V10: Hardware-Assisted NPU Multi-tenancy for Improved Resource Utilization and Fairness

Yuqi Xue
yuqixue2@illinois.edu
University of Illinois at Urbana-Champaign

Lifeng Nai
lnai@google.com
Google

Yiqi Liu
yiqiliu2@illinois.edu
University of Illinois at Urbana-Champaign

Jian Huang
jianh@illinois.edu
University of Illinois at Urbana-Champaign

ABSTRACT

Modern cloud platforms have deployed neural processing units (NPUs) like Google Cloud TPUs to accelerate online machine learning (ML) inference services. To improve the resource utilization of NPUs, they allow multiple ML applications to share the same NPU, and developed both time-multiplexed and preemptive-based sharing mechanisms. However, our study with real-world NPUs discloses that these approaches suffer from surprisingly low utilization, due to the lack of support for fine-grained hardware resource sharing in the NPU. Specifically, its separate systolic array and vector unit cannot be fully utilized at the same time, which requires fundamental hardware assistance for supporting multi-tenancy.

In this paper, we present V10, a hardware-assisted NPU multi-tenancy framework for improving resource utilization, while ensuring fairness for different ML services. We rethink the NPU architecture for supporting multi-tenancy. V10 employs an operator scheduler for enabling concurrent operator executions on the systolic array and the vector unit and offers flexibility for enforcing different priority-based resource-sharing mechanisms. V10 also enables fine-grained operator preemption and lightweight context switch in the NPU. To further improve NPU utilization, V10 also develops a clustering-based workload collocation mechanism for identifying the best-matching ML services on a shared NPU. We implement V10 with an NPU simulator. Our experiments with various ML workloads from MLPerf AI Benchmarks demonstrate that V10 can improve the overall NPU utilization by 1.64×, reduce the average latency of ML services by 1.56×, and tail latency by 1.74× on average, in comparison with state-of-the-art NPU multi-tenancy approaches.

CCS CONCEPTS

KEYWORDS
Neural Processing Unit, Multi-tenancy, ML Accelerator

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM ‘23, June 17–21, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589089

1 INTRODUCTION

Recently, we have seen an increasing demand for machine learning as a service (MLaaS) on various cloud platforms [7, 9, 40, 45, 47]. These machine learning (ML) services are becoming the backbone of cloud applications today, including smart AI assistants, language translation, image and video analysis, and recommendations. To support MLaaS, cloud platforms have deployed neural processing units (NPUs) such as Google Cloud TPUs [23] that are specialized hardware accelerators for deep neural networks (DNN).

A typical NPU design aims to accelerate the most common matrix-matrix multiplication and convolution operations in DNN models. Therefore, an NPU core usually consists of a large systolic array (SA) that exploits the data reuse pattern of matrix multiplications and a vector unit (VU) for other generic vector operations such as activations and reductions, as shown in Figure 2. To maximize the efficiency of NPUs, their compilers, such as TensorFlow XLA, usually conduct various code optimizations before offloading the compiled ML model to the device memory [6, 14, 49].

To use NPUs in the cloud, a simple approach is to assign each NPU core exclusively to an ML service, which completely disallows resource sharing (Figure 1a). In order to best utilize NPUs, modern cloud platforms enable the sharing of NPUs by queuing the incoming ML workloads and executing them following different scheduling policies, such as the first-come first-served policy and the priority-based policy [11, 12]. Such an approach enables the time sharing of NPUs at kernel-level granularity. Most recently, PREMA [16] proposed a preemption mechanism for NPUs, it can interrupt an executing ML kernel in the middle and schedule another kernel (Figure 1b). However, they conduct task scheduling at a coarse granularity, and do not support concurrent execution of multi-tenant ML workloads. This inevitably misses the opportunity to explore the underutilized hardware in NPUs.

To understand NPU utilization, we first conduct a thorough study on real Google Cloud TPUs. We run various ML workloads from MLPerf AI Benchmarks [44], and profile the resource utilization of the core components of a TPU, including the matrix multiplication
unit (MXU, the systolic array), the vector processing unit (VPU), and the high-bandwidth memory (HBM).

We find that (1) for the majority of ML inference services, a single workload with various input batch sizes can only utilize less than half of the total available FLOPS (floating-point operations per second) of a TPU core. (2) This is mostly due to the imbalanced use of the MXU and VPU in an ML workload. According to our study, ML inference workloads are either MXU-intensive or VPU-intensive. As many common tensor operators can only be executed either on MXU or VPU, it causes the idleness of compute units. (3) Even if the operators on MXU and VPU are balanced in an ML workload, we still cannot often occupy both compute units simultaneously, due to the data dependencies between these tensor operators. (4) The underutilized compute units further cause HBM bandwidth underutilization, as the off-chip HBM is usually designed to match the peak computation capability of the systolic array. (5) Although we enable the time sharing of TPs with multiple ML services and apply state-of-the-art preemption mechanisms [16], we still cannot significantly improve the compute utilization as discussed above.

Thus, we are motivated to rethink the NPU architecture for achieving both improved resource utilization and fairness for multi-tenant ML services. We develop a hardware-assisted NPU multi-tenancy framework V10. It enables fine-grained concurrent execution of ML workloads by employing an operator scheduler in the NPU. Such a scheduler exploits the idle cycles caused by the imbalanced use of SAs and VUs in an ML kernel and enables concurrent execution of operators from different ML workloads (Figure 1c).

Since the size of tensor operators varies, different operators require different amounts of hardware resources, which can cause unfairness and starvations in the operator scheduling. For instance, the large operators will block small operators of the collocated ML workload. This not only causes severe unfairness, but also prevents potential opportunities for overlapping the operator executions on the SA and VU. To overcome this challenge, we develop a fairness metric for multi-tenant ML workloads, and propose a unique preemption mechanism for NPUs at the operator-level granularity with low context switch overhead. Instead of re-executing an entire tensor operator after a context switch, we enable low-overhead recomputation by asynchronously checkpointing input data and overlapping the switching of two operators. In addition, our design offers the flexibility for enforcing different priority-based scheduling policies, such that we can satisfy different service-level agreements (SLAs) for ML services (see §5.6).

As we scale multiple ML workloads deployment on the shared NPUs, we wish to maximize the opportunities of exploring the idle cycles of NPU compute units, while minimizing the resource contention among these collocated ML workloads. Naively collocating two SA-intensive workloads on the same NPU will cause not only low resource utilization but also worse aggregated throughput due to resource contention. Therefore, it is critical to identify pairs of ML workloads that have compatible resource requirements. Unfortunately, prior studies [17, 32, 35] on workload collocation cannot be directly applied to ML workloads, as each ML workload has diverse computational characteristics. To this end, we develop a learning-based clustering mechanism, with the intuition that workloads having similar resource requirements would not be compatible with each other. We first categorize ML workloads into different clusters based on the similarities of their extracted features (e.g., operator size and compute resource requirements). After that, we pair ML workloads from disjoint clusters, and identify the compatible ones based on their estimated collocation performance. Our experiments show that such an approach achieves high accuracy (84.73%), compared to the brute-force methodology in which we manually examine all the possible workload combinations.

We implement V10 with an NPU simulator based on public TPU hardware parameters [37] as well as the profiled hardware behaviors on real Google Cloud TPUs. All the execution traces of ML workloads from MLPerf [44] are collected on the Google Cloud TPU platform. V10 requires minimal hardware modifications (0.003% die area overhead) to current NPU hardware. We show that V10 improves NPU utilization by 1.64×, increases the aggregated throughput by 1.57× for collocated ML workloads, reduces the average latency of each workload by 1.56×, and decreases the tail latency by 1.74×, compared to the state-of-the-art preemptive multitasking approach. We summarize our contributions as follows:

- We conduct a thorough characterization study of the resource utilization of NPUs using real hardware devices and ML inference workloads, and report our findings in §2.
- We develop a tensor operator scheduler and enable multi-tenancy for NPUs at operator-level granularity to fully utilize the compute units with multi-tenant workloads (§3.2).
- We enable a flexible fairness mechanism for shared NPUs by developing a lightweight operator-level preemption scheme with low context-switch overhead (§3.3).
- We propose a learning-based clustering scheme, which can efficiently identify compatible ML workloads with high accuracy for further utilization improvement (§3.4).
- We develop an end-to-end hardware-assisted NPU multi-tenancy framework V10 for multi-tenant ML services, and evaluate its efficiency with various ML workloads (§5).
2 CHARACTERIZATION STUDY OF NPU UTILIZATION

To facilitate our study, we first introduce the baseline NPU architecture. Then, we show our study results.

2.1 NPU Architecture

Without loss of generality, we show the NPU architecture derived from the state-of-the-art NPUs in production such as Google Cloud TPUv2 [27]. As shown in Figure 2, an NPU consists of a systolic array, a vector unit, a set of vector registers, and software-managed SRAM buffers (i.e., instruction memory and vector memory). The SRAM buffers are filled by DMA operations that execute independently from the core pipeline, such that the NPU can overlap computation and data movement between on-chip SRAM and off-chip HBM. The vector unit has multiple SIMD units, which access the vector memory via load and store instructions and perform computations with 8 × 128 2D vector registers. It also orchestrates the push and pop operations to stream data to/from the systolic array via dedicated FIFO buffers. The systolic array consists of a set of processing elements (e.g., 128 × 128 PEs) to exploit the data reuse and parallelism of matrix multiplications and convolutions. Each PE performs a multiply-accumulate operation per cycle. During execution, the systolic array first loads a weight matrix into the PEs, and then streams the input matrix from the left edge of the array. Simultaneously, the output will be streamed out to the vector register file via the FIFO, and written back to the vector memory.

Given a compiled DNN model that has a stream of tensor operators, each operator has diverse NPU instructions, including (1) push/pushw %src, which sends eight 128-wide vectors (input tensor or weight) from the vector register %src to the systolic array in 8 cycles; (2) pop %dst, which reads eight 128-wide vectors from the systolic array and write into %dst in 8 cycles; (3) ld %dst, [vmem], which loads from the vector memory; (4) st %dst, [vmem], which stores into the vector memory; and (5) various ALU instructions in the vector unit that perform element-wise SIMD operations.

To maximally exploit the parallelism of NPUs, various compiler optimizations have been developed in popular ML frameworks such as TensorFlow [14, 31, 48, 49]. In this study, we use TensorFlow v2.10 with the XLA compiler to run the ML models.
reshape, and element-wise, can only execute on the VPU, this leads to MXU idleness during the absence of matrix multiplications or convolutions. We show the temporal utilization of MXU and VPU in Figure 4 and Figure 5, respectively. Many DNN workloads leave the MXU idle for 48% of the total execution time on average. Likewise, the VPU is also significantly underutilized.

**Observation (O1):** The compute resources on the TPU core are significantly underutilized due to the temporal idleness of MXU and VPU.

**Imbalanced use of MXU and VPU.** To reduce the idleness of MXU and VPU, one solution is to overlap the execution of MXU and VPU with compiler optimizations. However, this yields limited benefits for the following reasons.

First, data dependencies limit operator-level parallelism. To overlap an MXU operator with a VPU operator, there must be no data dependency between them. However, this is rarely the case in DNN models. VPU favors element-wise operators like ReLU, while MXU favors spatial-reduction operators like matrix multiplication and convolution. In most DNN models, these two types of operators belong to different dependent DNN layers, and have to run sequentially. Therefore, within a DNN model, it is hard to overlap VPU and MXU operators. Although it is possible to pipeline some MXU and VPU operations in a fine-grained tile-by-tile manner (such as a fully-connected layer followed by element-wise activations), the VPU execution time is still much smaller than that of MXU [27].

To confirm this finding, we built a directed acyclic graph (DAG) with operators as nodes and dependencies as edges. Any path inside the DAG is a sequence of operators that cannot be parallelized. Thus, the total execution time of operators on the longest path is a lower bound of the execution time of the DNN model, with the assumption that all operators without data dependencies are executed in parallel. As shown in Figure 6, the speedup of such compiler-parallelized execution over sequential execution is marginal (6.7% on average).

Second, even if there are no inter-operator data dependencies, parallelism within a single DNN workload is still limited by the imbalanced use of MXU and VPU, as shown in Figure 4 and Figure 5. This limits operator-level parallelism, due to the fact that there are simply not enough VPU operators to be parallelized with these MXU operators, as the execution time of MXU operators in a DNN workload overwhelms that of VPU operators (or vice versa). Thus, DNN workloads with large MXU/VPU imbalances are bottlenecked by the type of compute unit that receives higher demands.

The imbalanced use of MXU and VPU is determined by the DNN model architecture and cannot be easily addressed by the compiler. For example, BERT and ResNet are matrix multiplication or convolution-intensive, so they involve more MXU operators. In contrast, recommendation models like DLRM and detection models like ShapeMask are bottlenecked by element-wise VPU operations. Because larger batch sizes help reduce MXU padding overhead, the XLA compiler will map more operators to the MXU for improved performance. This will make the VPU utilization relatively worse.

**Observation (O2):** The temporal idleness of MXU and VPU is caused by limited operator-level parallelism, a consequence of data dependencies and imbalanced use of MXU and VPU. As the limitations are inherent in the DNN model architectures, it is hard to further improve NPU utilization of a single DNN workload via compiler optimizations.

**Correlation between compute and memory bandwidth utilizations.** We also profile the average memory bandwidth utilization of each DNN workload in Figure 7. As the batch size increases, the memory bandwidth utilization decreases (except for Transformer1). This is because the computational intensity increases with larger batch sizes, so the DNN workloads observe more data reuse. To further understand this, we use the roofline model to demonstrate the correlations between the compute and memory intensity in Figure 8. With a larger batch size, the operation intensity (FLOPs/byte) increases for most DNN inference workloads. However, they still cannot reach the peak FLOPS due to the limited parallelism between MXU and VPU (O2). The corresponding memory bandwidth utilization also cannot achieve the peak bandwidth. As the per-core HBM bandwidth (e.g., 330GB/s) was designed to match the maximum input rate of the MXU (systolic array), the low compute utilization causes low HBM bandwidth utilization.

**Observation (O3):** The off-chip HBM bandwidth is underutilized as a consequence of FLOPS underutilization in NPs, as the HBM was usually designed to match the peak computation capability of the systolic array.

---

1The Transformer model uses a beam search decoder for generating the final outputs. It incurs more memory accesses as we increase its batch size.
Low NPU utilization with preemptive multi-tasking. NPU multi-tenancy allows two or more inference tasks to share the same NPU core, however, the current NPU support for multi-tenancy is limited. The most advanced approach is preemptive multitasking that enables the time sharing of an NPU core by preempting a workload at the task level. To examine the effectiveness of this approach, we collocate two DNN inference workloads from Table 4, and implement a preemptive multitasking scheme (see the details in §5.1). We show the MXU/VPU utilization of each collocated workload in Figure 9. For half of the workload combinations (e.g., BERT+NCF), both MXU and VPU still have low utilization (50% on average). As the preemptive multitasking scheme enables fair time-sharing of an NPU core between workloads at the task level, the overall MXU/VPU utilization appears "balanced". However, this does not fundamentally solve the underutilization problem, because preemptive multitasking still cannot enable overlapping execution of MXU and VPU operators at the architectural level. Even worse, for some collocations (e.g., BERT+RsNt and DLRM+RtNt), there is little room for overlapping execution as they intrinsically contend for the same type of compute units.

Observation (O4): The state-of-the-art preemptive-based multi-tasking scheme still suffers from low NPU utilization and imbalanced use of MXU and VPU. Blindly collocating workloads on the same NPU core causes resource contention, making it harder to improve NPU utilization.

3 DESIGN AND IMPLEMENTATION

Our study (§2.2) motivates V10, a multi-tenant NPU that improves resource utilization by simultaneously executing independent systolic array (SA) and vector unit (VU) operators from different workloads. This section presents the overview of V10 (§3.1) and the details of each design component.

3.1 Overview of V10

The core of V10 is a tensor operator scheduler as shown in Figure 10. It is located at the front end of the NPU pipeline, between the instruction memory and the SA/VU, to control the instruction fetch and issue logic. The scheduler minimizes hardware modifications by leveraging the existing hardware capability to dispatch SA and VU operations in parallel. First, the operator dispatch logic enables simultaneous execution of multiple operators on one NPU core with a flexible priority-based scheduling policy to enforce different SLAs for the ML services (§3.2). Second, since DNN operators have vastly different execution times, long-running operators may block short operators and cause severe unfairness and sub-optimal resource utilization. Thus, V10 employs a lightweight operator preemption mechanism to balance operator execution time, which greatly improves both utilization and fairness (§3.3). Third, even if the scheduler makes perfect scheduling decisions, improvements in resource utilization may still be marginal if the collocated workloads have conflicting resource demands. Thus, V10 minimizes resource contention with a clustering-based collocation mechanism to identify workloads with compatible resource demands (§3.4).

3.2 Tensor Operator Scheduler

To enable the simultaneous execution of multiple operators on functional units (FU) like SAs and VUs, the tensor operator scheduler selects independent operators from different DNN workloads and dispatches them to multiple FUs.

Operator Dispatch Logic. To track the execution states of each workload and their operators, V10’s scheduler maintains a workload context table shown in Figure 11. Because the operators within one workload execute sequentially, each row only need to track the most recent operator of the workload.

For each operator, the scheduler uses DMA to load the instructions from the off-chip HBM into the on-chip instruction memory. The Ready bit indicates whether the DMA is completed and the operator can start execution. Then, the scheduling policy uses the context table to decide which operator will be executed next, which will be elaborated on later. Periodically, a preemption timer will trigger the scheduling policy to examine whether an operator should be preempted. If so, the preemption module dynamically generates instructions to save the context for the preempted operator (§3.3).

Once a set of ready operators are selected for execution, the scheduler sets the Active bits and zeros out the Ready bits for them. It then fetches the instructions of these operators from the instruction memory and issues them to the corresponding FUs.
For example, an SA operator will involve push/pop instructions, while a VU operator will execute vector ALU instructions. Once an operator starts execution, it will occupy the corresponding FU until it finishes or is preempted. Then, if the DMA operation that fetches instructions for the next operator is also done, the Active bit will be zeroed, the Ready bit will be set, and the scheduling policy will be invoked to assign the next operator to the free FU.

**Scheduling Policies.** To keep the FUs busy and maximize resource utilization, the scheduler will issue an operator as soon as an operator is ready and an FU is idle. If there are more ready operators than available FUs, the scheduling policy will be invoked to decide which operator to execute next.

1) **Round-Robin Scheduling Policy.** The most basic scheduling policy is Round-Robin (RR), which circulates through all workloads with ready operators. This naive policy enables the basic operator scheduling logic on an NPU core, but it has at least two drawbacks. First, RR only balances the number of executed operators between workloads, rather than their execution time. As a result, workloads with longer operator lengths will occupy an FU and starve workloads with shorter operators, which leads to sub-optimal utilization and unfairness (see §3). Second, RR lacks the flexibility to support SLAs for multi-tenancy, e.g., a cloud platform provider may configure different priorities for different user workloads to guarantee SLAs, but RR treats all workloads equally.

2) **Priority-Based Scheduling Policy.** To enable flexible priority configuration on the multi-tenant NPU, and to reduce the unfairness caused by imbalanced operator length between workloads, we developed the priority-based scheduling policy in Algorithm 1. The intuition is that a workload should spend computation cycles proportional to its relative priority [20]. With this, we define active_rate of a workload as the ratio between the workload’s active execution time (active_time) and its total time since it has arrived at the NPU (total_time). Thus, active_rate = \frac{active_time}{total_time} indicates the relative throughput a workload gets compared to the ideal throughput when it runs on a dedicated NPU core. For example, a workload gets half of its ideal throughput when its active_rate = 1/2.

To maintain the proportionality between active rates and priorities, the scheduler aims to keep the proportional active rate

```plaintext
Algorithm 1 Priority-Based Scheduling Policy.

1: function pick_next_workload(act_list, tot_list, fu_type)
2: for i in range(num_workloads) do
3:    arp_list[i] = act_list[i] / tot_list[i] / priority_list[i]
4: end for
5: for i in range(num_workloads) do
6:    idx = index of the i-th smallest element in arp_list
7:    if workload_list[idx].running == False then
8:        if workload_list[idx].op_type == fu_type then
9:            return workload_list[idx]
10:       end if
11:    end if
12: end for
13: return NO_WORKLOAD_AVAILABLE
14: end function
```

For example, an SA operator will involve push/pop instructions, while a VU operator will execute vector ALU instructions. Once an operator starts execution, it will occupy the corresponding FU until it finishes or is preempted. Then, if the DMA operation that fetches instructions for the next operator is also done, the Active bit will be zeroed, the Ready bit will be set, and the scheduling policy will be invoked to assign the next operator to the free FU.

**Scheduling Policies.** To keep the FUs busy and maximize resource utilization, the scheduler will issue an operator as soon as an operator is ready and an FU is idle. If there are more ready operators than available FUs, the scheduling policy will be invoked to decide which operator to execute next.

**1) Round-Robin Scheduling Policy.** The most basic scheduling policy is Round-Robin (RR), which circulates through all workloads with ready operators. This naive policy enables the basic operator scheduling logic on an NPU core, but it has at least two drawbacks. First, RR only balances the number of executed operators between workloads, rather than their execution time. As a result, workloads with longer operator lengths will occupy an FU and starve workloads with shorter operators, which leads to sub-optimal utilization and unfairness (see §3). Second, RR lacks the flexibility to support SLAs for multi-tenancy, e.g., a cloud platform provider may configure different priorities for different user workloads to guarantee SLAs, but RR treats all workloads equally.

**2) Priority-Based Scheduling Policy.** To enable flexible priority configuration on the multi-tenant NPU, and to reduce the unfairness caused by imbalanced operator length between workloads, we developed the priority-based scheduling policy in Algorithm 1. The intuition is that a workload should spend computation cycles proportional to its relative priority [20]. With this, we define active_rate of a workload as the ratio between the workload’s active execution time (active_time) and its total time since it has arrived at the NPU (total_time). Thus, active_rate = \frac{active_time}{total_time} indicates the relative throughput a workload gets compared to the ideal throughput when it runs on a dedicated NPU core. For example, a workload gets half of its ideal throughput when its active_rate = 1/2.

To maintain the proportionality between active rates and priorities, the scheduler aims to keep the proportional active rate

```
Algorithm 1 Priority-Based Scheduling Policy.
1: function pick_next_workload(act_list, tot_list, fu_type)
2:   for i in range(num_workloads) do
3:     arp_list[i] = act_list[i] / tot_list[i] / priority_list[i]
4:   end for
5:   for i in range(num_workloads) do
6:     idx = index of the i-th smallest element in arp_list
7:     if workload_list[idx].running == False then
8:       if workload_list[idx].op_type == fu_type then
9:         return workload_list[idx]
10:     end if
11:   end if
12: return NO_WORKLOAD_AVAILABLE
13: end function
```

**Table 1: Average operator lengths of DNN models.** The batch size is 32 except for ShapeMask (8) and Mask-RCNN (16).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT</td>
<td>8.77 × 10^6</td>
<td>3.47 × 10^6</td>
</tr>
<tr>
<td>DLRM</td>
<td>1.70 × 10^6</td>
<td>4.43 × 10^6</td>
</tr>
<tr>
<td>EfficientNet</td>
<td>1.05 × 10^6</td>
<td>6.90 × 10^6</td>
</tr>
<tr>
<td>Mask-RCNN</td>
<td>1.38 × 10^6</td>
<td>1.46 × 10^6</td>
</tr>
<tr>
<td>MNIST</td>
<td>1.80 × 10^6</td>
<td>2.62 × 10^6</td>
</tr>
<tr>
<td>NCF</td>
<td>4.30 × 10^6</td>
<td>1.71 × 10^6</td>
</tr>
<tr>
<td>ResNet</td>
<td>1.54 × 10^6</td>
<td>1.28 × 10^6</td>
</tr>
<tr>
<td>ResNet-RS</td>
<td>3.20 × 10^6</td>
<td>6.19 × 10^6</td>
</tr>
<tr>
<td>RetinaNet</td>
<td>1.57 × 10^6</td>
<td>4.68 × 10^6</td>
</tr>
<tr>
<td>ShapeMask</td>
<td>1.91 × 10^6</td>
<td>2.62 × 10^6</td>
</tr>
<tr>
<td>Transformer</td>
<td>6.65 × 10^6</td>
<td>5.54 × 10^6</td>
</tr>
</tbody>
</table>

**Figure 12: Operator scheduling with or without preemption.**

active_rate_p = \frac{active_rate}{priority} the same for all workloads. Thus, the workload with the lowest active_rate_p suffers from the largest throughput degradation compared to other workloads and should be scheduled and executed first. Specifically, the scheduler will prioritize the workload that scores the lowest active_rate_p, as specified in Algorithm 1, where act_list, tot_list, priority_list, and arp_list are the lists of active_time, total_time, priority, and active_rate, respectively, for all workloads. With this scheduling policy, V10 dynamically controls the resource allocation to each workload.

**3.3 Tensor Operator Preemption**

Although the scheduling policy attempts to improve utilization and maintain fairness, it will not work well if operator lengths are imbalanced. As shown in Figure 12a, Workload 1 and Workload 2 have complementary SA and VU utilizations. Thus, we may expect them to collocate well without severe resource contention. However, in Figure 12b, we still observe surprisingly low utilization and severe unfairness. The root cause is that a long SA operator in Workload 1 blocks a short SA operator that is a dependency of a future VU operator. This degrades the performance of Workload 2 and prevents efficient overlapping of SA and VU operators. This issue happens frequently, since different workloads can have vastly different operator lengths, as shown in Table 1.

We propose a low-overhead operator preemption mechanism. In Figure 12c, by preempting long SA operators of Workload 1 and executing short SA operators of Workload 2, the dependencies for
Figure 13: SA operator preemption and restoration procedure for an example 3×3 SA (left) and the corresponding timeline for an example 128×128 SA (right). The cycle number indicates when a column/row of data is pushed/popped. An output element labeled x' is valid only after all input elements labeled x have been pushed into the SA.

VU operators in Workload 2 can finish sooner, which increases the chance of overlapping SA and VU execution.

**Preempting a VU Operator.** Since the VU contains no intermediate states, to preempt a VU operator, we pause its execution and save the PC and register values into the on-chip vector memory. Later, to resume the operator, we restore the register values and continue execution from the saved PC.

**Preempting an SA Operator.** Preempting an SA operator is more complicated because the SA contains intermediate states, including inputs, weights, and partial sums, which are shared by consecutive push/pop instructions. Since the PC and register saving mechanism for VU cannot preserve intermediate data left inside the SA, we need special context saving and restoration mechanisms to maintain the intermediate states for the preempted SA operator.

A naïve solution is to drain all intermediate data from the SA and save it into the on-chip vector memory for later restoration. However, this requires significant hardware changes to the SA for manipulating the registers in the PEs directly. It also incurs significant context storage overhead, as we must save 2×128×128×2B inputs and weights and 128×128×4B partial sums (128KB per SA).

Our first insight is that we can minimize hardware modification by saving the inputs before they are pushed into the SA, instead of reading them out from the SA itself. However, some previous inputs are already pushed into the SA before invoking the preemption. Thus, we continue execution until all computations related to them complete. Meanwhile, we save any new inputs to the vector memory when they are being pushed into the SA. On restoration, we simply recover the state of SA by replaying the saved inputs.

Our second insight is that we can minimize context storage overhead by storing 2-byte inputs instead of 4-byte partial sums^2, which can be later recovered using the saved inputs. Thus, we only save 128×256×2B inputs and 128×128×2B weights (96KB per SA), which is 25% less than the naïve approach.

^2Most systolic array designs use 2-byte float16 inputs and weights, and 4-byte float32 partial sums for better accumulation accuracy [27].

Figure 14: Training and inference procedures for the clustering-based workload collocation mechanism.

We use a 3×3 SA in Figure 13 to demonstrate our SA context switch mechanism. The preemption is invoked at cycle 1 (1). Instead of pausing the execution immediately, we keep the SA running while saving all further inputs into the vector memory (2). No cycles are wasted so far since the SA is still popping valid outputs. After all partial sums depending on earlier inputs are popped, we pause the execution (3). Then, we save the weight data of the preempted operator from the SA, and simultaneously start the restoration of the next operator by loading its weight data into the SA (4). The preempted operator exits completely at cycle 9 (5). Then, the next operator continues restoring its context by replaying its inputs to the SA, and then resumes normal execution (6).

With our approach, 128 cycles are spent for preemption, which is overlapped with 384 cycles for reinitialization. Thus, one context-switch for a 128×128 SA costs 384 cycles in total, which is negligible compared to the average SA operator length (Table 1). The storage overhead of SA preemption is 96 KB per SA per workload context, which is trivial compared to the vector memory capacity.

### 3.4 Clustering-based Workload Collocation

Although we have enabled workload collocation on an NPU core, randomly collocating two arbitrary workloads may negatively impact resource utilization if they have conflicting resource demands. For example, two SA-intensive workloads on one NPU core will keep contending for the SA while leaving the VU idle no matter how we schedule the operators. To prevent this scenario, we need a lightweight collocation mechanism to accurately identify workloads with compatible resource demands.

A simple heuristic-based mechanism is that the aggregated resource utilization of collocated workloads should not exceed the total available resource. However, this is inaccurate because it ignores dynamic resource contents (such as operator length mismatch). Another approach is to brute-force profile the collocation performance of two workloads, and use the profiled result to decide if they should be collocated. However, this is unrealistic due to significant profiling overhead despite being accurate.

To achieve the benefits of both approaches, our key insight is that DNN workloads with similar resource utilization patterns usually have similar collocation performance, so we can classify workloads into groups according to those patterns. Then, we can predict the compatibility between given workloads using the profiled compatibility between their groups. Thus, we can make more accurate
collocation decisions than the heuristic-based approach without suffering significant profiling overhead.

Based on this insight, we propose a clustering-based collocation mechanism to predict the collocation performance of two given workloads. As shown in Figure 14, it consists of an offline training phase and an online inference phase. During offline training, we build a database of clusters, where each cluster consists of similar workloads, as shown in Figure 15. Specifically, we leverage compiler techniques or offline profiling to extract workload features related to resource contentions, including SA/VU utilizations, HBM bandwidth consumption, and operator length statistics (e.g., mean, min, max). With these features, we apply principal component analysis (PCA) to extract important features, and then use K-Means [18] to classify the workloads into different clusters. Then, we perform offline profiling to obtain the pair-wise collocation performance between workloads from different clusters, and we represent the collocation performance of two clusters using the average collocation performance for all workload pairs across the two clusters.

For online inference, we decide whether to collocate two given workloads by first identifying which clusters they belong to, according to their resource utilization features. Then, we predict their collocation performance as the collocation performance of the representative clusters. We dispatch them to the same NPU core if the prediction is higher than a threshold, or to different cores otherwise.

To evaluate our clustering-based collocation mechanism, we compare three collocation schemes: (1) Random (randomly collocates two workloads), (2) Heuristic, and (3) Clustering. To prove V10 can make efficient predictions for DNN workloads unseen during training, we follow the classical cross-validation approach used in ML community [3]. We select 9 workloads from Table 4 as the training set to build the prediction model, and use the remaining 2 workloads as the testing set to evaluate the accuracy. The process is repeated for all possible combinations. As shown in Table 2, our clustering mechanism achieves an accuracy of 84.73% and prevents 73.19% of the non-beneficial collocations.

### 3.5 Put It All Together

In cloud platforms, incoming tasks are usually queued in a workload pool. After the pre-deployment compilation, the workloads will not change unless their model architectures are updated. V10 leverages such predictability of DNN inference workloads to perform offline profiling and make online collocation decisions.

Before serving ML inference services, V10 trains the clustering model offline. At runtime, V10 leverages the pre-built clustering model to identify groups of workloads with complementary resource demands ($\S 3.4$), and dispatches each group to each NPU core to maximize the potential of overlapped execution. The cloud provider can also specify different priorities for the workloads to guarantee SLAs. In each NPU core, the operator scheduler overlaps the execution of multiple operators from different workloads to maximize resource utilization. Periodically, the priority-based scheduling policy is invoked to examine whether all workloads get their fair share of resources with respect to their priorities ($\S 3.2$). The operator scheduler will preempt operators that run longer than they deserve and execute starved operators ($\S 3.3$).

### 3.6 V10 Implementation

#### V10 Framework

We implemented V10 with an NPU simulator based on public literature on Google TPUs [27, 37]. The simulator replays instruction traces captured on real TPUs and simulates operator scheduling and dispatching, SA/VU execution, as well as vector memory and HBM bandwidth. We show the detailed NPU configuration in Table 5, which represents the state-of-the-art NPU architecture. The clustering algorithm is implemented in Python 3.6 with numpy and scikit-learn, and it takes one millisecond on an Intel Xeon E5-2687W v4 for each prediction. We also prototyped V10’s core components in Verilog and synthesized the design in Cadence Virtuoso using the FreePDK-15nm standard cell library [1].

#### Memory Management

V10 employs a simple yet effective memory management scheme to reduce hardware complexity and runtime overhead. For vector memory, V10 partitions the address space evenly among collocated workloads and adds the partition offset on each memory access at runtime. Thus, operators in the same workload can share data in vector memory without interfering with collocated workloads. We evaluate the impact of various vector memory capacities in §5.8. For HBM, V10 uses the conventional segmentation scheme to divide the address space into several memory regions to host one workload per region. The region size depends on the workload memory allocation (e.g., batch size and model size). Thus, V10 incurs negligible address translation overhead.

### Table 2: Prediction accuracy and worst-case performance of different collocation schemes.

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
<th>True Positive</th>
<th>False Positive</th>
<th>True Negative</th>
<th>False Negative</th>
<th>Worst Perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>44.83%</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>0.965x</td>
</tr>
<tr>
<td>Heuristic</td>
<td>64.91%</td>
<td>95.21%</td>
<td>42.36%</td>
<td>57.64%</td>
<td>6.79%</td>
<td>0.992x</td>
</tr>
<tr>
<td>Clustering</td>
<td>84.73%</td>
<td>98.74%</td>
<td>73.19%</td>
<td>26.81%</td>
<td>1.26%</td>
<td>1.000x</td>
</tr>
</tbody>
</table>
V10 Overhead. The major hardware overhead of V10 comes from the tensor operator scheduler. We quantify the overhead by configuring the number of SAs/VUs and the number of collocated workloads. In Table 3, V10 incurs negligible area and power overhead compared to a Google TPUv3 chip. The scheduler latency is also negligible compared to the operator lengths (most are $\geq 10\mu s$ in Table 1), so it will not block the execution of already scheduled operators. Since an NPU core has only a limited number of SAs/VUs, and the number of workloads does not need to exceed twice the number of SAs/VUs for optimal utilization (Figure 25), the scheduler complexity is manageable in each NPU core.

4 DISCUSSION

Support for various scheduling policies. V10 developed a new lightweight context-switching mechanism with minimal hardware modifications (§3.3). With the priority-based operator scheduling algorithm, V10 provides the flexibility for supporting various scheduling policies based on the demands of cloud providers. For example, V10 enables equal- or fixed-share scheduler by setting different priorities for the workloads. V10 also enables collocating low-priority best-effort workloads with high-priority latency-sensitive ones to improve resource utilization without violating SLO requirements.

Alternative software-based approach. Though we can implement the operator scheduler in host runtime, the software-based solution has three limitations. First, the software scheduling overhead is too large (about $20\mu s$ to schedule 2 operators from 2 workloads on 1 SA and 1 VU) for most operators (Table 1). Second, for fine-grained scheduling, the frequent communications between host software and NPU hardware incur significant overhead. Third, even if the scheduler runs in software, new hardware support for simultaneous operator execution is still required. In contrast, implementing the entire scheduler in hardware has negligible overhead (Table 3). As commodity GPUs also employ hardware schedulers [2], we believe V10’s hardware-assisted design for NPU multi-tenancy is feasible.

Generalizability of V10 approach. Our observations in §2.2 apply to all DNN accelerators that consist of SAs and VUs. For these accelerators, we can improve the performance by pipelining SA computations and VU post-processing. However, the SA/VU imbalance still exists because the VUs are designed not only to post-process the SA outputs but also to execute generic vector operations that cannot run on the SAs. This allows the NPU to support more operators in diverse DNN models. Therefore, the main source of SA/VU imbalance comes from operators that can only run on VU. V10 addresses this problem with a hardware-assisted approach.

5 EVALUATION

Compared with state-of-the-art NPU multi-tenancy designs, V10
• improves the overall NPU utilization by $1.64\times$ (§5.2);
• increases the aggregated throughput by $1.57\times$ (§5.3);
• reduces the average latency of ML services by $1.56\times$, and tail latency by $1.74\times$ (§5.4);
• incurs minimal operator preemption overhead (§5.5);
• supports flexible workload priority settings (§5.6);
• works with various scheduler time slice settings (§5.7);
• can benefit different vector memory capacities (§5.8);
• scales with multiple workloads and SAs/VUs (§5.9).

5.1 Experimental Setup

From MLPerf v2.1 benchmarks [44] and Google’s official TPU reference models [5], we choose 11 ML models representing common MLaaS workloads from various domains (Table 4). We collocate two ML workloads in each experiment based on our clustering (§3.4), which identifies workloads with best-matching resource requirements. To measure the steady-state multi-tenant performance, we iteratively run inference requests for each collocated workload until all workloads complete a certain number of requests.

We compare V10 with the state-of-the-art NPU multi-tenancy solution, i.e., preemptive time-sharing of NPUs [16]. We also evaluate the effectiveness of the priority-based operator scheduling policy and the operator preemption mechanism in V10. To summarize, we compare the following designs:
• PMT: the baseline preemptive multi-tasking NPU, which supports time-sharing of an NPU core without simultaneous operator execution. It preempts a workload at the ML inference task level with $20\mu s$–$40\mu s$ context switch overhead.
• V10-Base: the basic V10 with simultaneous operator execution and non-preemptive round-robin operator scheduling.
• V10-Fair: this V10 variant uses the priority-based scheduling policy. We set equal priority for all workloads by default.
• V10-Full: the full V10 design with all proposed design components. It enables operator preemption over V10-Fair.

5.2 Resource Utilization Improvement

Figure 16 shows the NPU utilization with two collocated workloads. On average, the final design V10-Full improves the aggregated utilization of all compute units on NPU core by $1.64\times$ over PMT. The utilization improvement of SA, VU, and HBM bandwidth over PMT are $1.63\times$, $1.65\times$, and $1.47\times$, respectively.

The baseline PMT ensures fairness by assigning each workload an equal amount of NPU core time without overlapped execution of SA and VU. Thus, the aggregated utilization of PMT is the average, instead of the sum, of each collocated workload’s single-tenant utilization (see §2.2). For example, collocating the SA-intensive BERT

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbrev.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT</td>
<td>BERT</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>DLRM</td>
<td>DLRM</td>
<td>Recommendation</td>
</tr>
<tr>
<td>EfficientNet</td>
<td>ENet</td>
<td>Image Classification</td>
</tr>
<tr>
<td>Mask-RCNN</td>
<td>MRCN</td>
<td>Object Detection &amp; Segmentation</td>
</tr>
<tr>
<td>MNIST</td>
<td>MNIST</td>
<td>Image Classification</td>
</tr>
<tr>
<td>NCF</td>
<td>NCF</td>
<td>Recommendation</td>
</tr>
<tr>
<td>ResNet</td>
<td>RNN</td>
<td>Image Classification</td>
</tr>
<tr>
<td>ResNet/R5</td>
<td>RNNR5</td>
<td>Image Classification</td>
</tr>
<tr>
<td>RetinaNet</td>
<td>RNN</td>
<td>Object Detection</td>
</tr>
<tr>
<td>ShapeMask</td>
<td>SMask</td>
<td>Object Detection &amp; Segmentation</td>
</tr>
<tr>
<td>Transformer</td>
<td>TFM</td>
<td>Natural Language Processing</td>
</tr>
</tbody>
</table>

Table 4: ML models used in our evaluation. Batch size is 32 except for ShapeMask (8) and Mask-RCNN (16).

<table>
<thead>
<tr>
<th>Name</th>
<th>Systolic array (SA) dimension</th>
<th>Vector unit (VU) dimension</th>
<th>Frequency</th>
<th>Vector Memory</th>
<th>HBM Memory Size &amp; Bandwidth</th>
<th>Scheduler Time Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$128\times128$</td>
<td>$8\times128\times2$ FP32 operations/cycle</td>
<td>700MHz</td>
<td>32MB</td>
<td>32GB, 390GB/s</td>
<td>32768 cycles ($\approx 46\mu s$)</td>
</tr>
</tbody>
</table>

Table 5: The configurations of the NPU simulator.
Following previous literature [16, 20], we define the system throughput as the sum of the normalized forward progress of each collocated workload. On average, V10-Base and V10-Fair improve the throughput by 1.25x over PMT, and V10-Full further achieves 1.57x improvement. The throughput improvement of V10 directly reflects the amount of overlapping operator execution time in Figure 17. With preemption, V10-Full enables flexible operator scheduling by balancing operator lengths. This leads to overlapped execution of SA and VU for up to 81% (63% on average) of the time.

The clustering scheme also contributes to the throughput improvement, since less resource contention leads to more execution overlapping between operators from different workloads. Even with slightly imperfect resource compatibility (e.g., RNRS+MRCN), V10 still achieves 1.33x higher throughput than PMT.

### 5.4 Latency of ML Inference Requests

Figures 19 and 20 show the average and tail latency for the ML inference requests in each workload. On average, V10-Full reduces the average and tail latency by 1.56x and 1.74x over PMT, respectively. By utilizing more hardware resource than PMT, V10 allows both collocated workloads to make forward progress at the same time, which greatly improves the average and tail latency.
We also observe that for both collocated workloads, operator preemption is necessary for better latencies than PMT. In Figures 19 and 20, V10-Base causes up to 4.3× degradation on latency for one of the collocated workloads. Despite its improvement over V10-Base, V10-Fair is still not fair due to operator size contentions (e.g., BERT+DLRM). With operator preemption, V10-Full enforces fairness by preventing long operators from blocking short ones and achieves better latencies than PMT for both collocated workloads.

The preemption overhead of V10 is negligible (see §5.5), so the tail latency of workloads is not affected significantly by preemptions. Since resource contention between collocated workloads is a major reason for high tail latency, as shown in Figure 20, operator preemption reduces the latency by alleviating the contention.

5.5 Overhead of Preempting Tensor Operators

To evaluate V10’s operator preemption mechanism, we profile the context switch overhead and the number of preemptions per inference request for both V10 and PMT. As shown in Figure 21, the context switch overheads for both designs are negligible (less than 2%). However, with similar overhead, V10 makes significantly more preemptions than PMT with our lightweight operator preemption mechanism. This means V10 shares the hardware resource at a much finer granularity and allows more flexible workload scheduling. In contrast, PMT can only preempt at a coarse granularity for amortizing the high context switch overhead. Overall, the benefit of V10’s finer-grained scheduling enabled by more frequent preemptions clearly out-weights the negligible preemption overhead.

5.6 Impact of Varying Workload Priorities

The cloud platform provider may configure the priority of each user application to guarantee different levels of SLAs. V10 can sustain the performance for prioritized workloads while improving resource utilization. For simplicity, we refer to the relative priorities of two collocated workloads that sum to 100% [20].

Figure 22 shows the overall throughput of collocated workloads as we vary the priorities. By assigning time slices proportionally to each workload’s priority, PMT can only scale the performance of workloads proportionally to their relative priorities. Despite being fair, PMT cannot achieve better relative performance than a workload’s relative priority level.

With V10 utilizing more hardware resources, a workload can perform better than with PMT under the same priority. V10 also allows low-priority workloads to harvest remaining resources and improve the aggregated throughput. In cases such as MNST+NCF, prioritizing one workload leads to even higher aggregated throughput than assigning equal priority, as prioritizing workloads with shorter operators creates less resource contention. V10 fails to sustain performance for the prioritized workload in only DLRM+RsNt, which oversubscribes HBM bandwidth.

5.7 Impact of Varying Scheduler Time Slices

The scheduler time slice of the operator scheduler decides how frequently operators are preempted. Small time slices enable fine-grained scheduling to improve utilization and fairness but increase preemption overhead. Large time slices reduce preemption overhead but cause more head-of-line blocking on contending operators. By studying various scheduler time slices in Figure 23, we observe that a 32768-cycle time slice ($\approx$46$\mu$s) achieves an optimized throughput by balancing preemption overhead and scheduling granularity.
As shown in Figure 24, V10 outperforms PMT regardless of vector memory capacity. As V10 partitions the capacity evenly among collocated workloads, large operators are partitioned by the compiler to fit into smaller vector memory, which may slightly increase HBM bandwidth consumption due to lower data reuse. However, most inference workloads are not affected, since they underutilize HBM bandwidth even with smaller vector memory. In addition, vector memory bandwidth contention never occurs as vector memory is designed to satisfy the peak bandwidth from both SA and VU.

5.8 Impact of Varying Vector Memory Capacity

As we scale the NPU, we pick workloads randomly from the 11 DNN models listed in Table 4, increase the number of collocated workloads, and evaluate the throughput of V10 over single-tenant execution. V10 scales easily by having more NPU cores (each has one SA and one VU), since each core runs independently and can host two or more collocated workloads grouped by our clustering mechanism. Alternatively, we scale V10 by increasing the number of SAs/VUs in each NPU core and map all the workloads to these SAs/VUs in the shared NPU. As a common practice, NPU hardware designers scale the HBM bandwidth with the increasing number of SAs/VUs to balance compute and memory. As shown in Figure 25, V10 improves the throughput linearly until the number of workloads reaches the number of FUs. V10 dispatches operators from different workloads to keep all the FUs busy. With more FUs and collocated workloads, V10 tolerates more contentions, as it has a higher chance to identify operators that demand different FUs.

6 RELATED WORK

**Accelerator support for multi-tenancy.** Accelerator-based cloud [8, 9, 45, 52] motivates many studies on supporting multi-tenancy with hardware accelerators [10–12, 16, 29, 30, 34, 53, 54]. For instance, PREMA [16] developed a preemptive-based multi-tasking scheme. Upon preemption, it context-switches the entire NPU core and stores a large context in the off-chip HBM. In contrast, V10 enables fine-grained preemption of each SA/VU and manages a negligible amount of context in the on-chip SRAM. AI-MT [10] and Layerweaver [39] addressed the imbalance of compute and memory by employing DNN multi-tasking on NPUs. NB-SMT [46] exploited the resiliency of DNNs to avoid task blocking. Planaria [22] exploited the spatial underutilization caused by SA padding. However, it cannot address VU idleness. Orthogonal to Planaria, V10 spatially partitions the NPU core into individual SAs/VUs and temporally partitions each of them among different workloads.

**Hardware accelerators for ML workloads.** Hardware accelerator developments attract much attention in both industry and academic communities [24, 28]. Notable examples of systolic array architectures include Google TPU [37], Graphcore IPU [24], NVIDIA T4 [38], and Microsoft Brainwave [21]. Many studies on the NPU architecture are conducted in academia [13, 15, 19, 26, 41, 43, 51]. However, they mostly focused on high performance, and few investigated hardware utilization at the microarchitectural level. Our work V10 improves the NPU utilization for multi-tenancy by enabling simultaneous execution across compute units in NPUs.

**Accelerator resource management.** Researchers have proposed various techniques to improve the resource utilization of hardware accelerators in data centers [25, 33, 36, 42, 50, 55]. However, most studies focused on the system-level task scheduling, but cannot fully utilize the compute units in a single accelerator device, even with compiler optimizations [6]. As we scale out the deployment of cloud NPUs, it is desirable to have architecture-level resource management. V10 aims for such purpose by enabling fine-grained operator-level scheduling in hardware.

7 CONCLUSION

We conduct a thorough study on the NPU utilization with real TPU devices and identify the imbalanced use of compute units in NPUs as the main reason for low NPU utilization. To support multi-tenant NPUs with improved resource utilization, we develop a hardware-assisted NPU multi-tenancy framework V10, which enables operator-level scheduling, flexible priority-based resource management, and a unique workload collocation mechanism for ML services. Compared with the state-of-the-art multi-tenant NPU design, V10 greatly improves NPU utilization and performance while maintaining fairness for multi-tenant ML inference services.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments and feedback. We thank Haoyang Zhang for his help with analyzing the impact of varying vector memory capacity. This work was partially supported by NSF grants CCF-1919044 and CNS-2144796.
REFERENCES


