A Comprehensive Automated Exploration Framework for Systolic Array Designs

Suhail Basalama, Jie Wang, Jason Cong
University of California - Los Angeles
{*basalama, jiewang, cong*}*@cs.ucla.edu*

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10248016

***Abstract*—Many researchers studying the performance tuning of systolic arrays have based their works on oversimplified assumptions like considering only divisors for loop tiling or pruning based on off-chip data communication to reduce the design space. In this paper, we present a comprehensive design space exploration tool named Odyssey for systolic array optimization. Odyssey results show that limiting tiling factors to only divisors of the problem size can cause up to 39% performance loss, and pruning the design space based on off-chip data movement can miss optimal designs. We tested Odyssey using various matrix multiplication and convolution kernels and validated the results with FPGA implementations.**

1. Introduction

Performance optimization for a given class of microarchitectures, also called performance tuning, has long been an important topic given the complexity of hardware systems and applications. This issue is intensified on domain-specific architectures (DSA), which grant designers explicit control over the software stack and hardware architecture, opening up a vast design space to explore.

This paper focuses on the performance tuning of systolic arrays. A complete design space of systolic arrays contains multiple dimensions, such as the selection of dataflows, loop permutation, and loop tiling. These factors impact the final design performance in an intertwined manner and compose a vast design space that is intractable to explore exhaustively, especially for large problem sizes.

Many previous works have attempted this challenging task by looking into different dimensions of the design space and proposing various auto-tuning methods [17, 9, 8, 10, 5, 3, 14, 19]. However, after a thorough examination of the previous works, we identified several limitations that need to be addressed.

Limitation 1: Incomplete coverage of the design space. When selecting the tiling factors, many previous works only considered problem size divisors to reduce the design space [5, 19, 3, 8]. Such a simplification, though, could lead to inferior designs. We compare the throughput and DSP usage of best systolic arrays that are found by limiting the tiling factors to 1) divisors only and 2) both divisors and non-divisors for a $1024 \times 1024 \times 1024$ matrix multiplication (MM). Restricting tiling factors to divisors leads to a 39% performance loss. With the limited design space, the divisor-only design fails to fully exploit the on-chip resource, with a mere 60% DSP usage.

Limitation 2: Inaccurate performance modeling. An inaccurate performance model could also hurt the quality of search results. For example, the previous work TENET [14] estimated the design latency as the maximum of computation and communication latency. This model overlooks the prologue/epilogue phases when loading the first tile of data and writing out the final results. For the same MM problem, the best design identified by using the simplified maximum-based performance model results in a 9% slower design than the best design uncovered by a more accurate performance model that accounts for prologue/epilogue latencies.

Limitation 3: Inefficient search methods and imperfect pruning heuristics. When searching the design space, many previous works adopted pruning-based exhaustive search which may not scale to large-sized problems [5, 19, 3, 14]. To make matters worse, several works chose imperfect pruning heuristics, failing to cover optimal designs. For example, the previous work Marvel [3] pruned the design space based on the off-chip data communication and applied an exhaustive search in the pruned sub-space. However, an optimal design needs to balance both the data communication and computation and does not necessarily minimize the off-chip memory accesses. For the same MM example, we found that the design with the optimal performance has a $3.5\times$ more off-chip data movement and a $1.9\times$ higher throughput than the design with minimum off-chip data movement.

All of these limitations affect the quality of search results and further impact the architectural decisions that designers derive based on these results. To overcome this challenge, in this paper, we propose a new automatic design space exploration framework for systolic arrays, *Odyssey*[1], with the following contributions:

1) A comprehensive design space construction and accurate performance modeling for systolic arrays
2) Two design space explorers:
    a) A hybrid method using mathematical programming and a genetic algorithm
    b) A more accurate method using a novel padding-based search algorithm
3) A fully automated and open-source framework

2. Background

2.1. Automatic Systolic Array Generation

Automatic systolic array generation is an important research topic given the high performance of the systolic array architecture and the complexities of the designing process [12, 15]. The recent work, AutoSA [18], reported the best performance results in this field.

AutoSA takes in a C program as the input and applies a sequence of program transformations on this program to build and optimize systolic arrays.

With comprehensive coverage of hardware optimization techniques, AutoSA generates high-performance systolic arrays with

1. Odyssey is abbreviated from AUtomatic DEsign space exploration for SYstolic arrays.

comparable or better performance than previous works [18]. However, such a vast design also poses significant challenges to performance tuning. As an example, considering all the available tuning options, the size of design space bloats to $O(2^{40})$ for a $1024 \times 1024 \times 1024$ MM. This challenge has motivated us to develop Odyssey which provides efficient auto-tuning support to explore such a design space.

2.2. Previous Search Methods

The vast design space makes it infeasible to explore with an exhaustive search. Prior works have offered various approaches to tackle this challenge as summarized in Table 1.

An ideal performance tuning framework should achieve: 1) a *comprehensive coverage* and *accurate modeling* of the design space, and 2) *efficient* search methods to explore the design space. The failure in either of the two targets will impact the quality of search results, as well as the architecture conclusions derived from these results. Unfortunately, regardless of the plethora of past studies, we observe no prior work that reached a balance between these two goals. This situation has motivated us to tackle this challenge.

TABLE 1: Comparison between different architecture performance tuning frameworks

	Design Space	Performance Models		Search Methods	On-board
	Non-Divisors	Prologue/ Epilogue	Generation		Validation
Timeloop [17]	N	N	Manual	Exhaustive w/ Pruning Random Search	N
dMazeRunner [5]	N	N	Manual	Exhaustive w/ Pruning	N
Interstellar [19]	N	N/A	Manual	Exhaustive w/ Pruning	N
Marvel [3]	N	Y	Manual	Mathematical Programming Exhaustive w/ Pruning	N
ConfuciuX [9]	N/A	Y	Manual	RL Evolutionary Search	N
CoSA [8]	N	Y	Manual	Mathematical Programming	GPU
TENET [14]	N/A	N	Manual	Exhaustive w/ Pruning	N
Odyssey	Y	Y	Auto	Mathematical Programming Evolutionary Search Padding-based Algorithm	FPGA

3. Odyssey Design Space Construction

We consider all three dimensions of the design space of systolic arrays: dataflows, loop permutation, and loop tiling (array partitioning, latency hiding, and SIMD vectorization factors[2]).

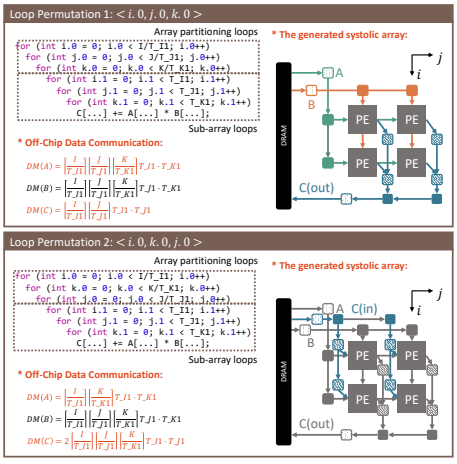

Figure 1: Two loop permutation examples for MM

Dataflows: Previous works [19, 11] utilized the term *dataflow* to identify different array topologies and execution patterns, which are equivalent to different space-time mappings within the scope of systolic arrays. In the rest of the paper, we use *dataflow* to refer to different space-time mappings. We annotate each dataflow in the format of $[i, j]$ that marks the selected ***space*** loops for this array (see Table 2).

TABLE 2: Parameters set to construct the whole design space for MM and convolution

Application	MM	CNN
Dataflows	[i], [j], [k], [i,j], [i,k], [j,k]	[o], [h], [w], [i], [o,h], [o,w], [o,i], [h,w], [h,i], [w,i]
Loop Permutation	<[i,j],k>, <[j,k],i>, <[i,k], j>	<[o,h,w],[i,p,q]>, <[o,i,p,q],[h,w]>, <[i,h,w,p,q],o>
# of Unique Designs	6 dataflows × 3 permutations = 18	10 dataflows × 3 permutations = 30
Tiling Factors per Design	$T_{i1}, T_{j1}, T_{k1}, T_{i2}, T_{j2}, T_{k2}$	$T_{i1}, T_{o1}, T_{h1}, T_{w1}, T_{i2}, T_{o2}, T_{h2}, T_{w2}$

Loop Permutation: Odyssey explores different loop permutation orderings in the array partitioning. Different loop orderings may lead to various array architectures. AutoSA enumerates all the loop permutation orderings. For MM, there are $3! = 6$ different orderings to consider. The number grows larger for more complicated applications like 2-D convolution. The six-level nested loops lead to $6! = 720$ different loop orderings. However, as pointed out by previous works [13, 5], among all the loop orderings, many of them are dominated by a few orderings in performance, thus can be safely pruned without leaving out the optimal points. Next, we show that with proper pruning, we can reduce the number of loop orderings to consider for both MM and CNN to only 3.

We consider resource usage and latency when assessing the design performance. Different loop orderings will impact the structure of the I/O network, resulting in different resource usage. For example, in MM, after hoisting the loop $k.0$ from the innermost position of the array partitioning loop band, additional I/O modules for transferring the intermediate results of matrix C are added, increasing the total resource usage. The key takeaway is: *By placing loops that carry the flow dependences innermost in the array partitioning loop band, intermediate data are accumulated on-chip, eliminating the resource overheads brought by the additional I/O modules.*

Latency-wise, different loop orderings will affect off-chip data communication. We compute the off-chip data movement for the two example designs in Figure 1. For the first ordering $< i.0, j.0, k.0 >$, final results of matrix C are only drained out at the last iteration of loop $k.0$, leading to a total amount of data movement as:

$$DM(C) = \lceil \frac{I}{T\_I1} \rceil \lceil \frac{J}{T\_J1} \rceil T\_I1 \cdot T\_J1 \quad (1)$$

As for the second ordering $< i.0, k.0, j.0 >$, the intermediate results of matrix C are swapped off-chip at each loop iteration of $j.0$. We compute the total data movement of matrix C as:

$$DM(C) = 2\lceil \frac{I}{T\_I1} \rceil \lceil \frac{K}{T\_K1} \rceil \lceil \frac{J}{T\_J1} \rceil T\_I1 \cdot T\_J1 \quad (2)$$

To take into account both the inbound and outbound traffic of matrix C, we multiply the factor 2. Compared to the first ordering, the second ordering introduces a higher amount of data movement for matrix C. For communication-bound designs, this could lead to a longer latency. In addition to the loops that carry the flow dependence, loops that carry the read dependence will impact the data communication as well. We use matrix A as an example. As shown in Figure 1, with the loop ordering $< i.0, j.0, k.0 >$, at each array partition, we load new array tiles with a size of $T\_I1 \times T\_K1$ from array A. In comparison, with the loop ordering $< i.0, k.0, j.0 >$, data of matrix A are reused along the loop $j.0$. New data tiles are only loaded at each new loop iteration of loop

2. Refer to the AutoSA paper [18] for an explanation of these terms.

$k.0$, reducing the data communication for matrix A compared to the first ordering. Detailed equations of data movement for matrix A can be found in Figure 1. The key takeaway is: *By placing array partitioning loops that carry the flow/read dependences innermost, data are reused on-chip, reducing the off-chip data communication and design latency.*

Putting it all together, we have a complete picture of the pruning strategy.

Theorem 3.1. *Given a program that can be mapped to systolic arrays, let $RL(r)$ be the set of array partitioning loops that carry the read/flow dependences associated with the array reference r, and $NRL(r)$ be the rest of the loops in the array partitioning loop band, the set of unique loop orderings O can be obtained as the union of loop orderings in the form of $< NRL(r), RL(r) >$ for each array reference r. All the other loop orderings are dominated by O in terms of resource usage and latency. Note that $RL(r)$ could be an empty set if there is no read/flow dependence associated with the r. For this case, the loop ordering is in the form of $< NRL(r) >$, and is added into O for consideration.*

Proof Sketch. Assume the above statement is false, i.e., there is a loop ordering o' out of the set O that achieves better performance than loop orderings in O. Then, there exists at least one loop l_{rl} in o' that carries the read/flow dependences for a certain array reference r, and is placed above a certain loop l_{nrl} that belongs to $NRL(r)$. We group all such loops l_{rl} into a set $RL'(r)$ and permute them to the innermost of the array partitioning loop band to generate a new loop ordering $\hat{o}$ that belongs to O. If r is associated with flow dependences, o' introduces additional I/O modules for loading in the intermediate results, increasing the resource usage compared to $\hat{o}$. If r is associated with read/flow dependences, o' increases off-chip data communication, and could lead to a longer latency than $\hat{o}$ if the design is bound with data communication of r. Overall, this loop ordering o' is dominated by $\hat{o}$ in both resource usage and design latency, which contradicts the initial assumption that o' dominates loop orderings from O in performance. □

For the MM example, loop $i.0$ carries the read dependence for array B, loop $j.0$ carries the read dependence for array A, and loop $k.0$ carries the flow dependence for array C. In total, there are three loop-ordering candidates which lead to systolic arrays with potentially different performances. Note that as long as the innermost loop is fixed, the ordering of other loops will not impact the performance. In the rest of this paper, we use the annotation $< [i.0, j, 0], [k.0] >$ to identify the set of loop orderings. All the loops in the same brackets can be permuted freely with equivalent performance. We will choose one ordering randomly in practice. Table 2 shows all the unique loop orderings for MM and Convolution.

To summarize, given an input program, we use AutoSA to generate different dataflows and loop permutation orderings of the array partitioning loops and leave the tiling factors as tunable parameters to be handled by Odyssey design space explorers as detailed in Table 2.

4. Odyssey DSE Methodology

In this section, we introduce our automatic performance model generation and the two design space exploration approaches for loop tiling.

4.1. Automatic Performance Model Generation

The accuracy of performance models plays an important role in performance tuning. In Section 1, we discussed the issue of using a simplified performance model that overlooks the epilogue and prologue phases of the hardware execution. Table 1 highlights several previous works with a similar issue [17, 5, 14]. In addition, such performance models are usually derived manually which is time-consuming and error-prone. Odyssey distinguishes itself from the prior works in that it automatically creates performance models by leveraging the AutoSA compiler. We have extended AutoSA to generate a design descriptor that contains all the necessary information for estimating the design performance.

The auto-tuner utilizes this description file to create a Python file containing functions for estimating the design performance. All the performance models are symbolic expressions of the tuning parameters. During the search, the auto-tuner samples the design space and plugs in different tuning parameters into the performance models for assessing the design performance.

4.2. Genetic-MP Hybrid Method

4.2.1. Genetic algorithm. For Odyssey's first explorer, we select evolutionary search as the backbone search method. Evolutionary search [7] is a generic meta-heuristic algorithm inspired by biological evolution, in which individuals of a population gradually improve themselves through a series of biological mechanisms such as *mutation*, *crossover*, and *selection*. In the context of hardware design space exploration, we have:

Encoding scheme: Each individual is encoded by the tiling factors used in AutoSA compilation passes. The encoded genome includes the problem size and the tiling factors used for each compilation pass.

Mutation: We randomly select one loop l_2 and mutate this loop by changing the loop bound to a random value $s \in [1, l_2]$. Next, we select another corresponding loop l_1 and change its loop bound to a new value s' computed by $s' = ceil(l_1 \times l_2/s)$. Figure 2 displays one example of mutation where the new tiling factor $T_{j1} = 36$ is not a divisor of 64.

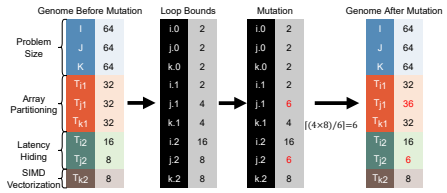

Figure 2: Example of a MM genome mutation

When performing the mutation, we assign a probability α to execute the mutation. Based on a grid search, we set α to 0.4 by default.

Crossover The crossover operation exchanges the genomes of two individuals. To guarantee the validness of the offspring, we exchange the tiling factors associated with the same original loop together.

4.2.2. Mathematical Programming. Although the mathematical programming (MP)-based method fails to identify the optimal design, the design it finds achieves relatively good performance and could be used as the initial population of the evolutionary search.

Odyssey implements a MP-based optimizer to produce high-quality seeds for the evolutionary search. We formulate the optimization problem as follows.

Objective Function: Given the high complexity of the hardware designs, it is usually difficult to have a close-formed performance model suitable for the solvers as the objective function. Instead, previous works chose different high-order functions that impact the performance as the objective functions [13, 3, 8]. We conducted an experiment on evaluating the effectiveness of several objective functions.

Objective 1: Computation resource We employ the total DSP usage U_{DSP} as the optimization target. The heuristic is that a design with higher performance utilizes more DSPs.

$$Obj1 : min(-U_{DSP}) \tag{3}$$

Objective 2: Off-chip communication We aggregate the off-chip data movement of all the arrays in the program. For communication-bound applications, reducing off-chip communication could improve a design's performance.

$$Obj2 : min \sum_{a \in Arrays} DM(a) \tag{4}$$

Objective 3: Off-chip communication - computation resource This objective function takes both computation and communication into consideration. Ideally, we would like to maximize the computation resource and reduce the off-chip communication.

$$Obj3 : min(\sum_{a \in Arrays} DM(a) - U_{DSP}) \tag{5}$$

Constraints: A valid hardware design should not overuse the available memory and computation resource. For FPGA designs, we consider the BRAM and DSP usage.

$$U_{mem} \leq Mem_{available},\ U_{DSP} \leq DSP_{available} \tag{6}$$

Where U_{DSP} and U_{mem} are the sum of the DSP and memory usage of each hardware module in the systolic array.

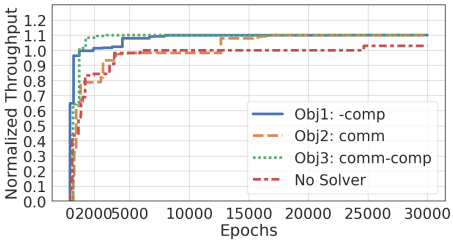

Figure 3: Search traces of the evolutionary search initiated with designs found by the MP-based optimizer with different objective functions

We engage the off-the-shelf solver (AMPL [6] with Ipopt [4]) to implement the optimization problem. All the metrics have been normalized. The best solution obtained from the solver is then fed to the evolutionary search as the initial population. Figure 3 shows the search traces of the evolutionary search with different optimization targets. As seen in the figure, all three objective functions help reduce the convergence time and yield better results compared to the evolutionary search-only method (annotated as *No Solver*). Specifically, Obj3 helps significantly reduce the convergence time. With Obj3, the auto-tuner locates a better design than the baseline (No Solver) within 2000 epochs. Thus, we utilize Obj3 as the optimization target of the solver.

4.3. Padding-based Algorithm

In Section 1, we established that limiting the tiling factors to the divisors of the problem size can lead to inferior designs. The reason is that sometimes divisor tiling factors do not allow the full utilization of the available resources such as DSPs, BRAMs, or bandwidth. Thus, considering non-divisor tiling factors can help maximize some or all of these resources.

Also, note that when tiling factors are non-divisors, the original problem size needs to be padded with zeros to yield integer values for the outer loops' trip counts. Therefore, considering non-divisor tiling factors in the search is equivalent to first finding a set of the possible zero-padded problem sizes, and then ***only*** searching for the divisor tiling factors of each zero-padded problem size.

Heuristic: *Since zero padding adds computation and communication overheads, we should explore the design space in the direction of increasing the zero-padding.*

Termination criterion: A naive termination criterion for the algorithm would be to search fixed N padded candidates on each dimension. However, this approach can easily miss optimal points if they are beyond the Nth padded candidate. A better termination criterion would be to employ adaptive counters with thresholds. The counters are reset to zero whenever a better design is found allowing the algorithm to explore more points in that direction; otherwise, if there is no improvement in $threshold$ consecutive candidates, it breaks. Empirically, we found that setting thresholds to be $\lceil 0.5\sqrt{dim} \rceil$ achieves the optimal solutions identified by the exhaustive search for most of the cases.

In fact, this behavior is independent of the architecture/accelerator because it is fundamentally related to the properties of the factorization of integers. In Figure 4, we plot the design space of a MM toy example using a 1-level tiling performance model (Equation 7) with 5000 DSPs as the only constraint (Equation 8). The plot shows the optimal divisor-only designs for each zero-padded candidate on the I and J dimensions (like $32 \times 33 \times 32$, $32 \times 34 \times 32$, etc.). Allowing non-divisor tiling factors (the green point at $33 \times 33 \times 32$) results in a $1.78\times$ speedup compared to considering divisors only (the blue point at $32 \times 32 \times 32$).

$$Cycles = \left\lceil \frac{I}{T_i} \right\rceil \left\lceil \frac{J}{T_j} \right\rceil \left\lceil \frac{K}{T_k} \right\rceil \tag{7}$$

$$5 \times T_i T_j T_k \leq DSP_{available}\ (5000\ DSPs) \tag{8}$$

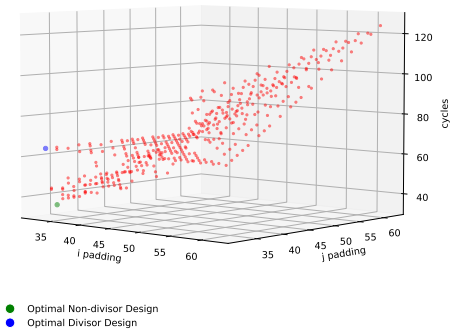

Figure 4: Design space for a $32 \times 32 \times 32$ MM toy example

5. Evaluation Results

First, we evaluate the performance and sample efficiency of our genetic-MP hybrid method against prior iterative and random search methods. Since our work targets the design space of systolic arrays while the previous works targeted different design spaces, it is difficult to conduct direct quantitative comparisons. Thus, we use the search methods adopted in these works and compare their quality with Odyssey. Next, we compare the performance of the padding-based algorithm. Finally, we validate our findings by implementing two systolic arrays for $1024 \times 1024 \times 1024$ MM.

5.1. Genetic-MP Hybrid Method

Workloads: We assess the performance of the genetic-MP hybrid method with a $1024 \times 1024 \times 1024$ MM kernel.

Baselines: We use the following methods as baselines.

1) *Random search*. We randomly sample the design space and update the best solutions.

2) *Exhaustive search with pruning*. We extend the random search by pruning the design samples based on the DSP utilization. As the smallest design among the 18 designs utilizes 30% of DSPs, we set the DSP pruning threshold to 25%.

3) *Simulated annealing*. We use the Python package [1] as the baseline. Based on a grid search, we designate the hyper-parameter temperature T to be 200. We implement a customized step-taking function employing the proposed mutation method for evolutionary search.

4) *Bayesian optimization*. We use the Python package [16] as the baseline.

5) *OpenTuner [2]*. OpenTuner is an auto-tuning framework built on an ensemble of several efficient search techniques. We use the latest release of OpenTuner from its GitHub repository [2].

6) *Reinforcement learning (RL)*. RL is a machine-learning algorithm that can be utilized for hardware optimization.

The previous work ConfuciuX [9] implemented a two-step search algorithm for tuning the dataflow architectures which employs RL as the first step to locate a good sub-design-space and utilizes evolutionary search to perform a more fine-grained search later to find the best design. We use the open-source implementation from ConfuciuX as the RL baseline [9]. ConfuciuX applied a 3-layer multi-layer perceptron (MLP) neural network for the policy network.

Designs: We compare our tuning methods against the baselines on all 18 different designs generated for MM by AutoSA (Table 2).

Evaluation setup: For each systolic array design, we run the search method for 5 minutes[3] and repeat it 3 times. The final results are averaged from the 3 runs after each method converged. All the search methods are executed with a single CPU thread. RL baseline uses Pytorch which implements multi-threading during the training of MLP. All experiments are executed on a workstation with an Intel Xeon E5-2680 v4 CPU.

Search results quality: Figure 5 compares the best throughput (1/latency) achieved by each tuning method on the 18 systolic array designs. The throughput is normalized against the optimal performance found by exhaustive search[4]. The genetic-MP method found design configurations with the best performance on 13 designs out of the total 18 designs. For the remaining 5 designs, the performance gap is within 1% of the best performance identified by other baselines (OpenTuner and simulated annealing). Overall, the genetic-MP method locates designs that achieve more than 95% of the optimal performance.

3. All search methods converged within 5 minutes.
4. We run an exhaustive search until it finishes.

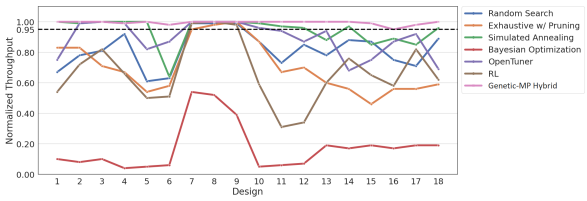

Figure 5: Comparison of the best designs found by different tuning methods considering all dataflows and loop permutations in Table 2

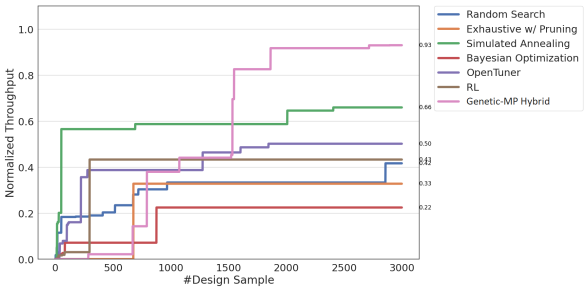

Figure 6: Comparison of sample efficiency of different tuning methods

Sample efficiency: In addition to the high-quality search results, the genetic-MP method achieves high sample efficiency. Figure 6 compares the convergence traces of all the tuning methods on the design with the highest optimal throughput. As shown by the figure, Odyssey detects a good design configuration resulting in 93% of the optimal performance after evaluating 3000 design samples. Simulated annealing earns the second-best performance, locating a design that reaches 66% of the optimal performance.

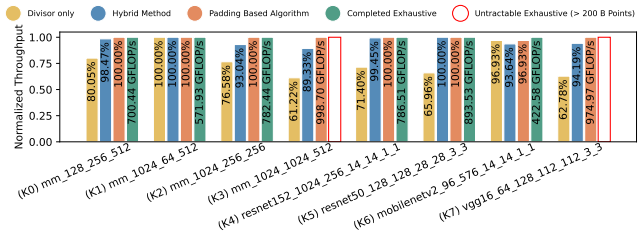

Figure 7: Normalized throughput comparisons for 4 MM kernels and 4 convolution layers

5.2. Padding-based Algorithm

We compare the padding-based algorithm against the divisor-only exhaustive search, the complete exhaustive search (including non-divisors), and the genetic-MP hybrid method in terms of performance and runtime. We used 4 MM kernels (K0 - K3) and 4 convolution layers from four famous CNNs (K4 - K7). We employed the complete exhaustive search as a baseline, except for K3 and K7 as these kernels have huge design spaces with more than 200 billion points. Figure 7 shows the best performance found by

each search method across all dataflows and loop permutations (18 unique designs for MM and 30 for convolution). Using thresholds of $\lceil 0.5\sqrt{dim} \rceil$, the padding-based algorithm identifies the optimal points for 15 out of the 16 kernels whose exhaustive search is completed (showing only 6 kernels in Figure 7 for page limit). For K6, increasing the threshold to $\lceil 0.6\sqrt{dim} \rceil$ locates the optimal point as well. Moreover, the designs found by the genetic-MP method are within less than 11% of the optimal designs.

Design	Exhaustive Search		Padding-Based Algorithm		Design Space Reduction	Runtime Speedup
	# of Searched Designs	Runtime (hours)	# of Searched Designs	Runtime (hours)		
K0	3.66 B	1.59	28.64 M	0.06	127.7×	25.3×
K1	7.75 B	3.24	41.27 M	0.09	187.7×	36.7×
K2	33.17 B	13.83	105.99 M	0.19	312.9×	73.1×
K4	34.80 B	14.71	198.17 M	0.19	175.6×	78.6×
K5	53.06 B	26.44	620.03 M	0.54	85.6×	49.0×
K6	84.68 B	34.56	356.91 M	0.28	237.3×	122.1×

TABLE 3: Comparing exhaustive search and padding-based algorithm in terms of the explored design space and runtime

Table 3 illustrates the design space reduction and runtime speedups of the padding-based algorithm compared to the complete exhaustive search. Both search methods are run using 72 threads on Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz.

5.3. On-Board Validation Results

To validate the results of this work, we implemented two systolic arrays for $1024 \times 1024 \times 1024$ MM on the Xilinx/AMD Alveo U250 FPGA. The first design is the best design identified by the exhaustive search considering divisor tiling factors only. The second design is found by Odyssey considering non-divisor tiling factors. The on-board results match our analysis in Section 1, and the non-divisor systolic array delivers a 1.72× throughput improvement.

Search Method	Padded Problem Size (I, J, K)	SA Size $(Cols, Rows, SIMD)$	DSPs	BRAMs	Frequency (MHz)	Throughput GFLOP/s	Speedup
Divisor only	(1024, 1024, 1024)	(32, 4, 8)	5133	2543	257	506.71	1×
Odyssey	(1032, 1040, 1024)	(43, 5, 8)	9258	2932	279	869.97	1.72×

TABLE 4: FPGA validation results

6. Conclusion

This paper presents Odyssey, an automatic design space exploration framework for systolic arrays. Odyssey covers a comprehensive and accurate design space of systolic arrays and incorporates two design space explorers: A hybrid search method consisting of the MP-based optimizer and evolutionary search, and a novel padding-based algorithm that locates the optimal designs discovered by the exhaustive search in most of the cases. The evaluation and validation results demonstrate the effectiveness of Odyssey in handling the huge design space of systolic arrays.

7. Acknowledgment

This work is partially supported by the NSF/Intel CAPA Program, the NSF NeuroNex Program, the DARPA/SRC JUMP program, and CDSC industrial partners.

References

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.basinhopping.html.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner. https://github.com/jansel/opentuner, 2021.

[3] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric compiler for DNN operators on spatial accelerators. *arXiv preprint:2002.07752*, 2020.

[4] COIN-OR. Interior point optimizer, 2021.

[5] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. Dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–27, 2019.

[6] David M Gay. The AMPL modeling language: An aid to formulating and solving optimization problems. In *Numerical analysis and optimization*, pages 95–116. Springer, 2015.

[7] John H Holland. Genetic algorithms. *Scientific American*, 267(1):66–73, 1992.

[8] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. CoSA: Scheduling by constrained optimization for spatial accelerators. *arXiv preprint:2105.01898*, 2021.

[9] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. ConfuciuX: Autonomous hardware resource assignment for DNN accelerators using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 622–636. IEEE, 2020.

[10] Sheng-Chun Kao and Tushar Krishna. Gamma: Automating the HW mapping of DNN models on accelerators via genetic algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[11] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings. *IEEE MICRO*, 40(3):20–29, 2020.

[12] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Jason Cong, Nithin George, Jose Alvarez, Christopher Hughes, and Pradeep Dubey. SuSy: A programming model for productive construction of high-performance systolic arrays on FPGAs. In *2020 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020.

[13] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P Sadayappan. Analytical characterization and design space exploration for optimization of CNNs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 928–942, 2021.

[14] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. TENET: A framework for modeling tensor dataflow based on relation-centric notation. *arXiv preprint:2105.01892*, 2021.

[15] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 804–817, 2021.

[16] Fernando Nogueira. Bayesian optimization: Open source constrained global optimization tool for python. https://github.com/fmfn/BayesianOptimization, 2014.

[17] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to DNN accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315. IEEE, 2019.

[18] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2021.

[19] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. Interstellar: Using Halide's scheduling language to analyze DNN accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–383, 2020.