CADOSys: Cache Aware Design Space Optimization for Spatial ML Accelerators

Ruihao Li
The University of Texas at Austin
Austin, USA
liruihao@utexas.edu

Siyuan Ma
The University of Texas at Austin
Austin, USA
siyuan.ma@utexas.edu

Krishna Kavi
University of North Texas
Denton, USA
Krishna.Kavi@unt.edu

Gayatri Mehta
University of North Texas
Denton, USA
gayatri.mehta@unt.edu

Neeraja J. Yadwadkar
The University of Texas at Austin
Austin, USA
neeraja@austin.utexas.edu

Lizy K. John
The University of Texas at Austin
Austin, USA
ljohn@ece.utexas.edu

Abstract

For efficient data transfer, modern machine learning accelerators use tiered memory systems. To buffer frequently accessed data, these systems typically include a small, high-speed on-chip memory implemented either as a software-managed scratchpad or a hardware-managed cache. Scratchpad memory provides a broad design space and greater flexibility for developers. Hardware-managed cache offers a more standardized solution, as cache automates the data transfer. Although there is extensive research on the design space exploration of scratchpad-based accelerators, the design space of cache-based accelerators has distinct characteristics and remains largely unexplored. Choosing an efficient design space configuration can yield significant performance improvements compared to a suboptimal configuration in cache-based accelerators. A framework to assist in guiding the design space search for cache-based accelerators is highly desirable.

We present *CADOSys*, a system that efficiently explores the design space of cache-based accelerators given the ML model and cache microarchitecture specifications. e.g., cache capacity. *CADOSys* begins by analyzing the data access patterns and quantifying the data locality of the ML model. It then utilizes a depth-first search (DFS) algorithm that considers the data locality across all layers of the ML model to determine the design space choice for each layer. We evaluate *CADOSys* using convolutional neural networks (CNNs), deep learning recommendation models (DLRMs), and transformers. The results demonstrate that *CADOSys* achieves up to 9.12× speedup, with an average speedup of 1.82× compared to state-of-the-art design space optimization methods.

CCS Concepts

• **Computer systems organization → Systolic arrays**; • **Hardware → Datapath optimization**.

Keywords

Accelerators, Cache, Design Space Exploration

ACM Reference Format:
Ruihao Li, Siyuan Ma, Krishna Kavi, Gayatri Mehta, Neeraja J. Yadwadkar, and Lizy K. John. 2025. CADOSys: Cache Aware Design Space Optimization for Spatial ML Accelerators. In *Great Lakes Symposium on VLSI 2025 (GLSVLSI '25), June 30–July 02, 2025, New Orleans, LA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3716368.3735222

1 Introduction

Spatial accelerators are increasingly adopted in machine learning (ML) systems due to their ability to efficiently exploit parallelism and enhance data locality [3, 5, 11, 12, 28]. As illustrated in Figure 1, modern spatial accelerators have customized compute units inside each Processing Engine (PE) to provide highly parallel computations [20, 26]. Beyond computation, they feature hierarchical memory subsystems that bridge high-throughput compute units with comparatively slower off-chip memory. To mitigate the latency gap between *off-chip memory* and *Local Mem*, these accelerators integrate a shared last-level scratchpad or last-level cache (LLC) accessible by all PEs. For example, Google's TPU used local (scratchpad) memory and Meta's MTIA used a cache as their last-level on-chip memory [5, 11, 12]. The design space of scratchpad-based accelerators has been extensively explored in prior research [6, 7, 9, 13, 32], whereas the design space of cache-based accelerators remains relatively underexplored.

Scratchpad memory is commonly used in accelerators for its flexibility through software-managed control. Programmers can determine when to load and evict data from the scratchpad. However, this flexibility requires exploring a vast design space to achieve globally optimal performance. Previous studies have focused on introducing constraints to narrow the design space [6, 7, 9, 10, 13, 30, 32]. However, even within a constrained design space (which may yield sub-optimal solutions), the search process can still take days [32]. Instead of relying on scratchpad memory, modern accelerators configure the last-level shared on-chip memory as a cache [5, 15], as the cache automates data transfer between off-chip memory and computing units.

The use of caches simplifies the integration of unified compiler and ISA support alongside host CPUs, making it a more standardized solution for datacenter ML accelerators [5]. For instance, benefiting from the compatibility of caches, Meta's MTIA leverages existing software support, such as PyTorch, and achieves up to 1.5× energy savings over GPUs.

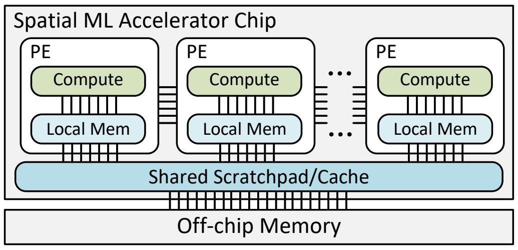


Figure 1: Overall architecture of spatial ML accelerators.

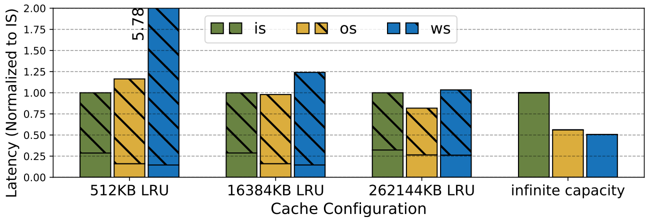


Figure 2: Comparing different dataflows on Conv1 of ResNet18. The bottom (no pattern) indicates computation time and the top (\\ pattern) indicates stall time due to cache misses. WS is the optimal dataflow for scratchpad-based accelerators since WS achieves the maximum data reuse, while IS or OS could be optimal for cache-based accelerators.

While caches enhance programmability and compatibility, fully realizing the potential of emerging accelerators necessitates thorough exploration of cache-based accelerator design spaces. Optimizing performance requires careful arrangement of computation sequences, such as loop ordering and tiling, to align with the cache architecture. For example, we find that the performance of a cache-based accelerator can vary significantly depending on the ML dataflow used (the most important aspect of the design space in cache-based accelerators, more details in § 2.2). As shown in Figure 2, we employ three kinds of dataflow [3, 26] in the spatial accelerator with different cache configurations. In the case of layer Conv1, weight stationary (WS) dataflow outperforms the input stationary (IS) dataflow and output stationary (OS) dataflow when using a cache with infinite capacity as it maximizes the PE data reuse. However, when using a 512KB least recently used (LRU) cache, the WS dataflow underperforms compared to the other two dataflows, as IS provides better data locality with limited cache capacity. Cache specifications have a significant impact on overall performance. Thus, a design space exploration framework that takes cache specifications into account is highly desirable.

In this paper, we present *CADOSys*, a system that automatically explores the design space of cache-based accelerators, using ML workloads and cache microarchitecture specifications as inputs. To be more specific, our contributions include:

- We characterize and quantify the cache capacity requirement within each layer to achieve different levels of data locality, considering data access patterns (§ 4).
- *CADOSys* uses depth-first-search (DFS) to search the design space choice of each layer, prioritizing inter- and intra-layer locality with constraints on actual cache capacity (§ 5.1).

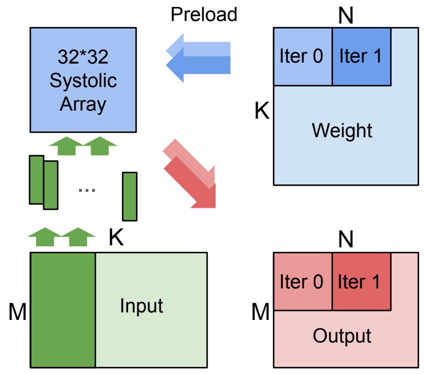


Figure 3: Mapping one convolution layer to a spatial accelerator with 32*32 PE array size using the WS dataflow.

- *CADOSys* incorporates additional factors, such as batch size and PE array size, into the performance model for design space exploration (§ 5.2).
- We evaluate *CADOSys*[1] using 6 ML models, including CNNs, DLRMs, and transformers. Results show that *CADOSys* achieves up to 9.12× and an average of 1.82× speedups compared to existing design space selection methods, specifically for scratchpad-based accelerators. *CADOSys* only has ~3% performance loss compared with brute-force search but saves hours or even days of design-space exploration time (§ 6).

2 Related Work and Motivation

This section first includes an overview of the ML accelerator scalability with the behind design search space (§ 2.1). We then discuss the design space of cache-based spatial accelerators and how it is distinguished from scratchpad-based accelerators (§ 2.2), which necessitates a cache-aware design space search model (§ 2.3).

2.1 Design Space of Spatial Accelerators

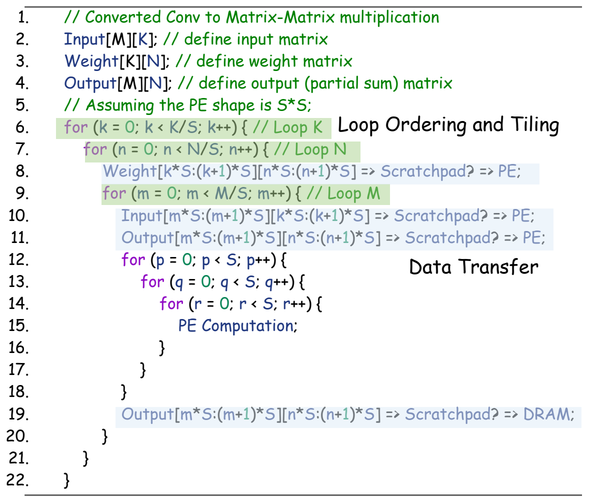
With the emergence of massive neural networks, data reuse is necessary to map ML workloads to spatial accelerators with a fixed amount of resources. To improve data reuse, modern spatial accelerators use various loop ordering and tiling strategies to improve data locality (§ 2.1.1). To further enhance performance and minimize the overhead of data movement between on-chip and off-chip memory, an efficient scheme for overlapping data transfer and computation in a pipeline is essential (§ 2.1.2). In modern accelerators, the key focus areas for design space exploration include analyzing data reuse patterns and optimizing the data transfer and compute pipeline. This presents a significant challenge, as the search space is high-dimensional and NP-hard (§ 2.1.3).

2.1.1 Loop Ordering and Tiling. To maximize data reuse, accelerators utilize efficient loop ordering and tiling strategies to map ML workloads onto spatial accelerators. Previous works categorize these loop ordering and tiling patterns into three primary types of dataflows [3, 19, 26]: OS, WS, and IS. The stationarity of a specific dataflow is defined by identifying the tensor whose elements remain in a fixed (stationary) position for the longest period during processing within PEs.

[1]Source code of *CADOSys* is available at https://github.com/UT-LCA/CADOSys/.

Table 1: Prior works in design space exploration. *CADOSys* stands out by integrating cache into its performance model.

Work	Locality	On-chip Memory	Search Speed
SmartShuttle [17]	Intra-Layer	Scratchpad	Slow
Timeloop [23]	Intra-Layer	Scratchpad	Slow
GAMMA [13]	Intra-Layer	Scratchpad	Fast
Mind Mappings [6]	Intra-Layer	Scratchpad	Fast
COSA [9]	Intra-Layer	Scratchpad	Fast
SET [2]	Inter-Layer	Scratchpad	Slow
TileFlow [32]	Inter-Layer	Scratchpad	Slow
***CADOSys* (ours)**	**Inter-Layer**	**Cache**	**Fast**



```
1.  // Converted Conv to Matrix-Matrix multiplication
2.  Input[M][K]; // define input matrix
3.  Weight[K][N]; // define weight matrix
4.  Output[M][N]; // define output (partial sum) matrix
5.  // Assuming the PE shape is S*S;
6.  for (k = 0; k < K/S; k++) { // Loop K
7.    for (n = 0; n < N/S; n++) { // Loop N
8.      Weight[k*S:(k+1)*S][n*S:(n+1)*S] => Scratchpad? => PE;
9.      for (m = 0; m < M/S; m++) { // Loop M
10.       Input[m*S:(m+1)*S][k*S:(k+1)*S] => Scratchpad? => PE;
11.       Output[m*S:(m+1)*S][n*S:(n+1)*S] => Scratchpad? => PE;
12.       for (p = 0; p < S; p++) {
13.         for (q = 0; q < S; q++) {
14.           for (r = 0; r < S; r++) {
15.             PE Computation;
16.           }
17.         }
18.       }
19.       Output[m*S:(m+1)*S][n*S:(n+1)*S] => Scratchpad? => DRAM;
20.     }
21.   }
22. }
```

Figure 4: Design space of conv layers in spatial accelerators by a loop nest representation. Scratchpad-based accelerators explore the design space of Loop Ordering, Tiling, and Data Transfer, whereas cache-based accelerators only need to focus on Loop Ordering and Tiling.

Figure 3 illustrates how to map one convolution layer (converted into matrix-to-matrix multiplication) onto a $(32 * 32)$ spatial accelerator using the WS dataflow. In this configuration, a $(32 * 32)$ portion of the $(K*N)$ filter remains stationary on the PE array, while the complete input feature map is successively passed through the $(32 * 32)$ array across multiple iterations. During each iteration, the PE array performs a matrix-to-matrix multiplication of size $(M * 32)$ and $(32 * 32)$ to generate $(32 * 32)$ output feature maps. The entire dataflow will continue until the entire $(K * N)$ filter has been processed. In WS, each element of the filter is loaded into the PE array only once, whereas each element of the input feature map typically needs to be loaded multiple times.

2.1.2 Data Transfer. Data transfer also impacts accelerator performance. To select an optimal dataflow, one should aim to maximize data reuse across PE arrays [3, 26]. However, this introduces additional complexity to the performance model.

Moving all data to the scratchpad ahead of computation is a solution to make the PE dataflow time more stable and deterministic. However, the scratchpad capacity is usually limited, making accelerators suffer from PE stall cycles waiting for data transfer [19]. One solution is to pipeline the data transfer from off-chip memory to scratchpad, scratchpad to the PE array, and PE computation. The granularity of the data transfer pipeline and the scratchpad partition between input, weight, and output tensors are the main design space factors affecting the spatial accelerator performance [9, 23].

2.1.3 Design Space Search. The design space of scratchpad-based accelerators covers multiple dimensions and is NP-hard, making it challenging to find a globally optimal solution in both intra-layer and inter-layer design spaces. Figure 4 shows the loop representation of the design space of a single convolution layer (implemented as matrix-matrix multiplication [3, 19, 26]) on a scratchpad-based accelerator. The order and tiling of loops K, N, and M (Lines 6,7,9) decide the data reuse pattern, i.e., dataflow. Another design space spec is pipelining data transfer and PE computation (Lines 8,10,11,19). For example, the computation (Lines 12-18) of the current loop can be overlapped with the data transfer of the next loop (Lines 10-11). In addition, the scratchpad partition between inputs, weights, and outputs will be another design spec. The large design space has led prior works to restrict their search to smaller subspaces to reduce search time [6, 7, 9, 13, 32], which might result in sub-optimal solutions, as shown in Table 1.

2.2 Design Space of Cache-based Accelerators

The performance of scratchpad-based spatial accelerators is sensitive to the choice of hardware mapping within the large design space. An efficient hardware mapping can achieve more than 10× energy efficiency than a random mapping [23]. However, the search cost is non-negligible, due to the NP-hard search space and long pre-silicon simulation time [9, 13, 23]. Using a hardware-managed LLC as an on-chip memory can result in a less complicated and easier-to-search design space.

In a cache-based system, the cache automates data transfers between off-chip memory and PEs. This means that read or write instructions will transfer data at the granularity of a cache line, as determined by the hardware, rather than being specified by software through design space exploration. Moreover, the hardware cache manages the partitioning of on-chip memory space among different data types, and the cache replacement policy optimizes data storage based on data locality. Therefore, the main design space to explore in cache-based accelerators is the loop ordering and tiling, i.e., ML dataflow (although there are multiple loop ordering and tiling solutions, IS, WS, and OS are the most common dataflows and have been successful in prior designs [26]). This simplification makes the exploration of the design space in cache-based accelerators more straightforward compared to scratchpad-based accelerators.

2.3 Why *CADOSys*?

Even though a hardware-managed cache automates the data transfer pipeline, the unique characteristics of ML workloads leave room for improving system performance further through design space exploration. ML dataflow stands as one of the design specifications in cache-based accelerators. Efforts are still necessary to guide the dataflow selection to improve both intra-layer and inter-layer data locality.

Intra-Layer. State-of-the-art design space exploration frameworks select IS, OS, WS dataflows, or their combinations, by balancing the trade-off between maximizing PE data reuse and minimizing off-chip memory access based on performance models [3, 9, 16, 17,

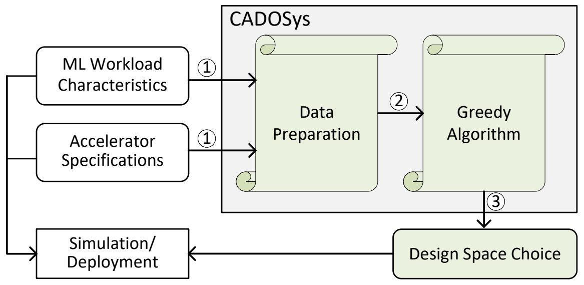


Figure 5: Overview of *CADOSys*.

23, 33]. While a dataflow that maximizes PE data reuse reduces on-chip memory data access, it does not necessarily guarantee minimal off-chip memory access. This approach is particularly efficient for scratchpad-based accelerators. In such systems, data transfers between off-chip memory, the scratchpad, and local memory are managed by software (typically through two instructions). Despite the large design space, the hit-and-miss behavior of the scratchpad becomes predictable once the data transfer amount for each pipeline loop, determined by tiling, is established.

In cache-based accelerators, the off-chip memory to scratchpad and scratchpad to Local Mem data transfer are merged as one operation (instruction). This makes the number of runtime off-chip to Local Mem data transactions non-deterministic and highly relies on the cache runtime hit-and-miss status (controlled by hardware cache replacement policies). As the example in Figure 3, the 32 elements of a single row/column of the PE data might span multiple cache lines, necessitating multiple cache line accesses instead of just one. Such data access patterns (with poor space locality) may trigger cache misses easily. Therefore, the performance model and design space selection mechanisms used for scratchpad-based accelerators cannot be directly applied to cache-based accelerators.

As illustrated in Figure 2, the impact of cache shows that different cache configurations have distinct optimal dataflows, even for the same workload, highlighting the need for cache-aware dataflow optimization. Furthermore, we observed a significant reduction in stall cycles for the WS dataflow when the cache capacity increased from 512KB to 1024KB. Therefore, before selecting a specific dataflow, it is ncessary to quantify the data access requirements of the dataflow and compare them with the available cache capacity.

Inter-Layer. Scratchpad-based accelerators employ layer fusion to enhance data locality by minimizing data transfers between off-chip memory and on-chip (scratchpad) memory [32]. In contrast, cache-based accelerators rely on cache replacement policies, rather than software-controlled instructions, to manage data transfers between off-chip memory and the LLC. Consequently, layers are automatically fused based on cache replacement policies that retain tensors across multiple layers. Optimizing the retention of tensors with cross-layer locality in the cache is a key area for improving inter-layer data locality.

3 *CADOSys* Overview

This section gives an overview of *CADOSys*. *CADOSys* takes ML workload characteristics and accelerator hardware specifications as inputs and generates the dataflow of each layer. We break down the entire process into three steps (shown in Figure 5).

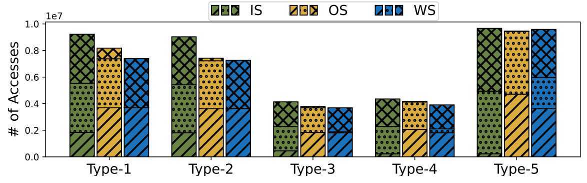


Figure 6: Amount of data access of 5 types of conv layers in ResNet18. From bottom (//) to top (xx), each pattern indicates the amount of inputs, weights, and outputs data access.

- *CADOSys* characterizes and quantifies the data reuse and cache capacity requirement of each layer (§ 4). This step takes ML workload information (size of inputs, weights, and outputs of each layer) and accelerator hardware specifications (PE array size and cache capacity) as inputs.
- *CADOSys* uses a DFS-based algorithm to generate the design space of each layer (§ 5). The search algorithm incorporates cache effects into its performance model and seeks to enhance data reuse at both intra-layer and inter-layer levels. Moreover, *CADOSys* considers various factors, such as ML model and accelerator scalability to ensure its robustness.
- The output of *CADOSys* (design space choice of each layer) can be used as the input either for pre-silicon architecture simulation, or post-silicon deployment.

4 Data Preparation for *CADOSys*

In this section, we prepare the necessary data for building *CADOSys*, by characterizing the data reuse (§ 4.1) and quantifying the cache capacity requirement (§ 4.2) of each design space choice.

4.1 Characterizing Data Reuse

We characterize the cache data reuse of each dataflow. ML accelerators make use of data locality by reusing either inputs, weights, or outputs through IS, WS, and OS dataflow, respectively. To provide a clear and intuitive explanation of this, we show the amount of data access of ResNet18 in Figure 6. Though different dataflows may have similar total amounts of cache data access, the difference in the breakdown of inputs, weights, and outputs can result in differences in cache data reuse (i.e., how many times each element is accessed from the cache) across different dataflows. This difference in cache data reuse can result in different cache capacity requirements for different dataflows. In this part, we analyze both intra-layer and inter-layer cache data reuse.

Intra-Layer. In ML workloads, different dataflows share different intra-layer data locality, resulting in a difference in cache capacity requirement. As the size of inputs, weights, and outputs are usually different in different ML workloads [1], different dataflows show distinct intra-layer cache data reuse (The variation in the amount of data access shown in Figure 6 across different dataflows reflects distinct data reuse patterns).

Inter-Layer. In addition to intra-layer, inter-layer cache data reuse is also important in ML workloads. For instance, the output of one layer may be the input for a specified subsequent layer, while weight parameters are not shared across layers. ML accelerator compilers, e.g., Glow [24], divide data into two categories, *placeholder nodes* and *constant nodes* to indicate the difference with respect to cache

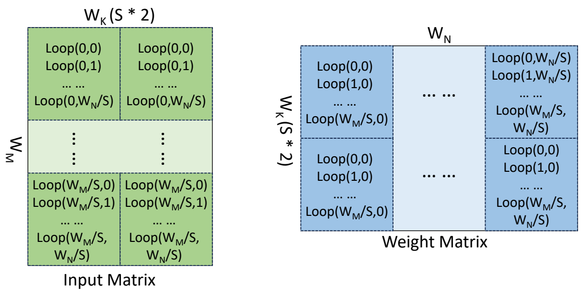


Figure 7: Data reuse in OS dataflow. Row-major access finishes computation of all columns for the first S rows of the output matrix before proceeding to the next S rows, where S represents the size of the PE array ($W_k = S*2$ in this example).

data reuse. In the inter-layer scenario, the outputs of a layer can achieve even higher locality, as long as it is used as inputs for subsequent layers. Such inter-layer cache data reuse also impacts our choice of dataflow in *CADOSys*. *CADOSys* prioritizes the dataflow that achieves inter-layer cache data reuse.

4.2 Requirement on Cache Capacity

After characterizing the data reuse of each dataflow, we calculate the cache capacity requirement of each dataflow. To achieve the best performance of each dataflow, the ideal solution is to cache all *placeholder nodes* and *constant nodes* as long as locality exists. For example, to achieve a full locality in the OS dataflow, either the entire input matrix or the entire weight matrix has to be cached based on the data access order. We assume a single PE array with grid size $S * S$. PE arrays implement convolution operations as general matrix-to-matrix multiplications [3, 19, 26]. We assume the input matrix size is $W_M * W_K$, the weight matrix size is $W_K * W_N$, and the output matrix size is $W_M * W_N$. Based on row-major data access order in Figure 7, S rows of the input matrix have to pass through all columns of the weight matrix to generate S rows of the output matrix. Under this scenario, the entire $W_K * W_N$ matrix has to be cached to achieve full locality.

With the increasing complexity of AI models, which can have billions of parameters as seen in large language models [31], it is impractical to preload all parameters into MB-level caches. Instead of achieving full locality, aiming for partial locality is an alternative solution when dealing with limited cache capacity. As the same example in Figure 7, caching $S * W_K$ of the input matrix enables the OS dataflow to achieve partial locality. When computing S rows of the output, all elements from the weight matrix must be retrieved from off-chip memory, while the S rows of inputs are already cached in the first loop. In the case of column-major data access, the locality pattern is the reverse of row-major access; therefore, $S * W_K$ of the weight matrix should be cached instead. For IS and WS dataflows, the minimum cache capacities to achieve partial locality are $S * W_N$ and $S * W_M$, respectively. The cache capacity requirements for achieving both full and partial locality across all three dataflows are summarized in Table 2.

5 *CADOSys*

Having determined the cache capacity requirement, we then demonstrate how *CADOSys* selects its design parameters. By employing a DFS-based algorithm, *CADOSys* efficiently explores the design space (§ 5.1). We also extend *CADOSys* to accommodate different batch sizes and various PE configurations (§ 5.2).

Table 2: Cache capacity requirement of different dataflow (row/col stands for row/col major access).

Order + Locality	IS	OS	WS
Row + Partial	$S * W_N$	$S * W_K$	$S * W_M$
Row + Full	$(W_K + S) * W_N$	$(W_N + S) * W_K$	$(W_K + S) * W_M$
Col + Partial	$S * W_N$	$S * W_K$	$S * W_M$
Col + Full	$(W_M + S) * W_N$	$(W_M + S) * W_K$	$(W_N + S) * W_M$

Algorithm 1: Cache aware design space selection.

```
Input  : Accelerator PE shape $S * S$, cache capacity $C$;
         Shapes of $N$ layers ($W_M[N]$, $W_K[N]$, $W_N[N]$);
         dependency Graph $G$;
Output : Dataflow of the $N$ layers $D[N]$;
         Data access order of the $N$ layers $O[N]$;
Initialize placeholder node priority queue $P$;
Initialize current available cache $C_A = C$;
foreach layer $n \in N$ do
    $D[n]$ = Dataflow which maximizes the PE data reuse;
    if $C_A$ is enough to make $D[n]$ achieve full locality then
        if Inputs of layer $n$ in $P$ then
            match $O[n]$ with the previous layer accordingly;
        else if $D[n]$ == Weight Stationary then
            $O[n]$ = ($W_K[n] < W_N[n]$) ? row : col;
        else if $D[n]$ == Input Stationary then
            $O[n]$ = ($W_K[n] < W_M[n]$) ? row : col;
        else if $D[n]$ == Output Stationary then
            $O[n]$ = ($W_N[n] < W_M[n]$) ? row : col;
    else
        if $P$ is not empty then
            Pop the first node from $P$ and update $C_A$;
            goto line 5;
        Recover $P$;
        if ($W_M[n]$ * $S$) <= $C_A$ then
            $D[n]$ = Weight Stationary;
        else if ($W_N[n]$ * $S$) <= $C_A$ then
            $D[n]$ = Input Stationary;
        else if ($W_K[n]$ * $S$) <= $C_A$ then
            $D[n]$ = Output Stationary;
        else
            if $P$ is not empty then
                Pop the first node from $P$ and update $C_A$;
                goto line 19;
        end
        $O[n]$ = row;
    end
    Release nodes from $P$ that will not be used in the future and update $C_A$;
    Insert nodes to $P$ that will be used in the future and update $C_A$;
end
```

5.1 Searching Algorithm

We propose a DFS-based search algorithm (Algorithm 1) to find the data reuse pattern that maximizes the PE data reuse as well as achieve full or partial intra-layer data locality. We model the end-to-end execution of an ML network as an execution tree, where each layer offers 6 options (3 dataflows with 2 data access orders for each dataflow). Each node in the execution graph branches into 6 leaves, representing 6 choices. Since the design decisions at one layer can affect the overall cache behavior, backtracking in a greedy-based manner[2] is essential to account for these global effects. This makes a DFS-like algorithm well-suited for this type of scenario.

CADOSys first identifies the dataflow that maximizes the PE data reuse, i.e., which dataflow achieves better PE data locality [3, 26] (lines 4). *CADOSys* then checks if the selected dataflow can achieve the full data locality based on the cache capacity requirement model in § 4.2 (lines 4-5). Regarding data access order, *CADOSys* will first pick the one that can achieve inter-layer data locality (lines 6-7). Otherwise, it will select the access order requiring less cache capacity than the alternatives (lines 8-13). If full locality cannot be achieved, *CADOSys* proceeds to check the feasibility of achieving partial locality (lines 14-31). The guideline is to prioritize cache *placeholder nodes* other than *constant nodes* since these nodes could be revisited in subsequent layers (lines 19-24). For instance, *CADOSys* prioritizes IS dataflow as input nodes are less likely to be accessed in subsequent layers than output nodes. In the worst-case scenario where the cache capacity is too constrained to achieve even partial locality, *CADOSys* selects the dataflow that maximizes PE data reuse.

After finishing each layer, *CADOSys* will update the *placeholder nodes* priority queue and cache capacity information for checking inter-layer data locality, e.g., outputs of the previous layer act as inputs for the next layer. If the *placeholder node* is accessed in subsequent layers, *CADOSys* will consider this node already occupying the cache (lines 32-33). If the cache capacity is insufficient to achieve inter-layer data locality, *CADOSys* will backtrack (similar to DFS) to maximize the intra-layer data full locality (lines 15-17) or partial locality (lines 26-28). *CADOSys* pops each element of the *placeholder nodes* priority queue as the backtracking process. The priority of each node is determined by its most recent anticipated future usage (node dependency graph is part of *CADOSys* input).

5.2 Scalability of *CADOSys*

With the emergence of massive neural networks, scalability is an important metric for evaluating the performance of ML accelerators. Algorithm 1 automates the process of design space selection under the context of single PE array and single input batch size. To scale the performance of accelerators [20, 21], datacenter ML accelerators are equipped with multiple PE arrays, and ML workloads are processed in batches. We enhance the Algorithm 1 to make it scale for the multi-PE and multi-batch scenarios.

Multi-Batch. In the multi-batch scenario, the amount of data access of both input and output feature maps of each layer scales according to the batch sizes. Assuming a batch size B, the input scales from $W_M * W_K$ to $B * W_M * W_K$ and the output scales from $W_M * W_N$ to $B * W_M * W_N$. The filter size remains the same $W_K * W_N$, making the cache requirement of the IS and OS dataflow remain the same as the single batch. For the WS dataflow, instead of a single input, all inputs within the same batch pass through the stationed filter. Therefore, the cache requirement of the WS dataflow scales based on the batch size B. We listed the scale cache capacity requirement of WS in Table 3.

Multi-PE. In the multi-PE scenario, both data parallelism and model parallelism can be used [20, 21]. For example, in WS (also applies to IS and OS), assuming the $1 * 1$ PE grid is scaled to a $n * n$ grid, we can have only one copy of the filter and distribute it into all $n*n$ PEs (data parallelism, left side of Figure 8) or have $n*n$ different parts of filter on each PE (model parallelism, right side of Figure 8). When using data parallelism, a *one-to-all* and *all-to-one* operation is required before and after processing the stationed data [20, 21], but the data locality remains the same as the single-PE scenario since data parallelism does not introduce additional non-stationary data. On the other hand, model parallelism introduces extra non-stationary data. The cache capacity requirement of each dataflow needs to scale accordingly (S in Table 2 and Algorithm 1 needs to scale to $n * S$).

Table 3: Multi-batch cache capacity requirement of WS.

Row + Partial	$S * B * W_M$	**Col + Partial**	$S * B * W_M$
Row + Full	$(W_K + S) * B * W_M$	**Col + Full**	$(W_K + S) * B * W_M$

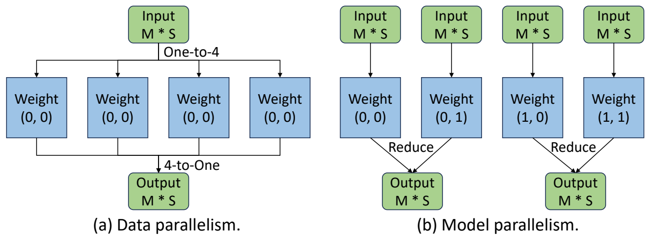


Figure 8: Data and Model parallelism in a 2*2 PE grid.

6 Evaluation

We demonstrate *CADOSys*'s effectiveness by answering the following questions through experimental evaluation:

- Does *CADOSys* perform better than state-of-the-art design space exploration solutions (§ 6.2)?
- Is *CADOSys* better than brute force search (§ 6.3)?
- Does *CADOSys* scale well for multi-batch and multi-PE (§ 6.4)?

6.1 Experimental Setup

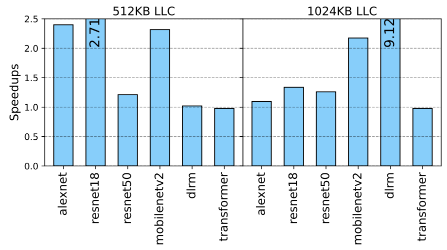
We build *CADOSys* following the model of Scale-sim [26, 27], which is one of the state-of-the-art simulators for modeling the performance of spatial accelerators. We set the LLC associativity as 16 with a 64-byte cache line and an LRU replacement policy.

We evaluate *CADOSys* using diverse workloads. We include representative CNNs (AlexNet, ResNet18, ResNet50, and MobileNetV2), which are widely used for image processing and span almost a decade of progress in CNN design, showcasing computational patterns that are broadly representative [7, 25]. We also evaluate DLRMs (deep learning recommendation models), which could benefit even more from a cache-based accelerator due to their sparse computations [5, 22]. In addition, we also include Transformers, the fundamental building blocks of large language models [4, 29].

We adopt a mapping scheme aligned with modern scratchpad-based optimization methodologies, such as Timeloop [23], GAMMA [13], and COSA [9], to maximize data reuse at the PE level and minimize the amount of unique (cache) data access as the baseline[3]. However,

[2]Enumerating all dataflows to achieve global inter-layer data locality is impractical for large networks due to the vast search space [8, 32].

[3]Assuming an ideal cache (i.e., sufficient data capacity), Timeloop [23], GEMMA [13], and CoSA [9] converge on the same design space selection.


(a) Speedups of using 512/1024KB LLC.

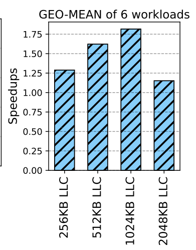
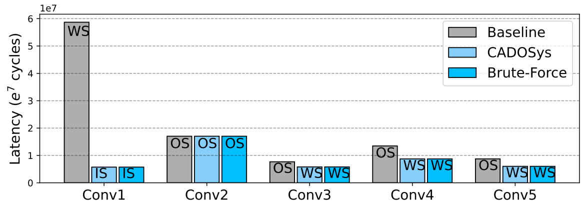

(b) GEO-MEAN.

Figure 9: Speedups of *CADOSys* in different cache sizes.


(a) Using a 512KB LLC.

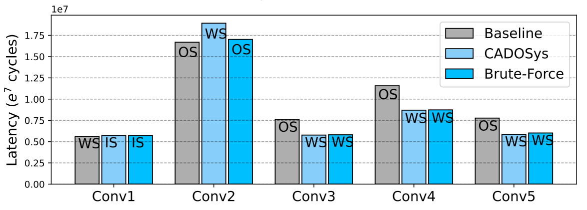

(b) Using a 1024KB LLC.

Figure 10: Layer-wise performance comparison of AlexNet between baseline, *CADOSys*, and brute-force search. *CADOSys* performs better than the baseline and almost the same as a brute-force search (saves search time significantly).

this baseline does not incorporate cache microarchitecture specifications when modeling data access patterns and on-chip memory capacity requirements.

6.2 How well does *CADOSys* perform?

We evaluate the efficacy of *CADOSys* by comparing it with the state-of-the-art design space selection scheme [3, 26] which picks the dataflow that achieves the maximum PE data reuse of each layer. We set the PE array size as $8 * 8$, with an LLC with the capacity of 256KB, 512KB, 1024KB, and 2048KB. As shown in Figure 9, *CADOSys* achieves a 1.29×, 1.62×, 1.82×, and 1.15× (geometric mean of the 6 workloads) speedup over the baseline on an LLC with a capacity of 256KB, 512KB, 1024KB, and 2048KB, respectively.

Among the 4 cache sizes, *CADOSys* demonstrates superior performance with 512KB and 1024KB LLCs. In scenarios with limited cache capacity, most layers cannot achieve full data locality across the three dataflows, highlighting the effectiveness of *CADOSys*. Conversely, with larger caches, all dataflows attain full locality, reducing the impact of *CADOSys*.

Among the 6 workloads, *CADOSys* achieves higher speedups in CNNs and DLRMs than Transformers. This is because performance differences between dataflows arise when there are disparities in the sizes of input, weight, and output matrices. For instance, in CNNs, the first few layers often have inputs larger than weights (i.e., $W_M > W_K$ in Table 2), while the last few layers have weights larger than inputs. Such imbalances in CNNs and DLRMs makes *CADOSys* more effective.

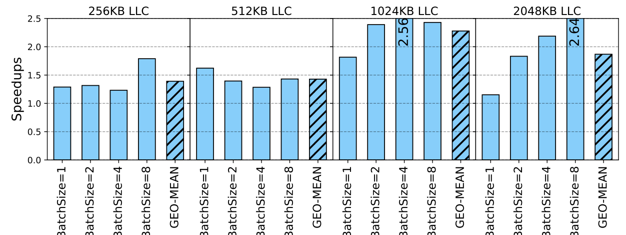


Figure 11: Speedups (GEO-MEAN of 6 workloads) on multi-batch. *CADOSys* is robust on all batch sizes.

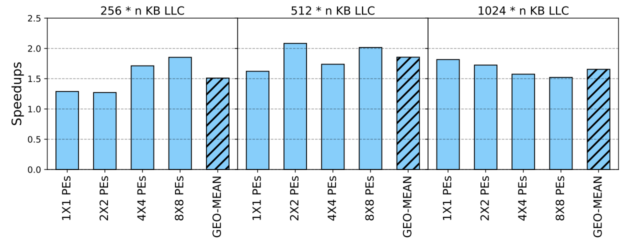


Figure 12: Speedups (GEO-MEAN of 6 workloads) on multi-PE. *CADOSys* is robust on all PE sizes.

6.3 How does *CADOSys* outperform brute force?

In this section, we compare *CADOSys* with brute-force search. *CADOSys* is able to find the dataflow combination close to brute-force but saves searching time significantly.

We present the layer-wise performance breakdown of AlexNet using a 512KB LLC (Figure 10a) and a 1024KB LLC (Figure 10b). The brute-force method explores all 3^5 dataflow combinations across the 5 convolutional layers in AlexNet.

When using a 512KB LLC, *CADOSys* is consistent with the brute-force result. *CADOSys* improves the performance on layer Conv1 significantly by using the IS dataflow instead of WS dataflow. When considering the data access pattern, the available cache capacity is insufficient to enable WS dataflow to achieve a full locality. The input matrix size is much larger than the weight matrix size. When using a 1024KB LLC, *CADOSys* only has a 3.1% performance loss compared with a brute-force search. The reason is the searching algorithm 1 (line 19-24) in *CADOSys* gives a higher priority to WS, as its potential to offer better data locality for larger batch sizes.

While *CADOSys* cannot guarantee a globally optimal design choice, it significantly reduces search time compared to a brute-force approach. For example, obtaining the optimal dataflow for AlexNet using brute force requires simulating all 243 combinations, with each simulation taking approximately 76 seconds (on a Neoverse-N1 server we used). In contrast, *CADOSys* eliminates the need for an end-to-end simulation before making the design choice, saving around 5 hours of simulation time. For larger networks like ResNet18, with its 3^{18} possible dataflow combinations, brute-force search becomes impractical due to the extensive simulation time required. In addition to pre-silicon simulation-based systems, *CADOSys* also reduces search time in post-silicon scenarios, where the routing and packaging process for generating an accelerator binary can be time-consuming [14, 18]. This makes *CADOSys* more efficient than brute-force search.

6.4 Does *CADOSys* scale well?

In addition to single-batch and single-PE, we also assess the scalability of *CADOSys* in multi-batch and multi-PE scenarios.

6.4.1 Multi-batch. In the multi-batch scenario, we evaluate the performance of *CADOSys* on batch sizes 1, 2, 4, and 8. The geometric mean speedups of the six workloads are shown in Figure 11. *CADOSys* achieves 1.39×, 1.43×, 2.28×, and 1.87× (geometric mean of the 4 batch sizes) speedups over the baseline on an LLC with a capacity of 256KB, 512KB, 1024KB, and 2048KB, respectively. Unlike single-batch, *CADOSys* is effective when using a 2048KB cache under the multi-batch scenario (compared with the 2048KB result in Figure 9). The reason is that as the batch size increases, the cache capacity requirement of each dataflow will also increase. As a result, the dataflow which maximizes the PE data reuse may not be able to achieve the full locality.

6.4.2 Multi-PE. In the multi-PE scenario, we evaluate the performance of *CADOSys* on the accelerator with 1x1, 2x2, 4x4, and 8x8 PEs. We scale the batch size and LLC capacity *n* times (assuming *n*x*n* PEs) compared with the single-PE scenario. The geometric mean speedups of the six workloads are shown in Figure 12. *CADOSys* achieves 1.51×, 1.86×, and 1.66× (geometric mean of the 4 PE array sizes) speedups over the baseline on an LLC with a capacity of 256**n* KB, 512**n* KB, and 1024**n* KB. We observe the efficacy of *CADOSys* does not grow linearly as the number of PEs (and batch size) scales. The reason is that the cache requirement of each dataflow does not increase linearly as the number of PEs increases based on the equations listed in Table 2. Fortunately, *CADOSys* can identify the margin of cache capacity for all batch sizes and the number of PEs.

7 Conclusion

Accelerators are generally categorized into two types based on their memory management: scratchpad-based and cache-based. Extensive research has been conducted on optimizing scratchpad-based accelerators. However, cache-based accelerators are emerging, and there is a lack of comprehensive optimization strategies for them. In this paper, we introduce *CADOSys*, a framework to optimize cache-based accelerators.

CADOSys takes both ML workload characteristics and cache microarchitecture specifications as inputs to quantify cache capacity requirements. It employs a DFS algorithm to determine the optimal design space configuration for each layer of the ML model. Results show that *CADOSys* achieves an average speedup of 1.82× compared to state-of-the-art scratchpad-based optimization methods. While *CADOSys* incurs only a ~3% performance loss compared to brute-force search, it significantly reduces simulation time by hours or even days. Additionally, *CADOSys* consistently improves performance for different batch and PE array sizes. The design space generated by *CADOSys* can be used in pre-silicon simulations or directly deployed in post-silicon accelerators.

Acknowledgement: We thank the anonymous reviewers for their helpful feedback. This research was supported in part by NSF grant numbers #2326894 and #2425655, the Texas ECE junior faculty start-up fund, UT iMAGiNE consortium, an award from the UT Machine Learning Lab (MLL), the AMD Chair Endowment, the Cisco Research Award, and the Amazon Research Award. The authors would also like to acknowledge the computing servers donated by Ampere and Nvidia, and the computing resources provided by the Texas Advanced Computing Center (TACC).

References

[1] Manoj Alwani, et al. 2016. Fused-layer CNN accelerators. In *MICRO*.
[2] Jingwei Cai, et al. 2023. Inter-layer scheduling space definition and exploration for tiled accelerators. In *ISCA*.
[3] Yu-Hsin Chen, et al. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ISCA*.
[4] Abhimanyu Dubey, et al. 2024. The llama 3 herd of models. *arXiv:2407.21783* (2024).
[5] Amin Firoozshahian, et al. 2023. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. In *ISCA*.
[6] Kartik Hegde, et al. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *ASPLOS*.
[7] Charles Hong, et al. 2023. Dosa: Differentiable model-based one-loop search for dnn accelerators. In *MICRO*.
[8] Qijing Huang, et al. 2022. Learning a continuous and reconstructible latent space for hardware accelerator design. In *ISPASS*.
[9] Qijing Huang, et al. 2021. Cosa: Scheduling by constrained optimization for spatial accelerators. In *ISCA*.
[10] Qijing Huang, et al. 2024. Mind the Gap: Attainable Data Movement and Operational Intensity Bounds for Tensor Algorithms. In *ISCA*.
[11] Norm Jouppi, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *ISCA*.
[12] Norman P Jouppi, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *ISCA*.
[13] Sheng-Chun Kao et al. 2020. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *ICCAD*.
[14] Sadaf Khan, et al. 2024. DeepSeq: Deep Sequential Circuit Learning. In *DATE*.
[15] Joyjit Kundu, et al. 2024. A System Level Performance Evaluation for Superconducting Digital Systems. *arXiv:2411.08645* (2024).
[16] Hyoukjun Kwon, et al. 2021. Heterogeneous dataflow accelerators for multi-DNN workloads. In *HPCA*.
[17] Jiajun Li, et al. 2018. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *DATE*.
[18] Zhe Lin, et al. 2022. Powergear: Early-stage power estimation in FPGA HLS via heterogeneous edge-centric GNNs. In *DATE*.
[19] Sparsh Mittal. 2017. A survey of techniques for cache partitioning in multicore processors. *CSUR* (2017).
[20] Dheevatsa Mudigere, et al. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *ISCA*.
[21] Maxim Naumov, et al. 2020. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv:2003.09518* (2020).
[22] Maxim Naumov, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv:1906.00091* (2019).
[23] Angshuman Parashar, et al. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *ISPASS*.
[24] Nadav Rotem, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv:1805.00907* (2018).
[25] Chirag Sakhuja, et al. 2023. Leveraging domain information for the efficient automated design of deep learning accelerators. In *HPCA*.
[26] Ananda Samajdar, et al. 2020. A systematic methodology for characterizing scalability of dnn accelerators using scale-sim. In *ISPASS*.
[27] Ananda Samajdar, et al. 2018. Scale-sim: Systolic cnn accelerator simulator. *arXiv:1811.02883* (2018).
[28] Vivienne Sze, et al. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* (2017).
[29] Ashish Vaswani, et al. 2017. Attention is all you need. *NeurIPS* (2017).
[30] Zi Yu Xue, et al. 2023. Tailors: Accelerating Sparse Tensor Algebra by Overbooking Buffer Capacity. In *MICRO*.
[31] Wayne Xin Zhao, et al. 2023. A survey of large language models. *arXiv:2303.18223* (2023).
[32] Size Zheng, et al. 2023. TileFlow: A Framework for Modeling Fusion Dataflow via Tree-based Analysis. In *MICRO*.
[33] Shixuan Zheng, et al. 2022. Atomic dataflow based graph-level workload orchestration for scalable DNN accelerators. In *HPCA*.