuse several operations into one.

D. Scripting & Tracing

Tracing and Scripting are both processes that turn models written in eager-mode Python code into a graph that describes computation. Scripting parses the Python source code of the model and compiles the code into a graph. Tracing runs a model with certain inputs and ”traces” all the operations that are executed into a graph.

They differ in operations that they could support. Scripting does not support complicated syntax and operation, but it is good for the most basic ones, so code quality could severely deteriorate. Tracing could handle and support any operations, but you should be careful with variables and dynamic control flow.

Another difference between them is how they affect code health. The trace only keeps one branch of the condition, but the script checks all branches. As a result, tracing has less effect on code than scripting.

Tracing and scripting both have problems, and the best solution is usually to mix them. This gives us the best of both worlds. To minimize the negative impact on code quality, we should use tracing for the majority of logic and use scripting only when necessary.

E. Performance Report and Optimization

Performance report provides us with crucial information about the performance of the ML model. Each of the performance reports is generated using Xcode, which is tightly integrated with Core ML. It shows us information about average/min/max load and model prediction times. In addition, a more important feature is that we can see which operations at which levels are performed on the ANE, CPU, or GPU. Our goal is to achieve the maximum number of operations (ideally all) performed on ANE. Since moving data between processing units is very time-consuming (sometimes so much so that it would be more profitable to use only the CPU), it is necessary to optimize and rewrite the operations that are performed on the CPU so that they would be executed on the ANE. Additionally, given the typically huge size of models, employing compression techniques offers benefits.

F. Compression techniques

There are several ways to compress model weights. The first way is to store them more efficiently using a sparse matrix representation. This can be achieved by using a technique called pruning. Another way is to reduce the precision used to store the weights. This can be achieved by both quantization and palettization. However, both strategies are lossy and generally less accurate than their uncompressed counterparts.

Pruning helps to store model weights efficiently by representing them as a sparse matrix. Pruning means setting some of the weights to 0. This means that only non-zero values need to be stored. This saves about 2 bytes of memory for each 0 value entered. (Of course, you will also need to store the location of the zeros to reconstruct the matrix later.)

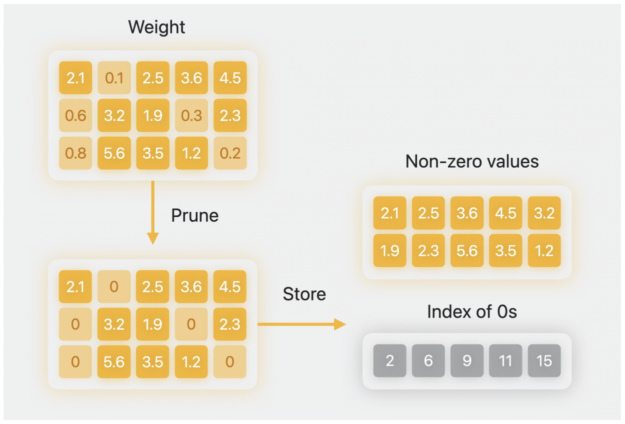


Fig. 3. Pruning technique [13].

The second method of compressing weights is **quantization**, which uses 8-bit precision to store weights. To perform quantization, you modify the weights (scale, shift, and round them) so that they fall within the INT8 range. In the example below, the scale is 2.35, and the bias is 0.

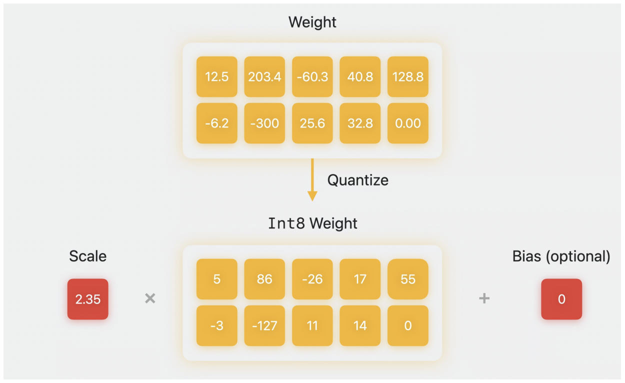
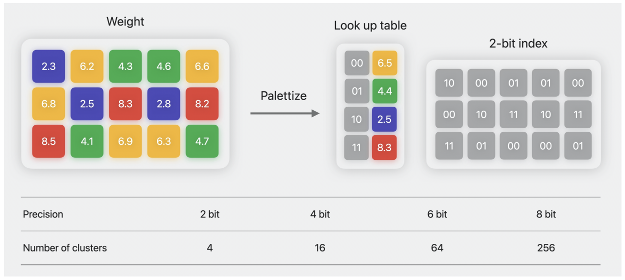


Fig. 4. Quantization technique [13].

To reduce the accuracy of the scales below 8 bits, you can use a technique called weight clustering or **palettization**. In this technique, weights with similar values are grouped and represented using the value of the center of the cluster to which they belong. These cluster centers are stored in a lookup table. The original weight matrix is transformed into an index table where each element points to the corresponding cluster center.

In this example, since we have four clusters, it is possible to represent each weight using 2 bits, which allows for a compression factor of 8 compared to Float16.



Precision	2 bit	4 bit	6 bit	8 bit
Number of clusters	4	16	64	256

Fig. 5. Palettization technique [13].

IV. Neural Processing Circuit

A. *Structure of Neural Processing Circuit*

A neural processor circuit is a circuit that performs various machine learning operations based on computations, including multiplication, addition, and accumulation. It performs neural network operations on the input data based on at least the kernel data.

It contains a Neural Task Manager, which controls the execution of tasks using the Neural Processing Circuit. Kernel Direct Memory Access (DMA) fetches kernel data from System Memory and sends it to each of the Neural Engines. Neural engines execute operations for neural network tasks concurrently. Data Buffer is a temporary storage for data associated with the Neural Network operations. The Buffer DMA retrieves a segment of input data from System Memory, storing it in a Data Buffer, while a write circuit transfers data from the Data Buffer to System Memory.

The overall work of the Circuit starts with loading data from System Memory to Buffer Data and then to Data Buffer, which distributes input data between all present Neural Engines. Then, Kernel DMA gets kernel data from System Memory and sends it to each of the Neural Engines. In this context, kernel data represents information from which kernel elements can be extracted. Later, it will be used to extract kernel coefficients (Neural Network weights for specific layers).

B. *Structure of Neural Engine*

The Neural Engine (NE) receives the input data and performs multiply-accumulate (MAD, aka convolution) operations on the input data based on stored kernel data, which is Kernel Extract extracts from Kernel DMA. Then, further post-processing operations are performed based on the result of the multiply-accumulate operations, and the output data is generated.

The input buffer circuit is a circuit that stores a portion of the input data as it is received from the Data Buffer and, after changing portion size by shifting read locations using a shifter, sends an appropriate portion of input data to the computation core for processing. An accumulator is a memory circuit for MAD operations results. A post-processor is a circuit that performs further processing of values received from the accumulator.

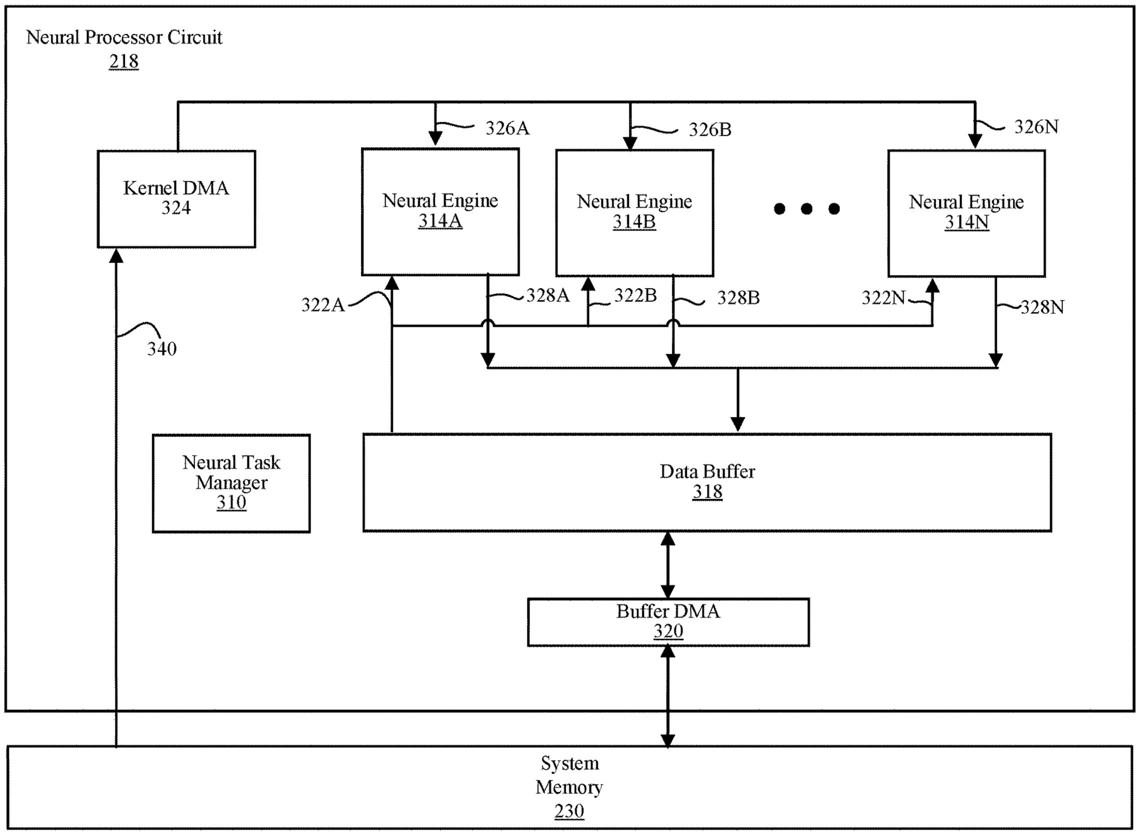


Fig. 6. Structure of Neural Processor Circuit [11].

C. Input Data Split

Before getting input data for performing MAD operations and post-processing, it is typically split into smaller pieces of data for parallel processing at multiple Neural Engines in an overlapping manner.

The overlapping portions are parts of the input data that are over-fetched in two adjacent slices to provide spatial support for a corresponding kernel. Within the loop for a slice is a processing loop for a tile of the slice, and the tile is a processing loop for a work unit. A work unit is a portion of the input data having a size that produces output values that fit into the accumulator of the neural engine during a single cycle of the computation core.

For Neural Engines to effectively analyze data at segment or tile edges, these edges must overlap with adjacent segments or tiles. This overlapping, called spatial support, offers additional context or space, enabling kernels to process data at the boundaries of selected areas without losing important information.

D. Rasterizers and Method for Processing Input Data in Neural Processor Circuit

A rasterizer is a component within various parts of the Neural Processor Circuit (Buffer DMA, Kernel DMA, Neural Engine), which tracks a segment of input data (for example, a tile, a work unit) and manages the components of the neural processor, a circuit to process the input data segment properly.

First, the Neural Task Manager programs rasterizers, and then the operating buffer DMA process is started. It is initialized by a rasterizer instructing Buffer DMA to cause Buffer DMA to receive a tile of input data from System Memory. The tile received by Buffer DMA is stored in Data Buffer.

The rasterizer in Data Buffer then instructs Data Buffer to send a work unit to one or more Neural Engines. The work unit is then stored in input buffer circuits of one or more Neural Engine.

The Input Buffer of the Neural Engine selects a portion of the work unit to be sent to the MAC (Multiply-Accumulate Module) for the convolution operation. The MAC then performs multiply-accumulate operations on the selected portion of the work unit using the appropriate kernel. It is then determined whether the entire work unit has been processed by one or several neural engines. If not, the selected portion of the work unit is shifted by shifters and returned for another round of multiply-accumulate operations.

Q: Has the entire work unit been processed?
A: If yes, the data buffer sends the next work unit to the neural engine.

Q: Have all tiles been processed?

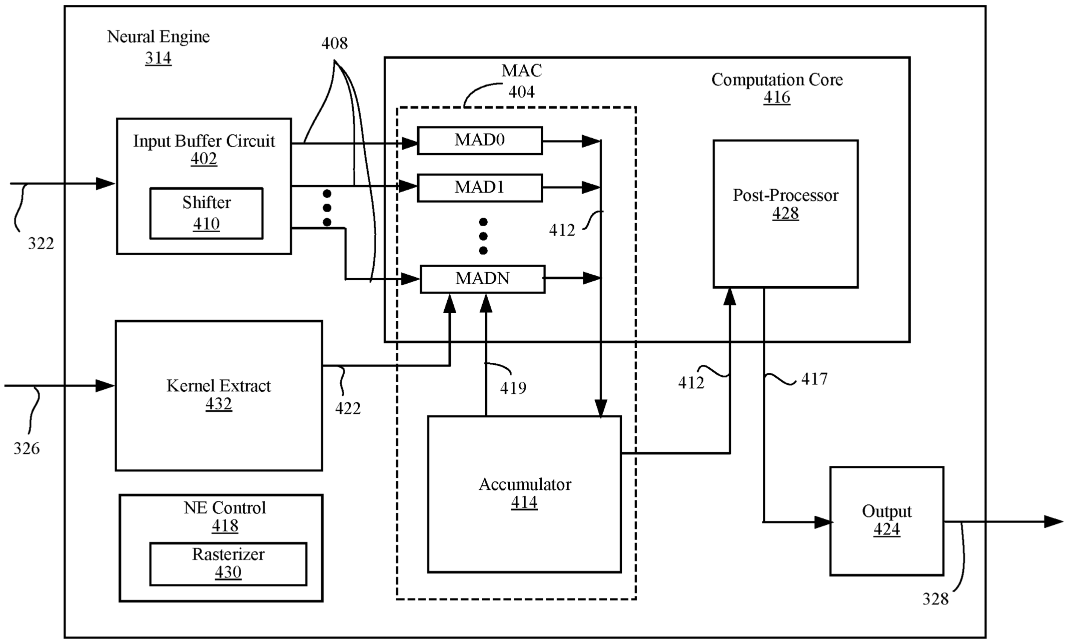


Fig. 7. Structure of Neural Engine [11].

A: If not, the process moves on to the next tile by instructing the buffer DMA through the rasterizer to receive the next tile from system memory and repeat the subsequent processes.

A neural network consists of a collection of layers, each performing specific computational tasks to process data. Before the model is executed on the Neural Processor Circuit, it is converted, such as by the CPU, into a task list. This task list includes a sequence of layers.

E. Neural Task Manager

The Neural Task Manager manages the execution of tasks in the neural processor network. It uses task queues to organize task execution from the task list. The task arbiter is a circuit that selects tasks, using a specific parameter – priority, from the Task Queues for execution by the Neural Processor Circuit. The CPU places references to the task list in the queue. The reference stored in the queue includes pointers and counters leading to the list of task descriptors in system memory. A task descriptor is a detailed description of the task, which defines how exactly the neural processing circuit should execute it.

After choosing a task, using reference, Task Manager DMA gets the task descriptor stored in System Memory. Then, it will be placed in the Fetch Queue until it is confirmed for execution. Configuration Queue stores ready for execution tasks. While a task is in a configuration queue, the neural processor circuit performs a prefetch for configuration and kernel data before other components of the neural processor circuit execute the task.

The task queue contains references to task descriptors that are physically stored in system memory. Key elements of the task queue include pointers to the first task to be processed, the network ID associated with the task, references for managing and tracking the number of tasks in the queue, priority settings for task arbitration, and indicators for managing queue execution interruptions and resumptions.

F. Configuration Queue

The configuration queue stores task descriptors of tasks committed for execution by the neural processor circuit. The configuration queue can have multiple distinct queues, each of which stores a portion of the configuration data extracted from the task descriptor. This setup allows the organization of data in a way that each component of the neural circuit has access only to the information it needs. The neural task manager uses this queue to program rasterizers, ensuring they track the necessary order for various components (Neural Engine, Data Buffer, Buffer DMA, Kernel DMA).

G. Task Switching Process

Through the task arbiter, the neural task manager can support task switching between task queues. Task switching may occur because a task queue has a higher priority parameter than a task queue being executed. The priority parameter of a task queue can be determined according to the task urgency, task length, and task source. If two or more task queues have the same priority parameter, the neural task manager can select a task queue by considering additional parameters, such as task queue indexes. The task manager can trigger a

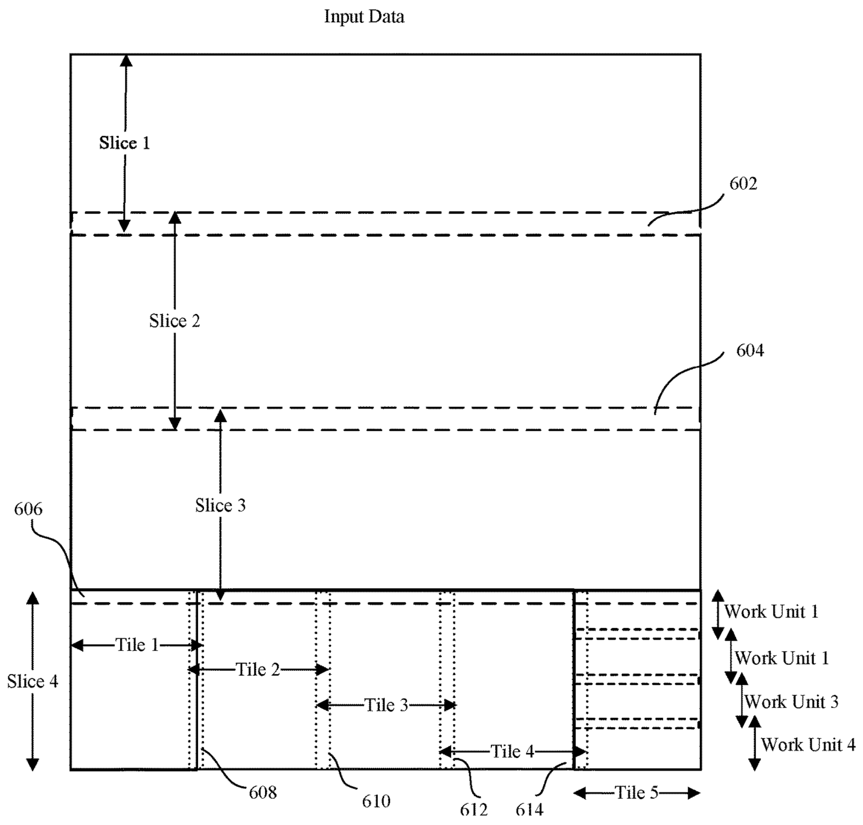


Fig. 8. Input Data Split [11]

task switch request after determining that a task or a task queue has a higher priority parameter than a current task or current task queue. Output data from a task from a task list can be directly stored in the system memory, provided there is a switch between queues. The output information will be stored in a data buffer when tasks are executed within a single queue. This is related to the fact that when execution takes place within one queue, it needs quick access to memory since each subsequent task is expected to need information from the previous one.

Task descriptor headers can include configuration data that configure behaviors related to task switching. Specifically, there are plenty of them that define the following workflow.

- Task Switch Enable (TSE) Parameter – defines whether the neural task manager can begin a task switch process after execution of the task.
- Task Switch Ready (TSR) Parameter – defines whether the neural task manager can task switch after execution of the task. The last task in a task list of a current task queue has TSE=1 and TSR=1. This can allow the neural task manager to switch to a new task queue after the current task queue is complete, even without receiving a task switch request.
- Source Pointer Change (SPC) Parameter – defines whether the input data for the task should be retrieved from the system memory or the data buffer.
- Destination Pointer Change (SPC) Parameter – defines whether the output data of the task should be stored in the system memory or the data buffer.
- Task Switch Ready (TSR) Parameter – defines whether the neural task manager can task switch after execution of the task.
- Source Pointer Last (SPL) Parameter – indicates that after returning to an interrupted task queue, the task is the last with input data stored in the system memory. The first task in a task list of a task queue may have SPL=1. This can clear any task-switching process that may have been set earlier for the task queue.
- Task Queue Switch Parameter (TQSP) – indicates whether the task queue was interrupted due to a task switch and needs to be completed.
- Global Task Switch Parameter (GTSP) – indicates that a task switch is in process.

The neural task manager can follow a set of rules, which relies on these configuration data parameters, that configures the neural task manager to direct a task switch process.

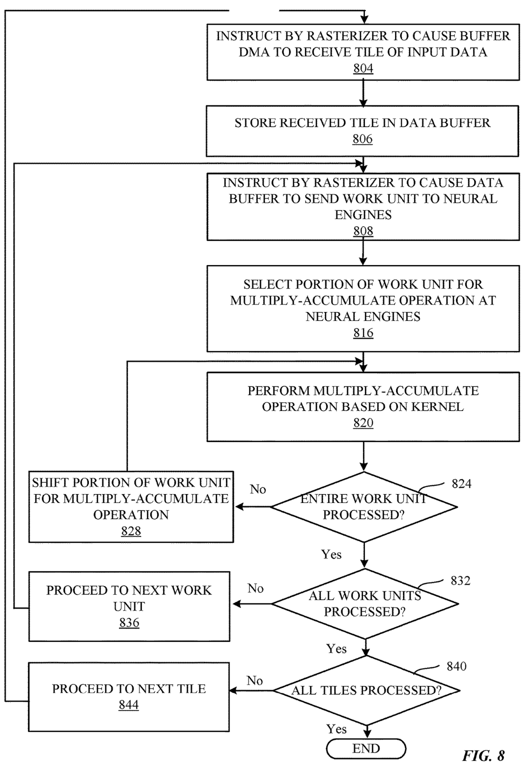

INSTRUCT BY RASTERIZER TO CAUSE BUFFER DMA TO RECEIVE TILE OF INPUT DATA 804
STORE RECEIVED TILE IN DATA BUFFER 806
INSTRUCT BY RASTERIZER TO CAUSE DATA BUFFER TO SEND WORK UNIT TO NEURAL ENGINES 808
SELECT PORTION OF WORK UNIT FOR MULTIPLY-ACCUMULATE OPERATION AT NEURAL ENGINES 816
PERFORM MULTIPLY-ACCUMULATE OPERATION BASED ON KERNEL 820
ENTIRE WORK UNIT PROCESSED? 824
No
SHIFT PORTION OF WORK UNIT FOR MULTIPLY-ACCUMULATE OPERATION 828
Yes
ALL WORK UNITS PROCESSED? 832
No
PROCEED TO NEXT WORK UNIT 836
Yes
ALL TILES PROCESSED? 840
No
PROCEED TO NEXT TILE 844
Yes
END
FIG. 8

Fig. 9. Method of Processing Input Data in Neural Processor Circuit [11]

H. Input Data Preparation

Usually, when we want to run a file, we need to compile it and get machine code to run it on the CPU. If we want our ML model to run on ANE, we first need to convert it into *.mlpackage* file, but to make instructions comprehensible to ANE, we need to compile the *.mlpackage* file to the *.mlmodelc* file. At this stage, the *.mlmodelc* file is a sequence of instructions for ANE to execute to run the model.

To make this instruction sequence accessible for ANE, we need to load them into System Memory via bus and network interface. System Memory stores instructions for execution by SOC components and data processed by SOC components. Both bus and network interfaces are used to exchange data between devices. After that, the Neural Processor Circuit via Buffer DMA and Data Buffer fetches and distributes instructions for each of the Neural Engines to execute.

Model execution starts from Stage C, which has prompt text as an input and operates in latent space. The output of Stage C will be an Input for the next Stage B, and the Output of this stage will be an Input for the last stage, Stage A. Output of Stage A is the final image generated by the model. Each stage input will be considered a different input for Neural Engines.

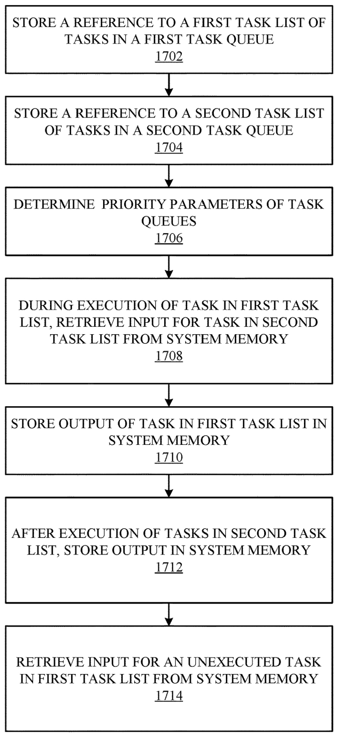

STORE A REFERENCE TO A FIRST TASK LIST OF TASKS IN A FIRST TASK QUEUE 1702
STORE A REFERENCE TO A SECOND TASK LIST OF TASKS IN A SECOND TASK QUEUE 1704
DETERMINE PRIORITY PARAMETERS OF TASK QUEUES 1706
DURING EXECUTION OF TASK IN FIRST TASK LIST, RETRIEVE INPUT FOR TASK IN SECOND TASK LIST FROM SYSTEM MEMORY 1708
STORE OUTPUT OF TASK IN FIRST TASK LIST IN SYSTEM MEMORY 1710
AFTER EXECUTION OF TASKS IN SECOND TASK LIST, STORE OUTPUT IN SYSTEM MEMORY 1712
RETRIEVE INPUT FOR AN UNEXECUTED TASK IN FIRST TASK LIST FROM SYSTEM MEMORY 1714

Fig. 11. Task Switching Process [11].

V. Stable Cascade

Stable Cascade is a text-to-image model, which consists of three models: Stage A, Stage B and Stage C, representing a cascade to generate images, hence the name “Stable Cascade”. This architecture, called Würstchen, aims to learn extremely compact image representation that can guide the diffusion process.

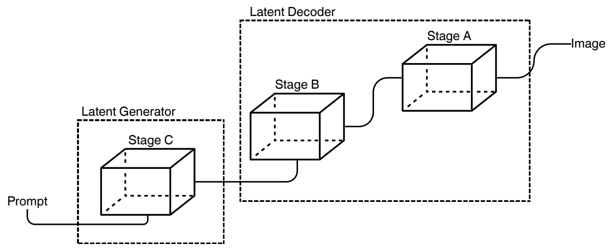


Fig. 12. The Würstchen architecture of Stable Cascade.

A. Architecture

The inference architecture consists of three stages [9], where the result of each stage is the input for the next one:

1) **Stage A** is the decoder of Vector-Quantized Generative Adversarial Network (VQGAN) operating in the latent space (a compressed representation of the original data where each dimension corresponds to a specific feature or characteristic) with a compression ratio of 4:1.

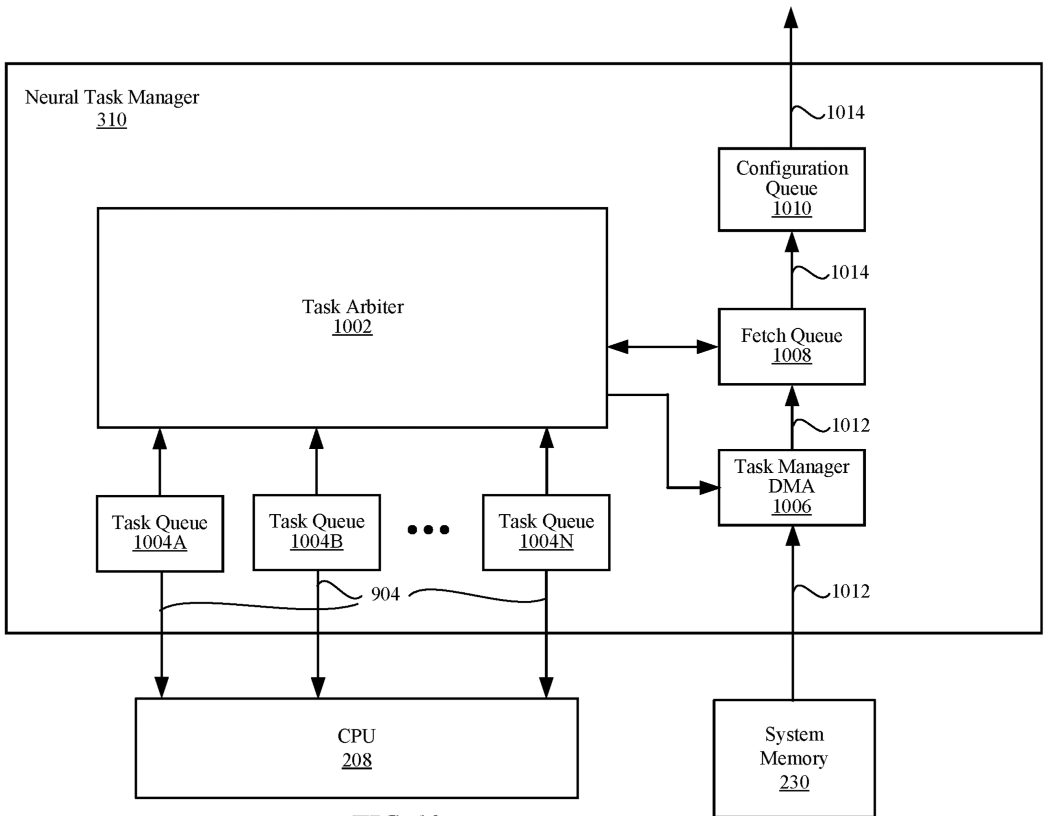


Fig. 10. Neural Task Manager [11]

2) **Stage B** is the Latent Diffusion Model (LDM) generating the sample in the latent space of the VQGAN in Stage A, conditioned on the text and the latent representation from Stage C.
3) **Stage C** is LDM that generates highly compressed latent representation sized in 16x24x24 (compression ratio of 42:1), conditioned on the text representation.

Stages A and B are used to compress images, similar to the job of the Variational Autoencoder (VAE) in Stable Diffusion. However, with this setup, a much higher compression of images can be achieved, as well as cheaper training and inference. Furthermore, Stage C is responsible for generating the small 24 x 24 latents given a text prompt.

B. Conversion

The code for our project and conversion specifically can be found here [12]. The conversion generally consists of the same stages described earlier. However, as a result, four essential models are obtained after Stable Cascade conversion: text-encoder, decoder, prior, and VQGAN. There are two pipelines from which we can get all necessary models: StableCascade-PriorPipeline and StableCascadeDecoderPipeline.

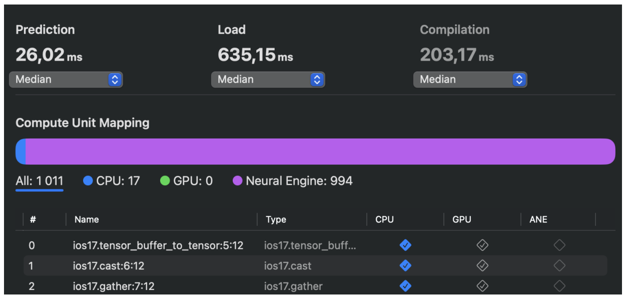
StableCascadePriorPipeline consists of exactly prior, text-encoder, and other auxiliary parameters. This pipeline is meant to be used with the Stage C model, which operates on the small 24 x 24 latents and denoise the latents conditioned on text prompts.

The Stage B and Stage A models are used with the **StableCascadeDecoderPipeline** (consisting of decoder, text-encoder, VQGAN, and other auxiliary parameters). They are responsible for generating the final image given the small 24 x 24 latents.

For convenience, we utilized a pipeline that consolidates all previously mentioned components – **StableCascadeCombinedPipeline**.

C. Performance report

The images below provide information about the performance of each component of the model.


Prediction 26,02 ms Median
Load 635,15 ms Median
Compilation 203,17 ms Median

Compute Unit Mapping

All: 1 011 ● CPU: 17 ● GPU: 0 ● Neural Engine: 994

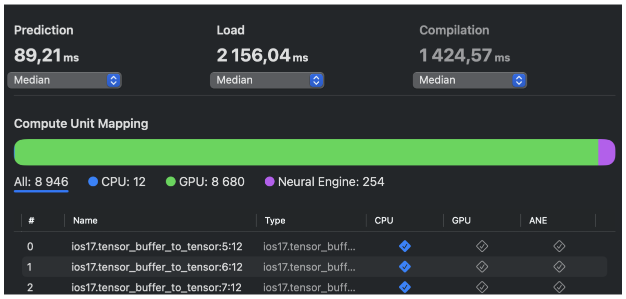
#	Name	Type	CPU	GPU	ANE
0	ios17.tensor_buffer_to_tensor:5:12	ios17.tensor_buff...	◆	◇	◇
1	ios17.cast:6:12	ios17.cast	◆	◇	◇
2	ios17.gather:7:12	ios17.gather	◆	◇	◇

Fig. 13. Performance report for the ’text encoder’ component.

1) Text encoder: The report offers an insightful overview of each layer within the text encoder. The majority of operations are efficiently handled by ANE, with only a small portion being processed by the CPU.

2) Decoder: For the decoder, the situation is completely different: the bulk of layer operations are offloaded to the GPU for computation, with the ANE handling only 254 layers, compared to the GPU’s 8946. The primary assumption behind this performance is that the ’decoder’ component of the Stable Cascade, along with the ’prior,’ is highly complex. Therefore, most operations cannot be offloaded to the ANE, and optimizing it would necessitate a complete overhaul of the model.

3) VQGAN: Lastly, VQGAN’s performance is highly optimized. Only 2 layer operations are handled by the CPU, while the remaining 239 are efficiently processed by the ANE.

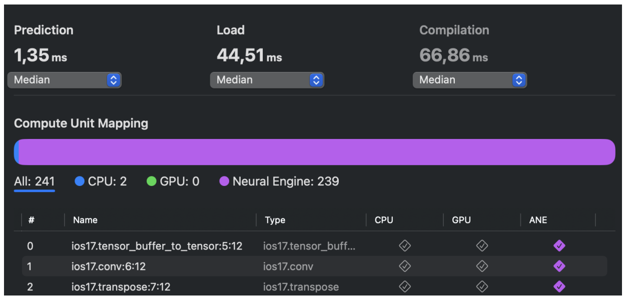

Prediction 89,21 ms Median
Load 2 156,04 ms Median
Compilation 1 424,57 ms Median

Compute Unit Mapping

All: 8 946 ● CPU: 12 ● GPU: 8 680 ● Neural Engine: 254

#	Name	Type	CPU	GPU	ANE
0	ios17.tensor_buffer_to_tensor:5:12	ios17.tensor_buff...	◆	◇	◇
1	ios17.tensor_buffer_to_tensor:6:12	ios17.tensor_buff...	◆	◇	◇
2	ios17.tensor_buffer_to_tensor:7:12	ios17.tensor_buff...	◆	◇	◇

Fig. 14. Performance report for the ’decoder’ component.


Prediction 1,35 ms Median
Load 44,51 ms Median
Compilation 66,86 ms Median

Compute Unit Mapping

All: 241 ● CPU: 2 ● GPU: 0 ● Neural Engine: 239

#	Name	Type	CPU	GPU	ANE
0	ios17.tensor_buffer_to_tensor:5:12	ios17.tensor_buff...	◇	◇	◆
1	ios17.conv:6:12	ios17.conv	◇	◇	◆
2	ios17.transpose:7:12	ios17.transpose	◇	◇	◆

Fig. 15. Performance report for the ’VQGAN’ component.

D. Core ML Instrument

While creating a performance report can demonstrate a model’s potential speed and efficiency, an additional tool is required to profile the model’s performance in real time. To do this, we can use the **Core ML Instrument**, which allows us to visualize the model’s performance while it is running in real-time, which helps identify possible performance issues and make necessary optimizations.

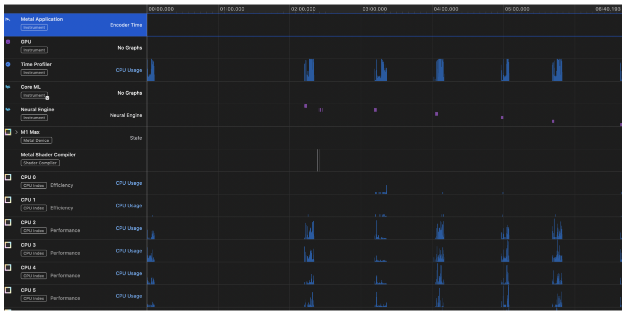


Fig. 16. Core ML Instrument for ’text encoder.’

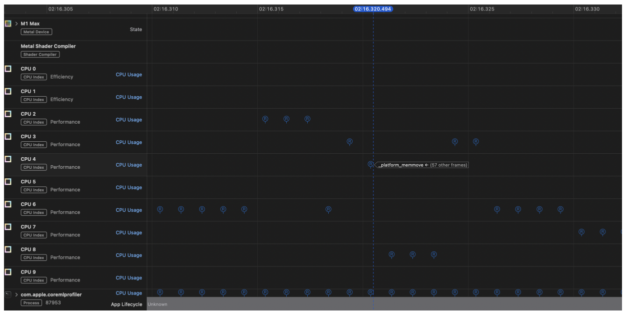


Fig. 17. Zoomed in Core ML instrument for ’text encoder’ showing the distribution of operations between cores.

VI. Summary

This research explores the ANE that offers specialized processing for specific ML operations, enhancing efficiency in on-device inference tasks. It dives into the structure of the Neural Processing Circuit and Neural Engine, data partitioning, and data processing operations research. There is an explanation of the functionality of different parts of neural processing, such as the Neural Task Manager, and processes, such as input preparation and task switching. Core ML facilitates model conversion and optimization for ANE utilization. Performance analysis reveals ANE’s efficiency in offloading tasks despite complexity constraints in certain components. Real-time profiling with Core ML Instrument aids in identifying optimization opportunities. The study also delves into the application of ANE in the Stable Cascade text-to-image model, showcasing its efficacy in on-device inference. The study has further potential and can be improved by performing optimizations on the converted model of Stable Cascade, as this will help with a better understanding of the operation and conversion of the model and further investigation of the performance of ANE. Overall, the research highlights ANE's significance, challenges, and optimization strategies in advancing on-device AI and ML capabilities. The code of the project is available at https://github.com/khrystynamk/ANE-Research.

References

[1] How to Scale Generative AI Without Hurting the Bottom Line
[2] Core ML tools framework GitHub
[3] Core ML Overview
[4] Converting PyTorch models to Core ML
[5] TorchScript: Tracing vs. Scripting
[6] What are convolutional neural engine?
[7] Convert PyTorch models to Core ML
[8] Introducing Stable Cascade
[9] Würstchen: An Efficient Architecture for Large-Scale Text-to-Image Diffusion Models
[10] Note on Stable Cascade and Würstchen architecture
[11] Scalable neural network processing engine
[12] GitHub of the project, ANE Research
[13] Use Core ML Tools for machine learning model compression