SPARK: An Efficient Hybrid Acceleration Architecture with Run-time Sparsity-Aware Scheduling for TinyML Learning

Mingxuan Li[1], Qinzhe Zhi[1], Yanchi Dong[1], Le Ye[1,2], Tianyu Jia[1,*]
[1] School of Integrated Circuits, Peking University, Beijing, China
[2] Advanced Institute of Information Technology of Peking University, Hangzhou, China
*Corresponding Author: tianyuj@pku.edu.cn

ABSTRACT

Currently most TinyML devices only focus on inference, as training requires much more hardware resources. In this paper, we introduce *SPARK*, an efficient hybrid acceleration architecture with run-time sparsity-aware scheduling for TinyML learning. Besides a standalone accelerator, an in-pipeline acceleration unit is integrated within the CPU pipeline to support simultaneous forward and backward propagation. To better utilize sparsity and improve hardware utilization, a sparsity-aware acceleration scheduler is implemented to schedule the workload between two acceleration units. A unified memory system is also constructed to support transposable data fetch, reducing memory access. We implement *SPARK* using TSMC 22nm technology and evaluate different TinyML tasks. Compared with the baseline accelerator, *SPARK* achieves 4.1× performance improvement in average with only 2.27% area overhead. *SPARK* also outperforms off-shelf edge devices in performance by 9.4× with 446.0× higher efficiency.

KEYWORDS

On-device learning, Sparsity-aware scheduling, In-pipeline acceleration, TinyML

1 INTRODUCTION

Deep neural networks (DNNs) have displayed remarkable capabilities across a wide range of domains with increasing parameters and computations. Resource-constrained edge devices, such as IoTs, are also required to be equipped with DNN-processing capability, forming Tiny Machine Learning (TinyML) devices [1–3]. Such TinyML solutions are able to collect private data and execute DNNs locally, avoiding time and energy-consuming data transmission and achieving efficient low-latency computing.

Prior solutions mostly focus on the TinyML inference at edge. To deploy inference on edge devices and reconcile the gap between growing model size with resource limitations, significant efforts have been explored across software and hardware stacks. For example, model sparsity [1, 4, 5] and neural architecture search (NAS) [6] are widely exploited to compress the DNN models to meet the device memory constraint. To realize high computation efficiency, many ASIC inference accelerators [3, 7] have been developed to improve the edge computing capability and efficiency.

Beyond inference, on-device TinyML training has shown its unique advantages and growing importance, pushing the chip architects to consider on-device training feature seriously. Compared to inference, on-device training offers the possibility to customize DNN models based on user habits without cloud help, protecting private data. By gathering private data and fine-tuning pre-trained models at edge, the prediction accuracy of personalized model can be continuously enhanced for specific user preferences.

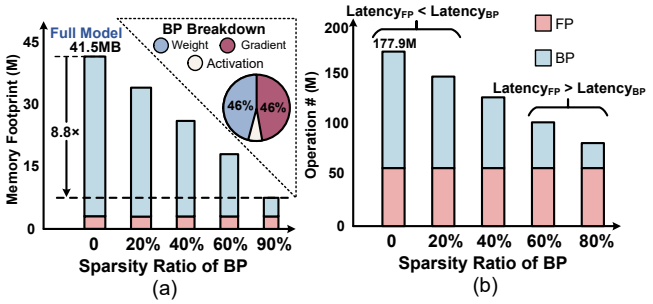


Figure 1: (a) Memory footprint of MobilenetV2 training and the benefits of sparse training. (b) Operation counts of MobilenetV2 with varying sparsity.

Although on-device training holds great appeal, the issue of limited hardware resources becomes more pronounced when it comes to training. As shown in Fig. 1(a) training process involves forward propagation (FP) and backward propagation (BP). BP involves weight and activation gradient computation, and hence extra memory is required for gradients and intermediate activation, taking up about 54% memory cost in BP. Even for training a lightweight model MobileNetV2-w0.35 [8], it still requires over 40MB of memory for the full model training. Moreover, transposed weight matrices are involved in BP, leading to irregular memory access patterns.

To alleviate the tight memory challenge, various learning techniques, such as transfer learning [9] and federated learning [10], facilitate on-device training by eliminating the need to train a model from scratch. However, most weights in transfer learning is frozen, leading to low accuracy. Thus, leveraging sparse training is another key technique to decrease memory cost in BP by skipping bias and weight update with negligible accuracy degradation. As shown in Fig. 1, up to 87.5% weight sparsity during training exists among model layers. Such sparsity pattern can reduce computation and lead to 8.8× memory footprint reduction. To better utilize the constrained hardware resource, several specialized accelerators have been developed for sparse training. For example, [11] exploits sparsity with intra-channel and inter-channel accumulation and [12] uses a speculation module to predict the unknown sparsity to speed up training. Another edge training accelerator [13] reduces memory access by reordering data inputs and simplifies computation by transforming the calculation format. Besides, in-pipeline acceleration, i.e. developing custom acceleration within CPU pipeline

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3657369

[14, 15], has also drawn great interest as it can achieve promising performance improvement with limited hardware overhead, which is attractive for resource-constrained TinyML devices.

However, the solutions mentioned above raise new challenges for on-device sparse training. As shown in Fig. 1(b), when the sparsity ratio varies significantly, the latency also exhibits a wide range of variation, leading to unpredictable and unstable latency in practical applications. Besides, prior on-device sparse training accelerators execute FP and BP in a sequential manner, i.e. executing BP after FP. Such sequential execution exhibits relatively high latency due to the lack of parallelism. Even if one more accelerator is dedicated to enable parallel FP and BP execution, the wide range of sparsity lying in FP and BP will be likely to result in the under-utilization of hardware accelerators.

In this paper, we introduce *SPARK*, an efficient hybrid acceleration architecture with run-time sparsity-aware scheduling for on-device TinyML learning. Compared to a stand-alone conventional dedicated accelerator, we speedup the sparse training with a hybrid acceleration architecture, which consists of a low-cost in-pipeline acceleration unit and a dedicated array-based acceleration engine. Besides two acceleration units, a run-time sparsity-aware scheduler to dynamically schedule FP and BP computation across acceleration units and a unified transposable-fetch memory system to reduce memory access are included in *SPARK*. *SPARK* is synthesized and implemented using TSMC 22nm process. Compared to the baseline accelerator and the off-shelf edge devices, our architecture achieves 4.1× and 9.4× improvement in average latency with negligible hardware overhead. The contribution of this work is summarized as following:

- An efficient heterogeneous acceleration architecture which supports simultaneous forward and backward propagation.
- A novel sparsity-aware scheduler which improves the utilization of hardware resources and reduces average latency.
- A unified memory system for on-device training which can handle transposed matrices under low memory access cost.
- A layout synthesized and implemented using TSMC 22nm process to evaluate different TinyML training tasks.

2 BACKGROUND

Sparsity in Forward Propagation: Sparsity is one of the most effective approaches to reduce DNN model size. In FP, sparsity mainly includes pruning weights. Weight pruning removes less important weights from the network based on a certain criteria, such as the weight magnitude, thus reducing computation. For a DNN model of l layers, each layer can be represented by a weight matrix $\mathbf{W}_i \in \mathbb{R}^{c_{i1} \times c_{i2}}$ and a bias vector $\mathbf{b}_i \in \mathbb{R}^{c_{i2}} (1 \le i \le l)$. An FP sparsity can be represented by l mask matrices $\mathbf{m}_{w,i} \in \{0,1\}^{c_{i1} \times c_{i2}}$ and vectors $\mathbf{m}_{b,i} \in \{0,1\}^{c_{i2}}$ and the sparse model layer should be $\widetilde{\mathbf{W}}_i = \mathbf{W}_i \odot \mathbf{m}_{w,i}$, $\widetilde{\mathbf{b}}_i = \mathbf{b}_i \cdot \mathbf{m}_{b,i}$, where $\odot$ denotes the entry-wise (Hadamard) product. As a result, some weights in the weight matrix $\mathbf{W}_i$ are pruned, i.e. become zeros in the updated weight matrix $\widetilde{\mathbf{W}}_i$. The sparsity ratio of layer i is defined as

$$\mathbf{r}_i = \frac{Num\left(zeros\ in\ \widetilde{\mathbf{W}}_i\ \&\ \widetilde{\mathbf{b}}_i\right)}{Num\left(all\ params.\ in\ \widetilde{\mathbf{W}}_i\ \&\ \widetilde{\mathbf{b}}_i\right)} \tag{1}$$

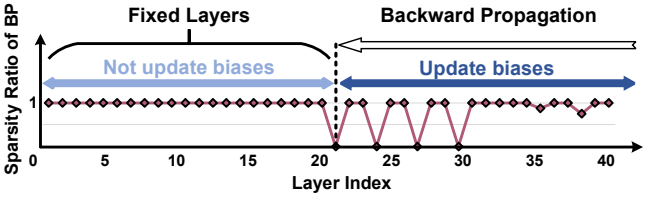


Figure 2: Sparsity scheme of MCUNet [1].

Sparsity in Backward Propagation: In BP, weight gradients are computed according to a certain loss function. By applying the derivative chain rule, the weight gradients of former layers are obtained based on the intermediate activation gradients, as is shown in formula (2).

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}_i} = \mathbf{A}_i^T \cdot \frac{\partial \mathbf{L}}{\partial \mathbf{A}_{i+1}}, \quad \frac{\partial \mathbf{L}}{\partial \mathbf{b}_i} = \frac{\partial \mathbf{L}}{\partial \mathbf{A}_{i+1}}, \quad \frac{\partial \mathbf{L}}{\partial \mathbf{A}_i} = \frac{\partial \mathbf{L}}{\partial \mathbf{A}_{i+1}} \cdot \mathbf{W}_i^T \tag{2}$$

$\mathbf{L}$ denotes loss and $\mathbf{A}_i$ denotes the i^{th} layer activation. Sparsity in BP mainly involves skipping bias, weight updates, i.e. the target biases and weights are fixed. Sparsity pattern may vary at three different granularity: (1) *bias only*, only update biases; (2) *partial weight*, update a subset of weights; (3) *full weight*, update all weights.

The sparsity ratio of layer i in BP is defined similarly to FP:

$$\mathbf{r}_i = \frac{Num\left(fixed\ params.\ in\ \widetilde{\mathbf{W}}_i\ \&\ \widetilde{\mathbf{b}}_i\right)}{Num\left(all\ params.\ in\ \widetilde{\mathbf{W}}_i\ \&\ \widetilde{\mathbf{b}}_i\right)} \tag{3}$$

Sparsity Scheme Exploration: To find a reasonable sparsity scheme with relatively low search complexity, the exploration is carried out under two basic strategies: (1) bias: only updating last k biases $\mathbf{b}_l, \mathbf{b}_{l-1}, ..., \mathbf{b}_{l-k+1}$. (2) weights: updating the weights of layer $\mathbf{i}$ with the sparsity ratio of $\mathbf{r}_i$. To further facilitate performance evaluation, the relative contribution of a certain strategy can be represented as $\Delta\mathbf{acc}_{i,r_i}$ and $\Delta\mathbf{acc}_{b,k}$. Under a certain memory constraint, an optimal sparsity scheme can be found by solving an optimization problem:

$$\max_{\mathbf{k},\mathbf{i},\mathbf{r}} (\Delta\mathbf{acc}_{b,k} + \sum_{i=1}^{l} \Delta\mathbf{acc}_{i,r_i})\ \ s.t.\ Memory(\mathbf{k},\mathbf{i},\mathbf{r}) \le constraint \tag{4}$$

where $\mathbf{i}$ is the layer index whose weights is updated, and $\mathbf{r}$ is the corresponding sparsity ratio. Fig. 2 displays the sparsity of MCUNet [1], in which a wide and significant sparsity exists during training across all layers. It is observed that updating the weights and biases of the first 20 layers can be totally skipped, and only some of the last 20 layers need to be updated.

3 SPARK: EFFICIENT TINYML LEARNING ARCHITECTURE

3.1 Architecture Overview

Fig. 3 illustrates the overall architecture of *SPARK*. To support resource-constrained TinyML use cases, we build *SPARK* on top a lightweight open-source TinyML platform [2], in which a 5-stage RISC-V CPU and peripheral IOs are provided. To well support sparse training with simultaneous FP and BP operations, we developed two acceleration units, i.e. an in-pipeline acceleration unit (IPA) designed inside CPU pipeline and a dedicated Eyeriss-like MAC

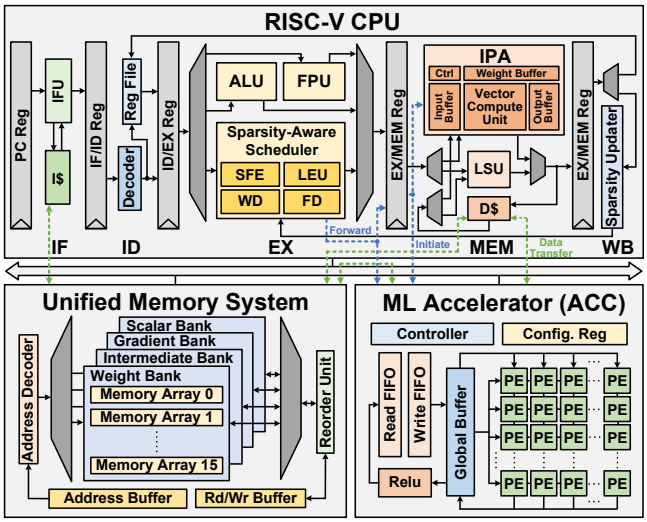


Figure 3: Architecture overview of *SPARK*.

array-based accelerator (ACC) attached at system bus. ACC is controlled by CPU instructions by reusing CPU master bus interface within MEM stage. The green dashed lines in Fig. 3 explain the dataflow across the system bus.

The in-pipeline acceleration unit can perform vector multiplication and summation by implementing multiple MAC units. We arrange IPA within the Memory Access pipeline stage for closer data movement with data cache and ACC. IPA takes multi-cycles to complete executing layer propagation depending on the workload. The input, weight and output buffers are implemented as sets of FIFOs. For each computing cycle, each FIFO pops a set of data for the vector computation unit to perform vector-multiply-vector operation. The vector computation unit consists of 4 multipliers and an adder tree to sum their output up.

To enable simultaneous FP and BP and adapt to a wide range of sparsity schemes, a sparsity-aware acceleration scheduler is designed within the Execution stage, which dynamically schedules FP and BP into IPA or ACC based on run-time sparsity. The scheduler dispatches FP and BP to achieve balanced computation and hence high utilization and throughput. Conventionally, if we switch workload between IPA and ACC, IPA/ACC output data is first stored to the memory, read by each other later. To avoid such data-moving overhead, we enable IPA and ACC to move data to each other directly, similar as data forwarding. A data-forward signal is triggered by the scheduler to indicate that the output data of both acceleration units is sent to each other. The blue dashed lines in Fig. 3 show the workload initiation signal and the data-forward signal. The sparsity updater within Write Back stage receives commit information from both acceleration units once their execution finishes. Following that, the sparsity ratio stored in scheduler will be updated.

Beyond the improved computing capability, a unified memory system is also specially developed in our work, in which a bank-associative memory structure is constructed to support unified normal weight fetch at FP and transposed weight fetch at BP. Four memory banks, i.e. weight, intermediate, gradient and scalar, are designed in the unified memory system. Two MSB bits of the address are used to distinguish the banks, i.e. similar as the sub-banking technique in CPU cache, and reduce the overhead of address decoding and encoding. The address decoder sends the address to the correct bank through a multiplexer according to the highest two bits. The output multiplexer also selects the correct data according to the fetch address.

Table 1: Custom RISC-V Extension for Simulatenous FP&BP.

instr. type	instr.	funct7	description
data transfer	*LD_IPA*	0x00	Load memory data to IPA. *rs1* denotes starting address, *rs2* denotes address offset.
	MV_IPA	0x01	Move IPA output data to itself or ACC input. Forward signal determines where to move.
	ST_IPA	0x02	Move IPA output data to memory. *rs1* denotes starting address, *rs2* denotes address offset.
	LD_ACC	0x03	Load memory data to ACC. *rs1* denotes starting address, *rs2* denotes address offset.
	MV_ACC	0x04	Move ACC output data to itself or IPA input. Forward signal determines where to move.
	ST_ACC	0x05	Move ACC output data to memory. *rs1* denotes starting address, *rs2* denotes address offset.
computation	*SCHE*	0x06	Invoke scheduler to schedule FP and BP. *rs1, rs2* denotes FP & BP layer ID.
	SMP	0x07	Simultaneous propagation in IPA & ACC. *rs1, rs2* denotes FP & BP layer ID.
update control	*SPUD*	0x08	Use sparsity updater to update sparsity.
	GAUD	0x09	Update gradient.

To effectively utilize custom in-pipeline unit, we develop several custom instructions via RISC-V extension to control the new developed hardware components, as shown in Table. 1. Our custom instructions follow RISC-V R-type format, i.e. *7 bit* funct7, *5 bit* rs2, *5 bit* rs1, *3 bit* funct3, *5 bit* rd, *7 bit* opcode. Once *opcode* == 0x0*B* && *funct3* == 0x7, our custom instructions are identified. The custom instructions contain three types: data transfer, computation and update control. *SCHE* generates the forward signal to decide whether *MV* sends data to itself or each other. The workload initiation signal generated by *SCHE* informs *LD, ST* to access the corresponding memory banks. Other instruction details are also listed in Table. 1.

3.2 Sparsity-Aware Acceleration Scheduler

In order to balance the simultaneous FP and BP computation, we design a sparsity-aware scheduler at EX stage. Fig. 4 illustrates the micro-architecture of this scheduler, which takes 6 cycles to complete. It consists of four parts: Sparsity Feature Extractor, Latency Estimate Unit, Workload Dispatcher and Forward Detector. The scheduler takes the layer ID, i.e. the layer number in the network, as input, and generates the data-forward signal to system bus and the workload initiation signal to the MEM/WB register as output. These signals will be sent to IPA or ACC at MEM stage to invoke their execution.

Sparsity Feature Extractor (SFE): In SFE, a model table is used to associate layer ID to its sparsity ratio, sparsity pattern and shape, i.e. the dimensions of the weight matrix. The model table receives the layer ID of FP and BP in *SCHE* instruction and returns their sparsity to forward and backward information buffer respectively. The extracted information will be used to estimate latency afterward.

Latency Estimate Unit (LEU): The run-time latency of each sparse layer is evaluated by the following equation:

$$\textit{Estimated Layer Latency} = \alpha \times \beta \times \textit{Shape} \times \textit{Sparsity Ratio} \quad (5)$$

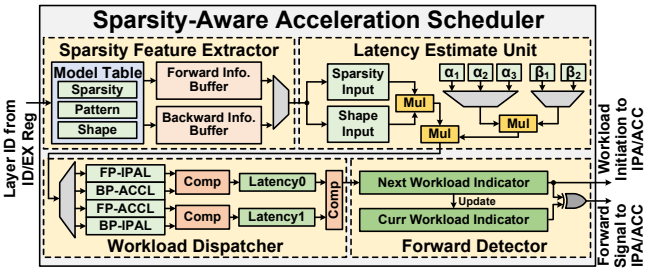


Figure 4: Micro-architecture of the sparsity-aware scheduler.

where α denotes the sparsity pattern (partial weight, full weight, bias only, etc.) coefficient, which is programmable before model deployment. We use another coefficient β to represent different computation performance of IPA and ACC. LEU reads forward and backward information buffer to get sparsity pattern, shape and sparsity ratio. Shape and sparsity ratio are directly loaded to input buffer, while sparsity pattern controls the selection of certain α. After all inputs are ready, the estimated latency is calculated and then fed to a latency list.

Workload Dispatcher (WD): A latency list stores four latency values in order: FP latency in IPA (FP-IPAL), BP latency in ACC (BP-ACCL), FP latency in ACC (FP-ACCL) and BP latency in IPA (BP-IPAL). WD reads this list and compares them to determine the final scheduling scheme (FSS) according to the following formula:

$$FSS = \min\big(\max(\textit{FP-IPAL}, \textit{BP-ACCL}), \max(\textit{FP-ACCL}, \textit{BP-IPAL})\big) \tag{6}$$

The scheduler can dispatch FP to IPA and BP to ACC, or vice versa. The *min* in Equation (6) chooses the dispatch combination with smaller latency. The comparison result is then sent to the subsequent FD unit.

Forward Detector (FD): To overlap the data movement within IPA/ACC switching, we refer the design insights from data forwarding in CPU micro-architecture and design the FD unit. The final scheduling is stored in a *1-bit* register called Next Workload Indicator (NWI), where *1'b0* denotes FP in IPA and BP in ACC, while *1'b1* denotes BP in IPA and FP in ACC. Besides NWI, we introduce another *1-bit* register called Current Workload Indicator (CWI) to record current acceleration unit status. An XOR gate is applied to generate the data forward signal. If the forward signal is positive, IPA and ACC will feed their output to each other's input buffer, preparing for next layer computation.

3.3 Unified Memory for Transposable Dataflow

To support diverse data fetching patterns from IPA and ACC, meticulously designed banks for training are dedicated in the unified memory system to accommodate various matrix access patterns. The normal read pattern and transposed read pattern are unified in out memory system. One bank includes 16 memory arrays and each array has one read-write port.

Without loss of generality, we use a mapping of $n \times n$ matrix to n memory arrays for illustration, with each array storing n data. The mapping strategy is described as follows:

$$Mat[i][j] \xrightarrow{mapping} Arr[(i+j) \bmod n][j], \quad i, j \in [0, n-1]$$

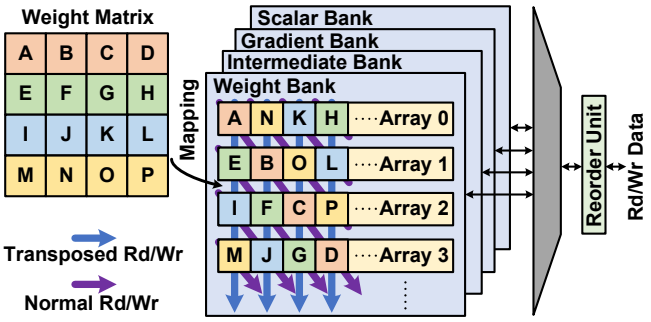


Figure 5: Unified memory mapping and read/write pattern.

For simplicity, we call the $(j+1)^{th}$ data of the $(i+1)^{th}$ array $Arr[i][j]$, the $(i+1)^{th}$ row and the $(j+1)^{th}$ column of the matrix $Mat[i][j]$. To read the $(k+1)^{th}$ row or column of the matrix, each memory array reads one data according to the mapping strategy. The data array then enters the reorder unit to cyclically shift left for k times. As for write, the data array first enters the reorder unit to cyclically shift right, writing to corresponding address later.

Fig. 5 shows an example of mapping a 4×4 matrix to memory arrays with diverse data fetch modes. The blue arrows indicate column read/write dataflow, i.e. transposed read/write, while the purple arrows indicate row read/write dataflow, i.e. normal read/write. With this cyclic-storage memory bank, we implement transposable dataflow of reading and writing matrices. $\mathbf{W}_i, \mathbf{A}_i, \frac{\partial L}{\partial \mathbf{W}_i}$ in equation (2) are stored in weight, intermediate and gradient bank respectively. During FP, only regular data load/store pattern is needed. As $\mathbf{A}_i, \mathbf{W}_i$ requires transposed load and store during BP, thus both weight and intermediate banks apply cyclic-storage format.

4 SPARSITY-AWARE SCHEDULING FOR SIMULTANEOUS FP AND BP

Conventional TinyML architecture may involve one CPU or a ML acceleration unit in low-power processor chip. To enable on-device training, the CPU or the accelerator need to perform FP and BP in sequence without any parallelism, as shown in Fig. 6 (a). When dynamic sparsity is considered, the execution time for each training iteration varies, i.e. latency inversely proportional to the model sparsity, leading to unpredictable training latency. This situation is fatal for scenarios that require stable latency.

By adding the IPA unit, we can utilize the in-pipeline acceleration from CPU to enable parallel computation together with ACC. However, without workload scheduling, e.g. FP always executed in IPA and BP in ACC, in-flight bubbles appear due to unbalanced execution, leading to limited performance improvement. As demonstrated in Fig. 6 (b), parallel acceleration such as the i^{th} FP in IPA and the $(i-1)^{th}$ BP in ACC, are always overlapped. The task dependency forces i^{th} BP cannot start execution until the i^{th} FP finishes. Hence the scheduling between IPA and ACC is critical to enable high utilization rate and performance improvement.

To improve utilization and throughput, the designed in-pipeline scheduler estimates the latency based on sparsity level and arranges FP and BP computation into different acceleration units. In our scheduler, layer-wise latency is estimated as the sparsity ratio varies among different layers.

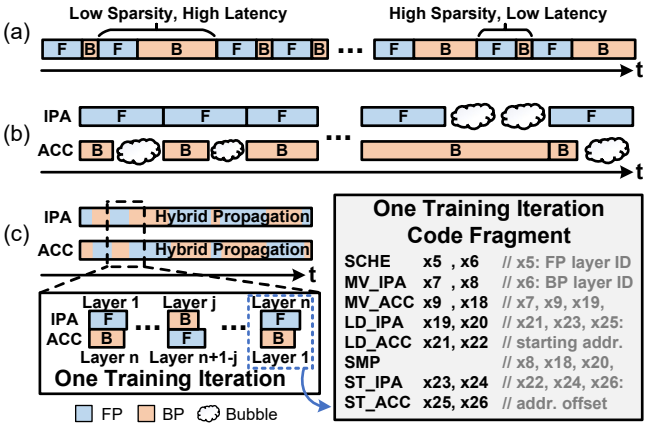


Figure 6: (a) Traditional architecture timeline. (b) *SPARK* timeline without scheduling. (c) *SPARK* timeline with scheduling and a code fragment for one training iteration.

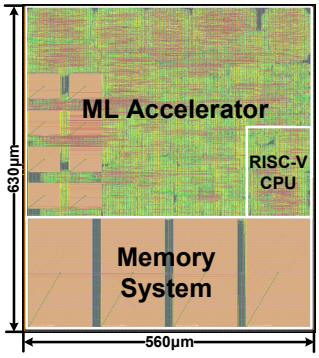
The operation of one sparse training iteration is described as follows. The scheduler is given the task of dispatching the j^{th} layer of the i^{th} FP and the $(n-j)^{th}$ layer of the $(i-1)^{th}$ BP to IPA and ACC if the TinyML model has n layers. Using equation (5) mentioned in Sec.3.2, the scheduler evaluates the possible two dispatch combinations. By comparing the latency of these two combinations, one combination is dispatched and IPA and ACC are simultaneously initiated to execute their workload. If both workload complete execution at the same time, the resulted performance bubble is zero. As for most cases, the one finished earlier will wait until both FP and BP are completed. Repeating the previous steps, the scheduler will then dispatch the $(j+1)^{th}$ layer of the i^{th} FP and the $(n-j-1)^{th}$ layer of the $(i-1)^{th}$ BP to IPA and ACC.

The dynamic sparsity scheme we adopt updates sparsity information at the end of one training iteration. After n times of dispatching the FP layer and the BP layer, the hyper-parameters recording sparsity information in IPA and ACC are sent to the sparsity updater. The sparsity updater then feeds back the latest sparsity information to the scheduler. Last, the weight is updated and the next training iteration starts. Fig. 6 (c) shows the timeline of our scheduling pattern and a code fragment based on our custom RISC-V instructions in Table. 1. As we dynamically allocate tasks to two accelerators at layer granularity, better performance and high resource utilization can be achieved.

5 EVALUATIONS

5.1 Implementations

Our *SPARK* architecture is designed and implemented using TSMC 22nm technology. *SPARK* contains an open-source 5-stage RISC-V Vexriscv CPU, a unified memory system with 64Kb SRAM and a in-house ML accelerator with local scratchpads. The IPA and the sparsity-aware scheduler are designed within RISC-V CPU with small area. The RTL is synthesized and placed and routed by commercial EDA tools, as the layout shown in Fig. 7. The energy, performance and area results are reported based on post-P&R netlist at typical corner. The total area of *SPARK* is 0.35mm^2, in which the in-pipeline acceleration unit and the scheduler only consume 2.27% area overhead. The control logic of our unified memory system accounts for negligible area overhead. *SPARK* can work under the voltage of 0.6V to 0.9V and the power consumption is 0.21mW to 9.45mW. When working at 200MHz and 0.9V, *SPARK* reaches its peak performance, 8GOPS at 8bit. The peak energy efficiency is 1.9TOPS/W at 10MHz and 0.6V.



Technology (nm)	22
Die Area (mm^2)	0.35
Supply Voltage (V)	0.6-0.9
Frequency (MHz)	10-200
Power (mW)	0.21-9.45
Peak Performance (GOPS)	8.0
Energy Efficiency (TOPS/W)	1.90

Figure 7: Layout of *SPARK* using TSMC 22nm technology.

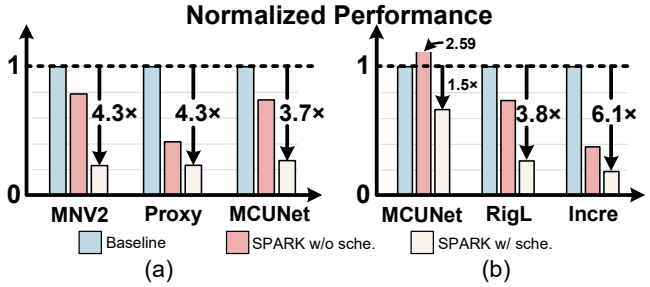


Figure 8: Normalized training cycles of (a) different TinyML models and (b) sparsity schemes.

5.2 Performance and Efficiency

We evaluate the performance and efficiency of *SPARK* for TinyML learning cases. Three popular TinyML models are utilized for our evaluations, i.e. MobileNetV2 [8], ProxylessNAS [4] and MCUNet [1]. The baseline is set as the performance when running one training iteration on a stand-alone ML accelerator. For each TinyML model, we apply several sparsity schemes and deploy the model on the baseline, *SPARK* without special scheduling, and *SPARK* with our sparsity-aware scheduling. Fig. 8(a) shows the normalized performance for training different models. Compared with the baseline ACC, the in-pipeline IPA bring latency reduction by 21.2%, 58.4%, and 25.9% to train three models respectively. With the help of sparsity-aware dynamic scheduling, the performance is further improved by 3.43×, 1.77× and 2.74×. Overall, *SPARK* achieves 4.3×, 4.3× and 3.7× improvements with almost negligible area overhead.

We also evaluate the performance improvement on either static sparsity schemes, e.g. MCUNet [1], or dynamic sparsity schemes, e.g. RigL [16] and IncrementalPruning [5]. MCUNet adopts a static sparsity where the sparsity level is fixed, while RigL redistributes sparsity among layers for every several iterations and IncrementalPruning gradually increases sparsity ratio during training process. Fig. 8(b) shows the normalized performance of different sparsity

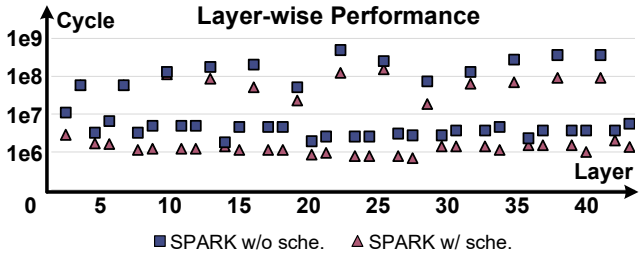


Figure 9: Layer-wise MCUNet using RigL sparsity.

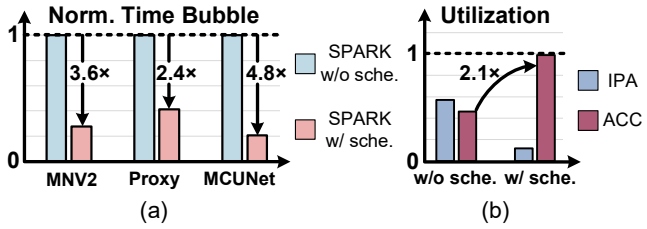


Figure 10: (a) Time bubble with/without scheduling. (b) Hardware utilization with/without scheduling.

schemes. *SPARK* achieves overall improvement by 1.5×, 3.8× and 6.1× on the three sparsity schemes respectively, demonstrating compatibility with various sparsity schemes. Note that *SPARK* without scheduling for MCUNet takes 1.59× more cycles than the baseline ACC and the overall improvement is only 1.5×, because it adopts a static sparsity with extreme backward gradient pruning, leading to significant imbalance between FP and BP. The layer-wise performance of MCUNet adopting RigL sparsity scheme is shown in Fig. 9. Although RigL is a dynamic sparsity scheme which changes layer-wise sparsity ratio every several iterations, our sparsity-aware scheduler can sense the sparsity updating and adjust workload scheduling timely, lowering the cycle counts of most layers.

Our sparsity-aware scheduler can effectively reduce time bubbles in training iterations. Fig. 10(a) demonstrates the normalized time bubbles in the training. Compared to no special scheduling, the sparsity-aware scheduler reduces bubble time by average of 3.6×, 2.4× and 4.8× in one training iteration for three models. We observe ACC utilization on MCUNet is improved from 46.30% to almost full utilization of 98.96% with the help of our scheduler.

We also compare *SPARK* with off-shelf low-power edge devices, i.e. a Cortex-M7 based micro-controller STM32F746 and a low-power embedded FPGA, Xilinx Arty A7-35T, for the TinyML learning applications. For one training iteration of MCUNet, *SPARK* only consumes 79ms and costs less than 0.3mJ, while low-power MCU and FPGA consumes 556ms with 11.3mJ and 927ms with 880.7mJ. As the comparison summarized in Table. 2, *SPARK* achieves 7.04× and 11.73× better performance with 38.45× and 2985.25× better efficiency than low-power MCU and FPGA.

6 CONCLUSION

In this paper, we present *SPARK*, an efficient hybrid acceleration architecture for TinyML learning with run-time sparsity-aware scheduling. Our simultaneous FP and BP outperforms conventional learning process and achieves significant speedup for TinyML learning. The in-pipeline sparsity-aware scheduler contributes to the latency improvement by 35.17% in average and improves hardware utilization from 46.30% to 98.96%, demonstrating the flexibility to adopting various sparsity schemes. The unified transposable-fetch memory system reduces memory access by 5.1× in average, making on-device training more efficient. We evaluate the performance improvement and power overhead using TSMC 22nm technology. The result shows that our custom units lead to only 2.27% area increase with 4.1× performance improvement. Compared with existing low-power edge devices, *SPARK* exhibits 9.4× greater training latency with 446.0× lower energy in average. *SPARK* can also be effectively combined with other optimizations such as quantization, making it possible to deploy larger models and achieve higher accuracy.

Table 2: One training iteration of MCUNet on edge devices.

	SPARK	STM32F746	Arty A7-35T
latency (ms)	79	556	927
energy (mJ)	0.295	11.342	880.65

ACKNOWLEDGEMENT

This work was supported in part by National Key R&D Program 2022YFB4400600, NSFC Grant No. U23A6007, 92164301, 62225401.

REFERENCES

[1] J. Lin *et al.*, "On-device training under 256kb memory," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2022.
[2] S. Prakash *et al.*, "CFU playground: Full-stack open-source framework for tiny machine learning (tinyml) acceleration on fpgas," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.
[3] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 52, no. 1, pp. 127–138, 2017.
[4] H. Cai *et al.*, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *International Conference on Learning Representations (ICLR)*, 2019.
[5] M. Zhu *et al.*, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.
[6] L. Yang *et al.*, "Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks," in *IEEE/ACM Design Automation Conference (DAC)*, 2020.
[7] D. Rossi *et al.*, "4.4 a 1.3tops/w @ 32gops fully integrated 10-core soc for iot end-nodes with 1.7w cognitive wake-up from mram-based state-retentive sleep mode," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 60–61, 2021.
[8] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.
[9] S. J. Pan *et al.*, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, pp. 1345–1359, OCT 2010.
[10] Q. Yang *et al.*, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology*, vol. 10, Feb 2019.
[11] J. Lee *et al.*, "7.7 LNPU: A 25.3 tflops/w sparse deep-neural-network learning processor with fine-grained mixed precision of fp8-fp16," in *IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 142–144, IEEE, 2019.
[12] S. Kang *et al.*, "7.4 GANPU: A 135tflops/w multi-dnn training processor for gans with speculative dual-sparsity exploitation," in *IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 140–142, IEEE, 2020.
[13] Z. Wang *et al.*, "CPE: An energy-efficient edge-device training with multi-dimensional compression mechanism," in *IEEE/ACM Design Automation Conference (DAC)*, 2023.
[14] D. Trilla *et al.*, "Novia: A framework for discovering non-conventional inline accelerators," in *International Symposium on Microarchitecture (MICRO)*, 2021.
[15] Z. Jiang *et al.*, "Blueface: Integrating an accelerator into the core's pipeline through algorithm-interface co-design for real-time socs," in *IEEE/ACM Design Automation Conference (DAC)*, 2023.
[16] U. Evci *et al.*, "Rigging the lottery: Making all tickets winners," in *International Conference on Machine Learning (ICML)*, pp. 2943–2952, PMLR, 2020.