AUTOSCRATCH: ML-OPTIMIZED CACHE MANAGEMENT FOR INFERENCE-ORIENTED GPUS

Yaosheng Fu [1] **Evgeny Bolotin** [1] **Aamer Jaleel** [1] **Gal Dalal** [1] **Shie Mannor** [1] **Jacob Subag** [1] **Noam Korem** [1] **Michael Behar** [1] **David Nellans** [1]

ABSTRACT

Taking advantage of the L2 residency control mechanism introduced with NVIDIA's Ampere GPUs, we propose a Machine Learning (ML) based framework called AutoScratch to automatically discover and optimize the L2 residency for inference-oriented GPUs, effectively removing any human involvement from the optimization loop. AutoScratch bridges the gap between the performance of an explicitly controlled scratchpad memory and the convenience of a hardware-controlled cache. We develop two versions of AutoScratch, AutoScratch-RL harnessing reinforcement learning (RL) and AutoScratch-EA leveraging a state-of-the-art evolutionary algorithm (EA). We integrate AutoScratch with NVIDIA's TensorRT framework to fully automate the optimization pipeline for arbitrary DL inference applications. We evaluate AutoScratch on NVIDIA's L4 GPU silicon using MLPerf inference workloads and show that AutoScratch reduces off-chip DRAM traffic by 29% and improves the overall performance by 9% (up to 22%).

1 INTRODUCTION

Recent GPUs have been augmented with high-throughput, low-precision matrix-multiply computing units such as NVIDIA Tensor Cores and AMD Matrix Cores that are specifically designed to improve DL training and inference (NVIDIA, 2020a; AMD, 2021). Through these technologies, GPUs have achieved significant DL performance improvement over the last several generations of devices. For example, NVIDIA's V100 GPU improved FP16 throughput by 6× compared to the previous generation, while NVIDIA's A100 GPU further increased it by 2.6× over the V100 (NVIDIA, 2017; 2020a; Fu et al., 2021). Unfortunately, GPUs' DRAM bandwidth scaling has not maintained similar growth, with the DRAM bandwidth of the V100 GPU scaling by only 25% compared to the previous generation and the bandwidth of the A100 GPU improving by 72% over the V100. If the divergence between compute and memory scaling continues, DRAM bandwidth is on track to become the biggest performance bottleneck for DL workloads on future GPUs (Fu et al., 2021).

Additionally, GPUs' DRAM power consumption is also becoming a more significant portion of their overall power and thermal envelopes. For example, the power consumption of a high-end GPU's High Bandwidth Memory (HBM) is as much as 40 watts for each 1TB/s of bandwidth and expected to reach hundreds of watts on single GPU in future generations (Chatterjee et al., 2017). The off-package bandwidth-optimized GDDR memories that are currently employed on GPUs for gaming and DL inference workloads also consume a significant portion of the total GPU power, which becomes prohibitive as their bandwidth scales beyond hundreds of GB/s (Chatterjee et al., 2017). These trends introduce significant challenges to continued performance improvements in future GPU designs – particularly for the passively cooled, low-power GPUs targeting DL inference, such as NVIDIA's T4 and L4 GPUs that have a maximum thermal design point (TDP) of just 70-72 watts (NVIDIA, 2020b; 2023c). The DRAM bandwidth and power challenges will apply not just to GPUs, but other state-of-the-art DL accelerators as well.

One commonly used technique to overcome these issues is to increase the capacity of on-chip, on-wafer, or on-package SRAM caches to reduce or eliminate DRAM bandwidth dependencies (NVIDIA, 2020a; 2023a; Fu et al., 2021; AMD, 2021; Knowles, 2021; Lie, 2019). Figure 1 shows the last level cache (LLC) capacities offered by recent generations of GPUs and DL accelerators. We see that GPUs are deploying larger LLC capacities with each generation, increasing over 12× in size from NVIDIA's P100 to H100 and from the T4 to L4. Other DL accelerators provide even more extreme levels of on-chip SRAM storage, up to 10x larger than a GPU's LLC, and in some cases completely replacing external DRAM memories.

[1]NVIDIA. Correspondence to: Yaosheng Fu <yfu@nvidia.com>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

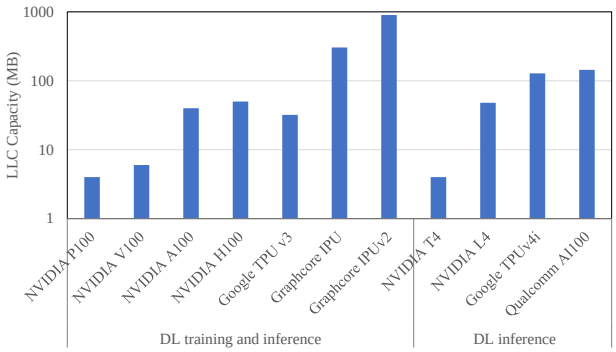


Figure 1. The last level cache capacity trends in GPUs and DL accelerators.

GPU architects have chosen to utilize easy-to-use hardware-managed SRAM-based LLCs, while most DL accelerators deploy SRAM-based scratchpad memories that are explicitly managed in software. Scratchpad memories require high programming effort, but enable precise management of precious on-chip storage resources compared to hardware-managed caches. To bridge the gap between easy to use hardware-managed caches and more conscientious control of cache resources (similar to scratchpad memories), NVIDIA's Ampere generation GPUs have recently introduced L2 cache Residency Controls (NVIDIA, 2021a; Krashinsky et al., 2020) that allow programmers to selectively specify data as being L2 cache persistent. This mechanism effectively enables explicit data pinning in the GPU's L2 and protects it from being evicted by the default hardware-managed cache replacement policy. This feature is intended to help GPUs match the efficiency of scratchpad memories for performance-critical data, while retaining the flexibility and convenience of the hardware-managed caches for less critical data.

Extensions enabling explicit control for hardware caches are attractive in theory, but it is difficult for programmers to reason about optimal L2 data residency selections. They must determine which data should be labeled as L2-resident in a DL application that can consist of hundreds to thousands of GPU kernels with complex data interaction and reuse patterns within the span of a single iteration. Moreover, these extensions can lead to adverse performance effects when data with low importance is misclassified as L2-resident, which reduces the capacity of the remaining hardware-managed portion of the cache. As a result, L2 residency controls have not been widely adopted yet by applications running on NVIDIA GPUs.

To address this challenge, we propose a machine learning (ML) based framework called AutoScratch that automatically discovers and optimizes L2 cache residency control configurations for DL applications, effectively removing programmer intuition from the optimization loop. We show that AutoScratch can be successfully trained to select a subset of the application's memory to be L2-resident (like a scratchpad), that maximizes the performance of a GPU. We develop and perform a detailed analysis of two possible implementations of AutoScratch's optimization component: (i) AutoScratch-RL – harnessing the power of Reinforcement Learning (RL) (Sutton & Barto, 2018), and (ii) AutoScratch-EA – leveraging a variant of the recently proposed regularized evolutionary algorithm (Real et al., 2019). We use the AutoScratch framework to accelerate the execution of DL inference applications, but our design is generic and could be applied to DL training and other iterative GPU workloads with similar dataflow regularity.

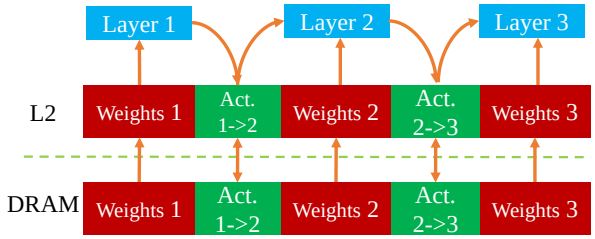
To fully automate the optimization process, we modified NVIDIA's TensorRT SDK (NVIDIA, 2021b) and exposed the L2 residency control application programming interface (API) so that the L2 residency configurations can be easily applied to any TensorRT optimized inference application. This integration enables AutoScratch training and optimization without any code modifications in the inference application itself. Post training, the learned L2 residency configurations can be directly deployed through the same APIs with near-zero overhead.

In this work, we make the following contributions:

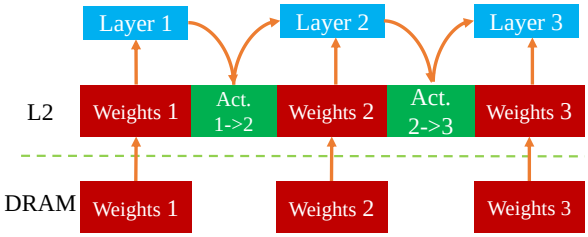
- We design AutoScratch that relies on ML to automatically generate optimized L2 residency configurations for GPU applications by leveraging the L2 Residency Controls APIs introduced in latest NVIDIA GPUs. We design and evaluate two types of ML techniques within AutoScratch, (i) reinforcement learning, and (ii) evolutionary algorithms.
- We develop a practical AutoScratch implementation for DL inference by integrating it with NVIDIA's TensorRT SDK, so that any workload optimized by TensorRT can automatically leverage AutoScratch to discover and deploy learned L2 residency configuration.
- We evaluate AutoScratch using the MLPerf inference suite on the state-of-the-art inference-optimized NVIDIA's L4 GPU silicon and show that it achieves a geomean 29% reduction in DRAM traffic and 9% performance improvement (up to 22%). We also compare AutoScratch to both random search and a human-designed heuristic and demonstrate its advantages.

2 Motivation and Background

DL inference applications are the backbone propelling the highly profitable personalized business models of modern cloud service providers. They are executed at a massive scale within the datacenters of internet giants such as



(a) Typically both activations and weights end up shuffling between L2 cache and DRAM in hardware-managed caches resulting in cache interference.



(b) Ideal reuse of L2 cache capacity results in cache resident activations, with only weights being fetched from DRAM, which are not shared between layers.

Figure 2. The memory access pattern for common DL inference workloads illustrating differing reuse patterns for weights and activations.

Google, Meta, Amazon, and Netflix (Underwood, 2020; Park et al., 2018; Jouppi et al., 2021; Anderson et al., 2021). Inference workloads serve billions of users and are available 24 hours a day. Some inference models are more important than others, i.e. recommendation models comprise up to 79% of total DL inference cycles in Meta data centers (Gupta et al., 2020). Moreover, Google has reported that the total cost of DL inference ownership is dominated not by the cost of the underlying hardware, but the power dissipated while executing DL inference workloads in their servers, over the hardware's lifetime (Jouppi et al., 2021). Due to the importance of providing best in class DRAM bandwidth utilization on power-constrained DL inference applications, this paper focuses on demonstrating how AutoScratch can be used to optimize DL inference workloads. Prior work such as vDNN(Rhu et al., 2016) discovered that data movement can be intelligently managed for DL workloads thanks to the regularity of their dataflow patterns. This work builds upon that observation by using machine learning to automate the data management process so that programmers do not have to manage data locality explicitly or build per-application solutions repeatedly.

The memory access pattern of a typical DL inference application is shown in Figure 2. There are two main types of accesses: weights and activations. Weights are the inputs fetched by each layer and different layers typically each

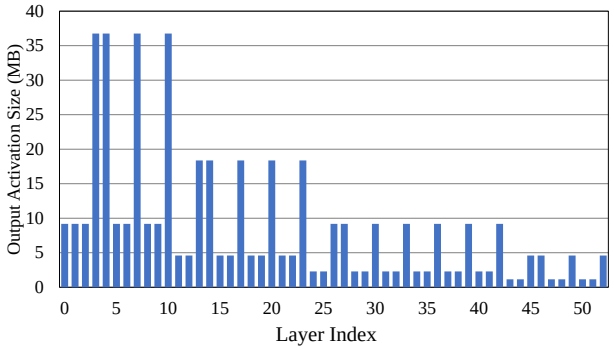


Figure 3. The per-layer activation sizes for int8 datatype in *resnet50* inference, with a batch size of 48. The largest per-layer activation size is less than 37MB.

have their own set of weights. Consequently, there is limited opportunity to reuse weights across network layers within one iteration. The memory footprint of each layer's weights is fixed and independent of the batch size.

Activations are the intermediate data generated by one network layer and passed into the next layer. There is an explicit producer-consumer relationship between adjacent network layers with respect to their activation data. Unlike weights, activations are generated on the fly during each inference iteration and become dead data after being consumed. The sizes of the activations passed between network layers are proportional to the inference batch size. Caching activation data is beneficial in reducing the overall DRAM traffic due to the temporal locality introduced by activation reuse among the neighboring layers. In an ideal case, the on-chip cache capacity would be sufficient to fit all activation data, resulting in zero activation-related memory traffic, as shown in Figure 2(b).

To illustrate the cache capacity needed to store inference activations on-chip, Figure 3 shows the distribution of the per-layer activation sizes for *resnet50*. In this example, all activations are of int8 precision using a batch size of 48. Figure 3 shows that although the aggregate activation size of 457MB is much larger than the available GPU L2 capacity (48MB in NVIDIA's L4 GPUs), the maximum per-layer size is smaller than 37MB and less than 10MB for a majority of layers.

In *resnet50*, some activations are also reused beyond just the neighbor layer, when skip connections are incurred, which is not shown in Figure 2. Therefore, for any network layer in *resnet50* at most three types of activations need to be stored in the memory system: the input, the output, and the skip connection. As a result, at any given time within an inference iteration, a maximum of 111MB of cache is required to store all the live activations on-chip,

however just 30MB is sufficient to cache the live activations for most layers as the majority of them are within 10MB each. Carefully orchestrated caching has the potential to maximize activation on-chip data reuse and prevent spilling into DRAM whenever possible, but *simply prioritizing all activations equally is not sufficient*.

Because activation data is only temporarily alive, a commonly used technique for logical management of memory is to allocate a single shared activation memory buffer capable of storing all live activations in the network (NVIDIA, 2021b). This buffer is then recycled for activations across layers, such that new activations directly overwrite old activations that have been consumed during inference. Because two consecutive activations often serve as input and output of a layer and need to be alive "at the same time", the activation buffer size is typically 2x the large activation size. Because the activation buffer is a linear array (reused through time), the L2 residency control problem for DL inference can be formulated as selecting a subset of the memory addresses from the shared activation buffer to be L2-resident, as shown in Figure 4. The L2 cache residency controls can theoretically change throughout the execution, but we decided to keep them static within one inference iteration to minimize the modification overheads. Generally speaking, an application's entire virtual memory heap space could be considered similar to a linear reuse array in other iterative GPU workloads, however in this work we focus on the more limited inference shared activation buffer to prove AutoScratch's viability.

To manage this buffer at a fine granularity, we propose to divide the shared buffer into n memory slices of equal size that can be as small as the size of a single L2 cacheline or at a larger granularity to reduce the selection problem complexity. Each slice is then either promoted to become L2-resident or marked as non L2-resident and managed by hardware replacement policy, which is the default configuration of the cache. Assuming a shared buffer size of 100MB with 1MB memory slices and the maximum L2-resident cache capacity of 30MB, an exhaustive search for an optimal L2 residency configuration would have to sift through $C(100, 30) \approx 10^{25}$ combinations, making a brute-force search intractable. We leverage reinforcement learning and evolutionary algorithms to help us automatically learn an optimized L2 residency configuration.

3 AutoScratch Framework

We developed AutoScratch to automatically discover and optimize GPU's L2 residency configurations. We describe its architecture along with two different optimizers based on RL and EA respectively.

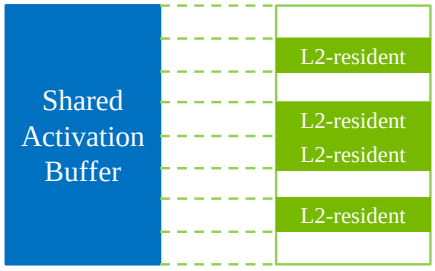


Figure 4. L2 residency selections within the shared activation buffer.

3.1 The AutoScratch Software Architecture

Figure 5 demonstrates the high-level architecture of AutoScratch. First, an unoptimized inference application is fed into NVIDIA's TensorRT library for compilation, optimization, and instrumentation to become AutoScratch-compatible by inserting L2 residency control APIs for the activation buffer. Then AutoScratch performs a machine learning based optimization on an AutoScratch-compatible and a TensorRT-optimized DL inference application. The AutoScratch ML-based optimization phase in Figure 5 encapsulates an ML process (described in Sections 3.2 and 3.3) resulting in an optimized L2 residency configuration. By integrating AutoScratch as part of the TensorRT optimization process, we eliminate the need for manual code modifications on the target DL application which greatly eases the use of AutoScratch.

In our implementation, the application's activation buffer address range is divided uniformly into n slices and each slice is represented by a bit in the n-dimensional L2 configuration vector, or mathematically $\{0, 1\}^n$. This vector is mapped to the address range allocated for the shared activation memory buffer as described in Section 2. The size of a memory slice could be as small as the size of an L2 cache line but we found that with a maximum L2-resident capacity of 36MB (a constraint of an NVIDIA's L4 GPU), a coarser grain of 1MB is sufficient for capturing most of the AutoScratch benefits without dramatically bloating the size of the search space. We use the following encoding: "1" indicates that the memory slice is L2-resident and cannot be evicted from the cache by hardware and "0" means that the corresponding memory slice is not L2-resident, and thus subject to the default hardware cache replacement mechanism.

The optimization loop starts from an initial state in which L2 residency is disabled, which is the default configuration on L4 GPUs. The optimization is performed through iterative execution of an inference application directly on GPU silicon with different L2 residency configurations and observing the application's performance. We focus on a single inference iteration as opposed to executing the entire end-to-end inference application as this is sufficient to learn

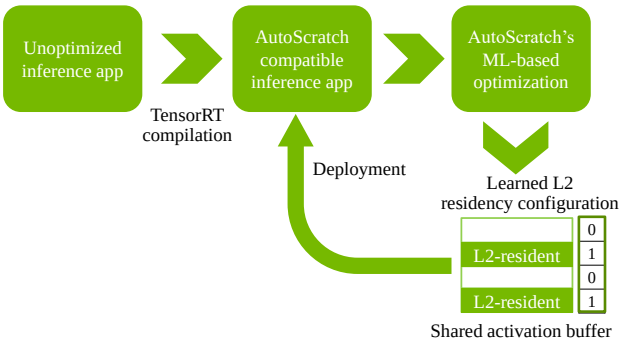


Figure 5. The high-level architecture and optimization flow of AutoScratch.

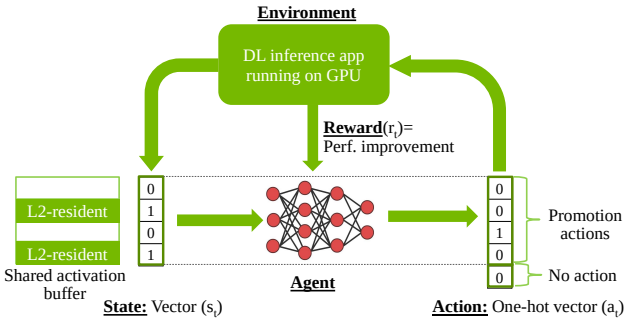


Figure 6. The AutoScratch-RL optimization framework for tuning GPU's L2 residency configuration.

the application behavior and the optimized L2 residency configurations, without making the optimization process unnecessarily long.

Upon completion of the optimization, regardless of the optimization technique (RL or EA) that was used, the best L2 residency configuration is then applied to the application for deployment in production. The overall end-to-end optimization process contains thousands to millions of iterations that can take from several seconds to several hours depending on application execution time, the optimization method, and the size of the optimization space. However, once the final L2 residency configuration is learned, integrating it into the DL inference application incurs only a trivial one-time initial setup (annotating memory allocations with the L2 residency configuration). This process is analogous to a profile-guided optimization (PGO) loop between the DL inference application and the underlying hardware (Pettis & Hansen, 1990). Note that AutoScratch can also be deployed *offline* – before the DL application is released to production as described here, or *online* – effectively tuning the application on the fly from iteration to iteration in parallel to the execution of the DL application in a production environment.

We implement and evaluate AutoScratch with two types of ML optimization algorithms: (i) AutoScratch-RL which is reinforcement learning (RL) based and (ii) AutoScratch-EA that is based on the recently proposed regularized evolutionary algorithm for image classifier architecture (Real et al., 2019). Both techniques are described in greater detail below.

3.2 AutoScratch with an RL Optimizer

In AutoScratch-RL, we harness the power of reinforcement learning (RL) as one possible implementation of the AutoScratch's ML-based optimization stage shown in Figure 5. Reinforcement learning (Puterman, 2014) is a field of machine learning in which an agent learns to maximize a utility function, often named the *reward*, by continually interacting with a simulated or real environment. One iteration of the RL loop is called an RL step and multiple RL steps forming a full interaction process with the environment comprise an RL episode.

It is rare for an agent to be able to train in a real-world environment on which it will be eventually deployed (Dulac-Arnold et al., 2019). Nevertheless, we manage to train and deploy AutoScratch-RL on a real GPU in this work. The ability to train the RL algorithm used by AutoScratch directly on the test environment (GPU silicon) is significant and is termed *on-policy*. When using on-policy algorithms, we are not confined to the restrictions imposed by the alternative *off-policy* scenarios that can significantly hinder training performance (Jiang & Li, 2016). To find the optimal L2 residency configuration in the AutoScratch framework, we leverage the Proximal Policy Optimization (PPO) (Schulman et al., 2017) algorithm, which is an on-policy algorithm. Our choice of PPO is not only because it is on-policy, but also because of its built-in isolation between two stages: (i) episode collection and (ii) agent update. This separation prevents interferences between the execution of the DL inference network in the RL environment and the RL agent training on the GPU. It results in a more accurate reward signal being propagated from an unobstructed GPU inference workload to the RL agent update.

We develop and train an AutoScratch-RL agent to find the best L2 residency configuration for activations that maximizes a workload's performance as shown in Figure 6. At any given time, the RL agent resides in a state of the system (s_t); at each step, it selects an action(a_t) and transitions to a new state; while at the same time receiving a reward signal(r_t) as the result of the action. The main components in the AutoScratch-RL training loop are described below:

Environment: The RL environment is defined by the combination of a DL inference workload and the specific GPU platform it is executing on. The environment receives an action and provides the corresponding reward as explained

later in this section.

State: We define the RL state as the current L2 residency configuration that is represented by a bit vector of size *n*, or mathematically $s_t \in \{0,1\}^n$. Each bit of the vector represents the L2 residency status of the corresponding memory slice in the shared activation buffer shown previously in Figure 4.

Action: We define the RL action as either *promote*, in which we promote a single memory slice to become L2-resident, or alternatively *no action*, to keep the current L2-resident state unchanged. The action is represented by a one-hot bit vector of size *n+1*, or $a_t \in \{0,1\}^{n+1}$. The first *n* bits represent a promotion in one of the *n* memory slices and the last bit is used to indicate *no action*. By design, any action can lead to (at most) a minor change of the current state. This design leads to smooth transitions between states, which improve the RL training convergence. Because each action modifies at most one memory slice, many actions are required to convert the state from the initial zero vector to the final state which form an RL episode.

Agent: In practice, there are two networks in the RL agent of the PPO algorithm: the actor and the critic. For simplicity, we only show the actor network in Figure 6. They are both implemented as multi-layer perceptron (MLP) neural networks (Goodfellow et al., 2016) with two hidden layers of 64 neurons each and the state vector of size *n* as shared input. The output of the actor network is the action vector of size *n+1*, while the output of the critic network is a single scalar value that is used to evaluate the quality of the action.

Reward: We denote the DL application execution time of a single DL inference iteration as reported by the RL environment as $T(s_t)$. The RL reward while transitioning from s_t to s_{t+1} is $r_t = T(s_t) - T(s_{t+1})$. Simply put, the reward is defined as the performance improvement resulting from the current RL action. Positive rewards indicate performance speedup, while negative rewards indicate performance slowdown. In general other metrics could also be used as the reward function, such as DRAM traffic reduction. Later in Section 5.1 we discuss the implication of using DRAM traffic reduction rather than performance as the reward.

During the training process, an episode always starts with the initial state $s_0 = (0, \ldots, 0)$. When an episode reaches the state where the best next action is *no action*, it remains in the same state and action unless system noise accidentally causes a new action. As a result, when we observe that the last consecutive *k* states are the same, we terminate the episode; i.e., the episode terminates at step t_f if $s_{t_f-k} = s_{t_f-k+1} = \ldots = s_{t_f}$. An episode typically contains tens of iterations in our experiments.

The final state is the L2 residency configuration learned by AutoScratch in this episode. The accumulated reward of an episode is the execution time improvement of the final configuration, compared to the baseline with no L2-resident caching. Leveraging the PPO algorithm, the full training process of AutoScratch contains many training episodes.

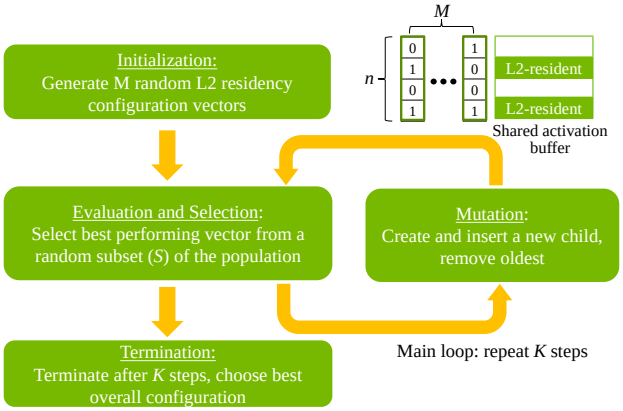


Figure 7. AutoScratch-EA with regularized evolutionary optimization for tuning GPU's L2 residency configuration.

In order to make sure we end up with the best L2 residency configuration overall, we keep track of the best performing configuration across the entire training process and use it as the final configuration to be deployed with the inference application. Both the target DL inference application and the RL agent run on the same GPU during training, but they cause little cross-interference in practice as their execution does not overlap on the GPU.

3.3 AutoScratch with an Evolutionary Optimizer

While AutoScratch-RL proved to be very successful in discovering near-optimal L2 residency configurations, it is quite computationally demanding as we later show in Section 5.2. Recently another family of ML, called Evolutionary Algorithms (EA), has shown that it can rival the performance of RL while being less computationally intense, providing a viable alternative to RL in many situations (Real et al., 2019). EA is simpler to implement as there is no need to train a separate neural network as the agent, is easier to scale in a distributed setting, and has fewer hyperparameters that need to be tuned.

Based on a regularized evolutionary algorithm originally developed for neural architecture search (Real et al., 2019), we develop AutoScratch-EA as summarized in Figure 7. We integrate AutoScratch-EA as an alternative for AutoScratch's ML-based optimization component in Figure 5. AutoScratch-EA keeps a population of M L2 residency configuration vectors, each of n bits, throughout the learning process. The population is initialized with M random configuration vectors where each configuration is evaluated for

its fitness (execution time) on the target GPU, then gradually evolved in a series of K steps. At each step, a random subset of S configuration vectors from the current population is selected and evaluated. Then the vector with the best fitness (shortest execution time) within this subset is selected as a parent.

A new child configuration vector is created by applying a mutation operation on the parent configuration vector. A mutation operation in AutoScratch-EA is defined as a single bit flip in a random position. This allows minimal change between the parent and the child configuration vectors in order to provide a smooth transition during the evolution process. The child configuration vector is then evaluated and added to the current population.

The original inference application is executed once during the evaluation process to return the execution time as the metric of fitness of the child's configuration. To keep the size of the population constant, an old configuration vector must be discarded. Traditionally, the worst configuration vector from the subset S is usually eliminated(Goldberg & Deb, 1991). In contrary to a traditional evolutionary algorithm, regularized evolutionary algorithm introduces a concept of age that leads to the continuous discarding of the oldest configuration in the population. As pointed out in the original publication (Real et al., 2019), an aging evolution helps better manage training noise throughout the experiments. We find it to be very helpful in this work as the execution time for each configuration vector on a target GPU can be noisy.

The algorithm terminates after finishing K steps and the configuration vector with the shortest execution time is selected as the final best solution. Notice that overall, $M + K$ configuration vectors are evaluated during the learning process because M vectors are evaluated during initialization and one vector is evaluated at each step.

4 METHODOLOGY

Our evaluations focus on the MLPerf inference suite (Reddi et al., 2020), one of the most widely-used DL inference benchmark suites across industry and academia. The detailed benchmark configurations are summarized in Table 1. We have set the individual benchmark batch sizes so that the shared activation buffer sizes are comparable to, but larger than the total GPU L2 capacity (48MB), with two notable exceptions. In *3d-unet*, the shared activation buffer size is larger than 1GB even at the minimum batch size of 1, making it impossible to scale it down any further. The *rnnt* application is comprised of several independent and pipelined sub-networks and we chose to optimize its major component, the encoder sub-network only. We observed that *rnnt* is based on a persistent RNN implementation (Diamos et al., 2016) which leads to significantly lower DRAM bandwidth requirements compared to other benchmarks with similarly activation buffer sizes. As a result, we chose to use a larger batch size for *rnnt* to maximize its DRAM utilization and performance. In *bert*, we use a fixed sequence length of 128 during both training and evaluation. When variable sequence lengths are applied during inference, we can train AutoScratch with an average sequence length.

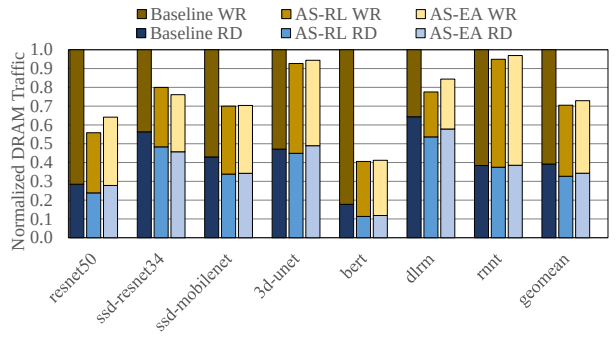
Table 1. MLPerf inference benchmark settings in AutoScratch.

BENCHMARK	PRECISION	BATCH SIZE	ACTIVATION BUFFER SIZE (MB)
RESNET50	INT8	32	63
SSD-RESNET34	INT8	6	104
SSD-MOBILENET	INT8	64	140
3D-UNET	INT8	1	278
BERT	INT8	32	81
DLRM	INT8	51200	106
RNNT	FP16	2048	4175

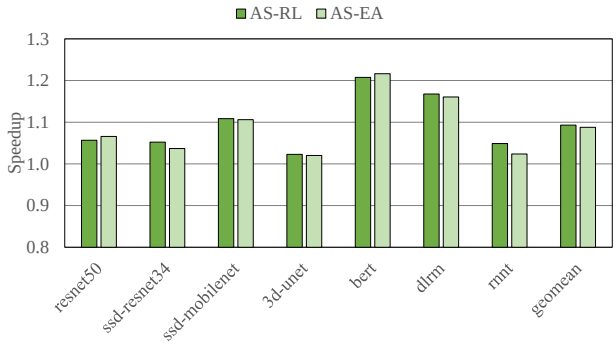
We evaluate AutoScratch using state-of-the-art NVIDIA's L4 GPU silicon (NVIDIA, 2023c). We select the L4 as the most representative evaluation platform for AutoScratch because it is the latest GPU targeting DL inference and features a relatively large L2 capacity of 48MB, of which 36MB can be configured in software using the residency control API. The L4 has a maximum TDP of 72W and 300GB/s of GPU memory bandwidth. Throughout our experiments, we found that L4 GPU often operates relatively close to its TDP, and its inference performance can vary between identical inference iterations due to its advanced power management events. Such variation in the per-iteration execution time can hurt the convergence of AutoScratch training. We worked around this issue by averaging the measured execution time collected from multiple consecutive iterations per workload.

We implement AutoScratch-RL using the stable-baselines3 (Raffin et al., 2019) codebase, leveraging the out-of-the-box PPO implementation with its default hyper-parameters. We perform 500,000 RL training steps in total for each workload during the training process. The AutoScratch training time is a function of both the total number of training steps and the execution latency of a single inference iteration performed during each RL training step. The entire training process typically takes a few hours for each workload, with the longest training time being around 13 hours for *bert*.

For AutoScratch-EA, we choose the population size M to be 100 and the sample subset size S to be 25, as recommended by the original paper (Real et al., 2019). We set the maximum number of steps (K) as 20,000, which we found to be sufficient across all benchmarks in our experiments. We add an early-stop mechanism which will terminate the training process if the best solution is not improved within 5,000



(a) Offchip DRAM Traffic



(b) Performance Speedup

Figure 8. The DRAM traffic and performance speedups measured on NVIDIA's L4 GPU silicon with AutoScratch-RL(AS-RL) and AutoScratch-EA(AS-EA) relative to the baseline with a hardware-controlled cache policy. The y-axis in (b) starts at 0.8 for better visibility.

steps. The total number of steps required for AutoScratch-EA is much smaller than AutoScratch-RL and as a result, the overall learning time of AutoScratch-EA is much shorter, as shown later in Section 5.2.

5 RESULTS

We evaluate AutoScratch by measuring its DRAM traffic reduction and speedup compared to the baseline, along with its training overhead and sensitivity to batch size. We also compare AutoScratch to two additional alternative optimization methods: Random Search (RS) and Human-Designed Heuristic (HDH) that has been integrated into the latest TensorRT library and is turned off by default (NVIDIA, 2021b). Finally we discuss the advantages and limitations of our approach, and also compare AutoScratch-RL with AutoScratch-EA.

5.1 DRAM Traffic Reduction and Performance

We evaluate AutoScratch on NVIDIA's L4 GPU silicon by measuring both DRAM traffic and overall application execution time, with and without AutoScratch optimization. Figure 8(a) shows the normalized off-chip DRAM traffic when using the L2 residency configurations learned by AutoScratch with both RL and EA optimizers (AS-RL and AS-EA). The DRAM traffic is subdivided into reads and writes (RD and WR). The results are normalized to the baseline in which L2 residency controls are disabled and the DRAM traffic numbers are collected from GPU performance counters over the execution of a single inference iteration.

Interestingly, the L2 resident configurations learned by two different AutoScratch-RL and AutoScratch-EA optimizers both resulted in similar and significant DRAM traffic reductions, with a respective geomean reduction of 29% and 27% across all benchmarks. AutoScratch-RL achieves a slightly lower average DRAM traffic overall.

In general, workloads with activation buffer sizes closer to the GPU's L2 capacity realize larger DRAM traffic reductions than those with larger activation buffer sizes because a larger fraction of the application's data benefits from improved on-chip caching. We also observe that DRAM write traffic reductions are generally more significant than the reductions in read traffic because AutoScratch's L2 residency configurations effectively prevent these activations from being written back to DRAM.

Figure 8(b) reports the realized performance benefits of the AutoScratch-RL and AutoScratch-EA optimizations compared to the baseline hardware-only cache management. The inference performance is measured by averaging the execution time of the workload across 100 inference iterations to reduce the execution noise. We also perform a 10-second warm-up run before measuring actual performance to exclude any initialization overheads from our measurements. Similar to the DRAM traffic reduction results, AutoScratch-RL and AutoScratch-EA both both achieve similar geomean performance speedups of 9.3% and 8.8%, while also reaching up to 21% and 22% performance improvements in *bert*, respectively. AutoScratch-RL achieves slightly better performance overall, but not in every workload. We believe it is due to AutoScratch-RL being more resilient to power-management induced variation on L4 as we did not observe such behavior in our early experiments using an NVIDIA's A100 GPU with a substantially higher TDP.

We observe that workloads with higher DRAM traffic reductions typically achieve larger performance improvements, but not always. For example, *ssd-mobilenet* achieves the larger speedup than *resnet50* using both optimizers, but its relative DRAM traffic reduction is lower. This is because

Table 2. The total training times (in minutes) of AutoScratch for MLPerf inference workloads on an NVIDIA's L4 GPU, and the relative time reduction achieved by AutoScratch-EA versus AutoScracth-RL.

WORKLOAD	AS-RL	AS-EA	TIME REDUCTION
RESNET50	241	6.8	35×
SSD-RESNET34	421	5.8	72×
SSD-MOBILENET	260	8.0	33×
3D-UNET	390	4.4	88×
BERT	765	10.2	75×
DLRM	238	5.4	44×
RNNT	227	2.5	90×
GEOMEAN	330	5.7	58×

ssd-mobilenet uses depthwise separable convolution operators that are more sensitive to memory bandwidth than regular convolution operators used in *resnet50* due to its unbalanced kernel shapes. In general, AutoScratch results in higher speedups for MLP-based workloads (*bert* and *dlrm*) compared to convolution-based workloads (*resnet50*, *ssd-resnet34*, *ssd-mobilenet* and *3d-unet*). This is because convolution operators need to be converted in order to take advantage of General Matrix Multiplication (GEMM) operators in modern GPUs, resulting in lower arithmetic intensities than MLP operators that can use GEMM directly (NVIDIA, 2023b). As a result, MLP-based workloads are typically more sensitive to memory bandwidth than convolution-based workloads.

To demonstrate the stability of AutoScratch's training performance, we train each workload multiple times with random initial seeds. Both AutoScratch-RL and AutoScratch-EA result in very consistent performance, with a coefficient of variance within 0.02 across runs.

5.2 The Overhead of ML-Based Optimization

While both versions of the AutoScratch optimizers achieve impressive optimization results, ML-based optimization does incur some overhead. The cost in this case is the additional training overhead that is added to the DL application compilation and optimization workflow. We compare the training time of AutoScratch-RL and AutoScratch-EA in Table 2. RL-based optimization takes substantially more time than the EA-based approach for all workloads. The training process with AutoScratch-EA takes several minutes across most workloads and is overall 58× faster than training the AutoScratch-RL module. The training time of AutoScratch-EA is short enough to make it a practical candidate for integration into a DL compilation framework such as NVIDIA's TensorRT as an optional profile-guided optimization where the TensorRT compilation time for those workloads range from 1 to 7 minutes. While this work describes how we apply AutoScratch as an offline optimization process, when applied in an online regime, the learning overhead of AutoScratch-EA can be amortized across many executions as it learns on-the-fly during deployment.

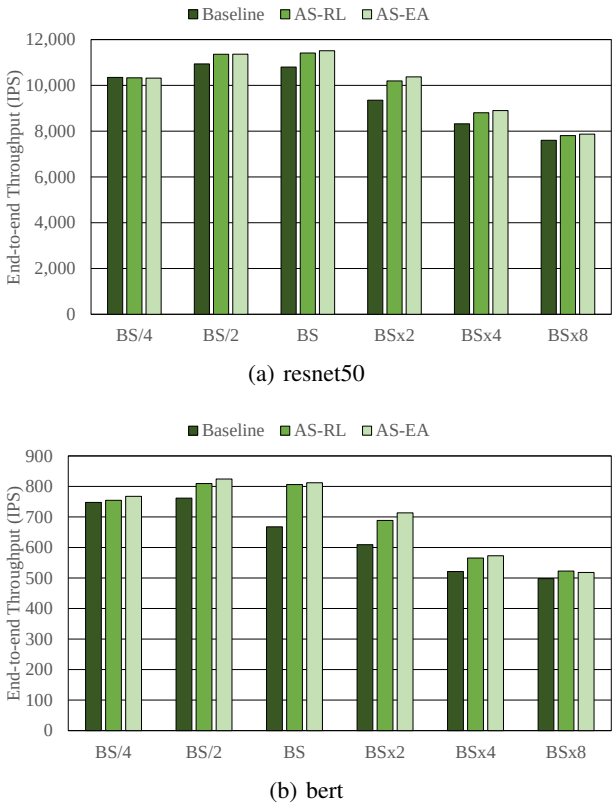


Figure 9. The absolute end-to-end performance measured in inferences per second (IPS) on NVIDIA's L4 GPU silicon, for the baseline without L2 residency controls and AutoScratch-RL and AutoScratch-EA for *resnet50* and *bert* at varying batch sizes.

5.3 Sensitivity to Batch Size

To understand the impact of the batch size selection on the relative efficiency improvement of AutoScratch, we show sweeps of the batch size in *resnet50* and *bert* as representative benchmarks for convolution-based and MLP-based DL workloads, respectively. Figure 9 summarizes the absolute end-to-end performance achieved with AutoScratch-EA and AutoScratch-RL on *resnet50* and *bert* while varying batch sizes from one quarter, to eight times the default batch sizes (shown in Table 1). We observe again that both versions of AutoScratch result in very similar performance speedups when compared to the baseline without AutoScratch and both work well across a wide range of batch sizes in each application. The best absolute performance is achieved in the middle of the batch size range for both workloads, where AutoScratch is most effective. This is because when the batch size is too small, the majority of activations fit in

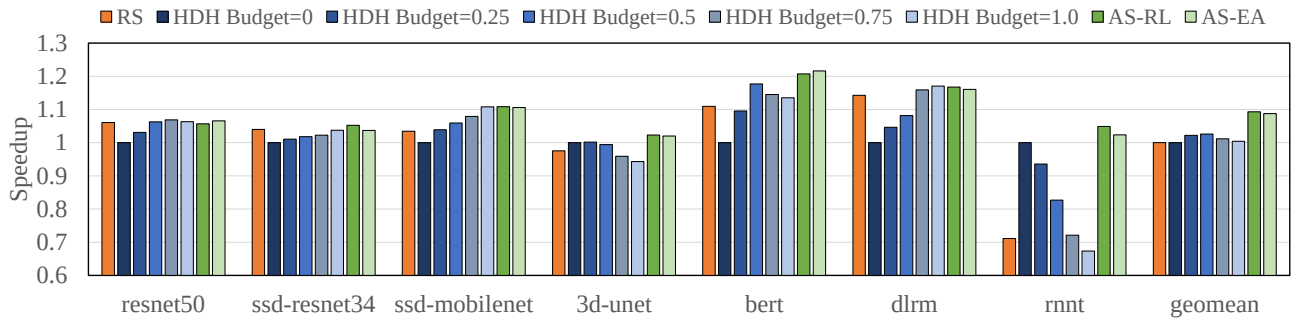


Figure 10. Comparing the performance speedups of AutoScratch-RL(AS-RL) and AutoScratch-EA(AS-EA) to Random Search (RS) and Human-Designed Heuristic (HDH) at various L2 residency budgets on NVIDIA's L4 GPU silicon. All results are relative to the baseline without any L2 residency controls (the default configuration). The y-axis in (b) starts at 0.6 for better visibility.

L2 cache even without L2 residency controls, leaving little room for AutoScratch to optimize. On the other hand, if the batch size is too large, only a small portion of activations can be pinned in the L2, even when optimized by AutoScratch.

Note that in *resnet50* the original best-performing batch size in the baseline is half of the default batch size (BS/2). AutoScratch shifts the overall best-performing point to the default batch size. In *bert* the best-performing batch sizes for both the baseline and AutoScratch are the same (BS/2). We conclude that AutoScratch could shift the best-performing batch size to larger values because it alleviates DRAM bandwidth bottlenecks. The realization that batch-size tuning is integrally linked with both DRAM bandwidth utilization and L2 residency control opens the door for future work where AutoScratch can optimize the batch size along with the L2 residency configuration.

5.4 Comparison Against Other Optimization Methods

We compare AutoScratch to two additional optimization techniques: Random Search (RS) and Human-Designed Heuristic (HDH). RS generates a random set of K configurations and picks the best configuration within the set. In our experiments, we set K to be 20,000 which is equal to the maximum number of training steps in AutoScratch-EA for fair comparison. HDH is designed by GPU experts at NVIDIA and has been integrated into the latest TensorRT library (since v8.5) (NVIDIA, 2021b). HDH is turned off in TensorRT by default, and can be enabled in software by providing the L2 residency budget ranging from 0 to 1.0 where 0 means no L2 residency and 1.0 means using the entire L2 budget that is available for residency controls (36MB in an L4 GPU). For a given L2 residency budget (set by a programmer), HDH comes up with the residency settings based on its built-in heuristic method. We sweep the L2 residency budget from 0 to 1.0 with increments of 0.25 to cover the entire search space for HDH. In contrast, RS and AutoScratch automatically search for the best L2 residency settings within a given L2 capacity budget (1.0 in our experiment). RS and AutoScratch may find a configuration that does not use the entire budget for example, while HDH is designed to always use up the entire budget.

Figure 10 summarizes the performance speedups normalized to the baseline hardware-controlled cache policy. We note that RS achieves good performance speedups in workloads with relatively small search spaces that correspond to small activation buffer sizes (Table 1) such as *resnet50* and *ssd-resnet34*. However, RS completely fails in discovering a comparable solution to AutoScratch in workloads with larger search spaces, and sometimes results in substantially worse performance than the baseline in *3d-unet* and *rnnt* (since random search does not necessarily find the baseline configuration). Overall, the geomean speedup of RS is close to 0 compared to the baseline, rendering it a useless solution. HDH is more effective than RS, and can lead to a geomean speedup of 2.6% with the best L2 residency budget of 0.5 overall. However, as clearly shown in Figure 10, the best L2 residency budget varies across different workloads, and additional parameter tuning on the programmer side is required to pick the best-performing L2 residency budget for each workload. AutoScratch, on the other hand, can automatically discover the optimal solution within the L2 residency budget without the additional parameter tuning step, resulting in larger performance speedups overall.

5.5 Discussion

We have discussed how AutoScratch can be applied to real GPU systems to optimize performance while treating both the application and GPU as black boxes, with limited information needing to be exposed to the optimizer. The main benefit of this black box approach is that it is practical, efficient, and applicable to arbitrary workloads running on an arbitrary GPU. Nevertheless, every time there is a change in

the workload, the GPU configuration, or even the batch size, AutoScratch should be re-trained to best optimize the L2 residency configuration. Though there may be cases where one learned solution can be applied, albeit sub-optimally, across a variety of configurations - we have not studied that effect in this work.

AutoScratch is mostly applicable to GPU hardware configurations in which the workload's performance is limited by DRAM bandwidth and/or energy, and a large L2 cache is equipped to combat these limitations. An ideal example of this is the NVIDIA's L4 GPU, as shown in this paper. We have also evaluated AutoScratch on a high-end NVIDIA's A100 GPU featuring a large L2 cache and high memory bandwidth (NVIDIA, 2020a). When trained on an A100 GPU configuration with reduced HBM bandwidth, AutoScratch could find highly optimized L2 residency solutions yielding a notable 44% geomean DRAM traffic reduction. However this DRAM traffic reduction did not translate into performance speedup when deployed on the original A100 configuration, because inference performance on A100 hardware was neither limited by the HBM bandwidth nor energy though it does improve the GPU's performance per watt.

When comparing the RL and EA optimizers, we found that the latter can achieve similar performance to the former, but with significantly shorter training time. This can be explained by the fact that in contrary to EA, RL learns more than the optimized residency configuration itself. In fact, its output is a policy that can reach the nearly best configuration. This could be exploited in future work for generalization over a search space spanning parameters such as neural architecture, batch size, cache size, etc. For example, a pre-trained RL agent can be deployed to generate optimized L2 residency configurations for arbitrary inference applications without re-training. Although a generalized approach can eliminate hours of per-application optimization overhead, it can also complicate the pre-training process and its convergence rate, potentially leading to inferior results. Therefore, we choose to leave it as future work.

6 Related Work

A variety of approaches have been used for explicit control of caches and scratchpad memories on CPUs (Nguyen, 2016), GPUs (Kerr et al., 2017; Bauer et al., 2011), heterogeneous systems (Komuravelli et al., 2015), and hardware accelerators (Lacey, 2020; Pellauer et al., 2019; Chen et al., 2014; 2016; Fowers et al., 2018). They are usually labor-intensive and often require both manual control and explicit software tuning. In contrast, AutoScratch is an automated process to tune software-based cache control operations for workloads with arbitrary memory allocations.

There have been a variety of attempts targeting RL, EA, or other ML techniques for cache management (Khadka et al., 2020; Li et al., 2020; Sethumurugan et al., 2021; Teran et al., 2016; Shi et al., 2019; Liu et al., 2020). These works typically leverage ML techniques in two ways: (1) directly applying ML-learned solutions to cache replacement policy management, or (2) using the insights observed from ML-learned solutions to improve heuristic-based traditional cache management schemes. A prior work (Khadka et al., 2020) on per-layer memory placement on the Intel NNP-I, using a combination of RL, graph neural networks, and evolutionary search is the closest to ideas proposed in this paper. The optimization objective in (Khadka et al., 2020) is to find the best data placement among memory components (DRAM, LLC and scratchpad) for the activation and weight components of each layer, with the search space growing exponentially with the number of layers. In contrast, AutoScratch focuses on finding the optimal selection of shared activation memory slices across all layers to be explicitly pinned in the LLC. The search space of AutoScratch depends only on the size of memory address range to be selected among and the memory slice granularity, but is independent of the number of layers. Because AutoScratch uses this generic memory array abstraction, AutoScratch could be generalized to other iterative GPU workloads with similar dataflow patterns.

Researchers have also applied RL and EA techniques to other problems in the field of computer architecture such as hardware prefetching (Bera et al., 2021; Peled et al., 2015), NoCs (Lin et al., 2020), memory controllers (Mukundan & Martinez, 2012; Ipek et al., 2008), and hardware resource assignments (Kao et al., 2020; Kao & Krishna, 2020; Kao et al., 2022). While being broadly related, they are orthogonal to the problem and solution described in this paper.

7 Conclusion

We developed AutoScratch to discover and optimize GPU cache residency controls for DL inference workloads. Our GPU silicon-based evaluations show that AutoScratch achieves an impressive DRAM traffic reduction of 29% and performance speedup of 9% (geomean) for the MLPerf inference workloads on an inference-oriented NVIDIA's L4 GPU. When comparing EA with RL, we found that AutoScratch with EA-based optimizer learns 58× faster, while providing nearly equivalent results, making it a suitable choice for practical deployment.

We believe the future trajectory of architectural innovations lies in a blending of hardware mechanisms and software ML-based tuning. In this work, we identify and explore one such co-design opportunity with the hope that it will inspire architects to design additional hardware hooks to enable these kinds of optimization mechanisms in other architectural domains.

REFERENCES

AMD. Introducing AMD CDNA™ 2 Architecture, 2021. URL https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf.

Anderson, M., Chen, B., Chen, S., Deng, S., Fix, J., Gschwind, M., Kalaiah, A., Kim, C., Lee, J., Liang, J., Liu, H., Lu, Y., Montgomery, J., Moorthy, A., Nadathur, S., Naghshineh, S., Nayak, A., Park, J., Petersen, C., Schatz, M., Sundaram, N., Tang, B., Tang, P., Yang, A., Yu, J., Yuen, H., Zhang, Y., Anbudurai, A., Balan, V., Bojja, H., Boyd, J., Breitbach, M., Caldato, C., Calvo, A., Catron, G., Chandwani, S., Christeas, P., Cottel, B., Coutinho, B., Dalli, A., Dhanotia, A., Duncan, O., Dzhabarov, R., Elmir, S., Fu, C., Fu, W., Fulthorp, M., Gangidi, A., Gibson, N., Gordon, S., Hernandez, B. P., Ho, D., Huang, Y.-C., Johansson, O., Juluri, S., Kanaujia, S., Kesarkar, M., Killinger, J., Kim, B., Kulkarni, R., Lele, M., Li, H., Li, H., Li, Y., Liu, C., Liu, J., Maher, B., Mallipedi, C., Mangla, S., Matam, K. K., Mehta, J., Mehta, S., Mitchell, C., Muthiah, B., Nagarkatte, N., Narasimha, A., Nguyen, B., Ortiz, T., Padmanabha, S., Pan, D., Poojary, A., Ye, Qi, Raginel, O., Rajagopal, D., Rice, T., Ross, C., Rotem, N., Russ, S., Shah, K., Shan, B., Shen, H., Shetty, P., Skandakumaran, K., Srinivasan, K., Sumbaly, R., Tauberg, M., Tzur, M., Verma, S., Wang, H., Wang, M., Wei, B., Xia, A., Xu, C., Yang, M., Zhang, K., Zhang, R., Zhao, M., Zhao, W., Zhu, R., Mathews, A., Qiao, L., Smelyanskiy, M., Jia, B., and Rao, V. First-Generation Inference Accelerator Deployment at Facebook. In *arXiv preprint arXiv:2107.04140*, 2021.

Bauer, M., Cook, H., and Khailany, B. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

Belady, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2): 78–101, 1966. doi: 10.1147/sj.52.0078.

Bera, R., Kanellopoulos, K., Nori, A., Shahroodi, T., Subramoney, S., and Mutlu, O. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1121–1137, 2021.

Chatterjee, N., O'Connor, M., Lee, D., Johnson, D. R., Keckler, S. W., Rhu, M., and Dally, W. J. Architecting an Energy-Efficient DRAM System for GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., and Temam, O. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

Chen, Y.-H., Emer, J., and Sze, V. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

Diamos, G., Sengupta, S., Catanzaro, B., Chrzanowski, M., Coates, A., Elsen, E., Engel, J., Hannun, A., and Satheesh, S. Persistent RNNs: Stashing Recurrent Weights on-Chip. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, pp. 2024–2033, 2016.

Dulac-Arnold, G., Mankowitz, D., and Hester, T. Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*, 2019.

Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., Heil, S., Patel, P., Sapek, A., Weisz, G., Woods, L., Lanka, S., Reinhardt, S. K., Caulfield, A. M., Chung, E. S., and Burger, D. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

Fu, Y., Bolotin, E., Chatterjee, N., Nellans, D., and Keckler, S. W. GPU Domain Specialization via Composable On-Package Architecture. *ACM Trans. Archit. Code Optim.*, 19(1), dec 2021.

Goldberg, D. E. and Deb, K. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms*, volume 1, pp. 69–93. Elsevier, 1991.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Gupta, U., Wu, C.-J., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Hempstead, M., Jia, B., Lee, H.-H. S., Malevich, A., Mudigere, D., Smelyanski, M., Xiong, L., and Zhang, X. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

Ipek, E., Mutlu, O., Martínez, J. F., and Caruana, R. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *International Symposium on Computer Architecture*, pp. 39–50, 2008.

Jain, A. and Lin, C. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 78–89, 2016. doi: 10.1109/ISCA.2016.17.

Jaleel, A., Theobald, K. B., Steely, S. C., and Emer, J. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 60–71, 2010.

Jiang, N. and Li, L. Doubly robust off-policy value evaluation for reinforcement learning. In *International Conference on Machine Learning*, pp. 652–661. PMLR, 2016.

Jouppi, N. P., Hyun Yoon, D., Ashcraft, M., Gottscho, M., Jablin, T. B., Kurian, G., Laudon, J., Li, S., Ma, P., Ma, X., Norrie, T., Patil, N., Prasad, S., Young, C., Zhou, Z., and Patterson, D. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2021.

Kao, S.-C. and Krishna, T. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2020.

Kao, S.-C., Jeong, G., and Krishna, T. ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 622–636, 2020.

Kao, S.-C., Pellauer, M., Parashar, A., and Krishna, T. DiGamma: Domain-aware Genetic Algorithm for HW-Mapping Co-optimization for DNN Accelerators. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 232–237, 2022.

Kerr, A., Merrill, D., Demouth, J., and Tran, J. CUTLASS: Fast Linear Algebra in CUDA C++, 2017. URL https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/.

Khadka, S., Aflalo, E., Marder, M., Ben-David, A., Miret, S., Mannor, S., Hazan, T., Tang, H., and Majumdar, S. Optimizing Memory Placement using Evolutionary Graph Reinforcement Learning. In *arXiv preprint, arXiv:2007.07298*, 2020.

Knowles, S. Graphcore Colossus Mk2 IPU. In *HotChips-33*, 2021.

Komuravelli, R., Sinclair, M. D., Alsop, J., Huzaifa, M., Kotsifakou, M., Srivastava, P., Adve, S. V., and Adve, V. S. Stash: Have your scratchpad and cache it too. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 707–719, 2015.

Krashinsky, R., Giroux, O., Jones, S., Stam, N., and Ramaswamy, S. NVIDIA Ampere Architecture In-Depth, 2020. URL https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

Lacey, D. Intelligent memory for intelligent computing, 2020. URL https://www.graphcore.ai/posts/intelligent-memory-for-intelligent-computing.

Li, B., Wang, Y., Wang, R., Tai, C., Iyer, R., Zhou, Z., Herdrich, A., Zhang, T., Haj-Ali, A., Stoica, I., and Asanovic, K. RLDRM: Closed Loop Dynamic Cache Allocation with Deep Reinforcement Learning for Network Function Virtualization. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pp. 335–343, 2020. doi: 10.1109/NetSoft48620.2020.9165471.

Lie, S. Wafer Scale Deep Learning. In *HotChips-31*, 2019.

Lin, T.-R., Penney, D., Pedram, M., and Chen, L. A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 99–110, 2020.

Liu, E. Z., Hashemi, M., Swersky, K., Ranganathan, P., and Ahn, J. An Imitation Learning Approach for Cache Replacement. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

Mukundan, J. and Martinez, J. F. Morse: Multi-objective reconfigurable self-optimizing memory scheduler. In *IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–12, 2012.

Nguyen, K. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family, 2016. URL https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html.

NVIDIA. NVIDIA Tesla V100 Architecture, 2017. URL http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

NVIDIA. NVIDIA A100 Tensor Core GPU Architecture, 2020a. URL https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

NVIDIA. NVIDIA T4 70W Low Profile PCIe GPU Accelerator, 2020b. URL https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-product-brief.pdf.

NVIDIA. CUDA Toolkit Documentation v11.5.0, 2021a. URL https://docs.nvidia.com/cuda/index.html.

NVIDIA. NVIDIA TensorRT SDK, 2021b. URL https://developer.nvidia.com/tensorrt.

NVIDIA. NVIDIA Ada GPU Architecture, 2023a. URL https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.0.pdf.

NVIDIA. Optimizing Convolutional Layers User's Guide, 2023b. URL https://docscontent.nvidia.com/dita/00000186-1a08-d34f-a596-3f291b140000/deeplearning/performance/pdf/Optimizing-Convolutional-Layers-User-Guide.pdf.

NVIDIA. NVIDIA L4 Tensor Core GPU, 2023c. URL https://nvdam.widen.net/s/rvq98gbwsw/l4-datasheet-2595652.

Park, J., Naumov, M., Basu, P., Deng, S., Kalaiah, A., Khudia, D., Law, J., Malani, P., Malevich, A., Nadathur, S., Pino, J., Schatz, M., Sidorov, A., Sivakumar, V., Tulloch, A., Wang, X., Wu, Y., Yuen, H., Diril, U., Dzhulgakov, D., Hazelwood, K., Jia, B., Jia, Y., Qiao, L., Rao, V., Rotem, N., Yoo, S., and Smelyanskiy, M. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. In *arXiv preprint arXiv:1811.09886*, 2018.

Peled, L., Mannor, S., Weiser, U., and Etsion, Y. Semantic Locality and Context-Based Prefetching Using Reinforcement Learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 285–297, 2015. ISBN 9781450334020.

Pellauer, M., Shao, Y. S., Clemons, J., Crago, N. C., Hegde, K., Venkatesan, R., Keckler, S. W., Fletcher, C. W., and Emer, J. S. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

Pettis, K. and Hansen, R. C. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp. 16–27, New York, NY, USA, 1990.

Puterman, M. L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., and Dormann, N. Stable baselines3. https://github.com/DLR-RM/stable-baselines3, 2019.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, 2019.

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., Chukka, R., Coleman, C., Davis, S., Deng, P., Diamos, G., Duke, J., Fick, D., Gardner, J. S., Hubara, I., Idgunji, S., Jablin, T. B., Jiao, J., John, T. S., Kanwar, P., Lee, D., Liao, J., Lokhmotov, A., Massa, F., Meng, P., Micikevicius, P., Osborne, C., Pekhimenko, G., Rajan, A. T. R., Sequeira, D., Sirasao, A., Sun, F., Tang, H., Thomson, M., Wei, F., Wu, E., Xu, L., Yamada, K., Yu, B., Yuan, G., Zhong, A., Zhang, P., and Zhou, Y. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, pp. 446–459, 2020.

Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. VDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Sethumurugan, S., Yin, J., and Sartori, J. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 291–303, 2021. doi: 10.1109/HPCA51647.2021.00033.

Shah, I., Jain, A., and Lin, C. Effective Mimicry of Belady's MIN Policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 558–572, 2022. doi: 10.1109/HPCA53966.2022.00048.

Shi, Z., Huang, X., Jain, A., and Lin, C. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 413–425, 2019.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Teran, E., Wang, Z., and Jiménez, D. A. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

Underwood, C. *Use Cases of Recommendation Systems in Business – Current Applications and Methods*. 2020. https://emerj.com/ai-sector-overviews/use-cases-recommendation-systems/.

Villa, O., Lustig, D., Yan, Z., Bolotin, E., Fu, Y., Chatterjee, N., Jiang, N., and Nellans, D. Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

Wu, C.-J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely, S. C., and Emer, J. SHiP: Signature-based Hit Predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 430–441, 2011.

Young, V., Chou, C.-C., Jaleel, A., and Qureshi, M. Ship++: Enhancing signature-based hit predictor for improved cache performance. In *The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA 2017)*, 2017.

A Comparison to Hardware-Based Cache Replacement Policies

We have explored AutoScratch performance on the state-of-the-art NVIDIA L4 GPU product. Prior research also has proposed several hardware-based cache replacement policies (Jaleel et al., 2010; Wu et al., 2011; Young et al., 2017; Jain & Lin, 2016; Shi et al., 2019; Shah et al., 2022), unfortunately most of them are less practical as the rely either on oracle information or on exposing the PC information to the LLC. Because modern high performance systems do not propagate expensive PC information from all cores to all LLC banks, we focus our comparison to high-performance cache replacement policy that is practical to implement, such as hardware-based Signature-based Hit Predictor (SHiP) (Wu et al., 2011). As such to further understand the effectiveness of AutoScratch, we compare it with both SHiP and Belady's algorithm (Belady, 1966) to measure the benefits of explicit cache control against the state-of-the-art hardware policy and the theoretical upper bound of performance. Inspired by AutoScratch that only learns for activations, we also implement a novel software-assisted SHiP (SHiP-SW) to try and close the performance gap between AutoScratch and SHiP. Compared to SHiP that learns the re-reference interval for all signatures, SHiP-SW only learns the re-reference interval for important signatures that are identified (activations for inference workloads) and prioritized by software while the remaining signatures are treated with low priority.

A.1 Hardware-Based Replacement Polices

Architects traditionally improve on-chip caching efficiency by optimizing hardware-based cache replacement policies. Recent work has proposed a variety of advanced hardware-based replacement policies (Jaleel et al., 2010; Wu et al., 2011; Jain & Lin, 2016; Shi et al., 2019; Shah et al., 2022) to address the limitations of least recently used (LRU) replacement policy. However, many recent proposals rely on program counter (PC) information at the last-level cache, which is not available in modern high performance processors due to the cost of passing that information from all execution pipelines across the on-chip interconnect hierarchy. Adding support for this functionality in the GPU's L2 cache would introduce significant overhead, as there are a large number of streaming multiprocessors (SMs) on a GPU and the memory subsystem would require PC information to be forwarded from each of the SMs to the L2 cache alongside each request. In this paper, we choose to compare AutoScratch to SHiP (Wu et al., 2011) and its variants that do not rely on PC information to understand if ML-based LLC control can be better than state-of-the-art hardware-based cache replacement policies.

Signature-Based Hit Predictor (SHiP): SHiP (Wu et al., 2011) is a fine-granularity cache replacement policy that categorizes cache insertions into different groups by associating a *signature* with each cache reference. SHiP proposed three different signatures: a memory address, a PC of the missing memory instruction, or an instruction sequence. In this work, we use the 4KB aligned memory address the cache reference falls into as the signature. SHiP relies on the insight that cache references having the same signature will likely have the same re-reference interval. The original SHiP paper utilizes the re-reference pattern learned by SHiP to improve the SRRIP replacement policy (Jaleel et al., 2010). In this work, we use SHiP to help classify L2 cachelines into either the L2-resident or regular class similar to what AutoScratch does.

To learn the re-reference pattern of a signature, SHiP maintains two additional fields with each cacheline: the signature itself and a single bit to track the *outcome* of the cache insertion. The *outcome* bit is initially set to zero and set to one only if the cacheline receives a hit after insertion. SHiP learns the re-reference interval of a given signature by maintaining a table of saturating counters called the *Signature History Counter Table (SHCT)*. SHiP updates the SHCT upon cache evictions – when a cacheline is evicted, the SHCT is indexed with the signature and the corresponding counter is decremented (if the outcome bit is zero) or incremented (if the outcome bit is one). The SHCT counter value indicates the re-reference behavior of a signature. If the counter value is zero, it implies that future cache insertions by the signature are unlikely to be re-referenced, thus the associated cachelines will be classified as the regular class. Otherwise, the associated cachelines will be classified as the L2-resident class.

Software-Assisted SHiP (SHiP-SW): SHiP learns a re-reference interval prediction for *all* signatures that exist in an application. However, there are two drawbacks with the conventional SHiP design. First, for practical SHCT sizes, multiple signatures indexing the same SHCT entry can cause destructive interference. In such scenarios, the SHCT is unable to provide an accurate re-reference prediction for the interfering signatures. Second, SHiP treats all signatures equally and makes the same re-reference prediction for *all* signatures that receive cache hits. While this approach attempts to maximize cache utility by retaining signatures that receive cache hits, it does not take into account the *criticality* of a signature to performance. For example, cache lines inserted by a given signature *S1* might be more critical to performance than cachelines inserted by a different signature *S2* (even if both of them receive cache hits). In such situations, it is more important to give more cache space to cachelines inserted by signature *S1* than by signature *S2*.

Signature criticality can either be learned dynamically or di-

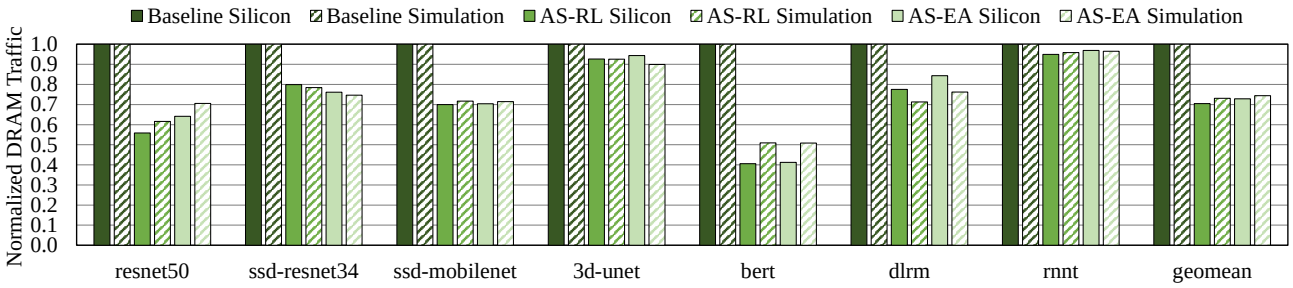


Figure 11. Normalized DRAM traffic comparison with Baseline, AutoScratch-RL, and AutoScratch-EA between silicon and simulation results.

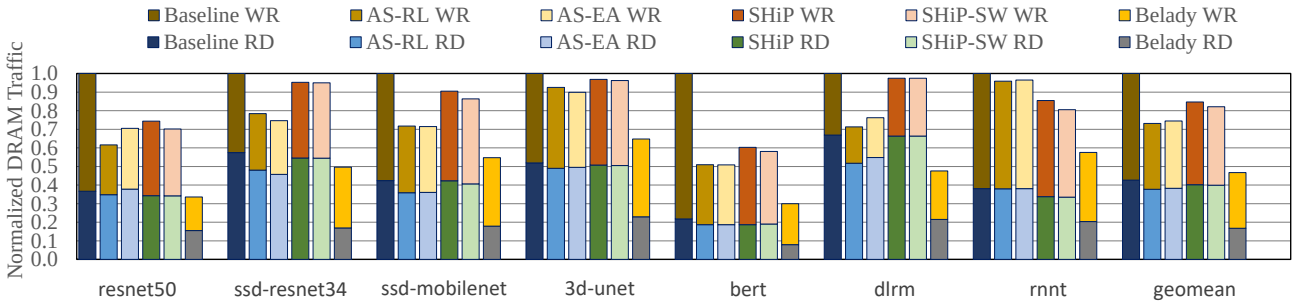


Figure 12. Simulated DRAM traffic with AutoScratch-RL, AutoScratch-EA, SHiP, SHiP-SW, and Belady's algorithm, normalized to the baseline without L2 residency controls.

rectly specified in the software. Motivated by AutoScratch, we propose SHiP-SW, a hardware-software co-designed cache replacement policy where the application provides SHiP hardware with critical signature(s) for the application. For example, when SHiP uses the memory address as the signature, the software will provide the cache hardware with the memory address region(s) that are deemed performance critical. We assume that the critical signatures are specified using a set of programmable registers within the SM and criticality information is sent down to the memory hierarchy as part of the memory request packet. This additional critical bit is still substantially less costly than a full PC used in some of previously proposed PC-based hardware replacement policies. At the L2, SHiP-SW then assigns cachelines at the regular class for all non-critical signatures and learns to classify cachelines as either regular or L2-resident for critical signatures only.

Belady's Algorithm: Belady's algorithm (Belady, 1966) is a theoretical cache replacement policy that provides optimal cache performance in terms of cache hit rate. The key idea is to always evict the cacheline that is reused furthest in the future during cache replacement. Because it requires oracular information about future cacheline accesses, it is impractical to implement in real silicon. Therefore in this work, we use it only to serve as a theoretical upper bound for comparison purposes.

A.2 Additional Results

Thus far we have reported the performance speedups and DRAM traffic reductions achieved by AutoScratch on NVIDIA L4 GPU silicon. To compare AutoScratch to a previously proposed non-PC-based hardware replacement policy (SHiP) and the theoretical best Belady's replacement algorithm, we perform a simulation-based evaluation.

To enable comparison with hardware-based cache replacement policies, we use a variant of a trace-based GPU simulator similar to NVArchSim (Villa et al., 2021) to simulate the same GPU configuration and workloads as in the silicon-based evaluation. Because the GPU's L2 cache is concurrently accessed by multiple SMs, similar to a multi-core CPU, the Belady's algorithm is not well defined in such scenario (Shah et al., 2022). The cache access order changes with every differing cache replacement decision. To overcome this issue, we choose to record the L2 access traces from the GPU simulator using the default cache replacement policy and feed them back into a separate functional python-based L2 cache model in order to simulate L2 and memory system behavior. By doing this, we generate a deterministic L2 access sequence for each workload, allowing us to use a

straightforward implementation of Belady's algorithm along with other replacement policies.

First, to demonstrate confidence in the fidelity of our simulation infrastructure, we compare the normalized DRAM traffic reductions of AutoScratch-RL and AutoScratch-EA against the silicon results in Figure 11. Our results indicate that the DRAM traffic reductions reported by our simulation infrastructure track very closely with the silicon results and are within 3% geomean across all evaluated workloads. Thus we report our simulation results as-is, without adding any manual correction factors when showing simulation results.

Figure 12 shows the simulated DRAM traffic with AutoScratch-RL (AS-RL), AutoScratch-EA (AS-EA), SHiP, software-assisted SHiP (SHiP-SW) and Belady's algorithm, normalized to the baseline without L2 residency controls. Both AutoScratch-RL and AutoScratch-EA outperform SHiP across all workloads except for *rnnt*, with an extra overall traffic reductions of 11% and 10% in geomean respectively, that mostly come from write traffic minimization. SHiP-SW slightly reduces the DRAM traffic by additional 2% over SHiP, but is unable to close the gap with AutoScratch. Compared to the theoretical best Belady's algorithm, AutoScratch-RL and AutoScratch-EA result in a 26% and 27% gap in terms of DRAM traffic reductions, respectively. We conclude that there is still potential room for improvement beyond AutoScratch.

In summary, we show in simulation that AutoScratch outperforms the state-of-the-art hardware-based replacement policy SHiP, by achieving a 11% larger DRAM traffic reduction and reducing the gap with an oracular Belady's to 26%. We augment SHiP with software-identified address regions (SHiP-SW) to achieve an additional 2% DRAM traffic reduction versus the original SHiP, but cannot close the gap with AutoScratch. We conclude that ML-based software control of caches is a promising new technique that can be utilized in modern GPU architectures and hope that our results motivate additional further research into advancing both hardware and software cache control techniques to close the gap with the theoretical best Belady's algorithm.