EUROPEAN WORKSHOP ON ON-BOARD DATA PROCESING (OBDP2021), 14-17 JUNE 2021

RAMON SPACE RC64-BASED AI/ML INFERENCE ENGINE

Ran Ginosar, David Goldfeld, Peleg Aviely, Roei Golan, Avraham Meir, Fredy Lange, Dov Alon, Tuvia Liran, Avi Shabtai

Ramon.Space, Yoqneam Illit 2069201, Israel

ABSTRACT

RC64 addresses the challenges facing execution of Machine Learning inference in Space: Long maintenance-free lifetime, high radiation, wide temperature span and numerous thermal cycles. Commercial, industrial and automotive AI accelerators cannot perform in the harsh environment of Space. A ML framework is implemented for on-board inference using RC64. Ground-based development converts standard ML models, and parallelizes the computations while considering the specific dimensions, sizes and operations of each layer. The parallelized model is transformed into an interpretable RC64 model, based on RC64 task-based programming model. The resulting RC64 ML model is beamed up to the on-board RC64-based Inference Engine, where it is interpreted. This approach offers high performance, power efficiency, and cyber security. RC64 architecture, capabilities and programming models, as well as RC64-based storage, are described.

1. INTRODUCTION

The fast proliferation of Artificial Intelligence (AI) and in particular Machine Learning (ML) applications, as well as their insatiable demand for high performance computing and for handling Big Data, brought about a wave of innovation, leading to numerous hardware accelerators and special purpose architectures for data center and for edge computing alike [4][5][6][7][4].

The Space domain is destined to take strong advantage of that trend. Many on-board Space applications of ML have emerged, as follows. In the area of EO & Remote Sensing, cloud detection, object identification, recognition, and change detection have been proposed; Spectrum analysis, anomaly and interference detection, interference mitigation, source location and modulation classifiers are being developed for telecommunications; Robotics and vision based navigation, docking and landing are applied to exploration, mission extension and debris management; Managing spectra, networks and users are proposed for communications; and ML-based cybersecurity is applicable to all Space uses.

Key challenges faced by on-board Space systems for the storage and high performance processing of ML applications are mostly due to the harsh conditions of Space. On-board processing should survive the duration of the mission, extending many years in high-end cases. Computing hardware and Big Data storage should withstand the heavy radiation and be resilient to the various radiation effects. Systems are exposed to very wide temperature ranges and undergo a very large number of thermal cycles. All these challenges may pose significant risks when depending on available AI/ML accelerators that have not been designed for use in Space.

RC64 manycore processor addresses both DSP and AI/ML computations. Not only can it operate in Space (practically) forever, it also demonstrates high performance and high power efficiency when executing both DSP and AI/ML applications.

This paper describes using RC64 on-board for inference. Training is to be carried out on the ground, followed by adapting the model for execution in Space. The Inference Engine (IE) interpreter, including a rich library of kernels for all sorts of ML layers, is installed on-board RC64. Once the adapted model is "beamed up" to the spacecraft, it is interpreted by IE. The model is never compiled into any code, in order to maintain security and cyber-protection of the on-board computer. Data for ML is either stored on-board or provided by sensors and receivers. Results are either stored on-board or transmitted elsewhere.

The paper briefly introduces RC64 in Sect. 2, presents the concept of on-board ML inference in Sect. 3, and explains the flow for model development in Sect. 4. The Inference Engine (IE) is discussed in Sect. 5 and is evaluated in Sect. 6.

2. RC64 AND ITS PROGRAMMING MODEL

The RC64 many-core has been presented in full in [1]. As shown in Figure **1**, 64 DSP cores are all interconnected in parallel to a many-bank shared memory, as well as to a central hardware scheduler. Each core is equipped with local instruction and data caches, a local scratchpad memory, four fixed-point multiplier-accumulators organized as a SIMD unit, one floating-point fused-multiply-add unit, and a VLIW controller. The cores employ soft-error-protected flip-flops, error detecting memories, and multiple error and fault monitors enabling uninterrupted operation in Space. The shared memory is similarly protected against SEU and SET. The entire RC64 is continuously

monitored and managed by its Fault Detection, Isolation and Recovery (FDIR) unit, which also interacts with off-chip FDIR controllers.

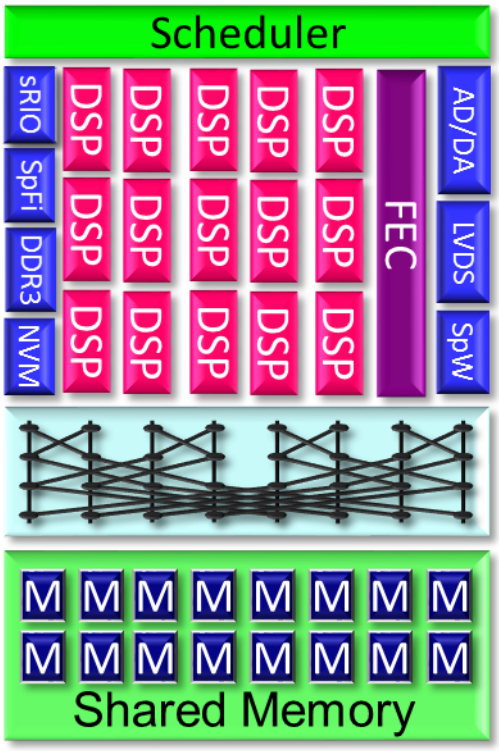

Figure 1. RC64 Many-Core Architecture. 64 DSP cores, modem accelerators and multiple DMA controllers of I/O interfaces access the multibank shared memory through a logarithmic network. The hardware scheduler dispatches fine grain tasks to cores, accelerators and I/O.

RC64 cores exchange data only with the shared memory, by means of Read and Write instructions, enabled by cache line fetch and write-through operations. RC64 cores do NOT exchange messages among themselves. Shared memory consistency is guaranteed by the programming model and by its implementation in the hardware scheduler.

Programs are organized as task-dependency-graphs and sequential task codes. The hardware scheduler, following the task graph, determines when a task is eligible for execution (all its predecessors in the graph have completed), and dispatches the task to an available core. The task executes in full, until completion, and then the core notifies the scheduler.

Tasks do not receive data arguments. Rather, they are programmed to find all their needs in shared memory. Before completing execution of a task, the core writes all results onto shared memory. No state is kept at the core past completion of a task.

Some tasks implement data parallelism by duplicating many instances. Each instance receives its unique ID as an argument, and uses that ID in order to assure that no two instances write into the same variable in shared memory.

This task-oriented programming model assures "Concurrent Read, Exclusive Write" (CREW) integrity and consistency, alleviating the need for (inefficient) MUTEX locks. The EW rule states that if one task writes into some variable, no other task that is concurrent to it is allowed to access that variable (for either read or write). The CR rules states that concurrent tasks may read from a shared variable but none of them is allowed to write into that variable.

An in-depth example of DSP application (DVB-S2) programming on RC64 can be found in [2]. Another notable application employs the radiation-hard RC64 as a storage controller, managing advanced highly-resilient storage that can be implemented using non-Space advanced flash storage devices. Figure 2 shows a first version of such a storage, achieving End-of-Life capacity of 1 TeraByte on a 10×10cm board.

Figure 2. 1 TB storage for Space. RC64 serves as the storage controller. Several 3D NAND flash devices make the highly protected, high endurance storage

3. MACHINE LEARNING INFERENCE ON RC64

ML models typically specify neural layers. Each layers receives tensor-organized inputs and many parameters (coefficients, or weights) organized in complex structure. The layer typically applies linear operations (input tensors multiplied by weight matrices) followed by non-linear operators such as pooling, min/max selection and an activation such as RELU. The inputs of most layers are the outputs generated by some previous layers, and the output of the last layer typically constitutes the result.

Prior to using a model on RC64, the model is transformed into a format compatible with RC64 programming model, as described in Sect. 2. The

Inference Engine (IE) interpreter is pre-designed as a loop of generic layer code. IE interprets the model one layer at a time, in the order specified by the model. The loop body invokes a kernel library according to the type of that layer. The kernel is obviously arranged as a parallel ("duplicable") task, so that RC64 parallelism can be utilized for the layer.

Many AI/ML applications need to handle "Big Data" and very large models (comprising millions of weights). The execution bottleneck shifts in such (typical) cases from the cores to fetching data and weights from off-chip memory (DRAM) and storage, as well as to receiving massive streams through high speed data links. The pre-programmed IE initiates pre-fetching of the data and weights for one layer while still computing a previous layer, in order to hide the fetching latency as much as possible.

Often, the data and weights required for a layer cannot fit within the shared memory. In such (common) cases, the layer is decomposed into *sub-layers*. While executing one sub-layer, the IE pre-fetches data and weights for the next sub-layer, to achieve latency hiding.

Next, Sect. 4 describes the (on-ground) model development, in preparation for beaming it up to Space, and Sect. 5 further explains IE. Both sections also relate to coordinating multiple RC64 processors for the same AI/ML inference model.

4. MODEL DEVELOPMENT FLOW

Model transformation for on-board interpretation by IE is carried out on the ground. A given model is first parsed. If desired, all or some floating-point computations are converted to fixed-point. Each layer is parallelized according to model and data dimensions and the required operations. A layer may have to be further decomposed into multiple consecutive stages, due to memory limitations. Parallelism may also extend multiple RC64 processors. The entire development flow is depicted in Figure 3.

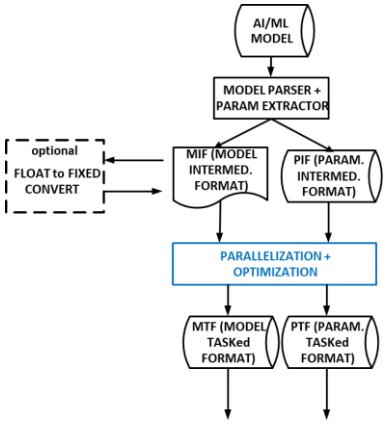

Figure 3. The Inference Development Flow

4.1. Parsing

A ML model specified in a standard framework such as KERAS is parsed into a Model Intermediate Format and a Parameter Intermediate Format (a self-describing data set of weights). Figure 4 and Figure 5 show an input sample and an example of the intermediate format.

Layer (type)	Output Shape	Param #
conv32_1 (Conv1D)	(None, 128, 32)	192
conv32_2 (Conv1D)	(None, 64, 32)	5152
conv32_3 (Conv1D)	(None, 32, 32)	5152
conv64_1 (Conv1D)	(None, 16, 64)	10304
conv64_2 (Conv1D)	(None, 8, 64)	20544
conv64_3 (Conv1D)	(None, 4, 64)	12352
flatten_1 (Flatten)	(None, 256)	0
dense128 (Dense)	(None, 128)	32896
dense_soft_max (Dense)	(None, 4)	516
activation_1 (Activation)	(None, 4)	0

Total params: 87,108
Trainable params: 87,108
Non-trainable params: 0

Figure 4. Sample input ML model

```
0:
  name: conv32_1
  type: Conv1D
  output_shape: (128, 32)
  activation: relu
  filters: 32
  input_shape: (256, 1)
  kernel_size: (5,)
  padding: same
  strides: (2,)
  use_bias: true
1:
  name: conv32_2
  type: Conv1D
  output_shape: (64, 32)
  activation: relu
  filters: 32
  input_shape: (128, 32)
  kernel_size: (5,)
  padding: same
  strides: (2,)
  use_bias: true
```

Figure 5. Model Intermediate Format (first two lines of the model of Figure 4).

Next, the given layers are examined. At times, splitting a layer in two may enhance performance or power efficiency. Similarly, when applying one library kernel to two consecutive model layers turns out to be more efficient than interpreting each layer separately, the two layers are combined. The two cases are exemplified in Figure 6, together with the resulting Model Task Format.

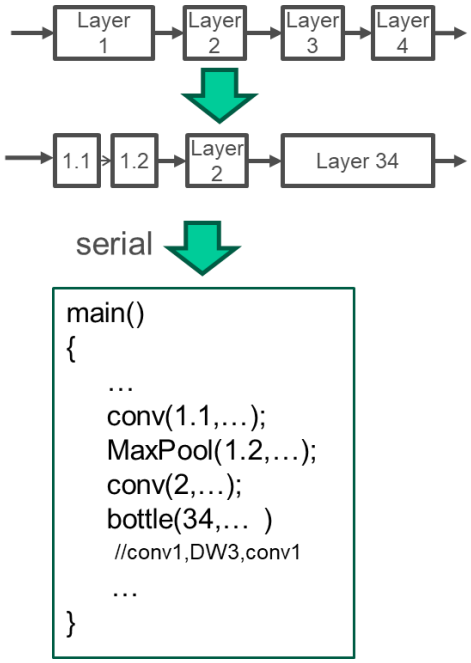

Figure 6. A given 4-layer model is re-arranged by splitting Layer 1 and merging Layers 3,4. The serial code defines order of interpretation by IE.

4.2. Parallelization

In principle, every line of the serial code in ***Figure 6*** can be mapped to a corresponding parallel (duplicable) task, as can be seen in Figure 7.

Figure 7. The serial model of Figure 6 can be mapped onto a corresponding list of parallel (duplicable) tasks.

Parallelizing layers in convolutional neural networks, where a tensor of inputs is multiplied by weight matrices, can be performed along the ideas shown in Figure 8. The input tensor may be parallelized along each one of its dimensions, and Figure 8 shows two two-way splits and one three-way parallelism. Similarly, output parallelization is possible by applying different convolution kernels in parallel.

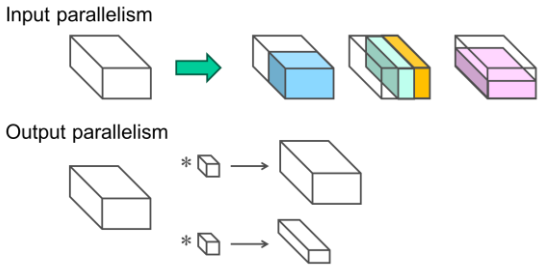

Figure 8. Input and output parallelism in a ML layer

Since conceptually all layers are similar to each other, a looping structure is formed as in Figure 9 rather than the format of Figure 7. The code loops over layers. In parallel with layer execution, outputs of the previous layer may be written off-chip to either DRAM or storage, and inputs and weights for the next layer are pre-fetched form either DRAM or storage, in order to hide latency.

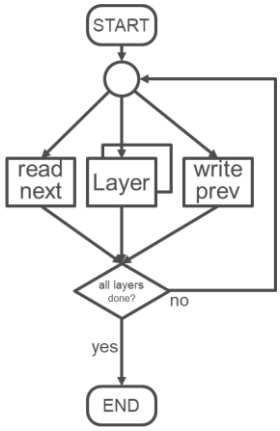

Figure 9. Generic task parallel code

4.3. Layer Optimization

The generic task parallel code of ***Figure 9*** often runs into the problem that the inputs and weights for a single layer exceed the capacity of the on-chip shared memory. In such cases, further optimization is applied. The layer is further decomposed into sub-layers, according to the various means of parallelization discussed in Figure 8. An inner loop is devised, as shown in Figure 10. While a sub-layer that is wholly contained in the on-chip shared memory is being executed, the outputs of the previous sub-layer are written out to either DRAM or storage, and the inputs and weights required for the next sub-layers are pre-fetched. Clearly, sufficient buffer space should be reserved in the on-chip shared memory.

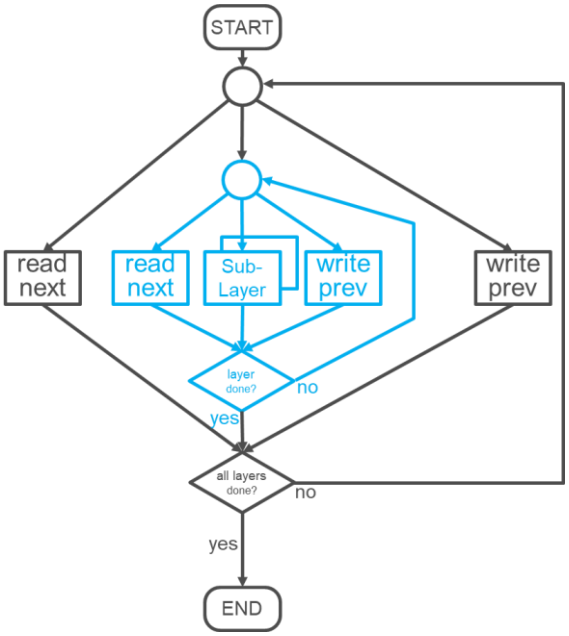

Figure 10. Fragmented generic task parallel code

4.4. Multi-RC64 Distribution

ML layer computations are readily parallelizable, since the computation of any one activation (output) is independent of the computations of all other activations of the layer. That observation has enabled the parallelization of ML computations as discussed above. Decomposing a layer computation over multiple RC64 processors is achievable in a similar manner. A higher level control task manages the synchronization of tasks performed in different processor in the same way as the hardware scheduler does within one RC64 processor.

Other alternative parallelizations are also considered: Applying same model to different input sets on different processors, pipelining computations over multiple processors, and more.

5. RC64 INFERENCE ENGINE

The outcome of the development process defines each layer in full using the Model Tasked Format (Figure 12). It is accompanied by the weight data set, using the Parameter Tasked Format. The IE interpreter executes the code shown in Figure 10. For each layer, that code reads the model, invokes the appropriate library kernels, and reads in the input data and weights from the addresses that are also specified by the Model Tasked Format (pointing at either storage, or DRAM, or on-chip shared memory). The outputs are stored at the addresses that are also specified in the model.

If the model is distributed over multiple RC64 processors, synchronization tasks coordinate the work.

6. RC64 INFERENCE ENGINE PERFORMANCE EVALUATION

IE on RC64 has been evaluated using the Keras version of VGG [8][9], opting for the VGG-19 model and datasets of 224×224 pixel images. VGG is a very large model, and when processing very large data sets it is indicative of expected on-board Space ML applications, listed in Sect. 1. High performance, low power results are demonstrated, and performance as well as power efficiency are compared with a Nvidia GPU testing the same benchmark.

6.1. VGG Benchmark

VGG, originally presented in [8], employs a very large model, as can be deduced from Figure 11 [4]. About 150 million weights are needed, but the model itself is relatively simple and is based on only a few types of layers.

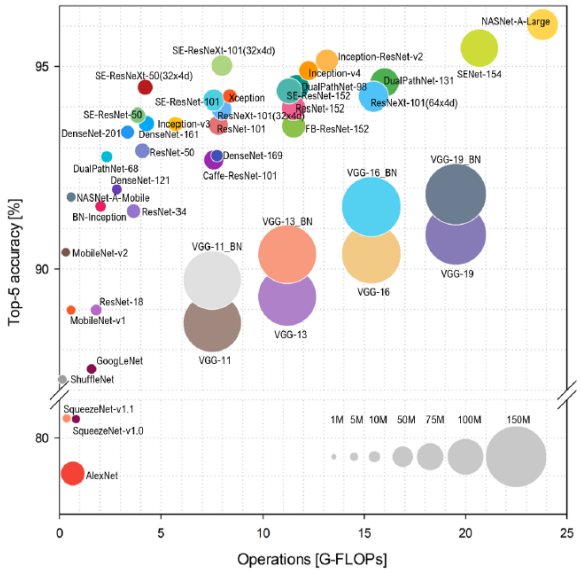

Figure 11. VGG-19 benchmark shown with many other model benchmarks. It contains 150M weights and requires 20 GOP per image frame.

When executing the VGG model (retrieved from [9]), RC64 needs to balance computing with massive data fetches. Inputs to the first layer are received at high speed over multiple SpaceFibre links. Inputs to other layers, as well as weights, are fetched from off-chip DRAM, and activations generated by the layers are written to the same DRAM.

Large models that may be useful for on-board Space ML applications are expected to also incur heavy I/O cost in parallel with high computing demands, similarly to VGG. Hence, VGG became the benchmark of choice for IE on RC64 processors.

Field	Description
Layer Type	0 - Conv1D, 1 - Conv2D, 2 – Dense, 3 – LocallyConnected1D, 4 – DepthwiseConv2D, 5 – Activation, 6 – AveragePooling1D, 7 – AveragePooling2D 8 – MaxPool1D, 9 – MaxPool2D
Fragment input volume	An array containing the fragment input dimensions
Layer input volume	An array containing the entire layer input dimensions
Input location	The input volume location in memory
Layer output volume	An array containing the layer output volume dimensions
Output location	The output volume location in memory
Kernel size	The convolution kernel dimensions (can be 4D)
Kernel location	The convolution kernel location in memory (buffer and offset)
Strides	Convolution stride (1D or 2D)
Padding	If the fragment is on the volume border, pass the required padding (1D or 2D)
Use bias	In case the fragment computes a point in an output feature of the layer (rather than an intermediate result), it's possible to add bias to the result
Bias location	Bias vector location
Apply activation	In case the fragment computes a point in an output feature of the layer (rather than intermediate result), it's possible to apply activation to the result
Activation type	0 – ReLu, 1 – Sigmoid, 2 - linear
Save output buffer	Output of current layer should be retained for future calculations (used in residual connections)

Figure 12. Model Tasked Format

6.2. VGG Performance

Figure 13 shows a snapshot of RC64 execution profiler when executing VGG-19 inference. The horizontal axis represents time, and the vertical axis indicates the number of active RC64 cores (out of a total 64 cores) at any point in time. As can be seen in the figure, all 64 cores are active (blue color) most of the time. As shown below, the benchmark manages to utilize about 80% of peak MAC performance.

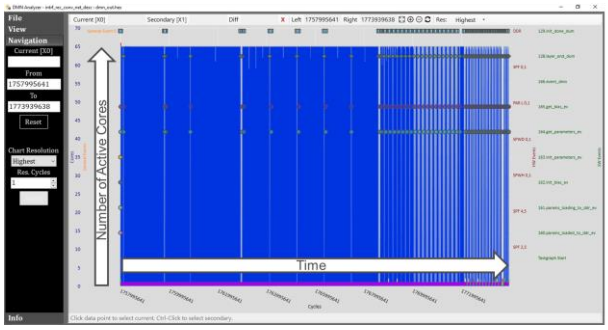

Figure 13. Execution profile of VGG-19 benchmark on RC64. Most of the time, all 64 cores are active.

Figure 14 presents a snapshot of the oscilloscope that measures instantaneous current consumption of the computing parts of RC64 (excluding SpaceFibre and DRAM interfaces). That current, consumed at 1.0V, is indicative of power consumption during the VGG benchmark inference execution. The different layers are readily distinguishable. The narrow vertical drops between layers indicate short pauses in processing, while tasks are being swapped. Power does not exceed the peak 4 Watt mark which is reachable only during maximum core utilizations. Total power, including I/O, is typically closer to 5 Watts (when all cores and all MACs are fully utilized and all I/O interfaces are active).

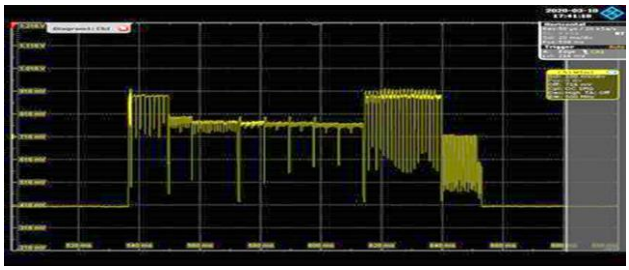

Figure 14. Power profile of RC64 while executing VGG-19 benchmark. Layers are clearly identifiable. Total power never exceeds 4 Watt.

Table 1 lists performance parameters. The first column shows RC64 results. Based on 65nm technology, RC64 consumes no more than 5W and processes 2.8 frames per second (FPS). As implied by Figure 11, each frame processed by VGG-19 inference requires 20 Giga Operations (20 GOP), hence 2.8 FPS implies 56 GOP/s. This execution rate is 80% of RC64 peak performance of 70 GOP/s. The bottom line indicates 0.56 FPS-per-Watt (FPS/W). Note that attempting to estimate ML inference performance by considering only peak performance may be misleading. A full VGG benchmark is more reliable for assessing ML performance.

The second column of the table speculates performance and power that may be achievable on Ramon Space's future roadmap product, RC256. If implemented on 16nm, the same 5W should lead to a conservative estimate of at least 25 FPS, or 5.0 FPS/W.

The last column is based on published results of benchmarking Nvidia Jetson Nano "low power, edge computing" GPU [3]. That GPU is also implemented using 16nm, making the comparison valid. Only 1.0 FPS/W is reported by Nvidia. Our conclusion is that RC64 architecture is more suitable for ML inference than that of the GPU. Moreover, the GPU is unsuitable for challenging missions in Space.

Table 1. VGG-19 benchmark performance of RC64, of a future roadmap scaling, and of Nvidia Jetson-Nano. The benchmark processes 224×224 image frames.

	Ramon Space RC64	Ramon Space RC256 (roadmap)	Nvidia Jetson Nano (non-Space)
Space Ready	Yes	Yes	NO
Process	65nm	16nm	16nm
Power	5 W	5 W	10 W
Frames Per Second	2.8 FPS	25 FPS	10 FPS
Perf/Power ratio	0.56 FPS/W	5.0 FPS/W	1.0 FPS/W

7. CONCLUSIONS

RC64 addresses all challenges facing execution of Machine Learning inference in Space: Long maintenance-free lifetime, high radiation, wide temperature span and numerous thermal cycles. A ML framework is implemented for on-board ML inference using RC64. Ground-based development converts standard ML models, and parallelizes the computations while considering the specific dimensions, sizes and operations of each layer. The parallelized model is transformed into an interpretable RC64 model, based on RC64 task-based programming model. The resulting RC64 ML model is beamed up to the on-board RC64-based Inference Engine, where it is interpreted. This approach offers high performance, power efficiency, and cyber security. Using VGG-19 benchmark and 224×224 pixel image frames, a 2.8 frames-per-second rate is achieved while consuming 5 Watts, implying 0.56 frames-per-second-per-Watt while working (practically forever) in the harsh conditions of Space.

ACKNOWLEDGEMENTS

The authors are grateful to the Israel Space Agency and the Government of Israel for their strong support of this activity.

REFERENCES

[1] Ginosar, R., P. Aviely, T. Israeli, and H. Meirov. RC64: High performance rad-hard manycore. In IEEE Aerospace Conference, 2016.

[2] Aviely, P., O. Radovsky and R. Ginosar. DVB-S2 software defined radio modem on the RC64 manycore DSP. In IEEE Aerospace Conference, 2016.

[3] Nvidia, Jetson Nano Deep Learning Inference Benchmarks, https://developer.nvidia.com/embedded/jetson-nano-dl-inference-benchmarks, undated.

[4] Bianco, S., R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. IEEE Access 6 (2018): 64270-64277

[5] Dai, W., & Berleant, D. (2019, December). Benchmarking contemporary deep learning hardware and frameworks: A survey of qualitative metrics. In 2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI) (pp. 148-155). IEEE.

[6] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi and J. Kepner, "Survey and Benchmarking of Machine Learning Accelerators," 2019 IEEE High Performance Extreme Computing Conference (HPEC), 2019.

[7] Fuketa, H., & Uchiyama, K. (2021). Edge artificial intelligence chips for the cyberphysical systems era. Computer, 54(1), 84-88.

[8] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[9] Keras on-line model for VGG, https://keras.io/api/applications/vgg/