IT UNIVERSITY OF COPENHAGEN

Department of Computer Science
KISPECI1SE - Master Thesis

Techniques for increasing efficiency of Intel Neural Compute Stick 2

Prepared by: Robert Bayer (roba@itu.dk)
Instructor: Pınar Tözün
Date: June 1, 2023

Abstract

The advancements in hardware allow for data processing closer to its source, such as sensors, which was traditionally performed in data centers. This approach has multiple advantages, such as decreasing latency overheads of data transfers and privacy. While the requirements and the constraints of the edge applications vary, the devices are expected to be highly efficient as they are usually run using limited harvested power. This work studies the performance of Intel Neural Compute Stick 2 and the possible optimizations for increasing its efficiency. We first compare the device to other edge devices used for the acceleration of machine learning inference workloads and characterize its placement in this space with regard to its computational power and power characteristics. Our results demonstrate that while the device provides good performance at lower power consumption in comparison to less specialized devices such as microcontrollers and GPUs, the newer specialized architectures, such as the CoralAI TPU, deliver similar performance at a fraction of the power consumption. We further study the performance of the Neural Compute Engine and SHAVE cores found on the device using fully-connected and convolutional layers. We show that the flexibility of the SHAVE cores comes at the cost of lower performance caused by large overheads attached to parallelism. The results demonstrate that the SHAVE cores should be utilized with caution and only if the operation is not already implemented on the Neural Compute Engine. We further analyze the effects of batching and show that this underutilized feature of the Neural Compute Engine can increase the throughput of inference on full-size models by up to an order of magnitude. This increase in performance comes from the amortization of the constant overheads of the operations and the data transfers.

Contents

1 Introduction

The developments in the hardware space led to an increase in the number of connected devices and the use of technology in more industries. Some of these industries rely heavily on deployments of sensors, which collect large amounts of data. This data was traditionally processed in data centers. However, the advancements in processors and the introduction of new specialized hardware architectures allow the processing to move closer to the source of the data, the edge. The processing close to the data has many advantages, including reduced latency connected with data transfers and privacy. Some of the deployments at edge rely on real-time processing that drives further actions, such as self-driving vehicles. Additionally, some deployments do not have the bandwidth to transfer the massive amounts of data collected by the devices and therefore need sophisticated methods for reducing the load on the network through smart filtering or other means of data compression.

The requirements and constraints of running such workloads at the edge vary immensely. The devices processing such data can include microcontrollers, which provide flexibility and a low power footprint while delivering relatively low performance due to the low specialization and degree of parallelism. More computationally heavy applications such as image processing, including classification and segmentation, however, have to leverage more specialized architectures ranging from edge-targeting SoC devices, such as the NVIDIA Jetson devices, or highly specialized ASICs, including CoralAI edge TPU or the Intel Neural Compute Stick 2.

The highly constrained nature of the deployment environment requires the highest degree of efficiency from the devices, which means providing the lowest latency/throughput at the lowest possible power footprint.

This work contributes to the field by studying the Intel Neural Compute Stick 2, an ASIC designed to accelerate deep neural network inference at the edge. Specifically, this work characterizes the placement of this device in the fast-growing space of edge-targeting accelerators through an application-specific benchmark against multiple other devices based on different architectures. Through the benchmark, we gain a deeper understanding of trade-offs connected to the use of this device, including its latency and power characteristics, such as its power draw under load.

After the comparison against other devices within the field, we analyze the device in isolation. We perform multiple micro-benchmarks to characterize the performance of the Neural Compute Engine, a deep neural network

accelerator on the device, and the SHAVE cores, a flexible part of the chip capable of running custom OpenCL kernels. Furthermore, we explore batching, an underutilized feature of the Intel Neural Compute Stick 2. At the end of our work, we use the knowledge gained through the micro-benchmarks to run inference using full-size neural networks to fully quantify the effects of the applied optimizations on real-life usage of the device.

The key contributions of this work are:

- Characterization of the performance of the Intel Neural Compute Stick 2 with respect to the other devices, commonly leveraged to accelerate workloads at the edge, through an application-specific benchmark.
- In-depth description of OpenVINO, a toolkit for optimization and runtime of inference using the Neural Compute Stick 2.
- Overview of the device's architecture, high-level optimizations available through OpenVINO, and the programming paradigm for the SHAVE cores.
- Performance and feasibility study of workload acceleration using custom OpenCL kernels running on the SHAVE cores, including characterization of effects of optimizations commonly leveraged on GPUs. This study uses a fully-connected layer as an example of a custom kernel.
- Analysis of the effects of batching on the throughput of the Neural Compute Engine.

2 Background

This section introduces the OpenVINO framework, a toolkit for deploying neural network inference on the Intel Neural Compute Stick 2, and its components, after which we introduce the Intel Neural Compute Stick 2 (NCS2) itself, including a description of the architecture of the MYRIAD X running on the device, its extensibility and the possible high-level optimizations applied through OpenVINO.

2.1 OpenVINO

OpenVINO [1] is a toolkit for the optimization and deployment of neural network inference on a range of Intel's devices, including CPUs, GPUs, FP-

GAs, and other hardware accelerators such as the NCS2. In addition to the devices produced by Intel, OpenVINO provides add-ons for deployment on ARM CPUs and NVIDIA GPUs.

In contrast to popular deep learning frameworks such as TensorFlow and PyTorch, OpenVINO focuses on inference only and uses the neural networks trained using these frameworks as the base of their inference engines. In addition to these two frameworks, OpenVINO also supports neural networks trained in Caffe, MXNet, PaddlePaddle, and Kaldi, as well as other frameworks, whose output can be converted to ONNX, an open format for the representation and exchange of neural networks.

The OpenVINO toolkit is composed of two major parts:

- **OpenVINO Development Tools** - Part of the toolkit responsible for converting the neural networks trained in other frameworks into OpenVINO's intermediate representation (IR).
- **OpenVINO Runtime** - Part of the toolkit responsible for running inference on top of the IR produced by the OpenVINO Development Tools on the variety of the supported hardware.

We now introduce these two parts of the toolkit in more depth.

2.1.1 OpenVINO Development Tools

The OpenVINO Development Tools is a set of utilities that make it easy to convert models developed in different frameworks and optimize them for deployment using OpenVINO.

Model Optimizer is responsible for converting the models to the OpenVINO IR. In addition to this conversion, the tool performs optional transformation passes over the IR for optimal execution on end-point target devices.

These transformations vary in complexity. However, the most common transformations include the compression of the default FP32 values to the more efficient FP16 data type, conversion of the data layout of the tensors, which can lead to improved performance due to a change in data access patterns (locality), and embedding pre-processing operations such as input normalization, commonly expected on data sets such as MobileNet.

In addition to these common transformations, the tool performs operation fusion, such as fusion of the batch normalization operation with the preceding convolutional or fully-connected layer. This layer can be decomposed

into multiplication and addition, which, together with the multiplication and the addition found in convolutional or fully-connected layers, create linear combination, therefore allowing for the fusion of these two layers into single multiplication and addition.

The IR is stored in two files (*.xml* and *.bin*), with the *.xml* file containing the layout of the graph of the IR and the *.bin* file containing the data tied to the nodes in the graph, such as the weights of the convolutional or the fully connected layers.

Benchmark Tool allows for easy benchmarking of the model inference performance on the end-point devices. This allows for comparison between the performance on different models or using varying parameters for controlling the performance of the hardware accelerators (NCS2-specific parameters described in section 2.2.3).

Post-Training Optimization Tool allows the users to quantize the models to 8 bits on supported devices (not applicable to NCS2), significantly improving the inference performance.

Accuracy Checker allows the user to check the predictive performance of the models before and after the quantization to 16 or 8 bits using the above-mentioned tools. This utility is especially useful before deployment on devices supporting multiple input data types.

Model Downloader and other Open Model Zoo tools , which offer example models, showcasing different applications and the supporting code for deploying these applications, including examples of input preparation and output handling.

2.1.2 OpenVINO Runtime

The OpenVINO Runtime is responsible for running the OpenVINO models produced by the OpenVINO Development Tools on the target hardware. Unlike the OpenVINO Development tools, which must be installed only on the development machine, the runtime must be installed on every host attached to the hardware accelerator or on the device performing the inference in the case of CPU.

In the background, the runtime takes the IR of the model and compiles the model to the operations specific to the hardware accelerators, such as OpenCL kernels in the case of GPUs.

In addition to MacOS and Windows, the runtime is also available for Linux, specifically on Ubuntu and Red Hat Linux distributions. For different distributions, the runtime must be built from sources, or users can leverage Ubuntu Docker images. To run the models, users have the option to interact with the runtime using the *Python, C++* and *C* APIs.

2.2 Intel Neural Compute Stick 2 (NCS2)

The NCS2 [2] is a hardware accelerator produced by Intel. It leverages the Intel Movidius Myriad X Vision Processing Unit (VPU) chip to accelerate the inference of convolutional neural networks.

Its small form factor and low power requirements make it a good candidate for deployment on edge. In addition, the communication and the power delivery between the host and the accelerator are handled by the onboard USB 3.0, allowing for the use of a variety of commodity hardware as the host device.

In addition to the onboard vision accelerators and 16 MIPI lanes responsible for streaming video data from multiple cameras and performing subsequent operations on these streams, including optical flow and stereo depth (not exposed on the NCS2), the VPU chip also contains a Neural Compute Engine and 16 SHAVE cores to accelerate deep neural network inference, delivering a total peak performance of 4 trillion operations per second (TOPS). The chip also features 2.5 MB of centralized on-chip memory allowing for 400 GB/s of internal bandwidth [3]. Figure 1 shows the architecture of the Myriad X chip.

Neural Compute Engine is a dedicated on-chip accelerator for deep neural networks delivering 1 TOPS peak inference performance. The accelerator was built and optimized for 16-bit floating point operations, which is the default option for deployment on the NCS2.

SHAVE Cores are programmable 128-bit VLIW vector processors optimized for vision processing workloads. Figure 2 shows the architecture of the SHAVE cores. Each core contains arithmetic units and register files for inte-

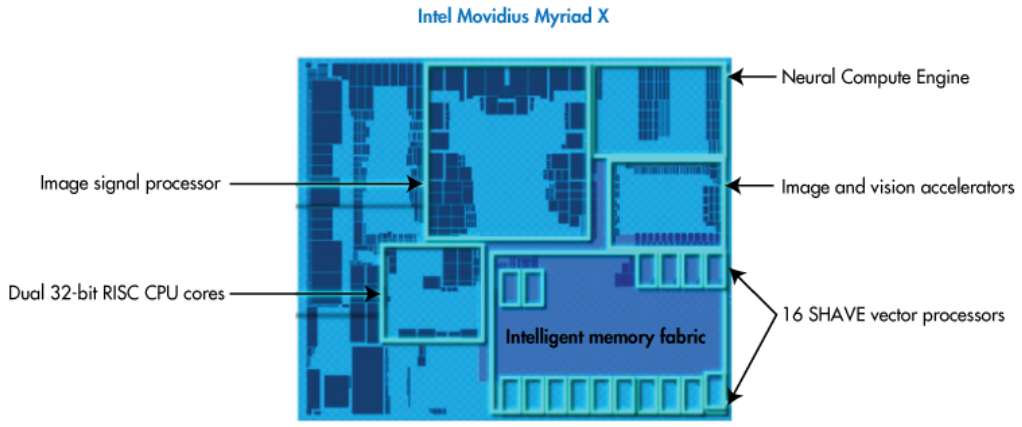


Figure 1: Diagram of the Intel Movidius Myriad X architecture. [4]

gers, scalars, and vectors of size 128 bits. In addition to the on-core register files, each core is connected to a 128 KiB slice of SRAM shared between the cores through a bus.

While the most common operations found in neural networks are included in the runtime, the users can extend the included operation set by writing custom OpenCL kernels to be run on these SHAVE cores. The following sections will detail the process of extending the set of operations and specifically writing custom OpenCL kernels.

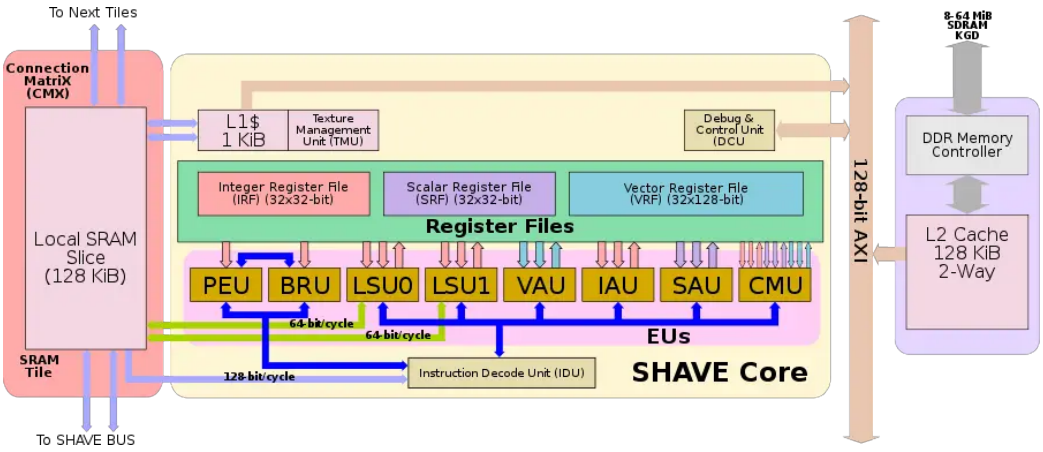


Figure 2: Diagram of the SHAVE core architecture. [5]

2.2.1 Extensibility of the VPU

With the fast-evolving field of research in new deep neural network architectures, it is important for hardware accelerators to be flexible enough to accommodate these new advances. The VPU architecture and the OpenVINO toolkit allow extending the set of operations supported out of the box or adding custom implementations of existing operations.

The addition of a new operation involves two steps:

1. Definition of the operation semantics, including the input ingestion and output of the operation. This includes mapping the operation represented in ONNX or representation tied to the framework used for training the model to OpenVINO's IR.

2. Implementation of this operation. In the case of NCS2, this requires the creation of the custom OpenCL kernel or decomposition of the operation into multiple smaller operations already available in OpenVINO's operations set.

In the case of custom re-implementation of an existing operation, the users must create the custom OpenCL kernel and supply the bindings between OpenVINO's IR and the compiled OpenCL kernel to be used during the inference.

2.2.2 Custom OpenCL Kernels

The implementation of the custom kernels is written in OpenCL version 1.2, with support for the *half float* extensions. These kernels are compiled using the proprietary OpenCL compiler provided by ComputeAorta included in the OpenVINO Runtime distribution (supported only in version 2022.1).

To tie the IR topology to a custom implementation, the users provide a configuration file written in XML, an example of which follows:

```
<CustomLayer name="FullyConnected" type="MVCL" version="1">
    <Kernel entry="transpose">
        <Source filename="fully_connected.bin"/>
        <Parameters>
            <Tensor arg-name="src_data" type="input" port-index="0" format="BF"/>
            <Data arg-name="weights" type="data" source="weights" format="ANY"/>
```

```
7             <Tensor arg-name="weights_transposed" type="
    output_buffer" port-index="0" dim="output,0" size="
    1024*1024*2"/>
8             <Scalar arg-name="IW" type="int" port-index="0"
    source="I.X"/>
9             <Scalar arg-name="OW" type="int" port-index="0"
    source="O.X"/>
10        </Parameters>
11        <WorkSizes dim="output,0" global="B,F,1" local="
    16,16,1"/>
12     </Kernel>
13 </CustomLayer>
```

Listing 1: Example configuration file mapping *FullyConnected* operation in OpenVINO IR to compiled kernel in the *fully_connected.bin* file.

This example ties the *FullyConnected* operation found in the model's IR to the compiled custom kernel found in the *fully_connected.bin* file. In addition to this, it defines the parameter mapping between the IR node and the OpenCL kernel, including the input tensors and the output buffer. The file also defines the work group sizes to be associated with this kernel, using either constant values or dimensions of either the input or output tensors.

The definition of the work group sizes translates to the number of work groups and the number of work items inside each of these work groups. These are terms standardly used in the context of OpenCL. In the context of CUDA, these could be translated as threads (work items) and thread blocks (work groups). These parameters effectively control the degree of parallelism in the kernel.

The execution of the code on the SHAVE cores and the meaning of these terms in the context of the SHAVE cores differs from the standardly used definitions in the context of the GPUs. The relatively low degree of parallelism of the device in comparison to the GPUs means that each work group maps to a single SHAVE core, on which the work items are iterated over sequentially unless the compiler detects a use of the *globalID* indicator of the work item (coalesced data accesses), after which the compiler automatically vectorizes its execution.

2.2.3 Inference Optimization Techniques in OpenVINO

Before resorting to custom re-implementation of available operations, the users should leverage high-level optimization techniques available in Open-

VINO. These include:

Data transfer pipelining can be leveraged to amortize the transfer times between the host and the hardware accelerator. Each device implements an on-device queue, which can store data for multiple inference requests and retrieve them at its own pace.

Asynchronous execution of inference request consists of the host thread sending data to the device and being notified once the execution is done. The time between the successful transfer of the input data and the callback from the device upon finishing its execution can be used to prepare input data for the subsequent request handled by that thread.

Batching is a technique commonly used on massively parallel hardware, such as GPUs. While the NCS2 has support for handling batched input data, its use is not promoted.

Performance hints is a parameter that exposes the option to hint to the device whether to optimize the inference to lower latency or increase throughput. In the background, this option changes multiple parameters depending on the target device. In the case of NCS2, the default performance hint value is throughput and sets the number of optimal inference requests to 4. Note that this does not affect the latency or throughput unless multiple inference requests are utilized.

3 Related Work

Machine learning at edge is now a well-established topic and is gaining even more traction as the advancements in edge-targeting hardware open new possibilities of models able to run in resource-constrained environments.

These works include vision and survey papers [6, 7, 8, 9], which provide a valuable overview of the field, in both the edge-targeting hardware, including emerging hardware accelerators, but also software, such as different frameworks for deploying machine learning applications or optimizations for running these applications on the resource-constrained devices.

These applications are accelerated using devices based on different architectures, chosen based on the application requirements and the constraints

posed by the deployment environment. These affect the size, power, and thermal requirements, leading to a high need for efficiency in all these aspects. Multiple works introduce and compare devices such as microcontrollers, edge-targeting SoCs leveraging the power of GPUs or other hardware accelerators, such as TPUs and VPUs [10, 11, 12], while others focus on creating benchmarks specific to machine learning at the edge to facilitate these comparisons [13, 14].

The Intel Neural Compute Stick 2 is the target hardware for many machine learning inference applications [15, 16, 17], but also a topic of in-depth exploration and optimization [18, 19, 20]. In addition to the newer version of the device, multiple works also explore the original version of the device (Intel Neural Compute Stick 1) [21, 22, 23, 24, 20].

While the topic of implementing custom kernels in OpenCL is, to our knowledge, not yet explored outside of Intel, some works leverage the not publicly accessible Myriad X SDK, through which they optimize and analyze the performance of the Myriad X chip [19].

In addition to the feasibility study of the deployment of custom kernels on this device, we complement these works by providing an application-specific comparison between a wide range of edge devices, including analysis of latency and the power characteristic of the devices under load, in addition to an in-depth analysis of the performance of the Intel Neural Compute Stick 2 and its Neural Compute Engine and the SHAVE cores. Furthermore, we give an overview of one of the possible deployment tools in the field of machine learning at the edge, the OpenVINO toolkit.

4 Fully-connected Layer

Together with the convolutional layer, the fully-connected layer is the most utilized layer in neural network architectures. It is a layer in which each neuron applies a linear transformation to the input vector. It is characterized by the following equation:

$$y = xW^T + b \tag{1}$$

, where x is the input vector, W is the weight matrix, b is bias, and y is the output vector, the result of this linear transformation.

The addition of the bias is often fused into the matrix-vector multiplication by prepending the weight matrix with an extra row, which contains the

elements of the bias vector, and prepending the input vector with a 1.

In this section, we further explore the matrix-vector multiplication and the optimizations of its computation on parallel hardware and then explore the implementation of the batched version of the layer using matrix-matrix multiplication [25].

4.1 Matrix-vector Multiplication

A simple matrix-vector multiplication is presented in fig. 3. The matrix-vector multiplication consists of performing a dot product of the input vector (x) with each of the columns of the weight matrix (W), which can be represented by a vector of size K, matching the size of the input vector. The result of the product corresponds to a new entry in the output vector (y). The size of this output vector matches the second dimension of the matrix W (dimension M not shared with the input vector), as the dot product is iterated over M times.

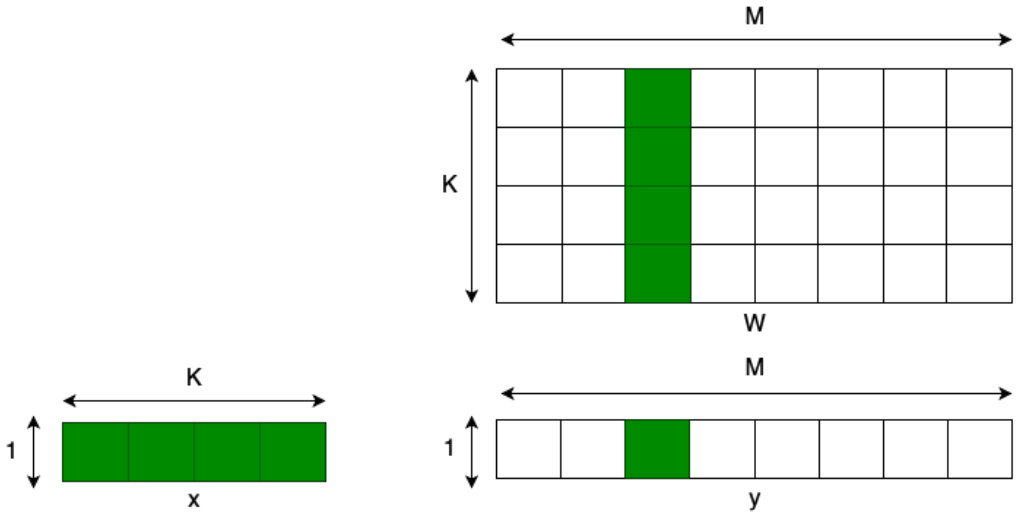


Figure 3: Example of matrix-vector multiplication.

This algorithm's simplest (naïve) parallel version runs in M threads, corresponding to the total number of vector dot products. Each thread contains an accumulator initialized to 0 or the bias value. Furthermore, it contains a sequential for-loop, which iterates over the common dimension and performs a multiply-accumulate (MAC) operation for the two scalar values. These

scalar values are loaded from global memory (DRAM). As there is no data reuse of the elements of the weight matrix between threads, the use of scratchpad memory does not bring benefits.

Now we introduce types of optimizations that can improve on the performance of this naïve implementation.

Coalesced access is a type of memory access in which threads in a work group access memory simultaneously while accessing consecutive elements in memory, as shown in fig. 4. The threads in this work group collaborate to load this data to local / scratchpad memory, in case of data reuse between the threads or into the threads' register otherwise.

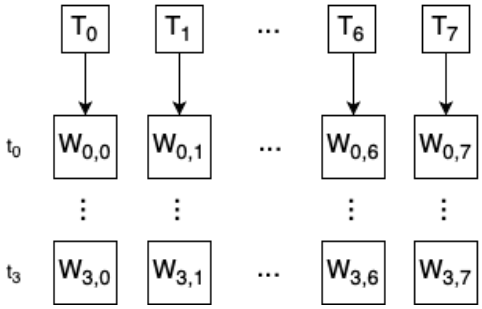


Figure 4: Example of coalesced access, where threads T_0 to T_7 access consecutive elements of matrix W in parallel (in this example whole row), only after which the threads access the next values.

Tiling is a type of optimization that breaks up the problem to achieve better memory access characteristics. In the case of matrix-vector multiplication, we can increase the number of work items in a work group, which will load a block of the weight matrix into local memory shared by the threads in a coalesced fashion, as shown in fig. 5.

The use of tiling increases computational intensity (number of instructions per memory access), as for the input vector, we access the global memory only once per work group while still using it once in each thread. The rows of the tile in the matrix are loaded sequentially. While this optimization increases the computational intensity for the input vector, this remains the same for the weight matrix, as tiling does not bring any data reuse

there. Matrix-vector multiplication is a special case of matrix-matrix multiplication, where the size of the input matrix along one axis is 1. Unlike in matrix-vector multiplication, matrix-matrix multiplication would increase the computational intensity in the weight matrix by parallelizing computation in the second dimension.

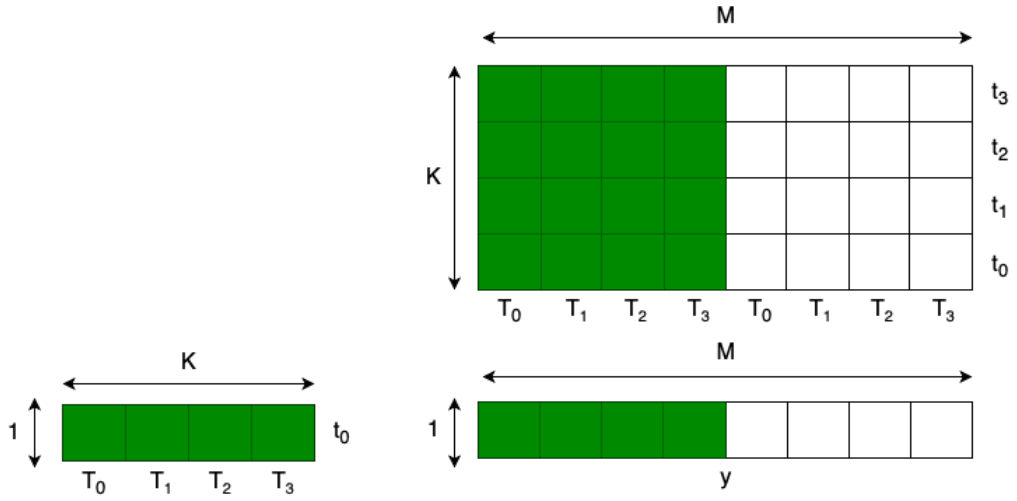


Figure 5: Example of tiled matrix-vector multiplication, where threads T_0 to T_3 collaborate on loading the highlighted tile of the matrix and input vector into scratchpad memory.

Increasing work per thread decreases the overall number of threads and increases utilization of faster register. Unlike in matrix-matrix multiplication, however, this optimization also does not bring any improvement in data reuse.

Vector operations leverage SIMD instructions within the threads. This optimization increases throughput and decreases the latency of the layer.

4.2 Matrix-matrix Multiplication

By stacking multiple input vectors into a matrix, we create a batched version of the fully-connected layer, implemented as a matrix-matrix multiplication. While this approach does not decrease the total number of MAC operations

compared to the non-batched approach, introducing a second dimension of the input vector allows for parallelism in this second dimension.

This second dimension for parallelism leads to multiple advantages we alluded to in the previous section. This includes the increase in data reuse of the tiled implementation. The whole tile of both the input and weight matrices can be loaded entirely in parallel, and each of the elements of the matrices reused T times, where T is the size of the tile.

Furthermore, the matrix-matrix multiplication allows for an increase in the number of elements processed by a single thread. In addition to lowering the total number of threads for this operation, this optimization allows for higher use of faster registers for performing the MAC operations through register tiling optimization.

5 Benchmark against other Devices

By comparing the NCS2 against other devices in the space of edge-targeting machine learning accelerators, we can gain a baseline for its performance and characterize its position in the space. We first describe the methodology and the experimental setup of the benchmark, after which we present the results of the described experiments, followed by a discussion of the results.

5.1 Methodology and Experimental Setup

This section describes the methodology and the experimental setup of the benchmark, including the introduction of the devices under test, workload definition, and description of the metrics used to evaluate the performance of the devices.

5.1.1 Devices under Test

The set of devices we conducted experiments on contains representative devices based on different architectures, including a microcontroller, edge SoC leveraging GPU for the workload acceleration, and multiple devices containing ASICs, including the NCS2 and representatives of the CoralAI family of edge TPUs.

ARM Cortex-M7 Microcontroller [26] This device represents the device with the lowest degree of specialization. It contains a single ARM Cortex-M7 core, on which we run a bare-metal application. The low-power CPU delivers low performance compared to the rest of the devices. However, it has very low power requirements, making it the best choice for running simple machine learning tasks in deployments with a very low power budget. Table 1 shows the device's full specifications.

Deployment of machine learning workloads on this device leverages the TensorFlow Lite for Microcontrollers framework [27]. This framework was designed to run on very resource-constrained hardware, with the runtime fitting in 16KB of memory, the most significant limiting factor for deployment on this type of device.

While this device supports floating point operations, we limit the precision of the neural network to 8-bit integers, which further reduces the memory requirement and lowers the expected latency. The choice of the 8-bit precision also means that no input scaling is needed, further reducing the preprocessing time.

Furthermore, we leverage CMSIS-NN kernels [28], a collection of neural network kernels explicitly crafted for running on ARM Cortex-M class devices. The use of these kernels further aids in minimizing the memory footprint and latency of inference. The deployment of the TensorFlow Lite for Microcontrollers utilizing the CMSIS-NN kernels was facilitated using the X-CUBE-AI [29], which provides a user-friendly interface, making the deployment easier.

Processor	ARM Cortex-M7 @ 300 MHz
SRAM	1 MB
Flash	2 MB
Dimensions	138.0 x 99.7 x 17.7 mm
Mass	59.2 g

Table 1: Specifications of the ARM Cortex-M7 microcontroller. The dimensions and the mass are based on the STM32 NUCLEO-H745ZI-Q development board.

NVIDIA Jetson Nano [30] This embedded SoC device leverages GPU and its high degree of parallelism to accelerate machine learning workloads

at the edge. It takes inspiration from the success of the GPU-accelerated servers used for machine learning workloads. It translates them into the edge environment, delivering lower performance but an order of magnitude lower power draw than its full-size counterparts.

While providing a high degree of parallelism, this device is more general-purpose than those leveraging ASICs, such as the NCS2. In addition to performing deep neural network inference, this device can perform full training and preprocessing operations, which can take advantage of the high degree of parallelism. The full specifications of the device are shown in table 2.

The inference deployment is performed using either the TensorFlow framework [31] or the TensorRT framework [32], with the TensorRT framework delivering a more optimized engine. Like OpenVINO, TensorRT is a framework that creates an optimized inference engine for neural networks trained outside this tool. It targets NVIDIA GPUs and creates an optimized inference engine, which can take advantage of all of the capabilities of the device, which are mostly unavailable through use of other frameworks including the TensorFlow.

The compute capability of this device corresponds to 5.3, which means that the device supports 16-bit floating point operations, which is the only alternative to the default slower and less memory-efficient 32-bit floating point operations.

The results presented in later sections were measured on a setup leveraging the TensorRT framework with 16-bit floating point operations and batch size of 16, as this combination was found to deliver the most efficient setup. Furthermore, the device was configured to operate at 5W by disabling two of its four ARM cores to better align with the other devices. Because the network uses 16-bit precision, the input must be scaled to fit in the $[0, 1)$ range. This operation was, however, fused into the network and accelerated by the GPU.

Intel Neural Compute Stick 2 The NCS2 device (described in more detail in section 2.2) was connected to a Raspberry Pi model 3 [33], which was used as a host for this accelerator. The host specifications, and the physical dimensions and the interfaces of the NCS2 are shown in tables 3 and 4, respectively.

The NCS2 device was set up with the OpenVINO's recommended settings (described in detail in section 2.2.3), which include asynchronous executions

CPU	Quad-core ARM A57 @ 1.43 GHz
GPU	128-core Maxwell
GFLOPS	472
RAM	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	64 GB SD card
Dimensions	100 x 80 x 29 mm
Mass	141 g

Table 2: Specifications of the NVIDIA Jetson Nano. The dimensions and mass are based on the developer kit version.

with four independent inference requests and throughput performance hint.

As with the NVIDIA Jetson Nano, the 16-bit precision of the network requires scaling to the range of $[0, 1)$, which was also fused with the network and executed on the accelerator.

CPU	Quad Core Broadcom BCM2837 @ 1.2GHz
RAM	1 GB
Storage	32 GB SD card
Dimensions	100 x 80 x 29 mm
Mass	42 g

Table 3: Specifications of the Raspberry Pi model 3.

Interface	USB 3.1, USB 2.0
Dimensions	72.5 x 27 x 14 mm
Mass	53.4 g

Table 4: Specifications of the Intel Neural Compute Stick 2.

CoralAI TPUs This family of devices leverages the CoralAI edge TPU [34], which is a hardware accelerator created to accelerate machine learning inference at the edge. It comes in multiple form factors, including:

- CoralAI Dev Board Micro [35] - an embedded device that uses the Cortex-M7 (comparable to the one used in the first device) and the Cortex-M4 cores. Unlike the rest of the devices in this family, this is the only device that runs FreeRTOS [36].

- CoralAI Dev Board Mini [37] - an embedded device that uses four larger ARM Cortex-A53 cores. Additionally, this device provides more memory and runs a fully-featured Mendel OS (Debian derivative) [38].
- CoralAI USB accelerator [39] - an accelerator with a form factor similar to NCS2. This device does not have a dedicated host and can be plugged into a dedicated host device (Raspberry Pi model 3 used to ease comparison to the NCS2.)

The full specification of these devices are shown in tables 5 and 3.

These devices accept models trained in TensorFlow and compiled through their dedicated compiler [40]. These devices only support 8-bit integer operations. Therefore, all models used on these devices were quantized to this precision. This also further reduces the latency by omitting the input rescaling operation.

	CoralAI Dev Board Micro	CoralAI Dev Board Mini
CPU	ARM Cortex-M7 @ 800 MHz, ARM Cortex-M4 @ 400 MHz	Quad-core ARM Cortex-A35 @ 1.5 GHz
RAM	64MB	2 GB
Storage	128MB NAND	8 GB eMMC
Dimensions	65.0 x 30.0 x 6.8 mm	64 x 48 x 14.6 mm
Mass	10.4 g	25.5 g

	TOPS	Interface	Dimensions	Mass
CoralAI USB accelerator	4	USB2	65 x 30 mm	4.3 g

Table 5: Specifications of the CoralAI Dev Board Micro and CoralAI Dev Board Mini, as well as the CoralAI USB accelerator. The CoralAI Dev Board Mini's and CoralAI Dev Board Micro's physical dimensions and weight include the on-board TPU, and TOPS is teraoperations per second.

5.1.2 Workload

The workload for this test was modeled after a real-world application, which consists of a land cover classification of satellite imagery. To simulate this application, we perform a 5-class classification on RGB images of size 4512x4512 pixels.

For this, we fine-tuned a MobileNetV1 model [41] on Flowers dataset [42]. MobileNet is a family of models, with the architecture specifically designed for deployment at the edge. Even though there are newer versions in the family of models, version 1 allows the user to scale the models' size down much further than the newer models, which is an important factor mainly for testing the most resource-constrained devices under test. This model can be scaled using the depth multiplier parameter, which adjusts the total number of layers in the network. The available options of pre-trained models are with the depth multiplier of 0.25, 0.5, and 1.0, all of which are included in our tests.

The models were pre-trained on the ImageNet dataset [43, 44] and fine-tuned on the Flowers dataset. Both of these datasets consist of RGB images of size 224x224 pixels. To align with this expected input size, we divide the 4512x4512 pixel images into 400 patches of this size, which are inferred individually. This step was also taken because inference on the full-sized image would be prohibitively expensive and exceed most of the devices' memory requirements. Furthermore, this has a favorable side-effect since the classification of the separate patches also acts as a coarse-grained image segmentation.

The results of the tests are reported as the average of 10 runs of classification of the 400 patches of the randomly generated 4512x4512 image and include the preprocessing steps, including the division into patches except for the Cortex-M7 microcontroller, which could only fit a single 224x224 pixel image into memory and the CoralAI Dev Board Micro, which could only fit quarter (100 patches) of the image into memory. The reported results for these two devices are extrapolated to represent the inference of all 400 patches of the image.

The division of the image into patches is performed in a separate thread on the host of the NVIDIA Jetson, using the inference request threads on the host of the OpenVINO and synchronously in the main thread for all devices accelerated by the CoralAI TPU.

The source code for the benchmark is available online [1].

[1] `https://github.itu.dk/roba/Edge-benchmark`

5.1.3 Metrics

To compare the performance of these devices on the studied workload, we use multiple time- and power-related metrics:

Latency Measures time to infer all of the 400 patches of the full-size image. The results are reported in seconds.

Power consumption Measures the total amount of power consumed during inference of the image. This shows the overall efficiency of the device as the metric combines the latency and average power draw of the inference, as shown in eq. (2).

$$E = \bar{P}(t/3,600) \tag{2}$$

The results for this metric are presented in milliwatt-hours (mWh).

Peak power draw Measures the maximum power draw of the device during the inference of the image. This metric is essential for design considerations of the power delivery for the device. The peak power draw is presented in Watts (W).

The power consumption and the peak power draw were measured using the Otii Ace Pro [45], a precision power supply and power analyzer.

5.2 Results

The results presented in this section show the latency, power consumption, and the peak power draw of the devices during the inference of a full-size image (400 patches) using models with increasing size (depth multiplier).

5.2.1 Latency

Figure 6 presents the per-image inference latency of the application-specific workload described in the previous section. The results of the benchmark vary based on the depth multiplier used to control the size of the model.

Even the smallest of the models posed high memory requirements for the ARM Cortex-M7 microcontroller. The model with the depth multiplier of 0.25 was the only model that could fit on this device. Furthermore, even

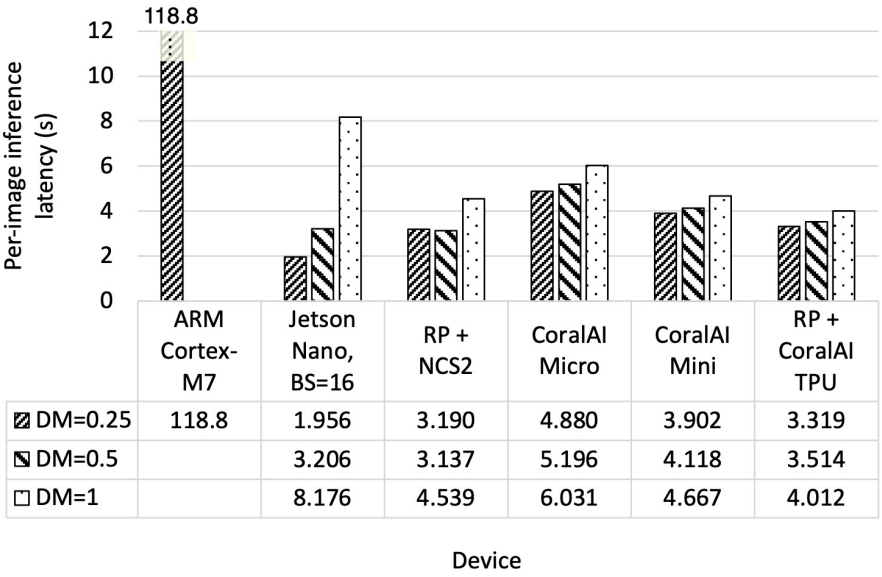


Figure 6: Latency of inference on a full-size image with different scaling factors (depth multiplier = DM) of the MobileNetV1 model as measured on corresponding devices. A batch size (BS) of 1 is used unless stated otherwise.

with the scaled-down model, the device is unable to hold the entire image in memory and can, therefore, only process a single patch at a time. This means that even if the device received the image already patched, the device would have to rely heavily on external storage to fetch the input data before an inference of each sample. This device's low degree of specialization and parallelism leads to 2 orders of magnitude higher latency than the rest of the devices.

The NVIDIA Jetson Nano performs the inference using the smallest model at the lowest latency. However, the device's performance on this workload deteriorates with the larger models due to its inefficient scalability. After scaling to the model with the depth multiplier of 0.5, the device still performs well while being surpassed only by the NCS2. Scaling to the largest model, however, shows a significant decrease in performance, leading to the highest latency of the devices that could perform the task.

The NCS2 showed improved scalability in comparison to the NVIDIA Jetson Nano. Using the smallest model, the performance of this device was surpassed only by the NVIDIA Jetson Nano while showing the lowest latency

for inference using the model with a depth multiplier of 0.5. In the case of inference using the largest model, the device achieves the second lowest latency, surpassed by the CoralAI USB accelerator using the same Raspberry Pi host, clearly showing the better scalability of this accelerator.

The CoralAI accelerators show the overall best scalability to larger models. They, however, show a considerable overhead, visible in the case of the two smaller models, where the devices achieve lower performance than the NVIDIA Jetson Nano or the NCS2. Overall, the CoralAI TPU hosted by the most powerful CPU found in the Raspberry Pi shows the highest performance, as the CoralAI Dev Board Micro and Mini suffer from larger preprocessing overhead, connected to the division of the full-sized image into the smaller patches, most notably visible in the case of very low power microcontroller hosting the CoralAI TPU on the CoralAI Dev Board Micro.

5.2.2 Power Consumption

While latency is an important metric for applications requiring real-time processing, after the devices fulfill the maximum acceptable latency, the power-related metrics become more important. The fig. 7 shows the power consumption of the devices under test. This metric shows the power efficiency of devices and can help decide between multiple devices in case multiple fulfill the latency requirements of a workload at hand.

Even though the Arm Cortex-M7 microcontroller draws the least amount of power on average, the high inference latency presented in fig. 6, leads to higher power consumption for inference of the full-size image than any other configuration of devices and models tested.

The low level of specialization of the NVIDIA Jetson Nano, in comparison to purpose-built ASICs found in the NCS2 and CoralAI TPU, leads to higher average power draw and, therefore also, higher power consumption during inference. The device shows the worst power characteristics than the rest of the device during inference using the two larger models while showing a power consumption lower than the NCS2 during the inference using the smallest model, caused by the significantly lower latency in this configuration.

In the two larger models, the NCS2 shows significantly lower power consumption compared to the NVIDIA Jetson Nano. It, however, shows power consumption significantly higher than the devices leveraging the CoralAI TPU accelerator.

The devices leveraging the CoralAI TPU for acceleration of the workload

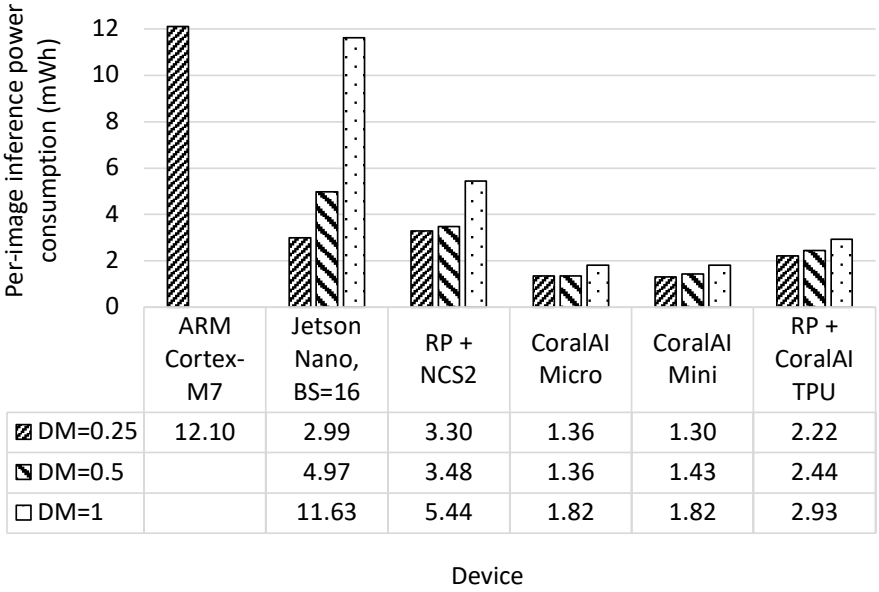


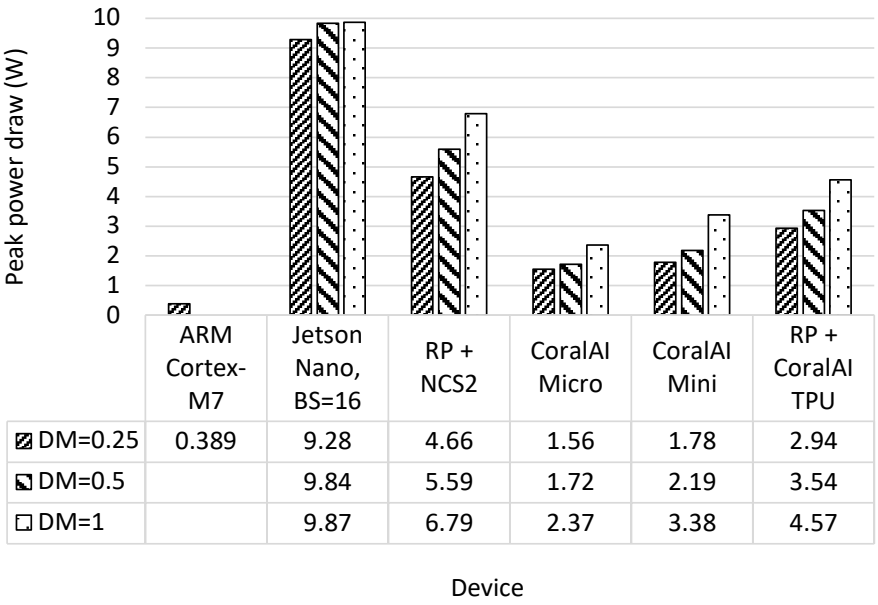
Figure 7: Power consumption of inference on a full-size image with different scaling factors (depth multiplier = DM) of the MobileNetV1 model as measured on corresponding devices. A batch size (BS) of 1 is used unless stated otherwise.

show the lowest overall power consumption. The choice of host CPU highly influences the average power draw of these devices. The fig. 7 shows an inverse trend compared to the results shown in the fig. 6. The devices with the less capable CPUs show better performance in this metric, while the device hosted by the more powerful Raspberry Pi consumes significantly more power during the inference of the full-size image, even though it can do so in a shorter time. Additionally, we can see that even though the NCS2 achieved lower latency in two of the three configurations compared to the CoralAI TPU hosted by the same device, the choice of the CoralAI TPU for acceleration leads to significantly better characteristics.

5.2.3 Peak Power Draw

The devices running inference at the edge are met with a strict power budget, as they most commonly draw power from batteries and are not connected

power grid but rather rely on energy harvesting. An important consideration when designing such a system is, therefore, power delivery. Figure 8 shows the peak power draw of the devices during the inference.



	ARM Cortex-M7	Jetson Nano, BS=16	RP + NCS2	CoralAI Micro	CoralAI Mini	RP + CoralAI TPU
DM=0.25	0.389	9.28	4.66	1.56	1.78	2.94
DM=0.5		9.84	5.59	1.72	2.19	3.54
DM=1		9.87	6.79	2.37	3.38	4.57

Figure 8: Peak power draw of of each device with MobileNetV1 model of different sizes (depth multiplier = DM). A batch size (BS) of 1 is used if not stated otherwise.

As mentioned earlier, the Arm Cortex-M7 draws a minuscule amount of energy compared to the rest of the device. This, however, comes at the cost of low performance. Furthermore, the figure shows that the high specialization of the devices leveraging ASICs for acceleration leads to higher efficiency and low peak power draw. The NVIDIA Jetson Nano, while providing a high degree of parallelism, is less specialized than these devices and, therefore, less efficient, with the peak power draw during the inference reaching up to almost an order of magnitude higher than the highly specialized ASICs. Furthermore, the newer architecture found in the CoralAI TPUs is more efficient than the older architecture found in the NCS2, reducing the peak power draw of these devices. These differences are even more pronounced when paired with more efficient CPUs for hosting these accelerators.

5.3 Discussion

Although the NCS2 does not provide the highest performance out of the devices tested in most of the configurations, it bridges the gap between the NVIDIA Jetson Nano, which, while providing the best performance for inference using the smallest model, fails to scale efficiently in comparison to the more specialized devices, and the devices leveraging the CoralAI TPU, which while providing the best scalability, mainly visible in the inference using the model with the highest scaling factor, have a large overhead for the inference using the smallest model.

On the other hand, the difference in performance between the Raspberry Pi combined with the NCS2 and the CoralAI TPU accelerator is minor compared to the efficiency differences between the two accelerators. The newer architecture found in the CoralAI TPUs, leads to power consumption during the inference of a sample up to 46% lower compared to the NCS2, including the non-negligible power consumption of the host device.

6 In-depth Analysis of the Device and Custom Kernels

After benchmarking against other devices in the edge ecosystem, we conduct an in-depth analysis of the NCS2 and its performance. This will give us a deeper understanding of the device's inner workings. Furthermore, we explore the viability of writing custom kernels and compare their performance to the already available implementations in OpenVINO through microbenchmarks. Finally, we examine how the findings gained through these microbenchmarks transfer to inference using a full-size deep neural network.

6.1 Methodology and Experimental Setup

In this section, we describe the methodology and experimental setup for the analysis of the NCS2. Specifically, we describe the device under the test, including the specific configurations used throughout the tests, describe the workload used to examine the device, and finally, the metrics used to evaluate the results of the benchmarks.

6.1.1 Device under Test

To conduct these experiments, the NCS2 was hosted by a desktop machine with specifications shown in table 6.

This is a machine running PopOS 22.04 [46], with the OpenVINO development toolkit and the runtime running inside of a Docker container. The NCS2 was attached to the desktop machine and the container through a USB3.0 connection.

CPU	8-core Intel Core i7-4790 @ 3.6 GHz
L1d$	128 KB
L1i$	128 KB
L2$	1 MB
L3$	8 MB
RAM	32 GB DDR3L
Storage	512GB SATA SSD
USB	3.0, 5 Gbit/s

Table 6: Specifications of the desktop machine used as a host for the NCS2.

6.1.2 Workload

To gain a deeper understanding of the device at hand in addition to the knowledge presented in section 2.2, we perform multiple microbenchmarks, which test the performance and the scalability of the on-board Neural Compute Engine.

The benchmarks focus on the performance of the NCS2 at running two of the most common layers found in deep neural network architectures, the fully connected layer and the convolutional layer, in isolation.

The weight matrix of the fully-connected layer is square such that the input and output shapes remain the same. The layer was tested with configuration with input vectors of size 128, 256, 512 and 1024.

The convolutional layers are based on the VGG-11 architecture [47] and mimic the first layer of the model (layer A) and a layer in the middle of the model (layer B). The kernel and stride sizes of the layer were fixed to 3x3 and 1, respectively. These layers differ in the number of input and output channels, with layer A accepting 3 channels and outputting 64 channels and layer B accepting 64 channels and outputting 256 channels.

The input to the convolutional layers is modeled after the CIFAR-10 [48] (small) and the ImageNet (large) dataset, with its size affecting the computational requirements of both types of convolutional layers.

The results of these microbenchmarks further serve as a baseline for the analysis of the performance of custom implementations of the fully connected layer, written in OpenCL and run on the SHAVE cores of the NCS2 device. Through these results, we assess how the incrementally added optimizations of this operation (detailed description in section 4), commonly used in programming massively parallel hardware, transfer to the SHAVE cores and how they affect the performance.

Furthermore, we assess the performance of the underutilized batching option on this device using the Neural Compute Engine and the OpenCL kernels running on the SHAVE cores. In connection with this, we explore the overheads attached to invoking the operations running on either of the accelerators.

Both the fully connected layers and the convolutional layers were run with randomly initialized weights on randomly generated input of varying sizes, as the value of these does not affect the performance. The results of all of the microbenchmarks are presented as an average of 1000 runs.

The inference of the samples was run synchronously in order to aid reasoning about the data transfer overheads.

Finally, we compare the findings gained through the microbenchmarks to running of inference on full-size deep neural network models. The first model is a feed-forward neural network with three fully-connected layers and ReLU activations, accepting grayscale images of size 28x28, emulating a small MNIST-like dataset [49].

In addition to the feed-forward neural network, we benchmark the device's performance on the Resnet-18 architecture, which accepts CIFAR-10-sized RGB images of size 32x32. This model has 11,511,784 trainable parameters, which is more than an order of magnitude higher than the 832,101 trainable parameters of the MobileNetV1 with a depth multiplier of 0.5.

The results for benchmarks using these models show averages of 1000 runs. The inference of these samples was run asynchronously with four concurrent inference requests in order to show the best achievable performance.

Source code of these benchmarks is available online [2].

[2] `https://github.itu.dk/roba/Master-thesis`

6.1.3 Metrics

To evaluate the performance of the device on the microbenchmarks as well as on the benchmark running inference using full-size deep neural networks, we use the following metrics:

Latency Reported in milliseconds (ms), this metric represents the time-to-inference. This metric is used in cases where batching is not utilized.

Throughput Reported in samples per second, this metric represents the total number of samples inferred in a time unit. Throughput is used to quantify the results of the experiments utilizing batching.

The latency of the microbenchmarks without batching was collected on a per-operation basis (referred to as *operation* latency) through the on-board performance counters. In addition to the per-operation latency, we present the *total* latency of the inference of a sample, including the transfer times to and from the device. This latency was measured on the host using *chrono*.

The throughput of the microbenchmarks with batching was derived from the time measured using the *chrono* utility, which includes the transfer times to and from the device, as was the case for latency.

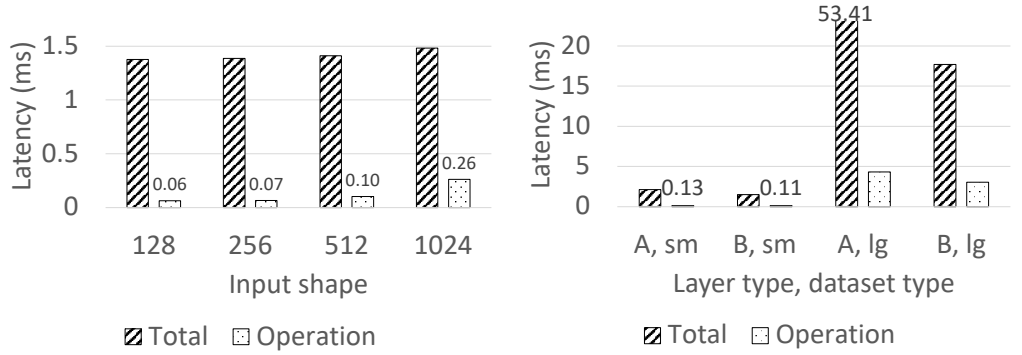
For the benchmark using the full-size models, we report throughput, measured the same way as in the case of the microbenchmarks.

In addition to the per-operation latency, the device also reports the time spent waiting for input. These results were, however, omitted as they were found to be unreliable.

6.2 Results

This section first presents the results of the performance analysis of fully-connected and convolutional layers running on the Neural Compute Engine. Then we present the results of the analysis of the custom OpenCL implementation of the fully-connected layer, after which we analyze the batched versions of this layer on both the Neural Compute Engine and the SHAVE cores. These results are then further analyzed to identify the overheads in the parallelism of the SHAVE cores. Finally, we present the results of inference using full-size deep neural networks to see how the findings gained through the microbenchmarks transfer to real-world usage.

6.2.1 Performance of the Neural Compute Engine



(a) Fully-connected layer

(b) Convolutional layer

Figure 9: The total and per-operation latency of (a) fully-connected layer with varying input sizes and (b) convolutional layers A and B on small (sm) and large (lg) datasets.

Figure 9a shows the total inference latency and the latency of the fully-connected layer operation, using an input of varying sizes, running on the Neural Compute Engine accelerator, as implemented in OpenVINO. The latency of the fully-connected layer operation does not scale linearly in the input shape but rather in the number of MAC operations, which can be derived for a fully-connected layer as follows:

$$\#MAC = M \cdot K \cdot N = K^2 \tag{3}$$

, where M is the batch size (in this case, 1), K is the common dimension between the input and the weight matrices, and N is the new output shape, which in the case of a square weight matrix equals the K dimension.

In addition to the latency contributed by the input shape, there is a significant overhead in latency, amounting to 0.05 ms, which is close to the whole operation latency of the fully-connected layer with an input shape of 128.

While the total latency also scales linearly with the number of MAC operations, it is mostly composed of a significant overhead, amounting to 1.38 ms, representing most of the execution time. Most of this overhead is the transfer time of both the weight matrix and the input vector, which

would get amortized after the execution of multiple layers, or by use of data transfer pipelining.

The fig. 9b shows the inference latency using a single convolutional layer with varying configurations.

We can see significant differences caused by the increase in the input size. Although the layers with small and large datasets use the same weight matrix, the difference between the latencies of the layers using differing input shapes is significant.

The number of elements in a weight tensor of a convolutional layer can be derived as:

$$\#W = K_x \cdot K_y \cdot C_{in} \cdot C_{out} \tag{4}$$

, where K_x and K_y are the width and the height of the convolutional kernel and the C_{in} and C_{out} the number of input and output channels, respectively.

Mainly in the case of layer B, the size of the weight tensor (147,456 elements) is not insignificant in comparison to the input sizes (4,096 elements in the case of a small dataset and 200,704 elements in the case of a large dataset). In the case of the small dataset, the transfer time of both the weight tensor and the input tensors amounts to a maximum of 1.38 ms, largely dominated by the transfer of the weight tensor. The weight tensor represents 42% of the total transfer size for the larger dataset, and the data transfer should, therefore, only take 3.29 ms, presenting a discrepancy.

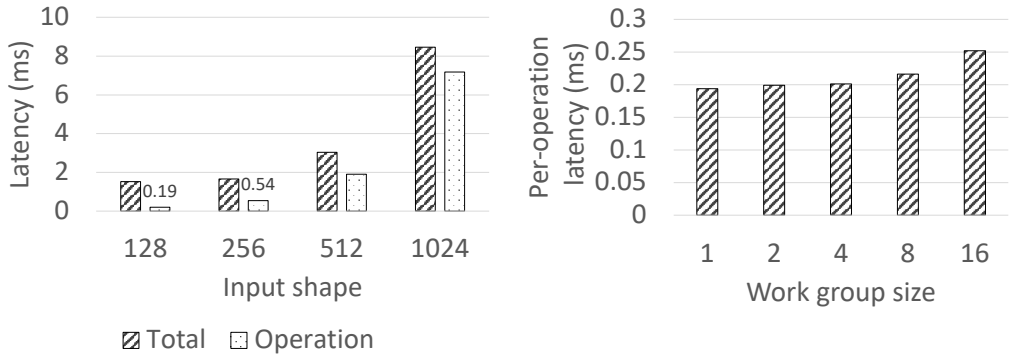
This effect can be attributed to the image-to-column operation, commonly used in a two-step convolution implementation, where the first step includes the image-to-column operation, followed by matrix multiplication. This operation unrolls the sliding window of the convolution over time, thereby significantly increasing the memory requirements.

6.2.2 Performance of the Custom Implementation of the Fully-connected Layer

Section 4 presents how the fully-connected layer is implemented and possible optimizations of this operation commonly used when programming massively parallel hardware such as GPUs. This section presents the results of a custom implementation of the fully-connected layer running on the SHAVE cores, which leverages these optimizations.

Naive implementation Figure 10a presents the total inference latency and the latency of the fully-connected layer kernel, processing a single sample.

This is a naïve implementation that does not leverage shared memory within work groups but relies on data accesses into the global DRAM memory. This implementation, however, does perform coalesced access.



(a) Naïve implementation (b) Effect of work group size

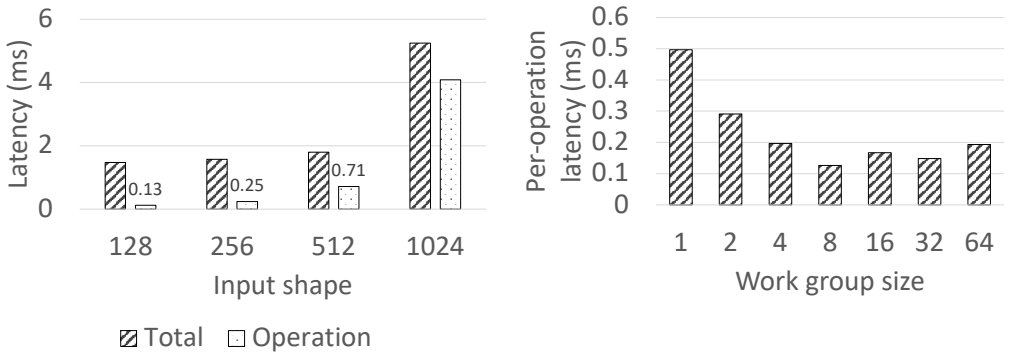
Figure 10: Analysis of the performance of the naïve implementation of the fully-connected layer using (a) work group size of 1 and (b) input shape of 128 with varying work group sizes.

This implementation performs much worse than the default implementation on the Neural Compute Engine. While the inference using the smallest size is close to the latency of the inference using the Neural Compute Engine, this implementation has unfavorable scaling. As the input size increases, the total inference latency becomes dominated by the latency of the fully-connected layer operation. As was the case in the implementation running on the Neural Compute Engine, this implementation also scales linearly with the number of MAC operations, with the total latency including an overhead of similar size.

Figure 10b shows the effect of the work group sizes on the performance. As the naïve implementation does not use the collaboration between work items within a work group, the latency does not improve with the increase in the size of the work groups. It rather has a negative effect, as the work items within the same work group are executed sequentially on a single SHAVE core, as mentioned in section 2.2.2.

Tiling This implementation uses tiling, which in the case of the matrix-vector multiplication, increases the computational intensity for the input

vector through data reuse between work items within a work group while not changing it for the weight matrix due to the operation being parallelized only in a single dimension.



(a) Tiled implementation (b) Effect of work group size

Figure 11: Analysis of the performance of the tiled implementation of the fully-connected layer using (a) work group size of 8 and (b) input shape of 128 with varying work group sizes.

Figure 11a shows the improvement in latency this optimization leads to. Specifically, this figure presents the results of the tiled implementation using the tile size of 8. The latency of the fully-connected layer operation drops by $35 - 62\%$ compared to the naïve implementation.

Figure 11b shows the effect of increasing the size of the work groups or the tiles (same in this implementation). Since this implementation leverages the collaboration between the work items within a work group, the size of the work groups has a large effect on the performance of the kernel. The performance increases with the increase in tile size until reaching tile size of 8. Past this point, the performance fluctuates. The performance peaking at a tile size of 8 might be attributed to the automatic vectorization the compiler applies when coalesced access is detected within a work group. The MAC operations can be processed in vectors containing 8 half-precision floating-point operators, leading to parallel execution of the work group.

Vectorized implementation As mentioned, the SHAVE cores support vector instructions. Figure 12 show results of custom implementation using manual vectorization.

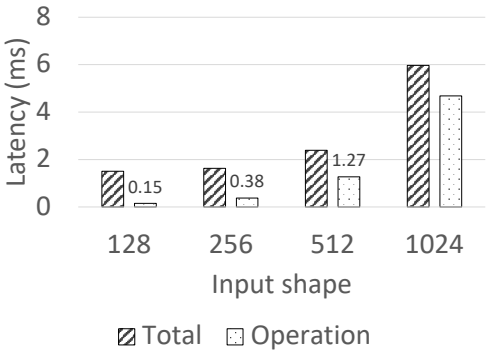


Figure 12: Total and per-operation latency of the vectorized implementation of the fully-connected layer running on SHAVE cores.

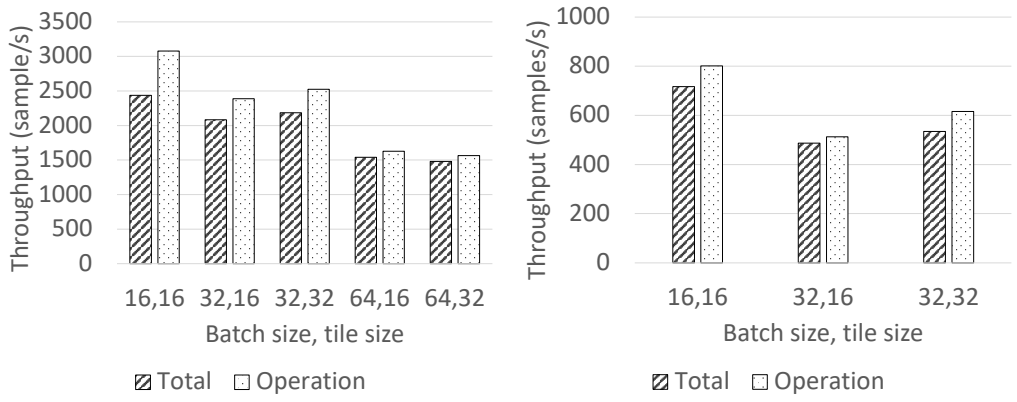
We could not improve on the tiled implementation and the automatic vectorization applied by the compiler. We can see a slight performance decrease compared to the tiled version.

6.2.3 Batching

This section explores batching, a technique used to increase the throughput of machine learning training and inference. We first build upon the custom implementation running SHAVE cores presented earlier and compare it to the performance of the non-batched versions. Then we explore the effect of batching on the performance of the operations running on the Neural Compute Engine.

Fully-connected layer using Shave cores As we alluded to before, the matrix-matrix multiplication (batched) implementation of a fully-connected layer brings multiple theoretical advantages over the non-batched matrix-vector multiplication. This is mainly the possibility of using the second dimension for parallelizing the operation and therefore gaining on data reuse not only in the case of the input data but also in the weight matrix.

Figure 13 shows the throughput of the batched fully-connected layer, implemented as a matrix-matrix multiplication, leveraging most of the optimizations we mentioned in section 4, including tiling in two dimensions (tile sizes of 16x16 or 32x32), increase in the amount of work per work item (8 elements per work item), coalesced access for all accesses to global DRAM



(a) Input shape = 128

(b) Input shape = 256

Figure 13: Analysis of batched custom implementation of the fully-connected layer with input shape of (a) 128 and (b) 256, running on the SHAVE cores.

memory and register tiling.

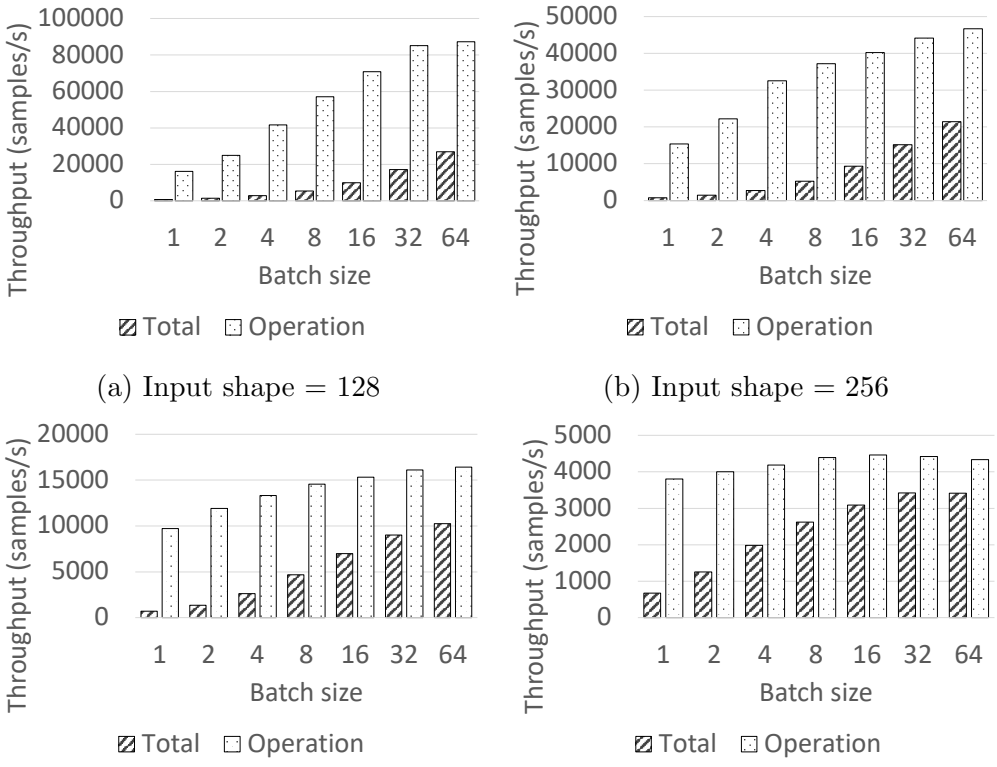
These results present the cases with image shapes up to 256 and batch sizes of 16, 32, and 64, which must be higher than the tile size. Although smaller tile sizes were tested, these need to be of the same size or larger than the number of elements processed by a work item. This number of elements processed by a work item was set to 8 to decrease the total number of work items spawned, as without it, the NCS2 crashes and stops responding.

In comparison to the fastest tiled implementation presented in the last section, this batched implementation is at least 61% slower in the case of the input shape of 128 and 82% slower in the case of the input shape of 256.

This slowdown and the costs associated with the increase in parallelism on the SHAVE cores will be explored in detail in section 6.2.4.

Fully-connected layer using Neural Compute Engine Section 6.2.1 pointed to the fact that the large portion of the operation latency of the fully-connected layer consists of a constant overhead. Here we explore whether we can take advantage of this fact and pay this constant overhead only once for larger batch sizes.

Figure 14 shows the total inference throughput and the operation throughput of the fully-connected layer with varying batch and input sizes. The overall trend points to the layers with smaller input sizes taking better ad-



(a) Input shape = 128

(b) Input shape = 256

(c) Input shape = 512

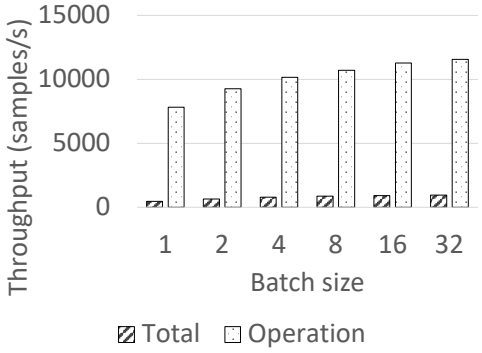
(d) Input shape = 1024

Figure 14: The total and per-operation latency of fully-connected layer with input shape of (a) 128, (b) 256, (c) 512 and (d) 1024; running on the Neural Compute Engine.

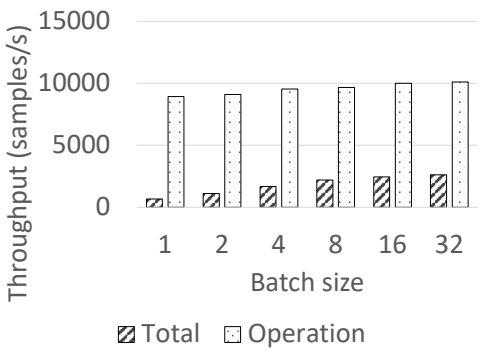
vantage of the batching. The layer with the input shape of 128 leads to a significant increase in throughput up to the batch size of 32, after which the performance starts tapering off. This effect of the performance tapering off is seen earlier as the input size increases. In the case of the input shape of 1024, we see negligible performance gains from increasing the batch size.

A similar positive effect can also be seen in the total inference throughput. The total inference latency of the non-batched version was shown to be composed of a large overhead, which we also take advantage of here. The total inference throughput shows even better scalability than the operation latency, the effects of which can be seen even in the case of the highest input shape tested.

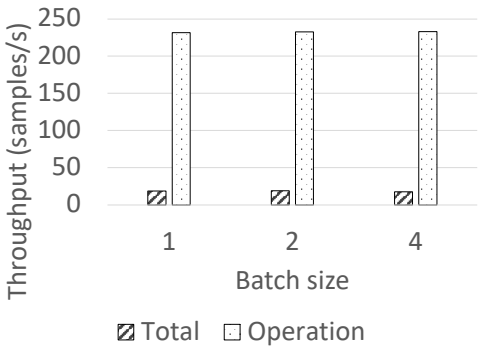
Convolutional layer using Neural Compute Engine Figure 15 show the effect of batching on the performance of the convolutional layers running on the Neural Compute Engine. The figures show that while batching does not have a positive nor negative effect on the throughput using the larger dataset (figures 15c and 15d), a positive effect can be seen for the smaller dataset (figures 15a and 15b).



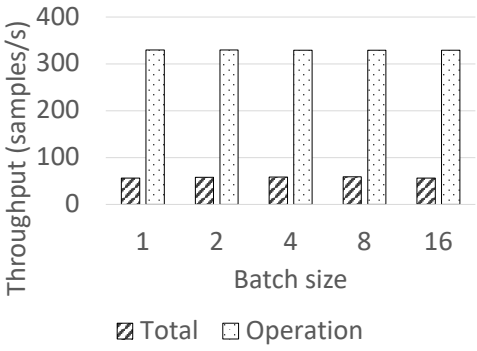
(a) Layer A, small dataset



(b) Layer B, small dataset



(c) Layer A, large dataset



(d) Layer B, large dataset

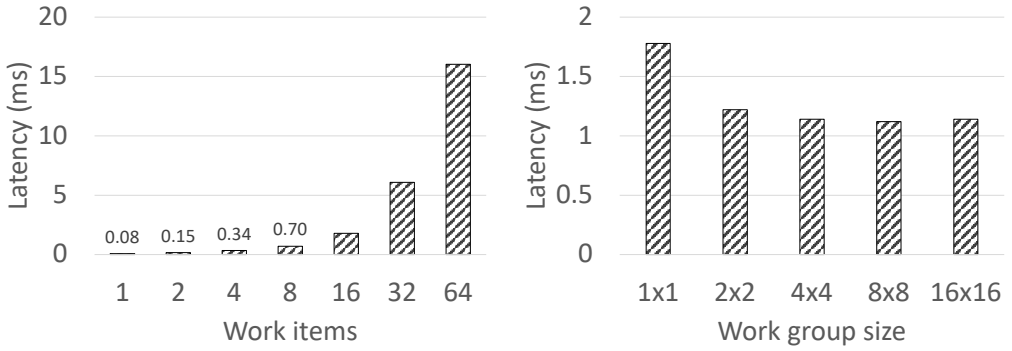
Figure 15: The total and per-operation latency of convolutional layer in multiple configurations, representing (a) layer A using small dataset, (b) layer B using small dataset, (c) layer A using large dataset and (d) layer B using large dataset; running on the Neural Compute Engine.

The largest effect on the operation throughput can be seen in layer A, which increases by up to 48%. The total throughput increases significantly for both layers using the small dataset, where the throughput was up to 2x

as high as the non-batched version in the case of layer A and up to 4x as high in the case of layer B.

6.2.4 Overhead of parallelism in SHAVE cores

As mentioned in section 2.2.2, there is a total of 16 SHAVE cores on the device, which each map to a single work group, executing these work groups in parallel if enough cores available. Furthermore, the work items within these work groups are executed sequentially on the SHAVE core assigned to the work group.



(a) (b)

Figure 16: (a) Shows the overhead of spinning multiple work groups with one work item per group (executed on separate cores). The number of work items is displayed for a single dimension. The second dimension is fixed to 128 work items. (b) Shows the effect of number of work items per work group with a set number of total work items (2048).

To test the overhead attached with executing multiple work groups/items in parallel, we replaced the kernel for fully-connected layer with the one that does not contain any instructions, but mimics this operation in the size of work groups and the number of total work items.

Figure 16a shows the latency for executing multiple work groups of size 1. This size of the work groups means that every work item is executed on a separate SHAVE core. The figure shows the number of work items in the first dimension, while the number of work items in the second dimension was fixed to 128, emulating an increasing batch size with vectors of size 128.

We can see that the increase in parallelism comes at a high cost. For the configuration emulating the batch size of 16, the overhead of the parallel execution is almost 8x higher than the operation latency of executing fully-connected layer on batch size of 16 using the Neural Compute Engine.

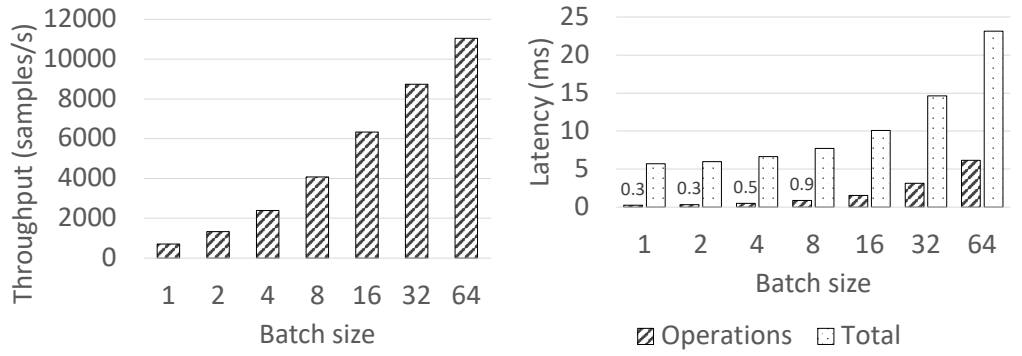
A reduction in parallelism on the device is possible through increase in work group size, which scales the number of total work groups down by their size. Figure 16b shows the effect of this for the case emulating the batch size of 16. We can see that the overhead associated with the parallelism decreases until the work group sizes of 8x8. This size reduces the number of groups in the larger dimension down to 16, which is the total number of SHAVE cores, allowing the whole dimension to be executed in parallel. Even though this reduces the number of work groups significantly, it is not able to close the gap between the overhead of the SHAVE cores and the execution time on the Neural Compute Engine.

6.2.5 Performance on Full-size Models

This section first shows the impact of batching and data transfer pipelining on inference using a feed-forward neural network, which builds on the microbenchmarks using the fully-connected layers. Then we present results for inference using the ResNet-18 architecture, a convolutional neural network comprised of both the convolutional and fully-connected layers.

Feed-forward neural network Figure 17a shows the throughput of the feed-forward neural network with increasing batch size. This figure largely follows the results for the batched inference using a single fully-connected layer presented in section 6.2.3. Since this neural network is comprised of multiple fully-connected layers and ReLU activation layers, which get fused with the preceding fully-connected layers, the effects presented earlier accumulate.

The fig. 17b shows the inference latency for the entire batch. Specifically, it shows the latency of the execution of all of the operations of the neural network and the total inference time, including the data transfers and the time data sits in the queue on the device. This figure shows that, especially for the smaller batch sizes, the execution time of all operations is significantly lower than that of the total inference latency. This means that even with multiple pipelined inference requests, we cannot feed the device with data



(a) (b)

Figure 17: (a) Shows the effect increasing the batch size has on the throughput of the feed-forward neural network. (b) Shows the operation latency for a single batch and the average total inference time.

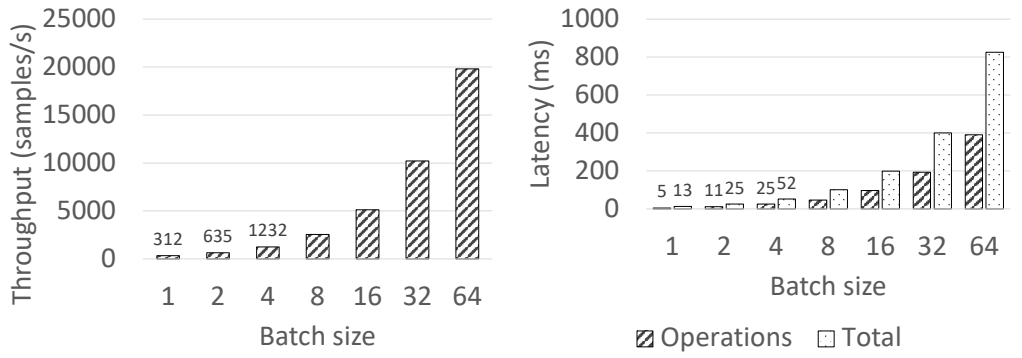
fast enough. This results in most of the latency being spent waiting for input until reaching higher batch sizes.

Resnet-18 Unlike the previous example, this does not follow the results presented earlier, as the convolutional layers have significantly different characteristics from the ones tested in section 6.2.3 and vary in kernel and stride sizes efecting their computational and memory requirements. As shown in fig. 18a, the throughput of the ResNet-18 model scales very linearly.

Figure 18b shows that unlike in the case of the feed-forward neural network, the execution time of the ResNet-18 model reaches half of the total inference latency of a batch. This, however, does not mean that the other half of the inference latency can be attributed to an inability to feed the device with data fast enough, but rather the contrary. This is the time the inference request sits in a queue on the device, waiting to be processed.

6.3 Discussion

SHAVE cores The results presented in this section show that the performance delivered by the Neural Compute Engine hardware accelerator is superior to the performance of the SHAVE cores. Using the SHAVE cores comes at the cost of a high overhead attached to parallelism. These units,



(a) (b)

Figure 18: (a) Shows the effect increasing the batch size has on the throughput of the Resnet-18 neural network. (b) Shows the operation latency for a single batch and the average total inference time.

however, provide far better extensibility and flexibility, which can accommodate the rapid pace of development in new neural network architectures and the layers that build them up.

While the use of the fully-connected layer on the SHAVE cores is not representative of real-life usage, it provided many opportunities to show the impact of the different optimizations and had a firm baseline for performance in the form of the fully-connected layer implemented on the Neural Compute Engine.

We showed that not all layers are suitable for deployment on this device. There is a need to identify the characteristics that make up a good candidate for acceleration by this device.

Furthermore, even though the Neural Compute Engine does not allow for operations other than with 16-bit floating point precision, the SHAVE cores contain integer register files and arithmetic units, which, together with the vector register files and arithmetic units, could be used to optimize the inference using novel layers by leveraging mixed-precision model, with the SHAVE-core executing the reduced precision operations.

Neural Compute Engine This accelerator's high degree of specialization delivers significant performance improvements at the cost of reduced flexibility. While the microbenchmarks leveraging the convolutional layers modeled

after the VGG-11 architecture did not significantly improve with increasing batch size, the ResNet-18 architecture did. We attribute this effect to the differences in the shapes of the kernels and the stride sizes, as these impact the memory requirements and latency of convolutional layers performed using the image-to-column operation. This topic requires further research to identify the best characteristics of these layers for running on the Neural Compute Engine accelerator.

The benchmarks using the full-size models show promising performance with the increasing batch size and leveraging the asynchronous execution and data transfer pipelining. Mainly in the case of the feed-forward neural network, we can see the potential for further optimization. The device is underutilized as the host is not able to saturate the device with input data. An increase in the number of concurrent inference requests could potentially help mitigate this issue. Since the number of inference requests can be changed at the runtime, the time spent waiting for the input could be monitored, and the number automatically adjusted. The limitations of the concurrent inference requests have to be explored further, however.

Furthermore, these benchmarks show performance on small datasets like CIFAR-10 and MNIST. This size was chosen based on the performance and scalability with respect to the batch size presented in the microbenchmarks. Change to a larger dataset, such as the ImageNet, could break the scalability. This, however, requires further experimentation.

7 Conclusion

This paper presented our work on the characterization of the performance and possible optimizations of the Intel Neural Compute Stick 2. We first described the device and the OpenVINO toolkit used for optimization and running of inference using this device and characterized the process of extending the device's functionality through custom OpenCL kernels running on the SHAVE cores of the device. Furthermore, through the use of application-specific benchmarks, we characterized the latency and power characteristics in comparison to other devices used to accelerate edge applications. We found that the device provides a superior ratio in latency and power draw in comparison to less specialized devices, such as an ARM Cortex-M7 microcontroller or the NVIDIA Jetson Nano. On the other hand, the devices leveraging the newer architecture found in CoralAI edge TPUs, provide sim-

ilar latency at lower power draw and power consumption.

After the benchmark against the other devices, we proceeded to test the device in isolation to analyze potential optimizations we can leverage in order to increase the efficiency of the device. First, we set the baseline of the device's performance on fully-connected and convolutional layers using the Neural Compute Engine. Then we presented the performance of the custom OpenCL kernels for fully-connected layer running on the programmable SHAVE cores, and the optimization we leveraged, and their effect on the performance. We found that parallelism on this accelerator is attached to large overheads. The performance of these custom kernels cannot match the performance of the implementation on the Compute Neural Engine, and the gap between the performance widens with the increase in parallelism caused by larger matrix sizes. We found that this device should only be used in the cases in which the operations needed do not come with OpenVINO and are mission-critical.

Furthermore, we identified that the underutilized option for batching on the Neural Compute Engine can deliver significant improvements in performance by amortizing the overheads of initialization of the operations and the data transfers.

Finally, we quantified the effects of the batching and asynchronous execution using multiple concurrent inference requests on inference using full-size neural networks. We found that the positive effects of batching accumulate within the larger models and lead to higher performance gains than when executed in isolation.

References

[1] OpenVINO toolkit. [Online]. Available: https://docs.openvino.ai/2022.3/home.html

[2] Intel Movidius Myriad X Vision Processing Unit (VPU). [Online]. Available: https://www.intel.com.au/content/www/au/en/products/docs/processors/movidius-vpu/myriad-x-product-brief.html

[3] Intel Movidius Myriad X VPU. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/processors/movidius-vpu/myriad-x-product-brief.html

[4] AI Benchmarks: A Barometer Before You Build. [Online]. Available: https://www.insight.tech/content/ai-benchmarks-a-barometer-before-you-build

[5] SHAVE v2.0. [Online]. Available: https://en.wikichip.org/wiki/movidius/microarchitectures/shave_v2.0

[6] G. Plastiras, M. Terzi, C. Kyrkou, and T. Theocharidcs, "Edge Intelligence: Challenges and Opportunities of Near-Sensor Machine Learning Applications," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–7.

[7] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "When Edge Meets Learning: Adaptive Control for Resource-Constrained Distributed Machine Learning," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 63–71.

[8] M. Mohammadi Amiri and D. Gündüz, "Machine Learning at the Wireless Edge: Distributed Stochastic Gradient Descent Over-the-Air," *IEEE Transactions on Signal Processing*, vol. 68, pp. 2155–2169, 2020.

[9] M. Merenda, C. Porcaro, and D. Iero, "Edge Machine Learning for AI-Enabled IoT Devices: A Review," *Sensors*, vol. 20, no. 9, 2020.

[10] M. Modasshir, A. Quattrini Li, and I. Rekleitis, "Deep Neural Networks: A Comparison on Different Computing Platforms," in *2018 15th Conference on Computer and Robot Vision (CRV)*, 2018, pp. 383–389.

[11] A. E. Tolmacheva, D. A. Ogurtsov, and M. G. Dorrer, "Justification for choosing a single-board hardware computing platform for a neural network performing image processing," *IOP Conference Series: Materials Science and Engineering*, vol. 734, no. 1, 2020.

[12] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, and H. Kim, "Characterizing the Deployment of Deep Neural Networks on Commercial Edge Devices," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019, pp. 35–48.

[13] X. Zhang, H. Ye, and D. Chen, "Being-ahead: Benchmarking and exploring accelerators for hardware-efficient AI deployment," *CoRR*, vol. abs/2104.02251, 2021.

[14] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. A. Patterson, D. Pau, J. Seo, J. Sieracki, U. Thakker, M. Verhelst, and P. Yadav, "Benchmarking tinyml systems: Challenges and direction," *CoRR*, vol. abs/2003.04821, 2020.

[15] O. Aleksandrova and Y. Bashkov, "Face recognition systems based on Neural Compute Stick 2, CPU, GPU comparison," in *2020 IEEE 2nd International Conference on Advanced Trends in Information Theory (ATIT)*, 2020, pp. 104–107.

[16] M. Esposito, S. S. Conticello, M. Pastena, and B. C. Domínguez, "In-orbit demonstration of artificial intelligence applied to hyperspectral and thermal sensing from space," in *CubeSats and SmallSats for Remote Sensing III*, vol. 11131, 2019, p. 111310C.

[17] G. Giuffrida, L. Diana, F. Gioia, G. Benelli, G. Meoni, M. Donati, and L. Fanucci, "CloudScout: A Deep Neural Network for On-Board Cloud Detection on Hyperspectral Images," *Remote Sensing*, vol. 12, p. 2205, 07 2020.

[18] G. Dinelli, G. Meoni, E. Rapuano, G. Benelli, and L. Fanucci, "An FPGA-Based Hardware Accelerator for CNNs Using On-Chip Memories Only: Design and Benchmarking with Intel Movidius Neural Compute Stick," in *International Journal of Reconfigurable Computing*, 2019.

[19] P. Minaidis, "Embedded Development of AI-based Computer Vision: Acceleration on Intel Myriad X VPU," Ph.D. dissertation, 2021.

[20] A. Misra, "Deep Learning Acceleration on the Edge," Ph.D. dissertation, 2019.

[21] Q. Li, J. Song, J. Ning, and J. Yuan, "The Detailed Data on the Neural Compute Stick Acceleration Performance," in *2019 Chinese Automation Congress (CAC)*, 2019, pp. 4959–4962.

[22] S. Rivas-Gomez, A. J. Pena, D. Moloney, E. Laure, and S. Markidis, "Exploring the Vision Processing Unit as Co-Processor for Inference," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 589–598.

[23] M. F. A. Hamid and F. H. K. Zaman, "Hand Gesture Recognition Using Movidius Neural Compute Stick," in *2019 IEEE 9th International Conference on System Engineering and Technology (ICSET)*, 2019, pp. 510–514.

[24] S. P. Kaarmukilan, A. Hazarika, A. Thomas K., S. Poddar, and H. Rahaman, "An Accelerated Prototype with Movidius Neural Compute Stick for Real-Time Object Detection," in *2020 International Symposium on Devices, Circuits and Systems (ISDCS)*, 2020, pp. 1–5.

[25] NVIDIA, *CUDA C++ Best Practices Guide*, 2023.

[26] STM32 Nucleo-144 development board. [Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-h745zi-q.html

[27] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, P. Warden, and R. Rhodes, "TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems," in *MLSys*, vol. 3, 2021, pp. 800–811.

[28] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," *CoRR*, vol. abs/1801.06601, 2018.

[29] (2022) AI expansion pack for STM32CubeMX. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html

[30] NVIDIA. (2019) Jetson Nano Developer Kit. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[31] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *OSDI*, 2016, pp. 265–283.

[32] NVIDIA. (2023) TensorRT. [Online]. Available: https://developer.nvidia.com/tensorrt

[33] Raspberry Pi. (2016) Raspberry Pi 3 Model B. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b/

[34] CoralAI. (2020) CoralAI TPU Technology. [Online]. Available: https://coral.ai/technology/

[35] ——. (2022) CoralAI Dev Board Mini. [Online]. Available: https://coral.ai/products/dev-board-micro/

[36] FreeRTOS. (2023) Freertos. [Online]. Available: https://www.freertos.org/index.html

[37] CoralAI. (2020) CoralAI Dev Board Mini. [Online]. Available: https://coral.ai/products/dev-board-mini/

[38] ——. (2023) Mendel Linux. [Online]. Available: https://coral.googlesource.com/docs/+/refs/heads/master/ReadMe.md

[39] ——. (2019) CoralAI Dev Board Mini. [Online]. Available: https://coral.ai/products/accelerator/

[40] ——. (2022) Edge TPU Compiler. [Online]. Available: https://coral.ai/docs/edgetpu/compiler/

[41] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *CoRR*, 2017.

[42] The TensorFlow Team. (2019) Flowers. [Online]. Available: http://download.tensorflow.org/example_images/flower_photos.tgz

[43] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[44] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[45] Qoitech. (2022) Otii ace pro. [Online]. Available: https://www.qoitech.com/otii-ace/

[46] System76. (2022) Pop!_OS. [Online]. Available: https://pop.system76.com

[47] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations*, 2015.

[48] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[49] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.